Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

ARTICLE TYPE

An Approach Based on Metadata to Implement Convention over Configuration Decoupled from Framework Logic

Everaldo Gomes*[1] | Eduardo Guerra[2] | Phyllipe Lima[3] | Paulo Meirelles[1]

[1]Institute of Mathematics and Statistics, University of São Paulo, São Paulo, Brazil
[2]Org Division, Free University of Bozen-Bolzano, Bolzano, Italy
[3]Institute of Mathematical and Computing Sciences, Federal University of Itajubá, Minas Gerais, Brazil

Correspondence
*Corresponding author name, This is sample corresponding address. Email: everaldogjr@gmail.com

Present Address
This is sample for present address text this is sample for present address text

Summary

Frameworks are essential for software development, providing code design and reuse for its users. Well-known Java frameworks and APIs such as Spring, JUnit, and JPA rely on metadata usage, commonly defined by code annotations. Those frameworks use the Java Reflection API for consuming and processing these annotations. Code elements usually have some similarities and can also have the same annotation. This paper proposes a model for defining conventions over configuration for annotations usage decoupled from the metadata reading logic. With this model, if a convention is present, the framework will consider that the element is configured by a specific target annotation, even if the code element does not have the annotation. We implemented this model in the metadata reading framework Esfinge Metadata API. The model implementation was evaluated refactoring an existing framework to add support to conventions using our approach. As a result, it was possible to introduce the conventions only by adding configurations to the annotations. The model was further evaluated in an experiment in which participants implemented the Conventions over Configuration pattern using the Esfinge Metadata API and Java Reflection API. Based on the results, approach fulfilled its goal of supporting the definition of conventions decupled from the framework logic, making the code more readable and easier to maintain according to the participants perception.

KEYWORDS:
metadata, code annotation, code convention, convention over configuration, frameworks

1 | INTRODUCTION

Frameworks can be considered incomplete software that can be specialized to add application-specific behavior. By providing a set of abstractions, frameworks can be composed and extended to create a complete application. Metadata-based frameworks[1] use metadata configuration to configure how the behavior should be adapted for each class. In the Java Language, code annotations constitute the primary approach for metadata configuration. Highly used frameworks, such as Spring, Hibernate, and JUnit, use annotations as the core of their APIs.

Through years after the addition of this language feature, we have empirical evidence that using annotations is pervasive in modern software development processes[2 3 4 5]. However, considering that several frameworks are adopting an annotated API approach, there is the risk of annotation overuse, which might harm the code readability and maintenance. We can find reports from practitioners about that problem[6 7 8 9] and empirical studies that evidence the existence of classes overloaded with annotations and highly repeated configurations[4 10 5].

Adopting code conventions, also called code standards, is considered an important practice when a development team collaborates in a code base[11]. However, they can be used as an approach for metadata definition[12] based on a pattern called `Convention over Configuration`[13]. This practice can significantly reduce the number of code annotations and other types of configuration[10], being an alternative to the problems

related to repetition and overuse of annotations. However, a lack of support for this design practice limits its usage. Accordingly, no metadata-reading API or tool supports frameworks on the implementation of `Convention over Configuration`. Even if design patterns for reading metadata, such as `Metadata Reader Strategy` and `Metadata Reader Chain` [14,15], could be used for this implementation, the convention would still need to be checked, and the corresponding metadata identified programmatically. We discuss these design patterns in Section 2.3.

In this work, we propose a model for defining a solution to support the usage of conventions over configuration for metadata-based frameworks to aid developers. Our approach uses additional convention annotations for configuring the framework annotations. The convention annotations define the convention that replaces the target annotations when the code element matches the convention. Accordingly, the metadata reading API will transparently return the same value when the framework annotation is present or the element matches the code convention. This way, the framework metadata reading logic can consider only framework annotations, letting the metadata reading API handle the conventions transparently. Additionally, the conventions will be declared in the target annotation definition, decoupled from the framework logic.

We created a reference implementation as a feature of the framework Esfinge Metadata [16 17 18] to evaluate the feasibility of our model. The Esfinge Metadata API is a meta-framework for reading and validating annotations. The mechanisms implemented in the Esfinge Metadata API are extendable. We extended the metadata reading, including our approach. Using the new version of Esfinge Metadata, we first conducted a case study by refactoring the Esfinge Comparison [1]. This tool already used Esfinge Metadata as its metadata reading API [17]. In this study, we refactored the comparison framework, adding convention annotations to the target annotations. We evaluated our solution using unit tests for each implemented convention. As a result, the conventions could be implemented by only adding their definition in the target annotations, decoupled from the comparison framework logic. As a second validation, to complement the case study, we also conducted an experiment with 28 undergraduate students using a subject framework for mapping command line parameters to class attributes. Each participant implemented two conventions using our proposed approach and two conventions using only the Java Reflection API. As a result, using our proposed approach, the added code was decoupled from the subject framework logic, and it was considered by the participants easier to read and maintain. Based on these results, we concluded that the proposed model fulfilled its goal as a suitable support to implement the Convention over Configuration pattern.

We defined the terminology used in this paper to guide the reader better. When referring to the Java annotations feature, we use the term **code annotation** or simply **annotation**. We use **convention annotation** to refer to the annotations that define the conventions. Finally, we use **target annotation** or **framework annotation** to refer to the ones that receive the convention. When we refer to frameworks in general, we say **framework**. About the frameworks developed or used in this research, **comparison framework** refers to the framework used in the case study, and **subject framework** refers to the framework used for evaluating our approach. Finally, when we say **reference framework** or **implementation framework**, we are referring to the Esfinge Metadata API, where we implemented our model.

The remainder of this paper is organized as follows: Section 2 presents the concepts for metadata in Object Oriented programming. Section 3 presents our novel approach for conventions over configuration. The model implementation is described in Section 4. We present in Section 5 a case study for implementing conventions for the Esfinge Comparison annotations. The study designed for evaluating our approach is presented in Section 6. Section 7 presents the results obtained, and Section 8 we discuss the results and present the threats to validity. Finally, we conclude this paper in Section 9.

2 | BACKGROUND

This section describes the software engineering concepts necessary for this work. We start discussing metadata configuration and how some programming languages implement this feature. We introduce the Code Conventions concept and how tools or frameworks use this practice. Then we discuss metadata-based frameworks, presenting the design patterns used by this framework to reduce the usage of annotations. Finally, we present the impacts of code annotations usage in code development.

2.1 | Metadata Configuration

The term “metadata” is used in various contexts in the computer science field. In all of them, it means data referring to the data itself. For instance, the metadata refers to the table’s structure in the database context. In the object-oriented context, the data are the object instances, and the metadata is the class definition. A class field, in turn, has its type, access modifiers, and name as its metadata. When a developer uses reflection, the metadata of a program can be accessed and manipulated, enabling the code to work with previously unknown classes [1].

One approach to defining custom metadata is to use external files, such as an XML file or a database [12]. However, in this approach, the metadata and the code element are distant since the external file needs to reference the element. Another drawback is the verbosity of this approach since the complete path must be provided so the framework may consume the metadata correctly. Another alternative some frameworks use for metadata definition is code conventions, which we further explore in subsection 2.2.

Some programming languages provide features that allow custom metadata to be defined and included directly on programming elements, like attributes in C# [19] and code annotations in Java [20]. This approach allows the definition of metadata closer to the programming element, being less verbose than using an external XML file. On top of that, the metadata is explicitly defined in the source code as opposed to other approaches. Some authors call the usage of code annotations attribute-oriented programming since it marks software elements [21] [22]. Annotations can be used for source code generation [23], compile-time verification [24] [25], class transformation [26], and framework adaptation [1].

Some base APIs, starting in Java EE 5, like EJB (Enterprise Java Beans) 3.0, JPA (Java Persistence API) [27], and CDI (Context and Dependency Injection), extensively use annotations to configure metadata. This native annotation support encouraged many Java frameworks and API developers to adopt the metadata-based approach in their solutions. They are also a response to the tendency to keep the metadata files inside the source code instead of using separate files [28]. The Java language provides the Java Reflection API, allowing developers to retrieve code annotations at runtime.

2.2 | Code Conventions

Code conventions are rules to guide developers to a programming style, practices, or methods. The rules defined by the code conventions include good naming of code elements that connect the identifiers to the domain and architecture role [29] and code formatting according to the control flow [30]. The goal of these recommendations is to develop secure and easier-to-maintain code [31], less error-prone [32], and more readable [33]. The no conformity with conventions is a recurrent issue pointed out by code reviewers, according to a recent study [34].

Code conventions can be used as an alternative to defining code metadata to be consumed by frameworks and APIs [12]. When consuming metadata, the framework uses information about the element, such as the identifier, to infer additional information about it. For instance, the JavaBeans specification [35] defines that `get` and `set` should be used as the prefix to identify methods that respectively retrieve and set information into a class. Reflection-based frameworks and APIs usually rely on application classes following some specific convention to execute their functions.

A pattern called Convention Over Configuration [13] [36] states that frameworks and APIs should enforce standard naming conventions for mapping classes to resources or events. Ruby on Rails is a framework known for adopting this practice as the core of its solution [37]. However, other frameworks like ASP.NET MVC, CakePHP, Laravel, Symfony, Yii, Grails, Ember.js, Meteor, Nest.js, and Spring Framework followed it. A complementary practice called Configuration by Exception considers that metadata is defined on elements that are the exception to the convention rule. For instance, the JPA API considers as a convention for object-relational mapping that fields of a class have the same name as its respective column in the database table. Requiring it to be configured explicitly by annotations only when this assumption does not apply [38].

Regardless of enhancing productivity in some contexts, these approaches have limited expressiveness. Thus it cannot be used for defining complex metadata. For instance, a code convention would be suitable to define a test method in a test framework (like JUnit). However, it would not define a valid range of numeric values in an instance validation API (like Bean Validation). A drawback of this approach is that the metadata are not explicit in the source code can lead to errors [39]. For instance, by changing a class name, a developer might inadvertently change the metadata that drives the application's behavior.

To our knowledge, no API or library offers any feature to support metadata reading based on code conventions. Currently, frameworks that support code conventions must implement them as part of their metadata reading algorithm. The metadata reading algorithms must verify if the code element matches the code convention, then it infers the metadata is present. There are documented design patterns for metadata reading and processing [14,15], further discussed in Section 2.3, that can be used as an alternative to decoupling metadata extraction from code conventions during the annotation reading and processing. However, even with these patterns allowing the extension of new metadata reading mechanisms, we did not find any register of this practice's usage to implement code conventions into a framework.

A particular type of convention used with annotations is to use a custom annotation defined by the application and configure the framework annotations to this custom annotation. A pattern called Annotation Mapping [40] implements this approach. The framework annotations are added to the definition of an application annotation, and the metadata reading mechanism search is also inside other annotations present in the target code element. Esfinge Metadata [4] [18], and the Daileon API [41] provide support to implement this pattern transparently. Some official Java APIs, such as CDI (Contexts and Dependency Injection for Java) [42], and Bean Validation [43], also support the Annotation Mapping mechanism. That can be used to implement domain annotations [44], that represent at a higher level domain metadata, but it is mapped to low-level framework metadata.

2.3 | Metadata-based Frameworks

Recent 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 [1]. Popular Java frameworks such as Spring, Hibernate, and JUnit use code annotations to configure metadata.

Metadata-based API can be designed using practices not present in the traditional object-oriented design. Several practices aim to reduce the usage of annotations. A previous work of Guerra et al. [40] shows several patterns used by metadata-based APIs. Applying two design patterns can

reduce the annotation usage, the *General Configuration* and the *Annotation Mapping*. The *General Configuration* proposes that the class definition must receive annotations instead of the methods or attributes. When configuring the class with the annotation, the framework should consider that all class elements have that default configuration. Frameworks like JAX-RS 1.1 [45] and EJB 3 API [46] implement this approach for some annotations. The *Annotation Mapping* can reduce the number of annotations on elements with common characteristics. To achieve this, developers must create a custom annotation and insert the framework annotations. Components can search for extra configuration at runtime, searching the annotations inside the created custom annotation. The Bean Validation API [47], Java EE [48], and Daileon [41] are an example of frameworks that implement this approach for reducing the number of annotations in the code element.

Many different contexts employ Metadata-based frameworks. A previous work of Guerra et al. [49] catalogs the usage scenarios of metadata-based frameworks, the problems they aim to resolve, and how they implement the solutions. The authors identified four patterns. (i) *Entity Mapping*, where metadata maps an application class to a different representation. Hibernate [50] and JPA [38] support mapping object-oriented paradigms to relational databases. (ii) *Metadata-based Graphical Component*, is a special case of the Entity Mapping pattern. This pattern uses metadata to generate graphical components based on application classes and the interaction with them [51,52]. (iii) *Configured Handler* uses the metadata to mark methods that should handle events [53 54]. (iv) *Crosscutting Metadata Configuration* uses metadata to configure variations on a crosscutting behavior [55]. The *Dependency Injection* [56] specifies that annotations configure an instance that should be injected into the class. The annotated class field indicates that an instance should be set to that field. Factories can be used as an alternative to annotation usage. The *Rule Definition* [57] specify that annotations can configure parameters for processing related to the target class.

