A Metadata Handling API for Framework Development: a Comparative Study

Eduardo Guerra
Computer Science Faculty
Free University of Bozen-Bolzano
Bolzano, Italy
guerraem@gmail.com

Phyllipe Lima
National Institute of Telecommunications (INATEL)
Santa Rita do Sapucaí, Brazil
phyllipe@inatel.br

Joelma Choma, Marco Nardes
National Institute for Space Research (INPE)
São José dos Campos, Brazil
{jh.choma70,marconardes}@gmail.com

Tiago Silva
Department of Computer Science
Federal University of São Paulo
São José dos Campos, Brazil
silva.tiago@unifesp.br

Michele Lanza
Faculty of Informatics
University of Lugano
Lugano, Switzerland
michele.lanza@usi.ch

Paulo Meirelles
São Paulo School of Medicine
Federal University of São Paulo
São Paulo, Brazil
paulo.meirelles@unifesp.br

ABSTRACT

Frameworks play an essential role in software development, providing not only code reuse, but also design reuse. Several Java frameworks and APIs such as Spring, JPA, and CDI rely on the use of metadata, mainly defined by code annotations. These frameworks usually use the Java Reflection API to consume code annotations, which only returns the annotations in a given code element. This strategy, however, is far from the needs of a real framework. The goal of this paper is to propose a novel API, named Esfinge Metadata, to assist in the development of frameworks based on metadata and applications based on custom annotations. Being based on annotations itself, this new API uses them to map metadata to class members. We carried out an experiment to evaluate our API and its impact on aspects such as code structure, complexity, and coupling, while also performing a comparison with the direct use of the Java Reflection API. The participants implemented a metadata-based framework based on realistic requirements in a sequence of 10 tasks that took an average of nine hours. As a result, participants that used our API maintained a more stable code evolution, regarding complexity and coupling as opposed to participants using the Java Reflection API, where the code metrics evolution and structure vary greatly.

CCS CONCEPTS

• **Software and its engineering** → *Software development techniques.*

KEYWORDS

Framework, Metadata, Meta-Framework, Annotations, Code Metrics, Exploratory Experiment, Java

ACM Reference Format:
Eduardo Guerra, Phyllipe Lima, Joelma Choma, Marco Nardes, Tiago Silva, Michele Lanza, and Paulo Meirelles. 2020. A Metadata Handling API for Framework Development: a Comparative Study. In *34th Brazilian Symposium on Software Engineering (SBES '20), October 21–23, 2020, Natal, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3422392.3422428

1 INTRODUCTION

Modern frameworks use metadata configuration to offer applications a high reuse level and a better adaptation to their needs. These are known as metadata-based frameworks since they process their logic based on the metadata configuration of the classes whose instances they are working with [11]. Highly used Java frameworks such as Spring, Hibernate, and JUnit use code annotations as their primary approach for configuring metadata.

The consumption of code annotations in Java usually make direct use of the Java Reflection API (Application Programming Interface). In this native API, the primary method for metadata retrieval returns the annotations that are configuring a given element. However, this approach is far from the needs of such a framework and may lead to a complex and high coupled code that may compromise its evolution and maintenance. For instance, it is typical for a framework to (i) retrieve all methods or fields with an annotation, (ii) to validate an annotation based on its context, and (iii) to retrieve annotations based on a common annotation that configure its type [11].

The Esfinge Metadata API is an extensible meta-framework for reading and validating annotations, which aims at simplifying and assisting, in particular, the development of metadata-based frameworks. Moreover, any application that defines custom annotations can also use it. The Esfinge Metadata API has features to retrieve and find annotations, and it was developed based on metadata-based framework patterns and best practices [11]. The API was designed to guide developers towards these best practices and provide a stable code evolution, with low complexity and coupling.

This paper aims to propose a novel API approach for metadata reading, processing, and consumption. As such, it provides functionality that assists the development of metadata-based frameworks, as well as applications that use custom annotations. A prior work presented a preliminary version of the Esfinge Metadata API and performed a demonstrative case study. The API was still under construction and no proper experiment was carried out for evaluation [18].

To evaluate our novel API, we performed a controlled experiment with two groups of carefully selected developers. One group used our new Esfinge Metadata API, while the other used the Java Reflection API. They both developed a metadata-based framework for mapping application parameters to an annotated class instance, which we named the "target framework". We divided the experiment into 10 tasks, where each one of them incrementally added functionality to the target framework. The complete experiment took, on average, nine hours of coding.

This paper aims to answer the following research question:

RQ: How does the Esfinge Metadata API provide support to maintain complexity and coupling in the evolution of a metadata-based framework compared to the Java Reflection API?

To answer this question, we executed the experiment and afterward analyzed the repository provided by each participant. We performed a careful code inspection in a qualitative analysis and observed the evolution of well-established software code metrics. All our findings and conclusions were obtained based on this code inspection, rather than statistical and numerical analysis.

From the results, we highlight the following finding. The usage of the Esfinge Metadata API provides a more consistent behavior in the evolution of coupling and complexity metrics, guiding metadata-based framework development to a more predictive way towards the best practices.

2 METADATA IN THE CONTEXT OF OBJECT-ORIENTED PROGRAMMING

The term "metadata" is used in a variety of contexts in the computer science field. In all of them, it means data referring to the data itself. When discussing databases, the data are the ones persisted, and the metadata is their description, i.e., the structure of the table. In the object-oriented context, the data are the instances, and the metadata is their description, i.e., information that describes the class. As such, fields, methods, super-classes, and interfaces are all metadata of a class instance. A class field, in turn, has its type, access modifiers, and name as its metadata. When a developer uses reflection, it is manipulating the metadata of a program and using it to work with previously unknown classes [6, 11].

2.1 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 annotations [12], and in C#, by attributes [4]. 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 software elements [20, 23].

Annotations are a feature of the Java language, which became official on version 1.5 [12] 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) [13], 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.

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.

