sid.inpe.br/mtc-m21d/2021/09.08.19.55-TDI

ASSESSING CODE ANNOTATIONS USAGE IN SOFTWARE PROJECTS

Phyllipe de Souza Lima Francisco

Doctorate Thesis of the Graduate Course in Applied Computing, guided by Drs. Eduardo Martins Guerra, and Paulo Roberto Miranda Meirelles, approved in September 16, 2021.

INPE
São José dos Campos
2021

sid.inpe.br/mtc-m21d/2021/09.08.19.55-TDI

ASSESSING CODE ANNOTATIONS USAGE IN SOFTWARE PROJECTS

Phyllipe de Souza Lima Francisco

Doctorate Thesis of the Graduate Course in Applied Computing, guided by Drs. Eduardo Martins Guerra, and Paulo Roberto Miranda Meirelles, approved in September 16, 2021.

URL of the original document:
<http://urlib.net/8JMKD3MGP3W34T/45DA8DH>

INPE
São José dos Campos
2021

Cataloging in Publication Data

Francisco, Phyllipe de Souza Lima.

F847a Assessing code annotations usage in software projects / Phyllipe de Souza Lima Francisco. – São José dos Campos : INPE, 2021.

xxvi + 173 p. ; (sid.inpe.br/mtc-m21d/2021/09.08.19.55-TDI)

Thesis (Doctorate in Applied Computing) – Instituto Nacional de Pesquisas Espaciais, São José dos Campos, 2021.

Guiding : Drs. Eduardo Martins Guerra, and Paulo Roberto Miranda Meirelles.

1. Code annotations. 2. Software metric. 3. Software visualization. 4. Polymetric View. 5. Metadata. I.Title.

CDU 004.4’41

INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

DEFESA FINAL DE TESE DE PHYLLIPE DE SOUZA LIMA FRANCISCO
BANCA Nº 261/2021 REG 137863/2016

No dia 16 de setembo de 2021, as 08h30min, por teleconferência, o(a) aluno(a) mencionado(a) acima defendeu seu trabalho final (apresentação oral seguida de arguição) perante uma Banca Examinadora, cujos membros estão listados abaixo. O(A) aluno(a) foi APROVADO(A) pela Banca Examinadora, por unanimidade, em cumprimento ao requisito exigido para obtenção do Título de Doutor em Computação Aplicada. O trabalho precisa da incorporação das correções sugeridas pela Banca Examinadora e revisão final pelo(s) orientador(es).

Título: "Assessing Code Annotations Usage in Software Projects "

Dr. Stephan Stephany - Presidente - INPE
Dr. Eduardo Martins Guerra - Orientador – INPE
Dr. Paulo Roberto Miranda Meirelles - Orientador – UFABC
Dr. Gilberto Ribeiro de Queiroz - Membro Interno – INPE
Dr. Elcio Hideiti Shiguemori - Membro Interno – IEAv
Dr. Marco Tulio Valente - Membro Externo – UFMG
Dr. Cláudio Nogueira Sant'Anna - Membro Externo – UFBA

fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Marco tulio de oliveira valente (E)**, **Usuário Externo**, em 08/10/2021, às 14:36 (horário oficial de Brasília), com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Claudio nogueira sant'anna (E)**, **Usuário Externo**, em 23/11/2021, às 14:35 (horário oficial de Brasília), com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020.

A autenticidade deste documento pode ser conferida no site http://sei.mctic.gov.br/verifica.html, informando o código verificador **8125820** e o código CRC **B56F5429**.

Referência: Processo nº 01340.006221/2021-36 SEI nº 8125820

“Praise the Sun”.

Solaire of Astora, Warrior of Sunlight
in *“Dark Souls”*, 2011

A meus pais ***Helena*** *e* ***Cezar***

ACKNOWLEDGEMENTS

This section is far too personal to be written in English, therefore, I will do it in the beautiful Brazilian Portuguese.....

Agradeço primeiramente a minha mãe, Lena, e ao meu pai, Cezar. Eles são as forças reais e concretas que me permitem arriscar e correr atrás de todos os meus sonhos. Tudo que eu faço, eu só tenho forças pra fazer, pois sei que se eu cair eles irão me levantar. Sempre senti o amor incondicional, e assim a escrita dessa tese é em homenagem a eles, as melhores pessoas que já conheci!

Quando decidi ingressar no doutorado, não poderia imaginar a incrível aventura que seria. Eu sempre achei bela a ideia de ser um acadêmico, de fazer pesquisa, descobrir coisas, provar teorias, e discutir ideias com outras pessoas. O caminho natural para me formar como um pesquisador seria então ingressar em um programa de doutorado. Mas surgiram as perguntas, para onde ir? Quem procurar? E foi assim que cheguei no INPE em São José dos Campos, uma das melhores decisões que tomei na minha vida.

Enquanto eu concluía o mestrado na UNIFEI, estava buscando professores que pudessem orientar meu doutorado e passei pelo Prof. Eduardo Guerra, que veio a se tornar meu orientador, e um grande amigo também. Fiz contato inicial, e ele me autorizou a cursar a sua disciplina Desenvolvimento de Frameworks, como isolada. Assim que cheguei ao INPE para fazer a matrícula pensei “Que lugar bonito! Devem ter muitas pessoas inteligentes aqui”. Quando fui assistir a primeira aula do Prof. Guerra, pude perceber o quanto eu não sabia nada e ainda tinha muito para aprender. Fiquei encantando com a aula, o tema, e tudo que poderia aprender ali com ele. Tive certeza que eu tinha escolhido a pessoa certa, na instituição certa, para me guiar. Passado algumas semanas, ainda cursando a disciplina isolada, fiz o processo seletivo, e consegui ingressar no programa. Agradeço imensamente ao Prof. Guerra, que meu deu a oportunidade de cursar a disciplina isolada, e por ter aceitado como aluno de doutorado. Com isso tive o privilégio de ser seu aluno, e ter tido uma formação de elite. É difícil medir o quanto eu aprendi com o Prof. Guerra, porque parece que existe um abismo entre quem eu era e quem me tornei durante a minha formação sob sua orientação.

Hoje posso dizer que o considero um grande amigo e espero que a gente ainda continue trabalhando, e jogando, juntos por muito tempo. Jogando? Sim! Além da minha formação como doutor o Prof. Guerra ainda me apresentou um outro

mundo que eu tinha explorado muito pouco, quase nada. O mundo dos jogos de tabuleiro, ou *boardgames*. O que começou para mim como uma "jogatina de vez em quando", se tornou um compromisso e fiquei viciado nesse mundo. A sorte era que o Prof. Guerra tinha todos os jogos e me permitia ter acesso aos novos lançamentos. Eu jamais esquecerei as "jogatinas de quarta", e confesso que várias vezes, eu me deslocava para São José dos Campos apenas para jogar! Os *boardgames* fizeram grande parte do meu doutorado, e me permitiram momentos de lazer para diminuir a tensão. Assim, agradeço ao Prof. Guerra, por me proporcionar esses momentos de tranquilidade, alegria e diversão. Agradeço principalmente por ter me viciado nesse mundo incrível, e hoje eu gasto mais do que eu deveria com *boardgame* (obrigado Guerra)

Assim que eu comecei a cursar a disciplina isolada do Prof. Guerra, antes mesmo de ingressar no programa, ele já me convidou para uma pesquisa, onde eu tive contato com o Prof. Paulo Meirelles, que veio a ser tornar meu coorientador, amigo e aliado. Na época morando em Brasília, como professor da UnB, fazia questão de se deslocar para São José dos Campos para nos reunirmos presencialmente e me dar ainda mais suporte. Posso dizer que tive a sorte de ter tido um coorientador, que na verdade eu chamo de "orientador". Esteve comigo durante toda a minha formação e com quem aprendi diversos valores. Aprendi muito com o Prof. Paulo a ideia de "compartilha a tela que a gente já faz agora". Nada de "depois eu faço". "Automatiza tudo". Peguei essas ideias, e as levo comigo no meu dia a dia.

Além de todo a orientação ao longo do doutorado, o Prof Paulo foi o responsável pelo meu primeiro artigo publicado. Ele teve a ideia de enviarmos um trabalho para o VEM que ocorreu no CBSoft 2017 (Congresso Brasileiro de Software). Fizemos a submissão e o trabalho foi aceito. Assim, fui ao meu primeiro congresso, minha primeira ida ao nordeste e pude ter contato com uma comunidade de brilhantes de pesquisadores e pesquisadoras que atuam engenharia de software. Sou imensamente grato ao Prof. Paulo por ter me dado a chance de participar do meu primeiro congresso, e ter contato com esse ambiente incrível. E finalmente, nos últimos dois anos, o Prof Paulo teve papel essencial na escrita da tese, acompanhando de perto cada nova página que eu adicionava! Tenho muito a agradecer ao Prof. Paulo, e espero que possamos continuar publicando e trabalho juntos, por muito mais tempo!

Mesmo que agora deixe de existir a relação orientador-aluno, na prática eu os verei, tanto o Prof Guerra, como o Prof Paulo, como guias eternos e sempre os procurarei para orientações e ideias!

Agradeço também aos colegas alunos da CAP que fiz enquanto cursava as disciplina no INPE, aos happy hours, risadas e diversões. Deixo um agradecimento especial ao Helder Casagrande, também conhecido como *Lord of Rélder*. Grande companheiro de jogatina, de zueira, de bagunça, de café no LAC, berros de café no LAC, pausa para café no LAC e mais jogatina, mais bagunça e mais café!

Agradeço a Jéssica Cristina, da secretaria da CAP, que estava disposta a ajudar e resolver todos os problemas burocráticos que surgiam no nosso dia a dia!

Agradeço a todos os professores da CAP, especialmente aos Prof. Gribeiro, Prof. Solon, Prof. Ezzat, Prof. Vijay, Prof Alan Calheiros e Prof. Rafael. Com os quais aprendi muito!

Agradeço também a um dos melhores momentos que todo aluno da CAP pode experimentar. As muitas gargalhadas nos corredores do LAC enquanto toma café na copa. Não lembro qual era a melhor parte: fazer o café? gritar que o café tava pronto? tomar o café? ou a prosa? Impossível escolher!

Agradeço a equipe e ex-membros do EMBRACE que gentilmente cederam parte do seu tempo para conversar comigo e ajudar a validar parte desse trabalho.

Não poderia deixar também de agradecer as agências de fomento FAPESP e CAPES que forneceram apoio financeiro durante toda a minha formação.

Se eu voltasse no tempo, faria tudo exatamente da mesma forma com as mesmas pessoas! Muito obrigado a todos!

ABSTRACT

Code annotation is a language feature that enables the introduction of custom metadata on programming elements. In Java, this feature was introduced on version 5, and it is widely used by the leading enterprise application frameworks and APIs. Although very popular to simplify metadata configuration, software engineering lack research and experiments about them. Also, its abuse and misuse can reduce source code readability, comprehension and complicate its maintenance. Our work proposes an approach to assess code annotations usage and distribution in a software project to overcome this. We begin defining a novel suite of software metrics dedicated to code annotations. We analyzed their distribution in open-source projects by extracting their values from 24,947 java classes and obtaining threshold values. We also provided a way to interpret these threshold values using a percentile rank analysis, revealing outliers. Afterward, we proposed a novel polymetric view tailored specifically to visualize code annotations distribution and usage using our metrics as input. We named it CADV - Code Annotations Distribution Visualization. To validate the CADV, we conducted two experiments. The first was an interview with six professional developers from EMBRACE, and the second was conducted asynchronously with 44 students through a form. As a target software, we used the *SpaceWeather* system. Our results show that the proposed visualization approach can aid developers and students in comprehending the distribution of code annotations, packages responsibilities and potentially detect misplaced ones. Furthermore, CADV provides a much quicker approach to identify code annotations and schemas when compared with manual code inspection.

Keywords: Code Annotations. Software Metric. Software Visualization. Polymetric View. Metadata.

AVALIANDO O USO DE ANOTAÇÕES DE CÓDIGO EM PROJETOS DE SOFTWARE

RESUMO

Anotações de código são uma característica de linguagem que permitem a configuração de metadados em elementos de programação. Na linguagem Java, essa característica foi introduzida na versão 5 e é utilizada por desenvolvedores de *frameworks* e APIs corporativas amplamente utilizadas. Mesmo que as anotações sejam muito populares para simplificar a configuração de metadados, a comunidade de engenharia de software possui poucos trabalhos que as investigam. Adicionalmente, o seu uso inadequado pode reduzir a legibilidade, compreensão e comprometer a manutenção do sistema. Com isso, esse trabalho apresenta uma abordagem para avaliar o uso e distribuição de anotações em projetos Java. O primeiro passo foi definir um novo conjunto de métricas de código fonte capaz de extrair características das anotações usadas. Para validar as métricas, fizemos uma coleta em 25 projetos de código aberto e foi possível observar como elas se comportam. Em seguida, com as métricas disponíveis, fizemos uma proposta de visualização polimétrica projetada para visualizar anotações de código e como estão distribuídas. Para validar a visualização conduzimos dois experimentos. O primeiro foi feito com seis desenvolvedores que participaram da criação do sistema *SpaceWeather* do EMBRACE, e o segundo foi conduzido com 44 alunos de graduação. Os resultados mostraram que a visualização permite compreender rapidamente a distribuição das anotações e identificar as responsabilidades dos pacotes.

Palavras-chave: Anotações. Metadados. Métricas de Software. Visualização de Software. Visão Polimétrica.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

AA	–	Arguments in Annotation
AED	–	Annotations in Element Declaration
AC	–	Annotations in Class
ANL	–	Annotation Nesting Level
ASC	–	Annotations Schemas in Class
API	–	Application Programming Interface
CADV	–	Code Annotations Distribution Visualization
JPA	–	Java Persistence API
JVM	–	Java Virtual Machine
LOCAD	–	Lines of Code in Annotation Declaration
UAC	–	Unique Annotations in Class

CONTENTS

Page

1 INTRODUCTION

Code annotations, or simply annotations, are a feature available on the Java programming language. They were introduced in version 5 to configure custom metadata directly on programming elements, such as methods and classes. Tools or frameworks usually consume these to gather additional information about the software, allowing the execution of more specific behavior. Since annotations are inserted directly on the source code, they are a convenient and quick alternative to configure metadata (GUERRA, 2014).

Main enterprise Java APIs make extensive use of code annotations, making them a relevant feature used daily by developers. Examples of APIs are the EJB[1] (Enterprise Java Beans) used to configure transactions and security restrictions, the JPA[2] (Java Persistence API) used to perform object-relational mapping and the JUnit[3] used for unit testing.

Although popular and increasingly used by developers, the software engineering community lacks work and research dedicated to studying and analyzing code annotations and their impact on software development. For instance, current software metrics only acknowledge annotations as being another line of code. They do not consider them when calculating complexity or cohesion, which can lead to an incomplete code assessment (GUERRA et al., 2009). A domain class can be considered simple using current complexity metrics. However, it can contain complex annotations for object-relational mapping. For example, when a mapping is being overridden for an entity relationship, several annotations may be used, and they can be nested.

Another example is that using a set of annotations couples the application to a framework that can interpret them. Current coupling metrics do not explicitly handle this situation, and therefore does not provide a precise coupling value considering also code annotations.

When annotations are misused, it can harm the evolution and maintenance of software. For instance, an excessive amount of annotations can reduce code readability, and annotations duplicated through the project might be hard to refactor given the repetitive work (LIMA et al., 2018).

Software evolution and maintenance is recognized as the most costly and challenging

[1] `www.oracle.com/technetwork/java/index-jsp-140203.html`
[2] `www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html`
[3] `https://junit.org/junit5/`

activity in the software development life cycle (RAJLICH, 2014). Given the increasing presence of code annotations, they also play their role in the cost and challenges of software evolution. However, code that is easier to comprehend usually is easier to evolve. Therefore, developers spend most of their time comprehending the software they are working on to be able to contribute and add/maintain features (HASSELBRING et al., 2020).

To support developers to comprehend software better, the field of software visualization has become increasingly used (FRANCESE et al., 2016; MERINO et al., 2018). Some approaches aim to represent software as a known environment, such as a city(WETTEL et al., 2011; ROMANO et al., 2019) or a forest (ERRA; SCANNIELLO, 2012). Another approach is to create what is known as a “polymetric view”, defined as a lightweight visualization enriched with software metrics (LANZA; DUCASSE, 2003).

We combined software metrics and visualization to create an approach to assess and comprehend code annotations within this context.

1.1 Goal

The goal of this work is:

To define an approach to measure and visualize code annotations ***to assess and comprehend*** *their usage and distribution in software systems*

Given the increased presence of Java annotations and the lack of work dedicated to studying them, in this work, we want to explore how to comprehend better the usage, distribution, and presence of code annotations. It is not a primary goal of our work to detect problems related to code annotations, such as misconfigurations or misplacements. However, once the developer is familiar with the system it is working on -our primary goal–it might also be easier to detect problems.

To reach our goal, in short, we first measure code annotations, and then we visualize them using the metrics values as input.

1.2 Motivation and relevance

Code annotations are an extensively used feature of the Java language. Observing the 30-top rated Java projects on GitHub, the one with the least amount of annotations has 22% of annotated classes. At the same time, the most annotated project has 97%

of annotated classes. On average, 76% of classes are annotated(LIMA et al., 2018). Also, a study performed by Rocha and Valente (2011) verified that from 106 projects of the Qualitas Corpus project database (TEMPERO et al., 2010), 65 projects used annotations, showing that it is a widely adopted feature in Java source code. With extensive usage and lack of research in the software engineering community, code annotations create a fertile ground for exploration. Furthermore, it is still unclear how code annotations impact software development and how developers perceive this feature. Hence we want to create means to aid in comprehending them through measurement and visualization techniques.

Code annotations are not only present in Java. The C# language also features a similar feature known as *attributes*[4], available since the release of the language (ECMA, 2017). Analyzing the top 30 ranked C# projects on GitHub, on average, 56% of the classes are using at least one attribute (BRAGA et al., 2019). Therefore, proposing approaches to comprehend code annotations better has benefits not only for Java programmers.

The National Institute of Space Research (INPE) has several research programs that use web applications to cope with a vast amount of data, process it, and make it publicly available. Some of these applications are developed using Java language, and code annotations are used. As such, they can benefit from the findings and tools presented by our work. For instance, we have the project LEONA[5] and the EMBRACE[6] program.

The EMBRACE[7] program, which stands for "Brazilian Studies and Monitoring of Space Weather" is composed of several products that follow the same reference architecture (SANT'ANNA et al., 2014) based on Java Enterprise APIs. The web application used to process and make data publicly available is divided into six modules spanning 94 components archived in independent deployment units. Furthermore, this software system has 1314 classes, with 837 containing at least one annotation, i.e., 64%.

As will be further detailed in future Chapters and Sections, the software systems developed for the EMBRACE program were used in our experiments.

[4]Given the similarity of these features, in this work, we will refer to them as code annotations as well

[5]http://www.dae.inpe.br/acatmos/

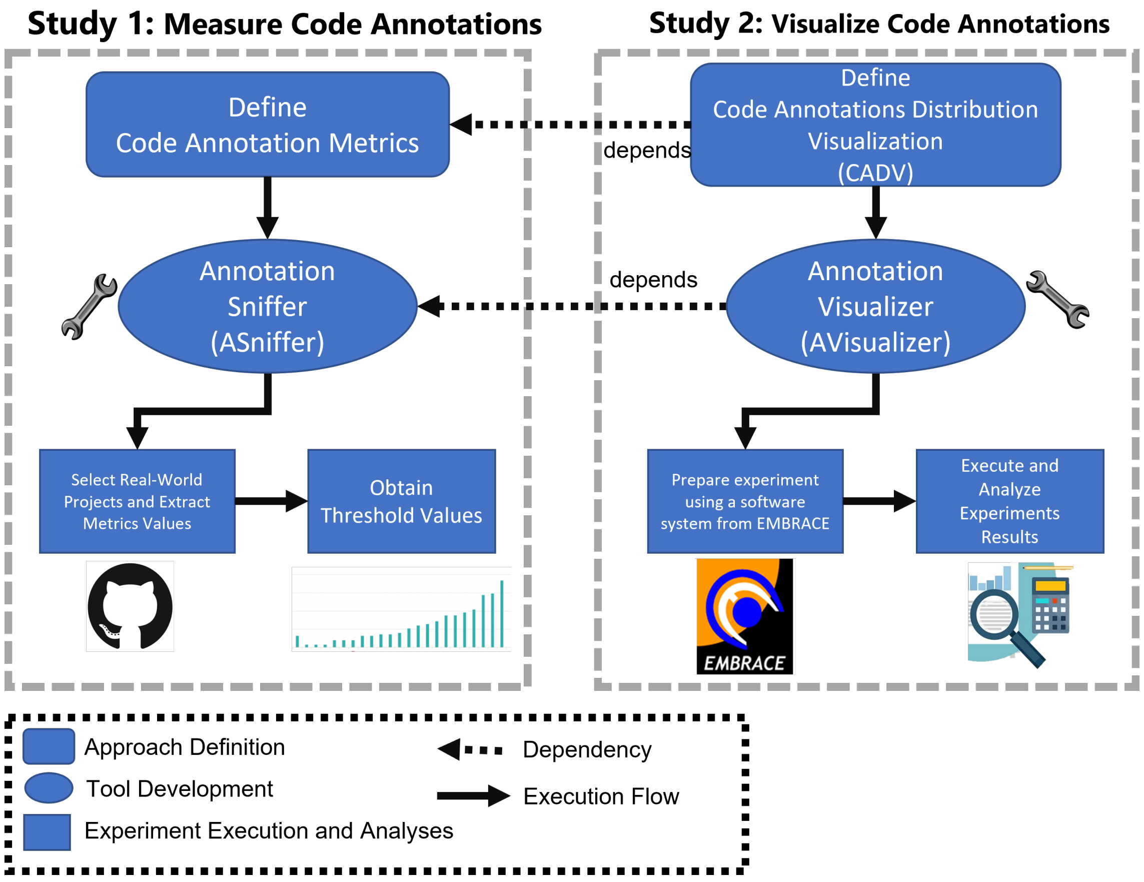
[6]http://www2.inpe.br/climaespacial/portal/pt/

[7]http://www2.inpe.br/climaespacial/portal/pt/

1.3 Methodology

This work is divided into two studies. The first defines and validates a novel suite of software metrics for code annotations. The second study uses the metrics values to create a visualization for code annotations in the system. The evaluation is carried out with a questionnaire-based experiment. For each study, we have a different methodology, presented as follows. Figure 1.1 presents a diagram of the two studies carried in this work. From this Figure, we see that parts we developed in the first study (Study 1) are carried as input to Study 2, better described below.

Figure 1.1 - Diagram of the Conducted Studies.



Study 1: Measure Code Annotations
Study 2: Visualize Code Annotations
Define Code Annotation Metrics
Define Code Annotations Distribution Visualization (CADV)
depends
Annotation Sniffer (ASniffer)
Annotation Visualizer (AVisualizer)
depends
Select Real-World Projects and Extract Metrics Values
Obtain Threshold Values
Prepare experiment using a software system from EMBRACE
Execute and Analyze Experiments Results
EMBRACE
Approach Definition
Tool Development
Experiment Execution and Analyses
Dependency
Execution Flow

Study 1 - Code Annotation Metrics

To propose the suite of metrics, we used a GQM (BASILI, 1992; BASILI et al., 1994) (Goal Question Model) where we identified some code annotations characteristics that we could measure. We were able to define 7 seven new metrics (LIMA et al., 2018).

Afterward, we selected 25 open-source Java projects following the guidelines of (NAGAPPAN et al., 2013). We extracted the seven metrics values and analyzed how they behaved among these projects by performing a statistical analysis using the Percentile Rank Analysis (MEIRELLES, 2013). To automate the process of collecting the metrics values, we developed an open-source tool called Annotation Sniffer (ASniffer) (LIMA et al., 2020a). With the statistical analysis complete, we were to identify threshold values as well as outliers.

In short, these were the steps carried out in the first study:

- Define a suite of annotation metrics
- Select real-world open-source projects
- Develop the ASniffer tool to perform the extraction
- Perform statistical analysis on the metrics values
- Define threshold values
- Identify common profiles and outliers

Study 2 - Code Annotation Visualization

We proceed to the second study of this work with the metrics defined and validated: to visualize the code annotations using the metrics values as input.

According to (MERINO et al., 2018), when defining software visualization approaches, the first step is to make the goals clear. Then, we design our approach consistent with them. Our primary goal is to visualize how code annotations are distributed in software systems, so we named it ***Code Annotations Distribution Visualization*** (CADV). It is a 2D polymetric view, using a hierarchical circle packing to represent the system. The innermost circles represent code annotations from source code, and their size is a function of a chosen metric value.

The CADV comprises three different views: The System View, The Package View, and The Class View. Each has its own goals, and they complement each other, providing different levels of visualization while also allowing the user to switch between them.

To implement CADV we created an open-source tool name Annotation Visualizer[8] (AVisualzer). The tool consumes the report generated by the ASniffer and draws our visualization approach. A demonstration is available as web application[9]

To evaluate our approach, we performed two experiments. The first, E1, was conducted as an interview with six developers of the ***SpaceWeather*** web application. The second experiment, E2, was conducted asynchronously with 44 students through a form. The majority were undergraduate students in computer areas. One of the goals of the experiment was to validate if the proposed visualization could easily present how code annotations are distributed in the system.

As a target software used for the experiments, we selected a module from the ***SpaceWeather***, which is a web application developed for the EMBRACE division of INPE. It made sense to choose this system and validate our approach as applied research to INPE internal demands.

The *SpaceWeather* web application is composed of the following modules:

- SpaceWeatherGNSSReprocessor
- SpaceWeatherION
- SpaceWeatherSunGif
- SpaceWeatherSWD
- SpaceWeatherTEC
- SpaceWeatherTSI

From these modules, we selected the *SpaceWeatherTSI* to be our target software, given that it uses many annotations from metadata-based frameworks. Which matches the characteristics we would like to display with our CADV approach.

In short, these were the steps carried out in the second study:

- Identify the goals of the visualization
- Define the visualization approach according to the goals

[8]`https://github.com/phillima/avisualizer`
[9]`https://avisualizer.herokuapp.com/`

- Develop the AVisualizer tool to implement the approach
- Select a target software and participants
- Train the participants how to use the tool
- Execute both experiments, E1 and E2.
- Analyze the results

From Figure 1.1, we see that the code annotations metrics and the ASniffer tool are also used in Study 2. The CADV is a polymetric view that uses the metrics defined in Study 1. Afterward, the AVisualizer tool requires metrics values extracted by the ASniffer to render our visualization. Therefore, it makes sense that our work begins with the proposal and definition of code annotation metrics. They will be used throughout the remaining parts of this work and other parallel research that our group is conducting.

1.4 Claimed contributions

The literature contains several works using code annotations to solve problems and implement solutions for diverse domains. Some of these papers apply annotations to support the implementation of design patterns (MEFFERT, 2006) or to enable architectural refactoring (KRAHN; RUMPE, 2006). However, only a few works evaluated the use of code annotations itself, focusing on design practices for metadata modeling or even performing studies to assess how annotations are currently being used by developers (LIMA et al., 2018).

Our first contribution, (C1), is a **novel suite of software metrics dedicated to code annotations**. They were designed to measure the size, complexity, and coupling of code annotations. For our second contribution (C2), we provided **threshold values** by extracting the metrics from open-source software and performing a statistical analysis. The third contribution (C3) is the **open-source tool developed to automate the metrics, called ASniffer**. Our fourth contribution (C4) is our **Code Annotations Distribution Visualization CADV** a novel polymetric view explicitly designed for code annotations. It is based on a nested circle packing approach. Finally, our fifth contribution (C5) is the **AVisualizer, an open-source tool that implements the CADV**. It was developed as a web application that uses the ASniffer as a dependency. We made it available as a single deployed unit to collect the metrics and draw the CADV.

The contributions of this work can all be applied to software systems developed at INPE. For systems using the Java programming language, such as the EMBRACE projects, developers can use our metrics and visualization approach to monitoring the growth and comprehend their system. As will be discussed in Chapter 5, one participant involved in the development of the **SpaceWeather** software system mentioned that *"it was possible to remember the architecture and even comprehend some interesting structures used during development that was hidden. It helps to monitor our code."*. Furthermore, INPE is also developing mobile applications using the Kotlin programming language. This language is compatible with the JVM[10] (Java Virtual Machine) and also has a feature similar to code annotations. Therefore, researchers can extend our work to measure and visualize code annotations in a Kotlin system.

In short, our contributions are:

- An approach to measure code annotations characteristics
- An approach to visualize code annotations characteristics

1.5 Published papers in the area of code annotations

Throughout the development of this work, we published the following papers:

- A Metrics Suite for Code Annotations Assessment (LIMA et al., 2018): We published this paper in the Journal of Systems and Software (JSS). We present and discuss our suite of metrics dedicated to code annotations. The contents presented in Chapter 3 was extracted from this paper.

- Annotation Sniffer: Open Source Tool to Extract Code Annotations Metrics (LIMA et al., 2020a). We published this paper in the Journal of Open Source Software (JOSS). We present the Annotation Sniffer (ASniffer), a tool we developed to support the extraction of code annotation metrics. The contents presented in Chapter 3 was extracted from this paper.

- Towards Visualizing Code Annotations Distribution (LIMA et al., 2020b). We published this paper in the Computer on the Beach (COTB). We present the first step we took in the direction of visualizing code annotations. From this paper, we were to obtain some feedback and improve our visualization. The contents presented in Chapter 5 is our new and unpublished visualization approach for code annotations.

[10]The virtual machine used to run Java applications

- A Metadata Handling API for Framework Development: a Comparative Study (GUERRA et al., 2020). We published this paper in the Brazilian Symposium on Software Engineering (SBES). We conducted an experiment to compare the Java Reflection API and a novel API, Esfinge Metadata, to consume and process code annotations. The ASniffer was used to extract code annotation metrics values to aid in the analysis and comparative study.

- Definição de clusters para classificação do uso de anotações em código Java (LIMA et al., 2017) (written in brazilian portuguese). We published this paper in the Workshop on Software Visualization, Maintenance, and Evolution (VEM). We presented an approach to classify Java classes based on code annotations usage. To support the extraction of metrics, the ASniffer was used.

- An Annotation-Based API for Supporting Runtime Code Annotation Reading (LIMA et al., 2017): We published this paper in the Workshop on Meta-Programming Techniques and Reflection (META). We presented an API to read, and process code annotations named the Esfinge Metadata API. As a demonstration, we carried a refactoring process and used the ASniffer tool to extract code annotations metrics used in the analysis.

- Does it make sense to have application-specific code conventions as a complementary approach to code annotations? (TEIXEIRA et al., 2018): We published this paper in the Workshop on Meta-Programming Techniques and Reflection (META). We investigated the presence of code conventions related to code annotations. To help extract code annotations information from the source code, we used the ASniffer.

- Attribute Sniffer: Collecting Attribute Metrics for C# Code (BRAGA et al., 2019): We published this paper in Tools Track of the Brazilian Conference on Software: Practice and Theory. In this paper, we investigated if the code annotations metrics designed for Java projects could be applied to C# projects, given the similarities of these features. Out of the seven metrics, we were able o directly apply five metrics to C# projects.

1.6 Organization

The background and related work is presented in Chapter 2. We discuss three topics in the chapter. It begins by presenting the concept of metadata in the context of object-oriented programming, defines code annotations, and shows the work other researchers have performed to assess code annotations. Then, we proceed to the second topic, which is software metrics. We define them and discuss why it is essential to analyze these metrics' distribution to obtain threshold values. Finally, we present what software visualization is and how other researchers presented and validated their visualizations for software systems.

In Chapter 3, we present the ASniffer, an open-source tool developed to extract code annotations metrics values. We divided this chapter into three parts. We begin presenting the tool's internal architecture and the source code responsible for extracting one example metric. In the second part, we describe how the tool can be used as a stand-alone application. Finally, in the third part, we briefly describe other works carried by our research team that used the ASniffer as a support tool. This final section is important because it reinforces that the ASniffer has become an important tool, initially developed for this work, but evolved and contributed to other research.

In Chapter 4, we present our novel suite of metrics. The chapter is divided into three parts. First is the research design and steps taken to propose and validate our metrics. Afterward, we present the definition of our novel suite of software metrics. Finally, we carry out a data collection and statistical analysis to obtain threshold values in the third part. This last part is important because just providing a metric value is not very useful for developers. This chapter comprises our first study, and the results were published on (LIMA et al., 2018).

In Chapter 5, we present the second part of our study, which is the definition and validation of our visualization approach for code annotations. We named this approach Code Annotations Distribution Visualization (CADV). The input data that builds the visualization are the values extracted using our novel metrics suite. This chapter follows a similar structure used in Chapter 4, dividing it into three parts. It begins with the research design for CADV. Then, in the second part, we proceed to define the views that, together, form CADV. Finally, we present the results and discussion of the conducted experiment to evaluate our visualization approach.

In Chapter 6, we conclude our work, analyzing the results, discuss the threats to validity, and where we want to take this research in future works.

2 BACKGROUND AND RELATED WORK

In this chapter, we present the theoretical foundation for our work. We also present similar work done by other researchers and discuss how these solutions differ from ours for each topic. The first topic discusses metadata configuration in object-oriented programming and how code annotations are used for this purpose. To further clarify, we present a simple example to enlighten how this feature works. The second topic regards software metrics and the importance of analyzing the distribution of the values using an adequate approach. To complete our background chapter, we present and discuss software visualization and how software metrics values can be used as input for visualization approaches.

2.1 Metadata in object oriented programming

In this section, we present what metadata is in the context of computer science, and more specifically, in the context of object-oriented programming. Afterward, we define code annotations, which is a feature of some programming languages to configure metadata.

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 table's structure. 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. When a developer uses reflection, it is manipulating the metadata of a program and using it to work with previously unknown classes (GUERRA et al., 2010; GUERRA, 2014).

Some tools or frameworks can consume metadata and execute routines based on class structure. For instance, it can be used for source code generation (DAMYANOV; HOLMES, 2004), compile-time verification (ERNST, 2008; QUINONEZ et al., 2008), class transformation (LOMBOK,), and framework adaptation (GUERRA et al., 2010). Often, the metadata contained in the class structure might not be enough to allow a specific behavior or routine to be executed. Therefore additional metadata can be configured on the programming elements.

2.1.1 Metadata configuration

One approach to defining custom metadata is to use external storage, such as an XML file or a database (FERNANDES et al., 2010). The drawback of this approach is the distance between the metadata and the code element since the external file needs some way to reference it. This adds some verbosity since a complete path must be provided so that the framework may correctly consume the metadata.

Another alternative, which is used by some frameworks, like Ruby on Rails (RUBY et al., 2009) and the CakePHP framework[1], is to define additional information using code conventions (CHEN, 2006). These are specific guidelines that developers use when creating code, such as naming pattern, return type, implementation of an interface, etc. Some frameworks are capable of identifying these conventions and execute specific behavior. A team of developers may also have their own code conventions that best suit their needs. Following these recommendations improve software maintenance, readability, and evolution. For instance, in the Java language, the Java Beans standard defined *getters* and *setters* naming convention for methods that read and write class attributes, respectively.

Although this choice can be very productive in some contexts, code conventions have limited expressiveness and cannot define more complex metadata. For instance, a code convention could be used to define a method as a test method as in JUnit 3. However, it could not be used to define a valid range of a numeric property as done by the Bean Validation API. Another drawback is that the metadata is implicit in the source code, hidden behind the conventions. As such, an unwary developer might alter a method's name without knowing that this actually is part of a convention, and some frameworks rely on this to execute a specific behavior properly.

2.1.2 Code annotations

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 code annotations (JSR, 2004), and in C#, by attributes (ECMA, 2017). A benefit is that the metadata definition is closer to the programming element, and its definition is less verbose than external approaches, such as using an XML file. 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

[1] `cakephp.org`

software elements (WADA; SUZUKI, 2005; SCHWARZ, 2004).

Code annotations (often referred to as "annotations") are a feature of the Java language, which became official on version 1.5 (JSR, 2004) spreading, even more, the use of this technique in the development community. Some base APIs, starting in Java EE 5, like EJB (Enterprise Java Beans) 3.0, JPA (Java Persistence API) (JSR, 2007), and CDI (Context and Dependency Injection), 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 (CÓRDOBA-SÁNCHEZ; LARA, 2016).

The Java language provides the Java Reflection API to allow developers to retrieve code annotations at runtime. It is also used to retrieve other information about the class structure, such as its fields, methods, and constructors. Additionally, developers may invoke methods, instantiate classes, and manipulate field values.

Consider the code on Figure 2.1. 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 configure which elements should be mapped to a column, table, primary key, and other mappings. Code annotations are suitable for this scenario.

Figure 2.1 - Example class without annotations.

```java
public class Player {

    private int id;
    private float health;
    private String name;

    private Date birthDate;

    //getters and setters omitted

}
```

Using code annotations provided by the JPA API, this mapping is easily achieved. Figure 2.2 shows the same `Player` class, but now using code annotations on appropriate elements. 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.

Figure 2.2 - Example class with annotations.

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

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

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

    @Column(name = "birthdate")
    private Date birthDate;

    //getters and setters omitted

}
```

The example in Figure 2.2 also shows that annotations may have parameters/arguments. For instance, `@Column` has a parameter, name, that receives a String. Another important aspect to observe is that these annotations - `@Column @Table`, and so forth - were created by framework developers, which implemented their associate behavior. The code annotations are exposed as part of the JPA public API, so that application developers can insert them on the source code as needed.

The code in Figure 2.2 presents an application that is needed to perform object-relational mapping, and it used a framework that offers this functionality through annotations. These are also known as metadata-based frameworks, i.e., an application developers' primary interaction with it is through metadata configuration instead of a method invocation or a class extension.

When discussing annotation-based APIs or metadata-based frameworks such as the JPA, JUnit, Spring, and others, an important definition is an “annotation schema” or simply “schema”. When the developers of metadata-based frameworks or APIs create new code annotations, their definition is placed in packages, just like any other type (class, enum, interface). They usually group code annotations in the same packages to better organize their responsibilities. These packages form what we define as “annotation schema” or “schema”. A single metadata-based framework may contain many schemas, and this decision is up to the framework developers. On the other hand, developers of applications that use such APIs or frameworks will import the packages containing the definition of the code annotations.

To better illustrate the definition of “schema”, consider the code on Figure 2.3. It is a simple class responsible for executing unit tests. In Java, we usually use the well-known JUnit framework. JUnit developers defined code annotations so that other application developers can test their code simply by configuring a method as a “test method” and other relevant features for testing.

Figure 2.3 - Example with *org.junit* schema.

