

RESEARCH ARTICLE - TECHNOLOGY OPEN ACCESS

An Approach Based on Metadata to Implement Convention Over Configuration Decoupled From Framework Logic

Everaldo Gomes¹  | Eduardo Guerra² | Phyllipe Lima³ | Paulo Meirelles¹¹Institute of Mathematics and Statistics, University of São Paulo, São Paulo, São Paulo, Brazil | ²Faculty of Engineering, Free University of Bozen-Bolzano, Bolzano, Italy | ³Institute of Mathematical and Computing Sciences, Federal University of Itajubá, Itajubá, Minas Gerais, Brazil**Correspondence:** Everaldo Gomes (everaldogjr@usp.br)**Received:** 30 March 2023 | **Revised:** 8 April 2025 | **Accepted:** 25 April 2025**Funding:** This study was supported by FAPESP (São Paulo Research Foundation, grants 2019/12743-4 and 2021/00071-1).**Keywords:** code annotation | code convention | convention over configuration | metadata configuration | software framework

ABSTRACT

Frameworks are essential for software development, providing code design and facilitating reuse for their users. Well-known Java frameworks and APIs rely on metadata configuration through code annotations, using Reflection API to consume and process them. Code elements that share the same annotations often exhibit similarities, creating the opportunity to use conventions as a metadata source. This paper proposes a model for defining Convention over Configuration (CoC) for annotation usage, decoupled from the metadata reading logic. With this model, if a convention is present, the framework will automatically consider that element to be annotated. We implemented this model in the Esfinge Metadata API and evaluated it in an experiment where participants implemented the CoC pattern using two approaches: our proposed one and the Java Reflection API. As a result, 75% of participants implemented our approach faster than with just the Reflection API, and we observed a higher failure rate with the Reflection API than with the Esfinge API. Moreover, the code produced with our approach also resulted in fewer lines of code. Based on these results, we confirmed that the proposed approach fulfilled its goal of supporting the definition of conventions decoupled from the framework logic, thereby improving code readability and maintainability.

1 | Introduction

Frameworks are incomplete software that developers can specialize to add application-specific behavior. By offering a set of abstractions, developers can compose and extend frameworks to create a complete application. Metadata-based frameworks [1] utilize metadata configuration to determine how to adapt behavior for each class. In the Java language, developers primarily use code annotations for metadata configuration. Widely used frameworks like Spring, Hibernate, and JUnit incorporate 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. Some empirical studies evidence the existence of classes overloaded with annotations and highly repeated configurations [2, 3]. Another abuse of the code annotations usage is that some annotations can require extensive configuration [4]. 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. For instance, practitioners complain that annotations mix configuration with actual source code [5]. Developers also claim that annotations are against the encapsulation principle [6] since the object behavior is not defined solely by its methods. Although the compiler checks the annotation parameters and if the code

This is an open access article under the terms of the [Creative Commons Attribution](https://creativecommons.org/licenses/by/4.0/) License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited.

© 2025 The Author(s). Journal of Software: Evolution and Process published by John Wiley & Sons Ltd.

element being annotated matches the annotation target type, it is not guaranteed if the context where the annotation is being used is correct [7, 8]. In contrast, annotations are less verbose than XML files [5] and bring configuration close to the source code. The code conventions approach can reduce the metadata configuration.

Code conventions are guidelines for source code writing. Adopting code conventions is important when a development team collaborates in a project since it can improve the code readability and maintainability [9]. These guidelines include several aspects of the development process, such as naming variables, code formatting, and documentation. Additionally, developers can use them as an approach for metadata definition [10] based on a pattern called *Convention over Configuration* [11]. This practice can significantly reduce the number of code annotations and other types of configuration [4], 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 developers could use design patterns for reading metadata, such as *Metadata Reader Strategy* and *Metadata Reader Chain* [12, 13], they would still need to check the convention and identify the corresponding metadata programmatically. We discuss these design patterns in Section 2.3. Additionally, practitioners report that conventions are often concealed in the source code, leading to unwanted or unexpected results [14, 15]. Lastly, developers must write extra code to overwrite some conventions imposed by frameworks [16].

In this work, we propose a model for defining a solution to support the usage of convention over configuration for metadata-based frameworks to aid developers in implementing metadata-based frameworks. Our approach uses additional convention annotations to configure 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 return the same value transparently 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 and decoupled from the framework logic.

We created a reference implementation as a feature of the framework *Esfinge Metadata* [17–19] 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], a framework that compares two class instances. This framework uses the *Esfinge Metadata* API as its metadata reading API [18]. In this study, we

refactored the *Esfinge Comparison* framework, adding convention annotations to the target annotations.

We evaluated our solution by employing 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 *Esfinge 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, our proposed approach decoupled the added code from the subject framework logic, and the participants found it 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. With this novel approach, we noted a significantly faster development speed in contrast with the usage of the Reflection API. The participants identified the learning curve as the main difficulty of this approach. The code complexity was also considered significantly lower.

The remainder of this paper is structured 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 detailed in Section 4. In Section 5, we present a case study for implementing conventions for the *Esfinge Comparison* annotations. We present the study designed to evaluate our approach in Section 6. Section 7 presents the results obtained, and in Section 8, we discuss the results and present the threats to validity. Finally, we conclude this paper in Section 9.

2 | Background

We define 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 those that receive the convention. When we refer to frameworks in general, we say **framework**. Regarding 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 following subsections begin by discussing metadata configuration and how some programming languages implement this feature. We also introduce the concept of Code Conventions and how tools or frameworks use this practice. Then, we discuss metadata-based frameworks, presenting the design patterns used by these frameworks to reduce the use of

annotations. Finally, we present the impacts of code annotation usage in code development.