2.2 Java Reflection API to Consume Code Annotations

The Java language offers means for developers to consume code annotations through the Java Reflection API. Using the `AnnotatedElement` interface, implemented by reflection classes that represent code elements (such as the classes `Method`, `Field` and so forth), developers have access to methods such as `isAnnotationPresent()` and `getAnnotation()`. The first one verifies if an annotation is present on a give code element, and the latter returns the desired annotation [6].

Using the Java Reflection API, the developer must write explicit code to read metadata and retrieve code annotations individually. Developers cannot recover all annotations from a method, or a field, or even a class using a single call. The search must be executed iteratively. Hence, for complex metadata reading, using the Java Reflection API, the code could get complicated and create barriers to allow evolution. With this approach, the developer is not guided into using known best practices for metadata reading [9].

3 ESFINGE METADATA API

The Esfinge Metadata API is an open-source[1] meta-framework developed to ease and support the process of metadata reading. It also functions as an API, and a preliminary version was published on [18] with no proper experiment, only a demonstration.

We believe the API is more suitable for metadata-based frameworks development since the Java Reflection API has only two main methods to access annotations. The first one checks if a specific annotation is present on a code element, the second retrieves an annotation from an element. Both of these methods are insufficient for the needs of developers. Hence they need to create much verbose code to retrieve the necessary data.

Our research group designed the API based on recurrent solutions found inside frameworks to read metadata from applications. They are documented as patterns on a previous work [9]. Using the API to read metadata is not limited to frameworks using these patterns. However, the developer will be guided towards them, just as MVC frameworks tend to do the same.

The core pattern is the "Metadata Container". It introduces a class role called `Metadata Container`, whose instance represents metadata retrieved at runtime. Figure 1 presents the pattern's basic diagram.

The class `MetadataContainer` is responsible for storing metadata read from annotations at runtime. The `FrameworkController` asks the `Repository` for metadata of a given target class. If it was

[1]github.com/EsfingeFramework/metadata

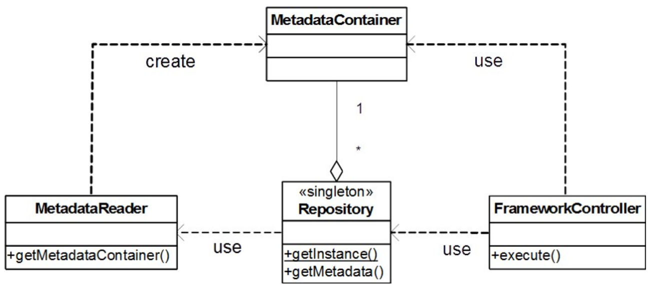

Figure 1: Basic structure of the Metadata Container pattern [9]

not yet retrieved, the `Repository` invokes the `MetadataReader` to fetch it wherever it may be located. The `MetadataReader` should handle all peculiarities of the annotation schema[2] and strategies for metadata definition, as well as it should return the `MetadataContainer` ready to be used by the `FrameworkController`.

Since this is a recurrent practice in several frameworks [9], we assume that storing metadata in a regular class when reading a complex annotation schema is an excellent general design. Therefore the "Metadata Container" pattern is central for the proposed API.

To demonstrate how to use this API, consider the class on Figure 2. It is annotated with `@Annotation1(name="Player")` and both private fields are annotated with `@Annotation2(attr=''attr1'')`. We wish to read the metadata using the Esfinge Metadata API.

To read the metadata of the class `Player`, consider the class on Figure 3. It contains a class using the Esfinge Metadata API. It is functioning as a class metadata container to store metadata from annotations used on the example code on Figure 2. Further, the class `FieldContainer` on Figure 4 is working as a field metadata container. This structure is following the "Metadata Container" pattern.

```
1   @Annotation1(name="Player")
2   public class Player {
3
4       @Annotation2(attr="attr1")
5       private int field1;
6
7       @Annotation2(attr="attr2")
8       private int field2;
9   }
```

Figure 2: Simple class with metadata to be read

The classes on Figures 4 and 3 use some annotations provided by the Esfinge Metadata API. The following list gives a brief understanding of some of the framework annotations, so interested developers and researches may have a grasp of how the Esfinge Metadata API works. The annotations presented on this list are just a small sample of the functionalities and capabilities offered by the Esfinge Metadata API[3].

[2]An annotation schema is defined as a set of associated annotations that belong to the same API. An annotation-based API usually uses a group of related annotations that represent the set of metadata necessary for its usage.
[3]For further details, refer to the official website of the Esfinge Metadata API http://esfinge.sourceforge.net/MetadataEN.html, available in english

```
1
2   @ContainerFor(ContainerTarget.TYPE)
3   public class PlayerMetadataContainer {
4       @ElementName
5       private String elementName
6
7       @AllFieldsWith(Annotation2.class)
8       private List<FieldContainer> fieldMetadata;
9
10      @AnnotationProperty(annotation = Annotation1.class, property = "name")
11      private String exampleAnnotProp;
12      //getters and setters omitted
13  }
```

Figure 3: Simple Class Metadata Container

```
1   @ContainerFor(ContainerTarget.FIELDS)
2   public class FieldContainer {
3
4       @ElementName
5       private String elementName
6
7       @AnnotationProperty(annotation = Annotation2.class,
8                       property="attr")
9       private String attr;
10
11      //getters and setters omitted
12  }
```

Figure 4: Simple Field Metadata Container

- `@ContainerFor()`: The container class can store metadata read from fields, methods, classes, enums, etc. This annotation informs the container class what type of code element it will store. When we pass in `ContainerTarget.TYPE`, we are informing that this class stores metadata from a class or enum. We pass in `ContainerTarget.FIELDS` to store metadata from a field, and so forth.
- `@ElementName`: This stores the name of the code element from where we are reading the metadata. In this example the name is "Player", since we are reading the metadata from the class `Player`. If we were reading metadata from a field, as shown in Figure 4, the `@ElementName` would be the name of the field, i.e, `field1` and `field2`.
- `@AnnotationProperty()`: Retrieves the specified property from a given annotation. Observing Figure 3, line 11, we want to retrieve from inside `@Annotation1` the value stored on the parameter "name". As shown on the code of Figure 2, line 1, this value is "Player".
- `@AllFieldsWith())`: This is done to retrieve metadata on fields inside the class. We specify what annotation we are looking for, and it searches for the fields that contain those annotations. In the code on Figure 3, line 8, we are searching every field that contains the annotation `@Annotation2`. Looking at Figure 2 the fields are "field1" and "field2". This annotations also has a version for methods, `@AllMethodsWith()`, i.e, it will search for methods with a given annotation.