Internally, metadata-based frameworks implement several design patterns. Design patterns for developing metadata-based frameworks can be classified into three categories [14]. (i) *Structural Patterns* documents the best practices on how the framework classes must be structured internally. (ii) *Logic Processing Patterns* are responsible for documenting solutions on how to design the main framework logic, allowing it to be extended and modified. Finally, (iii) *Metadata Reading Patterns* are used to document recurrent solutions about metadata reading by the frameworks. The `Metadata Container` design pattern can be employed to decouple the metadata reading and the application logic. This pattern is implemented by the subject framework and is further detailed in Section 6. The `Metadata Reader Strategy` uses one interface to abstract how metadata must be read, and different classes can implement their metadata reading approach allowing it to be read from different sources. Since conventions can be part of the metadata reading, the conventions reading can be one implementation of the `Metadata Reader Strategy` design pattern. Metadata can also be dispersed in more than one class or package. The `Metadata Reader Chain` provides the structure necessary for retrieving metadata from the different sources. This design pattern can be implemented as a *Chain of Responsibility* [58]. Using the `Metadata Reader Chain` allows the implementations of `Metadata Reader Strategy` to combine the metadata into the same `Metadata Repository`. In this design pattern, a container class stores metadata in runtime. The repository manages the metadata reading and is accessed by the framework to retrieve metadata. The Esfinge Metadata API [18] is a metadata-based framework developed to improve the metadata reading and processing of metadata-based frameworks. The Esfinge Metadata API implemented the previously discussed design patterns and was used to implement our proposed approach of adding conventions capability to the metadata reading and processing.

2.4 | Studies About the Impact of Using Annotations

Nowadays, several popular APIs and frameworks use annotations, resulting in many classes with annotations in existing projects [59,5]. Empirical studies presented evidence of very large annotations, classes with a high number of annotations [59], annotations repetition [10], and that they might introduce code problems [5].

Experiments comparing metadata-based solutions with others more based on regular object-oriented programming pointed out that the functionality indirectly defined through metadata configuration makes it hard to debug [60,61]. These studies also pointed out that the usage of annotations might reduce the coupling.

The study of Lima et at. [59] presents a suite of metrics for annotations usage in source code. The authors observed metric outliers by evaluating the data of 25 open-source Java projects, revealing abuses in using the annotation feature. For instance, the class `CoreMessageLogger` from Hibernate Core project has 1429 lines of code, of which 789 are annotations. The `@LogMessage` annotation is used 359 times in this class in all its methods.

A study on software from the Brazilian space weather program appointed the redundancy of configurations in code elements with the same characteristics [10]. As a result, the authors discovered that the same annotation might occur many times in many different classes. For instance, `@Column` annotation from JPA appeared more than 600 times in the analyzed project. As the main result, the authors found a total of 908 annotations (21.42% of the total) that 17 application-specific conventions could replace. The authors suggest that using code conventions has the potential to reduce hard-to-resolve inconsistency and improve the code structure.

Yu et al. [5] collected data from 1094 open-source Java projects and conducted a historical analysis to assess code annotations in a large-scale empirical study assessing the code annotations' usage, evolution, and impact. The study revealed that many changes in annotations occurred during

project evolution, implicating that some developers tend to use annotations subjectively and arbitrarily, introducing code problems. The authors suggest that developers associate annotation usage with code error-proneness. The study concluded that they could potentially enhance software quality. Another result from this study was identifying a solid relationship between developer ownership and usage of annotations, revealing that they might have difficulty contributing to the metadata configuration in a code created by others.

Recently, Guerra et al.[18] conducted an experiment on framework development. The authors compared the usage of annotation-based API against one approach based on object orientation for metadata reading using the Reflection API. The results present a more consistent behavior in the evolution of coupling and complexity metrics using an annotation-based approach. However, no significant differences in productivity were found.

Lima et al.[62] developed a software visualization approach to improving code comprehension of annotation-based systems to observe the code annotations distribution in a given project. The authors conducted an empirical evaluation with students and professional developers using a Java web application as the target system. From their findings, the authors observed a strong relationship between the presence of a code annotation and the responsibility of the package/class using that annotation. This evidence supports the claim that other information from the classes, such as its package, might be used as a convention to define metadata. This finding suggests that code annotations may also be used to assess the architectural design of software systems.

In short, the impacts of annotation usage have positive and negative effects. Annotations can reduce the coupling[60], provide a consistent code evolution[18], and improve the code quality[5]. The literature presents some drawbacks to the usage of annotations. Annotations declaration can be extensive[59] and used to configure similar code elements repetitively on the source code[10]. Another negative impact is that annotations can make the code hard to debug[60]. Annotations can be wrongly used, inconsistent, and redundant in the source code[5]. Finally, a study on developers' perceptions of code readability could not reach a consensus about annotation usage and code readability[63].

Several strategies aim to reduce the usage of annotations. The *General Configuration*[40] proposes using annotation in the class to enable a default configuration for the class elements. *Annotation Mapping*[40] can reduce the annotation number by suggesting developers create a custom annotation and configure it with the framework annotations. Some metadata-based frameworks implement the Convention Over Configuration pattern, making the metadata usage concealed in the code element properties, such as the element name[37]. Another practice is the Configuration by Exception, where only the exception of a defined rule must have the metadata configured[38]. The major problem with those approaches is that frameworks or APIs must implement convention reading as part of their metadata reading approach. The following section presents our model to decouple the convention over configuration reading from the code that reads annotations.

3 | A MODEL FOR IMPLEMENTING A DECOUPLED SOLUTION FOR CONVENTION OVER CONFIGURATION

This section presents a model that allows decoupling the definition of code conventions from the code that reads framework annotations. In other words, it should be transparent for a code that checks for the presence of a piece of metadata, if the correspondent target annotation is present, or if the code element matches a given convention. One of the goals of this model is also to enable a declarative definition of code conventions so they can be defined separately. One of the challenges to achieving this goal is that annotations cannot be instantiated directly in Java code, and the model also presents a solution for this issue.

Figure 1 presents the sequence for transparently identifying the convention. When the application client invokes the entry point method to verify the presence of metadata, it first reaches a proxy, called `ConventionsLocator`, that uses the standard reflection API to check if there is a target annotation present. It will check for the convention only if it does not find the framework annotation in any other possible way. When a code element haves the framework annotation, this framework annotation always has priority over the convention. If the target annotation is absent, then the code should verify if a convention annotation is associated with the target annotation and if the code element matches the convention specified by the convention annotation. If the answer is positive, the proxy should return the same response as if the target annotation is there. To locate the convention associated with a framework annotation, the component `ConventionsLocator` checks a repository where these conventions are represented and stored. Our model also proposes to define the conventions in the target annotation definition using convention annotations.

The structure presented in Figure 2 is based on the pattern Metadata Repository[14]. The class `ConventionsContainer` plays the role of the Metadata Container, and the class `ConventionsRepository` stores an instance of it, which provides metadata related to conventions to the `ConventionsLocator`. The `ConventionsContainer` represents all the conventions related to a target annotation. In our model, conventions are present in the target annotation definition. Convention annotations define the conventions marking the target annotation (the omitted one when the element matches the convention). Reader classes will load this metadata and add it to the repository.

One challenge to the transparency of this approach is in the scenarios where it is necessary to return framework annotations. For this transparency, the metadata reading API should return the target annotation even when only the convention is present. The difficulty is that it is impossible

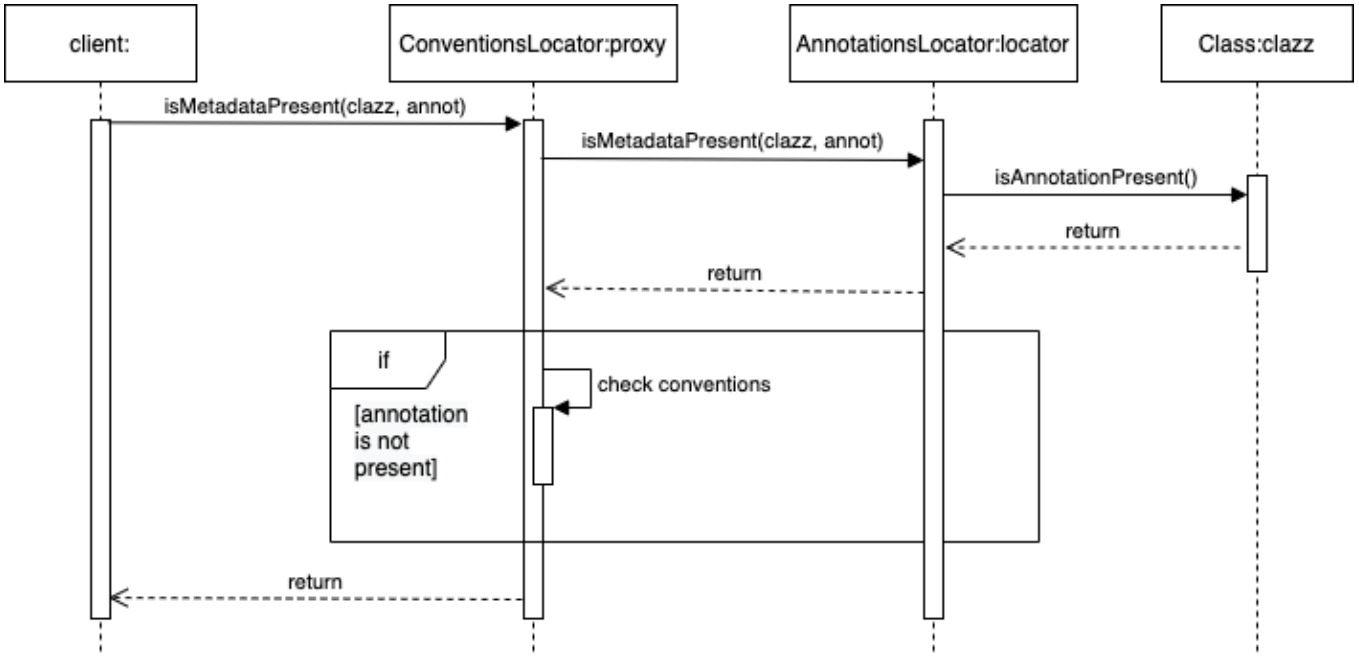

FIGURE 1 Sequence diagram for checking for conventions transparently

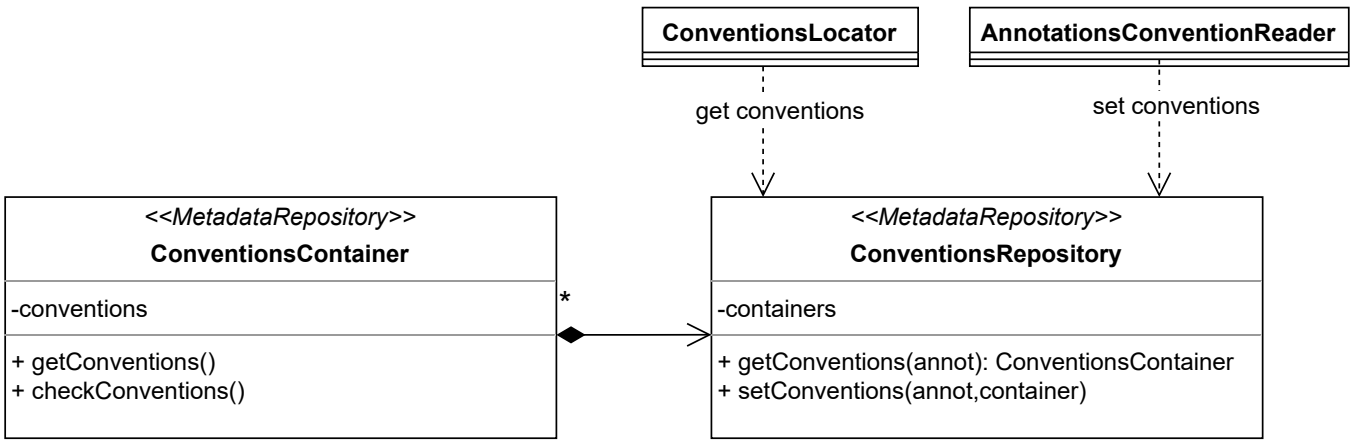

FIGURE 2 Getting conventions from the annotation definition

to instantiate annotations in Java, and the only way to retrieve them is from source code elements that already have them. As a strategy to overcome this limitation, a class with the target annotation is generated in runtime using bytecode manipulation. After that, the loaded class receives the target annotation, and the target annotation is retrieved and returned. Figure 3 presents the steps proposed by this model to achieve it. First, the application client reaches the `ConventionsLocator`, which must check if the element matches the convention. If it fails to check the presence of the target annotation, it must check if the convention matches. If the code element matches the convention, the `ConventionsLocator` must request the `BytecodeManipulator` to generate and load a class with the target annotation. Then the `ConventionsLocator` uses the Reflection API to read the target annotation from the class generated by the `BytecodeManipulator` and return this instance to the application client.