```
 1  import org.junit.After;
 2  import org.junit.Before;
 3  import org.junit.Test;
 4
 5  public class TestClass {
 6
 7      @Before
 8      public void setUp(){
 9          //initializations
10      }
11
12      @Test
13      public void testMethod(){
14          //Execute tests
15      }
16
17      @After
18      public void cleanTest(){
19          //clear resources allocated during initialiazation
20      }
21
22  }
```

Observe on line 12, that the method `testMethod` is being configured with the code annotation `@Test`. However, to use this code annotation, we must import the package where it was defined. We see on line 3 the import of `org.junit.Test`, we also see other imports for the two code annotations as `org.junit.After` (line 1) and `org.junit.Before` (line 3). In other words, the developers of the JUnit defined the code annotations `@Test`, `@After`, `@Before` and placed them in the package `org.junit` along with other code annotations. The package `org.junit` is what we define as a "schema". Developers wishing to use these code annotations must import this "schema".

By this example, we see there is a strong relationship between a "schema" and code responsibilities. Detecting their presence in software systems helps us understand how developers organize packages' responsibilities. Furthermore, it helps to measure the coupling between applications to specific annotation-based APIs or metadata-based frameworks.

2.1.3 Code annotations related work

Like any other language feature, code annotations can bring benefits to the application if appropriately used, but they can also be misused (GUERRA; FERNANDES, 2013; LIMA et al., 2018). Extracting source code metrics might help analyze how this resource is being used and understand its impact on software systems. Furthermore, obtaining threshold values can help understand the common profile of code annotations usage.

The literature, however, lacks work that analyzed and studied code annotations. Most authors have performed studies about how annotations can solve other software engineering problems but not study the code annotation itself. To the best of our knowledge, no previous research contributed to source code metrics and threshold values dedicated to code annotations.

Several works in the literature describe the use of annotations to solve problems and implement solutions for a diverse range of domains. In software engineering, some of them apply annotations to support the implementation of design patterns (MEFFERT, 2006) or to enable architectural refactoring (KRAHN; RUMPE, 2006). However, only a few works evaluated the use of annotations itself, focusing on design practices for metadata modeling or even performing studies to assess how developers are currently using annotations.

The first empirical study about the use of code annotations found in the literature was conducted in an application of fractal-based development (ROUVOY et al., 2006).

The evaluation was based on comparing applications that used a metadata based-framework called Fraclet and others that do not. As a result, there was a reduction of about 50 percent of the hand-written code without losing application semantics. Despite this study revealing some potential in using metadata-based solutions, the result was restricted to the fractal-based development domain.

As mentioned in Section 2.1, it is becoming a recurrent practice that applications interact with metadata-based frameworks through code annotations. To understand these metadata-based frameworks' internal structure, the work in (GUERRA et al., 2013) presents a study to identify and document recurrent solutions used in the development of such frameworks, focusing on how they read and process metadata. The authors analyzed the internal structure of many existing open-source metadata-based frameworks. For instance, one documented pattern is the "Metadata Container", which introduces a class role whose instance represents metadata retrieved at runtime read from code annotations.

Guerra and Fernandes (2013) performed a more general experimental study about the use of metadata-based frameworks. It considered the different architectural scenarios where a metadata-based framework can be applied. The experiment was conducted using undergraduate students, which developed three distinct implementations of four different software components. Each implementation applied a different approach: (i) with no frameworks, (ii) with an object-oriented framework and (iii) metadata-based framework. Unit tests were used to ensure that all implementations fulfilled the requirements correctly.

Guerra and Fernandes (2013) performed also a comparison between the implementations using object-oriented metrics (LANZA; MARINESCU, 2006), questionnaires, observations, development time, and manual code analysis. The strongest conclusion of this study indicates that metadata-based frameworks reduce the coupling between the framework and the application. It also found evidence that a metadata-based framework can reduce development time compared to traditional frameworks. However, the experiment also revealed that when a problem happened, it is harder to debug when a metadata-based framework is used since the behavior is defined indirectly through metadata.

Concerning annotation definition, Rocha and Valente (2011) investigated how annotations are used in open source Java systems. In this study, authors analyzed 106 open source projects from Qualitas Corpus project database (TEMPERO et al., 2010), from which 65 projects used annotations. The only information considered was the number of annotations and their type. A high density of annotations was detected in some of the evaluated systems, indicating a possible misuse. Some other data extracted from this study also revealed that more than 90% of the annotations are in methods, and framework annotations are the most used ones.

An initial survey about legibility on annotated code was conducted by (ALBA, 2011). The author used a questionnaire to present two similar codes that represented different approaches expressing the same semantics, where developers should choose the most legible one. The questionnaire was answered by more than a hundred developers and had 27 questions focusing on the usage of annotations and the implementation of annotation idioms for different domains. The study pointed out that annotated code is perceived as more legible than unannotated one. Besides that, the usage of annotation idioms can improve annotation readability and the context where the annotation was used influences the perception of legibility.

YU et al. (2019) performed a large-scale empirical study on code annotations usage, evolution and impact. The authors collected data from 1094 open-source Java projects on GitHub and performed a historical analysis to assess code annotations. The authors obtained ten interesting findings, of which seven are related to the evolution of code annotations during software development. For instance, most annotation changes occur consistently with the code elements it is configuring, while only 13% of the changes are independent. In code annotation deletion, 31% is because it is inconsistent, and 15% it is redundant. By "inconsistent", the authors refer to code annotations that describe facts on the code elements that are not true, suggesting that the code element evolved, but the code annotation did not. The most common case they found was the `@SupressWarning()`. It usually contained a message such as "deprecated method". However, the method changed for one that was not deprecated, but the developers still left the annotation behind. Another interesting finding is related to code annotations changes. The authors found that in 20% of the cases, only the schema was changed. In other words, the code annotation had the same name but belonged to a different metadata-based framework.

The authors also performed some measurements using a metric that counts the number of code annotations on code elements, very similar to the AED (Annotations in Element Declaration) metric that we propose and define in Chapter 4. They found that in 98.8% of the cases, AED is 1 or 2, and the largest is 41.

Comparing the work of YU et al. (2019) with our work being presented is that they do not propose metrics exclusively for code annotations. Even they use a measurement very similar to the AED metric (defined in Chapter 4), they do not define it as a metric. Their work is concerned with observing how code annotations evolve and the relation between annotations and other code elements. Furthermore, in our work, not only do we propose means to measure annotations, but we also propose a polymetric view for their visualization (Discussed in Chapter 5). As such, we can visualize and comprehend their presence and distribution in a software system. However, we can benefit from their findings and try to visualize, for instance, inconsistent annotations and annotations from wrong packages. In short, our work is more focused on usage rather than evolution.

2.2 Source code metrics and threshold values

Source code metrics help summarize particular aspects of software elements, detecting outliers in large amounts of code. They are valuable in software engineering since they enable developers to control complexity, making them aware of abnormal growth of specific system characteristics. They also aid in monitoring the quality of the code (LANZA; MARINESCU, 2006).

There are several well-known source code metrics such as LOC (Lines of Code), WMC (Weighted Methods per Class), CBO (Coupling Between Objects), NOM (Number of Methods), and many others (CHIDAMBER; KEMERER, 1991). However, to effectively take advantage of metrics, they should provide meaningful information and not just numerical values (LANZA; MARINESCU, 2006).

In this context, some studies on the analysis of source code metrics are limited to not assessing whether the average metrics values are statistically representative for such analysis. Some related works on software metrics, including source code metrics, have shown that their data follow the power-law distribution (CLAUSET et al., 2007), in particular, object-oriented metrics. It means that the source code metrics follow statistical distributions that, in general, the average value is not meaningful (CLAUSET et al., 2007).

For example, Wheeldon and Counsell (2003) evaluated 4 Java projects: JDK (Java Development Kit), Apache HTTP Server, Ant, and Tomcat. They were able to identify power-law distribution for metrics that measured the number of attributes and methods.

The work of (BAXTER et al., 2006) collected metrics from 56 Java free software projects. They showed that not all metrics follow a power-law distribution, such as fan-out, number of attributes, and public attributes.

Louridas et al. (2008) partially evaluated 11 projects (10 open sources and 1 restrict) written in Java, C, Ruby, and Perl. They investigated the values of fan-in and fan-out of modules and classes. Their results show that these metrics have distributions belonging to a power-law family, regardless of the programming paradigm.

Based on these studies, we can argue that source code metrics do not necessarily follow an exponential distribution (such as a power-law) or a normal distribution. However, Lanza and Marinescu (2006) defined three threshold values for source code metrics such as LOC (Lines of Code), NOM (Number of Methods), and CYCLO (Cyclomatic complexity) based on statistical measurements. They used these three threshold values on 37 projects developed in C++ and 45 developed in Java, some open-source. They generalized the analysis and considered that the value of the metrics follows a normal distribution. Using the average value and the standard deviation, they created reference regions and used thresholds to be a delimiter.

As seen, the statistical distribution of source code metrics has been extensively studied. On the one hand, metrics for object-oriented programs, written in Java, seem to follow an exponential distribution (WHEELDON; COUNSELL, 2003; POTANIN et al., 2005; CONCAS et al., 2007; FERREIRA et al., 2009; YAO et al., 2009). On the other hand, there is evidence that not necessarily some metrics will follow such distributions (BAXTER et al., 2006; LANZA; MARINESCU, 2006; HERRAIZ et al., 2011; HERRAIZ et al., 2012).

Our work proposes a novel suite of software metrics (discussed in Chapter 4 to measure code annotations. To understand their behavior and obtain threshold values, we extracted their values from 25 real-world open-source software. We worked with the hypotheses that they would follow a power-law distribution, similar to some object-oriented metrics, and therefore the average value would not be representative. Although all seven metrics follow a power-law distribution, some have pretty small values, as will present in our discussions. For these specific ones, a normal distribution could be used alongside the average value. However, our approach considering a power-law distribution and percentile analysis was shown to be more accurate.

2.3 Software visualization

Software systems are complex and, nowadays, can become very large. For instance, the well-known Hibernate[2], a metadata-based framework for object-relational mapping, has more than 40 thousand classes and more than 550 thousand lines of code. This is a huge system and an example of an amount of information challenging to manage and comprehend. According to Hasselbring et al. (2020), software comprehension is concerned with how software engineers maintain existing software, and it is an essential process since developers spend most of their time understanding existing software. Furthermore, when developers are newcomers to a project, they are like *explorers within an unfamiliar landscape*, who will encounter many obstacles before finally settling in (DAGENAIS et al., 2010).

Software comprehension is also related to software evolution, recognized as the most costly and challenging activity in the software development life cycle. Software systems require constant evolution and changes triggered by new requirements. Programmers must comprehend the existing source code before adding new functionalities or new properties (RAJLICH, 2014; SUN et al., 2015). Moreover, as stated in (WETTEL; LANZA, 2007a), the more familiar a team is with a program, the easier it becomes to understand the impact of any modification they may want to perform. Therefore, given the time spent understanding software, challenges newcomers face, and the need to evolve software systems, studying approaches that ease and speed up software comprehension becomes necessary.

In this context, software visualization has been researched and used to analyze aspects of complex software systems, such as comprehension (FRANCESE et al., 2016; ROMANO et al., 2019), and analyze version control repositories (GREENE et al., 2017). Furthermore, according to Diehl (2007), 75% of all information from the real world is perceived visually, which motivates researching strategy to *display* software systems. However, Brooks (1987) mentions that software is complicated to visualize because it is not inherently embedded in space, which means it has no ready geometric representation. Therefore even though most information is perceived visually, software systems have no natural shape for visualization. For this reason, researchers keep investigating how software can be displayed so that humans can visualize it and understand the underlying structure.

Following, we present two approaches for software visualization, as described by

[2]`https://github.com/hibernate/hibernate-orm`

(FRANCESE et al., 2016). The first, synthetic natural environments, is mapping software to familiar environments to humans, such as a city. The second is polymetric views, in which geometric forms are used to represent software. Naturally, both these approaches can and are combined into a single visualization, but for clarification and following the guideline in (FRANCESE et al., 2016), we will discuss them separately.

The final section presents our first draft for code annotations visualization, a polymetric view that combined circles and rectangles shapes. We also discuss the lessons learned from this first work that we used to improve the approach and present the novel version in Chapter 5.

2.3.1 Synthetic natural environments

When it comes to SNE (Synthetic Natural Environments), we are interested in creating familiar environments artificially (hence the term SNE). The strategy uses metaphors, such as a city, town, forest, and even the solar system, and map software characteristics to this ambient. According to (WETTEL; LANZA, 2007a), familiarity has an important influence on program comprehension strategies, and familiarity is strongly related to habitability. Therefore it makes sense that software engineers researches try to use known environments to represent software.

Possibly the most well-known metaphor used for software visualization is the *city metaphor* proposed first by (KNIGHT; MUNRO, 2000), further investigated and popularized by (WETTEL; LANZA, 2007b), where the authors developed the tool CodeCity to implement this metaphor.

A city is a familiar concept and has a natural sense of orientation. One of the goals is to bring this sense of orientation to software during an analysis process. Furthermore, a large city is a complex construct that is only explored incrementally, just as developers do with software. They explore the system in a step-by-step fashion, becoming familiar first with parts they are interested in (WETTEL; LANZA, 2007b).

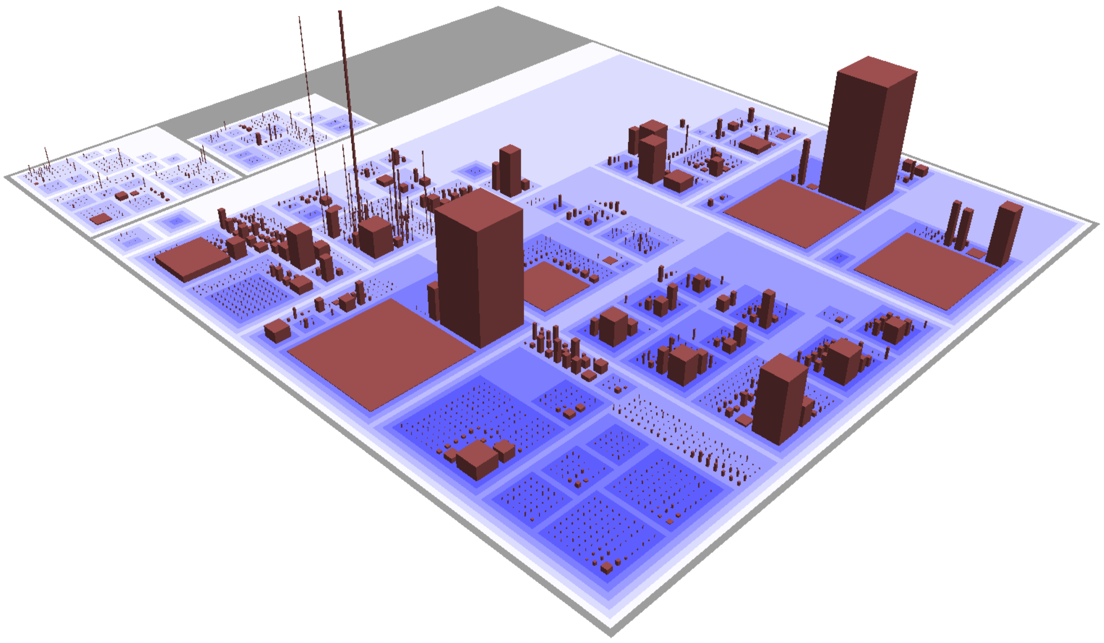
The *city metaphor* represents types (classes, interfaces, enums) as buildings (parallelepipeds). In turn, these buildings are localized inside districts, which represent packages. The visual properties of each parallelepiped represent software metrics of the class:

- The height of the building reflects the value of the metric Number Of Methods (NOM) —the taller the building, the higher the number of methods.

- The base size of the building corresponds to the value of the metric Number of Attributes (NOA) —the larger the base, the higher the number of attributes (or class members).
- The color of the building is mapped to LOC the value. — dark blue means few lines of code, while light blue means many lines of code.
- To represent the nesting level of packages, the color of district blocks ranges from dark grey to light grey based on the nesting level of the packages.[3].

Figure 2.4 presents an example of the project ArgoUML represented as a software city through the CodeCity tool.

Figure 2.4 - Example of project ArgoUML represented by the city metaphor.



Source: Wettel and Lanza (2007b).

As reinforced by Wettel et al. (2011), classes and packages are the primary orientation points of developers. For this reason, the city metaphor was initially designed

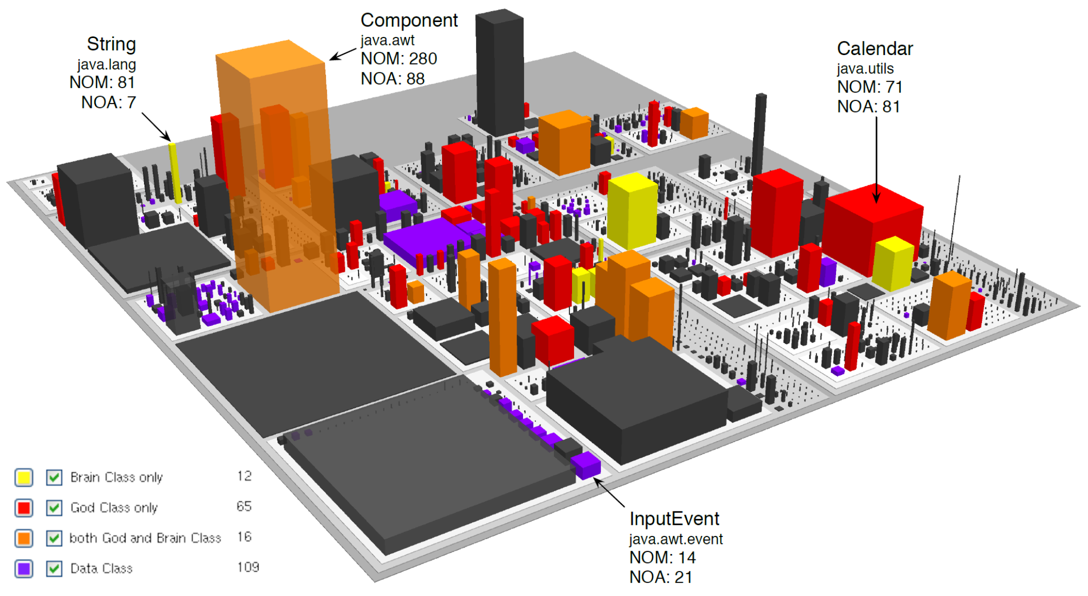
[3]The colors used in CodeCity may be different in each version of the tool. However, the color is still used to display the LOC value. We presented as it is implemented in the last version of the tool, Code2City

to show only these key elements and not class internals. For a largescale understanding, these finer-grained elements are not necessary. It would probably display overplotting problems and does not represent how someone explores a large city for the first time. When getting to know a city, we do not explore a specific house or street right at the beginning.

Although software engineers' researchers agree that displaying fine-grained details might be troublesome and confuse the user of a software visualization tool, as said in (WETTEL; LANZA, 2007a; WETTEL et al., 2011), they also mention that approaches should be studied to display these details. In our visualization approach, CADV (presented in Chapter 5), we also consider this aspect of preventing users from being overwhelmed with code annotation metrics.

The city metaphor was further explored in (WETTEL et al., 2011). The authors conducted experiments with 41 participants from industry and academia to empirically evaluate the CodeCity tool. They added more features to the CodeCity tool and allowed the visualization of design smells in the code. To reach this goal, they assign vivid colors to design problems and color the affected artifacts accordingly. Figure 2.5 presents an example of CodeCity displaying potential smells in the JDK 1.5.

Figure 2.5 - CodeCity displaying design smells in JDK version 5



Source: Wettel et al. (2011).

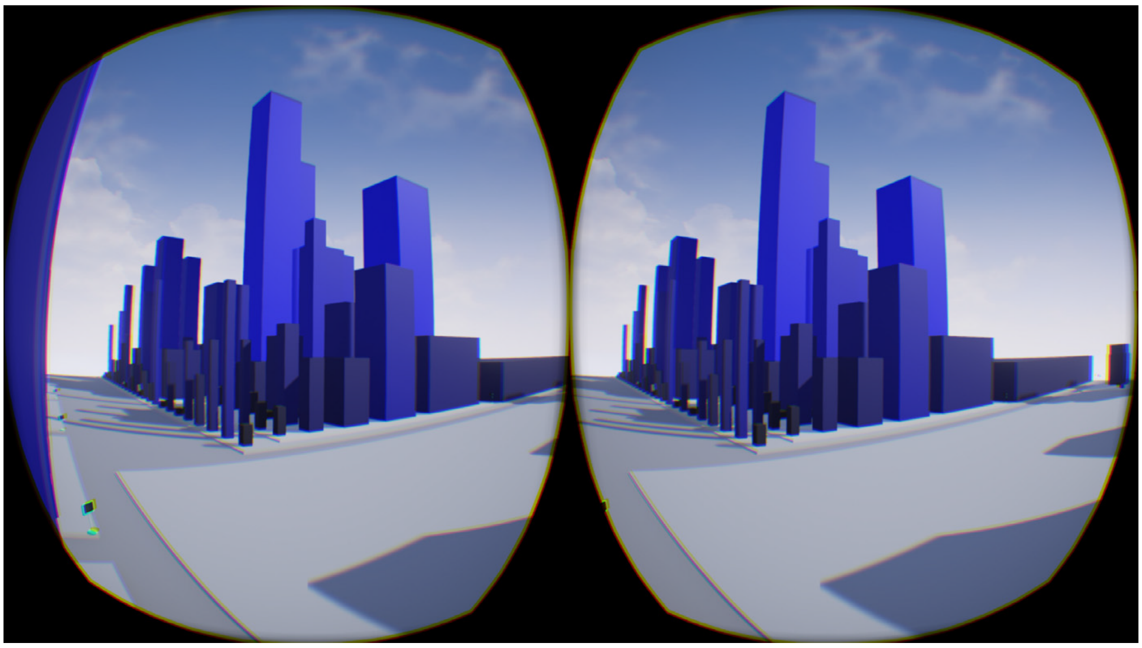
In (WETTEL et al., 2011) the authors performed an experiment to assess the CodeCity and *city metaphor*. They elaborated nine questions that the participants should answer using CodeCity. For instance, one question was to *Locate all the unit test classes*, and another *Find three classes with the highest number of methods (NOM)*. As for the identification of design smells, one question was *Identify the god class containing the largest number of methods in the system*. From the results, the authors obtained an increase of (+24%) task correctness and completion time (-12%) when comparing to traditional tools such as IDEs. Furthermore, the tasks that benefit from an overview of the system, CodeCity constantly outperformed manual code inspection. Quickly comparing these results with the ones from our experiments (discussed in Chapter 5), we also observed this behavior when it comes to questions related to a general view of the system.

The city metaphor also gained a Virtual Reality (VR) version (ROMANO et al., 2019), and a new tool, Code2City, was developed to support both VR and flatscreen visualization. The authors conducted experiments with 42 participants comparing three different approaches. They used the Code2City displayed on a regular computer screen (i) and the VR version (ii). They also used a plugin for the Eclipse IDE named *Metrics and Smells* that collects metrics and detects bad smells (iii). The authors concluded that the city metaphor increases software comprehension, and users using the VR version concluded the experiment tasks more quickly and were more satisfied. The tasks were the same as the ones used in the work of (WETTEL et al., 2011). Figure 2.6 displays a software system as a city in the Code2City VR version.

We mainly discussed the *city metaphor* in this section. However, there are other *synthetic natural environments* proposed by other authors. For instance, the Forest Metaphor (ERRA; SCANNIELLO, 2012) depicts software as a forest of trees, and (GRAHAM et al., 2004) proposes a software where each sun represents a package and planets are classes. Orbits represent the inheritance level of a class within its package.

The visualization for code annotations that we propose and discuss in Chapter 5 is not directly based on the *city metaphor* or other synthetic natural environments. However, we take some important features presented in these works. For instance, we also use colors to present relevant information to the user, we use metrics values to determine the dimensions of geometric shapes, and we also carry experiments with tasks similar to those used to validate the *city metaphor*. Our original contribution is that we are focused on displaying code annotations characteristics. Furthermore, our visualization approach is an ongoing work that we will continue to evolve. Therefore, one of the possible approaches is to adapt the *city metaphor* for code annotations and validate it.

Figure 2.6 - Code2City displaying a software system through the Oculus Rift.



Source: Romano et al. (2019).

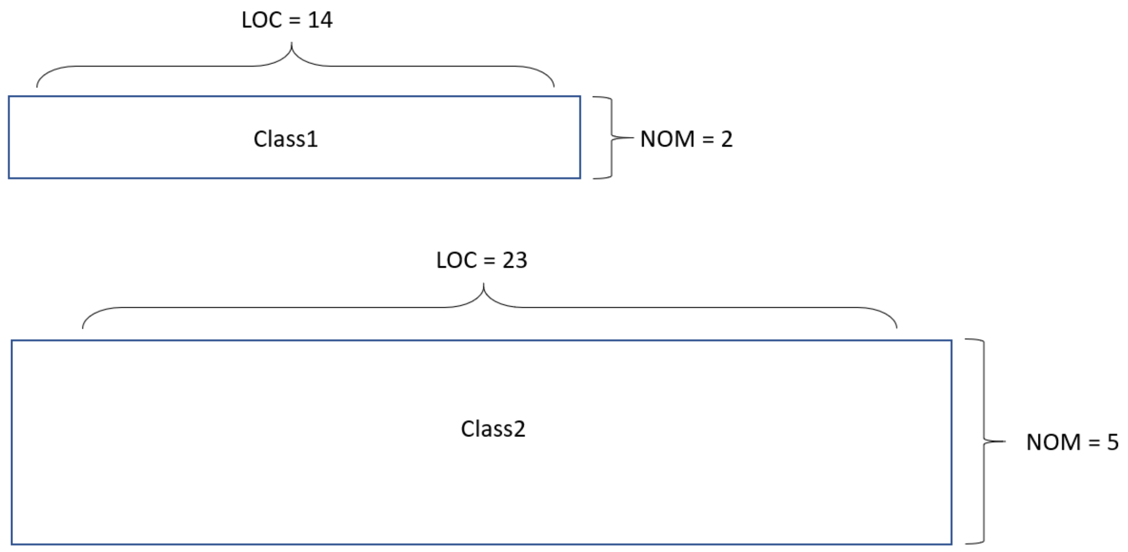
2.3.2 Polymetric view

Lanza and Ducasse (2003) introduces the concept of a polymetric view, a lightweight software visualization enriched with software metrics. In other words, the visualizations are created based on metrics values. Consider the rectangle on Figure 2.7 representing two hypothetical classes, `Class1` and `Class2`. The measures of this rectangle are extracted based on two metrics LOC (Lines of Code) and NOM (Number of Methods).

Observing the rectangle that represents `Class1`, we can see that the class has 14 lines of code (LOC) and two methods (NOM). Moreover, observing the rectangle that represents `Class2`, it has 23 lines of code (LOC) and five methods (NOM). One interesting observation is that we can quickly spot that rectangle representing

`Class2` is wider than the rectangle representing `Class1`. This means `Class2` has more lines of code than `Class1`. This information was obtained purely visually, and no code inspection was required. We assume that the user knows what metrics were used to draw the rectangles, which can vary depending on the designers of the polymetric views.

Figure 2.7 - Simple Polymetric View.

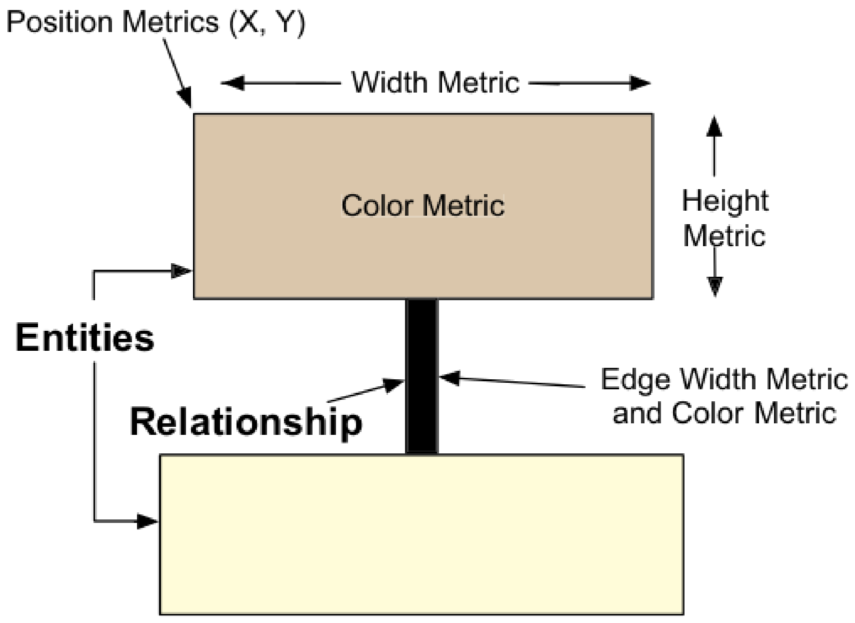


The original polymetric view proposed in (LANZA; DUCASSE, 2003) was designed to support metrics and relationships between entities. Figure 2.8 displays the original design for a polymetric view. Each entity is represented by its node with an edge connecting them. This visualization can support five metrics on each node and two metrics on edge representing the relationship.

Beginning with the width and height of the nodes, we have can have two metrics values. The example we illustrated in Figure 2.7 uses exactly these two characteristics to represent our entity, which was a class. Following the original polymetric view, we can add color to the node representing the third metric. The convention is that the higher the measurement, the darker the node. Finally, the X and Y coordinates can also be used to represent two additional metrics values. However, this strategy requires an absolute origin within a fixed coordinate system, which is not trivial for all kinds of metrics.

The edge used to connect these entities may also provide information. First, the width of the edge can be used to represent any measurement. The convention is that the wider the width of an edge, the higher the measurement. Secondly, we can also use the color of the edge to display a measurement. Similarly, the convention is that the higher the measurement, the darker the edge is. Therefore, the polymetric view was initially designed to support seven metrics through colors, position, and dimensions. The first tool developed to display polymetric views of a source code was the CodeCrawler (LANZA, 2004).

Figure 2.8 - Original Polymetric View.



Source: Lanza (2004).

Designers do not need to completely adhere to the original polymetric view and adapt to its needs. Furthermore, as we already discussed, some research finds that this number of displayed metrics may be too high and can confuse users. Our polymetric view designed for code annotations (discussed in Chapter 5) uses code annotation metrics values to enrich circles dimensions. However, we display few metrics in a view to not overwhelm the user with too much information. We use colors to identify what code annotation schema is being used. Therefore, we can say that the color also provides relationship information since a class becomes coupled to every code annotation it uses.

Francese et al. (2016) propose a polymetric view for traditional and object-oriented metrics. Their work aims to provide an overview of the observed software in terms of size, complexity, and structure. In other words, their view aims to aid software comprehension. Not only static information but also identify how classes are exchanging messages. Their view is created as a direct graph where each basic unit is a "type definition" (class, interface, enum, etc.), and it will be the node of the generated graph.

Each node is drawn as both an ellipsis and a rectangle. Their measures are directly extracted from several source code metrics. Figure 2.9 presents the node proposed in their work. From the Figure, we see that, for instance, the WMC denotes the ellipsis width, and LOC determines the rectangle height of a class definition. They also use colors to represent a metric value. For instance, the border's color in the rectangle is used for NM (Number of Methods). It ranges between light and dark red. Light red represents a smaller metric measurement than dark red. This color pattern is similar to the original definition of the polymetric view in (LANZA; DUCASSE, 2003), i.e., the darker the color, the higher the measurement.

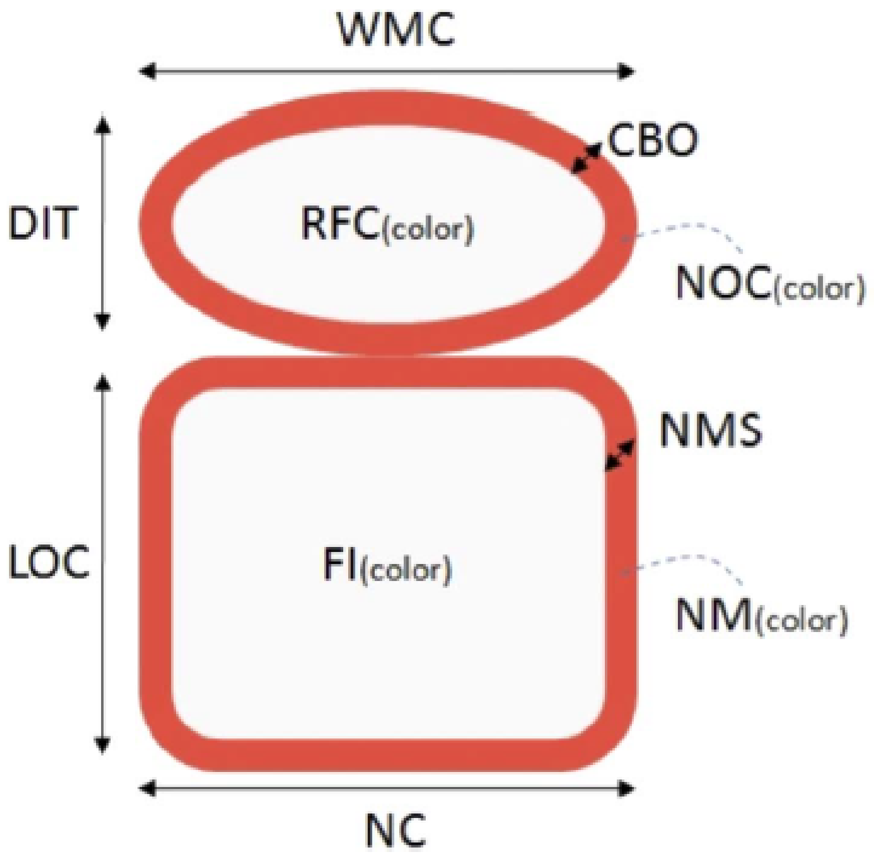
For clarification about the metrics used, we briefly present them here as defined in (FRANCESE et al., 2016).

- Weighted Methods per Class (WMC). It is defined as being the number of all member functions and operators defined in a class.
- Depth of Inheritance Tree (DIT). It measures the number of ancestors of a class.
- Number Of Children (NOC). It gives the number of direct descendants for a class.
- Coupling Between Object classes (CBO). It is the number of classes to which a given class is coupled with.
- Response For a Class (RFC). It measures the number of methods that can be potentially executed in response to a message received by an object of that class.
- Number of Comments (NC). It represents the number of comments lines of a class.
- Lines Of Code (LOC). It indicates the number of all non-empty and non-comment lines of a class.
- Number of Message Sends (NMS). It measures the number of messages from a method. This metric is conceived to be an unbiased measure of

method size. The value for a class is achieved by summing the NMS values of all the methods in a class.

- Number of Methods (NM). It represents the number of methods of a class.

Figure 2.9 - Polymetric View for Object Oriented Metrics.



Source: Francese et al. (2016).

The whole system will be drawn as a directed graph containing all the nodes and their relationship, i.e., how the classes interact. This visualization was implemented as an Eclipse Plugin called MetricAttitude(FRANCESE et al., 2014). To evaluate how their approach aided in Java software comprehension, they conducted a questionnaire-based experiment with 18 participants, being 10 undergraduate students and eight professional developers. The target software used was the JLine[4] Participants filled a *comprehension questionnaire* with 16 open-ended questions about the JLine software system. They had to use the MetricAttitude tool to answer these

[4]https://github.com/jline/jline3

questions. Afterward, they filled a *perception questionnaire* about their impressions and opinions of the visualization tool. According to the author's report, the participants found the tool easy to understand and expressed an overall favorable judgment. As for the comprehension, the average correctness of answers has values between 0.70 and 0.87.

In the work of (FRANCESE et al., 2016), the authors do not investigate how participants find the MetricAttitude more useful than code inspection. Instead, they investigate which metrics users find better in the visualization. They compare if users prefer visualizing object-oriented metrics, traditional code-sized metrics, or both combined. They found that users prefer to use them both together.

Comparing the experiments carried in the work of (FRANCESE et al., 2016), in our visualization approach (discussed in Chapter 5), we also conduct a *perception questionnaire* to find the usefulness and ease of use. However, we also ask questions comparing using our visualization approach and manual code inspection.

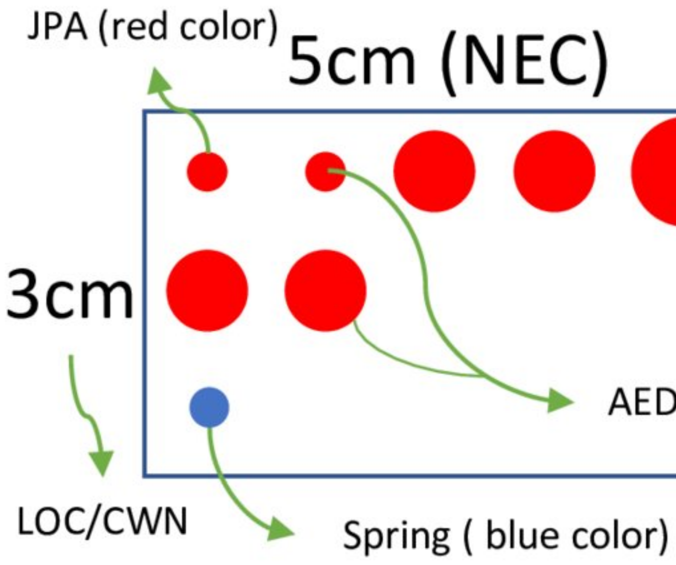
2.3.3 Previous work for code annotations visualization

Our research team designed a first draft for code annotations visualization, and we were able to publish it in (LIMA et al., 2020b). This was our first step in investigating how code annotations could be visualized. In this section we will present what was our first draft, the very first demonstration of the AVisualizer tool, and the lessons we learned from this work.

We used a polymetric approach based on rectangles and circles. We also used colors to identify annotation schemas. Classes and packages are represented as rectangles, and code annotations as circles. Figure 2.10 presents a detailed view of a hypothetical class.

As seen in Figure 2.10, the dimensions of the rectangle representing a class are proportional to the LOC and NEC values. The latter stands for Number of Elements in Class (NEC), a metric used to determine the number of code elements in a class that can be annotated. The rectangle length is determined by the NEC value, which is five in the example Figure. It means the example class has five elements that can be annotated. The rectangle's width is proportional to the LOC value normalized by CWN (Class Width Normalization). This factor is calculated to guarantee enough space in the rectangle to render the circles on top, i.e., the code annotations inside the class.

Figure 2.10 - First Polymetric View for Code Annotations Visualization.



We chose an approach that displays code annotations in columns for each code element they annotate. For instance, if a code element has three code annotations in the source code, then we draw three circles vertically in the column that represents that specific code element. In Figure 2.10, for example, the first column has three code annotations, two red and one blue. In Figure 2.10 we see five columns, this means we have five code elements. Moreover, since every column has at least one circle rendered, there are no code elements without code annotations in the source code.

Code annotations are represented as circles. The annotation schema specifies their color. In this example, we used red for the JPA API, also known as the `javax.persistence` schema. For Spring related schemas, we used the blue color. The radius of the circle is calculated based on the AA (Arguments in Annotation) value. We can also see the AED (Annotation in Element Declaration) metric value in each column, which measures the number of code annotations configuring a code element. Observing Figure 2.10, we see the second column has two arrows pointing at AED = 2. This reinforces that the code element in the second column has two code annotations. The metrics AED, AA, and others will be thoroughly discussed in Chapter 4.

We also draw rectangles to represent packages that contain these classes rendered as rectangles. In Figure 2.11 we present a hypothetical example with two packages.

Figure 2.11 - Polymetric View displaying packages and classes.

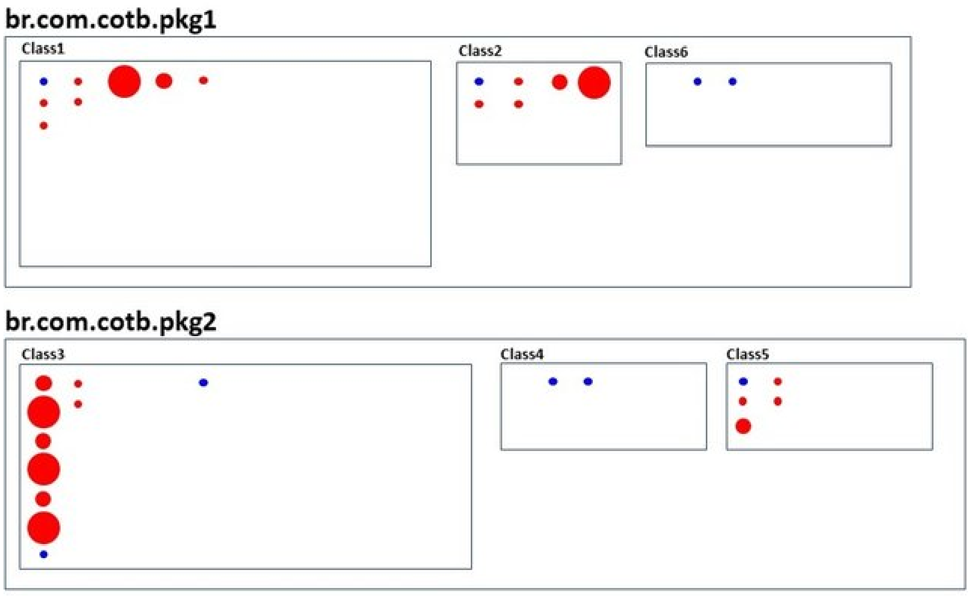
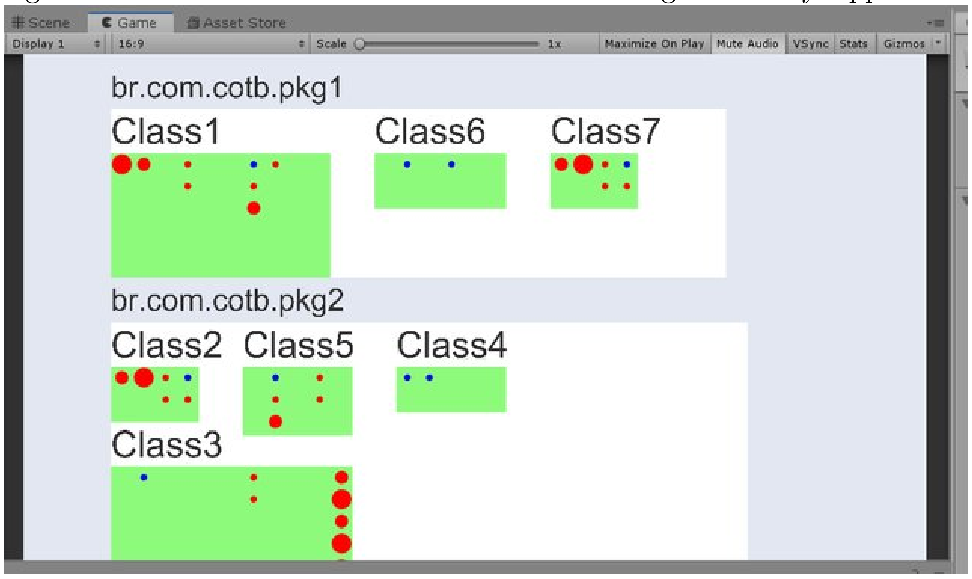


Figure 2.12 - First version of the AVisualizer running as a Unity Application.



Both packages have three classes each. The first version of the AVisualizer was developed using the Unity Game Engine[5], and in Figure 2.12 we present a visualization generated by Unity.

We had much feedback from this previous work and used them to improve our visualization approach for code annotations. For instance, using different geometric

[5]This is a game engine widely used to develop 2D and 3D games

shapes can be confusing in the same visualization. We used rectangles for classes and packages, while we used circles to represent code annotations. Also, this made the approach problematic present nicely the hierarchy naturally present in Java packages. Finally, our strategy posed several challenges in escalating to larger real-world projects. In our new approach, we switched to a circle packing structure that improved these points. Observing the *city metaphor*, in (WETTEL et al., 2011), they only used parallelepiped shapes to draw the buildings and packages. Reinforcing the idea of only using one geometric shape.

Furthermore, we tried to display several metrics in the same visualization, which was confusing for users. We chose only one metric to display in our new approach and instead created different views that individually displayed a metric.

Using the Unity Game Engine as a development environment for a static 2D visualization tool was not trivial. Much of Unity's available functionality is suited for Game Development, i.e., software based on the Game Loop Pattern (NYSTROM, 2014). Furthermore, we realized that using Unity to develop our tool consumes much more resources (memory and CPU) than a standard software engineering tool. This could be a significant drawback for potential researchers and developers adopting our tool to monitor their source code. The tool should be lightweight and have minimal requirements to run. For this reason, the updated version of the AVisualizer tool presents an entirely new approach based on the D3[6] JavaScript library.

As we presented in this section, and to the best of our knowledge, no previous discussions or work was conducted by other researchers to investigate or propose visualization approaches designed specifically to visualize and comprehend code annotations used in software systems (except for our own previous work published in (LIMA et al., 2020b)). As discussed in 2.1, code annotations are increasingly used to perform all kinds of different behaviors, such as source code generation (DAMYANOV; HOLMES, 2004), compile-time verification (ERNST, 2008; QUINONEZ et al., 2008), class transformation (LOMBOK,), framework adaptation (GUERRA et al., 2010), and data mapping. Therefore, we might be able to associate the responsibilities of classes and packages to code annotations. Visualizing them might aid in understanding how a specific software system is organized and structured.

In the following three chapters, we will present and discuss the novel contributions from this work. Beginning in Chapter 3, we present the ASniffer, the tool we devel-

[6] d3js.org

oped to extract code annotations metrics, and we also present how it was used to contribute to other researches. In Chapter 4, we define our novel suite of software metrics alongside a statistical analysis to obtain threshold values. Finally, in Chapter 5, we present our novel software visualization approach, the redesigned version of the AVisualizer tool, and experiments we conducted to validate the visualization approach. Even though, we completely redesigned the visualization approach, the work we published in (LIMA et al., 2020b) was essential to obtain new insights and learn from experts how we could improve our work.

3 ANNOTATION SNIFFER

In this chapter, we present the Annotation Sniffer (ASniffer) tool. Initially developed to support the research in this work, it has grown and been used in other works and researches for code annotations, reinforcing such a tool's usefulness. The ASniffer tool is open-source[1], constantly updated, and was published in the Journal of Open Source Software (LIMA et al., 2020a).

This chapter is divided into three sections. The first is strictly technical and presents details of the architecture and source code of how the tool was built. The second presents how to use the tool. The third section presents and briefly discusses other works and research using the ASniffer tool to support data collection.

3.1 Annotation Sniffer architecture

Initially developed as a plugin for the Eclipse IDE[2], it changed to be a component that can be plugged into other applications. We focused on making it a standalone tool executed through a command line. However, with this structure, it can easily be used in other Java applications and, for instance, be part of a plugin for IDEs such as Eclipse or IntelliJ.

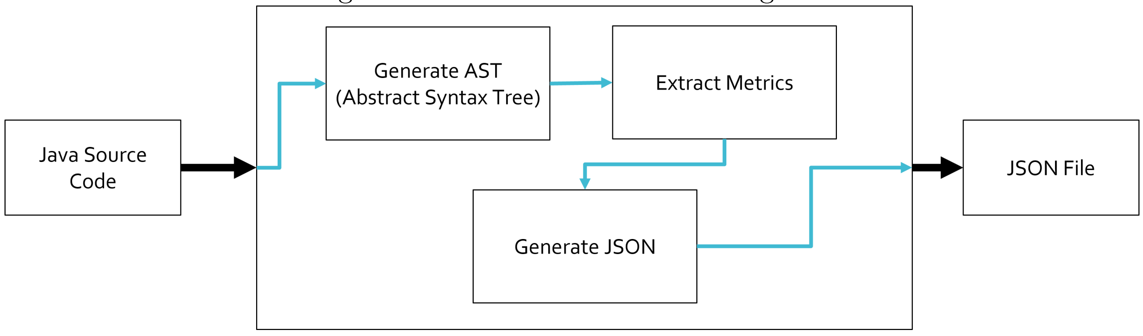
The main goal of the ASniffer is to automate the process of extracting the novel suite of software metrics for code annotations, published in (LIMA et al., 2018) and further discussed in Chapter 4. The ASniffer receives a java source code, extracts the metrics values, and outputs a JSON report. The first versions outputted an XML report but were later discontinued in favor of the JSON.

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

The ASniffer tool uses the JDT[3] (Java Development Tools) API 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. After the processing is done, it generates a JSON report as output. Figure 3.1 presents an overview diagram of the ASniffer tool.

[1]https://github.com/metaisbeta/asniffer
[2]https://www.eclipse.org/
[3]https://www.eclipse.org/jdt/

Figure 3.1 - Annotation Sniffer Diagram.



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

Figure 3.2 - Construction of the AST.

```java
//Code snippet with incomplete code
public AMReport calculate(String path, String projectName) {
    String[] srcDirs = FileUtils.getAllDirs(path);
    String[] javaFiles = FileUtils.getAllJavaFiles(path);

    MetricsExecutor storage = new MetricsExecutor(
                   () -> includeClassMetrics(),
            includeAnnotationMetrics(),
            includeCodeElementMetrics() ,
            projectName);

  ASTParser parser = ASTParser.newParser(AST.JLS8);
  parser.setEnvironment(srcDirs);
    JavaCore.setComplianceOptions(JavaCore.VERSION_1_8, options);
  parser.createASTs();

    return storage.getReport();
}
```

The code in Figure 3.2 presents a snippet of how the ASniffer creates the AST. In lines 3 and 4, we search all files with the extension ".java". In line 6, we prepare the `MetricsExecutor` instance that will invoke all classes responsible for collecting the metrics and will pass the compilation unit to them.

On lines 12-14, we prepare the AST to be built, and finally, on line 15, we create the AST. After the extraction process is complete, we return, on line 17, an instance of `AMReport` which is a class that holds in memory the report.

To exemplify the extraction process, we will use the code responsible for collecting the Annotations in Class (AC) metric, presented in Figure 3.3. We further discuss this and other metrics in Chapter 4, but for now, it is sufficient to know that the AC counts the number of code annotations in a class. As will be further explained, we have three types of metrics: Class Metric, Code Element Metric, and Annotation Metric.

Any class that wishes to traverse the AST must extend the `ASTVisitor` superclass. Furthermore, if it is a "Class Metric", it must implement our custom interface `IClassMetricCollector`. This custom interface provides two methods, the first one, `execute()`, initializes the extraction process, while the second one, `setResult()`, is where the result is stored. The superclass `ASTVisitor` provides methods that are used to visit the nodes from the compilation unit. For instance, for the AC metric, we visit every code annotation encountered.

As seen in the code on Figure 3.3, we have three different types of code annotations. The "MarkerAnnotation" represents a code annotation that has no arguments/parameters, such as the widely known `@Override`. The "SingleMemberAnnotation" represents code annotations with only one argument/parameter. Finally, the "NormalAnnotation" represents the remaining code annotations found in the source code.

The code in Figure 3.3 has three methods to visit each of these types of code annotations. The method in line 7 visits "MarkerAnnotation", line 13 visits "NormalAnnotation" and the method on line 19 visits "SingleMemberAnnotation". In each method, we increment the value of the `annotations` class member. Afterward, we store this value in the instance of the `AMReport` in line 32.

The instance of `AMReport` is a general model that contains every information extracted by ASniffer from the source code. It is from this instance that the default JSON report is generated.

Figure 3.3 - Extraction of the AC Metric.

```java
@ClassMetric
public class AC extends ASTVisitor implements IClassMetricCollector {

  private int annotations = 0;

  @Override
  public boolean visit(MarkerAnnotation node) {
    annotations++;
    return super.visit(node);
  }

  @Override
  public boolean visit(NormalAnnotation node) {
    annotations++;
    return super.visit(node);
  }

  @Override
  public boolean visit(SingleMemberAnnotation node) {
    annotations++;
    return super.visit(node);
  }

  @Override
  public void execute(CompilationUnit cu, ClassModel result, AMReport report) {
    cu.accept(this);

  }

  @Override
  public void setResult(ClassModel result) {
    result.addClassMetric("AC",annotations);
  }

}
```

3.2 Running the annotation sniffer

To use the ASniffer as a standalone tool, the user should run the following command:

`java -jar asniffer.jar`, with the following paramters:

- -p <path to project>
- -r <path to output report>
- -t <report type>
- -m <single/multi>

The “path to project” is mandatory and should be the path to the java project to be analyzed (i.e., contains the source code files). Considering that only one java project is being analyzed, the directory should have the arrangement below:

```
.
|-- project  #Directory containing the source file for the project.
```

In this case, the ASniffer will consider that every “.java” file inside the directory “project” belongs to the same project.

The ASniffer can also analyze multiple projects at once. In this case, the user should provide a directory with the arrangement described below.