Figure 5 shows how the containers are being used to read metadata from the class `Player`. We first create the `AnnotationReader` and the method `readingAnnotationsTo()` returns the container. It has two parameters; the first is the class with the metadata to be read, i.e., the `(Player)` class. The second parameter is the class that will store the metadata that was read, i.e., the (`PlayerMetadataContainer`) class. Afterward, the container is populated with metadata that can be accessed through getter methods. The output of running this code is presented on Figure 6.

```

public class Main {

    public static void main(String args[]){

    AnnotationReader reader = new AnnotationReader();
    PlayerMetadataContainer container = reader.readingAnnotationsTo
                                (Player.class,
                                PlayerMetadataContainer.class);

    System.out.println(container.getElementName());
    System.out.println(container.getExampleAnnotProp());

    for (FieldContainer fContainer : container.getFieldMetadata())
    {
            System.out.println(fContainer.getElementName());
            System.out.println(fContainer.getAttr());
    }

}
```

Figure 5: Reading Simple Metadata

```
Player \\ The name of the class with metadata
Player \\ The attribute inserted in the annotation @Annotation1
field1 \\ The name of the field
attr1  \\ The attribute inserted on the @Annotation2 on field1
field2 \\ The name of the field
attr2  \\ The attribute inserted on the @Annotation2 on field2
```

Figure 6: Output code

As seen in this example, the developer does not have to write Reflection code directly and has an intuitive way of retrieving metadata. The developer can map fields decoratively to receive information about specific annotations or to retrieve all annotations from methods, fields, and classes. By using this API, the developer is guided into using good practices to recover metadata, even if unaware. The creation and mapping of the class in the role of the metadata container can seem unnecessarily complicated to retrieve data of a simple set of annotations. Still, for a more complicated metadata schema, the gain in terms of maintainability might worth it.

4 EVALUATING METADATA-BASED FRAMEWORK DEVELOPMENT API

In this section, we present the methodology used to evaluate the Esfinge Metadata API. We begin discussing the research question that guided our work. Then we show how our experiment was carried out in terms of the participants' selection and how the target framework was divided into tasks, where each task added new functionality to the framework. To analyze our data, we performed a careful code inspection with the aid of source code metrics and also the time spent on each task.

4.1 Research Questions

The goal of this experiment is **to analyze** APIs for code annotation reading **for the purpose of** exploring **with regard to** object-oriented metrics **from the viewpoint** of experienced and skilled Java developers, **in the context of** development and evolution of metadata-based frameworks [2]. We focus this study on the following research question:

RQ: How does the Esfinge Metadata API provide support to maintain complexity and coupling in the evolution of a metadata-based framework compared to the Java Reflection API?. To answer this question, we performed a historical analysis and observed how object-oriented code metrics values evolved during the development. We analyzed well-established metrics such as Number of Methods (NOM) and others from the CK suite (as described in Section 4.2.4). Regarding these metrics, our hypothesis is composed of two parts. First (i), the complexity should be lower in codes that used our Esfinge Metadata API since some logic is being executed through annotations reading and processing. As for the Java Reflection API, similar rules are supposed to be implemented, using imperative code. Second (ii), the coupling should be lower in codes that used our Esfinge Metadata API if we do not consider the coupling to annotations. That is because several links should be bound by metadata configuration and not by method calls.

4.2 Experimental Design

From a practical point of view, we cannot measure an API's impact during the development of a small framework. We conducted an exploratory experiment to overcome this challenge and observe how API usage influences code evolution. The required time for implementation was much longer than other controlled experiments that last, on average, 110 minutes [14]. Due to this implementation time, it is not viable for the participants to implement all the features in a continuous period. The division of the experiment in tasks allowed the participants to perform it in small portions, being able to take a break between them.

Table 1: Time spent to conclude the experiment

API	Participant	Time Spent	Execution
Esfinge Metadata API	P1	6 days (432 min)	Correct
Esfinge Metadata API	P2	5 days (810 min)	Correct
Esfinge Metadata API	P3	4 days (733 min)	Correct
Esfinge Metadata API	P4	8 days (773 min)	Correct
Esfinge Metadata API	P5	2 days (318 min)	Correct
-	**Group Average**	**5 days (613 min)**	-
Reflection	P6	11 days (1110 min)	Correct
Reflection	P7	12 days (472 min)	Correct
Reflection	P8	21 days (611 min)	Incorrect
Reflection	P9	1 days (389 min)	Correct
Reflection	P10	6 days (350 min)	Correct
-	**Group Average**	**10.2 days (586 min)**	-
-	**Average**	**7 days (545 min)**	-

We provided an automated test suite to ensure that a given behavior is implemented at the end of each task. Thus, the task is considered complete only after the tests are passing. Afterward, the participant should commit the code to the repository. Given the nature of this experiment and the time spent executing the tasks, each participant remotely performed the experiment. We modeled our tests with a blackbox approach, considering only input and desired output. Hence the tests did not consider the internal structure used by the participants to develop their target framework

During the selection of the participants, we observed that the execution time of tasks varied greatly between them. Hence we concluded that a careful analysis of time would not be very meaningful to our experiment.

In the following subsections, we describe the design of our experiment based on the recommendations of Ko *at al.* [14].

4.2.1 Participants Selection and Training. We initially recruited 47 professional developers with at least three years of Java experience. The demographic data was gathered to understand better their level of expertise, background, and experience with the required tools. With this information, we were able to decide which participants were qualified to be in our experiment.