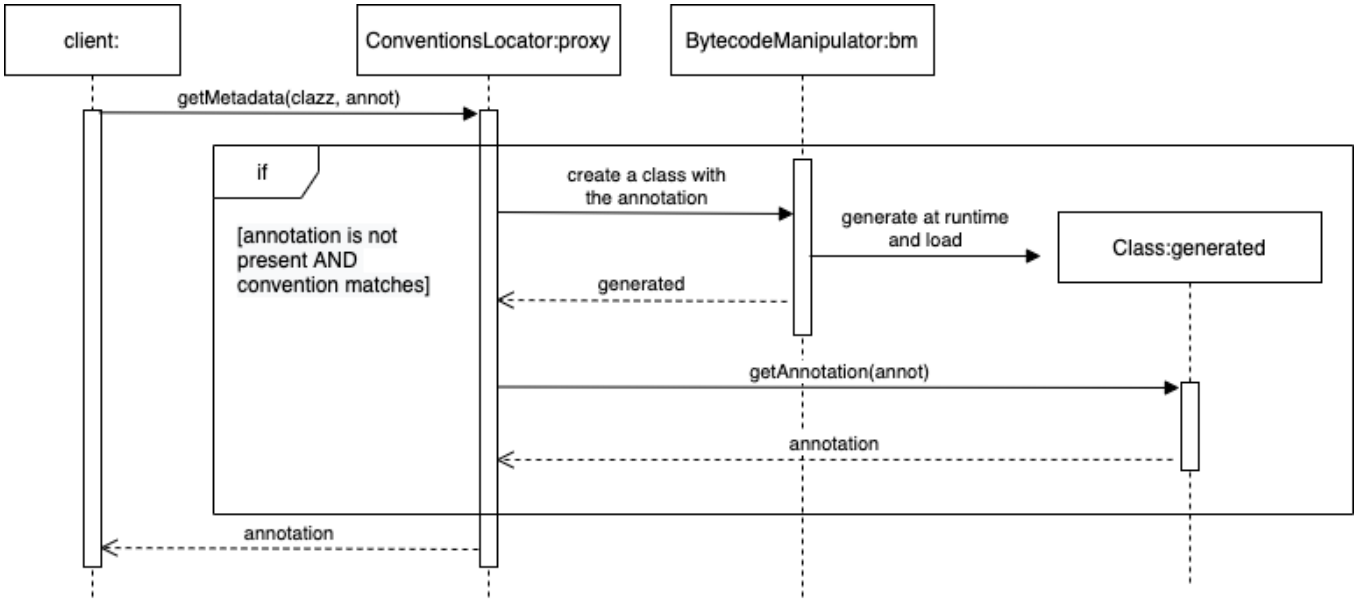

FIGURE 3 Generating a class at runtime to get the annotation targeted by the convention

4 | ESFINGE METADATA SUPPORT FOR CONVENTIONS OVER CONFIGURATION

This section presents how the evolved Esfinge Metadata supports the Conventions over Configuration. The Esfinge Metadata was developed to support the process of reading and validating metadata [17,16]. Unlike Java Reflection API, which only provides methods for retrieving annotations from code elements, Esfinge Metadata API uses Metadata Mapping [64] to map information on class annotations to attributes of an object. This information is helpful to keep this metadata at runtime [18]. Its structure guides the developers to implement metadata-based framework patterns and best practices [14,15]. Based on the evaluation study performed, this API guides the framework design to a more consistent evolution, with less complexity and less dependency between modules [18]. Esfinge Metadata provides several alternatives to retrieve metadata defined in annotations. This section does not describe its complete API but only how the conventions are defined and used. The new functionality implemented is compatible with all other Esfinge Metadata features. For further information on retrieving annotations, refer to the research papers about its novel metadata reading approach [17,18].

4.1 | Defining Conventions for Annotations

In our approach, to define conventions for a target annotation, we add in its definition a convention annotation, in our case, from Esfinge Metadata, that defines the desired convention. To avoid confusion, we use "target annotation" or "framework annotation" to refer to the ones that receive the convention, as well as "convention annotation" to refer to the one from the metadata reading API that defines the convention.

Convention annotations must be configured in each target annotation to use our approach for convention over configuration. Listing 1 presents the target annotation `@IsParameterPresent`. This target annotation haves the `@SuffixConvention` convention annotation. By configuring the `@IsParameterPresent` target annotation with the `@SuffixConvention(value = "Present")` (line 3) convention annotation, every field element whose name ends with the suffix **"Present"** will be implicitly considered as being configured with the `@IsParameterPresent` target annotation.

```
1  @Retention(RetentionPolicy.RUNTIME)
2  @Target(ElementType.FIELD)
3  @SuffixConvention(value = "Present")
4  public @interface IsParameterPresent {
5     String name();
6  }
```

LISTING 1: Example of annotation configured with conventions.

Consider the Listing 2, the fields "valuePresent" (line 2) and "value" (line 4) are both configured with the `@IsParameterPresent` target annotation since "value" is explicitly annotated and "valuePresent" ends with the suffix configured by the convention annotation. Even though the

framework annotation `@IsParameterPresent` is not explicitly configuring the member "valuePresent" (line 2), the Esfinge Metadata can detect the convention and read the annotation as if it were there.

```
1  public class ClassWithConventionExample {
2     private boolean valuePresent;
3     @IsParameterPresent(name="value")
4     private Boolean value;
5     //getters and setters omitted
6  }
```

LISTING 2: Example of a class with convention.

4.2 | Extending the Framework to Define a New Convention

Listing 3 presents an example of how the framework defines conventions. Convention annotations must have their retention policy set to runtime (line 1), and the target element must be an annotation type (line 2). The annotation `@Verifier` configures the respective class that the Esfinge Metadata API should call to verify if the target code element matches the convention (line 3). A `ConventionVerifier` is a concrete class that verifies if the code element matches the convention. Listing 4 presents the interface `ConventionVerifier`, implemented by every class responsible for verifying the convention presence. The method `init` is responsible for processing the parameters or the convention annotation, and the method `isConventionPresent` verifies if the `AnnotatedElement` matches the convention.

Convention annotations can have attributes, as `value` in this case, to parameterize the convention. For instance, the convention annotation `SuffixConvention` receives the `String value()`, specifying that the code element name must end with so that element matches the convention. Consider Listing 5 where we present the implementation of the `SuffixConventionVerifier`.

```
1  @Retention(RetentionPolicy.RUNTIME)
2  @Target(ElementType.ANNOTATION_TYPE)
3  @Verifier(SuffixConventionVerifier.class)
4  public @interface SuffixConvention {
5    String value();
6  }
```

LISTING 3: Convention for an Annotation.

```
1  public interface ConventionVerifier<A extends Annotation> {
2    public void init(A conventionAnnotation);
3    public boolean isConventionPresent(AnnotatedElement element);
4  }
```

LISTING 4: Convention verifier interface.

```
1  public class SuffixConventionVerifier implements ConventionVerifier<SuffixConvention>{
2    private String suffix;
3    @Override
4    public void init(SuffixConvention conventionAnnotation) {
5      suffix = conventionAnnotation.value(); // receives the suffix configured in the convention annotation
6    }
7    @Override
8    public boolean isConventionPresent(AnnotatedElement element) {
9      String name = AnnotatedElementUtils.getName(element); // get the annotated element name
10     return name.endsWith(suffix); // checks if it ends with the suffix configured in the convention
11   }
12 }
```

LISTING 5: Convention verifier for suffix convention.

The method `init` is responsible for reading the value for the suffix convention annotation (line 4). Then, when the convention existence must be verified, the method `isConventionPresent` (line 8) reads the `AnnotatedElement` name and checks if the element name ends with the suffix specified in the convention annotation.

4.3 | Conventions for Annotation Attribute Values

If one framework annotation has attributes, the developer can annotate its attributes to define which values the framework annotation attributes will have if the code element matches the convention. Listing 6 presents the `@Configurable` annotation to receive in the attribute "name()"

the value that comes before the “Config” suffix. The `@FixedStringValue` is an attribute convention annotation that defines the value of the attribute `name()` of the annotation `@Configurable` when the code elements match the convention of target annotation `@Configurable`.

```
1  @Retention(RetentionPolicy.RUNTIME)
2  @Target(ElementType.METHOD)
3  @SuffixConvention(value = "Config")
4  public @interface Configurable{
5     @FixedStringValue(value = "defaultName")
6     String name();
7  }
```

LISTING 6: Setting values for conventions attributes.

For instance, consider that the application asked to retrieve the framework annotation `@Configurable` in a method without the target annotation explicitly configured. Consider that the method matches the convention specified by the convention annotation. The annotation is returned with the value set in the respective attribute. Consider the class *ConfigValues* declared on Listing 7. The method named `userConfig()`, which matches the convention for having the suffix “Config”, will have as the value for the attribute `name` the value “`defaultName`”.

```
1  public class ConfigValues {
2     private String name;
3
4     public String userConfig(){
5        // config user
6     }
7
8     @Configurable(name = "name")
9     public String getUser(){
10       //return user
11    }
12 }
```

LISTING 7: Example of a method with convention for @Configurable annotation.

4.4 | Extension for New Attribute Conventions

The attribute conventions mechanism can be made similarly to extending new convention annotation. It is necessary to create an annotation to configure the attribute convention and a concrete class that implements the `AtributeConventionValueGenerator` interface to provide new attribute conventions. This concrete class should contain the logic that generates the desired value for the annotation parameters.

Listing 8 presents the declaration of the `@FixedStringValue` attribute convention. Like other conventions, attribute conventions must be found in runtime, so their retention policy must is runtime (line 1), and the target element must be a method (line 2). Each attribute convention has its own generator associated (line 3). Then, the value generator class must be implemented. Listing 9 presents the implementation of the `FixedStringGenerator` class. The interface `AtributeConventionVerifier` has one single method (`generateValue`) responsible for generating the value for the attribute convention. In this case, the generator must return the element name for which the convention was verified.

```
1  @Retention(RetentionPolicy.RUNTIME)
2  @Target(ElementType.METHOD)
3  @ValueGenerator(FixedStringGenerator.class)
4  public @interface FixedStringValue {
5     String value();
6  }
```

LISTING 8: ElementName attribute convention.

```
1  public class FixedStringGenerator implements AtributeConventionValueGenerator{
2     @Override
3     public Object generateValue(Class<? extends Annotation> mainAnnotation,
4                                 AnnotatedElement element,
5                                 Method annotationAtribute, Annotation atributeConvention) {
6
7        FixedStringValue ele = (FixedStringValue)atributeConvention;
8        return ele.value();
9     }
10 }
```

LISTING 9: Generator for @ElementName attribute convention.

4.5 | Locating Conventions for Annotations

The Esfinge Metadata API implements the `Metadata Reader Chain` design pattern to verify if the code element matches the convention since, with this pattern, different sources can read the metadata. Since our approach uses convention annotations to define conventions, we implemented, in the Esfinge Metadata API, the class `ConventionsLocator` that extends the `MetadataReader`, which is responsible for verifying if the code element matches the convention. In short, the `ConventionsLocator` evokes the metadata locators chain provided by the Esfinge Metadata API. According to Listing 10, the chain evokes the readers to locate metadata.

The class `LocatorsFactory` is responsible for creating the `MetadataLocator` chain. Each locator is responsible for searching metadata by different means. Before executing its logic, any locator will first evoke the next locator of the chain, meaning that it will only search for the metadata if the next locator does not find the metadata. That means the `ConventionsLocator` verifies if the element matches the convention only if all the other locators of the chain did not find the metadata. When any other locator does not find the annotation, the `ConventionsLocator` uses the convention verifier specified by the convention annotation to verify if the code element matches the convention. The next locator is the `InsideAnnotationsLocator`, which searches for metadata inside the target annotations. Following the chain, the `EnclosingElementLocator` searches for metadata on the element that encloses the configured code element. For instance, if a field has a target annotation, the `EnclosingElementLocator` also looks for metadata in the class declaration. The Esfinge Metadata API provides the `InheritanceLocator` to search for metadata on the interfaces and superclasses of a class. Finally, the API has the `RegularLocator` to check if the code element has the metadata.

```
1  public class LocatorsFactory {
2   public static MetadataLocator createLocatorsChain() throws AnnotationReadingException {
3     MetadataLocator conventionsLocator = new ConventionsLocator();
4     MetadataLocator insideAnnotationsLocator = new InsideAnnotationLocator();
5     MetadataLocator enclosingElementsLocator = new EnclosingElementLocator();
6     MetadataLocator abstractionsLocator = new InheritanceLocator();
7     MetadataLocator regularLocator = new RegularLocator();
8
9     conventionsLocator.setNextLocator(insideAnnotationsLocator);
10    insideAnnotationsLocator.setNextLocator(enclosingElementsLocator);
11    enclosingElementsLocator.setNextLocator(abstractionsLocator);
12    abstractionsLocator.setNextLocator(regularLocator);
13    return conventionsLocator;
14   }
15 }
```

LISTING 10: Chain of locators factory.

When a code element matches the convention, an instance of the respective target annotation must be returned. Figure 4 presents how to return an instance of a target annotation in runtime. The `MetadataReaderClient`, which represents the class that needs to read the metadata, requests the `ConventionsLocator` for an instance of that framework annotation if the code element matches the convention. If the target annotation is absent in the code element, the locator checks if the element matches the convention defined in the framework annotation. If it matches, the locator requests the class `AnnotatedElementUtils` to generate at runtime the bytecode of a class with target annotation, to be used as a source. The `AnnotatedElementUtils` uses the Esfinge ClassMock[65], a framework that generates classes at runtime using bytecode manipulation, to generate a class where an instance of the target annotation can be taken. After getting the generated class from Esfinge ClassMock, the `AnnotatedElementUtils` retrieve the respective annotation instance and returns it to the conventions locator.

5 | ESFINGE COMPARISON CASE STUDY

Esfinge Comparison[14] is an open-source metadata-based framework for comparing two instances of the same class. The framework provides annotations for customizing how the comparison algorithm between the instances. For better understanding, consider the term "comparison annotation" as the annotations used for customizing the comparison algorithm. The framework supports comparing the object properties or between lists of complex objects. The framework is extensible, allowing the developer to create new annotations and classes for reading and processing metadata.

The comparison annotations configure the *getters* methods of the class. The framework searches for all class attributes (public or private) since it uses the class getters to access them. The framework target annotations customize the comparison algorithm. Each framework annotation has the annotation `@DelegateReader`, which specifies the class responsible for executing the comparison algorithm for that target annotation. Some framework annotations that we added conventions are listed below.

1. **@IgnoreInComparison**: this framework annotation specifies the fields that do not participate in the instance comparison.