```
.
|-- projects        #Root directory for projects.
    |-- project1    #Contains the source files for project1
    |-- project2    #Contains the source files for project2
    |-- ...
```

In this case, the directory “projects” is a root folder, and the sub-directories “project1”, “project2”, and so forth are each different java projects. They can be completely different projects. The user should manually arrange their projects directories to fit the arrangement described above to use this ASniffer feature.

The second parameter, “path to output report”, is optional. It tells the ASniffer where to store the output report file. If no path is provided, the ASniffer will place the report in the “path to project”. This parameter is a path to a directory and should not include any file name or extension“.json” in its name. The output report file will be generated by the ASniffer, with the project’s name being the name of the report file, i.e., “projectsName.json”. The ASniffer assumes that the name of the root directory is the name of the project. If several projects are being analyzed, the ASniffer considers that each sub-directory (inside the provided root directory) is the name of a separate project. Each project will have its output report file placed in the “path to output report” (if provided, or in the “path to project" otherwise).

The third parameter determines the type of output report file. Currently, ASniffer outputs a “.json” file. If no option is provided, ASniffer outputs a default JSON. The output will be placed in a folder called “asniffer_results.”

The following parameters can be used for the type of output file:

- -t json #default output JSON

- -t jsonAV #outputs three json files suitable to be used by the Annotation Visualizer

In Chapter 5 we will perform a thorough discussion about the AVisualizer tool and the polymetric view (CADV) associated with it.

The fourth parameter (single/multi) informs the ASniffer if the "path to project" contains only one project or several projects. If not option is provided, ASniffer assumes it is a single project.

3.3 Annotation Sniffer Usage

As previously mentioned, the ASniffer was used to collect data and aid other research investigating code annotations and schemas. Following, we briefly discuss four works that used the support of the ASniffer tool to collect code annotations related data.

In the work of (LIMA et al., 2017) we were aiming to classify and visualize groups of classes according to code annotations. We created a self-organizing map (SOM), which allows us to visualize groups of data with previously unknown patterns. Therefore, we might be able to identify outliers in these groups. As input to the SOM, we used code annotations metrics values extracted from 24,947 java classes using the ASniffer. The SOM was created using the R language. As a result, we identified that 70% of the classes could be grouped, and the remaining classes can be clustered into three groups. In total, we identified four groups (or clusters).

Cluster 1, which we named 'frequent classes", comprises 70% of the classes according to code annotations usage. In these classes, code annotations are present in at least 50% of the code elements. On the other end, Cluster 4, named "rare classes", represents an outlier group with less than 1% of the analyzed classes. They have a high number of code elements, more than 1500, but very few are annotated. We cannot conclude the existence of a bad smell since future studies are still needed to address these issues.

The work of (LIMA et al., 2017) proposes an annotation-based API, called Esfinge Metadata[4], to retrieve metadata from code annotations and. In short, we proposed an API to aid framework developers in consuming code annotations. By default, the only way to retrieve code annotations is using the Java Reflection API, which can be a complex code.

[4]`https://github.com/EsfingeFramework/metadata`

To demonstrate the usage of the Esfinge Metadata, we refactored e metadata-based framework, Esfinge Comparison [5] to use this new API, instead of the pure Java Reflection API. This framework provides developers with methods that ease the process of comparing instances. After the refactoring, we compared using several code assessment techniques, such as software metrics, including object-oriented metrics, code annotation metrics, and bad smells detection, followed by a qualitative analysis based on source code inspection. The ASniffer was used to collect code annotations metrics values and measure the coupling to code annotation schemas.

As a result, we observed an increase in lines of code, but most were related to the import of code annotations. We also had a reduction in the number of dependencies in components that perform metadata reading. This was confirmed by the reduction in coupling metrics and the elimination of two coupling bad smells. On the other hand, the new version of the Esfinge Comparison framework also increased its number of code annotations, as expected and measured by the ASniffer.

The work of (TEIXEIRA et al., 2018) investigates the presence of code conventions related to metadata configuration that uses code annotations. There is evidence of open-source projects with classes using more than 700 code annotations, most of them being repeated ones. Conventions might be used to avoid these kinds of repetition. This is something used by frameworks that are based on "convention over configuration". The JPA API uses several code conventions, such as using the name of the class and the fields, respectively, for the mapped tables and columns without the need to configure a code annotation explicitly. However, several frameworks avoid adopting code conventions because they might not suit all applications and might be less flexible. In this work, the authors wanted to search if such conventions exist naturally in real-world software and conducted a study that involved a set of Java applications from the Brazilian EMBRACE Space Weather program. The authors analyzed 1314 classes and 5206 code annotations and collected metadata from several code elements. The ASniffer tool was used to extract information about code annotations such as schema, name, and arguments. Afterward, the authors verified if code annotations of the same type are used in code elements that share similarities. They found that 17 conventions could replace 908 annotations (21.42% of the total). In some cases, the same code annotation could be replaced by multiple code conventions or a stronger convention that englobed them all.

In the work of (GUERRA et al., 2020), the authors performed an experiment to com-

[5] https://github.com/EsfingeFramework/comparison

pare the Java Reflection API with the Esfinge Metadata API. The difference between the work in (LIMA et al., 2017) is that this last one only demonstrated the Esfinge Metadata API, and no proper experiment was carried out. To evaluate this API, the authors performed a controlled experiment with two groups of developers. One group used the Esfinge Metadata API, while the other used the Java Reflection API. Both groups developed a metadata-based framework that maps application parameters (command-line arguments) to an annotated class instance. Afterward, a code inspection was carried in every participant's repository. Furthermore, several source code metrics values were extracted, including code annotation metrics using the ASniffer. From the results, the authors observed that the usage of the Esfinge Metadata API provides a more consistent behavior in the evolution of coupling and complexity metrics.

As presented in this chapter, the ASniffer has become a tool basis for every research we conduct about code annotations. For this reason, it is constantly improved and updated to meet the demands we require. Furthermore, being open-source[6], the software engineering community can contribute to the evolution of the tool. The metrics that the ASniffer can extract were published in (LIMA et al., 2018) and will be detailed in Chapter 4. Finally, the ASniffer serves as a backend to our visualization tool AVisualizer, discussed further in Chapter 5, where we present our visualization approach, CADV.

[6]`https://github.com/metaisbeta/asniffer`

4 CODE ANNOTATION METRICS AND THRESHOLD VALUES

This chapter presents the first part of our work, where we define the novel suite of metrics dedicated to code annotations.

The chapter is organized into three parts: We begin presenting the research design used to define and analyze the metrics using open-source projects. Then, in the following section, we define the seven metrics that comprise our novel suite. The third and final section presents the statistical analyses with threshold values and a final discussion with the outlier values.

The contributions and findings from this chapter were published on (LIMA et al., 2018).

4.1 Research design - metrics suite

In this section, we describe the research design used to propose the suite of metrics to ***measure characteristics of code annotations***.

We begin by describing four research questions about what characteristics we can extract from code annotations. Following, we present the research method that we used to analyze and validate these metrics in real-world open-source software, including the process of projects selection and extracting the metrics values, and obtaining threshold values.

4.1.1 Research questions

#RQ1 - What measurements could be performed in the source code to assess the characteristics of its code annotations usage?

Code annotations and annotated code elements have several characteristics that can be measured. For instance, a code annotation has attributes/arguments values that can be defined. Code elements, such as classes and methods, can receive several annotations. To answer this question, we identify essential characteristics and propose a suite of metrics that enables their assessment. Moreover, to provide a means to generate and interpret the proposed metrics, we will use the Goal-Question-Metric (GQM) approach(BASILI, 1992; BASILI et al., 1994).

#RQ2 - For each metric, is it possible to define reference thresholds that can be used to classify its values?

Code Annotation metrics might behave as object-oriented metrics, that in general, might not have a meaningful average value. Our metrics will be extracted from the source code of a set of selected projects, and statistical analyses of the distribution of the values will be performed on them. An analysis based on percentile rank (MEIRELLES, 2013) will also be carried out to provide a first step in interpreting the metrics. Considering that code annotations metrics might behave according to an exponential graph, to answer this question, we also perform empirical analyses based on the percentile rank approach and Lanza's method (based on average and standard deviation values). Afterward, we determine which approach is the most suitable for thresholds values calculation.

#RQ3 - What is the common profile of a class that uses code annotations in Java? How large are the metrics outliers found?

The values obtained for each metric, their distribution, and the code inspection of outliers can help reach conclusions about how the projects use code annotations and how the metrics can aid in detecting potential design problems. It is important to detect huge outliers because they can become a problem in code maintenance. Detecting common profiles in annotated classes may help developers keep track of code quality.

#RQ4 - May the usage of code annotations create problems that can compromise code maintenance?

Currently, the software engineering community lacks studies dedicated to understanding how code annotations can impact software maintenance. Having threshold values available and pointing out the presence of outlier values might indicate if code annotations can indeed become troublesome in maintaining the code. This question will leave much ground to be researched in future studies.

4.1.2 Research method

This section describes the steps performed to answer the research questions.

Step 1 - Propose the metrics

Based on important characteristics of code annotations and annotated elements, as well as using the GQM model(BASILI, 1992; BASILI et al., 1994), we propose a metrics suite composed of the metrics presented in Section 4.2.

Step 2 - Create a tool for the metrics extraction

To enable the metrics extraction in projects, we developed a tool named Annotation Sniffer (ASniffer) that analyzes Java source code and collects information about code annotations. The tool generates a JSON report containing the metrics for all project classes, code annotations, and code elements (LIMA et al., 2020a).

Step 3 - Select projects to be analyzed

A set of open source projects were selected to enable the analysis of metric values in real-world projects. We chose projects that have a different number of annotated classes. For instance, some projects contain 80% of their classes annotated, while others contain only 10% of annotated classes. The selected projects are listed in Section 4.1.2.1.

Step 4 - Data extraction and processing

The metrics values were extracted from the selected 25 projects, providing 24,947 annotated Java classes as our actual sample data. The collected values were submitted to a Python script[1] to prepare the data. Afterward, an R script[2] performed the statistical calculations on the prepared data. A distribution graph was generated for every metric of each project.

Step 5 - Statistical and qualitative analysis

It was analyzed the distribution of the metrics values among all real-world projects and verified if the average value was meaningful. When the normal distribution could not fit the data, a percentile rank analysis was also performed. In short, we compared normal distribution (Lanza's approach) and the percentile rank. Based on the data and code inspection on outliers, a qualitative analysis was performed to identify how the metrics can characterize a typical usage of code annotations and how design problems can be detected.

[1]`https://gitlab.com/annotationmetrics/annotationmetrics/blob/master/data/output_FromXMLtoCSV/glueMetrics.py`

[2]`https://gitlab.com/annotationmetrics/annotationmetrics/tree/master/data/R`

4.1.2.1 Data collection

The collected data came from 24,947 annotated Java classes extracted from 25 real-world projects selected to participate in this analysis. Given that code annotations are an optional language feature, projects were elected considering the presence of code annotations and the domains of the project.

Table 4.1 - Selected Projects.

Project	Repository	Version
Agilefant	`github.com/Agilefant/agilefant`	3.5.4
ANTLR	`github.com/antlr/antlr4`	4.5.3
Apache Derby	`github.com/apache/derby`	10.12.1.1
Apache Isis	`github.com/apache/isis`	1.13
Apache Tapestry	`github.com/apache/tapestry-5`	5.4.1
Apache Tomcat	`github.com/apache/tomcat`	9.0.0
ArgoUML	`argouml.tigris.org/source/browse/argouml/trunk/src`	0.34
Eclipse CheckStyle	`github.com/acanda/eclipse-cs`	6.2.0
Dependometer	`github.com/dheraclio/dependometer`	1.2.9
ElasticSearch	`github.com/elastic/elasticsearch`	5.0.0-rc1
Hibernate Commons	`github.com/hibernate/hibernate-commons-annotations`	4.0.5
Hibernate Core	`github.com/hibernate/hibernate-orm`	5.2.0
JChemPaint	`github.com/JChemPaint/jchempaint`	3.3-1210
Jenkins	`github.com/jenkinsci/jenkins`	2.25
JGit	`github.com/eclipse/jgit`	4.5.0
JMock	`github.com/jmock-developers/jmock-library`	2.8.2
JUnit	`github.com/junit-team/junit5`	5.0.0-M2
Lombok	`github.com/rzwitserloot/lombok`	1.16.10
Megamek	`github.com/MegaMek/megamek`	0.41.24
MetricMiner	`github.com/mauricioaniche/metricminer2`	2.6
OpenCMS	`github.com/alkacon/opencms-core`	10.0.1
OVal	`github.com/sebthom/oval`	1.86
Spring Integration	`github.com/spring-projects/spring-integration`	4.3.4
VRaptor	`github.com/caelum/vraptor4`	4.2.0-RC4
VoltDB	`github.com/VoltDB/voltdb`	6.5.1

To conduct our analysis, we needed a large set of annotated classes as our sample. Selecting these 25 projects was not straightforward. Using random selection was not an option since we needed projects that contained enough annotated classes to provide meaningful data to us. For instance, selecting the top-rated projects on GitHub[3] did not provide the sample we expected since not all of these top projects had enough annotated classes. Projects such as Hibernate, JUnit, and Apache Tom-Cat are not listed in the top GitHub Java projects[4], but they are well known Java

[3]By top GitHub projects, we mean the projects rated with top stars

[4]`https://gitlab.com/annotationmetrics/annotationmetrics/blob/master/`

projects and are recognized for their high code annotation usage. Therefore, they were fundamental in our analysis, in particular, to discuss the existing outliers. The selected projects are shown in Table 4.1.

Table 4.2 - Project Dimensions.

Project	Type	PAC (%)	PAC Category	LOC	LOC Category
Agilefant	Application	10.50	Low Usage	43,539	Low
Dependometer	Application	35.00	Low Usage	28,123	Low
ANTLR	Application	10.60	Low Usage	101,600	High
ArgoUML	Application	31.70	Low Usage	195,670	High
Apache Derby	Application	18.80	Low Usage	689,869	High
Eclipse Checkstyle	Application	44.70	Medium Usage	20,453	Low
JChemPaint	Application	63.70	Medium Usage	27,371	Low
Jenkins	Application	64.00	Medium Usage	124,576	High
Elastic Search	Application	48.20	Medium Usage	615,637	High
Megamek	Application	70.60	High Usage	306,210	High
OpenCMS	Application	72.60	High Usage	476,074	High
VoltDB	Application	80.80	High Usage	542,030	High
Apache Isis	Framework	21.80	Low Usage	163,665	High
Apache Tapestry	Framework	25.60	Low Usage	156,450	High
Apache Tomcat	Framework	31.00	Low Usage	300,819	High
Hibernate Commons	Framework	54.40	Medium Usage	2,812	Low
JGit	Framework	64.80	Medium Usage	173,681	High
Hibernate Core	Framework	54.70	Medium Usage	593,854	High
JMock	Framework	66.40	High Usage	9,580	Low
Metric Miner	Framework	71.00	High Usage	23,602	Low
OVal	Framework	75.00	High Usage	17,381	Low
JUnit	Framework	68.00	High Usage	25,935	Low
Lombok	Framework	69.40	High Usage	50,324	Low
Spring Integration	Framework	76.70	High Usage	208,750	High
VRaptor	Framework	85.00	High Usage	26,660	Low

Based on (NAGAPPAN et al., 2013), we want to determine the similarity and diversity among the selected projects. For this analysis, we propose three dimensions: Type, Percentage of Annotated Classes (PAC), and LOC. Type can assume two values: "framework" or "application". This dimension is important because it allows capturing two different approaches of code annotation usage. A framework has its own code annotations from an overall perspective, and applications use code annotations provided by frameworks. PAC, the second dimension, exposes how many classes contain code annotations. This allows us to fetch projects ranging from low code annotation usage to heavily based ones. Since this analysis aims to show similarity and diversity, capturing projects with different sizes aids in reaching this goal. Therefore, we

`topStarsJavaProjects.txt`

include the well-known LOC metric to measure the project's size. With this dimension, this paper can discuss how code annotations behave in projects with different sizes. Table 4.2 associates each project with the proposed dimensions.

Table 4.3 - Pair Combination of Dimensions.

Combinations	**Number of Projects**
Application - Low Annotation	5
Application - Medium Annotation	4
Application - High Annotation	3
Application - Low LOC	4
Application - High LOC	8
Framework - Low Annotation	3
Framework - Medium Annotation	3
Framework - High Annotation	7
Framework - Low LOC	7
Framework - High LOC	6
Low Annotation - Low LOC	2
Low Annotation - High LOC	6
Medium Annotation - Low LOC	3
Medium Annotation - High LOC	4
High Annotation - Low LOC	6
High Annotation - High LOC	4
Average	**4.69**

For the dimension PAC, we defined three categories: below 35% - "low usage", between 35% and 65% - "medium usage", and greater than 65% - "high usage". For LOC, we defined two categories: below 100 thousand lines - "low" and above - "high". Among the projects, there are 12 frameworks and 13 applications, roughly 50% for each dimension. Considering projects with high annotations usage, we have seven projects classified as "framework" and three classified as "application". This observation highlights the fact that framework projects are usually more annotation-based. From an annotation perspective, the projects have a PAC ranging from 10% up to 85%, which shows the diversity. For LOC, there are 11 projects considered "low" and 14 considered "high".

Combining the proposed dimensions in pairs, there are a total of 16 possibilities, considering the categories that each dimension can assume. The combinations are Type with PAC, Type with LOC, and PAC with LOC. For the pair Type-PAC, we have "application" combined for each of the 3 PAC categories and "framework" for the same three categories. This pattern follows on to a total of 16 combinations. For each pair combination, we count the number of projects that fulfill both dimensions involved in it. Table 4.3 presents the pair combination between dimensions, with the obtained values. There is an average of 4.7 projects per combination, with a minimum value of 2 ("Low PAC - Low LOC") and a maximum of 8 ("Application - High LOC"). With this analysis, we show that, according to our proposed dimensions, we have similar projects and also diverse projects among the chosen 25.

4.1.2.2 Data analysis

We are interested in analyzing the metrics distribution values among all projects and verifying if the average value is representative. As already mentioned, some object-oriented metrics follow an exponential distribution graph, which means that the average value does not contribute to understanding the behavior. This paper investigates if annotation metrics also fall into that same category.

In exploratory data analysis (Section 4.3), based on an approach proposed by (MEIRELLES, 2013), it is verified if the normal distribution can fit the data and make a percentile rank empirical analysis to find where the data seems to be meaningful. We use Lanza's approach (LANZA; MARINESCU, 2006) to calculate possible thresholds values based on average and obtained the percentile rank (5%, 10%, 25%, 50%, 75%, 90%, 95%, 99%) of each metric to find the threshold where the average is not representative. Afterward, we confront each method to conclude whether calculating the average value correctly provides useful information.

On the one hand, Lanza's approach is based on a calculation using the average and standard deviation values used to define reference regions (Low, Medium, and High). In this paper, we use the following nomenclature:

- Lanza-Low = average - standard deviation
- Lanza-Medium = average
- Lanza-High = average + standard deviation

The value of a metric is considered an outlier when it is 50% greater than the maximum Lanza-High threshold. In summary, (LANZA; MARINESCU, 2006) assumes that the average value is meaningful.

On the other hand, adapted from (MEIRELLES, 2013), our percentile ranking analysis considers a more realistic approach and yielded better threshold values. Percentile analysis can be a more flexible way of obtaining the thresholds. As discussed in Section 4.3, we use the percentile 90 as a reference point. For instance, (MEIRELLES, 2013) showed that for object-oriented metrics, the percentile reference is 75. In a normal distribution, we observe that the median is the reference point because its value is close to the average one. The percentile analyzed in this paper is divided into three boundaries, explained below.

- Very Frequent = until percentile 90
- Frequent = between percentile 90 and 95
- Less Frequent = between 95 and 99

Using the average and standard deviation (Lanza's approach) is a strict rule. So it supposes that every metric will behave the same way, which is not true, as we present in Section 4.3. Some metrics distributions have an abrupt growth in the final percentiles, in general from the percentile 90. Hence, the percentiles analysis provides better visualization of what is happening.

4.2 Suite of code annotations metrics

This section presents the definitions of the metrics suite used to assess the quality and complexity of annotated codes, measuring how they are used to implement software. The metrics are classified in this work according to the code element used as a basis, which can be the annotation itself, an element (such as a class, method, or member declaration), or an overall class perspective. All of the proposed metrics are extracted directly from the source code by using the Annotation Sniffer tool. For this reason, the suite proposed can be considered a set of primary metrics and may allow future metrics to be derived from them (GRADY, 1987).

The source code defined in Figure 4.1 exemplifies how the metrics are extracted. The annotations of such code exist in real-world software. However, their usage mixed in the same class are only for illustrative purposes.

We designed a GQM model to guide the definition of the metrics suite. Our model consists of four questions, which aim to understand how we can assess annotated code. With these questions, we were able to propose seven metrics that, combined, provide enough information to answer the proposed questions from our GQM model presented in Table 4.4.

Figure 4.1 - Code used to exemplify the metrics values.

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

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

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

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

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

}
```

Alongside the GQM, we grouped the proposed metrics into three categories: i) Class metrics, ii) Annotation Metrics, and iii) Code Element Metrics. A class metric measures the annotation from a class perspective. An annotation metric measures the annotation declaration itself. A code element metric measures the annotation on a declared element (a class declaration, a method, or a member).

The following subsections define the metrics, explain how each one can help to answer the metrics elaborated in our GQM model, as well as, classifies them, and uses Figure 4.1 to illustrate a simple extraction of each of them.

Table 4.4 - GQM approach applied for our annotation metrics proposal.

Goal	(Purpose)	*Assess*
	(Issue)	*the usage of*
	(Object)	*annotated code*
	(Viewpoint)	*from software developer viewpoint*
(Question)	Q1	**What is the amount of information defined in an annotation?**
(Metric 1)	**AA**	Arguments in Annotation
(Metric 2)	**LOCAD**	LOC in Annotation Declaration
(Question)	Q2	**How complex can an annotation be?**
(Metric 1)	**AA**	Arguments in Annotation
(Metric 2)	**LOCAD**	LOC in Annotation Declaration
(Metric 3)	**ANL**	Annotation Nesting Level
(Question)	Q3	**What is the amount of metadata defined in a class by using annotations?**
(Metric 4)	**AED**	Annotation in Element Declaration
(Metric 5)	**AC**	Annotations in Class
(Metric 6)	**UAC**	Unique Annotation in Class
(Question)	Q4	**How a class is coupled with annotation schemas?**
(Metric 6)	**UAC**	Unique Annotation in Class
(Metric 7)	**ASC**	Annotation Schemas in Class

4.2.1 Arguments in Annotation - AA

Arguments[5] inside code annotations are an important characteristic to be used to provide additional information. A high number of arguments in the same code annotation can result in messy and hard-to-read code. The metric Arguments in Annotation (AA) measures the number of arguments inside an annotation in a code element.

It is possible to define an annotation or a list of annotations as the type of arguments inside an annotation. In the case of nested annotations, each one is reported separately with their respective argument number. AA is classified as an Annotation Metric.

Considering the example of Figure 4.1, the value of AA for the annotation @DiscriminatorColumn is 2, since it has the arguments discriminatorType and name. Another example is @AssociationOverrides, which contains some nested annotations. Its AA value is 1, since there is only 1 argument (value), while both @AssociationOverride contains 2 arguments each (value and joinColumns); therefore, their AA value is 2.

[5]Initially, we used the term *attribute*. However, to not conflict with the C# attribute, we opted for the term *arguments*. The reason was to keep the metrics as generic as possible.

Based on the GQM from Table 4.4, AA contributes to the answers of Q1 and Q2. That can be justified because the number of arguments is directly related to the amount of information present in the annotation, reflecting the number of parameters that should be defined, influencing code maintenance. The pseudocode in Algorithm 1 formally defines the AA metric.

Algorithm 1 Arguments in Annotation Pseudocode

```
procedure AA($annotation$)
    Initialize $aa = 0$
    for each $argument$ $in$ $annotation$ do
        $aa \leftarrow aa + 1$
    end for
end procedure
```

4.2.2 Lines of Code in Annotation Declaration - LOCAD

The annotation arguments are limited to primitive types, Strings, enums, instances of a Class, other annotations, or an array of any of these. Due to such limitations, sometimes the information of an argument can be defined using a large String. Because of that, only the number of arguments defined in an annotation might not be enough to reflect the amount of information defined in an annotation.

The metric LOC (Lines of Code) in Annotation Declaration (LOCAD) measures the number of lines of code used in the source file to define an annotation. LOCAD is classified as an Annotation Metric.

Looking at the code in Figure 4.1, it is possible to verify that the LOCAD for the annotation `@NamedQuery` is 4, although the value of AA is only 2.

Similarly to AA, LOCAD aids in Q1 and Q2. Q1 is justified since there is a direct relationship between the number of lines of code and the amount of information in an annotation declaration. Also, the number of lines declared in annotation influences its maintainability. It is similar to the LOC metric. The more lines a class(or any other code element) contains, the harder it gets to maintain (OGHENEOVO, 2014). The pseudocode in Algorithm 2 presents the extraction procedure for LOCAD. Notice the call `toString`, it returns the lines present in annotation declaration.

Algorithm 2 LOC in Annotation Declaration

```
procedure LOCAD(annotation)
    Initialize $locad = 0$
    for each line in annotation.toString do
        $locad \leftarrow locad + 1$
    end for
end procedure
```

4.2.3 Annotation Nesting Level - ANL

The definition of annotations as arguments of other annotations can help to define a more organized data structure. However, its overuse can lead to annotation definitions that are hard to understand and maintain. The metric Annotation Nesting Level (ANL) measures the maximum level of nesting reached inside an annotation. ANL is classified as an Annotation Metric.

As an example, the annotation @AssociationOverrides in Figure 4.1 has an ANL value of 0. It is easy to figure this out since this annotation is the declaration. Therefore, it is not nested in any other annotation. However, the code annotation @AssociationOverride has an ANL of 1, as it is defined inside @AssociationOverrides. Moreover, the annotation @JoinColumn has ANL value of 2, considering it is defined inside @AssociationOverride which already is the first level of nesting. Notice this metric does not measure the number of arguments in an annotation declaration, while AA does. ANL measures the maximum nesting level of an annotation declaration, which in this example is @JoinColumn, because it has a value of ANL = 2.

According to our GQM model, ANL contributes to the Q2. The higher an annotation is nested, the more complexity is involved in its definition, complicating its maintenance. This is similar to a conditional loop (if-else). The deeper a conditional loop gets, the more complicated it gets to maintain it. The pseudocode in Algorithm 3 presents the ANL extraction procedure. The call `previousCodeElement` returns the type of parent that contains the calling annotation. If it the parent type if an annotation declaration, the call `isAnnotation` returns `true`.

4.2.4 Annotations in Element Declaration - AED

An element in the source code, such as attributes (class members), methods, and classes, may need several annotations to inform some metadata. However, an excessive number of annotations in the same element can also reduce code legibil-

Algorithm 3 Annotation Nesting Level

```
procedure ANL($annotation$)
    Initialize $anl = 0$
    if $annotation.previousCodeElement.isAnnotation$ then
        $anl \leftarrow 1 + \text{ANL}(annotation.previousCodeElement)$
    else
        $anl \leftarrow 0$
    end if
end procedure
```

ity, maintenance, and reusability. The metric Annotations in Element Declaration (AED) focus on each code element individually and measures the number of annotations defined in its context. This metric also counts nested annotations. AED is a Code Element Metric since it focuses on a single element declaration and not the whole class.

As an example, the value of AED for the method exampleMethodA() in Figure 4.1 is 2 since it has the annotations @TransactionAttribute and @DiscriminatorColumn. A more complicated example is the value of AED for the class Example. Counting the nested annotations the value for this element is 7.

This metric is not concerned with the annotation definition itself but rather the number of annotations declared in a code element. With our GQM model, it contributes to Q3 since it reports the number of metadata configured in a class or code element. The pseudocode in Algorithm 4 exposes the AED extraction procedure. Notice there are two procedures involved. The first one counts how many annotations a code element contains in its declaration. For each annotation, nested annotations must be fetched and account for the total AED in a code element, hence the procedure ANNOTATIONS_IN_ARGUMENTS.

4.2.5 Annotations in Class - AC

The total number of annotations of a given class can also be important information for the assessment of how annotations are used in its context. The metric Annotations in Class (AC) counts the number of annotations in all elements of a given class, including nested annotations. AC is a Class Metric since it being measuring from a class perspective and not from a single element.

Although it can be directly obtained by counting all class annotations, it can also be

Algorithm 4 Annotation in Element Declaration

```
procedure AED($code\_element$)
    Initialize $aed = 0$
    for each $annotation\ in\ code\_element$ do
        $aed \leftarrow 1 +$ ANNOTATIONS_IN_ARGUMENTS($annotation$)
    end for
end procedure
procedure ANNOTATIONS_IN_ARGUMENTS($annotation$)
    Initialize $aiA = 0$                                    ▷ Arguments in Annotation
    for each $argument\ in\ annotation$ do
        if $argument.value.isAnnotation$ then
            $aiA \leftarrow 1 +$ ANNOTATIONS_IN_ARGUMENTS($argument.value$)
        end if
    end for
end procedure
```

calculated from the value of AED from all the class elements. The following equation can be used to calculate this metric:

$$AC = \sum_{each\ class\ element} AED \tag{4.1}$$

As an example, the class Example in Figure 4.1 has the value of 11 for the AC metric. All annotations, including the nested ones, are counted. As seen on our GQM model, Q3 is concerned with measuring the number of metadata configured in a class, and the AC metric provides a notion for this. The pseudocode in Algorithm 5 formally defines the AC metric.

Algorithm 5 Annotation in Class

```
procedure AC($class$)
    Initialize $ac = 0$
    for each $code\_element\ in\ class$ do
        $ac \leftarrow ac +$ AED($code\_element$)
    end for
end procedure
```

4.2.6 Unique Annotations in Class - UAC

This metric is similar to AC, but it measures the distinct number of annotations in a class. For an annotation to be considered similar to another, it should have the same annotation type and the same value for all of its arguments, which means that at least two code elements are configured with the same metadata. For instance, if two different code elements contain the annotation, such as @Annotation1(arg1 = “This is an example"), they will be counted only once. They contain the same name (@Annotation1), and their arguments have the same value (arg1 = “This is an example"). If, for example, the argument “arg1” had different values, they would not be a unique annotation since they are configuring different metadata. In short, this metric aims to register the number of distinct metadata configurations that used annotations in the class. Therefore, UAC is a Class Metric.

Based on the metric values of AC and UAC, it is possible to obtain the number of annotations that are similar to other ones in the same class. A high number of similar annotations can reveal repetition in configurations. A high number of distinct annotations means that several particular metadata configurations are performed in the given class. Therefore, if we have a high AC value and a low UAC value for the same class, it means several repeated metadata configurations. Maybe this excessive repetition could be refactored by some code convention related to code annotations. Having available the AC and UAC metric, we can identify these cases.

The UAC value for the class Example (Figure 4.1) is 9. The annotations @TransactionAttribute are considered similar as they have the same argument values, but it is counted only once. However, annotations @AssociationOverride and @JoinColumn are not similar because they have different values for their arguments. Thus, each one is counted separately. The pseudocode in Algorithm 6 formally defines the UAC metric.

4.2.7 Annotation Schemas in Class - ASC

As stated before, annotation schema can be defined as a set of related annotations that belong to the same API. For example, the JPA API has an annotation schema with a set of annotations that applications can use to define object-relational mapping. The code may become tightly coupled with their domains when using annotations from different schemas, raising reusability problems. Also, it may prevent software evolution as well as compromise its readability and maintenance. The metric ASC (Annotation Schemas in Class) measures the number of annotation schemas

Algorithm 6 Unique Annotations in Class

```
define ANNOTATIONS_LIST
procedure UAC($class$)
    Initialize $uac = 0$
    for each $code\_element\ in\ class$ do
        for each $annotation\ in\ code\_element$ do
            if !ANNOTATIONS_LIST.$contains$($annotation$) then
                $uac \leftarrow 1+$ UNIQUE_ANNOTATIONS_COUNT($annotation$)
                ANNOTATIONS_LIST.$add$($annotation$)
            end if
        end for
    end for
end procedure
procedure UNIQUE_ANNOTATIONS_COUNT($annotation$)
    Initialize $uaC = 0$                                ▷ Unique Annotations Counter
    for each $argument\ in\ annotation$ do
        if $argument.value.isAnnotation$ AND
!ANNOTATIONS_LIST.$contains$($argument.value$) then
            $uaC \leftarrow 1+$ UNIQUE_ANNOTATIONS_COUNT($annotation$)
        end if
    end for
end procedure
```

used by a class. For implementing this metric, we considered that annotations from the same annotation schema belong to the same package. Therefore, the extraction is simply done by counting the number of distinct annotation packages imported. ASC is a Class Metric. On our code example illustrated in Figure 4.1 the ASC is 2. The class is using the EJB schema as well as the JPA schema. Both of these were obtained by directly observing the imported package and identifying the packages `javax.ejb` and `javax.persistence`. The pseudocode in Algorithm 7 formally defines the ASC metric.

4.2.8 Measurements classification

The intent of the proposed metrics is to measure certain characteristics about code annotations usage. (BRIAND et al., 1996) proposed a generic mathematical framework that defines several concepts, such as size, length, complexity, cohesion, and coupling, used to classify the metrics. The goal of this section is to classify the proposed metrics according to the concepts proposed by (BRIAND et al., 1996), reasoning how they fulfill their required properties.

Algorithm 7 Annotation Schemas in Class

```
define SCHEMAS_LIST
procedure ASC($class$)
    Initialize $asc = 0$
    for each $code\_element$ in $class$ do
        for each $annotation$ in $code\_element$ do
            if !SCHEMAS_LIST.$contains(annotation.schema)$ then
                $asc \leftarrow asc + 1$
                SCHEMAS_LIST.$add(annotation.schema)$
            end if
        end for
    end for
end procedure
```

AA and LOCAD metrics reflect the **size** properties of an annotation definition: its number of arguments and its lines of code, respectively. These two metrics can be considered similar to measurements for other code elements using traditional object-oriented metrics. For instance, it is also possible to measure the number of lines of code and the number of fields from a class. AED and AC metrics use the same measurement strategy for the number of annotations for a single code element and the class as a whole, respectively. These four metrics fulfill all the requirements of size metrics since they cannot be negative (Non-negativity), are null or zero in the absence of annotations (Null value), and can be summed to reflect the size of two elements (Module Additivity).

ANL reflects a **length** property of an annotation. This classification is justified because the nesting level of a group of annotations is the greatest nesting level among them, which relates to the Disjoint Modules property. In other words, since the ANL is measured individually for each annotation, introducing a new annotation does not increase the overall ANL value. Annotation definitions are static and can always be considered disjoint to each other. The fact that it is not possible to add relationships between them makes this metric fulfil the properties of Non-Decreasing Monotonicity for Non-Connected Components and Non-Increasing Monotonicity for Connected Components.

UAC performs a measurement that counts the number of unique annotations. It captures the number of distinct metadata definitions and according to (BRIAND et al., 1996) properties can be considered a **complexity** metric. The unique annotations from a group of classes cannot be less than the UAC from a single class and cannot be greater than the sum of their values, fulfilling the properties of Module Monotonicity and Disjoint Module Additivity.

Finally, ASC reflects a **coupling** property that characterizes the relationship of a class with annotation schemas. Making an analogy with Afferent Coupling, which count the number of external classes accessed, ASC considers the annotations configured instead of method invocation. It fulfills the properties of coupling metrics since adding new annotations will never decrease its number (Monotonicity), merging two annotation schemas or two classes can only decrease its value (Merging of Modules) and merging unrelated classes will result in a sum of their ASC (Disjoint Module Additivity).

4.2.9 Metrics summary

We summarize the proposed metrics suite in Table 4.5, which has the metric name, its acronym, its reference for measurement, its type based on Briand et al. (1996) concepts as well as a brief summary of the metrics definition.

Table 4.5 - Metrics Summary.

Name	**Acronym**	**Reference**	**Type**	**Summary**
Arguments in Annotation	AA	Annotation	Size	Measures the number of arguments in an annotation definition
LOC in Annotation Declaration	LOCAD	Annotation	Size	Measures the number of lines in an annotation declaration
Annotation Nesting Level	ANL	Annotation	Length	Measures the nesting level of an annotation
Annotation in Element Declaration	AED	Code Element	Size	Measures the number of annotations declared on a code element
Annotations in Class	AC	Class	Size	Measures the total number of annotations in a class
Unique Annotation in Class	UAC	Class	Complexity	Measures the number of distinct annotations in a class
Annotation Schemas in Class	ASC	Class	Coupling	Measures the number of different annotations schemas in a class

The metrics in Table 4.5 refers to 3 different types of code element. The first type, called Annotation, measures information about the code annotation definition itself, such as the number of arguments. The second type, Code Element, deals with measuring code annotations on specific code elements. Finally, the Class type measures the definition of code annotations from a whole class perspective, such as the total number of annotations in a class.

These metrics will guide this study that aims to understand how annotations are used among projects. It is desired to understand how the metrics distribution will behave, hoping that it will be possible to define some threshold values that might enable the identification of scenarios when annotations are being misused.

4.3 Code annotation metrics analysis

This section presents the analysis of the metrics values extracted from the projects listed in Section 4.1.2.1. We worked with a hypothesis that the majority of the proposed code annotation metrics do not follow a normal distribution, possibly indicating that the average value is not meaningful. The average value can be considered the best approach to analyze data when no other information is provided. Thus it is considered that the data being analyzed follows a normal distribution.

All seven metrics values were collected from 24,947 Java classes from real-world projects, which provides enough data for our assessment. With the metrics values collected, the distribution can be analyzed to determine if the average value is meaningful for the metrics.

Values for all metrics were individually analyzed, providing a complete report. Based on that, it was defined the most suitable approach to find thresholds values for each metric. Having thresholds values allows the detection of outliers from these metrics, possibly finding misused annotations on projects. It can also help developers to monitor code annotations usage. For instance, a developer might notice that the number of annotations in a class is way beyond a typical value found in most projects. Thresholds values are not an absolute truth that developers should blindly follow. However, evaluating the metrics based on threshold values helps detect outliers, revealing a possible design problem. Our analysis considered two approaches to determine the thresholds, i.e., Lanza's approach and Percentile Rank Analysis, as discussed in Section 4.1.

We reinforce that our work is not primarily investigating the negative impacts of outliers in software projects, nor are we establishing rules using these thresholds to detect bad smells related to code annotations. These threshold values provide means to interpret the metrics values. Developers should use them to complement their own design decisions, which may vary greatly depending on the systems requirements and domain.

To avoid displaying large tables for each project and each metric, in this section, we chose only one metric, AC (Annotations in Class), to present how the threshold was calculated and discuss the results. The table contains the percentile rank values, ranging from percentile 5 up to 99, for each project. This table also presents the average and standard deviation values obtained from the metrics values per project. Appendix A presents the complete table with the percentile rank values for all projects and metrics.

We also present a second table containing threshold values obtained from Lanza's approach and the Percentile Rank Analysis.

We are proposing three threshold values from the percentile rank analysis:

- Very Frequent
- Frequent
- Less Frequent

The "very frequent" value is the average obtained from the values in the column "percentile 90" for all projects. The "frequent" value refers to the average obtained from the percentile 95 column. Lastly, the "less frequent" value comprises the average obtained from the percentile 99 column.

Finally, we also present the percentile rank chart for one selected project. To present all projects' percentiles rank charts in a single diagram would make the figure unclear and unreadable since the curves are somewhat close to each other. The result is a lot of overlapping curves. Therefore, we chose to select a single project and display its percentile rank. The criteria used to select a project is the one that represents the highest outlier value for that specific metric. For the AC metric, the project was *Hibernate Core*.

Appendix B, however, presents the percentile rank charts in a single diagram for each metric with all projects.

4.3.1 Calculating threshold values

We chose the AC as a target metric to obtain the threshold values, which measure the number of code annotations in a class. We believe this metric is suited to serve as an example because it has higher values making it easy and more comfortable to visualize. The other metrics are overloaded with "zero" values, possibly generating discomfort to read a large table full of zeros. However, as mentioned, Appendix A presents the complete table for every metric.

Since code annotations can be considered an optional feature, not all classes contain it. However, we notice that some classes may contain many annotations, reaching more than 500. Table 4.7 contains the thresholds values determined for the AC metrics.

Table 4.6 - Percentiles from AC metric in all projects.

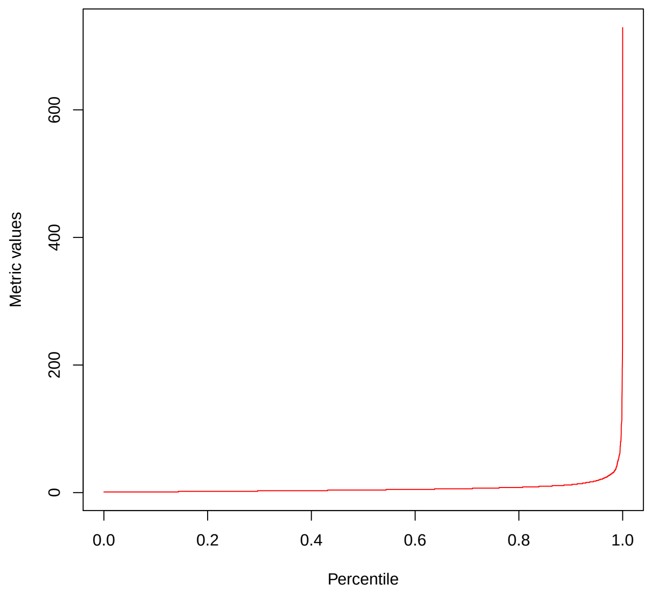
Projects	X5.	X10.	X25.	X50.	X75.	X90.	X95.	X99.	X100.	mean	std
Agilefant	1.00	1.00	3.00	5.00	11.00	22.10	34.05	62.41	128.00	9.67	13.47
ANTLR	1.00	1.00	2.00	4.00	12.00	26.00	33.00	97.68	98.00	10.15	15.25
Apache_Derby	1.00	1.00	1.00	2.00	5.00	10.00	12.50	28.90	53.00	4.27	5.91
Apache_Isis	1.00	1.00	1.00	2.00	5.00	9.00	15.00	33.00	151.00	4.70	7.82
Apache_Tapestry	1.00	1.00	1.00	2.00	4.00	9.00	15.00	35.26	161.00	4.43	7.87
Apache_Tomcat	1.00	1.00	1.00	3.00	8.00	20.00	34.00	82.70	265.00	8.72	19.45
ArgoUML	1.00	1.00	1.00	2.00	4.00	7.00	13.00	19.00	51.00	3.52	4.33
Checkstyle	1.00	1.00	1.00	1.00	1.00	2.00	2.45	3.00	3.00	1.21	0.54
Dependometer	1.00	1.00	1.00	1.00	2.00	2.80	3.00	7.74	9.00	1.65	1.48
ElasticSearch	1.00	1.00	2.00	3.00	5.00	10.00	14.00	30.00	269.00	4.70	9.00
Hibernate_commons	1.00	1.00	1.00	1.50	2.00	3.20	5.00	5.00	5.00	1.85	1.23
Hibernate_core	1.00	1.00	2.00	4.00	7.00	12.00	19.00	46.60	729.00	6.67	14.37
JChemPaint	1.00	1.00	1.00	1.00	2.00	4.40	9.80	19.72	27.00	2.47	4.25
Jenkins	1.00	1.00	2.00	4.00	7.00	14.00	22.00	53.52	131.00	6.85	10.21
JGit	1.00	1.00	2.00	4.00	8.00	16.00	21.00	53.58	169.00	7.28	12.23
JMock	1.00	1.00	1.00	2.00	3.00	4.20	7.20	14.00	14.00	2.52	2.60
Junit	1.00	1.00	2.00	3.00	5.00	12.00	21.00	50.68	228.00	6.11	15.47
Lombok	1.00	1.00	2.00	3.00	5.00	12.00	21.00	50.68	228.00	6.11	15.47
Megamek	1.00	1.00	1.00	1.00	3.00	8.00	14.25	64.20	121.00	4.39	10.41
Metric_Miner	1.00	1.00	1.00	1.00	3.00	11.80	85.60	596.64	723.00	28.94	119.68
OpenCMS	1.00	1.00	1.00	2.00	4.00	8.00	11.00	27.88	63.00	3.82	5.37
Oval	1.00	1.00	2.00	3.00	8.00	8.00	8.00	11.02	22.00	4.44	3.12
Spring	1.00	1.00	2.00	4.00	10.00	17.00	24.75	47.00	281.00	7.66	11.19
VoltDB	1.00	1.00	1.00	3.00	8.00	18.00	27.00	59.92	316.00	7.90	18.78
VRaptor	1.00	2.00	3.00	4.00	8.00	13.00	17.80	40.24	76.00	6.46	7.80

Table 4.7 - Threshold Values for AC metric.

Metric	Percentile Reference	Very Frequent	Frequent	Less Frequent	Outlier Value	Lanza-Low	Lanza-Medium	Lanza-High
AC	90	11.00	20.00	62.00	729.00	-7.24	6.26	19.76

As shown in Table 4.6, the metric values start to become meaningful after percentile 90, showing that only 10% of the analyzed classes has relevant information about its number of annotations. The obtained Lanza-Low value is negative, which indicates

Figure 4.2 - Percentile of AC metric: Hibernate Core.



a high standard deviation value. The AC distribution graph has an abrupt growth after percentile 90, which concludes that the average value of this metric is not meaningful.

For example, projects like MetricMiner and Hibernate Core may reach high values, but most classes have low values. Thus, the average of 28.94 for the MetricMiner project is not representative. More than 90% of the classes have values lower than 12, shown by observing the percentiles.

Using our percentile analysis for the AC metric, considering all 24,947 Java classes, the value chosen as a starting point was percentile 90. The "very frequent" value obtained was 11, "frequent" was 20, and 62 the "less frequent" value. For code annotations, it cannot be stated that a class has a very low number of annotations since it is an optional feature. However, it makes sense to assume that a class has a high number of annotations. Based on this analysis, the frequent region was determined to be between 11 and 20 annotations per class, while values greater than 62 denote outliers.

The higher number found for AC was the CoreHibernateLogger class from Hibernate project with 729 annotations, as presented in Figure 4.2. These annotations configure logging information for all interface methods. For each method, a code annotation is configured the logging level and its respective message. In this case, the high number of methods is probably more problematic than their annotations.

As seen, the percentile rank analysis (MEIRELLES, 2013) is not only based on calculations but also observations. On the other hand, Lanza's approach applies a formula and assumes that the average value is meaningful, possibly resulting in unrealistic values. Therefore, we believe the Percentile Rank Analysis yields more realistic results.

The threshold calculations for the other six metrics follow the same pattern and are presented in Appendix A.

4.3.2 Annotation metrics results summary

This section presented how to analyze and calculate threshold values of the metrics collected from 24,947 Java classes. As an example, we used the AC metric. We used the percentile rank analysis to analyze the distribution of those values and obtain the threshold values. We believed that the distribution would follow an exponential graph. Therefore, the average value would not represent the data.

From the metrics values extracted from 25 real-world projects and with the empirical percentile rank analysis, we were able to reach some thresholds values presented on Table 4.8. Table 4.9 displays threshold values collected using Lanza's approach. These values were approximated to reflect feasible numbers, as no Java class can contain 1.2 annotations. Both these tables summarize the individual thresholds tables obtained for each metric, using our and Lanza's approaches. Notice that we only displayed the individual table for the AC metric in the previous subsection, with the remaining presented in Appendix A. As discussed in the previous subsections, we concluded that the threshold values presented in Table 4.8 are more realistic and might better represent code annotations on real-world projects.

Of the seven analyzed metrics, AA, LOCAD, and ANL have low values even after percentile 90, except for the outliers. Since the values are not very sparse, the average value can be used as a good first guess to represent the data. The other four metrics using the percentile analysis yields better results in representing the data. When the distribution graph has an abrupt growth, such as the AC metric, the Lanza-Low margin yields a negative result because the standard deviation is high. In practical usage, this negative number should be considered 0, but it shows that this calculation is not indicated for every metric, and its distribution should be analyzed first.

Table 4.8 - Percentile Rank Threshold.

Metric	Very Frequent	Frequent	Less Frequent
AA	1.00	1.00	2.00
LOCAD	1.00	1.00	2.00
ANL	0.00	0.00	0.08
AED	1.00	1.00	2.00
AC	11.00	20.00	62.00
UAC	3.00	4.00	9.00
ASC	1.50	1.80	2.40

Table 4.9 - Lanza's Threshold Values: approximated and without negative numbers.

Metric	Lanza-Low	Lanza-Medium	Lanza-High
AA	0.00	0.00	1.00
LOCAD	1.00	1.00	1.00
ANL	0.00	0.00	0.04
AED	0.00	0.00	1.00
AC	0.00	6.00	20.0
UAC	0.00	2.00	4.00
ASC	0.00	0.60	1.30

Our percentile analysis considered a more realistic approach and yielded better threshold values. For instance, the metric AED has a Lanza-High value of 1, while our analysis concluded the number to be 2. Having 2 annotations declared on an element can be considered a "frequent" value. Above this, we consider it a high value or a "less frequent" one. Percentile rank analysis can be a more flexible way of obtaining the thresholds.

To conclude our analysis, the research questions on Section 4.1 are answered in the following paragraphs. Through these questions, we formulated what steps were necessary to take, and by the end, we managed to obtain the needed results and observations to provide to *measure* code annotations characteristics.

#RQ1 - What measurements could be performed in the source code to assess the characteristics of its code annotations usage?

To assess the characteristics of annotated code elements, we proposed a suite of metrics. A total of 7 metrics were proposed, each measuring a particular characteristic of annotations in the source code. We extracted information such as the number, the arguments, and the nesting level of annotations. With that suite of metrics, we gathered all needed data from 24,947 Java classes to perform our empirical analysis. The extracted values and the qualitative and quantitative analysis demonstrated the usefulness of this suite. We consider the proposed metrics to be an important contribution to the Software Engineering community.

#RQ2 - For each metric, is it possible to define reference thresholds that can be used to classify its values?

We were expecting that the average value would not be meaningful since it is known that object-oriented metrics follow an exponential distribution graph. It is no different for annotation metrics; we showed that all seven metrics behave exponentially. The values extracted for the metrics AA, LOCAD, and ANL are low, and therefore, the average could be considered a medium point. However, it fails to find an appropriate high margin threshold, while the percentile rank analysis yields a more realistic value. The average value does not bring much information for the remaining four metrics due to the abrupt growth after percentile 90. Overall, using the percentile rank analysis yields better results due to its flexible nature. Therefore, knowing the distribution of the metrics, we can propose a more realistic threshold value for them.

Threshold values determine boundaries to classify data that falls within a specific region. If thresholds values are obtained, they can help developers maintain control and quality of their source code. For our proposed metrics, the obtained threshold values are intended to be a guide for developers. However, every project has its design criteria, and therefore it is up to the developer to check whether the threshold values make sense for that specific project.

#RQ3 - What is the common profile of a class that uses code annotations in Java? How large are the metrics outliers found?

Since code annotations are optional information added on classes, in general, annotation metrics present low values. Usually, an annotation has 0 or 1 argument. It is defined in 1 line of code. Also, it is not nested on other annotations and is the only annotation in the element where it is defined. A class using around 20 annotations can be considered normal. However, usually, there are many repetitive definitions since the common number of unique annotations is 4. Considering the number of different annotations schemas present, a class usually has configurations of 1 or 2.

Although most cases have low values, we found some outliers with really high values for the metrics in our analysis. We found classes with more than 700 annotations and with 13 different annotation schemas. We also found a single annotation containing 9 arguments and another defined through 58 lines of code. There was an element with 27 annotations, and the nesting level reached the value of 4 (i.e., an annotation in the fourth level of nesting). Having found these outlier opens room for additional investigation on future works.