To select the participants, we applied a test to assess their knowledge of Java Reflection. We invited them to develop a short program in which the participants would have to demonstrate their skills in advanced programming techniques through reflection and code annotations. As a selection criterion, they would have to implement the activity correctly and in less than one hour. As a result, 10 participants completed the exercise successfully and agreed with the terms of the study. In short, we submitted the candidates through a rigorous selection process and selected ones with suitable programming skills. Once the selected participants agreed to participate in the study, they all signed a consent form.

During the recruitment stage, all the candidates watched a set of video lectures on Java Reflection API, Code Annotations, Apache Bean Utils, and metadata-based frameworks to align the knowledge needs for the experiment. Later, after the group's assignment, the participants of the second group – the one required to use the Esfinge Metadata API – received further training by dedicated video lectures about our Esfinge Metadata API.

As the participants executed the experiment, they filled a diary where they made comments and personal notes/opinions. We did not consider this, however, when performing the code analysis, since the information we needed was in the provided code.

Finally, this research was funded, and we were able to reward the participants. At the end of the experiment, we also debriefed the participants by explaining (i) what exactly the study was investigating; (ii) why the research was necessary; (iii) how we were going to use the data; and that (iv) we were going to spread the results as soon as we were finished analyzing the data.

4.2.2 Procedure. Firstly, we created a GitLab repository for each participant with the initial configuration already set up. All these repositories belong to a group[4].

A pilot experiment was conducted to observe if the target framework and the instructions were adequate. Two authors of this paper executed the experiment, and the first round of revision was performed for general improvement. One author used the Java Reflection API purely, and a second author used our Esfinge Metadata API. Afterward, four graduate students also executed the experiment, and a final round of revision was done. At this point, we considered the experiment ready to be carried out with professional programmers. We did not address these subjects in the analysis.

Before starting the experiment, we provided a set of detailed instructions to the participants to prepare and test their environment, including the Eclipse IDE, FLUORITE plugin, and the GitLab repository access. After that, the participants read the problem and the framework description to start the experiment tasks. We also provided a detailed set of general instructions to the participants for each task. That guide cover unit tests, logging, time measurement, criteria to consider a task finished, and instructions to commit the code to the repository.

Although we designed the experiment to be performed in several days, the participants were advised to be prepared to execute a task without interruption. We instructed the participants to reserve at least 2 hours to work on each task. If an unforeseen interruption is necessary, the participant should pause the time measurement and register it in FLUORITE.

The instructions also specify a protocol if the participant has some problem and its unable to continue the implementation. We oriented the participant not to spend more than half an hour trying to solve a problem. The procedure to ask for help requires the participant to pause the execution following the same instructions of an unforeseen interruption.

A different author of this paper executed the experiment using both approaches after the experiment was entirely carried out, with every participant handling their source code in the repository. He provided a reference source code that we used for the qualitative analysis. This verification was done through mining the participant's repositories and carefully observing the source code and its evolution.

4.2.3 Tasks. According to Sjoberg *at al.* [21], one of the challenges in controlled experimental design is the trade-off between realism and control. The task design is at the heart of this trade-off, as tasks represent the essence of realistic and messy software engineering work [14].

One aspect is to consider what features the target framework will contain and how they will be broken down into tasks for the participants to execute. From previous research outcomes [7, 9–11, 17], our group was able to identify common characteristics of metadata-based frameworks and used such knowledge to prepare the tasks to have good feature coverage.

These frameworks validate the metadata inserted in the programming elements before they are processed to execute custom behavior. For instance, some annotations can only be configured on specific element types, or the values of the attributes may have some restrictions. As such, there are tasks where the participants will execute routines that validate metadata.

Metadata reading and processing are also two common features present in metadata-based frameworks. The first is about identifying the code annotation and the code element. The second is regarding executing the behavior. The target framework developed by the participants also contained such tasks. Finally, there was also a task about the introduction of an extension point and others that impacts the framework control flow. With this, we believe that the target framework has the same characteristics and features that are implemented in a real one.

[4]gitlab.com/metadataexperiment

In practice, the participants implemented a metadata-based framework that maps the command-line parameter array received in the `main()` method to a JavaBean class. This mapping is done through metadata configurations, i.e., using annotations. The features of this framework were organized into tasks that tackle common features of a metadata-based framework. Following is a list describing the ten tasks.

1. **Mapping Boolean Properties**: Verify if a parameter is present on the command-line.
2. **Mapping String properties**: Store the String attribute passed after a parameter.
3. **Mandatory Parameters**: Verify if a parameter marked as mandatory is present on the command line. If not, an error occurs.
4. **Mapping Numeric properties**: Store the numeric attribute passed after a parameter.
5. **Validating text properties**: Validate the string attribute passed on the command line.
6. **Validating numeric properties**: Validate the number attribute passed on the command line.
7. **Supporting composite classes**: Provide functionality to map command-line attributes to encapsulated classes.
8. **Annotation extension**: Add an extension point to create custom annotations.
9. **Supporting parameter lists**: Stores a list of attributes, both numeric and textual, passed on the command line.
10. **Supporting dates**: Store the date attribute passed after a parameter.

The participant should perform these tasks strictly in the specified order and not read the subsequent tasks before finishing the current one. Each experiment task contains (i) a textual description of the feature to be developed; and (ii) a set of classes that consist of automated tests to check the correctness of the implemented feature.

Finally, the target framework tasks were also designed to represent realistic functionality that would be implemented for retrieving command-line parameters. Nonetheless, they were not based on features present in any of the APIs, emulating the development of a real-world metadata-based framework.

4.2.4 Qualitative Analysis with Source Code Metrics. Source code metrics are used to retrieve information from software and assess its characteristics. They 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 keep control of complexity, making them aware of the abnormal growth of specific system characteristics. To effectively take advantage of metrics, they should provide meaningful information and not just numerical values [16].

We used object-oriented code metrics to analyze the complexity and coupling of the source code from the developed target framework. Respectively, these metrics are from the CK suite [3], such as Weight Method Count (WMC) and Coupling Between Objects (CBO). We also use the Number of Methods (NOM) metric.