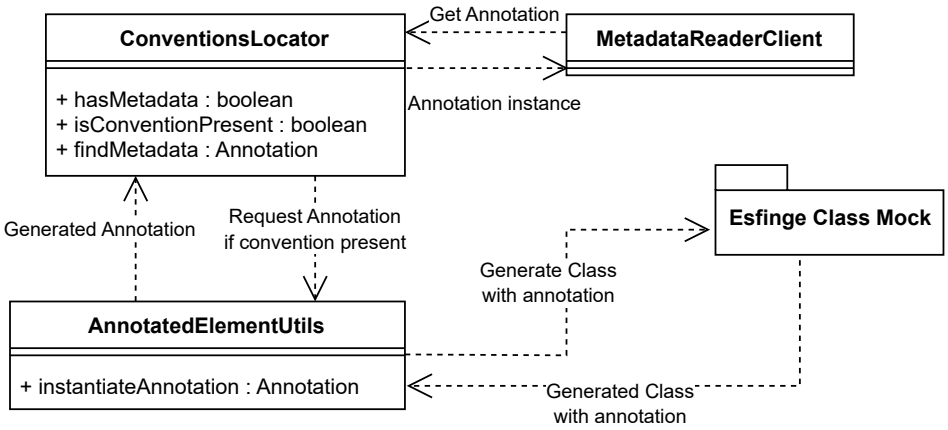

FIGURE 4 Getting generated annotations at runtime.

2. **@DeepComparison**: executes the comparison of the property, comparing its internal properties.

3. **@Tolerance**: configures the allowed numeric tolerance for an attribute. This target annotation configures floating point attributes, which can exist as differences caused by floor or ceiling.

This case study aims to identify the effort of refactoring the comparison framework, to support code conventions using the proposed approach. Since the Esfinge Comparison uses the Esfinge Metadata as its metadata reading API, we decided to improve the framework by adding support for conventions for some of its comparison annotations. To confirm that our approach reached its goal, the only modification needed to add the convention should be configuring the convention annotations in the framework annotations.

Table 1 presents the convention annotations added to the comparison framework target annotations. To the `@IgnoreInComparison` target annotation, we added the `@PrefixConvention(value = "getIgnore")` convention annotation; this way, any *getter* method with a name starting with "getIgnore" will not have its return attribute considered in the comparison. For the `@DeepComparison` framework annotation, we added the `@SuffixConvention(value = "Deep")`; this form, all *getter* methods ending with "Deep", will be considered as having the `@DeepComparison` target annotation. Finally, for the @Tolerance target annotation, the convention annotation `@MethodReturnTypeConvention(double.class)` was applied; this way, all getters that return a double value will be considered to have the `@Tolerance` annotation, with a default tolerance value of 0.01. Note that for the `@Tolerance` framework annotation, we also added the `@FixedDoubleValue` attribute convention to the parameter `value()`.

Target Annotation	Convention Annotation	Convention Description
@IgnoreInComparison()	@PrefixConvention(value = "getIgnore")	Any method with name prefix "getIgnore", will be considered as having the @IgnoreInComparison framework annotation.
@DeepComparison	@SuffixConvention(value = "Deep")	Any method ending with "Deep" will be considered as having the target annotation @DeepComparison.
@Tolerance(double value)	@MethodReturnTypeConvention(returnType = double.class) @FixedDoubleValue(0.01)	Any getter for fields of type *double* will be considered as having the @Tolerance target annotation, with a default value of 0.01.

TABLE 1 Conventions used for comparison annotations.

Automated integration tests were added to the framework testing suite to evaluate if the new conventions were being considered in the framework functionality and aligned with the case study's goal to implement the conventions. The only change was the addition of convention annotations to the existing framework annotations.

For instance, consider the Listing 11 presenting the code for the `@Tolerance` framework annotation before the refactoring for adding conventions support, and Listing 12 the refactored code. To support conventions, the `@MethodReturnTypeConvention` configures the convention,

and `@FixedDoubleValue` the tolerance value in case the convention verifies. Similarly, the other convention's annotations were added as defined in Table 1.

```
1  @Target({ElementType.METHOD})
2  @Retention(RetentionPolicy.RUNTIME)
3  @DelegateReader(ToleranceReader.class)
4  public @interface Tolerance {
5    double value();
6  }
```

LISTING 11: Tolerance annotation before refactoring.

```
1  @Target({ElementType.METHOD})
2  @Retention(RetentionPolicy.RUNTIME)
3  @DelegateReader(ToleranceReader.class)
4  @MethodReturnTypeConvention(returnType = double.class)
5  public @interface Tolerance {
6    @FixedDoubleValue(0.01)
7    double value();
8  }
```

LISTING 12: Tolerance annotation after refactoring.

After these changes, all the automated integration tests, including the new ones to verify the new conventions, were executed successfully. Since we implemented the conventions by adding convention annotations to the framework annotations, we confirmed that our solution fulfilled its goal. The full source code of this case study is available at the Esfinge Comparison framework Github repository[1], at the branch "conventions", where all changes and tests can be accessed.

6 | EVALUATION STUDY DESIGN

This section presents the research design to evaluate the model proposed in Section 3. This experiment aims to address the following research question:

RQ: How does the proposed model for Conventions over Configuration implementation impact the development of a metadata-based framework? To answer the question, we conducted an experiment in which the participants implemented conventions in an existing framework with Reflection API and Esfinge Metadata API. Different development aspects were analyzed separately. We evaluated the success rate, considering a successful implementation with all unity tests passing and implemented according to all specifications provided. Another aspect considered was the total development time using each approach. Finally, we also assessed the participants' experience implementing conventions with each approach, aiming to find the strengths and weaknesses of each API.

First, we implemented the model presented in Section 3 on the Esfinge Metadata API. Then we conducted an experiment with undergraduate students to evaluate our research question. Carrying out experiments using students is a feasible approach and a common option to advance the software engineering field via empirical studies, as shown by Falessi[66] and HÖst[67]. We designed programming tasks for the participants to implement conventions in an existing subject framework. The participants had to implement the tasks in an implementation created only with the Reflection API approach and another that used the Esfinge Metadata API. We designed four tasks, meaning we had four different conventions but with similar behavior. We divided the participants into groups using the crossover design[68]. Each participant had to implement two tasks with Reflection API and two with the Esfinge Metadata API. After completing the tasks, the participants answered an evaluation questionnaire. We asked the participants about the time necessary for implementing the tasks, and we asked questions about the difficulties that the participant had in implementing the tasks for both approaches. We then analyzed their solution, observing the changes made to the code.

In the first analysis, we evaluated if the solution was correctly implemented, if all unity tests passed and if the task was implemented with the correct approach. Then we observed the lines changed by each participant, accounting for the amount and elements added, like loops and conditional. Then we evaluated the questionnaire responses, accounting for the implementation time, advantages and disadvantages of each approach, and the overall experience. The following subsections present the framework used in the experiment, referred to as the subject framework used in this study, and describe the tasks the participants implemented.

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

6.1 | Experimental design

In the first step of our work, one of the researchers implemented the subject framework for the evaluation. The subject framework is a metadata-based framework for mapping the command line arguments to a class instance, using as reference the same specification used in another study [18]. Secondly, we prepared the documentation for the Esfinge Metadata API and the subject framework. The participants could access this documentation while implementing the tasks to support the development. Then, we prepared the description for the participants' tasks. Each participant had access to two GitHub repositories, where we provided the subject framework implementation and where their solution must be pushed. The repositories are private, and the participants had access exclusively to their two repositories. We conducted two pilot experiments with two research group members to evaluate our documentation and test if the repositories meet the correct configuration. The pilots were not considered in the final evaluation but served to improve documentation and ensure that the participants did not face any repository misconfiguration.

The experiment was executed in a class of 28 students, which tried to execute the tasks and answered a questionnaire. We evaluated the source code that the participants pushed to the GitHub repositories, considering them correct if all unit tests were passing and the participants implemented the tasks with the correct approach. Solutions with compilation errors failed tests or were implemented without following the specification served for the correctness analysis. We then evaluated the changes made by the participants, considering only the students that managed to implement the tasks correctly. The last step was to evaluate the questionnaire responses with participants' experiences with their difficulties, advantages, and disadvantages while implementing the conventions with both approaches.

6.2 | Evaluation Framework Description

We provided the participants with two versions of the subject framework to evaluate the model proposed in Section 3, one using Esfinge Metadata API and another one using only Reflection API. The same versions were developed for a previous study that compared the usage of these two APIs [18], and both implementations have the same test suite. This test suite comprises 67 unit tests to ensure the subject framework can correctly process the target annotations. This subject framework maps an array of parameters from the command line to a class using framework annotations. This parameter array is received in the main() method to a JavaBean class. This mapping is done through metadata configurations using the framework annotations. This subject framework implements the Metadata Container pattern [14], with its basic structure presented in Figure 5. The Metadata Container pattern introduces a class called `MetadataContainer`, whose instance represents the metadata read at runtime. This class is responsible for storing the metadata read from the framework annotations. The `FrameworkController` class requests the repository for metadata of a specified class. If the metadata are not yet retrieved, the repository must evoke the `MetadataReader` to find the metadata. The framework annotations used for this experiment are:

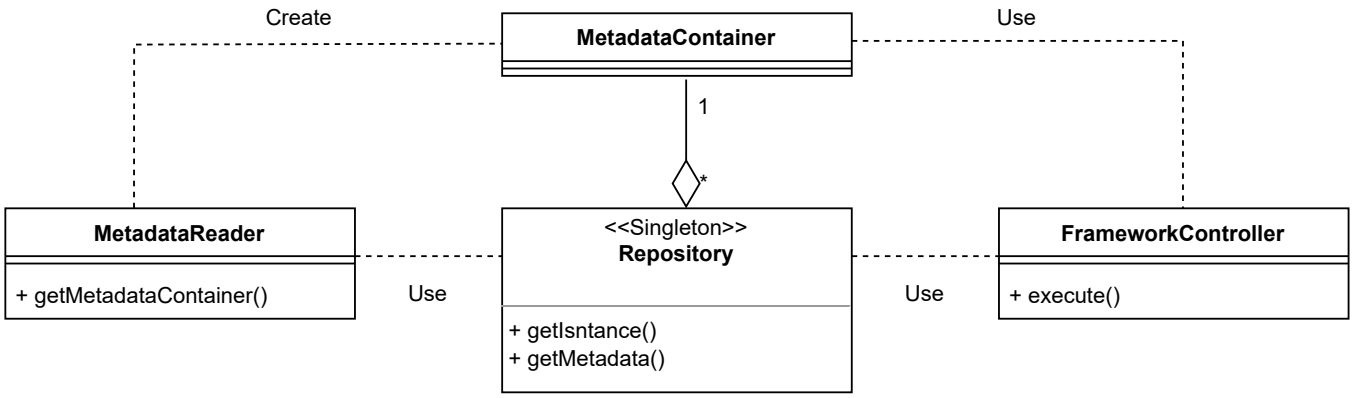

FIGURE 5 Basic structure of the Metadata Container pattern.

1. **@IsParameterPresent(String paramName)**: This framework annotation sets a class boolean attribute as true or false if the parameter *paramName* is present on the command line.

2. **@ParameterPrecision(int decimalPlaces)**: This framework annotation configures the decimal tolerance for a field of float or double type. For instance, if the value of decimalPlaces is 2, the parameter value should not have more than two decimal places.

3. **@TextValue(String value)**: The String attribute annotated with this framework annotation will receive the string *value* as its value.

4. **@CompositeParameter**: This framework annotation maps values for a field of a non-primitive type, that is, another class.

6.2.1 | Tasks

Participants had to implement two different but similar conventions for each subject framework version. With two different approaches to the subject framework, we could compare the resulting code and collect the participant's feedback about their experience developing convention over configuration for each scenario. We divided the tasks into two groups **A** and **B** according to Table 2.

TABLE 2 Description of tasks

Group	Task	Convention to be implemented
A	1	Any boolean field ending in "Present" should be considered as having the framework annotation @IsParameterPresent with the parameter value equal to what comes before this suffix.
A	2	Any field with the type Float (not the primitive float) should be considered as having the framework annotation @ParameterPrecision with the parameter decimalPlaces equals to 1.
B	1	Any field ending in "Text" should be considered as having the framework annotation @TextValue with the parameter value equal to what comes before this suffix.
B	2	Any field of non-primitive type with the @OneToOne should be considered as having the framework annotation @CompositeParameter.

We divided the participants into four groups using the crossover design for two treatments[68]. This design suggests that participants must execute different tasks in different orders to eliminate bias. The groups are specified in Table 3. We provided an automated test suite to evaluate whether conventions were correctly implemented. Thus, the code convention is correctly implemented if all unit tests pass and are implemented according to the specification. After finishing the implementation of the conventions, the participant must commit his/her code to the repository.

6.3 | Code repositories

We created two GitHub repositories for each participant to conduct this experiment, one containing the implementation using the Esfinge Metadata API and another using Reflection API. These repositories belong to a GitHub organization[2]. Each participant had access only to his private repositories. For instance, the repositories' names are like *participantOne-metadata* (Esfinge Metadata repository) and *participantOne-reflection* (Reflection repository), where *participantOne* is the participant GitHub username. The repositories from other participants are invisible to *participantOne*.

6.4 | Documentation

Before implementing the code conventions, the participants had a set of instructions on how to execute each task. The experiment instructions provided information about the tasks execution order, how to access the documentation, how to access the code repository, and how to access the description of the tasks. Also, they had access to documentation about the Esfinge Metadata API and the subject framework where the conventions needed to be implemented. While executing the tasks, the participants were allowed to take breaks. We considered reading the documentation and implementing the conventions as the total time necessary for concluding the tasks.