#RQ4 - May the usage of annotations create problems that can compromise code maintenance?

Code metric considered outlier values are usually related to problems in code maintenance. Some examples are documented code smells, such as Brain Method and Good Class (FOWLER, 1999). The detection strategies for these code smells have rules that consider threshold values for a group of metrics (LANZA; MARINESCU, 2006).

Analyzing the impact of code annotations in code maintenance, the contribution of this study is to reveal the existence of these outliers. These extreme cases illustrate well that abuses in the usage of annotations can happen in the development of real-world projects. For instance, an annotation with 58 lines of code, classes with more than 700 annotations, and a class with 13 different annotation schemas are a clear abuse of annotation usage. As a large class and a large method can compromise code maintenance, the same applies to an annotated element or class with a high number of configurations. Based on that, this finding is evidence that it is important to monitor and evaluate the usage of annotations to avoid such situations.

Each metric captures a particular characteristic. Although we should pay attention to outliers, it is important to combine the metrics for a more precise analysis to verify the problem of how code annotations are used. For instance, the `Java8Parser` class from the MetricMiner project has more than 700 annotations, but the UAC value is only 3, and the number of annotations on each element is only 1 or 2. Observing the class, we notice that the problem is the high number of methods with few annotations. Consequently, due to the repetition of annotations, a general change in the annotations on these methods would be hard to perform. However, the classification and detection of design problems in annotations are out of

> the scope of this work.
>
> Even though the presence of these outliers reveals possible maintenance problems, we consider this result the first step in this direction. A further study should investigate the relation of issues and bugs with code changes in annotations. This question, RQ4, remains open and leaves a lot of ground work for future research.

The metrics discussed and presented in this chapter have also been used in other research on code annotations since they allowed us to extract code annotation information from the source code and carry out these works. Section 3.3 discussed further other works that used these metrics to support the research. As mentioned at the beginning of this chapter, the metrics definition and statistical analysis results have been published in (LIMA et al., 2018).

Now that we have presented our approach to *measure* code annotations, we conclude the first part of our work. With this novel suite of metrics, we can take the next step and propose an approach to visualize code annotations. With this, we begin the next chapter of this work.

5 VISUALIZING CODE ANNOTATIONS DISTRIBUTION

This thesis is divided into two parts. The first one was concerned with proposing the novel suite of software metrics and finding threshold values, discussed in Chapter 4. Now we are interested in visualizing code annotations distributions using these metrics values as input in this second part. We follow the same organization used in Chapter 4 but adapted to a software visualization approach proposal.

The first section presents the research design used to develop and assess our visualization approach. Then, in the following section, we define our visualization approach. We named it Code Annotations Distribution Visualization (CADV) approach. In the third section, we present the conducted experiments, results, and discussion with our findings.

5.1 Research design - Code Annotations Distribution Visualization

Before we present our research design for the CADV, let us revisit the goal of this thesis:

To define an approach to measure and visualize code annotations ***to assess and comprehend*** *their usage and distribution in software systems*

As seen, we have to define an approach to measure and visualize code annotations. In Chapter 4, we defined our novel suite of software to *measure* code annotations. In this current Chapter, we are focused on presenting our approach to *visualize* code annotations, i.e., the CADV.

We built our research design following best practices and guidelines in previous works for software visualization approaches (FRANCESE et al., 2016; MERINO et al., 2018; ROMANO et al., 2019). We begin discussing the goals for CADV, the steps to reach these goals, and our experiment design. When we defined our novel suite of software metrics, presented in Chapter 4, we performed a statistical analysis to assess our metrics. To assess our CADV approach, however, we conducted experiments with software developers and software engineering students.

5.1.1 Visualization goals

When we defined our suite of software metrics in Section 4.1, we used research questions to guide the work. However, for CADV, we will present in terms of *visualization goals*. The work from (FRANCESE et al., 2016; MERINO et al., 2018) reinforces that software visualizations approaches should make the goals clear at the very beginning.

The primary goal is to aid in the comprehension and provide an overview of how code annotations are distributed in the analyzed software system. The visualization can assist in comprehending responsibilities, size(large annotations), and coupling (classes coupled to several frameworks). Following the guidelines proposed on (FRANCESE et al., 2016), these are the primary goals of the visualization:

(#G1) - **Detect annotations schemas and how they are distributed in the packages**: At first (not a general rule), code annotations belonging to the same schema should be grouped in packages since they are usually related to a specific behavior or responsibility. For instance, annotations from the `javax.persistence` schema, are concerned with object-relational mapping. Therefore, one might expect that packages with these annotations have a well-defined responsibility. With our CADV approach, we want the user to spot all schemas and where they are being used quickly. Suppose code annotations from the same schema are present in several different packages. In that case, it may suggest that classes responsibilities are not very clear, which imposes challenges in evolving and maintaining the software. To reach this goal, we need a view to present the whole system and not overwhelm potential users with details closer to the source code. In short, we want to visualize packages and annotation schemas for the entire system.

(#G2) - **Detect how annotations are distributed per class in packages**: After a general view of the system is provided, the user might want to investigate specific packages to obtain more information about code annotations. For instance, (i) how are schemas distributed inside classes, (ii) are there classes coupled to several schemas, and (iii) is there a class with a large number of annotations. All these aspects help developers comprehend the responsibilities of existing classes and their profiles. We must also decide what metric to use when drawing annotations. They have several characteristics such as the number of arguments (AA), nesting level (ANL), Lines of Code (LOCAD), and others already measured by the proposed metrics described in Chapter 4. However, trying to display several metrics values in a two-dimension visualization can be troublesome and become a mess to interpret the data. Therefore we will only display one metric per view. To reach this goal, we need a view that can present more code annotations and classes inside packages. In short, we want to visualize a single package and annotations grouped in classes.

(#G3) - **Detect how annotations are distributed and grouped per code elements inside the classes**: Code annotations are placed on code elements, such as methods, members, and type definitions. Several annotations can configure the same code element. The user may be interested in observing how the annotations are grouped inside these code elements and detect if (i) a specific code element is overloaded with annotations, (ii) several code elements contain repeated annotations, (iii) a specific code element is coupled to several schemas, and so forth. We again face the challenge of choosing what annotation metric to use to generate the visualization. Our first approach is to use a different metric than the one used for reach Goal #G2 so that we can present different metrics but from a different perspective of the visualization. To reach this goal, we need a view that can present code elements and code annotations inside classes. In short, we want to visualize a single class and how code annotations are grouped in classes.

(#G4) - **Provide a navigation system between views with different granularity**: For each goal presented previously, #G1, #G2, and #G3, we are proposing a different view consistent with each of these three goals. Therefore, our visualization approach should provide some mechanism to allow the navigation between these different views.

(#G5) - **Detect misconfigurations**: As seen, our goals are concerned with the distribution of code annotations in the system rather than detect problems related to code annotations. However, if we can visualize code annotations, we might also detect potential problems. According to (YU et al., 2019), code annotations are deleted because they are redundant (15.4%) or are wrong (24.5%). For instance, if a developer spots that in the same class there are annotations for unit testing (`org.junit`) and persistence (`javax.persistence`), it may be considered for a refactoring or a closer look at the source code.

5.1.2 Research method

This section describes the steps performed to reach our visualization goals.

Step 1 - Propose the visualization

Based on the polymetric view concepts(LANZA; DUCASSE, 2003; FRANCESE et al., 2016), the guidelines presented on (MERINO et al., 2018), the GQM model(BASILI, 1992; BASILI et al., 1994) and the characteristics of annotations measured by our suite of metrics(LIMA et al., 2018) we propose our own polymetric view for code annotations metrics. We named our approach CADV (Code Annotations Distribution Visualization)

Step 2 - Create a tool to implement the CADV

To implement the CADV, we developed a tool called AVisualizer. This tool works as a front-end to the ASniffer(LIMA et al., 2020a). While the latter is a tool that scans Java software systems and extracts the metrics, the first can draw the CADV using these metrics values as input. Just as the ASniffer, the AVisualizer is open-source and available at `https://github.com/phillima/avisualizer`. A working demo that was used for the experiment can be accessed at `https://avisualizer.herokuapp.com/`

Step 3 - Design Evaluation Approach

To assess our visualization approach, which we named CADV, we performed two different experiments. The first one was an interview with developers that worked on the *SpaceWeatherTSI* module, part of the *SpaceWeather* web application developed for the EMBRACE division inside INPE. The second experiment was a survey with students. In both experiments, the participants used the AVisualizer tool and answered/discussed some questions about the *SpaceWeatherTSI* software. To gather opinions and impressions of the AVisualizer tool, in the second experiment, we used a questionnaire based on the TAM (Technology Acceptance Model) (DAVIS, 1989)

To ease the writing process, we will refer to the first experiment as E1 (interview with EMBRACE developers) and the second experiment as E2 (survey with the students).

Step 4 - Data Analysis

For experiment E1, we manually transcribed each recorded interview and performed the qualitative analysis. For each question, present in Table 5.3, we thoroughly discussed the interviewee's answers and commentaries to validate our novel CADV.

For experiment E2, we first analyzed the overall success rate of the close-ended questions to measure the student's performance. Then we discuss how these numbers and how the AVisualizer tool aided in comprehending the analyzed software. To verify the student's impressions and opinions, we analyzed the answers from the questions that measured how "easy" and "useful " the students found the AVisualizer tool. With these answers, we could determine the strong points and the required improvements.

Finally, we perform a discussion comparing E1 and E2 to understand how the CADV approach can reach its goals, i.e., allow the visualization of code annotations distribution and usage to aid the comprehension.

5.1.3 Target audience

Another crucial aspect of any software visualization approach is to clearly define the target audience, which should be consistent with the goals, as mentioned by Merino et al. (2018), Romano et al. (2019). Following, we describe our target audience in relevant order.

- Newcomer Developer[1]: According to DAGENAIS et al. (2010), whenever a developer approaches a new software system, it feels like explorers who must orient themselves within an unfamiliar landscape. Even for senior developers, this might happen. Even though there can be a mentor, newcomers still face challenges, and every resource available can be useful to ease the path. Furthermore, developers spend most of their time comprehending software (HASSELBRING et al., 2020). Our Goal 1 (#G1) deals directly with providing a general overview of the system that might aid newcomers.
- Student: Given that students are in a constant learning process, they are also potential users of our software visualization approach. They could use the visualization like newcomer developers and better understand some concepts related to code annotations and metadata-based frameworks usage. They can also identify aspects of the software system and further study them using other methods.
- Software Architect: Already have a firm grasp of the whole system. They analyze it focusing on aspects such as: how to make this system better

[1]We define a newcomer developer as someone who has just joined a team. It does not necessarily mean that it has no software experience. Even a senior software developer can be a newcomer

adhere to the proposed software architecture? Is the system growing out of control? Are packages responsibilities clearly defined? For this audience, the visualization can be used to investigate and potentially find: misplaced or misconfigured code annotations, redundant code annotations, several repeated annotations, outliers (like code annotations with huge numbers of arguments or large lines of code), packages overloaded with different annotation schemas and to have means to generate a general view of the system quickly

The target audience, however, does not need to adhere strictly to this definition. A newcomer can use the visualization to detect outliers, just as a seasoned developer can also gain new insight about the system software it did not know.

5.1.4 Experimental design

The work of (MERINO et al., 2018) mentions that several software visualizations work has failed in its evaluations, with lack of rigorous experiments. Therefore, they provide little evidence of the effectiveness of their approach. They mention that 62% of proposed approaches do not include any evaluation or a weak one, such as anecdotal evidence of simple usage scenarios. They conclude that software visualizations research should use more surveys with the target audience (to extract helpful information) and perform a thorough experiment using real-world open-source software systems and a controlled system with practical tasks.

To evaluate the CADV, we performed two experiments that we named E1 and E2. E1 was a recorded interview with six members involved in the development of the *SpaceWeatherTSI* software system. In this experiment, the interviewees used the AVisualizer tool to analyze the *SpaceWeatherTSI*. To guide our interview, we formulated 15 questions, present in Table 5.3. The conversation was very informal, and the participants could freely discuss the topics. They also provided valuable feedback on how we could improve our CADV approach and how to make the AVisualizer tool better. We also collected some demographics information about these participants. The interview was conducted with the Google Meet[2] web application, and we used the OBS[3] desktop application to record the interview.

The second experiment, abbreviated as E2, was conducted with students using a survey. In the experiment, the students had to answers ten objective questions about

[2]`https://meet.google.com/`
[3]`https://obsproject.com/`

the *SpaceWeatherTSI* software system using the AVisualizer tool. To assess their opinions and impressions about the tool, we used the TAM (Technology Acceptance Model) (DAVIS, 1989) as a basis. In this model, there are two variables that we are interested in. The first is the "Perceived Ease of Use", in other words, how users find the technology easy to use. The second variable is "Perceived Usefulness", which measures how users find the technology helpful. These two variables feed other parts of the model that will eventually output a "Actual System Usage". For our evaluation, we created 19 questions based on the two mentioned variables to guide us in understanding how easy and useful the students perceived our CADV (implemented through the AVisualizer). These questions used the Likert scale, ranging from *strongly disagree* to *strongly agree*, and were elaborated similar to the work of (CHOMA et al., 2019). The experiment was carried out through Google Forms in both English and Portuguese[4]. Finally, using students to carry out experiments is a viable option to advance software engineering, as shown by Falessi et al. (2018) and HÖst et al. (2000).

Participants Selections

The participants for E1 were personally invited to join the experiment. It was important to assess our CADV approach in a software system that was developed for INPE. As mentioned, we used the *SpaceWeatherTSI* web application as a target system. Hence, we invited the members that were involved in the construction of this project. Given that they know the system, they could assess if the AVisualizer tool could display a coherent general view of the project under analysis. They could also analyze if the AVisualizer brought new insights and valuable information for a system they were already familiar with. They are all familiar with Java and code annotations and provided valuable feedback to our wok. In short, one of the goals of E1 was to also validate this work as applied research for INPE internal solution.

As for our second experiment, E2, we invited students from INATEL[5] (Instituto Nacional de Telecomunicações), UniBz[6] (University of Bolzano) and IPT[7] (Instituto de Pesquisas Tecnológicas). The majority of the students were undergraduates in Computer Science/Engineering courses, and only six were master students. We managed to obtain 36 students to participate voluntarily in our study. None of these students

[4]Experiment in Portuguese `https://forms.gle/rFEoNHaBuW8RKVaf7` and English `https://forms.gle/zK1UddR2mQMLQ7zp7`

[5]`https://inatel.br/home/`

[6]`https://www.unibz.it/`

[7]`https://www.ipt.br/`

were involved in the development of the *SpaceWeatherTSI* software and provided us a complementary perspective alongside E1.

Procedure

To conduct the E1 experiment, we first provided a link with a recorded video as a tutorial on how to use the AVIsualizer tool. We sent it via e-mail to each participant 1 week before the interview. We also sent a Google Form link to collect some personal data and their consent to participate in the experiment. Following is a list of the contents in the form [8].

a) Clarified Consent Term: We explained the experiment they were invited to, the goals and requested their consent to participate. The developers could choose not to participate

b) Personal Information: We asked what their role was during the development of the *SpaceWeatherTSI* software and their familiarity with code annotations.

The interview occurred according to the following list:

a) Video call: Using Google Meet, we initiated the call.

b) AVisualizer Demo: We provided the link of the deployed demo of the AVisualizer tool to the interviewee

c) Recording: Initiated the recording using the OBS desktop application.

d) Training Session: The interviewer shared the screen with the interviewee and provided another 15-minute training. Therefore, the participants were submitted to two training sessions of the AVisualizer tool. One was a video link previously sent. The second was live training right before the interview properly began.

e) AVisualizer Execution: The interviewees could use the tool themselves (on their computer), but they preferred to have the interviewer manipulate it for them (in the interviewer's computer, with the screen shared). The interviewees were issuing all the commands while answering and discussing the questions. The interviewer merely clicked on the desired location. In

[8]Only available in Portuguese: `https://forms.gle/RYn4XEK2Dnyy61aH7`

parts of the interview, the interviewees also played and experimented with the tool on their computer.

f) Questions: The interviewer started asking questions presented in Table 5.3 to guide the interview. However, the conversation was informal so the interviewees could freely answer and discuss these questions/topics.

On average, the interviews lasted 60 minutes in total, i.e., 15 minutes of training with 45 minutes of questions/discussions.

For E2, the experiment was carried through a survey using Google Forms. The only external link they had to access was the AVisualizer tool. Following is a list of events of how the experiment unfolded as soon as the students accessed the form.

a) Clarified Consent Term: We explained the experiment they were invited to, the goals and requested their consent to participate. The students could choose not to be involved.

b) Personal Information: We gathered demographic information, experience with code annotations, current role, and primary programming language

c) Code Annotations: We briefly explained code annotations and our novel metrics (defined in Chapter 4 and published in (LIMA et al., 2018)).

d) AVisualizer Tutorial: We provided a textual explanation of the AVisualizer tool and a video (the link was also provided if the participant preferred to watch directly on YouTube. This was the same link sent to participants of E1), which was the training section. The participants were free to watch the video during the whole experiment.

e) Using the AVisualizer: We provided the link where the demo of the AVisualizer is deployed and presented ten close-ended questions about the *SpaceWeatherTSI* system. The demo version of the AVisualizer tool is already generating the visualization for this system.

f) Impressions of the AVisualizer: In this final section, we provided 19 questions using the Likert Scale to measure the *usefulness* and *ease of use* based on the TAM model. Also, we left a final last question to students to describe their overall opinion and impressions.

In both experiments, E1 and E2, we used different software for the training session to not generate any bias and potentially compromise our results. The chosen system was the Guj[9] web application. The code is publicly available at `https://github.com/caelum/guj.com.br`. This is a well-known Brazilian Q&A forum for software development, and the source code has several different annotation schemas and was suited for the training.

5.2 Code Annotations Distribution Visualization - CADV

This section presents the definition of our visualization approach, which we named CADV - Code Annotations Distribution Visualization. As previously mentioned, our approach is composed of three different polymetric views enriched with code annotation metrics values. The metrics was proposed and discussed in Chapter 4.

Even though we are focused on displaying code annotations, they are part of the source code. So our approach needs to display classes and packages since these two will contain the code annotations. The source code is a hierarchical structure where packages contain classes that contain code elements that contain code annotations. Therefore, we chose a circle pack approach to serve as a basis for CADV.

We had previous experience using a combination of squares and circles to draw the visualization. We felt this approach was problematic to represent the hierarchy as well as scale with larger software systems. It made it challenging to observe the code annotations distribution. However, we were able to detect the shortcomings of this approach and published these results in (LIMA et al., 2020b).

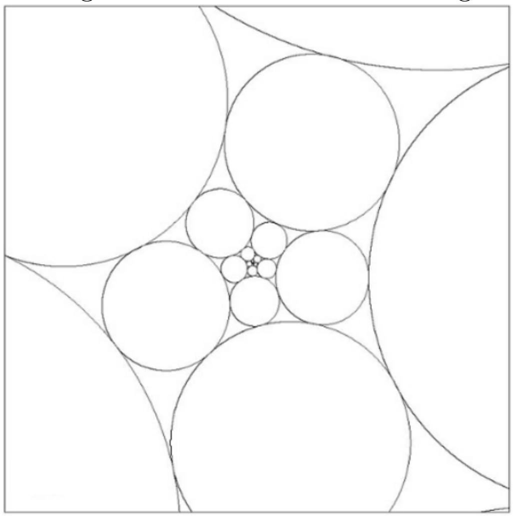
For the CADV, we display everything as a circle. In other words, packages are circles, and classes are circles. Code elements and code annotations are also circles. We use strategies like a different color, outline, and some elements that are only visible in specific views to differentiate them. The radius of the leaf circles will be calculated based on values of code annotation metrics that we discussed in Chapter 4.

A circle packing, in its simplest form, is a collection of circles in the plane or 2-sphere such that each gap between circles is triangular, abutting three pair-wise tangent circles (STEPHENSON et al., 2007). The area of each leaf circle in a circle-packing diagram is proportional to its value (here, code annotation metrics value). Although nested circles do not use space as efficiently as a treemap, the “wasted” space better reveals the hierarchical structure (BOSTOCK, 2017). Since we wanted to provide a

[9] `https://www.guj.com.br/`

visualization approach that displayed the hierarchical structure of the source code, the circle packing is well suited.

Figure 5.1 - Basic Circle Packing



Source: Stephenson et al. (2007)

When defining the metrics in Chapter 4, we first proposed four research questions, and then we designed a GQM model to guide the definition of the metrics. For the CADV approach, we first proposed five goals (Section 5.1.1). From them, we will design a GQM model to guide the definition of the polymetric views that compose our visualization approach.

As seen from our goals, we want to provide a visualization that can capture and display different characteristics of the source code. We also require different granularity levels. One single view would not be enough to reach these goals. We need, therefore, different views and also a navigation system between them.

From the five goals proposed on 5.1.1 we obtained four questions for our GQM model. With these, we were able to propose three polymetric views that, combined, form our Code Annotations Distribution Visualization - CADV. Table 5.1 presents the GQM model.

We deliberately left #G4 - **Provide a navigation system between views with different granularity** - out of the GQM model since it is a goal concerned with the navigation system. This has to be dealt with during the development of the tool that implements the CADV.

Table 5.1 - GQM applied for the CADV Approach.

Goal	(Purpose) (Issue) (Object) (Viewpoint)	*Visualize* *the usage and distribution of* *annotated code* *from software developer viewpoint*
(Question)	Q1	**How are annotations schemas distributed by packages? (Extracted from G1)**
(View 1)	**System View**	Provides a polymetric view that displays annotation schemas being used by packages
(Question)	Q2	**How are annotations schemas distributed inside packages? (Extracted from G2)**
(View 2)	**Package View**	Provides a polymetric view that displays annotations being used in classes of a package
(Question)	Q3	**How are annotations schemas distributed inside classes? (Extracted from G3)**
(View 3)	**Class View**	Provides a polymetric view that displays annotations, and how they are grouped, inside a class.
(Question)	Q4	**How to detect potential misplaced code annotations? (Extracted from G5)**
(View 1)	**System View**	–
(View 2)	**Package View**	–
(View 3)	**Class View**	–

Notice from #G5 -**Detect misconfigurations**- that every view can aid in detecting misconfigurations or misplacement. Each one will contribute differently and within their scope. For instance, to detect an extensive annotation (considering high LOCAD), the Package View seems suited, while the System View does not help much. On the other hand, an annotation schema potentially misplaced might be quickly spotted in the System View. The Class View may contribute to detecting a specific code element overloaded with code annotations. As previously said, we are not focusing on detecting these potential issues, so the views were not designed considering these aspects. However, during experiments, we can still explore how the views handle such topics.

In the upcoming subsections, we will define the three views that compose the CADV. Since this is a software visualization approach, we will heavily use figures to ease comprehension. These images are being generated by the AVisualizer tool with the *SpaceWeatherTSI* project under analysis. The CADV and the AVisualizer are firmly related, and we believe it would be unnecessary to discuss the CADV as a completely separate concept.

Before we discuss the views themselves, we first present the basic layout provided by the AVisualizer. Just as done by other software visualization authors, such as (LANZA; DUCASSE, 2003), to better comprehend this section and our CADV approach, we recommend reading on a colored screen monitor. The tool is heavily based on colors.

The demonstration of the AVisualizer is available at `https://avisualizer.herokuapp.com/`

5.2.1 AVisualizer layout

As soon as the AVisualizer tool is executed, the user is presented with three areas, as shown in Figure 5.2.

These three areas are The Header, The View, and the Schema Table. Following, we detail each of these areas.

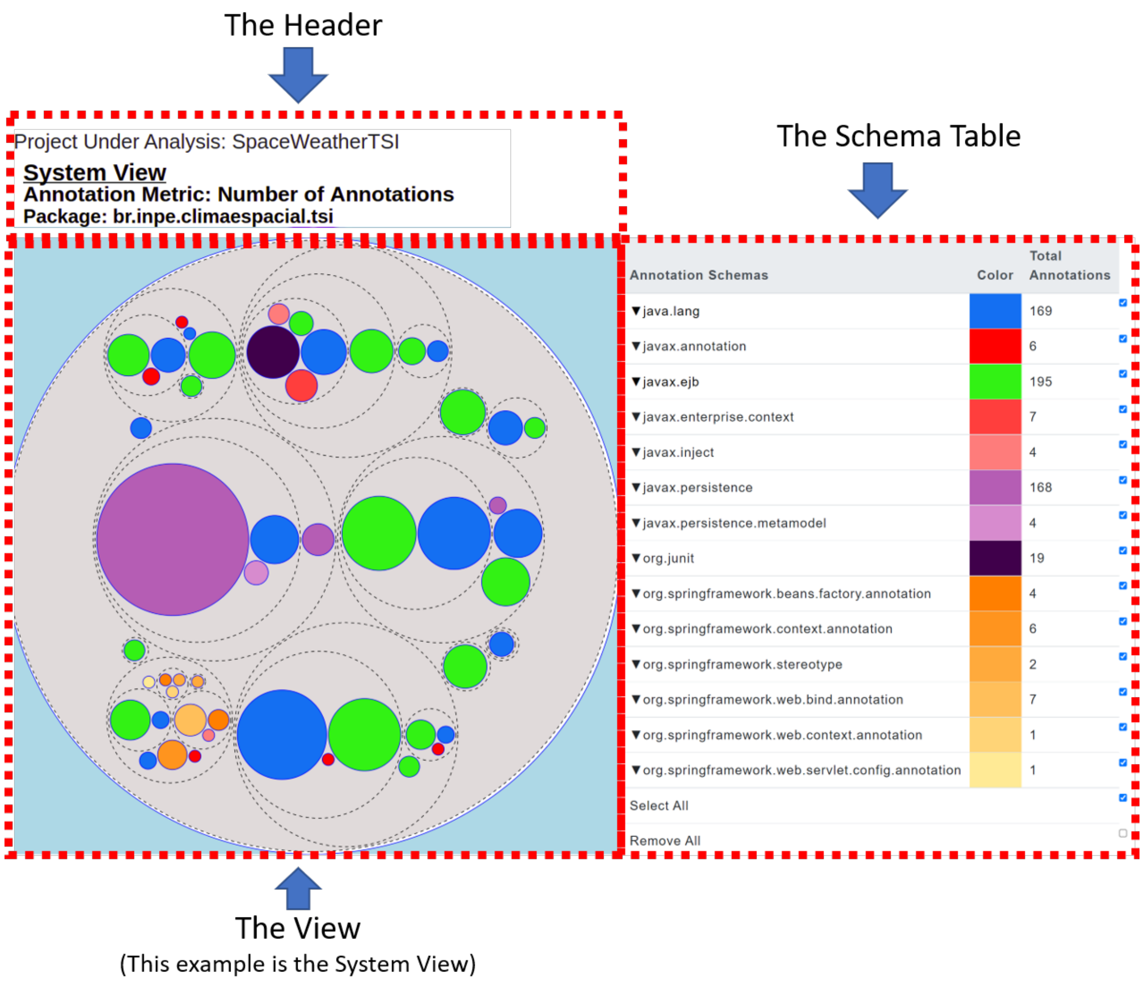
The **Header** contains four parts, described below:

- **Project Under Analysis**: It displays the name of the software being analyzed. For our experiments, we used the *SpaceWeatherTSI* software.
- **View**: Since the CADV is composed of three views, the user must be aware of which one is currently rendered. The three possible options are: System View, Package View, and Class View
- **Annotation Metric**: We need to inform the user what metric is being used to generate the size of the leaf circles. These colored ones 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.

- **Class or Package**: Informs the user what package or class it is currently inspecting. For instance, the package being inspected on Figure 5.2 is the `br.inpe.climaespacial.tsi`, which is the root package of the *SpaceWeatherTSI* software.

Figure 5.2 - Annotation Visualizer.



Project Under Analysis: SpaceWeatherTSI

System View

Annotation Metric: Number of Annotations

Package: br.inpe.climaespacial.tsi

Annotation Schemas	Color	Total Annotations	
▼java.lang	🟦	169	✓
▼javax.annotation	🟥	6	✓
▼javax.ejb	🟩	195	✓
▼javax.enterprise.context	🟥	7	✓
▼javax.inject	🟥	4	✓
▼javax.persistence	🟪	168	✓
▼javax.persistence.metamodel	🟪	4	✓
▼org.junit	🟪	19	✓
▼org.springframework.beans.factory.annotation	🟧	4	✓
▼org.springframework.context.annotation	🟧	6	✓
▼org.springframework.stereotype	🟧	2	✓
▼org.springframework.web.bind.annotation	🟨	7	✓
▼org.springframework.web.context.annotation	🟨	1	✓
▼org.springframework.web.servlet.config.annotation	🟨	1	✓
Select All			✓
Remove All			☐

The **Schema Table**, contains four parts, described below:

- Annotation Schemas: A list of every annotation schema found in the project under analysis.
- Color: Display what color was assigned to a given annotation schema. For instance, Figure 5.2 shows that a pink tone was assigned to the

`javax.persistence` schema.

- Total Annotations: The total number of annotations that belong to a given schema being used in the whole project. It includes repeated annotations. For instance, Figure 5.2 shows that the project *SpaceWeatherTSI* uses 169 annotations from the `javax.persistence`. It could even be 169 identical annotations, for example, the `@Entity` annotation.
- Check Box: The user can filter out some annotations schemas to better explore the project using the check box on the final column.

The **View** is the area that displays the actual polymetric view. There are three different views: The System View, The Package View, and the Class View. For instance, Figure 5.2 is displaying the System View for the *SpaceWeatherTSI*

The **View** is the constantly changing and modifying to display one of the proposed view. 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.

5.2.2 System View

The ***System View*** is the default view displayed to the user. It presents the whole project in a single view, allowing users to grasp the project under analysis quickly. The System View displays only packages and annotation schemas, and both are rendered as circles. We chose a light gray background color to remain as neutral as possible. The following list presents the characteristics of these circles:

- Packages: Every circle that represents 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 propor-

tional 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.

To further clarify the ***System View***, consider the following hypothetical project "Example" with four classes and two packages. Figures 5.3 - 5.4 presents the source code of these classes.

Figure 5.3 - Example package "example.pk1" with classes: Class1 and Class2.

```java
package example.pk1;

import javax.schema1.Annotation1;
import javax.schema2.Annotation2;
import javax.schema2.Annotation3;

public class Class1 {

    @Annotation1 // belongs to javax.schema1
    @Annotation2 // belongs to javax.schema2
    @Annotation3 // belongs to javax.schema2
    private int member1;
}
//Separate File
package example.pk1;
import javax.schema2.Annotation4;
import javax.schema2.Annotation5;

public class Class2 {

  @Annotation4 //belongs to javax.schema2
  @Annotation5 //belongs to javax.schema2
  private int member2;
}
```

Figure 5.5 presents the ***System View*** for this project. To make it clearer, we removed the caption for the ***Header***, ***View*** and ***Schema Table***, but kept the red-dotted frame to highlight each area. We added some arrows and numbers on the figure, that we will use to guide our explanation.

Analyzing the ***Schema Table*** on Figure 5.5, we have three schemas: `javax.schema1`, `javax.schema2`, and `javax.schema3`. In the source code they are the imports. Since this is a hypothetical example, we assigned random colors for the schemas (with the exception of gray and white). In the ***View*** area, which is displaying the ***System View***, we have three dashed outlined circles with gray background, that represents the three packages in the project: `example`, `example.pk1`, and `example.pk2`. From Figures 5.3 - 5.4 we know this project has four classes, but we cannot see them in the ***System View***.

Figure 5.4 - Example package “example.pk2” with classes: Class3 and Class4.

```java
package example.pk2;

import javax.schema2.Annotation6;
import javax.schema2.Annotation7;

public class Class3 {

  @Annotation6
  @Annotation7
  private int member3;
}
//Separate File
package example.pk2;
import javax.schema3.Annotation8;

public class Class4 {

  @Annotation8
  private int member4;

}
```

The circle marked with (1) is slightly larger than the circle marked with (2), meaning the package represented by the circle (1) has classes with more code annotations than the package represented by the circle (2). Inspecting the Figure, we cannot tell which package these circles represent, but hovering the mouse on top will reveal a label with that information, which is package *example.pk1*. By elimination, circle (2) represents package *example.pk2*. We deliberately chose this approach to avoid polluting the visualization with textual information. By looking at the visualization, we can quickly extract the following information:

- Circle (1) has code annotations from two schemas -`javax.schema1` and `javax.schema2`- and has more code annotations than the package represented by the circle (2).
- Circle (2) has code annotations from two schemas, i.e., `javax.schema2` and `javax.schema3`.
- The schema `javax.schema2` is present in both packages represented by the

circle (1) and circle (2). We highlight this by drawing two arrows marked with (3) on Figure 5.5. Notice that the two circles pointed by the arrows share the same color, which means they represent the same schema. Furthermore, the circle on the left is larger than the one on the right. It means the package represented by the circle (1) has more code annotations from *javax.schema2* than the package represented by the circle (2).

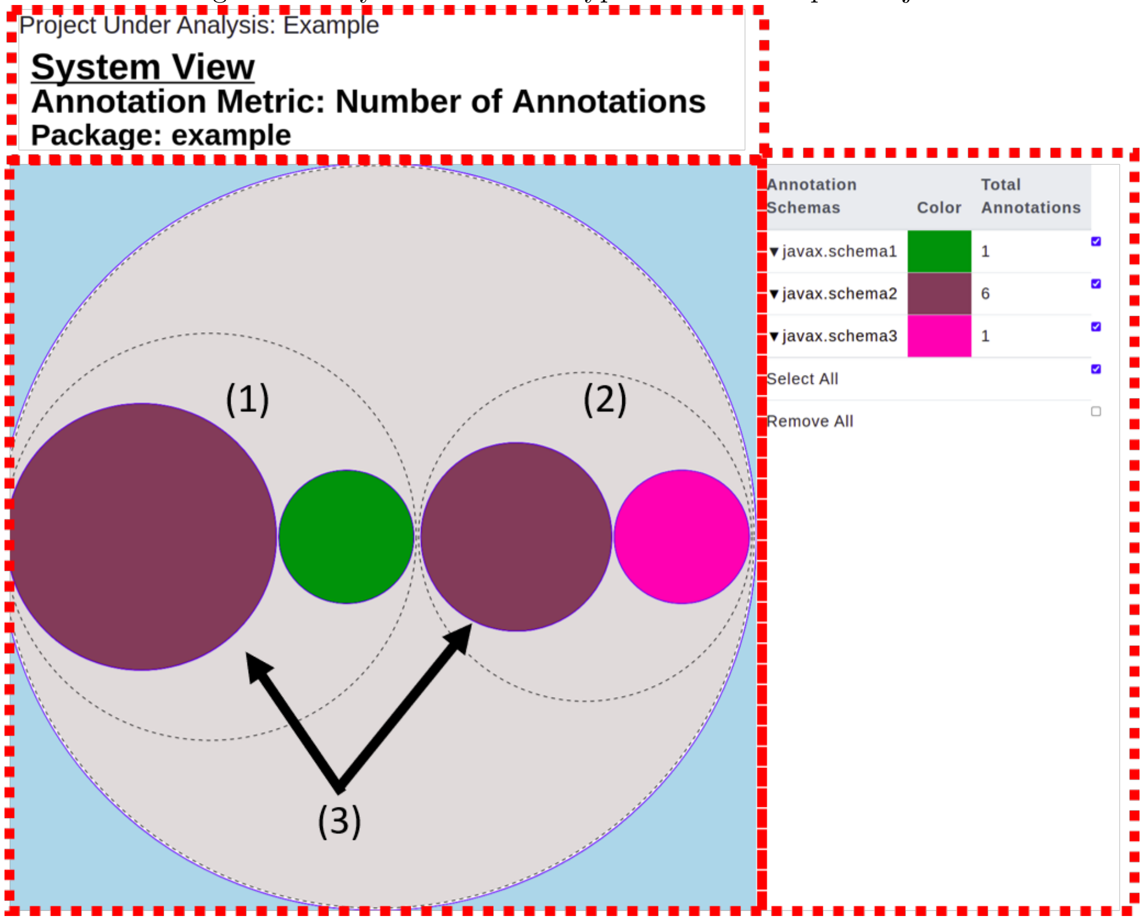
- Circle (1) represents a package with more code annotations from schema `javax.schema2` than schema `javax.schema1`. We can see the "green" circle is smaller.
- Circle (2) represents a package with more code annotations from schema `javax.schema1` than schema `javax.schema3`. In other words, the "pink" circle is smaller.
- Circle (1) represents a package without code annotations from schema `javax.schema3`. In other words, there are no "pink" circles.]
- Circle (2) represents a package without code annotations from schema `javax.schema1`. In other words, there are no "green" circles.

The previous information was mainly obtained visually, with the support from the ***Schema Table***. To obtain further information, the user can hover the mouse over the circles. Labels will reveal more information such as package name, schema name, and the number of code annotations occurrences of that schema. We believe this approach is interesting because it leaves the visualization clean and draws attention to the size and color of the circles. In other words, it draws attention to code annotations and schemas.

To confirm this, let us manually inspect the source code. Figure 5.3 represent two classes from package *example.pk1*. Class `Class1` has three annotations and class `Class2` has two annotations, i.e., a total of five annotations. Figure 5.4 show two classes from package *example.pk2*. Class `Class3` has two annotations and class `Class4` has one annotation, a total of three. In other other words, package `example.pk1` has more annotations than package *example.pk2*. It is now clear why, in Figure 5.5, the circle (1) is larger than circle (2). We reinforce that the visualization had already revealed this information much quicker, and without having to manually inspect the source.

We can also zoom in on packages. For instance, let us zoom on the circle marked with (1). To do this we must simply perform a click action on the desired circle representing the package. Figure 5.6 shows the result.

Figure 5.5 - System View for Hypothetical Example Project.


Project Under Analysis: Example

System View

Annotation Metric: Number of Annotations

Package: example

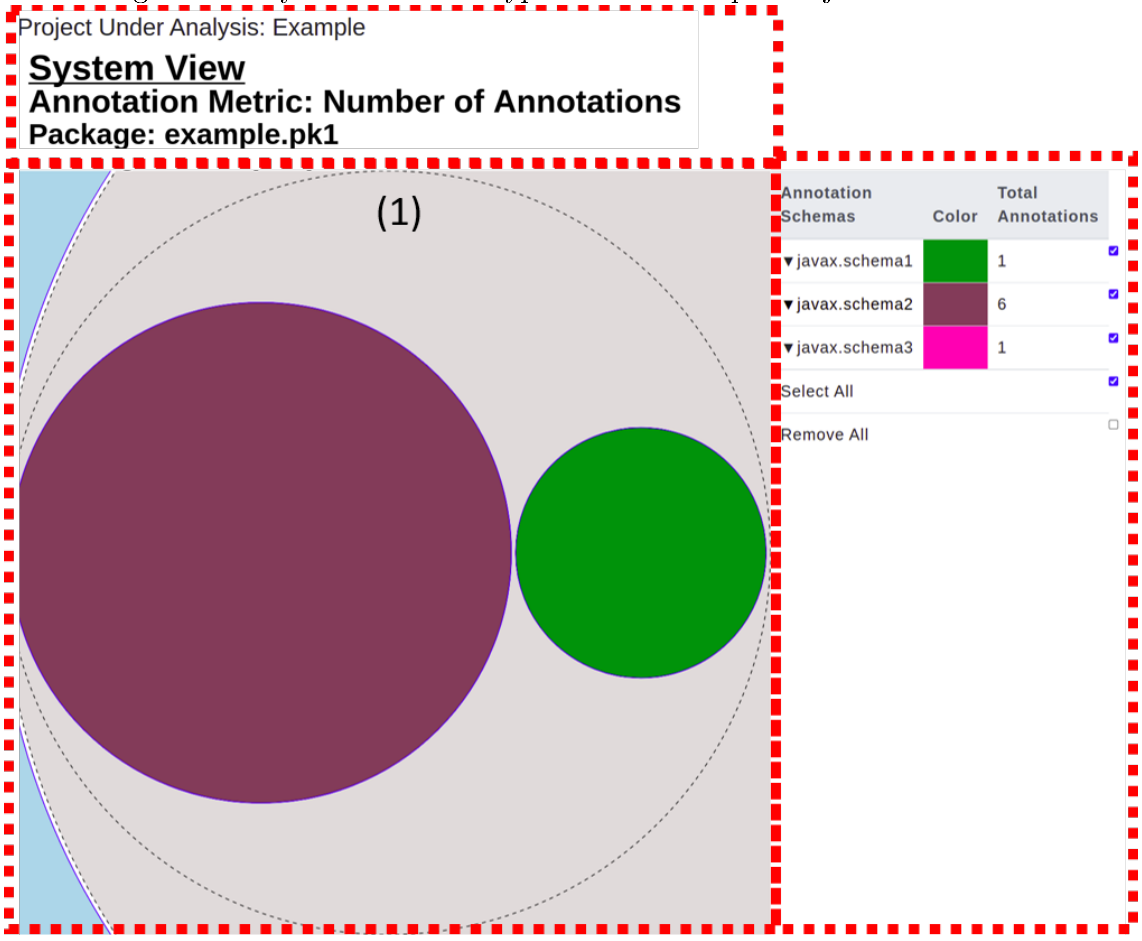
Annotation Schemas	Color	Total Annotations	
▾javax.schema1	🟩	1	☑
▾javax.schema2	🟥	6	☑
▾javax.schema3	🟪	1	☑
Select All			☑
Remove All			☐

The ***Header*** was updated to inform that we are now in package *example.pk1* but still using the ***System View***. As known, the ***Schema Table*** remains fixed since we want to keep a reference to the whole system still, even if the user is inspecting a small part.

To further clarify, there are two possible outcomes when we perform a click action.

- If we click on a circle that does not represent a schema or individual code annotation, a zoom happens.
- If we click on a circle representing a schema or individual annotation, we change the type of view being rendered. These circles are, always, the colored ones, except for the colors gray and white. The change of views happens in the following order ***System View*** to ***Package View*** to ***Class View***.

Figure 5.6 - System View for Hypothetical Example Project With Zoom.



Project Under Analysis: Example

System View

Annotation Metric: Number of Annotations

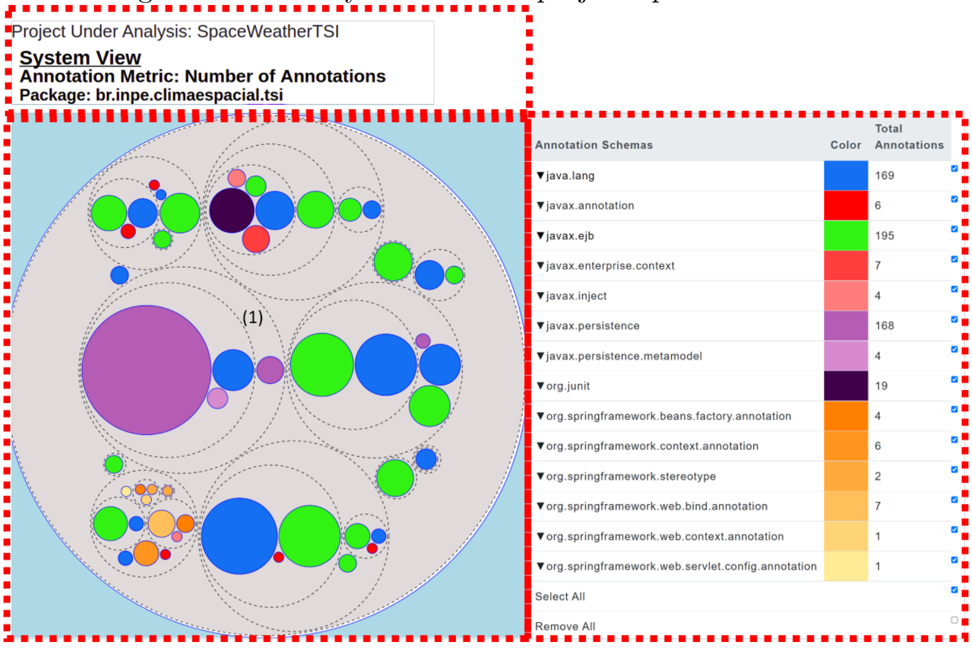
Package: example.pk1

Annotation Schemas	Color	Total Annotations	
▾javax.schema1	🟩	1	✓
▾javax.schema2	🟪	6	✓
▾javax.schema3	🟪	1	✓
Select All			✓
Remove All			☐

From the presented description, there is no class information in the ***System View***, which is what we wanted to achieve in Goal 1 (**Detect annotations schemas and how they are distributed in the packages**). We only wish to see schemas and packages. At this point, we are not interested in visualizing how annotations are configured inside classes. Therefore, we do not display characteristics of individual annotations, only their schema (or annotation schema). These last two pieces of information are left to other views to handle, i.e., display classes and individual annotations.

To finish presenting the ***System View*** let us now return to the real-world software *SpaceWeatherTSI* presented on Figure 5.7. This Figure is displaying the ***System View*** for the *SpaceWeatherTSI* software. We marked a package with (1), to zoom in.

Figure 5.7 - The System View of project SpaceWeatherTSI.


Project Under Analysis: SpaceWeatherTSI

System View

Annotation Metric: Number of Annotations

Package: br.inpe.climaespacial.tsi

Annotation Schemas	Color	Total Annotations	
▼java.lang	🟦	169	✓
▼javax.annotation	🟥	6	✓
▼javax.ejb	🟩	195	✓
▼javax.enterprise.context	🟥	7	✓
▼javax.inject	🟥	4	✓
▼javax.persistence	🟪	168	✓
▼javax.persistence.metamodel	🟪	4	✓
▼org.junit	🟪	19	✓
▼org.springframework.beans.factory.annotation	🟧	4	✓
▼org.springframework.context.annotation	🟧	6	✓
▼org.springframework.stereotype	🟧	2	✓
▼org.springframework.web.bind.annotation	🟧	7	✓
▼org.springframework.web.context.annotation	🟨	1	✓
▼org.springframework.web.servlet.config.annotation	🟨	1	✓
Select All			✓
Remove All			☐

Figure 5.8 displays the *System View* but with a zoom on package `br.inpe.climaespacial.tsi.entity` of project *SpaceWeatherTSI*. The ***Schema Table*** remains unchanged. The only change in the ***Header*** is the name of the package, which was updated to reflect the package represented by the outermost dashed-line circle. From Figure 5.8 we can see this package has annotations from the `javax.persistence` schema (pink circle) and also it has one inner package (inner circle with dashed-line). This inner package has annotations from the `javax.persistence` (large pink circle), `javax.persistence.metamodel` (small, slightly lighter pink circle) and `java.lang` (blue circle) schemas.

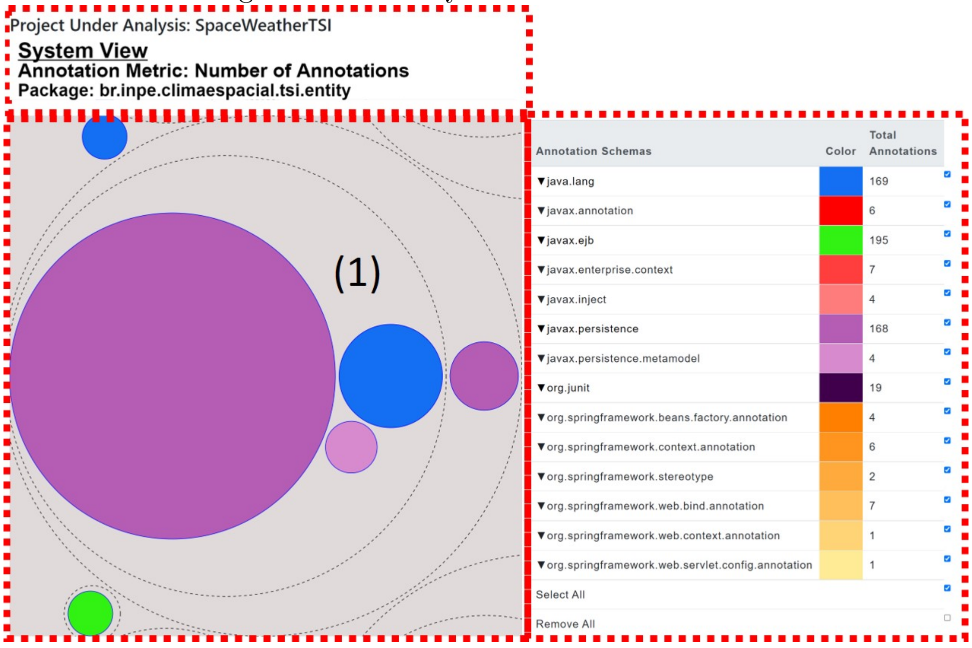
We can further zoom in on this package. Figure 5.9 displays this zoom. Notice that the Header updates to reflect the outermost dashed-line circle. In other words, the package zoomed in.