5 DATA ANALYSIS

In this section, we present the quantitative and qualitative analysis of the data collected from the experiment to answer our research question. We monitored the evolution of some selected software metrics as the participants were developing the framework. For the qualitative analysis, we performed an inspection of the source code developed by the participants to extract information about their use of both Java Reflection and our Esfinge Metadata API.

As mentioned in Section 4.2, we recruited and trained ten participants. However, participant P8, as presented in Table 1, did not execute the experiment correctly. As such, we excluded P8 from the analysis. Therefore the final sample has nine participants: four for the Java Reflection Group and five for the Esfinge Metadata Group.

We analyzed how three well-established software metrics behaved during the evolution of the target framework by both approaches designed for the experiment. These metrics are WMC (Weight Method Class), NOM (Number of Methods), and CBO (Coupling Between Objects). We designed the development of the target framework into ten tasks, and at the end of each task, we extracted these metrics values. We present graphs that show how these metrics behaved in the next sub-sections, and we use them to analyze the experiment further.

5.1 Complexity

The tool used to extract values for WMC has implemented it as McCabe's Complexity, i.e., counting the number of branch instructions in a class. We observe in Figure 7 that the Esfinge Metadata Group has larger WMC values. However, we also notice that these participants evolve in a similar pattern when compared to the Java Reflection Group. We expected this greater value since to use the API developers are required to write `getters` and `setters` to populate the `MetadataContainer` class, as part of the `MetadataContainer` design pattern. Developers on the Java Reflection Group, although it was mandatory to use the design pattern mentioned earlier, could take another route during the development. We also know that every single method, regardless of its complexity, adds a number to the WMC count. As such, we are purely looking at this number might not be a true measure of complexity.

To complete the complexity analysis, we also looked at the Number of Methods (NOM) metric. The graph in Figure 8 presents how the NOM metric evolved during the development. This graph shows that the Esfinge Metadata Group created more methods, roughly twice the Java Reflection Group, which is in agreement with the higher WMC number presented in Figure 7.

The code evolution from two participants, P6 and P10, represented as dashed blue lines on both Figures 7 and 8 presented an abnormal value of WMC metric. However, observing Figure 8, we notice that these two participants had practically a constant value for NOM. This metric behavior suggests that these methods were constantly growing to accommodate the new features of the framework. WMC values were growing from the complexity of these methods rather than the addition of new methods.

From this analysis, we plotted a new graph that is the ratio of WMC per NOM, as presented in Figure 9. Thereby, we observed how the participants P6 and P10 has a curve with much higher values than all other participants, confirming they have created

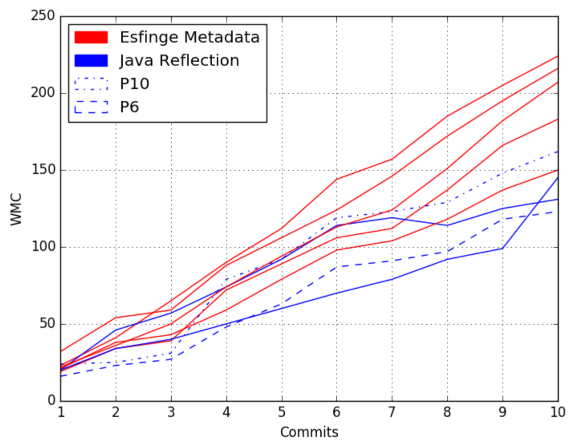

Figure 7: WMC produced overtime

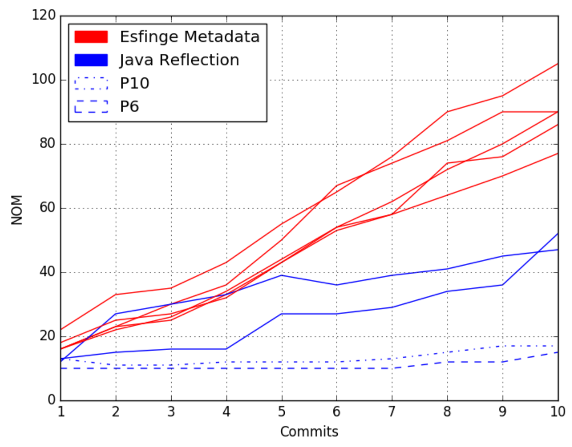

Figure 8: NOM produced overtime

highly complex methods. Moreover, all participants from the ous Group present lower values of this ratio, which confirms that even though a higher number of methods, their complexity is inferior. This information was not clear, only analyzing the WMC graph in Figure 7. With this analysis complete, it becomes clear why we included the NOM metric in this section. Although it does not measure complexity, it was used as a normalization factor for the WMC values.

5.2 Coupling

We calculated the CBO metric as Fan-Out, i.e., how many external classes a specific class refers too. Observing Figure 10, we notice that the Esfinge Metadata group has higher values and, therefore, presents more upper coupling. By analyzing two participants of the Java Reflection Group, P7, and P9, we observed unexpected CBO values. The latter presents a decrease in its CBO value after

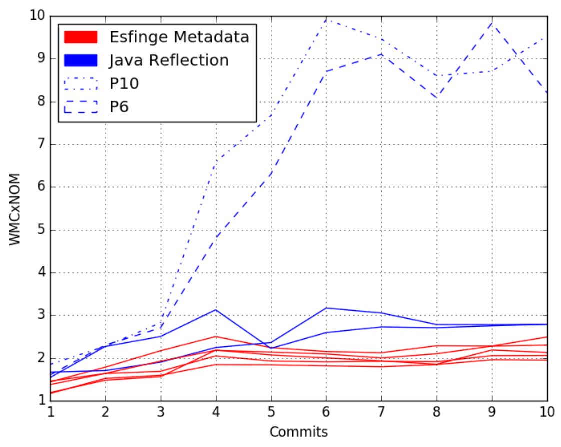

Figure 9: WMC x NOM produced overtime

completing Task #8, and the former shows an abrupt growth on the CBO value after the final task's conclusion.