The participants are undergraduate students. We had 28 students that manifested interest in participating in the experiment. We divided the students into four groups of 7 students. The groups are described in Table 3. We divided the participants into these groups so we could have different tasks implemented in both subject framework implementations. Also, we could avoid all participants starting with the same subject framework implementation using the crossover design[68]. After implementing its tasks, the participants answered an online questionnaire.

[2]https://github.com/Metadata-Experiment

TABLE 3 Participants groups distribution

Group	Description
1	The students of this group implemented tasks 1 of group A and 2 of group B using Reflection API and tasks 2 of group A and 1 of group B using Esfinge Metadata API. Also, they did the tasks with Esfinge Metadata API first.
2	The students of this group implemented tasks 1 of group A and 2 of group B using Reflection API and tasks 2 of group A and 1 of group B using Esfinge Metadata API. Also, they did the tasks with Reflection API first.
3	The students of this group implemented tasks 2 of group A and 1 of group B using Reflection API and tasks 1 of group A and 2 of group B using Esfinge Metadata API. Also, they did the tasks with Esfinge Metadata API first.
4	The students of this group implemented tasks 2 of group A and 1 of group B using reflection and tasks 1 of group A and 2 of group B using Esfinge Metadata API. Also, they did the tasks with Reflection API first.

6.5 | Questionnaire design

The questionnaire starts asking for the participant's consent. Then it has a section of personal questions, like the participant's familiarity with annotations and its primary programming language. After that, the participants answered 18 questions, divided into two groups of 9. The first group is about the participant experiences implementing the tasks using the Esfinge Metadata API. Then we asked the same questions about its experience implementing the tasks with Java Reflection API. The questions of this section are presented in Table 4.

TABLE 4 Questionnaire questions.

ID	Question
	Overall framework experience
Q1	How many tasks have you accomplished?
Q2	How much time you spent implementing the tasks?
Q3	You felt the necessity to interrupt the task for any reason?
Q4	If you answered "Yes" to the previous question, answer how many interruptions did you make
Q5	Do you think interruptions affected your total time for completing the tasks?
Q6	Select the difficulties that you had while executing these tasks
Q7	Did you needed to search for extra information besides the provided documentation?
Q8	If you answered "yes" to the previous question, tell which material and why you need this extra material
Q9	Describe your overall experience while implementing the tasks using the Esfinge Metadata API
	Approaches evaluation
Q1	Which you consider the advantages of using Esfinge Metadata API for creating Code Conventions compared to the other approach
Q2	Which you consider the disadvantages of Esfinge Metadata API for creating Code Conventions compared to the other approach
Q3	Which you consider the advantages of using standard Java reflection API for creating Code Conventions compared to the other approach
Q4	Which you consider the disadvantages of standard Java reflection API for creating Code Conventions compared to the other approach
Q5	If you were responsible for a metadata framework and needed to implement code conventions for its annotations, which approach would you use.
Q6	Justify your previous answer

Two sections divide the questionnaire, a first set of questions about the overall experience using each framework and one section where the participant was asked about the advantages and disadvantages of developing the code conventions with each framework. Question Q6 has eight default options presented in Table 5 where the participant can select all the difficulties found and one open field for the user to inform any other difficulty not present in the default list. The options OPT2 and OPT5 have slightly modified text since those options are related to the framework usage. Questions Q4 and Q8 of the first section are optional open questions and are only answered when the participant answers "yes" to questions Q3 and Q7, respectively. The questions Q1 and Q3 of the second section have a default set of possible answers, presented in Table 5, and one open field where the participant can state any other advantages of each framework. The participants had to answer the disadvantages of each framework on questions Q2 and Q4 of the second part of the questionnaire and have their options presented in Table 5. Finally, Q5 asks which framework the participant would consider using to develop a metadata-based framework, and Q6 is one obligatory open question where the participant must justify the previous answer.

TABLE 5 Options for approaches evaluation questions

ID	Question
	Overall framework experience - options for Q6
OPT1	Understand the unit tests
OPT2M/OPT2R	Understand Esfinge Metadata documentation / Understand the usage of Reflection in the code
OPT3	Understand the task that needed to be done
OPT4	Understand what the target framework does
OPT5M/OPT5R	Use the Code Conventions feature from Esfinge Metadata /Use Reflection features in the implementation of the Code Conventions
OPT6	Read and navigate in the target framework code
OPT7	Implement the conventions feature
OPT8	Find where the conventions should be implemented
	Approaches evaluation - options for Q1 and Q3
OPT1	Development speed
OPT2	Easy to be implemented
OPT3	Resulting code is easy to be changed
OPT4	Resulting code is more readable
OPT5	Easy to learn
	Approaches evaluation - options for Q2 and Q4
OPT1	Development is hard to be performed
OPT2	Development takes more time
OPT3	Requires a large learning curve
OPT4	Resulting code is more complex
OPT5	Resulting code is hard to maintain
OPT6	Resulting code has duplicated code

7 | RESULTS

This section presents the data collected from the questionnaire answered by the participants. We start describing the experiment execution and evaluating the correctness of the code implemented by the participants. Then we present the implementation time necessary for both approaches and some code snippets of solutions presented by the participants. Then we present the advantages, disadvantages, and difficulties. We use the results presented in this section to address our research question.

7.1 | Experiment Execution

We executed the experiment following the steps of Figure 6. Of the 28 participants that volunteered, one did not accept the repository invitation. Of the 27 participants that accepted the invitation, 23 participated in the experiment, implementing their solutions for the tasks. Two participants modified the unit tests, so we removed them from the results since the instructions clarified that the participants must only modify the production code, not the unity tests. Analyzing the source code of the remaining 21 participants, we found that one participant did not correctly implement the tasks with Esfinge Metadata having unity tests failing. We conducted the same analysis for the Reflection tasks source code, and we noticed that five participants could not finish or correctly implement the solutions. Lastly, three participants could not correctly implement some or all tasks for both frameworks. After these steps, 12 participants managed to implement all tasks correctly.

7.2 | Correctness Analysis

According to Figure 6, considering only the participants that participated in the experiment, we had 12 participants that correctly implemented the tasks in both frameworks. We had five participants that implemented all tasks using the Esfinge Metadata API correctly but failed to implement at least one task using the Java Reflection API. Only one participant managed to complete the tasks using the Java Reflection API and failed to implement the Esfinge tasks correctly. One participant failed to implement tasks with both approaches. Two participants modified the unit tests, and two implemented the tasks we asked to be with the Esfinge Metadata API with Java Reflection API.

We can see that the number of participants that failed to accomplish any reflection task (five) is higher than the number of participants that failed to implement the Esfinge Metadata tasks (one), which means that participants had more difficulty implementing the tasks with reflection in this

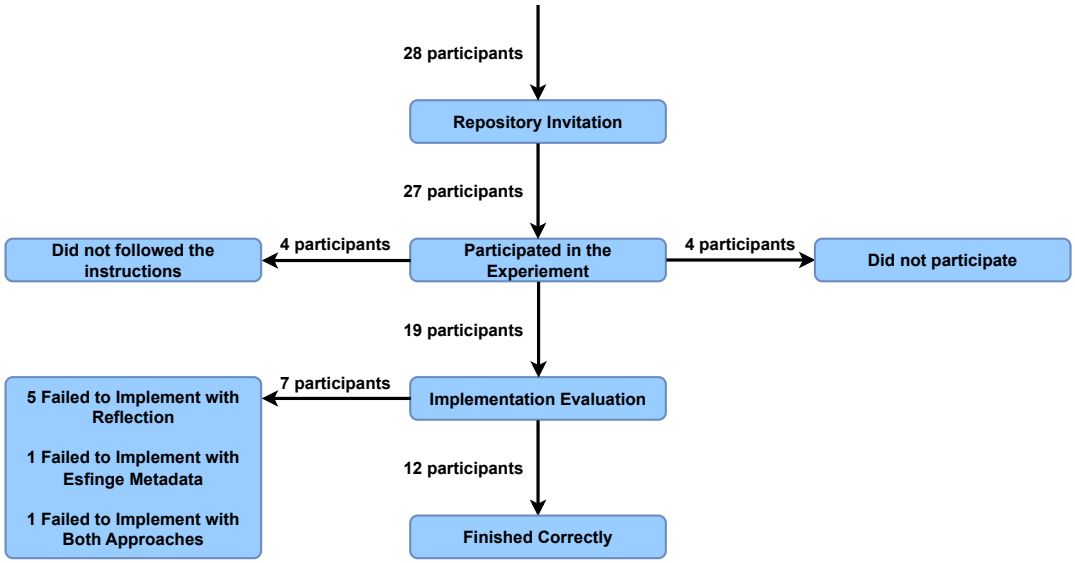

FIGURE 6 Total of participants for each experiment step

experiment. We focus on their appointed experience to better understand why the participants had more difficulty implementing the tasks with Reflection API. One participant stated that although understanding the task, it was challenging to navigate through the code. Also, the participant stated that it was difficult to understand the test failures caused by the changes made to the code. Further analysis of the participant's code shows that he tried to implement the solution on the metadata reading logic of the subject framework. Their changes made the framework wrongly process some field types, leading to unexpected behavior.

Another participant that failed to implement the tasks with reflection stated that the subject framework documentation was unclear, making it hard to understand the subject framework and find where to do the modifications. The participant also felt that he could accomplish the tasks with better documentation about the subject framework. The remaining three participants from this group gave feedback on accomplishing the tasks. However, their code still had unity tests failure, or they implemented only one of the two tasks since they spent 1 hour implementing the task. By analyzing their code, we found that their logic for implementing the conventions was correct for the convention tests but was producing failures in tests of the framework functionality, such as a numeric field receiving a String value, causing an exception error. Although they correctly implemented the conventions, their solution still presented errors when executing the unity tests.

The participant who failed to implement one of the tasks with Esfinge Metadata stated that it was difficult and stressful initially, but this was mitigated by reading the documentation. Since the participant used much time reading the documentation, he could not accomplish the tasks in time. By analyzing the code, the participant missed one of the annotations necessary for completing the task. The participant states that the error message generated by the API was "vague" and did not help to solve the errors. This feedback can be helpful for the researchers since we can improve the error message generated by implementation errors for code conventions. The participant faced this error since it did not add attribute annotations necessary for implementing the task. This participant also failed to implement one task using the reflection API. The participant states that navigating in the boilerplate code was frustrating, and it found hard to understand the core of the framework, making it challenging to implement the tasks. Lastly, we had participants who did not follow the instructions for the experiment. The participants modified the unit tests, which were specified not to be done in the instructions. We consider the 12 participants who successfully implemented all tasks in our analysis.

7.3 | Time spent

We asked the participants to measure the total time spent implementing their tasks, that is, the time needed to implement two code conventions using each approach. We also asked the participants to consider the time spent implementing the solution, reading the provided documentation, and searching for any extra material. Figure 7 presents the data. For the Esfinge Metadata API, the minimum implementation time was two minutes, and for the Reflection API, the minimum was 15 minutes. We notice that 25% of the values for the Esfinge Metadata API are lower than 23.75 minutes against 21.75 for the Reflection API. The median for the Reflection API was 34.5 minutes, and the Esfinge Metadata API had a median of 28 minutes. 75% of the Esfinge Metadata API sample had times lower than 36.25 minutes and 51 minutes for the Reflection API. The highest values are the same for both approaches since one participant informed us that about one hour was needed to complete each framework's tasks.

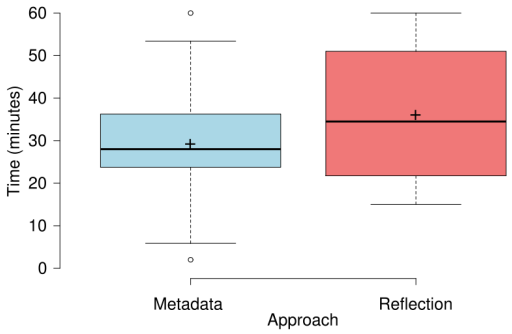

FIGURE 7 Boxplot for implementation time for both approaches

7.4 | Modified lines of code

Having access to the code repositories, we inspected the source code produced by the participants. Table 6 presents the collected results. Each column represents the total lines of code changed to implement the conventions with each approach. Since the Esfinge Metadata API relies only on adding annotations, the maximum value was four, and the minimum was three. That happened because one task needs only one annotation for its implementation. For the Reflection API, the maximum value was 14, and the minimum was three. However, as the Esfinge Metadata API lines of code are composed only of one annotation, these lines contain more code elements for the Reflection API. Observing Table 7, we can see that the Reflection lines of code have more elements than the Esfinge Metadata API. For instance, participant P5 that implemented the solution with three lines of code made 12 method calls. We present some code snippets with the solutions for each convention implemented by each participant. Participants P1 and P3 implemented the same tasks using the Reflection API. Listing 13 presents the convention's implementation for the `@IsParameterPresent` framework annotation, and Listing 14 the solution of participant P3. Lines commented with "existing code" are lines of code present before the participant refactoring, and lines commented with "added code" are lines of code added by the participant to implement the convention. We can see that their implementation are similar. This code was implemented mixed with the metadata reading logic. Both participants relied on Reflection API to implement the convention and used, in this case, `String` class methods. In particular, P3 added one extra field to verify if the suffix was present. On top of that, P3 implemented the *getter* method for this field. We had different code implementations for different conventions using the Reflection API. However, the implementations with the Esfinge Metadata API have a standard implementation.

```
1  this.hasIsParameterPresent = f.isAnnotationPresent(IsParameterPresent.class) ||
2                       f.getName().endsWith("Present"); // modified code
3  if (this.hasIsParameterPresent) { // existing code
4           IsParameterPresent pv = f.getAnnotation(IsParameterPresent.class); // inserted code
5           boolean valid = IsParameterPresentValidator.validate(this.parameterType); // existing code
6           if (valid) // existing code
7              if (pv == null) { //added code
8                 this.paramIsParameterPresent = f.getName().substring(0,
9                                                 f.getName().length() - 7).toUpperCase(); //added code
10             } else { //added code
11                this.paramIsParameterPresent = pv.name(); // modified code
12             }
13 }
```

LISTING 13: Code solution of participant P1 for convention to @IsParameterPresent.

```
1  this.hasPresentSuffix = f.getName().toLowerCase().endsWith("present"); // added code
2  if(this.hasPresentSuffix) {// added code
3           boolean valid = IsParameterPresentValidator.validate(this.parameterType);// added code
4           if(valid)// added code
5              this.paramIsParameterPresent = f.getName().
6                                          toLowerCase().replace("present", "").toUpperCase();// added code
7  }
8  //getter for the created field
9  public boolean hasIsPresentSuffixPresent() { // added code
10       return hasPresentSuffix; //added code
11 }
```

LISTING 14: Code solution of participant P3 for convention to @IsParameterPresent.