To further inspect this package and annotations, we need to switch to the following the ***Package View***.

5.2.3 Package View

The **Package View** can display classes and individual code annotations inside a given observed package. Differently from the ***System View*** designed to visualize the whole system, in the ***Package View*** we are interested in a specific package.

Figure 5.8 - The System View with a zoom.



Project Under Analysis: SpaceWeatherTSI

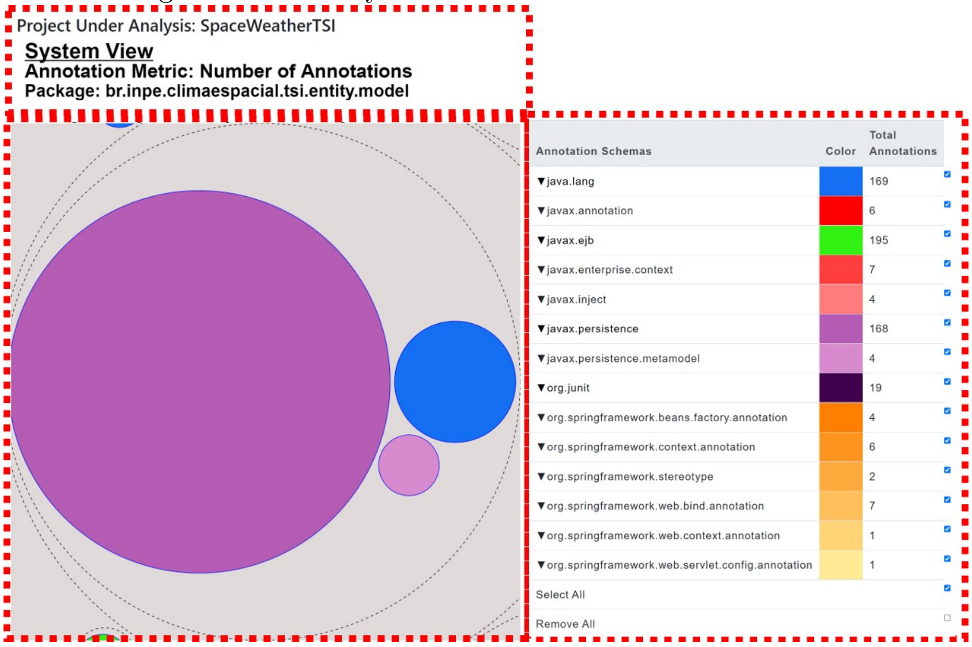
System View

Annotation Metric: Number of Annotations

Package: br.inpe.climaespacial.tsi.entity

Annotation Schemas	Color	Total Annotations	
▼java.lang	🟦	169	✓
▼javax.annotation	🟥	6	✓
▼javax.ejb	🟩	195	✓
▼javax.enterprise.context	🟥	7	✓
▼javax.inject	🟥	4	✓
▼javax.persistence	🟪	168	✓
▼javax.persistence.metamodel	🟪	4	✓
▼org.junit	🟪	19	✓
▼org.springframework.beans.factory.annotation	🟧	4	✓
▼org.springframework.context.annotation	🟧	6	✓
▼org.springframework.stereotype	🟧	2	✓
▼org.springframework.web.bind.annotation	🟧	7	✓
▼org.springframework.web.context.annotation	🟨	1	✓
▼org.springframework.web.servlet.config.annotation	🟨	1	✓
Select All			✓
Remove All			☐

Figure 5.9 - The System View with a second level zoom.



Project Under Analysis: SpaceWeatherTSI

System View

Annotation Metric: Number of Annotations

Package: br.inpe.climaespacial.tsi.entity.model

Annotation Schemas	Color	Total Annotations	
▼java.lang	🟦	169	✓
▼javax.annotation	🟥	6	✓
▼javax.ejb	🟩	195	✓
▼javax.enterprise.context	🟥	7	✓
▼javax.inject	🟥	4	✓
▼javax.persistence	🟪	168	✓
▼javax.persistence.metamodel	🟪	4	✓
▼org.junit	🟪	19	✓
▼org.springframework.beans.factory.annotation	🟧	4	✓
▼org.springframework.context.annotation	🟧	6	✓
▼org.springframework.stereotype	🟧	2	✓
▼org.springframework.web.bind.annotation	🟧	7	✓
▼org.springframework.web.context.annotation	🟨	1	✓
▼org.springframework.web.servlet.config.annotation	🟨	1	✓
Select All			✓
Remove All			☐

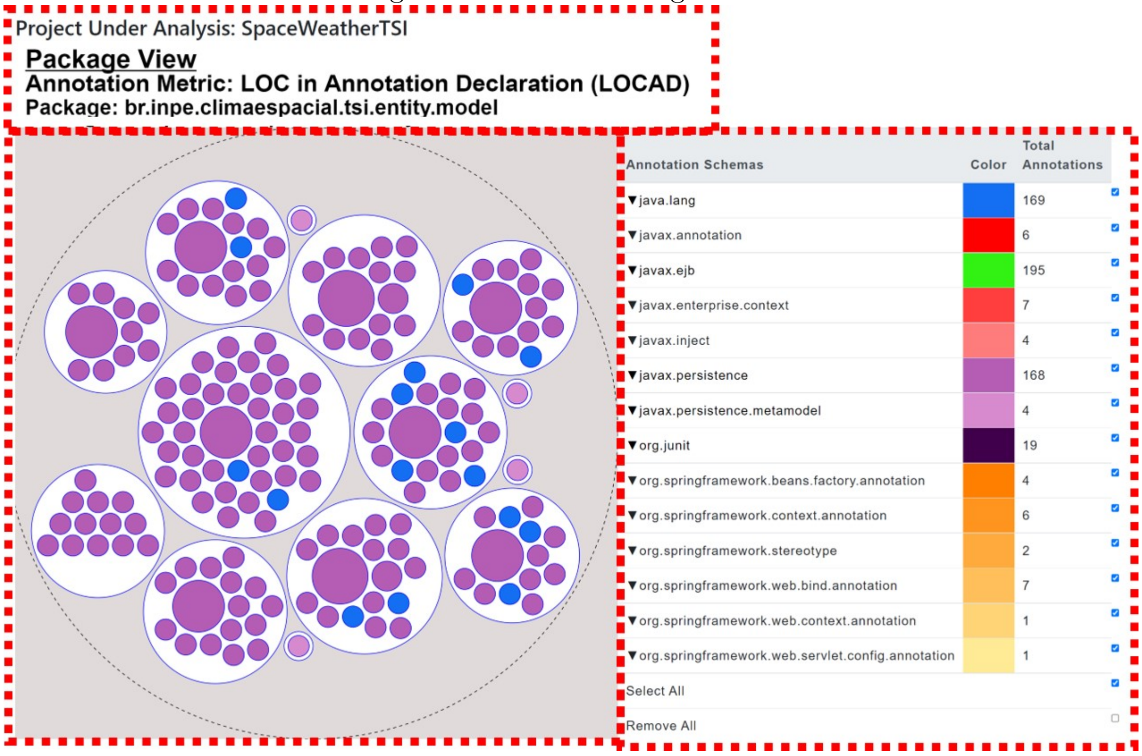
The circles are rendered with the following characteristics.

- 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 the annotations. 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 being 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.

Figure 5.10 - The Package View.



Project Under Analysis: SpaceWeatherTSI

Package View

Annotation Metric: LOC in Annotation Declaration (LOCAD)

Package: br.inpe.climaespacial.tsi.entity.model

Annotation Schemas	Color	Total Annotations	
▼java.lang	🟦	169	✓
▼javax.annotation	🟥	6	✓
▼javax.ejb	🟩	195	✓
▼javax.enterprise.context	🟥	7	✓
▼javax.inject	🟥	4	✓
▼javax.persistence	🟪	168	✓
▼javax.persistence.metamodel	🟪	4	✓
▼org.junit	🟪	19	✓
▼org.springframework.beans.factory.annotation	🟧	4	✓
▼org.springframework.context.annotation	🟧	6	✓
▼org.springframework.stereotype	🟧	2	✓
▼org.springframework.web.bind.annotation	🟨	7	✓
▼org.springframework.web.context.annotation	🟨	1	✓
▼org.springframework.web.servlet.config.annotation	🟨	1	✓
Select All			✓
Remove All			☐

To exemplify the ***Package View***, we will use the package `br.inpe.climaespacial.tsi.entity.model`. We already displayed this package on Figure 5.9, but in the ***System View***. Figure 5.10 displays this same package but now in the ***Package View***. To access the ***Package View*** simply click on any schema (colored circle) in the desired package. To revisi the result of a click action:

- If we click on a circle that does not represent a schema or individual code annotation, a zoom happens.
- If we click on a circle representing a schema or individual annotation, we change the type of view being rendered. These circles are, always, the colored ones, except for the colors gray and white.

Since Figure 5.9 is displaying a ***System View***, after the click action we will transition to the ***Package View***

From Figure 5.10 we see the ***Schema Table*** is exactly the same. The ***Header*** has changed to inform that we are now in the ***Package View*** and also informs the new metric being used to draw individual annotations, i.e, LOCAD - Lines of Code in Annotation Declaration. Revisiting this metric, presented in Chapter 4, it measures the number of lines of code used to write the code annotation. Consider the code on Figure 5.11. The annotation `@Id` (line 3) has LOCAD = 1, while the annotation `@GeneratedValue` (line 4) has LOCAD = 3.

Figure 5.11 - Example Class to revisit LOCAD.

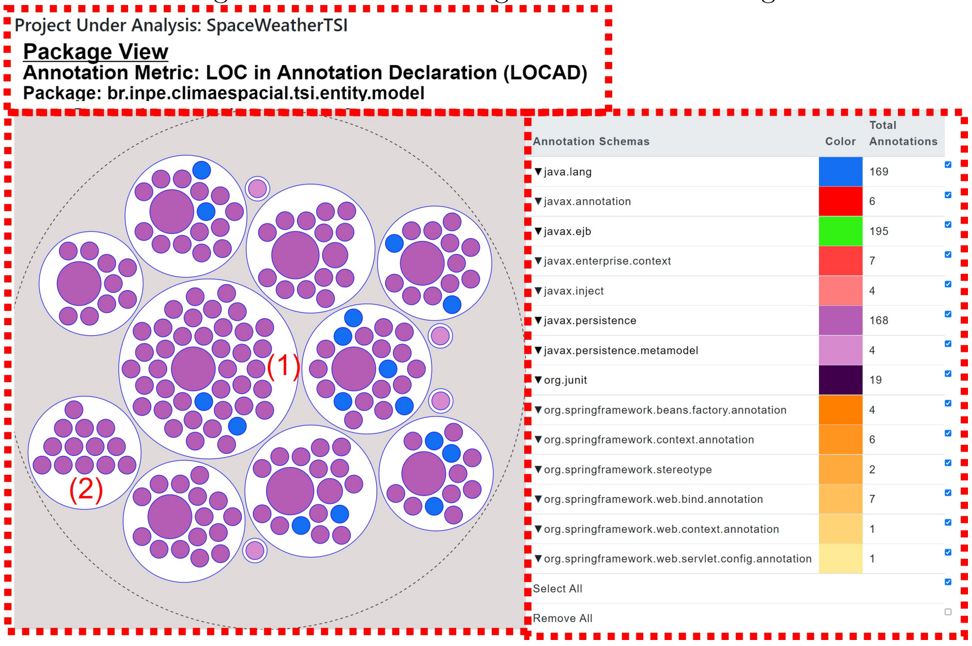
```
1  public class Player {
2
3      @Id
4      @GeneratedValue(
5          strategy = GenerationType.IDENTITY
6      )
7      private int id;
8  }
```

We could also switch the LOCAD metric for another that belongs to our suite defined in Chapter 4, such as AA(Arguments in Annotations) or ANL (Annotation Nesting Level). The CADV was designed to display only one code annotation metric value in the same view. We chose to display fewer metrics values as possible to avoid any confusion and end with a visualization that overwhelms the user. The reason we used the LOCAD metric in the ***Package View*** is that the traditional LOC metric has been traditionally used to represent the size, and it is well-known to developers.

Just as with the ***System View*** we do not have textual information being displayed to create a clean visualization. However, the user can hover the mouse over so that labels are revealed with more information.

To further inspect the ***Package View***, consider the Figure 5.12. It shows the same package presented on Figure 5.10, but with we marked two white circles (classes) with the numbers (1) and (2) for further explanation.

Figure 5.12 - The Package View with markings.



Project Under Analysis: SpaceWeatherTSI

Package View

Annotation Metric: LOC in Annotation Declaration (LOCAD)

Package: br.inpe.climaespacial.tsi.entity.model

Annotation Schemas	Color	Total Annotations	
▼java.lang	🟦	169	✓
▼javax.annotation	🟥	6	✓
▼javax.ejb	🟩	195	✓
▼javax.enterprise.context	🟥	7	✓
▼javax.inject	🟥	4	✓
▼javax.persistence	🟪	168	✓
▼javax.persistence.metamodel	🟪	4	✓
▼org.junit	🟪	19	✓
▼org.springframework.beans.factory.annotation	🟧	4	✓
▼org.springframework.context.annotation	🟧	6	✓
▼org.springframework.stereotype	🟧	2	✓
▼org.springframework.web.bind.annotation	🟧	7	✓
▼org.springframework.web.context.annotation	🟨	1	✓
▼org.springframework.web.servlet.config.annotation	🟨	1	✓
Select All			✓
Remove All			☐

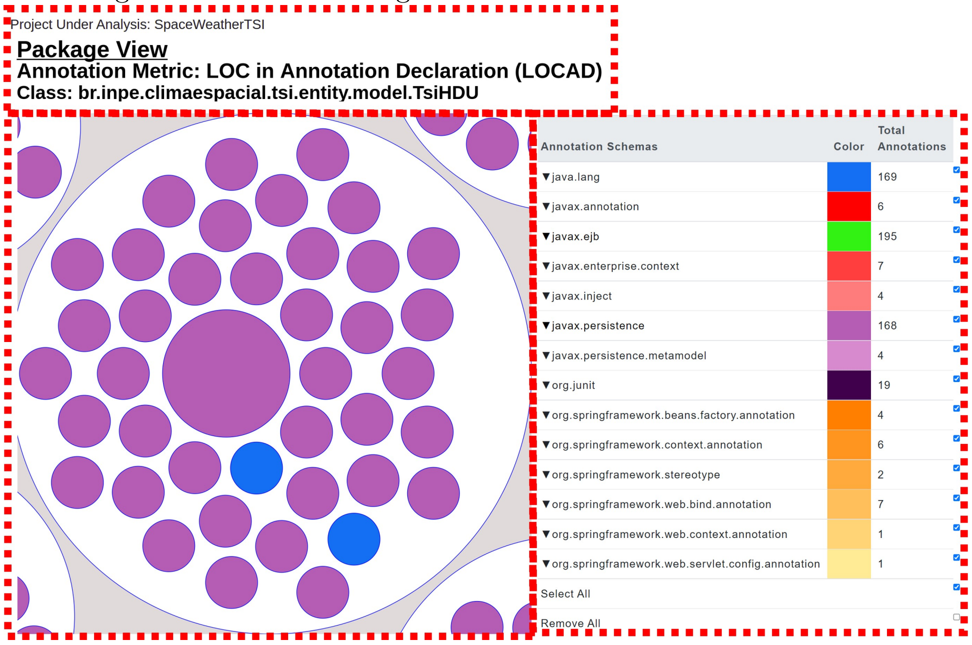
By inspecting Figure 5.12 we can, visually, obtain the following information:

- There are 14 classes with code annotations in this package. In other words, there are 14 white circles.
- The ***Header*** displays the name of the package, the used metric and the type of the ***View***
- Class marked with (1) appears larger than the others. That means it has more code annotations than the other classes inside this package.
- Class marked with (1) has one large dark-pink circle. It means this code annotation has a high value of LOCAD.
- Class marked with (1) also has code annotations from the schema `java.lang`
- Class marked with (2) seems to have all code annotations with the same LOCAD value, i.e., all circles have the same size.
- Class marked with (2) has no blue circle, which means no code annotation from the schema `java.lang`.
- Class marked with (2) has no code annotations from schema

`javax.persistence.metamodel`. It is not, however, easily observed since the color (pink tone) is close to the `javax.persistence`. As already mentioned, this was a design decision to keep schemas from similar families with a similar color. Since they are related, it does make sense to have similar colors. The same occurs with the schemas related to `org.springframework`, where we used an orange tone.

The information listed above can also be obtained from source code inspection, but not as quickly as our visualization enables.

Figure 5.13 - The Package View with Zoom on Class *TsiHDU*.



Project Under Analysis: SpaceWeatherTSI

Package View

Annotation Metric: LOC in Annotation Declaration (LOCAD)

Class: br.inpe.climaespacial.tsi.entity.model.TsiHDU

Annotation Schemas	Color	Total Annotations	
▾java.lang	🟦	169	✓
▾javax.annotation	🟥	6	✓
▾javax.ejb	🟩	195	✓
▾javax.enterprise.context	🟥	7	✓
▾javax.inject	🟥	4	✓
▾javax.persistence	🟪	168	✓
▾javax.persistence.metamodel	🟪	4	✓
▾org.junit	🟪	19	✓
▾org.springframework.beans.factory.annotation	🟧	4	✓
▾org.springframework.context.annotation	🟧	6	✓
▾org.springframework.stereotype	🟧	2	✓
▾org.springframework.web.bind.annotation	🟧	7	✓
▾org.springframework.web.context.annotation	🟨	1	✓
▾org.springframework.web.servlet.config.annotation	🟨	1	✓
Select All			✓
Remove All			☐

Just as with the ***System View***, we can also zoom in a class on the ***Package View***. We will choose the class **TsiHDU** because it has the highest number of annotations. Figure 5.13 displays this class with a zoom. We can see several pink-toned circles and two blue-toned. From the **Schema Table** we see the blue ones belong to the *java.lang* schema, and the remaining pink-toned are from *javax.persistence*. We can also see a large pink-toned circle that contrasts with all the other colored circles. We can extract some information just by look at this view:

- This class is coupled to two annotation schemas: `java.lang` and `javax.persistence`.
- This class has one annotation that used more lines of code (LOCAD) than

any other. Visually, we do not know which annotation it is. By hovering the mouse on the code annotation, a label appears and gives more details: name of the code annotation, LOCAD value, and schema. If we hover the mouse over this annotation is the `@Table` and the LOCAD value is 6.
- The `Header` changed to display

In the ***Package View*** we cannot see, however, how these annotations are configuring a specific element inside the class. For instance, we do not know what methods have more code annotations.

Comparing with the ***System View***, more information is now available to the user since the visualization seems a little more crowded with circles.

To observe how code annotations are configuring specific code elements, we must transition to the ***Class View***

5.2.4 Class View

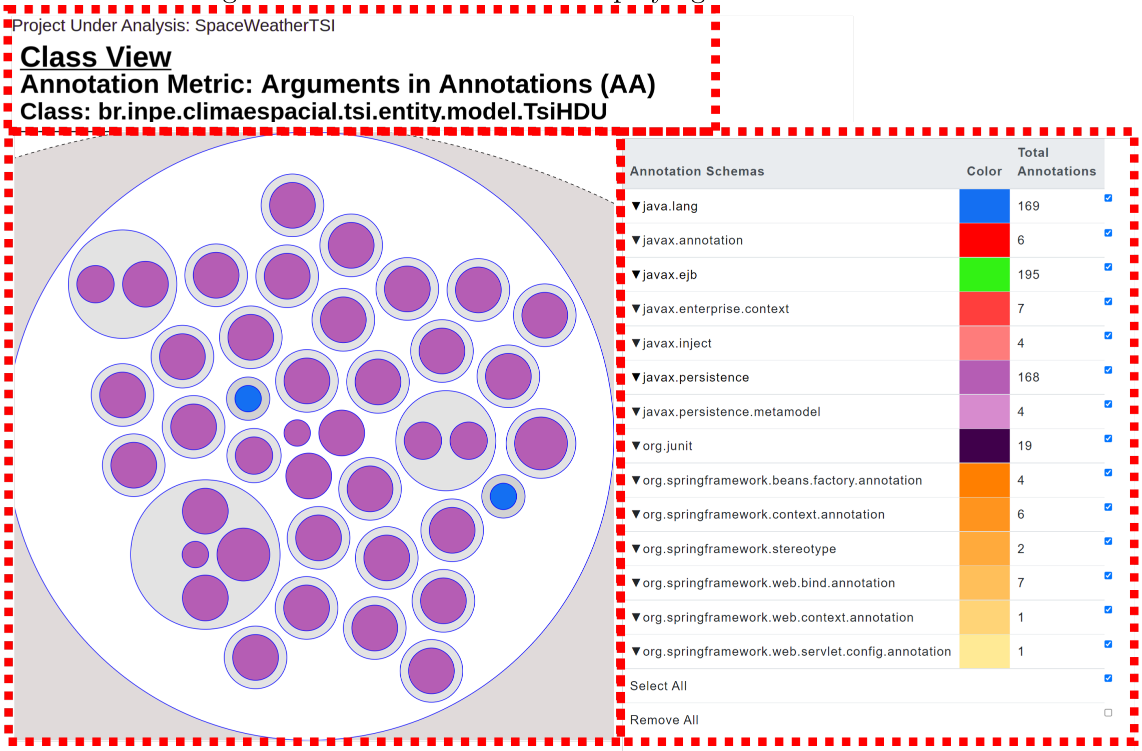
The **Class View** can display classes and individual code annotations inside the observed class. It allows users to visualize how code annotations are configuring a specific programming element, such as a method, class member, or the class itself. The circles are rendered with the following rules.

- Classes: Just as in the ***Package View***, they are rendered as a white circle. There is only one white circle at a time since we are analyzing a specific class.
- 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.

Therefore, we have two reserved colors. White represents classes, and gray represent code elements. It is the same gray used to represent the color of a package (dashed outlined circle), but it will not affect our ***Class View*** analysis since we are focused on a single class. The gray color provides a suitable color to be used as background. Therefore, we reused it in the ***Class View*** to represent the background color of circles representing code elements. Every other color represents a schema.

When we see a gray circle, we cannot tell if it is a method, enum, or other code elements. We did this as a design decision to avoid confusing users by showing more information and more colors. We were focused on displaying the grouping of code annotations by code elements, which is seen based on the size of gray circles. Figure 5.14 displays the ***Class View*** for class *TsiHDU*.

Figure 5.14 - The Class View displaying the class *TsiHDU*.



Project Under Analysis: SpaceWeatherTSI

Class View

Annotation Metric: Arguments in Annotations (AA)

Class: br.inpe.climaespacial.tsi.entity.model.TsiHDU

Annotation Schemas	Color	Total Annotations	
▼java.lang	🟦	169	✓
▼javax.annotation	🟥	6	✓
▼javax.ejb	🟩	195	✓
▼javax.enterprise.context	🟥	7	✓
▼javax.inject	🟥	4	✓
▼javax.persistence	🟪	168	✓
▼javax.persistence.metamodel	🟪	4	✓
▼org.junit	🟪	19	✓
▼org.springframework.beans.factory.annotation	🟧	4	✓
▼org.springframework.context.annotation	🟧	6	✓
▼org.springframework.stereotype	🟧	2	✓
▼org.springframework.web.bind.annotation	🟨	7	✓
▼org.springframework.web.context.annotation	🟨	1	✓
▼org.springframework.web.servlet.config.annotation	🟨	1	✓
Select All			✓
Remove All			☐

From Figure 5.14 we see the ***Schema Table*** is the same. The ***Header***, however, has changed to inform that we are now in the ***Class View*** and also informs the new metric being used to draw code annotations, i.e, AA - Arguments in Annotations. Revisiting this metric, presented in Chapter 4, it measures the number of arguments passed to the annotations. Consider the code on Figure 5.15. The annotation `@Id` (line 3) has AA $= 0$, while the annotation `@GeneratedValue` (line 4) has AA $= 1$. We are unable to draw a circle with diameter value of zero, therefore we normalize the value by adding one. This is just to allow the correct rendering, but the labels displays correctly that the AA value is zero.

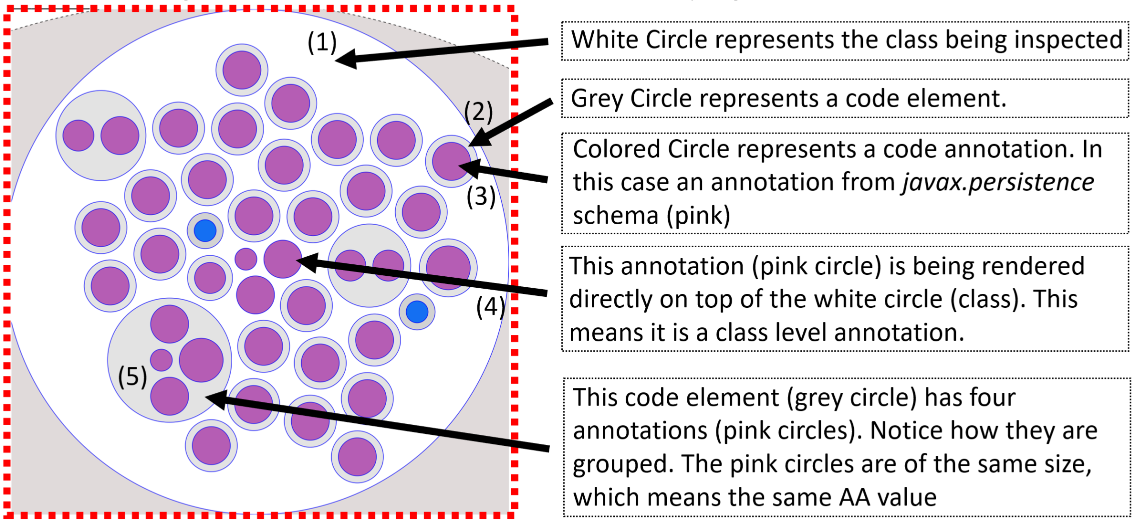
Figure 5.15 - Example Class to revisit AA.

```
public class Player {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
}
```

To further clarify the ***Class View***, Figure 5.16 displays a more detailed version with some markings we will use to guide the explanation. We removed the ***Schema Table*** and the *Header* to focus on explaining details of the view itself. We added some text to the Figure, and also we numbered the arrows to discuss them further here.

Figure 5.16 - A detailed Class View displaying the class *TsiHDU*.


(1)
(2)
(3)
(4)
(5)
White Circle represents the class being inspected
Grey Circle represents a code element.
Colored Circle represents a code annotation. In this case an annotation from *javax.persistence* schema (pink)
This annotation (pink circle) is being rendered directly on top of the white circle (class). This means it is a class level annotation.
This code element (grey circle) has four annotations (pink circles). Notice how they are grouped. The pink circles are of the same size, which means the same AA value

Arrow (1) points to the white circle, which is already known to be a class. The second arrow (2) points to the gray circle, which may seem like just a border. This circle is a code element. Just by looking at it, we cannot tell what element, or type, it is. However, if the user hovers the mouse over, a label appears with extra information about this code element. The arrow (3) points to a pink circle, representing an annotation from the `javax.persistence` schema. This last information can be obtained by the ***Schema Table*** (not shown in this Figure). In other words, the code element pointed by arrow (2) has one code annotation, pointed by arrow (3), configuring it.

Arrow (4) points to a pink circle, which is a *javax.persistence* annotation, being rendered directly on the white circle (class). This means that this code annotation configures the class directly. One example of such annotation would be the *@Table* or *@Entity.* Finally, the arrow (5) points to a large gray circle, with four pink circles inside (or being rendered on top). It means it is a code element with four code annotations. In short, the size of the gray circles (code elements) is proportional to the number of annotations configuring them.

This strategy aimed to quickly identify code elements with more code annotations and how they are grouped per code elements. A label appears with more information by hovering the mouse over this specific code element or the code annotations inside. We do not pollute the visualization and let the user explore only code elements or code annotations it is interested in.

5.2.5 Code Annotations Distribution Visualization summary

We summarize the proposed visualization, CADV - Code Annotations Distribution Visualization, suite in Table 5.2, which has the name of the view, its acronym, the goal is meant to achieve, as well as a summary of the metrics definition.

Table 5.2 - CADV Summary.

Name	**Acronym**	**Goal**	**Summary**
System View	SV	G1 and G5	Displays packages and schemas used. Each schema is assigned a color, and packages are gray circles with dashed outline
Package View	PV	G2 and G5	Displays a package, classes and code annotations used in the classes. Classes are white circles. Code annotations are assigned the schema color and are rendered on top of the white circles. Packages are gray circles with dashed outline
Class View	CV	G3 and G5	Displays a class, code elements and code annotations grouped by code elements. Classes are white circles. Code elements are gray color. Code annotations are assigned the schema color and are rendered on top of either the white circles , or gray circles.

The CADV is a polymetric view, using circle packing designed to aid in the visualization of code annotations used in software systems. We developed three different views to display information at a different granularity level. This strategy allows users to switch between these views and analyze the parts of the software with the desired granularity.

The CADV allows the visualization of the following characteristics:

- What schemas are used in the system?
- What schemas are concentrated? What schema is spread between several packages?
- What classes contain more code annotations?
- How coupled is a class to a schema?
- Is a code element overloaded with code annotations?
- Are there potentially misplaced code annotations?

5.3 Code Annotations Distribution Visualization assessment

This section presents the discussion and results of both experiments, E1 and E2, that we conducted to assess our novel software visualization approach for code annotations distributions - CADV. The first experiment, E1, was conducted by interviewing six former members of the EMBRACE software project. For the second experiment, E2, we managed to survey 44 students. We begin discussing the results for E1, followed by E2. In the end, we present a summary with a final discussion highlighting our findings.

Ultimately we are assessing our CADV approach that is implemented in the AVisualizer. We clarified to the participants that the tool, AVisualizer, is an ongoing work and still requires tuning, especially in the UI design. We also collected opinions and impressions of the tool and investigated the students' perceived ease of use and perceived usefulness.

5.3.1 Experiment E1 analysis

To guide our interview, we elaborated 15 questions presented in Table 5.3. We separated these questions into four categories using our goals (discussed in Section 5.1.1 as guidelines. Following, we present these categories:

- General Organization and Distribution: Assess how the AVisualizer tool can quickly provide useful information on how code annotations are distributed in the system under analysis.

- Code Responsibilities: Code annotations configure code elements so that a specific behavior can be executed. It is usually tied to responsibilities delegated to classes or packages. We wanted to find out if the participant could quickly detect the responsibilities of packages.
- Misusage: If the AVisualizer can display code annotations usage and distributions, can the AVisualizer also display potentially problematic and misplaced annotations?
- Tool Assessment: These questions were designed to obtain feedback from the participants about the AVisualizer tool and make it a better product to help software developers in their activities.

Table 5.3 also links the questions with the goals on the third column. The last four questions (Q12-Q15) deal directly with the tool's opinions and impressions (not only the CADV approach), so we feel they help every goal at once.

We now present a thorough discussion and analysis for each answer to every question we asked the participants. To enrich this section, we also provide our analysis and opinions. Some selected quotes of the participants are also presented.

As mentioned, the system under analysis was the *SpaceWeatherTSI* and the AVisualizer demo used to conduct this experiment is available online.[10]

5.3.1.1 Category A - general organization and distribution analysis

With these first five questions, Q1 - Q5, we were interested in validating how the interviewees were able to navigate between the three views, associate the colors displayed on the **Schema Table** with each view, and identify in what package or class it was currently inspecting.

Q1- What annotation schemas are concentrated in fewer packages?

We wanted to see if the interviewees could quickly spot schemas being used, or present, in fewer packages of the *SpaceWeatherTSI* project. We agreed that schemas such as the `org.junit`, `javax.persistence` and `org.springframework` were acceptable answers, since they are the ones used in less packages. The `org.junit`, for instance, is present in only one package. Since this was the first question, the AVisualizer displayed the **System View** which is the default view of the tool.

[10] `https://avisualizer.herokuapp.com/`

Table 5.3 - Code Annotations Comprehension Interview.

Question ID	Question	Goal
Category A - General Organization and Distribution		
Q1	What annotation schemas are concentrated in fewer packages?	G1
Q2	What annotations schemas are present in more packages?	G1
Q3	Is the circle packing able to represent the hierarchical structure of packages adequately?	G1
Q4	When changing to the ***Package View*** (any package), can you tell which class(es) contain the largest number of code annotations?	G2/G4
Q5	When changing to the ***Class View*** (any class), can you tell which code elements contain the largest number of annotations ?	G3/G4
Category B - Code Annotations Schema Responsibilities		
Q6	Which package(s) contain model classes mapped to databases?	G2
Q7	Which package(s) contain web controllers classes?	G2
Q8	Which package(s) contain unit testing classes? Is there enough unit testing code?	G2
Q9	Is it possible to identify packages with specific responsibilities by visualizing schemas?	G2
Category C - Misusage		
Q10	Is it possible to detect potentially misplaced code annotations? Describe the steps	G5
Q11	Is it possible to detect code annotations potentially being used excessively?	G5
Category D - Tool Impressions/Assessment		
Q12	Out of the three views, System, Package, and Class, which one did you prefer?	G1-G5
Q13	Do you believe the tool eased the process of seeing how the code annotations are distributed in the system? Without the tool, would you say there would be more or less effort?	G1-G5
Q14	Do you believe a newcomer developer could benefit from such a tool? (Consider that the developer already knows code annotations)	G1-G5
Q15	What role in a software team do you think can better use the AVisualizer tool?	G1-G5

We got very similar answers from all interviewees, but some took a different approach. Except for participant P5, all others were able to identify the schemas quickly. As seen on Table 5.4, most participants pointed both `org.springframework` and `javax.persistence`, with the latter being the most picked one. Even though participant P5 took more time exploring the visualization, it was the only one that

Table 5.4 - Answers for Q1 - Schemas Present in Less Packages.

Participant	javax.persistence	org.junit	org.springframework
P1			X
P2	X		
P3	X		
P4	X		X
P5	X	X	X
P6			X

also pointed the `org.junit` schema.

Participant P2 stated: *"The `javax.persistence` draws much attention since it has the highest number of annotations (169) in the project and the vast majority is concentrated in a single package".*

In the **System View**, the rendered circle was much larger than any other circle. Furthermore, since "pink circles" (that represents the `javax.persistence` schema) were available in fewer places, it drew even more attention. Participants P3, P4, and P5 had the same impression.

We also had participants that favored other schemas, still correct, for the answer. Participant P6 stated: *"Looking at the schema table, the `org.springframework` does not contain large amounts of annotations being used, so I would say that this schema is present in fewer packages"*

After this answer, it was clear that P6 searched for schemas with fewer code annotations instead of schemas used in fewer packages. P1 also had the same impression as P6.

Although there is a correlation, they are different things. For instance, consider a hypothetical schema, `schema1`, with ten annotations being used in the system. Suppose now they are spread between ten packages, i.e., each package contains one annotation from `schema1`. The AVisualizer tool would show ten circles of the same color in ten different packages. If the participant focused on the **Schema Table**, it would see ten annotations. It could wrongly conclude that this schema is present in fewer packages while, in reality, it is present in 10 different packages.

Consider now another hypothetical schema, `schema2`, that has 200 code annotations being used, but all of them are concentrated in the same package. Using only the

Schema Table the participant would see the number 200 and wrongly conclude that this schema is present in several packages, while it is used in only one package.

We reinforce that the provided answer by P1 and P6 was adequate, but the approach used could have lead to wrong conclusions of the system being analyzed.

As for participants P2-P5, they all "searched for colors grouped or present in fewer packages". Moreover, the tool was able to display such information. We believe that the reason only participant P5 identified the `org.junit` schema, was because the package with said schema was rendered a little further from packages with the `javax.persistence` and `org.springframework`. Given that P5 spent more time observing the visualization, it was also able to spot `org.junit`.

Q2 - What annotations schemas are present in more packages?

This question is the opposite of Q1. We wanted to spot schemas present in several packages. We agreed that both `java.lang` and `javax.ejb` were good answers. Table 5.5 presents the schemas answered by the participants.

Table 5.5 - Answers for Q2 - Schemas Present in More Packages.

Participant	java.lang	javax.ejb	javax.persistence
P1	x	x	
P2	x	x	
P3	x	x	
P4	x	x	
P5	x	x	
P6	x	x	x

For this second question, the interviewees answered faster than Q1. Half of the participants already answered Q2 while answering Q1. Participant P3, stated: *"I can clearly see that the `javax.persistence` is present in less packages, while the `java.lang` and `javax.ejb` is present in almost all packages"*

Participant P4 stated something similar, but was more confident that the `java.lang` is present in more packages: *"I can quickly see that green (`javax.ejb`) and blue (`java.lang`) circles are everywhere. But I can also spot packages with blue circles and no green circles. Therefore, I believe that the `java.lang` schema is present in more packages "*

Participant P6 also pointed the `javax.persistence` as being present in more packages. Which we marked as a wrong answer (we colored it red in Table 5.5) since it is the opposite, i.e., the `javax.persistence` is present in fewer packages. P6 stated: *"Looking at the schema table, I can quickly see that the `javax.ejb`, `java.lang` and `javax.persistence` are the schemas with more code annotations being used on the system under analysis. So I believe these are the schemas more present in several packages."*

Participant P6 used the same strategy as done for Q1, i.e., count the number of code annotations using the **Schema Table**. We already discussed, in Q1, that this strategy could lead to a wrong analysis, which came true now. P1, however, analyzed differently now (instead of only using the **Schema Table**) and correctly identified the schemas.

We considered that the question might have been formulated inadequately, but since all other 5 participants quickly reached the correct answer, we discarded this. The other option was perhaps our visualization approach, CADV, was not clear enough or missing details. Further questions clarified this for us.

In short, from the answers of participants P1-P5, the proposed approach showed how annotation schemas were distributed in the system.

Q3 - Is the circle packing able to represent the hierarchical structure of packages adequately?

We wanted to see if our approach using circle packing and the dashed-outlined circles could provide the hierarchical structure of packages in Java systems.

Except for participant P6, all others could perceive the Java packages being displayed. P6 stated that: *"For me, it is not evident that these nested circles (with dashed outlines) are representing the same Java package structure found in the system under analysis. The tool should provide more labels to aid."*

This answer was essential because it made sense why P6 struggled with the previous two questions. The core problem was that the visualization was not providing sufficient information about circles represented packages and how they were distributed. So P6 was relying more on the **Schema Table** to provide answers.

The other 5 participants thought that the used strategy was apparent to display packages and their nesting structure. Participant P3 stated: *"I can identify the Java* *packages looking at the provided view. I can also identify circles that represent inner packages. It is clear. Of course, the training session was helpful and the familiarity with Java packages. Programmers of other languages might not feel the same way"*

Given that five participants felt that the circle packing did indeed represent the Java packages, we believe some minor improvements on the UI could make it more transparent for all potential users of the tool. The requirement is that the user must be familiar with the Java package system.

Q4 - When changing to the Package View (any package), can you tell which class(es) contain the largest number of code annotations?

In this question, we wanted to see if the ***Package View*** could display code annotations, their schemas, and their size in classes. It is essential to remind that in this view (***Package View***), we are unable to see how code annotations are distributed inside the class, i.e., if they are configuring a specific method or field. The metric used to determine the size of the circle representing code annotation was LOCAD (Lines of Code in Annotation Declaration).

The participants were free to choose any package they wanted to inspect. Depending on their choice, the answers could greatly vary and still be correct. At this point in the interview, the tool was displaying the ***System View***. Table 5.6 shows the packages the participants chose to inspect and the class they pointed as being the "one with more code annotations".

Table 5.6 - Answers for Q4 - Classes with More Annotations.

Participant	**entity.model**	**business.msg**	**collector.scheduler.missing**
P1	TsiHUD		
P2		PropertiesBusiness	
P3	TsiHUD		
P4	TsiHUD		
P5	TsiHUD		
P6			JsocPeriodServiceTest

As observed, three participants chose the `entity.model` package (which contains code annotations from the `javax.persistence` schema). When asked why they mentioned the familiarity with the schema and the size of the circle. P3 stated: *" The pink circle (with the `javax.persistence`) is very large and draws attention, which* *means there are many code annotations in that package. It might make it easier to answer the question. Packages with fewer code annotations might be harder to inspect."*

Even though the tool should be able to provide enough information for the participants to answer this question regardless of the chosen package, it is no surprise that they would choose to inspect packages with familiar annotation schemas (such as `javax.persistence`).

After the tool switched to the ***Package View***. All participants were able to detect the class with the highest amount of annotations quickly. Participant P2 stated: *"I know that white circles are classes, and the class* ***`PropertiesBusiness`*** *(the label showed the name) has the highest number of colored circles (which are code annotations). So this surely is the class with the highest number of code annotations"*

Participant P4 also mentioned that: *"I can easily see that the class* ***`TsiHDU`*** *has the highest number of annotations. However, this does not mean an excessive use or design fault. Some classes require more code annotations".*

This was an interesting observation, even though this question was not addressing concerns about misusage or potential problems. Another aspect was the LOCAD metric used to determine the size of the code annotations in the ***Package View***. Since code annotations usually take up 1 line of code (LIMA et al., 2018), most colored circles (code annotations) were the same size. What drew the participants attention was the color (that represents the schema) and the number of colored circles inside a class (nested colored circles rendered on top of the white circle)

With these answers, we believe the tool could ease the process of "identifying code annotations on classes, their schemas and what class has more annotations".

Q5 - When changing to the Class View (any class), can you tell which code elements contain the largest number of annotations ?

In this question, we wanted the participants to explore the ***Class View***. In this view, we can see how code annotations are distributed inside any class being inspected. We can see their schema (colors), their grouping by code elements (methods, fields), and a chosen metric value to display their size. For the experiment, we are using the AA metric (Arguments in Annotations). In short, code annotations with more arguments are rendered as larger circles. Notice that we change the metric from the ***Package View***, which was LOCAD. We did this on purpose since we wanted to display more metrics values but not on the same view. It is known that displaying several metrics at once can confuse the user. We used the change of views (Package View to Class View) to change the metrics and analyze the results.

Just as with Q4, the participants were free to choose any class to inspect and determine which code elements contained more code annotations. Since this question was immediately after Q4, the participants were already on their chosen package (in the ***Package View***), and the majority of them almost immediately picked the class with the highest amount of code annotations.

After switching to the ***Class View***, the users were at first a little uncomfortable and spent some time analyzing the view, even though they were already trained. They issued different commands to explore the labels and better understand what was being displayed to them. Table 5.7 shows the class and the code elements they chose as having the highest number of annotations.

Table 5.7 - Answers for Q5 - Code Elements with More Annotations.

Participant	**TsiHUD**	**PropertiesBusiness**	**TsiRingsArea**
P1	Field - Id		
P2		PropertiesBusiness - class	
P3	Field - Id		
P4	Field - Id		
P5			Field - Id
P6	Field - Id		

The participants that chose the `TsiHDU` class were able to identify that "there is a code element with four code annotations". By hovering the mouse over the circle, the label revealed that it is a field called `Id`. In other words, the class `TsiHDU` has a field named `Id` that has four code annotations.

Participant P2 chose the class `PropertiesBusiness` and was also able to detect "four code elements configuring the class itself". It means that four annotations were together but had no circle frame grouping them. In other words, these are class-level annotations and P2 correctly identified.

Participant P6, however, was having trouble because the class `JsocPeriodServiceTest` had the same amount of annotations configuring all the elements. It is a unit testing class, with the `@Test` annotation on all methods.

The participant could not conclude that all code elements had the same amount of code annotations.

Another participant, P3, predicted something like this could happen: *"Packages and classes with fewer annotations might be harder to inspect and draw conclusions because nothing will stand out."*. Therefore, we suggested to P6 to choose another class or package for the analysis. The participant started to navigate the packages and chose to inspect `entity.model`. This package contained classes with code annotations from the `javax.persistence` schema, and the majority of the participants were already using it as a "default" package to answer questions.

After P6 switched packages and chose the class `TsiHDU`, it was able to spot an element with four code annotations: *"I can see a code element with four colored circles grouped. However, I do not know what type of code element it is."*

We explained that was the correct answer and gave more details about the ***Class View***. We returned to the `JsocPeriodServiceTest` and asked the same question to P6. Now the participant was able to see that all code elements had the same amount of code annotations and could comprehend the ***Class View***.

When comparing to the *System View* and *Package View*, the *Class View* was more confusing for the participants. Given that this view displays more details of code elements, we expected this behavior. Furthermore, the participants felt that it was troublesome to comprehend the change of metrics (LOCAD to AA) when switching views.

As P1 stated: *"When we were in the **Package View**, the code annotations (colored circles) size was derived from the lines of code (LOCAD metric), when we switched to the **Class View**, the size of the code annotations is derived from the number of arguments. This can be very confusing, and it made analyzing the **Class View worse**"*

We did this change on purpose to experiment if this approach would be interesting to display different metrics. P2 also stated: *"I do not see how using the LOCAD metric was helpful in the **Package View**. Perhaps using the AA metric (or at least the same metric) in both **Package View** and **Class View** would be more interesting. Even better would be to offer means to customize the desired metric, but without losing reference to the annotations being analyzed"*

In general, the participants said that improvements in labels would suffice. After answering Q5, participants felt more comfortable with the tool and started to grasp what was being displayed.

5.3.1.2 Category B - schemas responsibilities

Questions Q6 - Q9 are more technical about code annotation usage. We wanted to measure how the tool can aid in detecting packages with specific responsibilities based on annotations schemas. They all have a similar analysis, but we will discuss each question separately.

Q6 - Which package(s) contain model classes mapped to databases?

All participants took the same approach to answer this question. They first looked at the **Schema Table** and searched for words like `hibernate` or `persistence`. Then they saw what colors were assigned to these schemas and scanned the visualization with packages that contained circles with those colors. This particular schema was easy due to the high number of annotations present. In other words, it was easy to spot a "big pink circle". With this, they quickly detected what packages contained "classes being mapped to databases".

P2 stated that: *"This was pretty straightforward. Just search the schema table, find the color for `javax.persistence` and look in the view. It is quick to find the packages with pink circles. However, the user must be familiar with the `javax.persistence`"*

Q7 - Which package(s) contain web controllers classes?

This question was also quick to answer, but not as much as Q6. This is because the Spring Framework contains several different schemas[11]. The Java Persistence API, on the other hand, has few schemas, for instance, the `javax.persistence`. Nevertheless, they were able to find the package concerned with web controllers. Participant P4 stated: *"I know that the Spring Framework deals with web controllers, so I will just search the table to see what are the schema colors for spring. "*

Participant P6 was not so sure that it was related to the Spring Framework. So the approach was a little different: *"I will search the schema table for the words web or controllers. Then I'll see the schemas with these words and search the package with these colors (orange for the Spring framework). However, If I were unable to detect*

[11]Revisiting, we define an annotation schema (or simply schema) as the package that created the code annotation. This decision is made by the developer of the framework annotation

through the schema, I would have to manually navigate each package, switch to the Package View and search labels with the words web or controllers. In this case, I would be searching by code annotations, instead of the schema"

The schema does indeed contain the word "web", which helped P6 conclude that the schema is `org.springframework.web` only using the Schema Table.