During code inspection, it was not explicit why P9 had a decrease in its value. It was subtle the reason that caused this behavior. An important observation is that this participant was the only one in the Reflection Group that implemented the `MetadataContainer` design pattern as expected. This behavior means storing the metadata in a container class for later analysis. It is interesting to observe because its CBO value approximates that of the Esfinge Metadata Group, suggesting that this design pattern presents a higher coupling factor.

For this reason, P9 had separate classes responsible for metadata reading of a specific data type. During most of the development, an abstract class was used as the superclass for metadata reading classes, but, during Task #8, P9 switched to an interface. During the refactoring process of these classes, some became less coupled, which resulted in a lower CBO value overall.

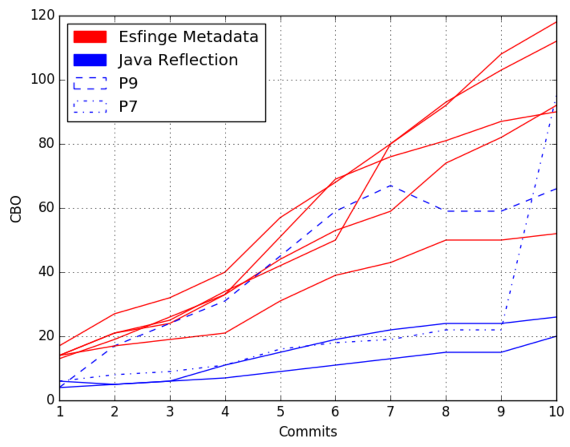

Figure 10: Coupling between objects (CBO) over time

As for participant P7, he was not implementing the pattern correctly `MetadataContainer`, i.e., he was not storing metadata in a container class. On the last task, however, he performed a significant refactoring of the code. In short, he created different classes to read and process the metadata according to its type, hence the CBO value had an abrupt growth.

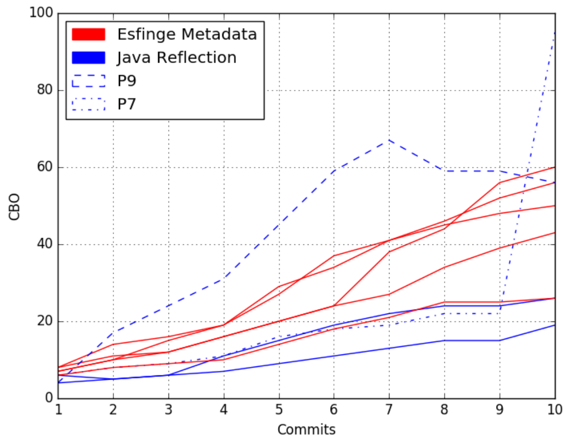

Figure 11: CBO without considering code annotations

Given that the Esfinge Metadata API is a metadata-based framework, it relies heavily on code annotations, which is considered when calculating the CBO metric. We altered the CBO calculation of our metric extraction tool and generated a new graph in Figure 11.

This graph shows how the CBO evolved without considering code annotations. As expected, the CBO values for the Esfinge Metadata Group had a decrease, lower than P9, for most of the development, and P7, after the significant refactoring. The two developers from the Java Reflection group that stayed below are P6 and P10. As seen in the WMC analysis, these two participants did not implement the `MetadataContainer` design pattern correctly. They created large and complex methods to avoid separating the functionality of metadata reading and processing. The complexity of their methods outweighs the lower coupling values they presented since the produced code shows lower readability and more effort to maintain.

Overall, from Figure 11, the Esfinge Metadata Group presents a more similar growth pattern, and it is more homogeneous than the Reflection Group. This behavior is a reflection of having a design pattern that is mandatory when using the Esfinge Metadata API. Developers using pure Java Reflection API have to build this architecture themselves, and as reinforced in this CBO analysis, may divert.

Another analysis is to observe how the CBO metric evolved as the number of classes evolved, i.e., normalizing the CBO value by the number of classes. This observation is similar to the analysis performed on how the WMC metric evolved as the NOM evolved presented in Figure 9. Figure 12 presents the normalized CBO graph. From this figure, it is clear that the Esfinge Metadata Group has

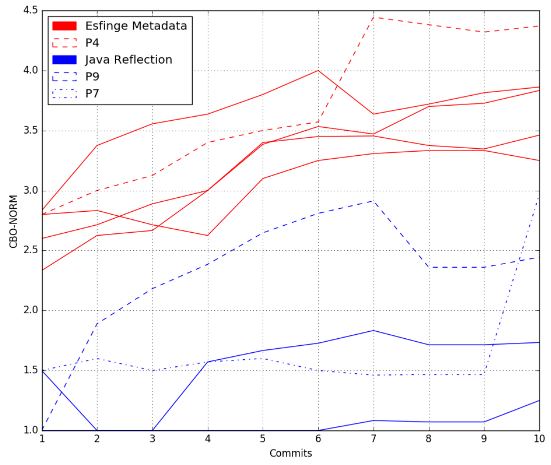

Figure 12: CBO normalized by the number of classes

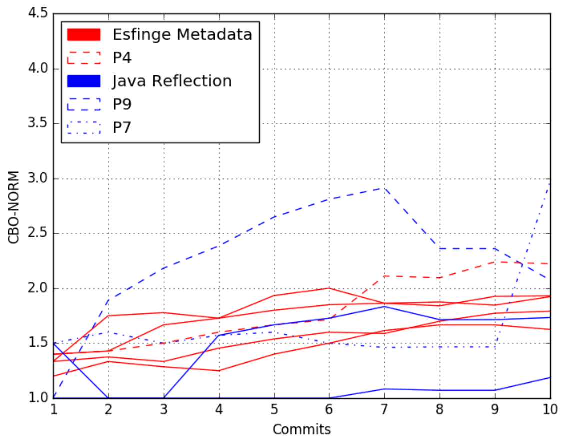

Figure 13: CBO normalized by the number of classes, without considering annotations

more coupled classes. However, this graph also reinforces they have, overall, more homogeneous growth. However, participant P4 shows an abrupt growth on commit seven, surpassing other participants. This behavior indicates that the CBO is growing more rapidly than the number of classes, which suggests high coupling in the framework's classes. After code inspection, one reason for this growth was the use of unnecessary Esfinge Metadata API annotations for validation, since they have no effect where it was placed. Using just the CBO graph in Figure 10 this observation is not clear. To analyze the normalized CBO without considering annotations, we generated the graph in Figure 13. From this, we see that most developers on the Esfinge Metadata Group have very similar CBO values, while the Reflection Group is more scattered. This factor is yet another confirmation that the CBO value from the Esfinge Metadata Group is strongly related to code annotations. It also has the API enforcing the use of the `MetadataContainer` pattern leads towards a homogeneous code even from different developers.