We present on listing 15 the implementation of the same convention but using the Esfinge Metadata. Participants P2, P4, P6, P11, and P12 made this implementation. We highlight that the participants who implemented the framework annotation task `@IsParameterPresent` using the Reflection API did not implement the same convention using the Esfinge Metadata API. Thus we present the solution of different participants. Since our approach relies on inserting convention annotations on the framework annotations, every participant that correctly implemented the convention produced the same resulting code. Participants P2, P4, P6, P11, and P12 changed three lines of code. That is, they added three annotations to implement two conventions. That happened because the implementation of the convention for `@CompositeParameter` framework annotation needs only one annotation, as presented in Listing 16. The participants P1, P3, P5, P7, P8, P9, and P10 implemented this convention using the Reflection API. Their solution was overall very similar. Listing 17 presents the implementation of the convention for `@CompositeParameter` made by participant P5. Although the listing presents it in 4 lines, the participant implemented one "if" statement, which checks if the parameter is annotated with the `@OneToOne` annotation.

```
1  @Retention(RetentionPolicy.RUNTIME)
2  @Target(ElementType.FIELD)
3  @SuffixConvention(value = "Present") //added code
4  public @interface IsParameterPresent {
5     @ElementNameBeforeSuffix(suffix = "Present")//added code
6     String name();
7  }
```

LISTING 15: Code solution for convention to @IsParameterPresent framework annotation using Esfinge Metadata API.

```
1  @Target(ElementType.FIELD)
2  @Retention(RetentionPolicy.RUNTIME)
3  @HaveAnnotationOnElementConvention(annotationClass = OneToOne.class) // added code
4  public @interface CompositeParameter {
5  }
```

LISTING 16: Code solution for convention to @CompositeParameter framework annotation using Esfinge Metadata API.

```
1  if(p.hasCompositeParameter() || p.getField().isAnnotationPresent(OneToOne.class) &&
2                                  !p.getField().getType().isPrimitive()){
3     this.hasCompositeParameter = true;
```

LISTING 17: Code solution for convention to @CompositeParameter framework annotation using Esfinge Metadata API.

TABLE 6 Lines of code modified by the participants

Participant	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
Changed lines of code with Reflection API	7	6	7	4	3	7	14	7	3	7	13	14
Changed line of codes with Esfinge Metadata API.	4	3	4	3	4	3	4	4	4	4	3	3

TABLE 7 Code elements present on the lines of Reflection implementations

Participant	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
method call(s)	9	6	8	17	12	7	8	10	3	7	1	4
new code elements	1	0	1	0	0	0	6	0	0	0	0	0
conditional or loops	3	6	3	2	1	3	3	4	1	2	4	5
exception	0	0	1	0	0	0	1	0	0	0	0	1
attributions	2	2	3	2	1	2	3	2	2	3	6	7

7.5 | Difficulties and disadvantages

We collected from the questionnaire the reported difficulties and disadvantages that each participant felt about implementing tasks with each approach. We believe the appointed difficulties can help us understand why the participants failed to implement the tasks using each approach successfully.

The participants provided feedback about their difficulties in implementing the tasks using the Esfinge Metadata API approach and the Reflection API approach. Figure 8 presents the appointed difficulties by the participants. Observing the participants' experience, we observed eight participants appointed the required learning curve for the Esfinge Metadata API. We had three participants who stated that understanding the task was challenging. Three participants appointed finding where to implement the conventions as tricky. Implementing the conventions feature had three answers. Three participants stated that navigating and understanding the target framework was a difficulty. Lastly, no participant appointed the unit tests as a difficulty. According to Figure 9, we had 10 participants appointing the learning curve as a disadvantage. Only one participant stated that the produced code is hard to maintain and more complex. Regarding the development, three participants stated that it is hard to perform and two that it takes more time to implement, compared with the Reflection API.

The most appointed difficulties concerning the Java Reflection API are related to the source code. A total of six participants stated that reading and navigating the source code was hard, and four stated that finding and implementing the code conventions feature was difficult. Also, four participants stated that using the reflection features was challenging. However, as presented in Figure 9, seven participants stated that the resulting code is harder to maintain. The produced code is more complex and has duplication appointed by eight participants. Only one participant stated that a significant learning curve is required. The development took more time and was appointed by five participants. Lastly, three participants stated that development is complex.

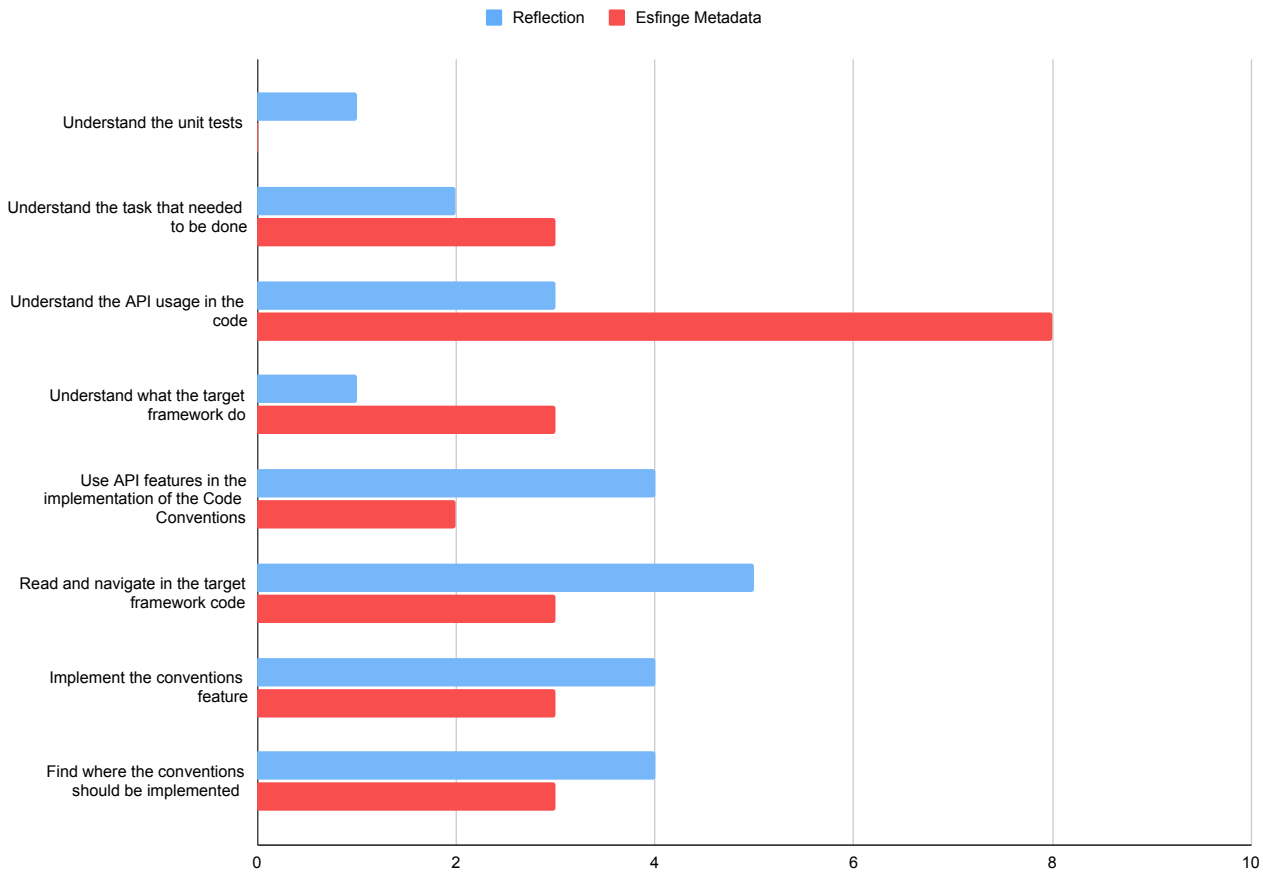

FIGURE 8 Difficulties for both approaches

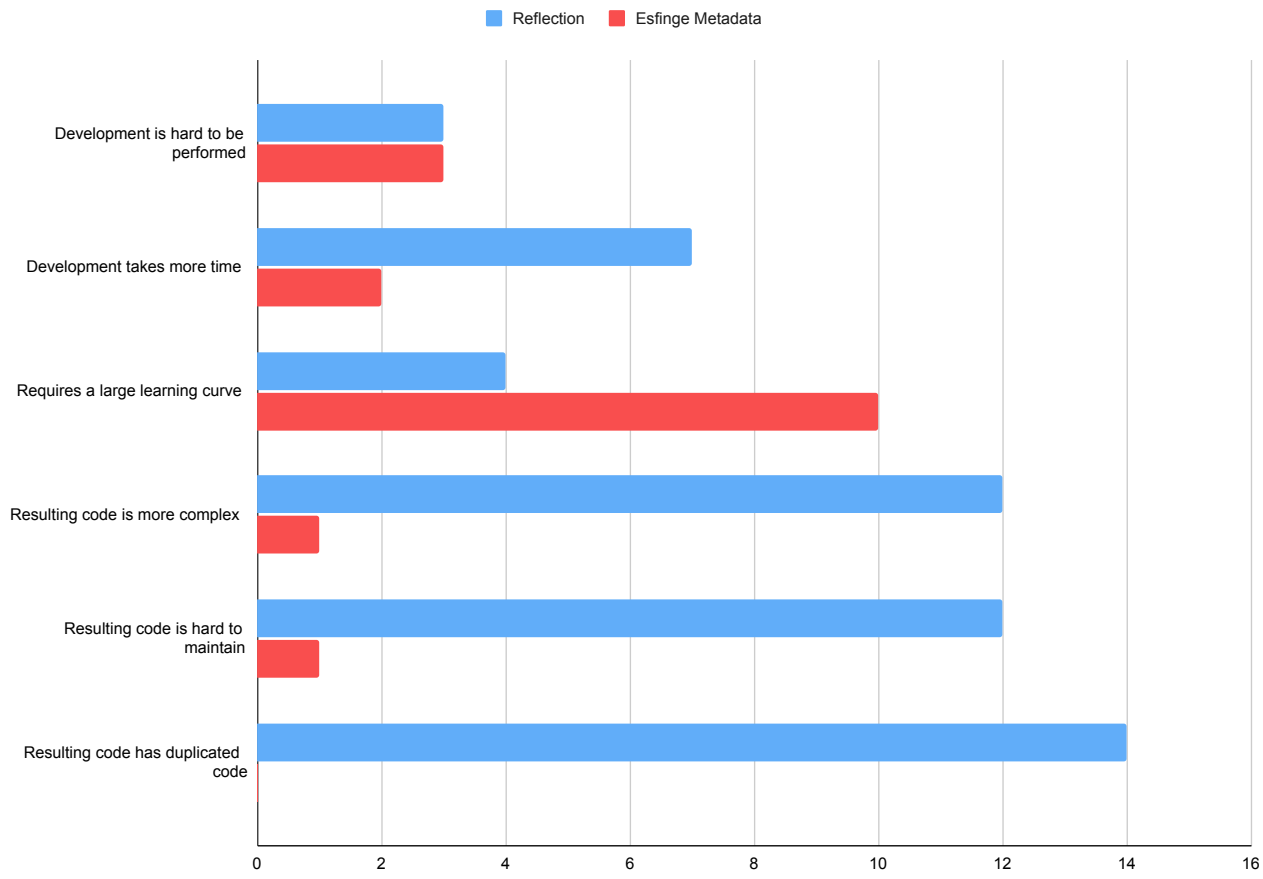

FIGURE 9 Disadvantages for both approaches

7.6 | Advantages with Esfinge Metadata API vs. Reflection API

Figure 10 present the advantages of both approaches. Six participants stated that the implementation with Reflection is easy, and seven for the Esfinge Metadata API. Only two stated that the Esfinge approach is easy to learn, and five stated that the Reflection API is easy to learn. The difference between both approaches was related to the resulting code and development speed.

Most participants (ten) stated that the Esfinge Metadata API has a faster development speed than the Reflection API, whereas only three participants considered the development speed an advantage. A total of nine participants also stated that the Esfinge Metadata API produces a code that is easy to be changed, and two for the Reflection API. The participants stated that the readability of the Esfinge Metadata API code is also better than the Reflection API. In contrast, nine participants said the code with Esfinge Metadata API is more readable than with Reflection (one).