Q8 - Which package(s) contain unit testing classes? Is there enough unit testing code?

Given the popularity of the JUnit framework, this was the most obvious one. Even Java developers not familiar with web development are familiar with the JUnit.

Participant P1 stated: *"I will follow the same strategy I did for the other two questions. No difference. Look at the Schema Table to see the color, then go to the visualization searching for circles of the assigned color"*

We also wanted to see if the developers could conclude if the number of unit testing classes were enough.

Only participant P3 believed that there was insufficient unit testing classes: *"Looking at the **System View**, it is pretty clear that there is only one package with JUnit classes. This could lead to the conclusion that there might not be enough unit tests in this project."*

All other participants felt the available information was insufficient to conclude whether there are enough unit testing classes.

As the developers of the AVisualizer tool, since there is only one package with unit testing classes, we believe that there is not enough code coverage in the system being analyzed.

Q9 - Is it possible to identify packages with specific responsibilities by visualizing schemas?

This question summarizes the previous three (Q6, Q7, and Q8). We wanted to know if our CADV approach allows the user to detect packages responsibilities based on code annotations or schemas quickly.

P4 had a very interesting observation: *" There are annotation schemas, such as `org.junit` or `javax.persistence`, in which their role is very clear in Java soft-*

ware system. For these schemas, it is straightforward to point out the responsibility of a package. However, some schemas, like `java.lang` and `javax.ejb` is not very clear. Packages that only contain code annotations from these schemas only, I am unsure what they are doing. A closer look (changing to Package View, Class View or even inspecting the code itself might be necessary."

P2 stated:*" Yes, definitely it is very easy. However, the system under analysis must use code annotations to execute behaviors. Although they are very popular, some Java systems simply do not use them as much as web applications, for instance. "*

The other participants reinforced that it is easy to detect packages responsibilities using the AVisualizer tool. However, there are two basic requirements.

- The system must use code annotations to execute behaviors
- The user must be familiar with code annotations

5.3.1.3 Category C - misusage

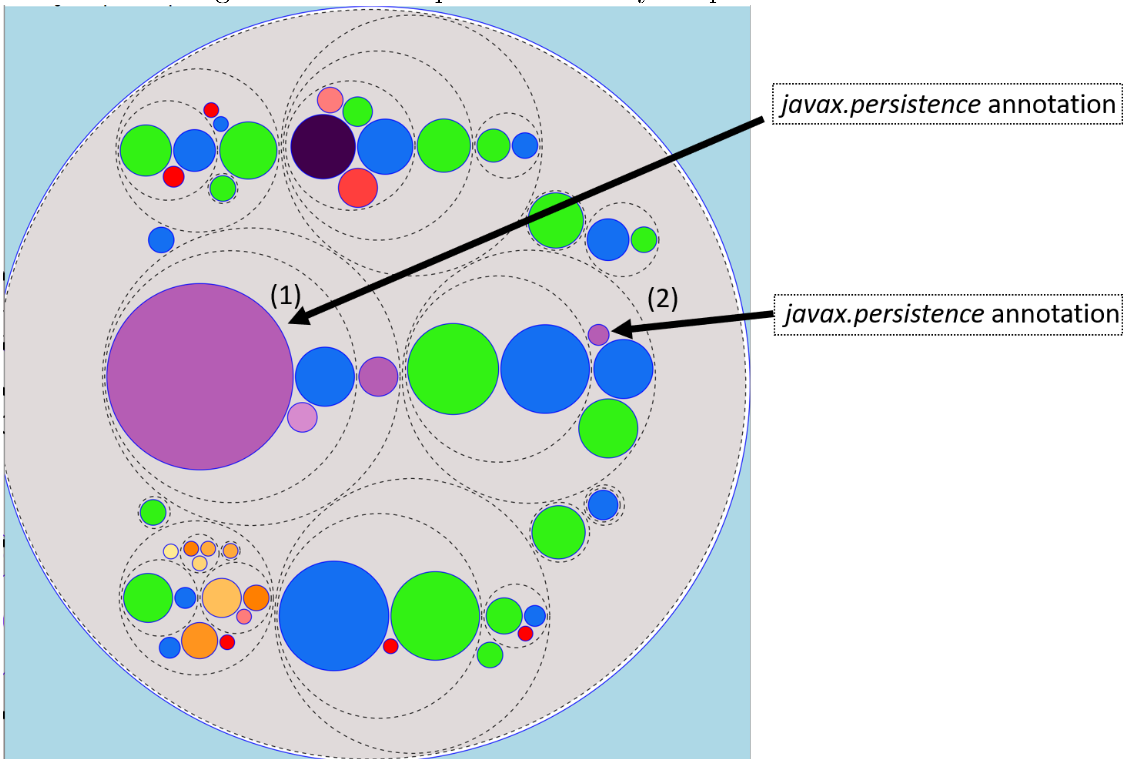
We are now interested in observing how the tool can help detect potential errors or misusage of code annotations. It is not the primary goal of the CADV approach, and we did not conduct any specific study related to code annotations bad smell, or design flaws. However, we still wanted to see how the participants could use it to detect potential problems.

Q10 - Is it possible to detect potentially misplaced code annotations? Describe the steps

If a code annotation was potentially misplaced, we believe the quickest way find detect it would be to "spot a circle of a given color, far away from all circles of that same color". Another approach would be to "spot a tiny circle of a given color in a package with no other circles of the same color". As an example, consider Figure 5.17, showing the **System View** for the *SpaceWeatherTSI* project.

Arrow (1) points to a large pink circle, which means this package contains many code annotations from the `javax.persistence` schema. However, the arrow (2) points to a tiny pink circle, meaning that the package also contains code annotations from the `javax.persistence` schema. However, that code annotation is isolated from other *javax.persistence*, in another entirely different package. This, however, does not mean it is potentially misplaced, but it may indicate that further inspecting is required.

Figure 5.17 - Example of Potentially Misplaced Annotation.



During the interview, we did not point or suggested anything to the participants. We wanted them to speak freely about this topic. Most participants manifested that they were able to use the tool to detect potentially misplaced code annotations. Nevertheless, some reinforced that knowledge of annotation schemas improves this analysis.

P1 stated: *"Yes, it is very simple to detect potentially misplaced codet annotations. If I spot a pink circle in a package with only orange circles, I would say something is wrong. Why is a `javax.persistence` code annotation alone in a package with only `org.springframework` code annotations?"*

Other participants had a very similar analysis. P3 stated: *"The colors help a lot in detecting these potentially misplaced annotations. However, I argue that whoever is analyzing the system should also be familiar with code annotations and schemas. This way, a better decision is made to determine if the package or class requires* *further investigation or code inspection."*

Moreover, P6 stated: *"Yes, it is possible. I can even detect frameworks not present in the software being analyzed. I think the colors are very useful, but I also feel the Schema Table helps a lot. For instance, if I inspect the table and do not see the `org.junit` schema, I know this project does not contain unit tests. Using the Schema Table, developers familiar with metadata-based frameworks can easily spot schemas they believe would improve the project.*

Even though we did not consider finding errors a primary goal, we have collected evidence that the CADV approach can also be used for such analysis.

Q11 - Is it possible to detect code annotations potentially being used excessively? Or very large?

From what we presented in Chapter 4, we have evidence of extensive code annotations, using several lines of code and a high number of arguments. We wanted to see if the tool could easily show these cases and how participants would use it.

Most participants agree it is easy to use the tool for such purpose, meaning basically to inspect "huge circles". However, the participants could not see actual values or usefulness in searching "large code annotations", unless they were huge (which we meant by outlier). Seeking code annotations that are potentially misplaced seemed more exciting and brought more benefits to the code quality from the participants' point of view.

P2 stated: *"With the current CADV approach, I would seek the size of the circles. However, it is very vague to declare if this size is a problem. Also, the CADV does not show classes or code elements without code annotations, so I feel it is even harder to determine if a code annotation is problematically large."*

P3 reinforces that it is tough to determine anything just by using the size, stating: *"At first, I would use the diameter of the circle. However, I feel the tool is not adequate for this type of analysis. Many design decisions may include using several extensive code annotations, with many arguments."*

As we mentioned, we have evidence of huge code annotations present in open-source software. However, we were not interested in investigating these aspects in this experiment, and we did not present software with such problems to the interviewees. If, for instance, we presented to the user a code annotation that takes 53 lines of code, the answers might have been different. Future experiments focused on bad smells will address this.

5.3.1.4 Category D - tool impressions/assessment

In this category of questions, we are interested in general opinions of the tool and usage scenarios.

Q12 - Out of the three views, System, Package, and Class, which one did you prefer?

We designed each view of the CADV with its own goals. They were meant to complement each other. However, during our exploration, we used the **System View** far more often than the other views.

As we expected, five participants quickly answered that they preferred the **System View**. The exception was P5 that preferred the **Class View**, stating: *"I enjoyed being able to see how code annotations were grouped inside the classes."*

While P6 also preferred the **System View**, it was not as direct as the other: *"I had only very brief contact with the tool, and it is difficult to tell which view I enjoyed most or found more useful. As a first answer, I will say the System View because it is a lot cleaner, and I can see the whole system. But to give a better answer, I would have to experiment a little more with the tool."*

Participant P3 explained why the **System View** was the preferred one: *"I would say the System View is more useful and less confusing. It has less information being presented, but I can see the whole system and have an excellent understanding of how code annotations are distributed in the whole software under analysis."*

As other software visualizations work mentioned, users tend to prefer views with less information but simultaneously allows helpful information to be obtained. That is what the **System View** provides. Although this is not an original contribution, we have provided new evidence to support previous results about users preferring views with less information. Our original contribution was to provide a view specifically tailored for code annotations that is both simple and useful.

Q13 - Do you believe the tool eased the process of seeing how the code annotations are distributed in the system? Without the tool, would you say there would be more or less effort?

Most participants agreed that the tool is handy to visualize code annotations distribution and much faster than manual code inspection. P6 stated: *"Once, I had to check the pom file (configuration file for maven dependency) to see what metadata-based frameworks the project was using. If I had access to a tool such as the AVisualizer, I would have immediately identified these frameworks, at least the metadata-based ones"*

P2 stated: *"The AVisualizer tool does a great job in showing the code annotations and schemas used. If comparing with a* ***find option*** *(ctrl+f) in a text editor or IDE, the amount of effort spent is much greater than using the AVisualizer tool"*

An interesting observation made by participants was to implement the CADV as a plugin for an IDE. In other words, port the AVisualizer to a plugin. This way, they would benefit the most. While coding in their IDE, they could quickly inspect code annotations by executing the "AVisualizer plugin". It would not be necessary to switch environments, and the AVisualizer could also link back to the code. P6 stated: *"If I was using the plugin to see code annotations, and if I double-clicked the circle (that represents the code annotations), it would be handy if the IDE highlighted or jumped to the code"*

P6 also reinforced that: *"The AVisualizer is much faster than other IDE resources to find code annotations. However, having to open an external tool or browser is not very much appealing."*

Participants also suggested integrating the AVisualizer with a CI/CD pipeline such as SonarQube[12].

With this information, we will still keep the AVisualizer as a web application since it is easy to maintain and releases developers from installing applications. However, we will begin porting it as a plugin for the IntelliJ IDE[13].

Q14 - Do you believe a newcomer developer could benefit from such a tool? (Consider that the developer already knows code annotations)

Given that a newcomer developer is part of the target audience of our tool, we wanted to see how participants felt about it.

Except for participant P5, the others felt that it could indeed be helpful to a new-

[12]https://www.sonarqube.org/
[13]https://www.jetbrains.com/idea/

comer developer. Moreover, they should have access to such tools right from the start.

P5 stated: *"I do not feel that a developer should start the comprehension process from code annotations point of view. A much better approach would be first to understand what design patterns are being used in the project. Only after that, the developer should seek annotation schemas, or the metadata-based framework"*

As mentioned, the other participants felt differently. P2, for instance, stated: *"That was an excellent question. For instance, a newcomer developer that needs to add features related to persistence can use the tool and see all packages using `javax.persistence` schema. The new feature should likely be added to those packages. This eliminates a manual process of code inspection using IDE shortcuts. The AVisualizer tool is much quicker."*

P1 also feels the tool is helpful for a newcomer, but there are some considerations. *"Maybe it can be helpful. However, the developer must be very familiar with annotation schemas. This way, the tool can be used to check where, in the code, the developer should go next"*

Q15 - What role do you think can better use the AVisualizer tool

In general, the participants believe the tool can be used by every member that belongs to a software team. Nevertheless, some participants feel that the software architect can take advantage of the tool during the whole development life cycle. It is beneficial to see the organization and distribution of code annotations.

P1 stated: *"Usually developers are interested in quickly starting to code and adding new features. On the other hand, the architect is much more concerned with the structure and the organization. And also, a QA can quickly detect packages that are not being unit tested"*

And also, P4 stated: *"The architect can use this tool to control the growth of the system, checking if the responsibilities are being placed correctly in packages"*

P6 believes every member can benefit and stated: *"Every one that touches source code can benefit from this tool"*

5.3.2 Experiment E2 analysis

As we presented in Section 5.1.4, we also conducted an experiment with 44 students that we named E2 experiment. This section discusses the results of this experiment. We first discuss the student's technical background. Then we analyze the results from the ten close-ended questions. The students used the AVisualizer tool to identify code annotations information in the analyzed system to answer these questions. Just as for E1, the *SpaceWeatherTSI* was the used software. Finally, we analyzed the opinions and impressions of the students about the AVisualizer tool. The procedures and design of the experiment were discussed in Section 5.1.4.

Highlights for the E2 Analysis:

- Questions that required using the System View and Package View the students performed very well. More than 80% of the students answered these questions correctly.
- Questions that required using the Class View the students performed reasonably. Between 50% and 75% of the students answered these questions correctly.
- Students were not eager to use the AVisualizer to detect misconfigurations, but rather identify what code annotations are being used and further study them in other sources.
- Students do not see a big difference if the tool is an embedded plugin in the IDE or an external tool.
- Students find that the AVisualizer quickly helps to identify code annotations being used in the system. Much faster than manual code inspection.
- Students find that the AVisualizer requires improvements in the UI to understand better the metrics being presented.

5.3.2.1 Students technical background

The first part of the experiment was to obtain the technical background of the students. We asked about their familiarity with code annotations and what programming language they used primarily. They should have also considered experiences with C# attributes and Python decorators since they are very similar to Java code annotations. Figures 5.18 and 5.19 presents, respectively, the familiarity and primary language in a column chart.

Figure 5.18 - Students Familiarity with Code Annotations.

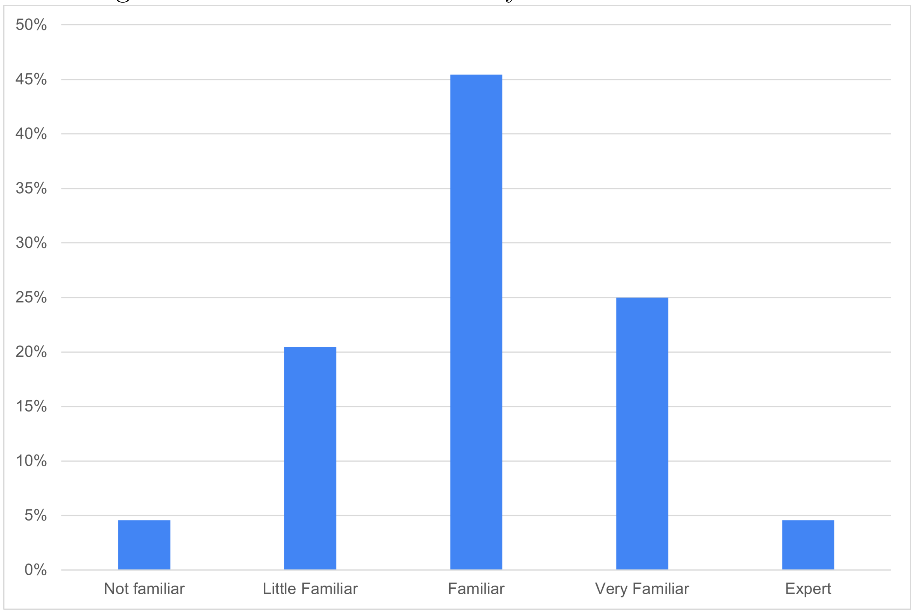
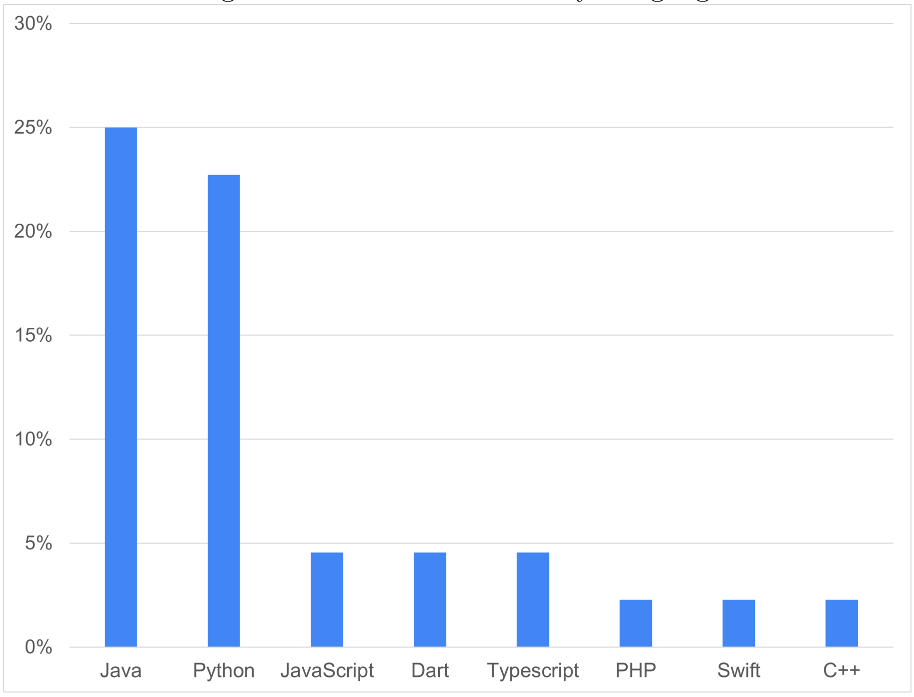


Figure 5.19 - Students Primary Language.



Roughly half of the students considered themselves *familiar* with code annotations, and less than 5% considered an expert or unfamiliar. We understand this is because the software engineering course taught in the involved schools is a Java-based curriculum. Therefore, students are constantly exposed to the `org.junit` code annotations such as `@Test` and `@Before` when studying unit testing. However, to be considered an expert would probably require skills such as creating their own code annotations and using reflection to retrieve and process them at runtime. Usually, students and even most developers are less exposed to such topics. Since the CADV is focused on visualizing code annotations being used and not creating annotations, we consider the target audience adequate for the experiment. As for the student's primary language, we can see the top two languages are Java and Python. This can also be explained given the nature of the software engineering curriculum used in the schools. As for Python, the courses that deal with machine learning use Python as the primary language. Therefore, it was expected that Java and Python would be highly present.

Table 5.8 - Code Annotations Students Questionnaire.

Question ID	**Question**	**Goal**
Q1	What annotation schema is located in a single package?	G1
Q2	What annotation schema is located in more packages ? (more distributed)	G1
Q3	Which annotation schema contains the largest amount of annotations being used?	G1
Q4	What class contains the highest number of ***javax.persistence*** annotations?	G2/G4
Q5	What package contains classes being mapped to databases (usage of ***javax.persistence***)?	G3/G4
Q6	What package is mostly concerned with web controllers (usage of ***org.springframework.web.bind.annotation***)?	G2
Q7	How many packages contains unit testing class(es)?	G2
Q8	What ***javax.persistence*** annotation has the highest LOCAD (lines of code per annotation) value?	G2
Q9	In the class `br.inpe.climaespacial.tsi.entity.model.TsiHDU` (fully-qualified name), what code element has more annotations configuring it?	G2
Q10	What annotation from the org.springframework.web.bind.annotation schema contains more attributes/arguments (Annotation Attribute metric - AA)?	G3

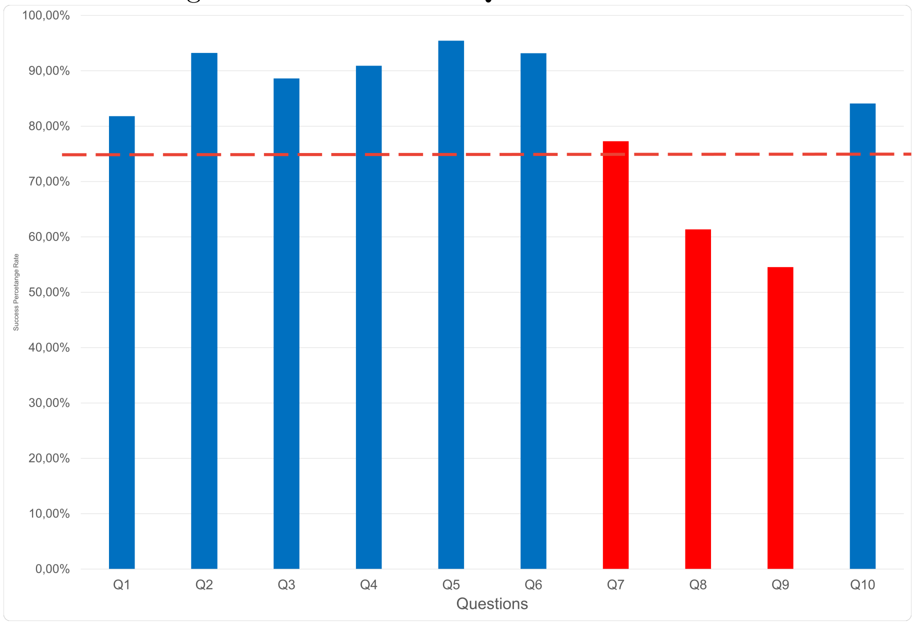
5.3.2.2 Technical questionnaire analysis

Since experiment E2 was conducted asynchronously, we did not use the same questions used in E1. Roughly half of those questions were very open-ended and, therefore, suited for an interview (like E1). For E2, we elaborated ten close-ended questions with five alternatives each. These questions aimed to detect if the students could identify code annotations characteristics from a target project using the AVisualizer tool. Just as E1, the target software used for E2 was the *SpaceWeatherTSI*. Table 5.8 presents the questions.

To prepare these questions, we used the goals proposed and discussed in Section 5.1.1. The relation between the goals and the questions is available in the last column in Table 5.8. Our Goal #G5 is concerned with detecting misconfigurations, and since this is not a primary goal of the CADV, we chose not to insert questions related to this topic in these ten questions. Since E1 was an interview, it was easier to discuss other topics.

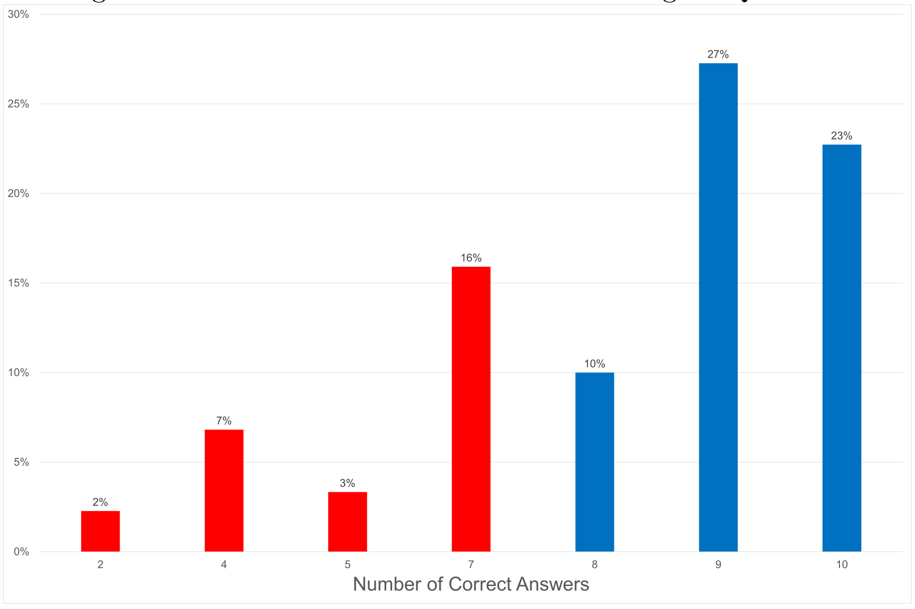
To analyze and discuss the student's performance, we have two Figures to guide us. First, on Figure 5.20 shows the success rate of each question. For instance, roughly 82% of the students got question Q1 correct. And Figure 5.21 shows the students performance considering all ten questions. For instance, we see that 23% of the students got all tens questions correct, and 27% answered nine questions correctly.

Figure 5.20 - Individual Question Success Rate.



We begin analyzing the success rate of each question presented in Figure 5.20. As seen, questions Q2 and Q4-Q6 had the highest hit rate, with more than 90% of the students answering correctly. Question Q9, on the other hand, 55% of the students got it correctly. The success rate of questions Q1 through Q6, and Q10, were considered very good, above 82%. However, it dropped considerably from Q7 (75%) till Q9 (55%). The first questions could be answered quickly using the **System View** or **Package view**, while Q9, for instance, required manipulating the **Class View**. This result has some correlation to the ones obtained in the E1 experiment analysis. Participants had more trouble dealing with the **Class View** in both experiments while performing better on the **System View**.

Figure 5.21 - Students Performance Considering all Questions.



Our research group considered that the first three questions were the easiest because they practically required no manipulation of the AVisualizer tool,i.e., navigating the views and checking the labels. On the other hand, we pointed the last three questions, Q8-Q10, as the hardest ones. To answer these final questions required manipulating the tool and minimal knowledge of code annotations metrics from our novel suite (discussed in Chapter 4).

To answer questions Q1-Q3 required a quick look at the **System View** or the **Schema Table**. It should have been enough to obtain the correct answer.

From questions Q4 through Q6, it required a little more effort and navigation of the tool, but at most to the **Package View**, which means only one level of zoom. These questions asked about a specific schema, and the students needed to identify the correct package and change it to the **Package View**. In short, it required medium manipulation of the AVisualizer tool.

For instance, for Q5-***What class contains the highest number of javax.persistence annotations?***- the students needed to identify a package with the schema *javax.persistence*, and change to the **Package View**. In this view, it is possible to identify the class with the highest number of annotations.

In general, the students did very well with these previous discussed questions (Q1-Q6).

Analyzing Question Q7, we observed the first decline of the success rate, with 73% of the students getting it correct. Our team assumed the students would have reached a similar success rate as the previous questions but was slightly lower. This question asked ***How many packages contains unit testing class(es)?***. The correct answer is "only one".

To answer the question, the students needed to know that the schema responsible for unit testing is the `org.junit`. In other words, Q7 also deals with the relationship between responsibilities and schemas. In questions Q5 and Q6, we made this explicit. To answer the question, the students needed to search how many packages contained code annotations of the schema ***org.junit***. It was also very visual and color-dependent since they had to search for "circles with the matching color of org.junit (with is a dark purple tone)". The **Schema Table** also has a check box for each schema that helps to find the location of these schemas (or colored circles).

Since the success rate dropped, we can assume that there is a need to reinforce the concept of "annotation schema". Furthermore, improvements in the UI of the tool and better filtering could have helped more students reach the correct answer. The idea of reinforcing the "annotation schema" concept was also obtained from the E1 results. Some participants mentioned that even though most Java developers are familiar with the annotation "@Test", most of them are not familiar with the concept that this annotation belongs to the "org.junit" schema.

Therefore, we can assume that the success rate of questions Q5 and Q6 could have dropped to 73% if we had not explicitly mentioned the schema required to answer the question.

For the final three questions, Q8-Q10, they required the students to manipulate the tool altogether, use the labels, and understand what some code annotation metrics, such as LOCAD and AA, were measuring. As observed in Figure 5.20 the success rate also dropped for these questions, especially Q9, where 56% of the students got it correct. Let us observe what the crucial points analyzed in each question was

- Q8 - Involved using the LOCAD (Lines of Code in Annotation Declaration) metric. The success rate was 63%
- Q9 - Involved observing the Class View and identifying the code element with more annotations. The success rate was 56%
- Q10 - Involved using the AA (Arguments in Annotation) metric. The success rate was 80%.

To answer question Q9, no metric was required, only detecting a "grey circle with the largest amount of colored circles". Questions Q8 and Q10 required using the labels and would help detect a "big circle", but the numerical information on the label would have been enough. We can assume that using the metric was not a big problem since the labels were helping. However, for Q9, it required understanding the **Class View**, the meaning of the rendered circles, and how they are grouped. The **Class View** was also a problem for the participants in E1. The problem with this view is that it shows more information and details for a single class which seemed to confuse the users. Even though we learned from previous works and tried to display only the utmost important information, it proved not as efficient as the **System View** and the **Package View**. Therefore, using our evidence for E1 and E2, we need to improve the **Class View** or redesign how we intend to display class details on a polymetric view.

As for the overall student's performance, showing in Figure 5.21, 23% of the students got all of the questions correct. Few students performed poorly, with 2% getting two correct answers, and 3% got five answers correct. We believe this is a good result, considering the students are not very familiar with code annotations and not familiar with using software engineering tools to visualize source code.

5.3.2.3 Perceived ease of use

As we mentioned in Section 5.1.4, we based on the TAM model to understand how easy and useful the students found the AVisualizer tool.

To measure the "perceived ease of use", we elaborated eight statements that measured if the students felt it was easy to use the tool, navigate between the views, and identify packages and classes. We used the label "General Navigation and Usage" to classify this group of statements.

These statements used the Likert Scale, ranging from *strongly disagree* to *strongly agree*. Table 5.9 presents the complete statements and an ID with used to identify them. Figure 5.23 present a diverging bar chart with the short versions of these statements, their ID, and the results based on the student's answers.

Table 5.9 - General Navigation and Usage Statements.

Statement ID	Statement
S1	I can easily identify java packages with different responsibilities using the AVisualizer tool
S2	I can easily see how code annotations are distributed in the system under analysis using the AVisualizer tool
S3	Learning how to use the AVisualizer was easy to me
S4	I can easily see how many annotation schemas are being used inside a java class using the AVisualizer
S5	I can easily see how many annotation schemas are being used inside a java package using the AVisualizer
S6	I can easily identify what java package I am currently inspecting using the AVisualizer
S7	I can easily identify the class I'm inspecting in the AVisualizer tool
S8	I can easily navigate to and from the packages and classes being analyzed with the AVisualizer tool

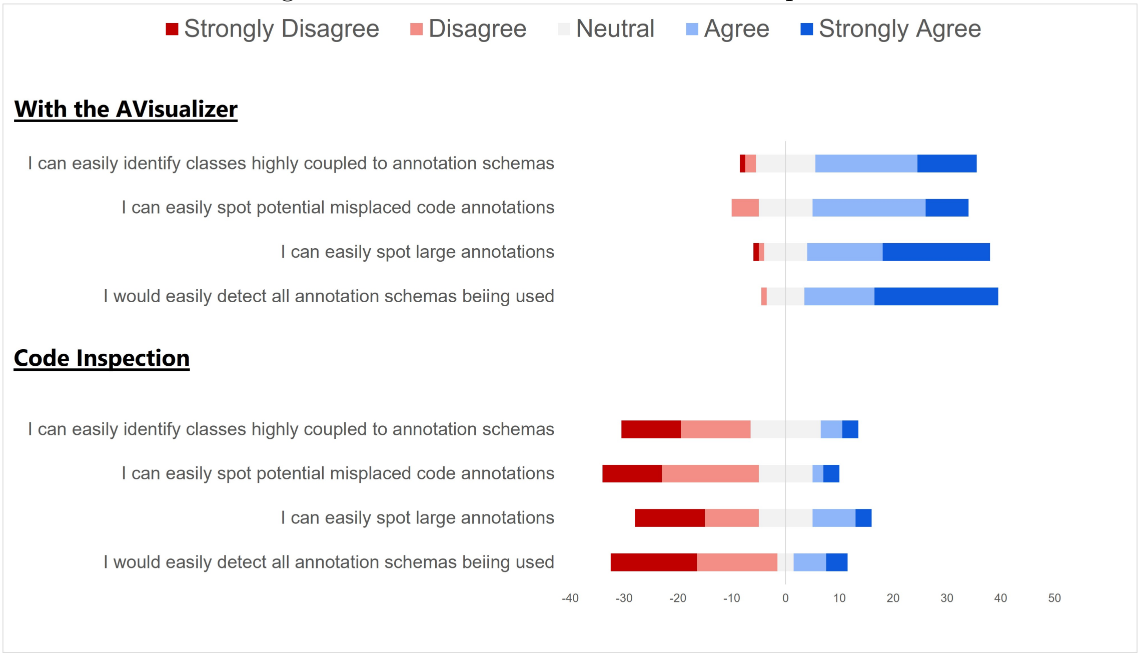
As seen, no student strongly disagreed with any statements. In fact, most of the students agreed or strongly agreed that, in general, using the AVisualizer tool is easy. We also had *disagree*, and the two most disagreed statements (S7 and S8) are dealing with navigating between all views and identifying classes, which is strongly related to the **Class View**, which has already been found to require improvements. This is yet another evidence that visualizations that require further inspection or display multiple information may be troublesome to interpret what is being displayed.

5.3.2.4 Perceived usefulness

To measure the "perceived usefulness", we created three different categories of questions:

- Code Inspection versus AVisualizer: We asked the students to perceive the effort to perform the same task with and without the AVisualizer tool. These were a total of eight questions.
- IDE Plugin versus External Tool: Developers prefer to use tools as a plugin for an IDE. Do students also feel that way?
- Primary Usage for the Tool: We created three possible scenarios for the students to use the tool and measure their primary usage.

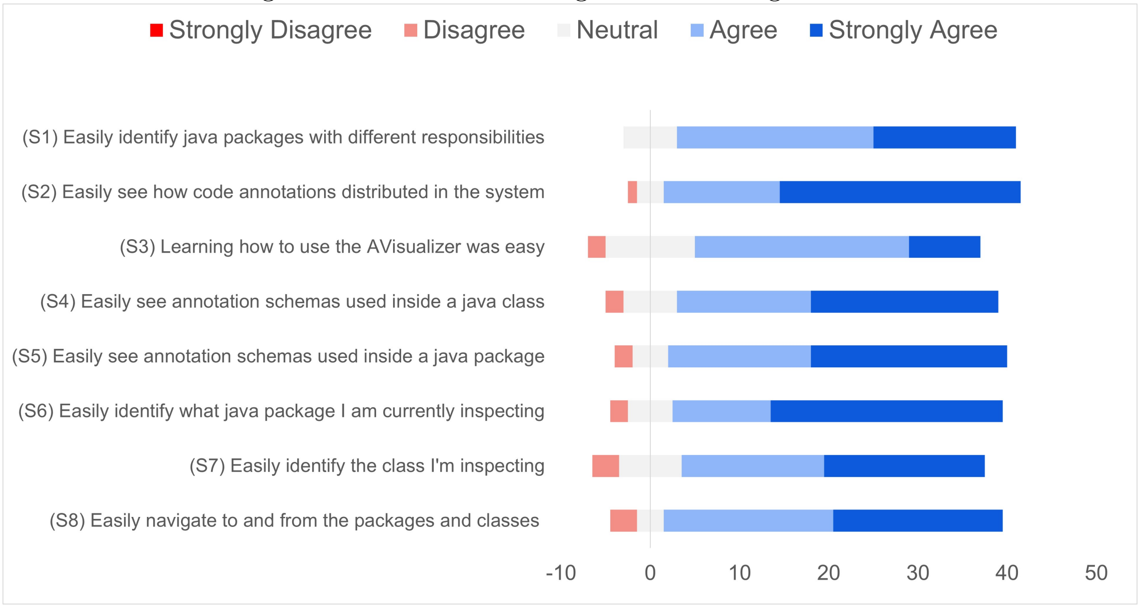
Figure 5.22 - AVisualizer Versus Code Inspection.



For *Code Inspection versus AVisualizer* we wanted to measure how students perceived using our tool to find code annotations related characteristics would compare to find the same information but only inspecting the source code with regular IDE shortcuts. Figure 5.22 shows our results using a diverging bar chart. We divided it into two regions. The top region, *With the AVisualizer*, presents the results considering the student was using the tool. The lower region, *Code Inspection*, has the same statements, but considering the students could only use source code inspection.

Most of the students found that using the AVisualizer to detect code annotations-related information is far easier when compared to manual code inspection. The statement that had the higher difference was S4 *I would easily detect all annotation* *schemas that are being used in a Java system.* Most students *strongly agreed* it would be easy using the AVisualizer, and *strongly disagreed* it would be easy without the AVisualizer, using only code inspection.

Figure 5.23 - General Navigation and Usage Results.

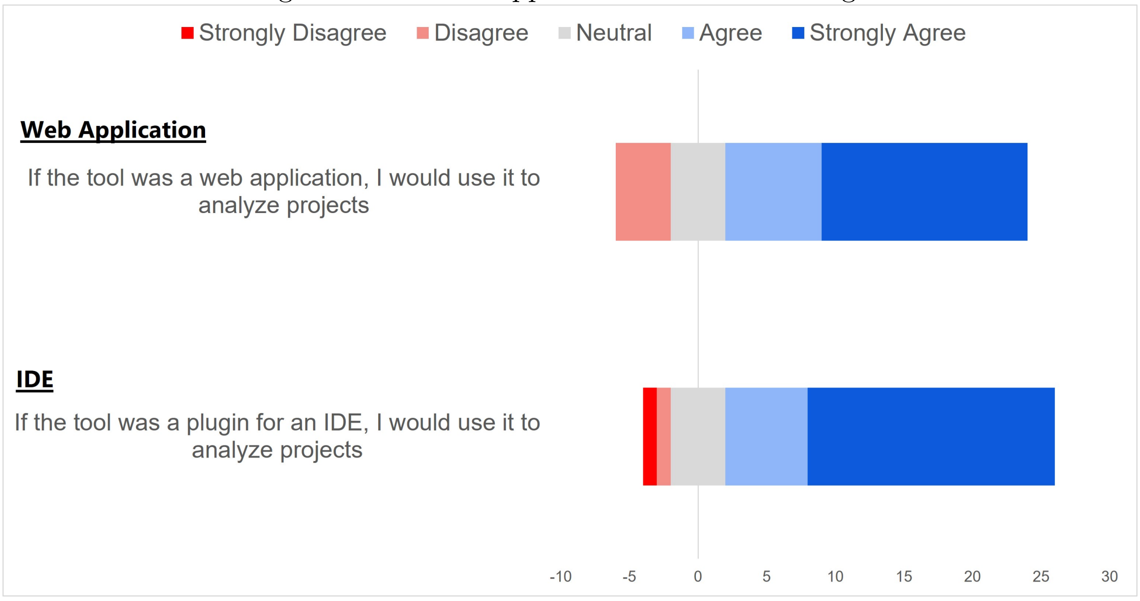


These results also correlate with how most participants from E1 felt. P6, from E1, stated: *"I once had to find every metadata-based framework reading the pom file. If I had access to something like AVisualizer, it would have been much easier".*

Furthermore, this result also correlates with our primary goals, presented in Section 5.1.1. Finding annotation schemas is strongly related to understanding how code annotations are distributed in the whole system, not just a specific class or package. Our primary goal, G1, is precisely concerned with this general distribution. Furthermore, is another evidence that the **System View** can reach that goal.

For the second category of questions, *Plugin IDE versus External Tool*, we wanted to know how the students feel about plugins embedded in IDEs (like IntelliJ and Eclipse). We made two statements, one considering that the AVisualizer tool was a finished product and available as a web application, which is actually how the students conducted the experiments. The second statement considered the AVisualizer tool a finished product available as a plugin for any IDE. Figure 5.24 presents the results using a diverging bar chart.

Figure 5.24 - Web Application versus IDE Plugin.



The results differ from what we obtained interviewing the developers in experiment E1. Although developers from E1 did recognize the advantages that our visualization tool/approach brings to analyzing code annotations, they are not very eager to access an external tool to analyze their source code. However, if a plugin were embedded in their IDE, they would feel much more interested in using it. As for the students, we conclude they did not feel that using an external tool is a problem. Furthermore, turning into a plugin for an IDE does not make much of a difference. Observing Figure 5.24 there are only slightly more *strongly agree* considering the tool as an IDE plugin a better option.

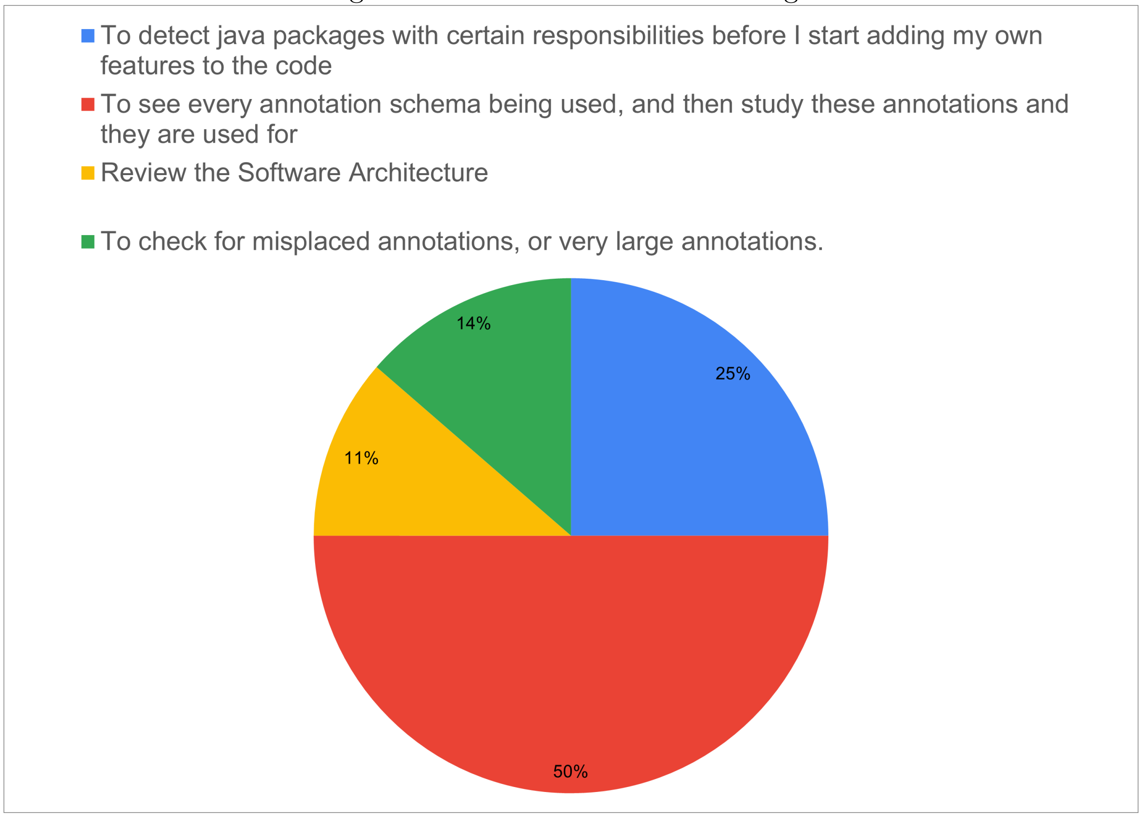
Finally, in our third category of questions, *Primary Usage*, we gave the students four possible scenarios and asked them in which one they would use the tool. Figure 5.25 display a pizza chart with the results.

As seen, 50% of the students would prefer to use the tool to detect what annotation schemas are present in the system. And then, they would seek other sources to learn what these annotation schemas are used for. They are not very interested in detecting misplaced annotations or analyzing the architecture of the system. From this result, we can assume that:

- The AVisualizer tool does not teach code annotations. If it did, the students could use the tool itself to learn about annotations.

- The AVisualizer helps detect what annotations (or annotations schemas) are being used in the system. Even though the students cannot use the tool to learn annotations, they can use the tool to guide what annotations they need to learn, i.e., the ones being used in the system they are working on

Figure 5.25 - Potential Scenario Usage.



Finally, we present some quotes from students answered in the final question about their overall opinions and impressions of the CADV and the AVisualizer tool. In this work, we did not perform a statistical analysis to describe how the AVisualizer tool impacted students with different code annotations familiarity or different primary programming languages. This analysis will be performed in future works. However, we did a simple sampling when selecting these students' quotes to obtain some variability in their familiarity with code annotations and the success rate of their answers in the closed-ended questions. Table 5.10 presents the demographics information of these selected students.

- (S1) *"The tool is easy to navigate e quite direct, and intuitive, something I thought was positive. However, suppose the user is not familiar with Java packages and schemas (for instance, the* `org.junit` *deals with unit tests). In that case, the idea of detecting the responsibilities of packages is not very useful. Perhaps further explanation of schemas would be nice inside the tool. A tooltip can help."* Very familiar, undergrad, typescript, a 10/10

- (S2) *"I thought it was a wonderful tool. I have just started developing professional Java systems and found the AVisualizer extremely intuitive to use. I find code annotations a very complex subject, and having a visualizer tool helps the overall learning process."*

- (S3) *"As a person with little experience in developing software projects, I was able to understand it quickly and found it very intuitive for anyone who needs such an application. The idea of combining different colors, captions, and geometric shapes with different sizes helped me to visualize code annotations and packages."*

- (S4) *"The tool itself makes it very easy to understand what class or package uses what annotations. Furthermore, it is straightforward to see where the largest annotations are and identify the classes which use them. For example, it was very easy for me to find which class is used for testing and which package is contained. The only problem I see is that it could become difficult for larger projects, which have many packages and classes, to navigate through them as these would be highly nested. With a high amount of elements, it becomes more difficult to analyze."*

- (S5) *"Neat idea. Extremely useful if your system heavily uses code annotations. However, I have noticed that it is still is very buggy. Traversing the project sometimes resulted in the graph multiplying and becoming unresponsive. Also, changing the view often resulted in an unusable application state. I often had to refresh the page to use the tool again."*

- (S6) *"The tool is easy to use, but sometimes I got confused if I was still in the same package when switching views. I had to check the Header to make sure where I was. Maybe animations would help. Also, a toggle option to hide annotation schemas that are not relevant in the current zoom level would be a nice feature since it is a bit confusing if you are zoomed in and have to search for the wanted annotation schema again. Otherwise,* *a great tool to get quick impressions about the project (assuming it uses annotations"*

- (S7) *"I would definitely use the tool in case I have to work on a system that heavily depends/uses annotations in order to get a better understanding of it and view the systems structure and its annotation usage"*

- (S8) *"The tool was easy-to-use, user-friendly, and intuitive. After watching the clear video tutorial, I managed to understand how it worked. I would use it to analyze a system, especially if I plan to add new code annotations"*

- (S9) *"I feel the tool is nice, but would be even better if it was able to give more details about the packages we are navigating and classify the code annotations (for instance, if it is for testing, or persistence)."*

Table 5.10 - Demographics Data For Selected Students.

Students	Familiarity with Annotations (1-5)	Primary Language	Questions Answered Correctly (%)
S1	4	TypeScript	100%
S2	2	Java	70%
S3	1	Python	70%
S4	2	Java	70%
S5	1	Java	70%
S6	2	Java	100%
S7	2	Java	100%
S8	1	Java	90%
S9	3	AdvPL	50%

Notice that even though most students in Table 5.10 did not consider themselves "very familiar" (4) or "expert" (5) with code annotations, they still managed to get 70% of the questions answered correctly, with only one student S9, getting 50% success rate. However, this student S9 is familiar with the AdvPL (Advanced Protheus Language), a language not widely popular as Python or Java.