6 DISCUSSION

Answering our **RQ (*How does the Esfinge Metadata API provide support to maintain complexity and coupling in the evolution of a metadata-based framework compared to the Java Reflection API?*)**, we observed how three metrics – WMC, NOM, and CBO – evolved during the evolution of the target framework, dividing the analysis into two parts: complexity and coupling.

We were expecting that the complexity for participants using our Esfinge Metadata API would be lower, and the analysis concluded that. We used the WMC metric to measure the complexity, but as shown and discussed, purely analyzing its values would not suffice, since simple access methods contribute to the metrics values. To further investigate the complexity, we normalized the WMC value per NOM. Thus, we were able to identify that methods from the Esfinge Metadata API Group were less complex than methods created by the Java Reflection Group. Moreover, this was a consequence of having the `MetadataContainer` pattern being implemented correctly by the Esfinge Metadata API Group since the API also guides the developers to use best practices.

For the coupling analysis, we used the CBO metric. The expected result was that the overall coupling would be lower for the Esfinge Metadata API Group if coupling to code annotations were not considered. The analysis showed that the group using Esfinge Metadata API has higher CBO value, i.e., more coupled classes. What was observed is that developers from the Reflection Group that did not use the `MetadataContainer` pattern correctly had low values for CBO. However, they had more complex methods. Developers who did implement the pattern successfully had higher coupling values, including both Esfinge Metadata API and Java Reflection Group. Hence we conclude that the pattern has a high coupling factor. When code annotations were removed from the calculation, the CBO values for the Esfinge Metadata API Group dropped roughly fifty percent, while it stayed stable for the Java Reflection Group. This behavior shows that the coupling for the Esfinge Metadata API Group has a substantial contribution from code annotations, which means the coupling is due to metadata configuration and not method calls. We conclude that using a metadata approach to implement the `MetadataContainer` pattern increased the coupling factor.

Overall, our Esfinge Metadata API allowed developers to have a steady evolution of the code while developing the target framework. The main reason is that our API guides the developers in using best practices, precisely, the `MetadataContainer` pattern. Although participants from the Java Reflection API group also had to use this pattern, it was still possible to diverge and create more complex code. This behavior became apparent during the code inspection phase.

There is a learning curve required to use our Esfinge Metadata API, and although it does not require developers to directly deal with reflection code, they still need to have a firm grasp of the involved concepts. During our inspection, we observed that codes using our API were much cleaner and readable when compared with code directly using the Java Reflection API. This is the main goal of our API, that is, to guide developers towards best practices when implementing metadata-based frameworks or code annotations-based solutions. Creating such kinds of solution does not necessarily mean less effort.

7 THREATS TO VALIDITY

In this section we present the threats to validity of our work following the guidelines proposed by [24].

- *Conclusion Validity*: Our findings were based on manual code inspection, therefore our own coding skills and knowledge of Java Reflection might have affected our conclusions.
- *Internal Validity*: Initially, the participants received the same training through a set of video lectures on the topics addressed by the experimental study. We applied an exercise to evaluate each participant. Thus, we distributed them in balanced groups in terms of their performance. The Esfinge Metadata API group received further training to learn about this new API, which also potentially reinforced their knowledge about reflection code and techniques.
- *Construct Validity*: The Esfinge Metadata API guides the developers towards best practices, while the Java Reflection API does not. Hence, developers using Java Reflection had nothing forcing them to keep following these practices, which may have led to low quality code.
- *External Validity*: The experiment was conducted remotely by the participants in their own environment. They were also responsible for managing their execution time. Some developers might have had a more adequate environment than others, or less interruption during the experiment.

8 RELATED WORK

We exhaustively searched and did not find other solutions of APIs for metadata reading in this context. However, we have related works about studies involving annotated code and metadata-based frameworks. Yu et al. [25] perform a large-scale and empirical study about Java annotation on 1,094 open-source Java projects hosted on GitHub. The authors presented 10 novel empirical findings about Java annotation usage, annotation evolution, and annotation impact. For instance, the authors show that annotations are actively maintained, and most of their changes are consistent with other code changes.

Regarding annotation definition, Rocha and Valente [19] investigated how annotations are used in open source Java systems. The authors analyzed 106 open source projects from Qualitas Corpus project database [22], from which 65 projects used annotations. Only the number of annotations and their type was considered in this study. In some of the evaluated systems, a high density of annotations was detected, 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.

Alba [1] performed a study regarding legibility on annotated code. 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. The study pointed out that annotated code is perceived as more legible than the unannotated one. Besides that, the usage of annotation idioms [8] can improve annotation readability and context where the annotation was used has an influence on the perception of legibility.

9 CONCLUSION

In this paper, the main contribution is the proposal of a novel API approach to support the development of metadata-based frameworks, as well as applications based on custom annotations. The proposed API provides features based on frameworks needs, such as (a) support to search annotations in other code elements related to the target code element; (b) mapping for class metadata and annotation attributes; (c) chain processing of methods and field metadata; (d) support for the implementation of an extensible metadata schema; and (e) extension point that allows the creation of new metadata reading annotations.

We conducted an experiment that compared our Esfinge Metadata API with the Java Reflection API, evaluating the effects of their usage under various aspects related to software quality and development practices, such as code complexity and coupling. As a result, our Esfinge Metadata API allowed developers to have a more stable evolution of the code while developing the target framework. The detailed code inspection revealed evidence that the proposed API guides the developers into using best practices, not necessarily with less effort. Although participants from the Java Reflection API group were also instructed to use a similar structure, the experiment revealed that in some cases, the developer could diverge to add accidental complexity, ending up with a higher coupling.