7.7 | Framework Preference

The last two questionnaire questions were about the participant's preferred approach to development. A total of 11 participants preferred the Esfinge Metadata API, and one preferred the Reflection API. Most participants appointed the learning curve necessary for using the Esfinge Metadata API as a difficulty. According to the participants, code readability, less complexity, and easier maintainability were the advantages of the Esfinge Metadata API. Reflection API was reported as easy to learn, but the resulting code was more complex and harder to maintain. The participants stated that despite the large required learning curve, they preferred the Esfinge Metadata API, stating that at the beginning of the development, it takes more time to learn and use the code conventions mechanism. They would consider using the Esfinge approach. Participants P2 and P9 stated that the Esfinge Metadata API performed better in code readability even with a significant learning curve. Participant P9 also stated that even being challenging at the early development stages, the framework outperforms Reflection regarding development time. Participants P3,

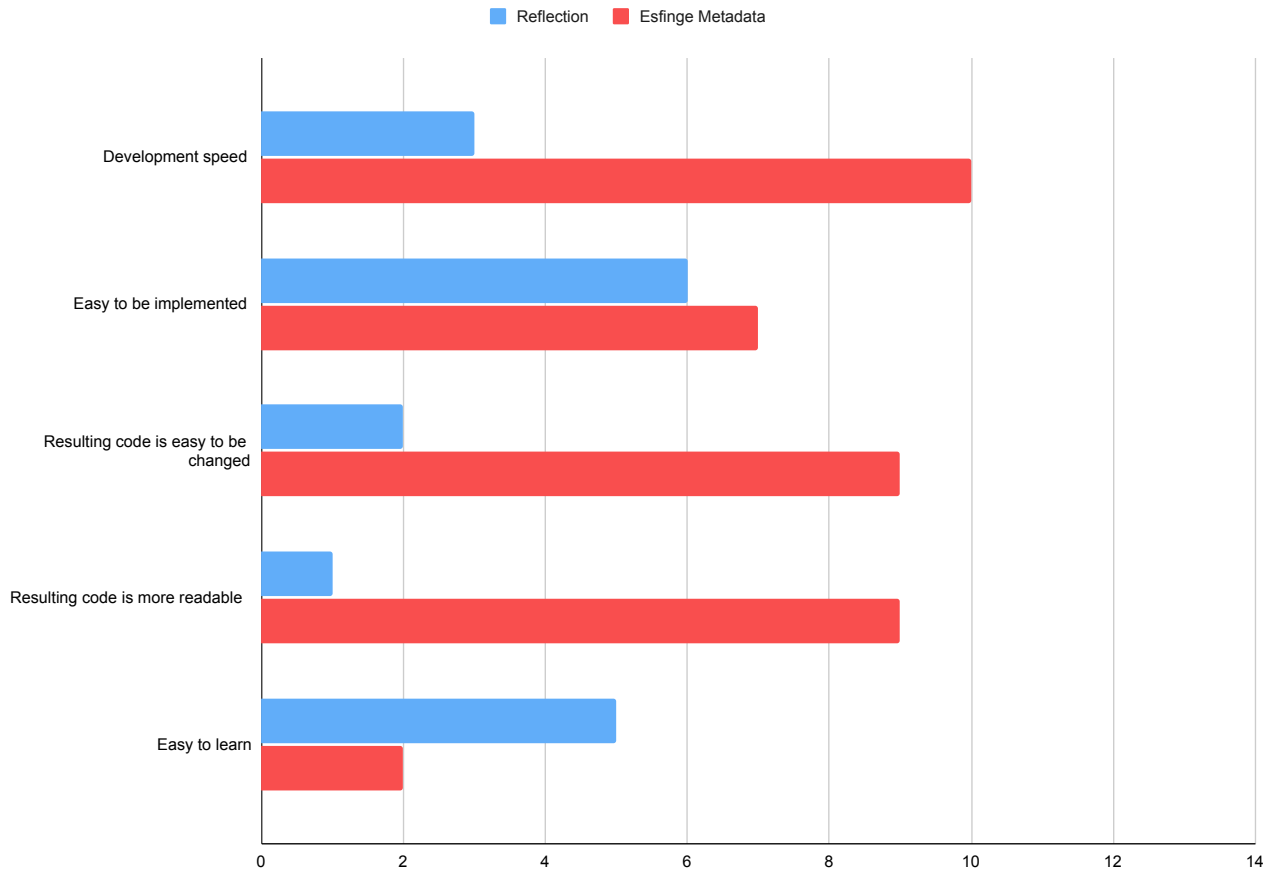

FIGURE 10 Advantages for both approaches

P6, P7, P9, and P12 stated that even with learning curve difficulty, the produced code is more readable, less duplicated, and easier to maintain when compared with the code produced by using the Reflection API. Participant P1 was the only participant who preferred the Reflection API, but the participant stated that he would consider using the Esfinge Metadata API over the Reflection API for larger projects.

8 | DISCUSSION

In this section, we discuss the results presented in Section 7 to answer our research question. To address this question, we evaluate the motives of participants' failure to implement the tasks. Then we analyzed the answers from the questionnaire about their experience while implementing the code conventions, aiming to understand the difficulties faced and how they affected their development. We also used their responses from the questionnaire to understand what affected the implementation time during the experiment execution. We used their feedback to evaluate the code's readability and maintainability. Finally, we present threats to our validity.

8.1 | Implementation correctness

We analyzed the source code provided by the participants to determine if the code conventions implementation was correct. We excluded from the analysis the participants who could not implement the conventions in time or produced code that did not pass on all unit tests. However, we expected that some participants could not implement some of the conventions. To implement the code conventions, the participants needed to read the documentation about the Esfinge Metadata API, with which the participants had no previous familiarity. For the Reflection API, the participants had difficulties navigating and understanding the source code.

We can extract some information from the participants that failed to implement the code conventions. Only one participant failed to implement the code conventions using the Esfinge Metadata API and successfully implemented the tasks using the Reflection API. The total of participants that failed to implement the tasks with the Reflection API and successfully implemented the tasks with the Esfinge Metadata API was five. Then we evaluated the difficulties appointed by the six participants. The participant that failed to implement the task with the Esfinge Metadata API had difficulty understanding the tasks. The participant made some mistakes with the GitHub repositories, pushing the Esfinge Metadata code into the Reflection code repository. The participant states that after resolving his confusion with the code repositories implementing one of the tasks with Esfinge Metadata API was quick. However, spending so much time fixing the repositories problem made the participant run out of time to conclude the tasks. The participant states that the documentation about the Esfinge Metadata API was beneficial for the task accomplishment. This participant successfully implemented both tasks with Reflection API but stated that to implement the tasks deeper in the code. Although failing to implement one task with the Esfinge Metadata API, the participants stated that using the Esfinge Metadata API was much easier than implementing with Reflection and stated that his/her confusion with the code repositories contributed to his task failure.

The participant that failed to implement both approaches managed to implement one task with Reflection and none with the Esfinge Metadata API. For the tasks with Esfinge Metadata API, the participant stated that the exception message error from the Esfinge Metadata API was vague. Inspecting the participant's solution, we found that the participant did not provide the attribute convention annotations for its tasks, which led to the null pointer exception error. This feedback can help us to improve the error messages from the Esfinge Metadata API. The participant failed to implement one task with Reflection API. Reading his feedback, we see that navigating the source code wasted his time and was difficult to understand.

From the five participants that failed to implement the tasks with Reflection and managed to implement with the Esfinge Metadata API, the main reasons were navigating the source code, understanding where the conventions must be implemented, and understanding the documentation about the subject framework. The main reason for the failure was the implementation time, meaning the participants used their full time to implement one or no tasks. Some participants from this group state that they accomplished the tasks, but some unity tests are still failing.

Finding 1: *The success rate was higher with Esfinge Metadata than using Reflection API.*

This result met our expectations, even though the participants did not have previous experience with the Esfinge Metadata API. Implementing conventions with the Esfinge Metadata API relies on including annotations to the framework's existing annotations. To implement the same conventions with Reflection, the participants must add or change more lines of code. On top of that, as presented in Section 7, and according to the participants' experience implementing the tasks with Reflection API can be made using different methods and means. Also, it is made mixed with the framework processing logic, which requires the developer to inspect more source code than implementing with the Esfinge Metadata API.

8.2 | Implementation time

Based on the results collected from the questionnaire, we can perceive that the participants needed about the same time on average to implement the tasks with both approaches. We can notice in Figure 7 that the third percentile for the Esfinge was 36.25 minutes, almost equal to the average time for the Reflection approach, which was 36 minutes. That means that 75% of the participants implemented the tasks for the Esfinge Metadata API with lower or equal time to the average time of the Reflection API. Consequently, the average time necessary for implementing the tasks with Esfinge Metadata API was lower than with the Reflection API, where the former had an average of 29 minutes and the last 36 minutes. The highest value for both approaches was 60 minutes. However, for the Esfinge Metadata API, this value was considered one outlier since only one participant needed 60 minutes to implement the tasks. In contrast, this value appeared three times for the reflection API.

We expected much lower implementation times for the Esfinge Metadata API. Since implementing conventions requires less code to be written, the participants could implement the solution much faster. However, our result shows that the participants spent less time implementing the solution but reading the documentation was necessary. In contrast, for the Reflection API, the participants needed more time navigating the framework and understanding where the metadata must be processed for implementing the conventions. The difficulty of understanding and using the documentation on Esfinge Metadata API and navigating the code when implementing with Reflection made the implementation time very similar. With much more training and experience with the framework, the implementation time with the Esfinge Metadata API will tend to be as our initial expectation.

Finding 2: *The time necessary for implementing the conventions was similar with both approaches. However, the time was influenced by different reasons for each approach.*

By analyzing the data collected from Figure 8, we can see that understanding the documentation affected the implementation time of the participants since eight participants stated that s difficult. The learning curve disadvantage, presented in Figure 9 difficulty, complements this

argument since 10 participants stated that the large learning curve is a disadvantage for the Esfinge Metadata API. However, we see this result as positive since understanding the Esfinge Metadata API code conventions mechanism is a one-time-only task. After learning this approach, the users will not spend much time reading the documentation. However, for the Reflection API, disadvantages like complex code (eight), duplicated code (eight), and hard to maintain (seven), as presented in Figure 9, are independent of learning the Reflection approach and are dependent on the application being implemented. Another aspect that supports this affirmation is that for the Reflection API, the participants stated difficulties related to the source code and the implementation of the tasks. According to Figure 8, the most frequent answers were navigating the source code, finding where the conventions must be implemented, and implementing the features, as well as using the Reflection API. These appointed difficulties can also justify that the time necessary for implementing code conventions using the Reflection API will rely on navigating the code.

8.3 | Code complexity

The responses of the participants are consistent with their produced code. We can see in Table 7 that the Reflection API code conventions had more lines of code. Also, the participants who managed to implement the code conventions with few line codes, like participant P5, produced much more complex line codes than the Esfinge Metadata API. Since the Esfinge Metadata API relies only on inserting code annotations to the framework annotations, when one line of code is added to the Esfinge Metadata implementation, the only code element added is a code annotation. However, participant P5 used 12 method calls, one attribution, and one conditional for the Reflection API. We can also note that participant P10 implemented the Esfinge Metadata API code conventions using four lines and three with the Reflection approach. However, participant P10 made three method calls, one conditional or loop and two attributions.

Finding 3: *The implementations with Reflection API produced more complex source codes than the solutions with Esfinge Metadata API.*

As expected, the Esfinge Metadata API produced much less complex code. This extra complexity happens because the Esfinge Metadata API approach configures code conventions in the framework annotations. However, when using the Reflection API, the logic is implemented mixed with the metadata reading code. The participants had also implemented very similar code for different conventions, adding some redundancy to the source code. We believe this complexity will grow even more if more conventions are added since processing them with reflection requires more lines of code to be written, and in some cases, complex methods must be implemented. This result reinforces one of the findings of Guerra et al.[18], where the authors also find that frameworks implemented with the Esfinge Metadata API produce less complex code. On top of that, we can find evidence about the code complexity when observing Listing 14, which presented the solution of the convention for the framework annotation `@IsParameterPresent` using Reflection API. This solution was implemented by participant P1 and Listing 15, where the solution for the same convention is implemented using the Esfinge Metadata API.

Finding 4: *The students had a better development experience with the novel approach.*

8.4 | Threats to validity

The first half of this experiment was conducted in class, and the students could finish remotely in their environment. They were responsible for measuring their own execution time. Some participants may have a better environment, with fewer interruptions, and could focus better on the execution. Also, the time being measured by themselves could lead to some errors. Another threat to our validation is that we only had undergraduate students meaning some of them may not have any experience implementing or developing a metadata-based framework. To mitigate this threat, we tried to provide documentation about the Esfinge Metadata API and lessons about the Reflection API. We also provided the steps that they must follow in this experiment to ensure that the participants executed the tasks in the defined order and followed the same guidelines for concluding the experiment.

The participants had training for both approaches. However, they had different training for each approach. The participants had several classes (in person) about Java Reflection API. For the Esfinge Metadata API, they had access to the provided documentation, and it was their first interaction with the framework. Also, Reflection API has much documentation, such as tutorials, Javadoc, and other media sources. The participants had only the provided documentation for the Esfinge framework, and tutorials about the framework are less common when compared with Reflection API. Another point about the documentation is that the researchers prepared the documentation, meaning that our previous knowledge about the framework made us miss some crucial aspects when producing the documentation. The total number of students is another threat to our conclusions. Since we had a small number of participants, we cannot generalize our results for a large population. We evaluated the responses and

the produced source code qualitatively. We had to manually inspect the source code, meaning our knowledge about evaluating the Reflection and Metadata may have affected our conclusions.