As seen in the quotes, the students thought the AVisualzer tool was nice and could comprehend how code annotations were distributed. However, as with every software tool, there is always room for improvement.

5.3.3 Experiments summary and highlights

This section summarizes the results obtained from both E1 and E2 experiments and a discussion of our goals. While E1 was a recorded interview with six professional developers involved in the construction of the *SpaceWeatherTSI* software, E2 was conducted with 44 students asynchronously using a survey. In both experiments, the participants had to use the AVisualizer tool and answer some questions about code annotations distribution and usage of the *SpaceWeatherTSI* software.

To conduct the experiment E1, We prepared 15 questions to guide us. Afterward, we carried out a qualitative analysis of these interviews with the following findings:

- The interviewees found the **System View** very useful because they could quickly obtain knowledge about the general organization and code annotations distribution of the software being analyzed. The **System View** was the most used view by the interviewees, which demonstrates they felt more comfortable with it.
- The interviewees found the **Class View** more confusing because it is very close to code elements and it shows more details of a smaller part of the software being analyzed. They avoided exploring this view and were not confident in answering questions.
- Although the interviewees agree the tool is far superior to detect and inspect code annotations than code inspection, they would much rather have it as a plugin for their IDE or integrated into a CI/CD pipeline.
- Generally speaking, the interviewees enjoyed the CADV approach and the nested circle packing approach to represent the software system. However, the **Class View** should be redesigned to be more intuitive and less overwhelming.
- Concerning the AVisualizer tool, they pointed out several UI improvements that the tool needs to address. Moreover, more customization, such as configure the metric, choose the color for annotation schemas, and group annotation schemas.

The second experiment, E2, was carried with 44 students asynchronously. First, they had to use the AVisualizer tool and answer ten close-ended questions about code annotations usage and distribution of the *SpaceWeatherTSI* software. Afterward, they answered a total of 20 questions about their opinions and impressions of the AVisualizer tool as well as potential usage scenarios. They aimed to measure the "perceived ease of use" and "perceived usefulness". The first 19 questions used the Likert scale ranging from "strongly disagree" to "strongly agree". Finally, the last question was open to the students to describe their overall impression of the tool. In short, throughout experiment E2, the students answered 30 questions.

Findings from the ten close-ended questions with the students in E2:

- Questions with the highest success rates were related to the **System View**. In other words, questions about the general view and code annotations distributions in packages were easier to answer.
- Questions with the lowest success rates were related to the **Class View**. In other words, questions about code annotations inside a class or in specific code elements were harder to answer.

These findings are highly correlated to what we obtained in experiment E1, where interviewees felt much more comfortable answering questions that required using the **System View**.

Findings from the impressions and opinions of the students about the AVisualizer tool:

- Using the AVisualizer to detect code annotations is easier than purely inspecting the code.
- Students prefer to use the tool to search code annotations being used to learn about annotations and metadata-based frameworks.
- Students do not find that using IDE Plugins is a huge advance in using web applications or external tools.
- Students are not eager to use the tool to detect bad smells, errors, or misconfigurations.
- Students enjoyed the circle packing approach but suggested improvements on the tool UI.

We have evidence that the CADV could provide a software visualization that allowed developers to quickly detect annotation schemas and how they are grouped in the packages. Developers found it helpful to detect package responsibilities, potentially misplaced annotations, and general architecture being used. Students, on the other, found it very useful to detect annotation schemas being used on projects they are working on and further improve their knowledge of annotations and metadata configuration.

We have extensively discussed our goals throughout this section, but let us revisit them and present a brief final discussion.

(#G1) - **Detect annotations schemas and how they are distributed in the packages**: To reach this goal, we proposed the **System View**. The results of our experiments show that both students and developers enjoyed this view and were able to get a general view of how code annotations were distributed in the system.

(#G2) - **Detect how annotations are distributed per class in packages**: To reach this goal, we proposed the **Package View**. The results of our experiments show similar results when compared to the **System View**. Both students and developers were able to visualize code annotations usage and distribution inside a specific package.

(#G3) - **Detect how annotations are distributed and grouped per code elements inside the classes**: To reach this goal, we proposed the **Class View**. The results of our experiments show this view was not as effective compared to the **System View** or **Package View**. However, in the worst scenario, 56% of the students were able to inspect a class and detect code annotations characteristics. Although not as successful as 90% when compared to the **System View**, it can be improved with further training and UI labels and tooltips.

(#G4) - **Provide a navigation system between views with different granularity**: The AVisualizer tool was developed with this feature, enabling users to navigate between all three views. From the results, after proper training, the users were able to navigate between views.

(#G5) - **Detect misconfigurations**: Although this was not a primary goal, during experiment E1, we asked participants about two specific problems: Misplaced and extensive code annotations. For the first, the **System View** was shown to be very useful, and our color-based strategy to identify schemas was handy to spot potentially misplaced code annotations. Concerning extensive annotations, although the interviewees saw this as simply detecting “large circles”, they were not particularly worried about this problem. We argue that future experiments and a proper study should be carried to investigate code annotations bad smells.

6 CONCLUSION

Code annotations is a Java language feature used by several frameworks and tools. Its usage is widespread, especially in web and enterprise applications. Despite this, there are only superficial studies in the literature about design problems and the assessment of annotated code. Before our work, to the best of our knowledge, there was no suite of software metrics and no visualization approach dedicated to code annotations.

We began this work according to the goal *to define an approach to measure and visualize code annotations* ***to assess and comprehend*** *their usage and distribution in software systems.* From this goal, we divided the work into two parts. The first focused on measuring code annotations, where we proposed our novel suite of software metrics. The second part focused on visualizing code annotations, where we defined our CADV approach, a polymetric view based on circle packing.

Software developers, researchers, and practitioners at INPE can use our open-source tool ASniffer to measure code annotations, monitor these metrics values in their systems, and be aware of any abnormal growth. They can also use our visualization tool AVisualizer to comprehend the system's architecture, internal structure and even detect potential misplaced or misconfigured code annotations. To reinforce the application of our approach to internal demands for INPE, we used the **SpaceWeather** software system as a target for both of our experiments E1 and E2. Furthermore, we conducted interviews with developers from this system to highlight the importance of developing software engineering tools and approaches to support the evolution and maintenance of software systems developed by INPE.

6.1 Code annotations measurement concluding remarks

To define and evaluate the novel suite of metrics, we proposed four research questions. From them, we elaborated on the five steps required to answer these questions. Initially, a suite of candidate metrics was proposed based on a GQM approach. These metrics provide values that measure how annotations are present in the source code and their characteristics. They are a new group of metrics used exclusively for annotations in the source code. A sample of 24,947 Java classes extracted from 25 real-world projects was analyzed. These projects contain a wide range of annotated classes to provide diversity in our analysis. We generated a percentile rank for all metric values to understand their behavior and identify possible threshold values. It was pointed out that all metrics have an exponential distribution, which indicates that the average and standard deviation might not be a good representation of the data. Some metrics, such as ANL, have small overall values, and the average value could be considered a reference point, but metrics such as AC have an abrupt growth at percentile 90. Thus, the average value is not a good middle point.

To determine threshold values, we used the Percentile Rank Analysis based on Meirelles' findings (MEIRELLES, 2013) and compared it to Lanza's approach (LANZA; MARINESCU, 2006), which defines three threshold values: low, medium, and high. We also defined three threshold values: very frequent, frequent, and less frequent. Lanza's thresholds are obtained through the average value, which directly yields the medium point. Our "very frequent" was obtained by analyzing the percentile rank, and for some metrics, the medium point and "very frequent" values are close to each other. However, when determining the "high value" (Lanza's) and "less frequent" value (our approach), the percentile rank analysis provides a more realistic value, usually higher than Lanza's "high value". That is to say, the percentile rank analysis considers a broader range of values, and they can still be considered common values. These values might help to indicate a potential misleading in annotations usage.

Our analysis showed that most of the classes have low values for all of the metrics. However, we have found some outliers with really high values for some candidate metrics. These extreme cases reinforce the need to evaluate and further improve the techniques to study code annotations. Before the proposed metrics suite, there was no suitable way to measure code annotations, and therefore we would not have identified these extreme cases. Our work was not focused on further investigating outliers or bad smells, but with the metrics and threshold values available, future work can be used for such goals.

6.2 Code annotations visualization concluding remarks

To propose the CADV approach, we defined five goals that we wanted to reach with our visualization. From these goals, we elaborated a GQM model that led to three views that compose the CADV. They are the **System View**, *Package View*, and *Class View*. All three are a polymetric view that uses a nested circle packing strategy to represent the analyzed software. The *System View* displays packages and annotation schemas used in them. The *Package View* displays classes and annotations used inside the package being analyzed. Finally, the *Class View* displays code elements and how code annotations are grouped. The size of the circles is determined by code annotation metrics values that we extract from the software that we wish to analyze. These metrics belong to our novel suite previously discussed.

To demonstrate our CADV approach, we developed an open-source tool named AVisualizer. This tool implements the three views of the CADV and a navigation system that allows the user to switch between them. The AVisualizer uses the ASniffer as a dependency to generate the metrics values of the project the user wishes to analyze. We did not want potential users to manually generate the metrics and use the report as input to the AVisualizer. Hence, we distribute it as a single package. The use of the ASniffer in the background is entirely transparent to the user.

Using the *SpaceWeatherTSI* as the target software, we organized two different experiments to validate the CADV approach. This software is a module of the *SpaceWeather* web application that belongs to the EMBRACE division of INPE. We chose the *SpaceWeatherTSI* module because it uses several metadata-based frameworks, and therefore, many annotations are available to be visualized.

The first experiment, E1, was carried out by interviewing six developers of *SpaceWeatherTSI*. We prepared 15 questions to guide us, but the interview was carried informally, and the participants were free to explore the AVisualizer tool providing as much information as possible to use. The second experiment, E2, was carried with 44 students asynchronously using a survey. They had to use the AVisualizer tool and answer ten close-ended questions about code annotations usage and distribution of the *SpaceWeatherTSI* software. Afterward, they answered 20 questions about their opinions and impressions of the AVisualizer tool and potential usage scenarios. We performed a qualitative and quantitative analysis using these two experiments, interviews (E1) and surveys (E2). Furthermore, we could assess the CADV from the point of view of two different target audiences, i.e., professional developers and students.

From our findings, in both E1 and E2, participants felt the CADV approach was handy to quickly detect annotation schemas and how they are distributed in packages. In other words, they felt more comfortable using the *System View*. On the other hand, the CADV did not reach the same success rate in E1 and E2. The issue was related to the amount of information being displayed to the user. We also found that students prefer to use the tool to detect and learn about annotations.

6.3 Threats to validity

To keep our threats analysis clean and following the same pattern we have been using to write this work, we will also separate the threats into two subsections.

6.3.1 Code annotation metrics threats

As a threat to the obtained results, the ASniffer accuracy was done manually, checking if some obtained metric values were correct since no other similar tool was found to compare the results. Therefore, this can potentially compromise the ASniffer accuracy. Since it is open-source, other researchers and practitioners can also improve the tool, further minimizing these threats.

The idea behind the selected 25 real-world Java projects was to combine different domains and different annotation usage. To validate our sample data, we used the concept of diversity and similarity, using dimensions. We defined three dimensions: Type, LOC, and Percentage of Annotated Classes (PAC). We combined these three dimensions in a pairwise fashion, and for each combination, we had at least two projects and a maximum of 8. So, we guarantee diversity and similarity among our chosen projects. However, the list of projects is not easily reproduced, bringing variability in future research and threshold values. One of the reasons for this is that the defined dimensions were used to validate the chosen projects. Instead, they could have been used to choose the projects from a more expansive universe.

6.3.2 Code annotation visualization threats

A significant threat to the CADV approach is the heavy use of colors, severely impacting its usage by colorblind users. Currently, the approach revolves around colors to identify annotation schemas, being a core of the CADV approach. Our research team has not developed a road map to address this issue. However, we are considering using different shapes or symbols to differentiate annotation schemas. Furthermore, the hardware being used to display the CADV can also vary from user to user. Different types of displays and GPUs can cause the color tones to become different, impacting the use of a color-based visualization. We tried to minimize this using labels. As mentioned, when users hover the mouse over annotations (colored circles), a label appears and displays textual information. It is not as direct as the color approach, but it does minimize this threat.

For software visualization, we argue that this written thesis is also a threat to the interpretation and complete comprehension of the CADV. Our approach is visual and interactive, and it is not trivial to represent this complex system in a static medium, such as this written text one. We minimize this threat by also complementing this text with a recorded video.

The AVisualizer tool, the first implementation of the CADV approach, is ongoing work and requires several improvements in the UI and labels. This could have impacted and compromised our experiments because participants might incorrectly answer a question due to a lack of better-designed labels to guide the usage.

Both experiments, E1 and E2, were conducted remotely due to the COVID-19 Pandemic, which could have also compromised our results. For instance, the training session was conducted asynchronously through a previously recorded video, and the participants might come across potential doubts. For experiment E1, this was mitigated during the interview itself, but for participants in E2, this was not appropriately addressed. Also, participants from E1 were part of the *SpaceWeather* development team, so they were already familiar with the system. In contrast, participants from E2 were students, and it was their first contact with any software from the *SpaceWeather* project. Hence, this represents an advantage that E1 participants had.

The project *SpaceWeatherTSI* was chosen due to the variety of metadata-based frameworks being used, but also to validate our work as applied research to INPE internal demands. A careful project selection should be used to validate the CADV in other systems, especially open-source ones, from a software engineering perspective.

Finally, we had to choose metrics from our suite that we thought would best represent the system. For the *System View* this is less critical since we are not interested in the size or number of arguments for a specific annotation. However, depending on the chosen metrics and the target software, the *Package View* and *Class View* can change completely.

6.4 Future work

Future studies can use this work as the foundation to enable assessment and evolution of annotation usage through the annotation metrics suite and threshold values calculated in Chapter 4. For instance, frameworks can be improved by searching for bad smells related to their annotations in applications using them. The proposed techniques and the ASniffer tool can be used on real case studies, analyzing the impact of an improved annotation structure on application maintenance. Since the ASniffer is an open-source tool, other developers can improve it in several ways: adding new metrics, implementing bad smell detection, and adding visualization techniques.

When calculating threshold values for the metrics, we found that classes with very high values for annotation metrics exist in real projects. We consider the detection of these outliers as evidence that a deeper investigation of this issue is essential.

As for the CADV approach and the AVisualizer tool, there is much we can explore. The CADV was the first visualization technique proposed for the metrics, and as we already mentioned, other techniques can also be proposed. As for our CADV, the first step is to redesign the *Class View* in a way that is not confusing and does not overwhelm potential users. Furthermore, we will improve the UI, add labels, and create a version of the AVisualizer as a plugin for the IntelliJ IDE. We chose this last one as it has become increasingly popular with Java developers.

We have also done a small work with C# attributes. However, we would like to perform a deeper investigation of the metrics and CADV approach for the C# and Kotlin, which also use code annotations and are languages very similar to Java. Currently, the EMBRACE project is also developing solutions for mobile devices using the Kotlin programming language. Therefore, it makes sense to take our work and span to this environment from an applied research perspective.

REFERENCES

ALBA, A. **Análise de legibilidade de código usando padrões de anotações**. Especialização em Engenharia de Software — Instituto Técnológico de Aeronáutica, São José dos Campos, 2011. 18

BASILI, V. R. **Software modeling and measurement: the goal/question/metric paradigm**. College Park, MD, USA, 1992. 4, 45, 46, 76

BASILI, V. R.; CALDIERA, G.; ROMBACH, D. H. The goal question metric approach. **Encyclopedia of Software Engineering**, John Wiley & Sons, I, 1994. 4, 45, 46, 76

BAXTER, G.; FREAN, M.; NOBLE, J.; RICKERBY, M.; SMITH, H.; VISSER, M.; MELTON, H.; TEMPERO, E. Understanding the shape of java software. In: ANNUAL ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING SYSTEM, LANGUAGES, AND APPLICATIONS, 21., 2006. **Proceedings...** New York: ACM, 2006. p. 397–412. ISBN 1-59593-348-4. 19, 20

BOSTOCK, M. **Circle packing**. dec. 2017. Available from: <`https://observablehq.com/@d3/circle-packing`>. 82

BRAGA, L.; LIMA, P.; GUERRA, E.; MEIRELLES, P. Attribute sniffer: collecting attribute metrics for c# code. In: CONGRESSO BRASILEIRO DE SOFTWARE: TEORIA E PRÁTICA, 10., 2019. **Anais...** Porto Alegre: SBC, 2019. p. 96–101. ISSN 2177-9384. 3, 9

BRIAND, L. C.; MORASCA, S.; BASILI, V. R. Property-based software engineering measurement. **IEEE Transactions on Software Engineering**, v. 22, n. 1, p. 68–86, Jan 1996. ISSN 0098-5589. 60, 61, 62

BROOKS, F. P. No silver bullet essence and accidents of software engineering. **Computer**, v. 20, n. 4, p. 10–19, 1987. 21

CHEN, N. **Convention over configuration**. 2006. Available from: <`http://softwareengineering.vazexqi.com/files/pattern.html`>. Access in: dez 2018. 12

CHIDAMBER, S. R.; KEMERER, C. F. Towards a metrics suite for object-oriented design. In: ANNUAL ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING SYSTEM, LANGUAGES, AND APPLICATIONS, 6., 1991. **Proceedings...** New York: ACM, 1991. p. 197–211. 19

CHOMA, J.; GUERRA, E.; SILVA, T. S. da; ZAINA, L. A. M.; CORREIA, F. F. Towards an artifact to support agile teams in software analytics activities. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING AND KNOWLEDGE, 31., 2019, Lisbon, Portugal. **Proceedings...** Pittsburgh: KSI, 2019. p. 88–122. 79

CLAUSET, A.; SHALIZI, C. R.; NEWMAN, M. E. J. Power-law distributions in empirical data. **SIAM Reviews**, jun. 2007. Available from: <http://arxiv.org/abs/0706.1062>. 19

CONCAS, G.; MARCHESI, M.; PINNA, S.; SERRA, N. Power-laws in a large object-oriented software system. **IEEE Transaction of Software Engineering**, Piscataway, NJ, USA, v. 33, n. 10, p. 687–708, october 2007. 20

CÓRDOBA-SÁNCHEZ, I.; LARA, J. de. Ann: a domain-specific language for the effective design and validation of java annotations. **Computer Languages, Systems & Structures**, v. 45, p. 164 – 190, 2016. ISSN 1477-8424. Available from: <http://www.sciencedirect.com/science/article/pii/S1477842416300318>. 13

DAGENAIS, B.; OSSHER, H.; BELLAMY, R. K. E.; ROBILLARD, M. P.; DE VRIES, J. P. Moving into a new software project landscape. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 32., 2010. **Proceedings...** New York: ACM, 2010. p. 275–284. 21, 77

DAMYANOV, I.; HOLMES, N. Metadata driven code generation using .net framework. In: INTERNATIONAL CONFERENCE ON COMPUTER SYSTEMS AND TECHNOLOGIES, 5., 2004. **Proceedings...** New York: ACM, 2004. p. 1–6. 11, 34

DAVIS, F. D. Perceived usefulness, perceived ease of use, and user acceptance of information technology. **MIS Quarterly**, v. 13, n. 3, p. 319–340, 1989. ISSN 02767783. Available from: <http://www.jstor.org/stable/249008>. 76, 79

DIEHL, S. **Software visualization: visualizing the structure, behaviour, and evolution of software**. Berlin, Heidelberg: Springer-Verlag, 2007. ISBN 3540465049. 21

ECMA. **ECMA - 334: C# language specification**. dec. 2017. Available from: <https://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf>. 3, 12

ERNST, M. D. **Type annotations specification (JSR 308)**. [S.l.]: November, 2008. 11, 34

ERRA, U.; SCANNIELLO, G. Towards the visualization of software systems as 3d forests: the codetrees environment. In: ANNUAL ACM SYMPOSIUM ON APPLIED COMPUTING, 27., 2012. **Proceedings...** New York: ACM, 2012. p. 981–988. ISBN 978-1-4503-0857-1. Available from: <http://doi.acm.org/10.1145/2245276.2245467>. 2, 25

FALESSI, D.; JURISTO, N.; WOHLIN, C.; TURHAN, B.; MÜNCH, J.; JEDLITSCHKA, A.; OIVO, M. Empirical software engineering experts on the use of students and professionals in experiments. **Empirical Software Engineering**, v. 23, n. 1, p. 452–489, 2018. Available from: <https://doi.org/10.1007/s10664-017-9523-3>. 79

FERNANDES, C.; RIBEIRO, D.; GUERRA, E.; NAKAO, E. Xml, annotations and database: a comparative study of metadata definition strategies for frameworks. In: SIMÕES, A.; CRUZ, D.; RAMALHO, J. C. (Ed.). **XATA 2010**. Instituto Politécnico do Porto, 2010. p. 115. Available from: <https://recipp.ipp.pt/handle/10400.22/7647>. 12

FERREIRA, K. A. M.; BIGONHA, M. A. S.; BIGONHA, R. S.; MENDES, L.; ALMEIDA, H. C. Reference values for object-oriented software metrics. In: BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING, 23., 2009. **Proceedings...** New Jersey: IEEE, 2009. v. 1, p. 62–72. 20

FOWLER, M. **Refactoring: improving the design of existing code**. Boston: Addison-Wesley Professional, 1999. 70

FRANCESE, R.; RISI, M.; SCANNIELLE, G.; TORTORA, G. Proposing and assessing a software visualization approach based on polymetric views. **Journal of Visual Languages and Computing**, v. 34, n. C, p. 11–24, jun. 2016. ISSN 1045-926X. Available from: <https://doi.org/10.1016/j.jvlc.2016.05.001>. 2, 21, 22, 29, 30, 31, 73, 74, 76

FRANCESE, R.; RISI, M.; SCANNIELLO, G.; TORTORA, G. Viewing object-oriented software with metricattitude: an empirical evaluation. In: INTERNATIONAL CONFERENCE ON INFORMATION VISUALIZATION, 18., 2014. **Proceedings...** New Jersey: IEEE, 2014. p. 59–64. 30

GRADY, D. R. C. R. B. **Software metrics: establishing a company-wide program**. Hoboken, New Jersey: Prentice-Hall, 1987. 52

GRAHAM, H.; YANG, H. Y.; BERRIGAN, R. A solar system metaphor for 3d visualisation of object oriented software metrics. In: AUSTRALASIAN SYMPOSIUM ON INFORMATION VISUALIZATION, 32., 2004. **Proceedings...** Sydney: ACS, 2004. v. 35, p. 53–59. ISBN 1920682171. 25

GREENE, G. J.; ESTERHUIZEN, M.; FISCHER, B. Visualizing and exploring software version control repositories using interactive tag clouds over formal concept lattices. **Information and Software Technology**, v. 87, p. 223–241, 2017. ISSN 0950-5849. Available from: <https://www.sciencedirect.com/science/article/pii/S0950584916304050>. 21

GUERRA, E. **Componentes reutilizáveis em Java com reflexão e anotações**. São Paulo: Casa do Código, 2014. ISBN 978-85-66250-50-3. 1, 11

GUERRA, E.; FERNANDES, C. A qualitative and quantitative analysis on metadata-based frameworks usage. In: ______. **Computational science and its applications – ICCSA**. Berlin, Heidelberg: Springer, 2013. p. 375–390. ISBN 978-3-642-39643-4. Available from: <http://dx.doi.org/10.1007/978-3-642-39643-4_28>. 16, 17

GUERRA, E.; LIMA, P.; CHOMA, J.; NARDES, M.; SILVA, T.; LANZA, M.; MEIRELLES, P. A metadata handling api for framework development: a comparative study. In: BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING, 34., 2020. **Proceedings...** New York: ACM, 2020. p. 499–508. ISBN 9781450387538. Available from: <https://doi.org/10.1145/3422392.3422428>. 9, 43

GUERRA, E.; SOUZA, J. de; FERNANDES, C. Pattern language for the internal structure of metadata-based frameworks. In: ______. **Transactions on pattern languages of programming**. Berlin, Heidelberg: Springer, 2013. p. 55–110. ISBN 978-3-642-38676-3. Available from: <http://dx.doi.org/10.1007/978-3-642-38676-3_3>. 17

GUERRA, E. M.; SILVEIRA, F. F.; FERNANDES, C. T. Questioning traditional metrics for applications which uses metadata-based frameworks. In: WORKSHOP ON ASSESSMENT OF CONTEMPORARY MODULARIZATION TECHNIQUES, 3., 2009. **Proceedings...** New Jersey: IEEE, 2009. v. 26, p. 35–39. 1

GUERRA, E. M.; SOUZA, J. T. de; FERNANDES, C. T. A pattern language for metadata-based frameworks. In: CONFERENCE ON PATTERN LANGUAGES

OF PROGRAMS, 16., 2010. **Proceedings...** New York: ACM, 2010. ISBN 978-1-60558-873-5. Available from: <http://doi.acm.org/10.1145/1943226.1943230>. 11, 34

HASSELBRING, W.; KRAUSE, A.; ZIRKELBACH, C. Explorviz: research on software visualization, comprehension and collaboration. **Software Impacts**, v. 6, p. 100034, 2020. ISSN 2665-9638. Available from: <https://www.sciencedirect.com/science/article/pii/S2665963820300257>. 2, 21, 77

HERRAIZ, I.; GERMÁN, D. M.; HASSAN, A. E. On the distribution of source code file sizes. In: INTERNATIONAL CONFERENCE ON SOFTWARE AND DATA TECHNOLOGIES, 6., 2011. **Proceedings...** Setúbal: Scite Press, 2011. p. 5–14. ISBN 978-989-8425-77-5. 20

HERRAIZ, I.; RODRIGUEZ, D.; HARRISON, R. On the statistical distribution of object-oriented system properties. In: INTERNATIONAL WORKSHOP ON EMERGING TRENDS IN SOTWARE METRICS, 3., 2012. **Proceedings...** New Jersey: IEEE, 2012. p. 56–62. 20

HÖST, M.; REGNELL, B.; WOHLIN, C. Using students as subjects—a comparative study ofstudents and professionals in lead-time impact assessment. **Empirical Software Engineering**, v. 5, n. 3, p. 201–214, nov. 2000. ISSN 1382-3256. Available from: <https://doi.org/10.1023/A:1026586415054>. 79

JSR. **JSR 175: A metadata facility for the Java programming language**. aug. 2004. Available from: <http://www.jcp.org/en/jsr/detail?id=175>. 12, 13

______. **JSR 220: enterprise JavaBeans 3.0**. aug. 2007. Available from: <http://jcp.org/en/jsr/detail?id=220>. 13

KNIGHT, C.; MUNRO, M. Virtual but visible software. In: CONFERENCE ON INFORMATION VISUALIZATION, 4., 2000. **Proceedings...** New Jersey: IEEE, 2000. p. 198–205. 22

KRAHN, H.; RUMPE, B. Towards enabling architectural refactorings through source code annotations. **Lecture Notes in Informatics**, P-82, p. 203–212, 2006. 7, 16

LANZA, M. Codecrawler - polymetric views in action. In: INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING, 19., 2004. **Proceedings...** New Jersey: IEEE, 2004. p. 394–395. 28

LANZA, M.; DUCASSE, S. Polymetric views-a lightweight visual approach to reverse engineering. **IEEE Transactions on Software Engineering**, v. 29, n. 9, p. 782–795, sep. 2003. ISSN 0098-5589. Available from: <https://doi.org/10.1109/TSE.2003.1232284>. 2, 26, 27, 29, 76, 85

LANZA, M.; MARINESCU, R. **Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems**. Heidelberg: Springer, 2006. 17, 19, 20, 51, 70, 140

LIMA, P.; GUERRA, E.; MEIRELLES, P. Definição de clusters para classificação do uso de anotações em código java. In: WORKSHOP ON SOFTWARE VISUALIZATION, EVOLUTION AND MAINTENANCE, 5., 2017. **Proceedings...** Porto Alegre: SBC, 2017. 9, 42

______. Historical analysis of code annotations. In: BRAZILIAN CONFERENCE ON SOFTWARE, 9., 2018, São Carlos. **Proceedings...** Porto Alegre, Brasil: SBC, 2018. 3, 7

______. Annotation sniffer: a tool to extract code annotations metrics. **Journal of Open Source Software**, v. 5, n. 47, p. 1960, 2020. Available from: <https://doi.org/10.21105/joss.01960>. 5, 8, 37, 47, 76

______. Towards visualizing code annotations distribution. **Computer on the Beach**, v. 11, n. 1, p. 277–284, 2020. ISSN 2358-0852. Available from: <https://siaiap32.univali.br/seer/index.php/acotb/article/view/16779>. 8, 31, 34, 35, 82

LIMA, P.; GUERRA, E.; MEIRELLES, P.; KANASHIRO, L.; SILVA, H.; SILVEIRA, F. A metrics suite for code annotation assessment. **Journal of Systems and Software**, v. 137, p. 163 – 183, 2018. ISSN 0164-1212. Available from: <http://www.sciencedirect.com/science/article/pii/S016412121730273X>. 1, 5, 8, 10, 16, 37, 44, 45, 71, 76, 81, 110

LIMA, P.; GUERRA, E.; NARDES, M.; MOCCI, A.; BAVOTA, G.; LANZA, M. An annotation-based api for supporting runtime code annotation reading. In: INTERNATIONAL WORKSHOP ON META-PROGRAMMING TECHNIQUES

AND REFLECTION, 2., 2017. **Proceedings...** New York: ACM, 2017. ISBN 9781450355230. Available from: <https://doi.org/10.1145/3141517.3141856>. 9, 42, 44

LOMBOK, P. **Project Lombok**. Available from: <https://projectlombok.org/>. Access in: jul 2018. 11, 34

LOURIDAS, P.; SPINELLIS, D.; VLACHOS, V. Power laws in software. **ACM Transactions on Software Engineering and Methodology**, ACM, New York, v. 18, n. 1, p. 2:1–2:26, oct. 2008. ISSN 1049-331X. Available from: <http://doi.acm.org/10.1145/1391984.1391986>. 20

MEFFERT, K. Supporting design patterns with annotations. In: INTERNATIONAL SYMPOSIUM AND WORKSHOP ON ENGINEERING OF COMPUTER BASED SYSTEMS, 13., 2006. **Proceedings...** New Jersey: IEEE, 2006. 7, 16

MEIRELLES, P. R. M. **Monitoring source code metrics in free software projects**. PhD Thesis (PhD) — University of São Paulo, São Paulo, 2013. Available from: <http://www.teses.usp.br/teses/disponiveis/45/45134/tde-27082013-090242/pt-br.php>. 5, 46, 51, 52, 67, 140

MERINO, L.; GHAFARI, M.; ANSLOW, C.; NIERSTRASZ, O. A systematic literature review of software visualization evaluation. **Journal of Systems and Software**, v. 144, p. 165–180, 2018. ISSN 0164-1212. Available from: <https://www.sciencedirect.com/science/article/pii/S0164121218301237>. 2, 5, 73, 76, 77, 78

NAGAPPAN, M.; ZIMMERMANN, T.; BIRD, C. Diversity in software engineering research. In: JOINT MEETING ON FOUNDATIONS OF SOFTWARE ENGINEERING, 9., 2013, Saint Petersburg, Russia. **Proceedings...** New York: ACM, 2013. p. 466–476. ISBN 978-1-4503-2237-9. Available from: <http://doi.acm.org/10.1145/2491411.2491415>. 5, 49

NYSTROM, R. **Game programming patterns**. Lexington: Genever Benning, 2014. ISBN 9780990582908. 34

OGHENEOVO, E. On the relationship between software complexity and maintenance costs. **Journal of Computer and Communications**, v. 2, p. 1–16, 2014. ISSN 2327-5227. Available from: <https://www.scirp.org/journal/paperinformation.aspx?paperid=51631>. 55

POTANIN, A.; NOBLE, J.; FREAN, M.; BIDDLE, R. Scale-free geometry in oo programs. **Communications of the ACM**, v. 48, n. 5, p. 99–103, may 2005. ISSN 0001-0782. 20

QUINONEZ, J.; TSCHANTZ, M.; ERNST, M. Inference of reference immutability. In: VITEK, J. (Ed.). **ECOOP 2008: object-oriented programming**. Berlin: Springer, 2008. p. 616–641. Available from: <http://www.springerlink.com/index/6M5U5M330T81763T.pdf>. 11, 34

RAJLICH, V. Software evolution and maintenance. In: FUTURE OF SOFTWARE ENGINEERING, 2014. **Proceedings...** New York: ACM, 2014. p. 133–144. ISBN 978-1-4503-2865-4. Available from: <http://doi.acm.org/10.1145/2593882.2593893>. 2, 21

ROCHA, H.; VALENTE, H. How annotations are used in java: An empirical study. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING AND KNOWLEDGE, 23., 2011. **Proceedings...** Pittsburgh: KSI, 2011. p. 426–431. 3, 17

ROMANO, S.; CAPECE, N.; ERRA, U.; SCANNIELLO, G.; LANZA, M. On the use of virtual reality in software visualization: the case of the city metaphor. **Information and Software Technology**, v. 114, p. 92 – 106, 2019. ISSN 0950-5849. Available from: <http://www.sciencedirect.com/science/article/pii/S0950584919301405>. 2, 21, 25, 26, 73, 77

ROUVOY, R.; PESSEMIER, N.; PAWLAK, R.; MERLE, P. Using attribute-oriented programming to leverage fractal-based developments. In: INTERNATIONAL ECOOP WORKSHOP ON THE FRACTAL COMPONENT MODEL, 5., 2006. **Proceedings...** Heidelberg: Springer, 2006. 16

RUBY, S.; THOMAS, D.; HANSSON, D. **Agile web development with rails**. 3. ed. Springfield: Pragmatic Bookshelf, 2009. ISBN 1934356166, 9781934356166. 12

SANT'ANNA, N.; GUERRA, E.; IVO, A.; PEREIRA, F.; MORAES, M.; GOMES, V.; VERAS, L. G. Modelo arquitetural para coleta, processamento e visualização de informações de clima espacial. In: SIMPÓSIO BRASILIEIRO DE SISTEMAS DE INFORMAÇÃO, 10., 2014. **Anais...** Porto Alegre, RS, Brasil: SBC, 2014. p. 125–136. ISSN 0000-0000. Available from: <https://sol.sbc.org.br/index.php/sbsi/article/view/6107>. 3

SCHWARZ, D. **Peeking inside the box: attribute-oriented programming with Java 1.5, Part**. 2004. Available from: <http://archive.oreilly.com/pub/a/onjava/2004/06/30/insidebox1.html>. 13

STEPHENSON, K.; CANNON, J.; FLOYD, W.; PARRY, W. Introduction to circle packing: the theory of discrete analytic functions. **The Mathematical Intelligencer**, v. 29, p. 63–66, 01 2007. 82, 83

SUN, X.; LI, B.; LEUNG, H.; LI, B.; LI, Y. Msr4sm: using topic models to effectively mining software repositories for software maintenance tasks. **Information and Software Technology**, v. 66, p. 1 – 12, 2015. ISSN 0950-5849. Available from: <http://www.sciencedirect.com/science/article/pii/S0950584915001007>. 21

TEIXEIRA, R.; GUERRA, E.; LIMA, P.; MEIRELLES, P.; KON, F. Does it make sense to have application-specific code conventions as a complementary approach to code annotations? In: INTERNATIONAL WORKSHOP ON META-PROGRAMMING TECHNIQUES AND REFLECTION, 3., 2018. **Proceedings...** New York: ACM, 2018. p. 15–22. ISBN 9781450360685. Available from: <https://doi.org/10.1145/3281074.3281078>. 9, 43

TEMPERO, E.; ANSLOW, C.; DIETRICH, J.; HAN, T.; LI, J.; LUMPE, M.; MELTON, H.; NOBLE, J. The qualitas corpus: a curated collection of java code for empirical studies. In: ASIA PACIFIC SOFTWARE ENGINEERING CONFERENCE, 17., 2010. **Proceedings...** New Jersey: IEEE, 2010. p. 336–345. ISSN 1530-1362. 3, 17

WADA, H.; SUZUKI, J. Modeling turnpike frontend system: A model-driven development framework leveraging uml metamodeling and attribute-oriented programming. **Model Driven Engineering Languages and Systems**, p. 584–600, 2005. Available from: <http://www.springerlink.com/index/l166363337837142.pdf>. 13

WETTEL, R.; LANZA, M. Program comprehension through software habitability. In: INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION, 15., 2007. **Proceedings...** New Jersey: IEEE, 2007. p. 231–240. 21, 22, 24

______. Visualizing software systems as cities. In: INTERNATIONAL WORKSHOP ON VISUALIZING SOFTWARE FOR UNDERSTANDING AND ANALYSIS, 4., 2007. **Proceedings...** New Jersey: IEEE, 2007. p. 92–99. 22, 23

WETTEL, R.; LANZA, M.; ROBBES, R. Software systems as cities: a controlled experiment. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 33., 2011. **Proceedings...** New York: ACM, 2011. p. 551–560. 2, 23, 24, 25, 34

WHEELDON, R.; COUNSELL, S. Power law distributions in class relationships. In: INTERNATIONAL WORKSHOP ON SOURCE CODE ANALYSIS AND MANIPULATION, 3., 2003. **Proceedings...** New Jersey: IEEE, 2003. p. 45–54. 19, 20

YAO, Y.; HUANG, S.; REN, Z.-p.; LIU, X.-m. Scale-free property in large scale object-oriented software and its significance on software engineering. In: INTERNATIONAL CONFERENCE ON INFORMATION AND COMPUTING SCIENCE, 2., 2009. **Proceedings...** New Jersey: IEEE, 2009. p. 401–404. 20

YU, Z.; BAI, C.; SEINTURIER, L.; Monperrus, M. Characterizing the usage, evolution and impact of java annotations in practice. **IEEE Transactions on Software Engineering**, 2019. 18, 75

APPENDIX A - THRESHOLD VALUES FOR CODE ANNOTATION METRICS

In this appendix, we present the obtained threshold values, with a complete table and figure for the remaining six metrics that were not presented in Chapter 4. In other words, we present the calculations for the metrics: AA, LOCAD, ANL, AED, UAC, and ASC.

A.1 Arguments in Annotations (AA)

The Arguments in Annotations (AA) metric measures how many arguments are present in a specific annotation declaration. Table A.1 shows the AA percentile values for all projects analyzed.

Analyzing Table A.1, we notice that below the percentile 90, for most projects, the metric has a value of 0. From the percentile 90, it starts to manifest and may reach the value 4. This result shows that only 10% of the data is useful for analysis, but the overall values are still low. Therefore, the average value may bring information to characterize the data. Table A.2 shows our threshold values compared to the thresholds obtained by using Lanza's approach.

Our analysis shows that having 1 argument declared in an annotation can be considered a reliable average value and not 0 (as obtained by Lanza's approach). Annotations containing more than 2 arguments are "less frequent", and values greater than this might reveal uncommon scenarios.

Figure A.1 presents an example of an outlier and illustrates an AA metric percentile rank chart. From our sample, the greatest value found was 9 for project Apache Tomcat. The `ParamServlet` class has the annotation `@WebServlet` with all possible arguments configured. The annotation is even more complicated because 1 of its arguments contains 2 other nested annotations.

Table A.1 - Percentiles from AA metric in all projects.

Projects	X5.	X10.	X25.	X50.	X75.	X90.	X95.	X99.	X100.	mean	std
Agilefant	0.00	0.00	0.00	0.00	1.00	1.00	1.00	2.00	4.00	0.31	0.53
ANTLR	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	4.00	0.03	0.20
Apache_Derby	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.04	0.20
Apache_Isis	0.00	0.00	0.00	0.00	0.00	1.00	1.00	2.00	8.00	0.15	0.45
Apache_Tapestry	0.00	0.00	0.00	0.00	1.00	1.00	1.00	2.00	4.00	0.28	0.53
Apache_Tomcat	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	9.00	0.04	0.23
ArgoUML	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.03	0.18
Checkstyle	0.00	0.00	0.00	0.00	1.00	1.00	1.00	1.00	1.00	0.27	0.45
Dependometer	0.00	0.00	0.00	0.00	1.00	1.00	1.00	3.00	3.00	0.42	0.65
ElasticSearch	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	5.00	0.06	0.31
Hibernate_commons	0.00	0.00	0.00	0.00	1.00	1.00	1.20	2.00	2.00	0.49	0.61
Hibernate_core	0.00	0.00	0.00	0.00	1.00	1.00	2.00	3.00	6.00	0.35	0.65
JChemPaint	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00	0.06	0.23
Jenkins	0.00	0.00	0.00	0.00	0.00	1.00	1.00	2.00	4.00	0.21	0.48
JGit	0.00	0.00	0.00	0.00	0.00	0.00	1.00	3.00	5.00	0.16	0.59
JMock	0.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00	1.00	0.25	0.43
Junit	0.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00	2.00	0.22	0.42
Lombok	0.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00	2.00	0.22	0.42
Megamek	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	2.00	0.08	0.27
Metric_Miner	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.06
OpenCMS	0.00	0.00	0.00	0.00	0.00	1.00	1.00	3.00	4.00	0.20	0.51
Oval	0.00	0.00	0.00	1.00	1.00	2.00	3.00	4.00	4.00	0.84	0.97
Spring	0.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00	9.00	0.18	0.44
VoltDB	0.00	0.00	0.00	0.00	0.00	1.00	1.00	2.00	2.00	0.16	0.40
VRaptor	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	4.00	0.10	0.31

Figure A.1 - Percentile of AA metric: Apache Tomcat.

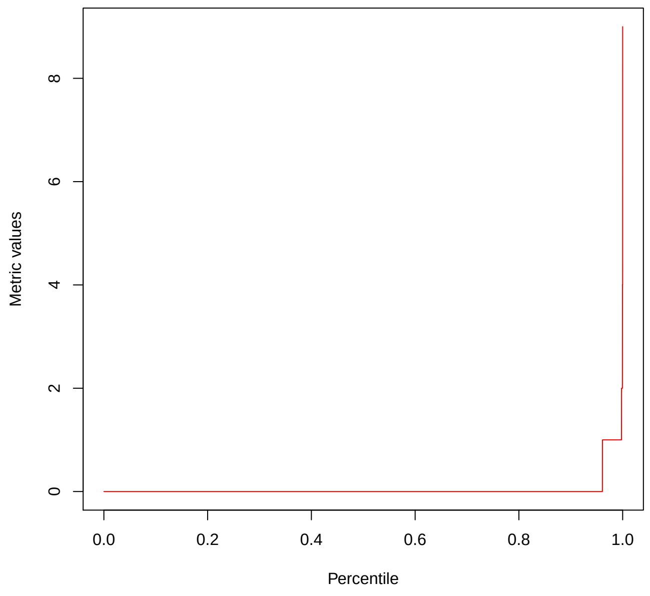


Table A.2 - Threshold Values for AA metric.

Metric	Percentile Reference	Very Frequent	Frequent	Less Frequent	Outlier Value	Lanza-Low	Lanza-Medium	Lanza-High
AA	90	1.00	1.00	2.00	9.00	-0.21	0.21	0.63

A.2 LOC in Annotation Declaration (LOCAD)

The LOCAD metric measures how many lines are used to declare the annotation fully. Annotations that take too many lines to be declared can compromise its readability, maintenance, and evolution. Usually, annotations with high LOCAD also have a high number of arguments (measured by AA). However, it is not a one-to-one relation since a single line of annotation can have multiple arguments. On the other hand, 1 argument that receives a long string or a list of values might be defined using several lines of code.

Table A.3 - Percentiles from LOCAD metric in all projects.

Projects	X5.	X10.	X25.	X50.	X75.	X90.	X95.	X99.	X100.	mean	std
Agilefant	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	6.00	1.01	0.20
ANTLR	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	5.00	1.00	0.07
Apache_Derby	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.00
Apache_Isis	1.00	1.00	1.00	1.00	1.00	1.00	1.00	4.00	13.00	1.08	0.52
Apache_Tapestry	1.00	1.00	1.00	1.00	1.00	1.00	1.00	2.00	9.00	1.02	0.21
Apache_Tomcat	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	8.00	1.00	0.12
ArgoUML	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.00
Checkstyle	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.00
Dependometer	1.00	1.00	1.00	1.00	1.00	1.00	1.00	4.00	4.00	1.08	0.50
ElasticSearch	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	44.00	1.00	0.34
Hibernate_commons	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.00
Hibernate_core	1.00	1.00	1.00	1.00	1.00	1.00	1.00	4.00	58.00	1.08	0.91
JChemPaint	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	29.00	1.20	2.36
Jenkins	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	4.00	1.00	0.06
JGit	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	6.00	1.00	0.08
JMock	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.00
Junit	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	2.00	1.00	0.04
Lombok	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	2.00	1.00	0.04
Megamek	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.00
Metric_Miner	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.00
OpenCMS	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	4.00	1.00	0.07
Oval	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.05
Spring	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	8.00	1.01	0.14
VoltDB	1.00	1.00	1.00	1.00	1.00	1.00	1.00	4.00	7.00	1.05	0.37
VRaptor	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.03

Figure A.2 - Percentile of LOCAD metric: Hibernate Core.

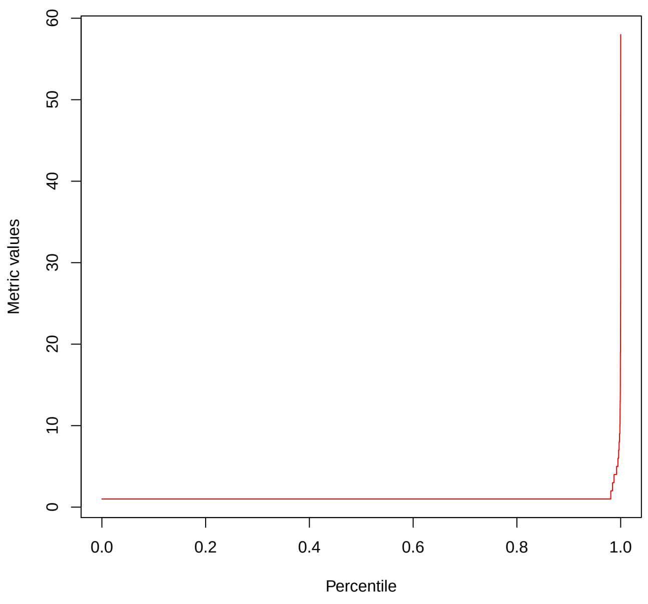


Table A.4 - Threshold Values for LOCAD metric.

Metric	Percentile Reference	Very Frequent	Frequent	Less Frequent	Outlier Value	Lanza-Low	Lanza-Medium	Lanza-High
LOCAD	90	1.00	1.00	2.00	58.00	0.78	1.02	1.27

Table A.3 shows that even though LOCAD gains some values higher than 1 in the percentile 99, the majority of the values are still 1, so the average value is meaningful for this metric. Table A.4 presents the comparison between thresholds values calculated using percentiles and the values based on average and standard deviation values. They are not very different from each other. We conclude that values greater than 2 are a “less frequent” value instead of the Lanza-High value of 1.

Although this metric has low values on average, it is possible to find some annotations with a high number of lines of code. As an example, the highest number found for the LOCAD metric was 58 in a class from Hibernate Core as presented in Figure A.2. This annotation is used for query definition and has both a high number of nested annotations and arguments values containing large strings that spread through several lines.

A.3 Annotation Nesting Level (ANL)

Annotation Nesting Level measures how deep an annotation is nested. As expected, nesting levels are usually very low since there are a few annotation types that use other annotations as attributes. The values only slightly increase from percentile 99, and the majority of the values are zero. Hence, the average value is a good approximation compared to our approach based on the percentile rank.

Table A.5 - Percentiles from ANL metric in all projects.