As future work, we intend to add new features in our Esfinge Metadata API, especially to enable the support for code conventions and external metadata configuration. We also plan to carry out other studies about the usage of the proposed API, such as evaluating it from the Developer eXperience (DX) [5, 15] point of view and by using it in the development of projects.

ACKNOWLEDGMENT

We would like to thank the support granted by Brazilian funding agencies CAPES (Higher Education Improvement Coordination) and FAPESP (São Paulo Research Foundation, grant 2014/16236-6 and 2019/12743-4).

REFERENCES

[1] A. Alba. 2011. *Code Legibility Analysis by Means of Annotation Patterns.* Technical Report. Aeronautical Institute of Technology, Brazil. [in portuguese].

[2] Victor R Basili and H Dieter Rombach. 1988. The TAME project: Towards improvement-oriented software environments. *Software Engineering, IEEE Transactions on* 14, 6 (1988), 758–773.

[3] S. R. Chidamber and C. F. Kemerer. 1994. A Metrics Suite for Object Oriented Design. *IEEE Trans. Softw. Eng.* 20, 6 (June 1994), 476–493. https://doi.org/10.1109/32.295895

[4] ECMA. 2017. ECMA - 334: C# Language Specification. https://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf

[5] Fabian Fagerholm and Jürgen Münch. 2012. Developer experience: Concept and definition. In *Proceedings of the International Conference on Software and System Process.* IEEE Press, 73–77.

[6] Eduardo Guerra. 2014. *Componentes Reutilizáveis em Java com Reflexão e Anotações* (1st ed.). Casa do Código. [in portuguese].

[7] Eduardo Guerra, Felipe Alves, Uirá Kulesza, and Clovis Fernandes. 2013. A reference architecture for organizing the internal structure of metadata-based frameworks. *Journal of Systems and Software* 86, 5 (2013), 1239 – 1256. https://doi.org/10.1016/j.jss.2012.12.024

[8] Eduardo Guerra, Menanes Cardoso, Jefferson Silva, and Clovis Fernandes. 2010. Idioms for Code Annotations in the Java Language. In *Proceedings of the 8th Latin American Conference on Pattern Languages of Programs* (Salvador, Bahia, Brazil) *(SugarLoafPLoP '10).* ACM, New York, NY, USA, Article 7, 14 pages. https://doi.org/10.1145/2581507.2581514

[9] Eduardo Guerra, Jerffeson de Souza, and Clovis Fernandes. 2013. *Pattern Language for the Internal Structure of Metadata-Based Frameworks.* Springer Berlin Heidelberg, Berlin, Heidelberg, 55–110. https://doi.org/10.1007/978-3-642-38676-3_3

[10] Eduardo M Guerra, Fábio F Silveira, and Clóvis T Fernandes. 2009. Questioning traditional metrics for applications which uses metadata-based frameworks. In *Proceedings of the 3rd Workshop on Assessment of Contemporary Modularization Techniques (ACoM'09), October*, Vol. 26. 35–39.

[11] Eduardo Martins Guerra, Jerffeson T De Souza, and Clovis T Fernandes. 2009. A pattern language for metadata-based frameworks. In *Proceedings of the 16th Conference on Pattern Languages of Programs.* ACM, 3.

[12] JSR. 2004. JSR 175: A Metadata Facility for the Java Programming Language. http://www.jcp.org/en/jsr/detail?id=175

[13] JSR. 2007. JSR 220: Enterprise JavaBeans 3.0. http://jcp.org/en/jsr/detail?id=220

[14] Andrew J Ko, Thomas D Latoza, and Margaret M Burnett. 2015. A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering* 20, 1 (2015), 110–141.

[15] Kati Kuusinen, Helen Petrie, Fabian Fagerholm, and Tommi Mikkonen. 2016. Flow, intrinsic motivation, and developer experience in software engineering. In *International Conference on Agile Software Development.* Springer, 104–117.

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

[17] Phyllipe Lima, Eduardo Guerra, Paulo Meirelles, Lucas Kanashiro, Hélio Silva, and Fábio Silveira. 2018. A Metrics Suite for code annotation assessment. *Journal of Systems and Software* 137 (2018), 163 – 183. https://doi.org/10.1016/j.jss.2017.11.024

[18] Phyllipe Lima, Eduardo Guerra, Marco Nardes, Andrea Mocci, Gabriele Bavota, and Michele Lanza. 2017. An Annotation-based API for Supporting Runtime Code Annotation Reading. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Meta-Programming Techniques and Reflection* (Vancouver, BC, Canada) *(Meta 2017).* ACM, New York, NY, USA, 6–14. https://doi.org/10.1145/3141517.3141856

[19] H. Rocha and H. Valente. 2011. How Annotations are Used in Java: An Empirical Study. In *23rd International Conference on Software Engineering and Knowledge Engineering (SEKE).* 426–431.

[20] Don Schwarz. 2004. Peeking Inside the Box: Attribute-Oriented Programming with Java 1.5, Part. http://archive.oreilly.com/pub/a/onjava/2004/06/30/insidebox1.html

[21] D. I. K. Sjoberg, B. Anda, E. Arisholm, T. Dyba, M. Jorgensen, A. Karahasanovic, E. F. Koren, and M. Vokac. 2002. Conducting realistic experiments in software engineering. In *Proceedings International Symposium on Empirical Software Engineering.* 17–26. https://doi.org/10.1109/ISESE.2002.1166921

[22] E. Tempero, C. Anslow, J. Dietrich, T. Han, Jing Li, M. Lumpe, H. Melton, and J. Noble. 2010. The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific.* 336–345. https://doi.org/10.1109/APSEC.2010.46

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

[24] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering.* Springer Science & Business Media.

[25] Z. Yu, C. Bai, L. Seinturier, and M. Monperrus. 2019. Characterizing the Usage, Evolution and Impact of Java Annotations in Practice. *IEEE Transactions on Software Engineering* (2019).