9 | CONCLUSION

In this paper, we proposed a model that allows the developers to define code conventions separately from the code that reads code annotations. Our model enables a declarative definition of conventions over configuration, which can be configured separately from the code that reads the metadata. To use this model, developers must add convention annotations on the framework annotations declaration to configure conventions. We implemented this approach to the Esfinge Metadata API. As a first case o use, we added conventions over configuration to the Esfinge Comparison API to measure the necessary effort for implementing this approach to an existing application. The result of this study was very motivating as we were able to configure conventions annotations to the framework annotations substituting their usages with just a few lines of code.

We conducted an experimental study with 28 undergraduate students to evaluate our proposed model. The students had to implement two code conventions with our approach and two other conventions with the Reflection API. The students measured the time necessary for implementing the code conventions with both approaches. After implementing the conventions, the participants answered a questionnaire. We used the data collected from the questionnaire answers to evaluate how this novel approach can support the development of code conventions for metadata-based frameworks. Our analysis suggests that the proposed approach produced more readable and easier-to-maintain codes. However, the participants had difficulties initially stating that this approach requires a significant learning curve. The total time necessary for the students to implement the conventions was similar for both approaches, but they had different difficulties. While they needed to spend time reading and learning about the Esfinge API for the Reflection approach, the participants had to navigate the subject framework source code for more time.

As an extension of this work, we intend to implement new features to the Esfinge Metadata API to support the definition of code conventions in external files. That feature will allow applications to extend the framework conventions, enabling the introduction of application-specific conventions. Future works might also extend this support for creating decoupled code conventions to existing frameworks. That can be done by adding the framework annotations in the class bytecode at compile time based on conventions specified in a configuration file. Another future study can investigate the impact of replacing annotations with conventions in existing applications, verifying its potential to reduce the number of annotations by removing annotation repetition.

ACKNOWLEDGMENTS

We want to thank the support granted by FAPESP (São Paulo Research Foundation, grant 2019/12743-4 and 2021/00071-1).

References

1. Guerra EM, Souza dJT, Fernandes CT. A Pattern Language for Metadata-based Frameworks. In: CONFERENCE ON PATTERN LANGUAGES OF PROGRAMS. ACM; 2010; New York

2. Rocha H, Valente MT. How Annotations are Used in Java: An Empirical Study.. *SEKE* 2011: 426–431.

3. Dyer R, Rajan H, Nguyen HA, Nguyen TN. Mining billions of AST nodes to study actual and potential usage of Java language features. *Proceedings of the 36th International Conference on Software Engineering* 2014: 779–790.

4. Lima P, Guerra E, Nardes M, Mocci A, Bavota G, Lanza M. An annotation-based API for supporting runtime code annotation reading. *Proceedings of the 2nd ACM SIGPLAN International Workshop on Meta-Programming Techniques and Reflection* 2017: 6–14.

5. Yu Z, Bai C, Seinturier L, Monperrus M. Characterizing the Usage, Evolution and Impact of Java Annotations in Practice. *IEEE Transactions on Software Engineering* 2021; 47(5): 969-986. doi: 10.1109/TSE.2019.2910516

6. Stackoverflow . Arguments Against Annotations. https://stackoverflow.com/questions/1675610/arguments-against-annotations; 2009.

7. Bugayenko Y. Java Annotations Are a Big Mistake. Personal blog: https://www.yegor256.com/2016/04/12/java-annotations-are-evil.html; 2016.

8. Warski A. The case against annotations. SoftwareMill Tech Blog: https://blog.softwaremill.com/the-case-against-annotations-4b2fb170ed67; 2017.

9. Jungle T. Annotations and its benefits in java. http://technicaljungle.com/annotations-in-java-introbenefits-avoidance/; 2018.

10. 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?. *Proceedings of the 3rd ACM SIGPLAN International Workshop on Meta-Programming Techniques and Reflection* 2018: 15–22.

11. Deissenboeck F, Pizka M. Concise and consistent naming. *Software Quality Journal* 2006; 14(3): 261–282.

12. Fernandes C, Ribeiro D, Guerra E, Nakao E. Xml, annotations and database: a comparative study of metadata definition strategies for frameworks. *Proceedings of XATA 2010: XML, Associated Technologies and Applications* 2010: 115–126.

13. Chen N. Convention over configuration. http://softwareengineering.vazexqi.com/files/pattern.html; dez 2018.

14. Guerra E, Souza dJ, Fernandes C. Pattern language for the internal structure of metadata-based frameworks. In: Springer. ; 2013: 55–110.

15. Guerra E, Alves F, Kulesza U, Fernandes C. A reference architecture for organizing the internal structure of metadata-based frameworks. *Journal of Systems and Software* 2013; 86(5): 1239–1256.

16. Siqueira dJL, Silveira FF, Guerra EM. An approach for code annotation validation with metadata location transparency. In: Springer. ; 2016: 422–438.

17. Lima P, Guerra E, Nardes M, Mocci A, Bavota G, Lanza M. An Annotation-Based API for Supporting Runtime Code Annotation Reading. *Proceedings of the 2nd ACM SIGPLAN International Workshop on Meta-Programming Techniques and Reflection* 2017: 6–14. doi: 10.1145/3141517.3141856

18. Guerra E, Lima P, Choma J, et al. A metadata handling api for framework development: A comparative study. *Proceedings of the 34th Brazilian Symposium on Software Engineering* 2020: 499–508.

19. ECMA . ECMA - 334: C# language specification. https://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf; 2017.

20. JSR . JSR 175: A metadata facility for the Java programming language. http://www.jcp.org/en/jsr/detail?id=175; 2004.

21. 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* 2005: 584–600.

22. Schwarz D. Peeking inside the box: attribute-oriented programming with Java 1.5, Part. http://archive.oreilly.com/pub/a/onjava/2004/06/30/insidebox1.html; 2004.

23. Damyanov I, Holmes N. Metadata Driven Code Generation Using .NET Framework. In: ACM. ; 2004: 1–6.

24. Ernst MD. Type Annotations specification. JSR 308: http://types.cs.washington.edu/jsr308/; 2008.

25. Quinonez J, Tschantz M, Ernst M. Inference of reference immutability. *ECOOP 2008–Object-Oriented Programming* 2008: 616–641.

26. Lombok P. Project Lombok. https://projectlombok.org/; jul 2018.

27. JSR . JSR 220: enterprise JavaBeans 3.0. http://jcp.org/en/jsr/detail?id=220; 2007.

28. Córdoba-Sánchez I, Lara dJ. Ann: a domain-specific language for the effective design and validation of Java annotations. *Computer Languages, Systems & Structures* 2016; 45: 164 - 190. doi: https://doi.org/10.1016/j.cl.2016.02.002

29. Binkley D, Davis M, Lawrie D, Maletic JI, Morrell C, Sharif B. The impact of identifier style on effort and comprehension. *Empirical software engineering* 2013; 18: 219–276.

30. Hindle A, Godfrey MW, Holt RC. Reading beside the lines: Using indentation to rank revisions by complexity. *Science of Computer Programming* 2009; 74(7): 414–429.

31. Weinberger B, Silverstein C, Eitzmann G, Mentovai M, Landray T. Google C++ style guide. *Section: Line Length. url: http://google-styleguide. googlecode. com/svn/trunk/cppguide. xml# Line_Length* 2013.

32. Gnome Developer Documentation. https://developer.gnome.org/documentation/guidelines/programming/coding-style.html#; . Accessed: 2023-01-24.

33. Johnson J, Lubo S, Yedla N, Aponte J, Sharif B. An empirical study assessing source code readability in comprehension. In: IEEE. ; 2019: 513–523.

34. Gunawardena S, Tempero E, Blincoe K. Concerns identified in code review: A fine-grained, faceted classification. *Information and Software Technology* 2023; 153: 107054. doi: https://doi.org/10.1016/j.infsof.2022.107054

35. DeMichiel LG, Yalçinalp LÜ, Krishnan S. Enterprise javabeans tm specification, version 2.0. *On-line at< http://java. sun. com/products/ejb/docs. html>, year2001.(Cité page 57.)* 2001.

36. Miller J. Patterns In Practice-Convention over configuration.. *MSDN magazine* 2009: 39.

37. Ruby S, Thomas D, Hansson D. *Agile web development with rails*. Springfield: Pragmatic Bookshelf. 3 ed. 2009.

38. Marx D. Basic Java Persistence API Best Practices. https://www.oracle.com/technical-resources/articles/javaee/marx-jpa.html; 2008.

39. Nazário MFC, Guerra E, Bonifácio R, Pinto G. Detecting and Reporting Object-Relational Mapping Problems: An Industrial Report. *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* 2019: 1-6. doi: 10.1109/ESEM.2019.8870163

40. Guerra E, Cardoso M, Silva J, Fernandes C. Idioms for code annotations in the java language. *Proceedings of the 8th Latin American Conference on Pattern Languages of Programs* 2010: 1–14.

41. Perillo JRC, Guerra EM, Fernandes CT. Daileon: A Tool for Enabling Domain Annotations. *Proceedings of the Workshop on AOP and Meta-Data for Software Evolution* 2009. doi: 10.1145/1562860.1562867

42. Sabot-Durand A. JSR 365: Contexts and Dependency Injection for Java 2.0. https://docs.jboss.org/cdi/spec/2.0/cdi-spec.html; 2017.

43. JSR . Bean Validation 2.0 (JSR 380). https://beanvalidation.org/2.0-jsr380/; 2017.

44. Duncan S. The ThoughtWorks® Anthology, Essays on Software Technology and Innovation. *Software Quality Professional* 2009; 11(2): 53.

45. JSR . JAX-RS: The Java™ API for RESTful Web Services. https://www.jcp.org/en/jsr/detail?id=311; 2009.

46. JSR . JSR 220: Enterprise JavaBeans 3.0. :<http://www.jcp.org/en/jsr/detail?id=220>; 2007.

47. JSR . JSR 303: Bean Validation. https://jcp.org/en/jsr/detail?id=303; 2009.

48. JSR . JSR 299: Contexts and Dependency Injection for the JavaTM EE platform. http://www.jcp.org/en/jsr/detail?id=299; 2009.

49. Guerra E, Fernandes C, Silveira FF. Architectural patterns for metadata-based frameworks usage. *Proceedings of the 17th Conference on Pattern Languages of Programs* 2010: 1–25.

50. Hibernate . Hibernate Validator 8.0. https://hibernate.org/validator/; 2022.

51. Pages JS. Servlets and JSP Pages. https://www.oracle.com/technical-resources/articles/java/servlets-jsp.html; 2022.

52. RCP S. The spring rich client project.. http://www.springsource.org/spring-rcp; 2010.

53. Framework G. Genesis Framework. https://genesis.dev.java.net/; 2010.

54. Cavalcanti L. *VRaptor: Desenvolvimento ágil para web com Java*. Editora Casa do Código . 2014.

55. SwingBean . SwingBean: aplicações Swing a Jato!. http://swingbean.sourceforge.net/; 2009.

56. Yang HY, Tempero E, Melton H. An empirical study into use of dependency injection in java. *19th Australian Conference on Software Engineering (aswec 2008)* 2008: 239–247.

57. Guerra E. Design patterns for annotation-based apis. *Proceedings of the 11th Latin-American Conference on Pattern Languages of Programming, SugarLoafPLoP* 2016; 16: 9.

58. Gamma E, Helm R, Johnson R, Vlissides J, Patterns D. Elements of Reusable Object-Oriented Software. *Design Patterns* 1995.

59. Lima P, Guerra E, Meirelles P, Kanashiro L, Silva H, Silveira F. A Metrics Suite for code annotation assessment. *Journal of Systems and Software* 2018; 137: 163 - 183. doi: https://doi.org/10.1016/j.jss.2017.11.024

60. Guerra E, Fernandes C. A qualitative and quantitative analysis on metadata-based frameworks usage. In: Springer. ; 2013: 375–390.

61. Guerra E, Dias ADO, Véras LGD, Aguiar A, Choma J, Da Silva TS. A Model to Enable the Reuse of Metadata-Based Frameworks in Adaptive Object Model Architectures. *IEEE Access* 2021; 9: 85124–85143.

62. Lima P, Melegati J, Gomes E, Pereira NS, Guerra E, Meirelles P. CADV: A software visualization approach for code annotations distribution. *Information and Software Technology* 2022: 107089. doi: https://doi.org/10.1016/j.infsof.2022.107089

63. Guerra E, Gomes E, Ferreira J, et al. How does annotations affect Java code readability?. 2022.

64. Guerra E, Fernandes C, Silveira FF. Architectural Patterns for Metadata-Based Frameworks Usage. *Proceedings of the 17th Conference on Pattern Languages of Programs* 2010. doi: 10.1145/2493288.2493292

65. Guerra E, Silveira F, Fernandes C. Classmock: A testing tool for reflective classes which consume code annotations. *Proceedings of the Brazilian Workshop for Agile Methods (WBMA 2010)* 2010: 1–14.

66. Falessi D, Juristo N, Wohlin C, et al. Empirical software engineering experts on the use of students and professionals in experiments. *Empirical Software Engineering* 2018; 23(1): 452–489. doi: 10.1007/s10664-017-9523-3

67. HÖst M, Regnell B, Wohlin C. Using Students as Subjects—A Comparative Study OfStudents and Professionals in Lead-Time Impact Assessment. *Empirical Software Engineering* 2000; 5(3): 201–214. doi: 10.1023/A:1026586415054

68. Vegas S, Apa C, Juristo N. Crossover designs in software engineering experiments: Benefits and perils. *IEEE Transactions on Software Engineering* 2015; 42(2): 120–135.

How to cite this article: E. Gomes, E. Guerra, P. Lima, and P. Meirelles (2023), An Approach Based on Metadata to Implement Convention over Configuration Decoupled from Framework Logic, *Journal of Software: Evolution and Process*, *2023;00:1–6.*