Projects	X5.	X10.	X25.	X50.	X75.	X90.	X95.	X99.	X100.	mean	std
Agilefant	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	2.00	0.02	0.16
ANTLR	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Apache_Derby	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Apache_Isis	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.03
Apache_Tapestry	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.02
Apache_Tomcat	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	2.00	0.00	0.05
ArgoUML	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Checkstyle	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Dependometer	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
ElasticSearch	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Hibernate_commons	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Hibernate_core	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	4.00	0.03	0.23
JChemPaint	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Jenkins	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
JGit	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
JMock	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Junit	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.01	0.08
Lombok	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.01	0.08
Megamek	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Metric_Miner	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
OpenCMS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.03
Oval	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.01	0.09
Spring	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.05
VoltDB	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
VRaptor	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table A.5 presents the percentile values. Table A.6 shows the thresholds values. As showed in Figure A.3, the outlier value found was 4 for the project Hibernate Core which can be considered an extremely high value since it is common for the ANL to be 0. Using the percentile rank, we obtained the value 0.08 to be a "less frequent" boundary, considering that Lanza's approach was 0.04.

Figure A.3 - Percentile of ANL metric: Hibernate Core.

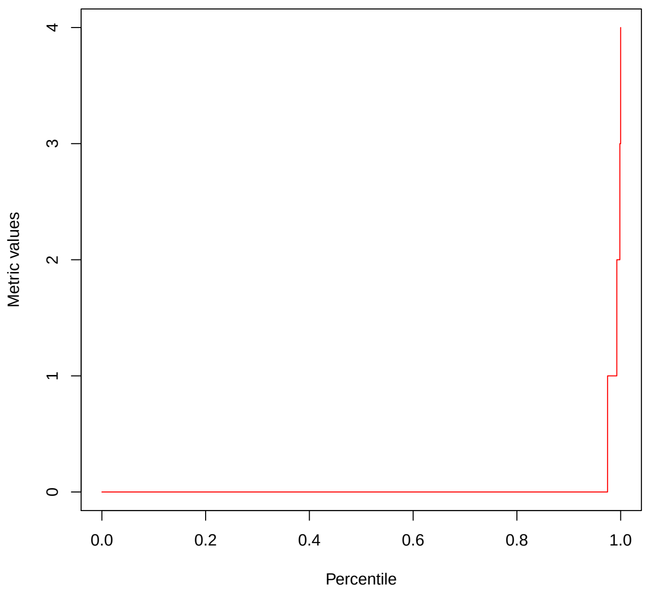


Table A.6 - Threshold Values for ANL metric.

Metric	Percentile Reference	Very Frequent	Frequent	Less Frequent	Outlier Value	Lanza-Low	Lanza-Medium	Lanza-High
ANL	90	0.00	0.00	0.08	4.00	-0.03	0.00	0.04

A.4 Annotations in Element Declaration (AED)

Source code elements such as methods, members, and classes can be annotated. AED measures how many annotations are declared for a specific element, and counting also nested annotations. A high number of annotations in the same element might reveal a code that is hard to maintain. An element that has an excessive amount of annotations might prevent the code from evolving without breaking other parts.

Table A.7 shows the values of percentiles for AED. Before percentile 90, several projects present value 0, meaning no annotation is declared on these elements. As annotations are not mandatory, this value is perfectly acceptable.

From percentile 90, the value stabilizes at 1 until percentile 99, when the value reaches 2. We can then create a region with the number 1 being the delimiter of "Very Frequent" values. Between 1 and 2 we have a "Frequent" value, and above 2

Table A.7 - Percentiles from AED metric in all projects.

Projects	X5.	X10.	X25.	X50.	X75.	X90.	X95.	X99.	X100.	mean	std
Agilefant	0.00	0.00	0.00	0.00	1.00	2.00	2.00	4.00	8.00	0.59	0.85
ANTLR	0.00	0.00	0.00	1.00	1.00	1.00	1.00	1.00	3.00	0.54	0.50
Apache_Derby	0.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00	2.00	0.12	0.33
Apache_Isis	0.00	0.00	0.00	0.00	1.00	1.00	1.00	3.00	7.00	0.45	0.65
Apache_Tapestry	0.00	0.00	0.00	0.00	1.00	1.00	1.00	3.00	5.00	0.44	0.65
Apache_Tomcat	0.00	0.00	0.00	0.00	1.00	1.00	1.00	1.00	5.00	0.38	0.50
ArgoUML	0.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00	3.00	0.18	0.39
Checkstyle	0.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00	1.00	0.12	0.32
Dependometer	0.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00	3.00	0.19	0.42
ElasticSearch	0.00	0.00	0.00	0.00	1.00	1.00	1.00	1.00	18.00	0.27	0.46
Hibernate_commons	0.00	0.00	0.00	0.00	0.00	1.00	1.00	2.00	2.00	0.18	0.45
Hibernate_core	0.00	0.00	0.00	0.00	1.00	1.00	2.00	3.00	27.00	0.54	0.79
JChemPaint	0.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00	2.00	0.24	0.43
Jenkins	0.00	0.00	0.00	0.00	1.00	1.00	1.00	2.00	4.00	0.43	0.61
JGit	0.00	0.00	0.00	0.00	1.00	1.00	1.00	1.00	3.00	0.38	0.50
JMock	0.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00	2.00	0.21	0.43
Junit	0.00	0.00	0.00	1.00	1.00	1.00	1.00	3.00	6.00	0.58	0.69
Lombok	0.00	0.00	0.00	1.00	1.00	1.00	1.00	3.00	6.00	0.58	0.69
Megamek	0.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00	3.00	0.15	0.36
Metric_Miner	0.00	0.00	0.00	0.00	1.00	1.00	1.00	1.00	2.00	0.41	0.50
OpenCMS	0.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00	3.00	0.17	0.41
Oval	0.00	0.00	0.00	0.00	0.00	1.00	2.75	4.00	4.00	0.39	0.91
Spring	0.00	0.00	0.00	1.00	1.00	1.00	2.00	3.00	7.00	0.59	0.68
VoltDB	0.00	0.00	0.00	0.00	1.00	1.00	1.00	2.00	24.00	0.30	0.51
VRaptor	0.00	0.00	0.00	1.00	1.00	1.00	1.00	3.00	5.00	0.59	0.64

Table A.8 - Threshold Values for AED metric.

Metric	Percentile Reference	Very Frequent	Frequent	Less Frequent	Outlier Value	Lanza-Low	Lanza-Medium	Lanza-High
AED	90	1.00	1.00	2.00	27.00	-0.19	0.36	0.91

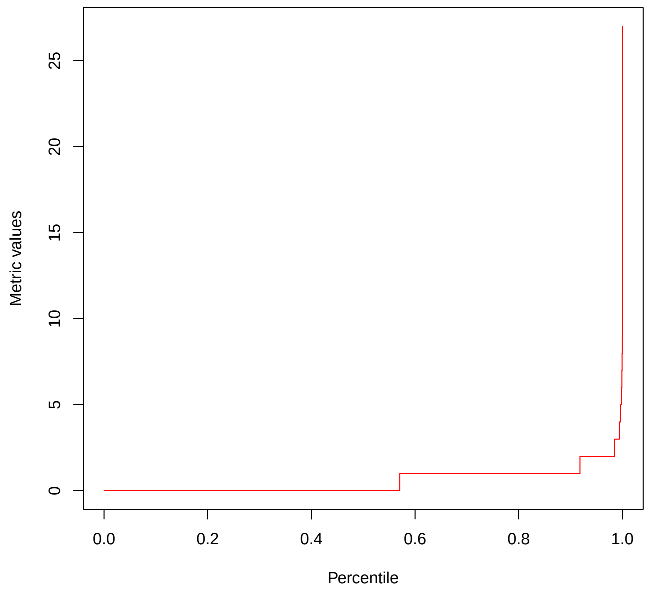
the value can be considered “Less Frequent”.

Using Lanza’s approach, we obtain an average value of 0 and high margin of 1. This can be seen as an AED value greater than 1 being considered high. In our analysis, we conclude that this number can be pushed up to 2. The comparison between the 2 kinds of thresholds is presented in Table A.8.

Although the statistical threshold shows that values of AED are higher than 2, which can be considered high, it is common to find elements with 3 or 4 annotations that are not overloaded with metadata. Therefore, this metric should be interpreted with others, such as AA and LOCAD, to find annotation declarations that can cause maintenance problems.

Figure A.4 presents the distribution graph from the project Hibernate Core that

Figure A.4 - Percentile of AED metric: Hibernate Core.



have the highest value for LOCAD. The highest number found for AED was 27 in the same class. In this case, most of the annotations are nested inside a single one.

A.5 Unique Annotations in Class (UAC)

Unlike the AC metric, which counts equivalent annotations, the UAC metric is focused on the number of different annotations in a class. An annotation is considered equivalent to another when it has the same type and the same argument values. By definition, the UAC value is never greater than the AC value for a class.

Table A.10 presents the obtained threshold values and Table A.9 shows the percentile values for all projects. In our analysis, the percentile 75 could have been used as a reference point to determine the threshold values. However, this would lead to a smaller "Very Frequent" region. Since the goal is to make the threshold values as flexible as possible, the reference point was pushed to the percentile 90 to allow a broader "Very Frequent" region. Moreover, the UAC metric never assumes values below 1 since our analyzed classes contain at least 1 annotation.

Therefore, we obtained 3 as the "Very Frequent" value, 4 as "Frequent" and 9 as "Less Frequent". In Lanza's approach, the "Lanza-Medium" is 2, the "Lanza-Low" is 0, and the "Lanza-High" is 4. Hence, our analysis covers a broader range of values,

Table A.9 - Percentiles from UAC metric in all projects.

Projects	X5.	X10.	X25.	X50.	X75.	X90.	X95.	X99.	X100.	mean	std
Agilefant	1.00	1.00	2.00	3.00	5.00	7.00	9.05	29.87	40.00	4.22	4.70
ANTLR	1.00	1.00	1.00	1.00	2.00	2.00	2.00	4.00	14.00	1.38	0.93
Apache_Derby	1.00	1.00	1.00	1.00	1.00	1.00	1.00	2.00	3.00	1.05	0.25
Apache_Isis	1.00	1.00	1.00	1.00	2.00	4.00	5.00	9.00	29.00	1.92	1.95
Apache_Tapestry	1.00	1.00	1.00	1.00	3.00	4.00	6.00	11.26	36.00	2.19	2.35
Apache_Tomcat	1.00	1.00	1.00	1.00	1.00	2.00	3.00	5.00	15.00	1.40	0.97
ArgoUML	1.00	1.00	1.00	1.00	1.00	2.00	2.00	3.00	4.00	1.14	0.44
Checkstyle	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.49	2.00	1.02	0.14
Dependometer	1.00	1.00	1.00	1.00	1.00	2.00	2.90	4.74	6.00	1.30	0.89
ElasticSearch	1.00	1.00	1.00	1.00	2.00	2.00	3.00	5.00	13.00	1.36	0.78
Hibernate_commons	1.00	1.00	1.00	1.00	1.25	2.00	2.15	4.43	5.00	1.40	0.94
Hibernate_core	1.00	1.00	1.00	2.00	4.00	7.00	10.00	16.00	375.00	3.41	5.87
JChemPaint	1.00	1.00	1.00	1.00	1.00	1.00	1.20	2.00	2.00	1.05	0.23
Jenkins	1.00	1.00	1.00	3.00	4.00	6.00	7.60	14.52	27.00	3.15	2.55
JGit	1.00	1.00	1.00	1.00	2.00	3.00	4.90	11.58	22.00	1.88	2.04
JMock	1.00	1.00	1.00	1.00	2.00	3.00	3.10	5.00	5.00	1.61	0.97
Junit	1.00	1.00	1.00	2.00	2.00	4.00	5.00	12.28	36.00	2.24	2.58
Lombok	1.00	1.00	1.00	2.00	2.00	4.00	5.00	12.28	36.00	2.24	2.58
Megamek	1.00	1.00	1.00	1.00	1.00	1.00	2.00	5.00	80.00	1.31	3.59
Metric_Miner	1.00	1.00	1.00	1.00	2.00	2.80	3.00	3.48	4.00	1.43	0.75
OpenCMS	1.00	1.00	1.00	1.00	1.00	2.00	3.00	12.00	59.00	1.58	3.22
Oval	1.00	1.00	1.00	3.00	5.00	5.70	6.00	9.17	13.00	3.13	2.17
Spring	1.00	1.00	1.00	2.00	4.00	7.00	8.00	15.15	102.00	3.06	3.95
VoltDB	1.00	1.00	1.00	1.00	2.00	4.00	7.00	14.00	25.00	2.08	2.51
VRaptor	1.00	1.00	2.00	3.00	4.00	5.00	6.00	8.56	29.00	3.08	2.00

Table A.10 - Threshold Values for UAC metric.

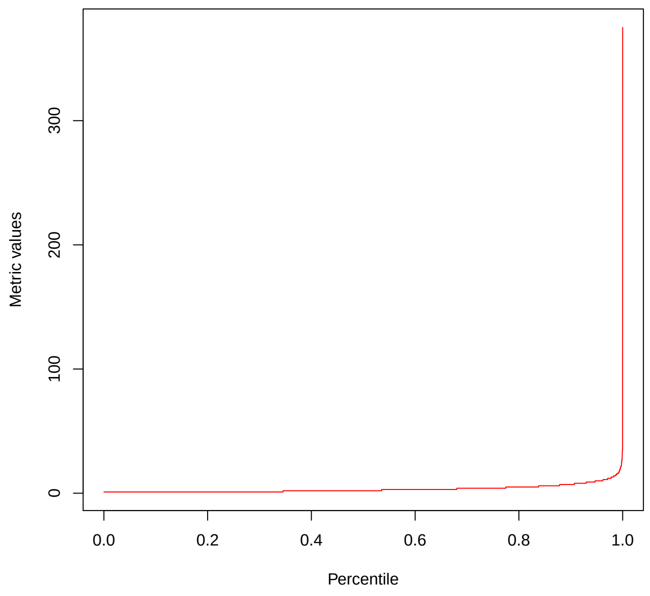
Metric	Percentile Reference	Very Frequent	Frequent	Less Frequent	Outlier Value	Lanza-Low	Lanza-Medium	Lanza-High
UAC	90	3.00	4.00	9.00	375.00	0.01	1.98	3.95

and the obtained results share some balanced growth with the AC percentile rank. Accordingly, the obtained values are coherent.

Some Lanza-low values are negative for the same reasons explained in the AC section. Since the UAC value is never greater than the AC value, as expected, the standard deviation is not as high as for the AC. Therefore, the average value can still be useful. However, using our analysis based on percentiles, a more precise threshold value can be obtained.

The highest value found for UAC was 729 in the same class on Hibernate Core that has the highest value of AC (CoreHibernateLogger), as seen in Figure A.5. Although it has 729 annotations, only 375 are unique. This class has several annotations of the type @Message, but their argument values are different.

Figure A.5 - Percentile of UAC metric: Hibernate Core.



A.6 Annotations Schemas in Class (ASC)

When a developer creates a new code annotation, they are grouped in a set representing metadata for a given domain, which is called schema. Usually, code annotations are created by framework developers. When application developers use a metadata-based framework, it uses the code annotations to configure the necessary metadata. An application may use as many schemas from the metadata-based framework as needed.

Stricly speaking, we define the annotation schema (or simply "schema") as a package where the code annotation was created. For instance, a well-known schema is the `org.junit` that comprises code annotations such as `@Test`, `@Before`, and `@After`. The ASC metric represents the number of schemas that a class is using. It can be identified by the different number of annotation packages being imported.

Table A.11 shows the percentile values for the ASC metric. We observe that before percentile 90, the majority of the projects have a maximum value of 1 ASC per class. Based on that, we conclude that a "very frequent" value for ASC is 1, which means that when annotations are used in a class, most of the time, all of them belong to a single schema. From percentile 90 to 99, we have 2 as a "frequent" value for the ASC metric. Beyond that, the value can be considered high. From Lanza's point of view, the higher value is 1, while the average is 0. Once again, our approach has proved to be flexible, yielding a wider range of values for the thresholds, which better accommodates real-world projects.

Table A.11 - Percentiles from ASC metric in all projects.

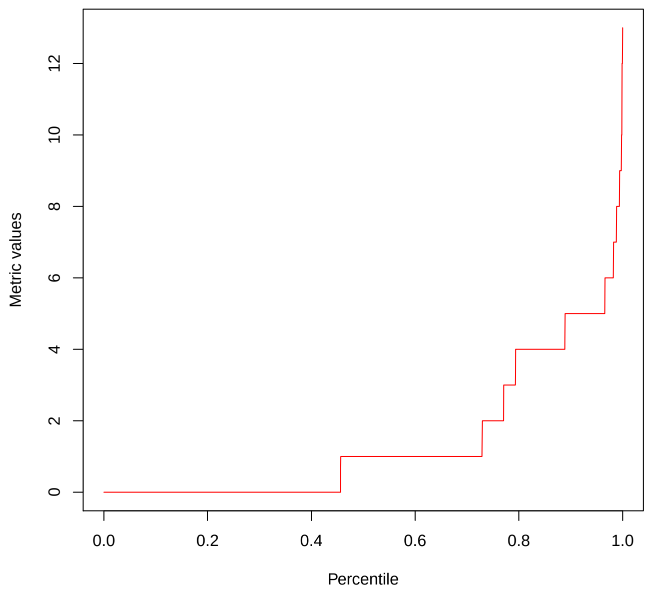
Projects	X5.	X10.	X25.	X50.	X75.	X90.	X95.	X99.	X100.	mean	std
Agilefant	0.00	0.00	1.00	2.00	3.00	5.00	5.00	5.41	6.00	2.29	1.62
ANTLR	0.00	0.00	0.00	0.00	1.00	1.00	1.00	1.00	1.00	0.38	0.49
Apache_Derby	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Apache_Isis	0.00	0.00	0.00	0.00	1.00	1.00	2.00	3.00	4.00	0.49	0.73
Apache_Tapestry	0.00	0.00	0.00	1.00	1.00	2.00	2.00	3.00	5.00	0.84	0.77
Apache_Tomcat	0.00	0.00	0.00	0.00	1.00	1.00	1.00	2.00	3.00	0.32	0.52
ArgoUML	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.04	0.20
Checkstyle	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Dependometer	0.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00	1.00	0.19	0.39
ElasticSearch	0.00	0.00	0.00	0.00	0.00	1.00	1.00	2.00	3.00	0.26	0.48
Hibernate_commons	0.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00	1.00	0.15	0.37
Hibernate_core	0.00	0.00	0.00	1.00	2.00	2.00	3.00	3.00	5.00	0.95	0.93
JChemPaint	0.00	0.00	0.00	1.00	1.00	1.00	1.00	1.44	2.00	0.58	0.53
Jenkins	0.00	0.00	1.00	1.00	2.00	3.00	4.00	5.52	10.00	1.51	1.30
JGit	0.00	0.00	0.00	1.00	1.00	1.00	1.00	3.00	3.00	0.57	0.60
JMock	0.00	0.00	0.00	0.00	1.00	2.00	2.00	2.00	2.00	0.51	0.75
Junit	0.00	0.00	1.00	1.00	1.00	2.00	2.00	2.00	3.00	1.00	0.62
Lombok	0.00	0.00	1.00	1.00	1.00	2.00	2.00	2.00	3.00	1.00	0.62
Megamek	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	2.00	0.04	0.22
Metric_Miner	0.00	0.00	0.00	0.00	1.00	1.00	1.00	1.00	1.00	0.30	0.46
OpenCMS	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	2.00	0.10	0.31
Oval	0.00	0.00	0.00	1.00	2.00	2.00	2.00	3.00	4.00	1.00	0.91
Spring	0.00	0.00	0.00	1.00	2.00	5.00	5.00	8.00	13.00	1.44	1.98
VoltDB	0.00	0.00	0.00	0.00	1.00	1.00	2.00	2.92	4.00	0.52	0.70
VRaptor	1.00	1.00	1.00	2.00	2.00	3.00	4.00	4.00	5.00	1.74	0.94

Table A.12 - Threshold Values for ASC metric.

Metric	Percentile Reference	Very Frequent	Frequent	Less Frequent	Outlier Value	Lanza-Low	Lanza-Medium	Lanza-High
ASC	90	1.50	1.80	2.40	13.00	-0.01	0.65	1.30

Figure A.6 presents the Spring project distribution graphs as an example of an outlier. This project has a class, `StompIntegrationTest`, with the value of 13 for ASC. Even though most of the classes have low values, we found classes coupled with several annotations schemas. Table A.12 presents the Thresholds value obtained for ASC.

Figure A.6 - Percentile of ASC metric: Spring.



APPENDIX B - PERCENTILE RANK CHARTS

This appendix intends to bring additional material regarding the metrics distribution graph presented on Chapter 4. We present a total of seven figures. Each one of them contain the distribution graph for every project in a single graph. Each metric has its own graph.

Figure B.1 - AA Distribution Graph.

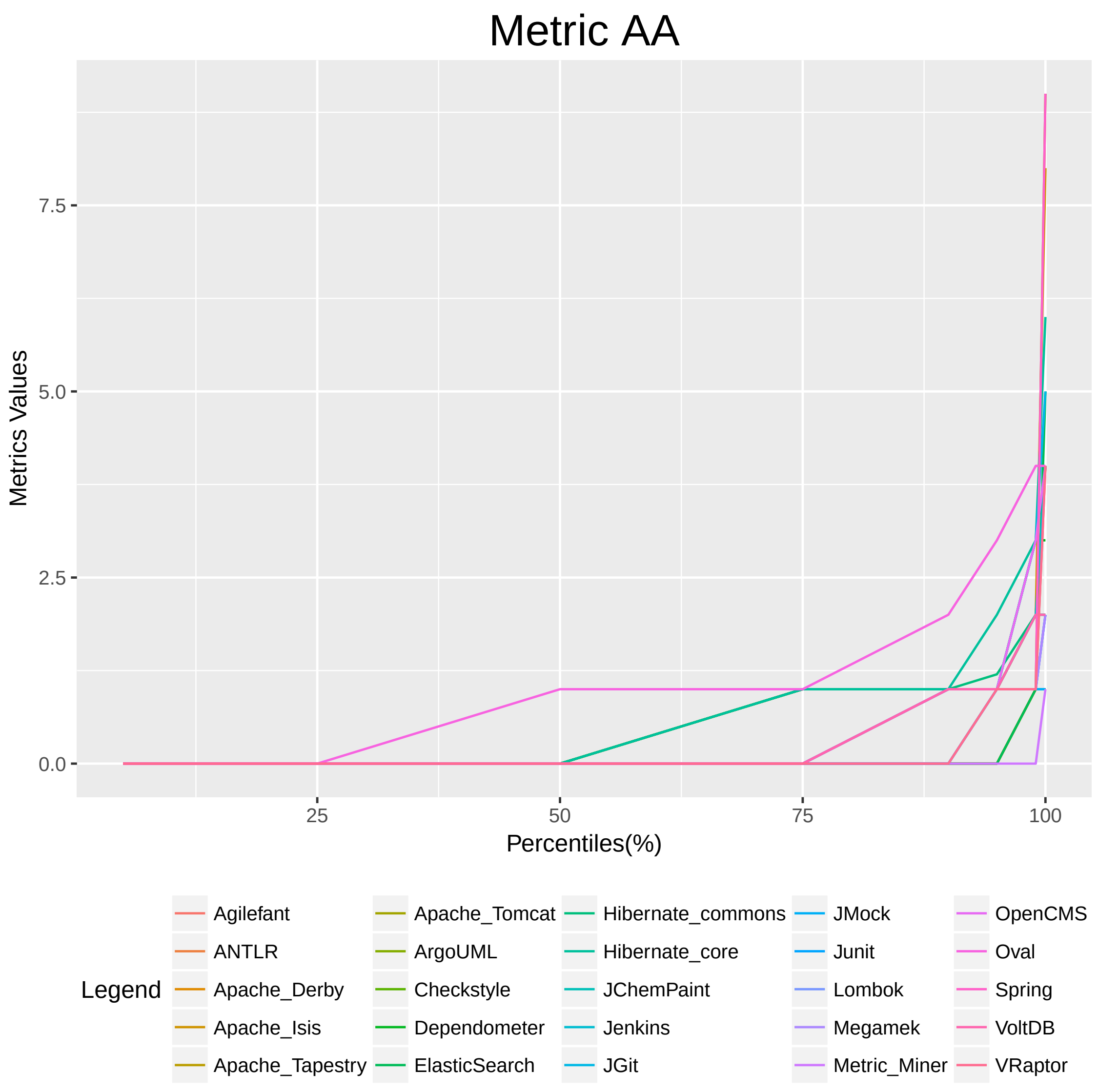


Figure B.2 - AC Distribution Graph.

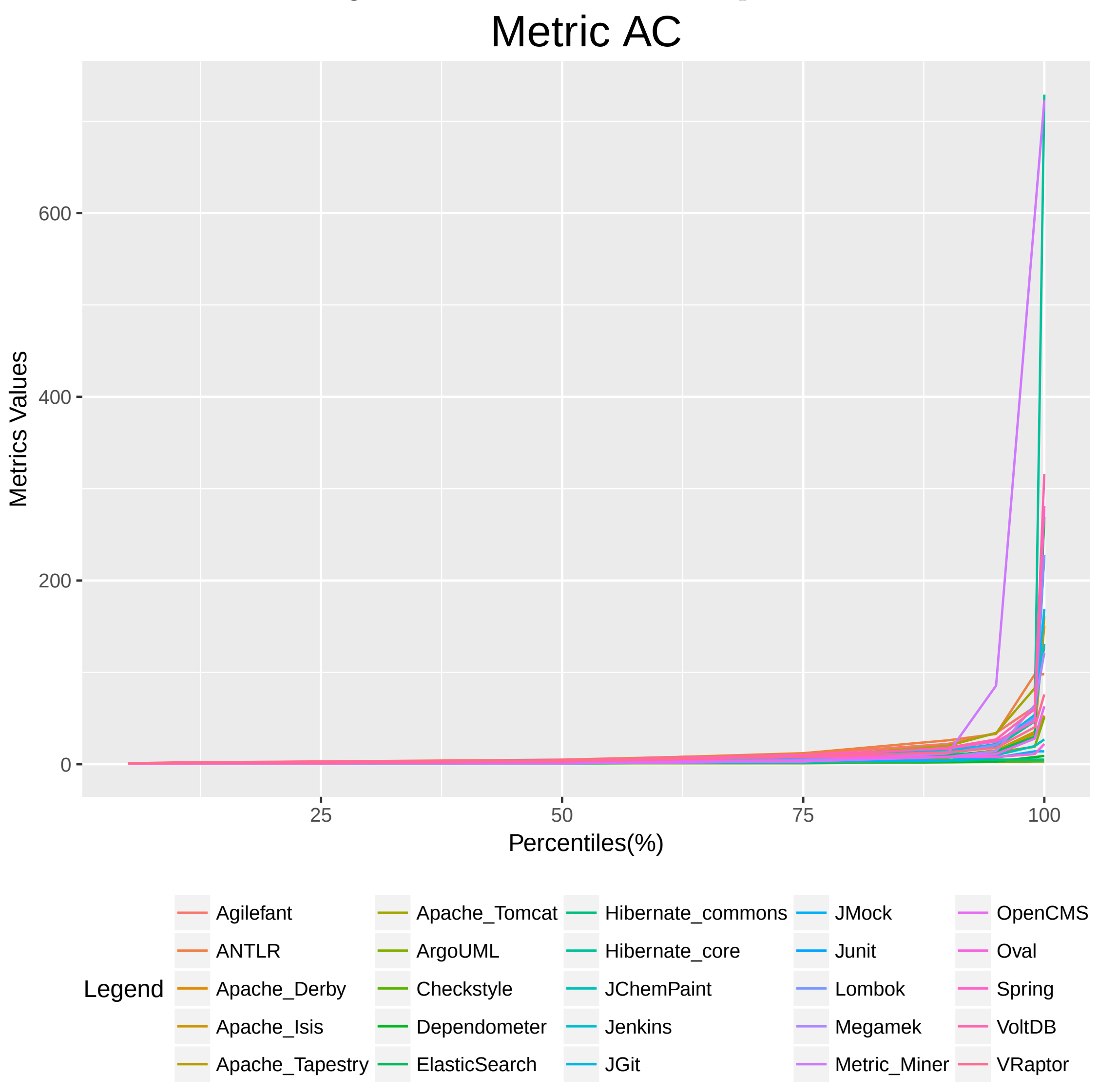


Figure B.3 - AED Distribution Graph.

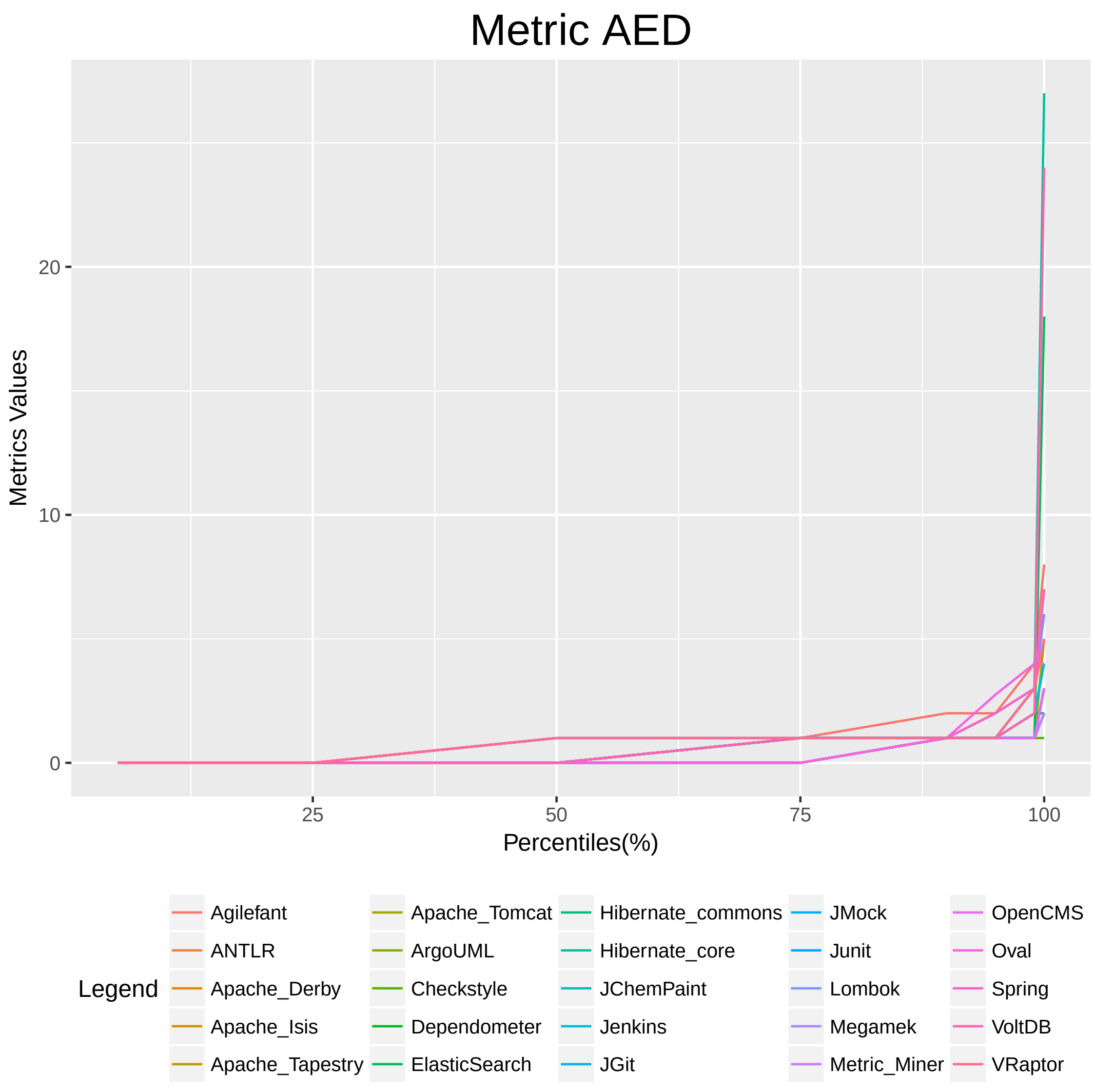


Figure B.4 - ANL Distribution Graph.

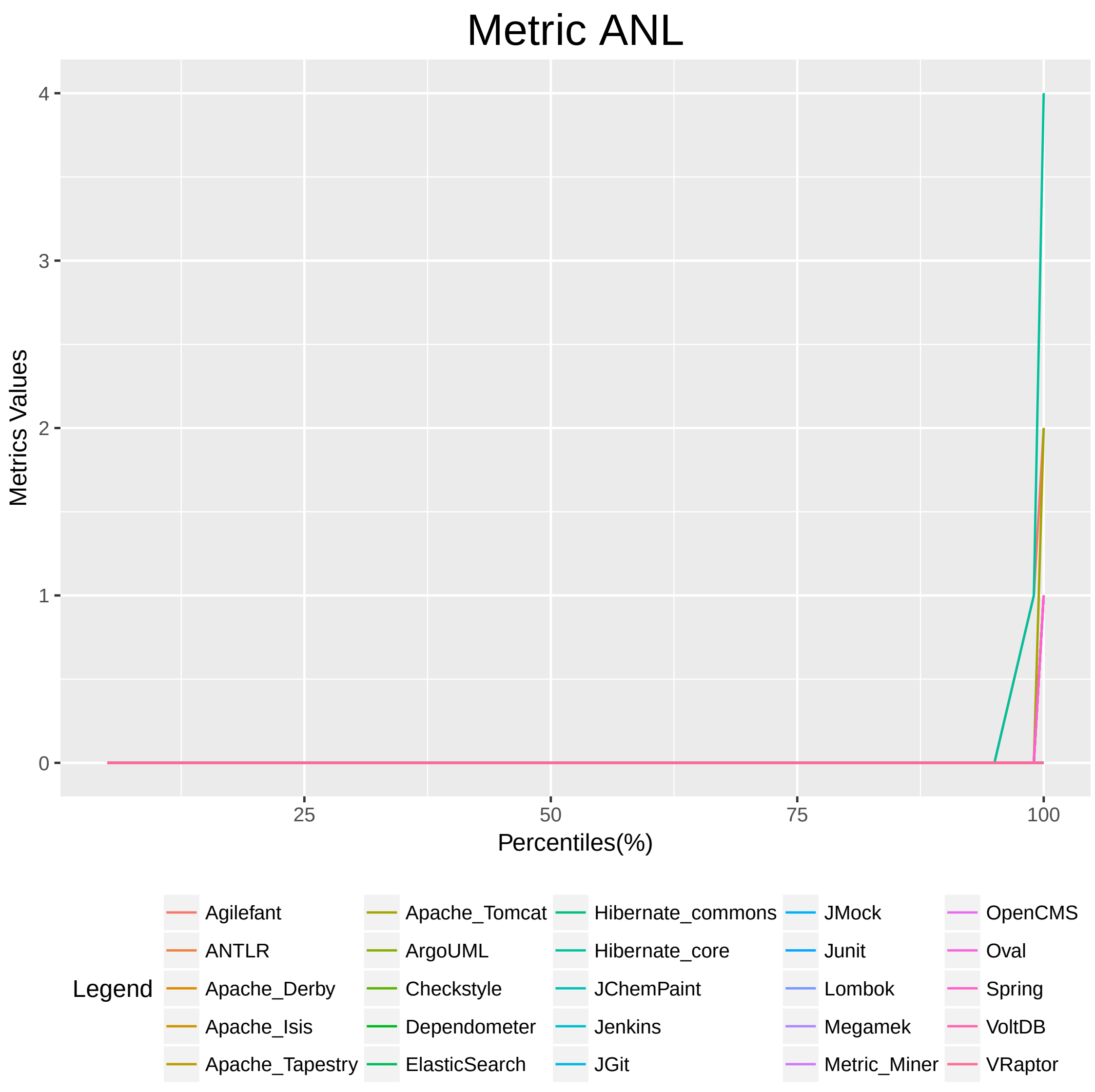


Figure B.5 - ASC Distribution Graph.

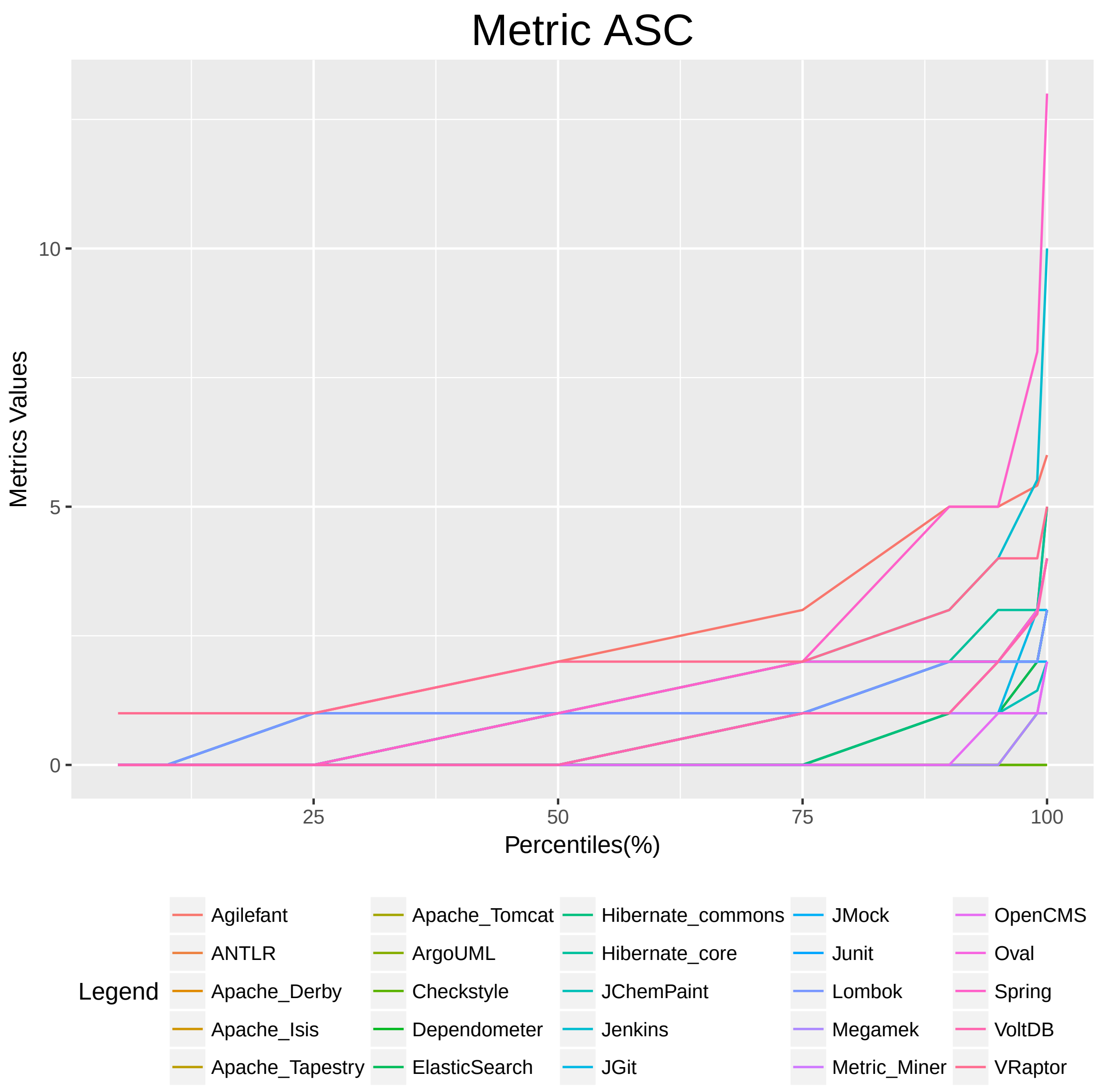


Figure B.6 - LOCAD Distribution Graph.

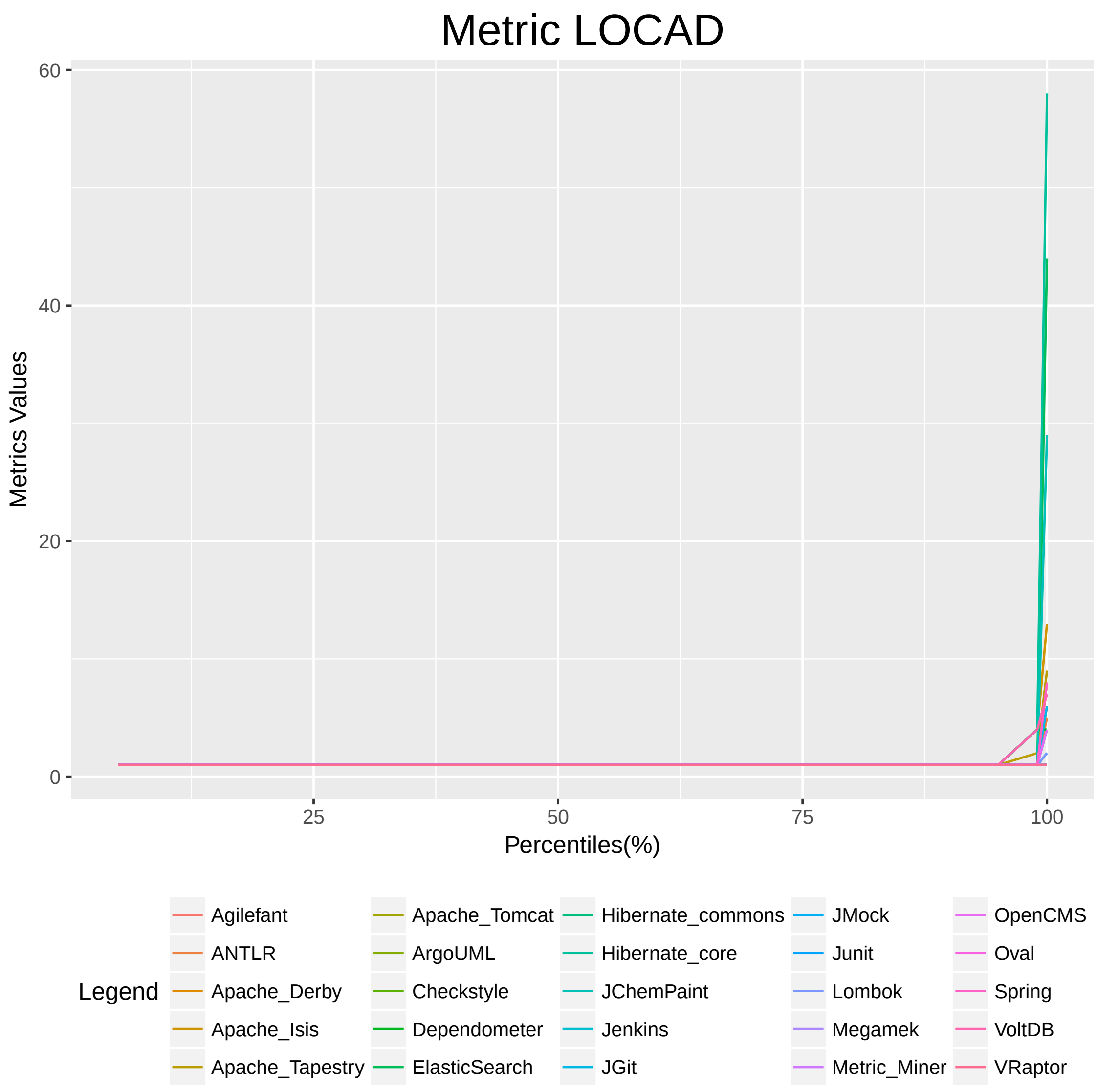
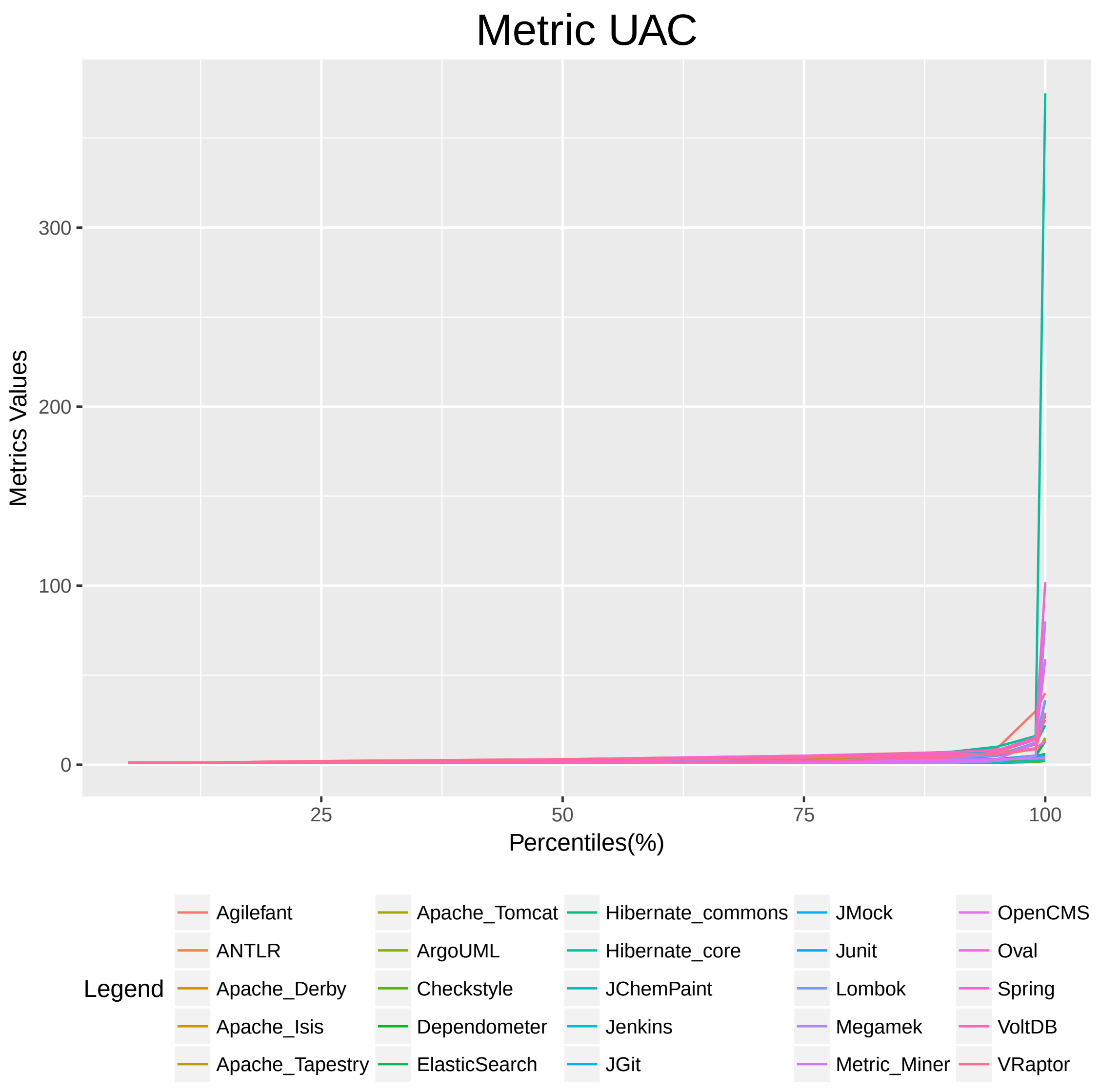


Figure B.7 - UAC Distribution Graph.


Metric UAC
Metrics Values
Percentiles(%)
Legend
Agilefant
ANTLR
Apache_Derby
Apache_Isis
Apache_Tapestry
Apache_Tomcat
ArgoUML
Checkstyle
Dependometer
ElasticSearch
Hibernate_commons
Hibernate_core
JChemPaint
Jenkins
JGit
JMock
Junit
Lombok
Megamek
Metric_Miner
OpenCMS
Oval
Spring
VoltDB
VRaptor