2.1 | Metadata Configuration

The term “metadata” is used in various contexts in computer science. 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 [10]. However, in this approach, the metadata and the code element are distant since the external file needs to reference the element. Another drawback of this approach is its verbosity, as it requires providing the complete path for the framework to consume the metadata correctly. Another alternative some frameworks use for metadata definition is code conventions, which we further explore in Section 2.2.

Some programming languages provide features that allow custom metadata to be defined and included directly on programming elements, like attributes in C# [20] and code annotations in Java [21]. 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, unlike other approaches, developers explicitly define the metadata in the source code. Some authors call the usage of code annotations attribute-oriented programming since it marks software elements [22, 23]. Annotations can be used for source code generation [24], compile-time verification [25, 26], class transformation [27], and framework adaptation [1].

Some base APIs 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 also respond 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].

Developers can use code conventions as an alternative to defining code metadata to be consumed by frameworks and APIs [10]. 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 [11, 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], and other frameworks followed it. A complementary practice, Configuration by Exception, considers defining metadata on elements that deviate from 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, developers cannot use it to define complex metadata. For instance, a code convention would be suitable for defining 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, which 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 [12, 13], 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 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 present in 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 [2, 19], and the Daileon API [41] provide transparent support to implement this pattern. Some official Java APIs also support the Annotation Mapping mechanism,

implementing domain annotations [42] that represent at a higher level domain metadata 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 work with [1]. Popular Java frameworks such as Spring, Hibernate, and JUnit use code annotations to configure metadata.

Developers often design metadata-based API using practices not present in 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. 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 for annotations inside the created custom annotation.

Many different contexts employ Metadata-based frameworks. A previous work of Guerra et al. [43] 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 [44] 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 [45, 46]. (iii) *Configured Handler* uses the metadata to mark methods that should handle events [47, 48]. (iv) *Crosscutting Metadata Configuration* uses metadata to configure variations on a crosscutting behavior [49]. The *Dependency Injection* [50] 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* [51] 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 are classified into three categories [12]. (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 designing 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. The subject framework implements this pattern 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* [52]. 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 [19] 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 [3, 53]. Empirical studies presented evidence of extensive annotations, classes with a high number of annotations [53], annotations repetition [4], and that they might introduce code problems [3].

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 [54, 55]. These studies also pointed out that the usage of annotations might reduce the coupling.

The study of Lima et al. [53] 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 the 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 [4]. 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 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. [3] 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 annotation changes occurred during project evolution, implying that some developers 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.

Guerra et al. [19] 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. [56] 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, annotation usage has positive and negative impacts. Annotations can reduce the coupling [54], provide a consistent code evolution [19], and improve the code quality [3]. The literature presents some drawbacks to the usage of annotations. Annotations declaration can be extensive [53] and used to configure similar code elements repetitively on the source code [4]. Another negative impact is that annotations can make the code hard to debug [54]. Annotations can be wrongly used, inconsistent, and redundant in the source code [3]. Finally, a study on developers' perceptions of code readability could not reach a consensus about annotation usage and code readability [57].

Despite several strategies mitigating the use of annotations, and in particular, some metadata-based frameworks implementing the Convention Over Configuration pattern, the major issue is that frameworks or APIs must implement convention reading as part of their metadata reading approach. In this context, the following section presents our model to decouple convention over configuration reading from the code that reads annotations.

3 | A Model for Implementing a Decoupled Solution for Convention Over Configuration

A model that allows decoupling the definition of code conventions from the code that reads framework annotations should be transparent for code that checks for the presence of a piece of metadata, whether the corresponding target annotation is present or if the code element matches a given convention. One of the goals of our proposed 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 a target annotation is 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 has 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 [12], whose participating classes are highlighted by a stereotype with the same name. This pattern creates a repository to store metadata at runtime, avoiding unnecessary metadata readings and, in this case, encapsulating the identification of the conventions. 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.

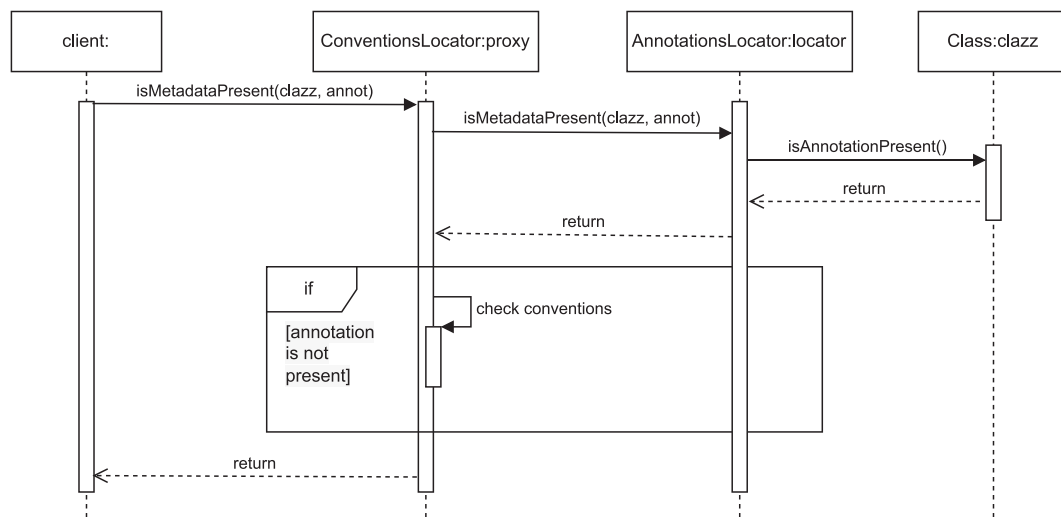


FIGURE 1 | Sequence diagram for checking for conventions transparently.

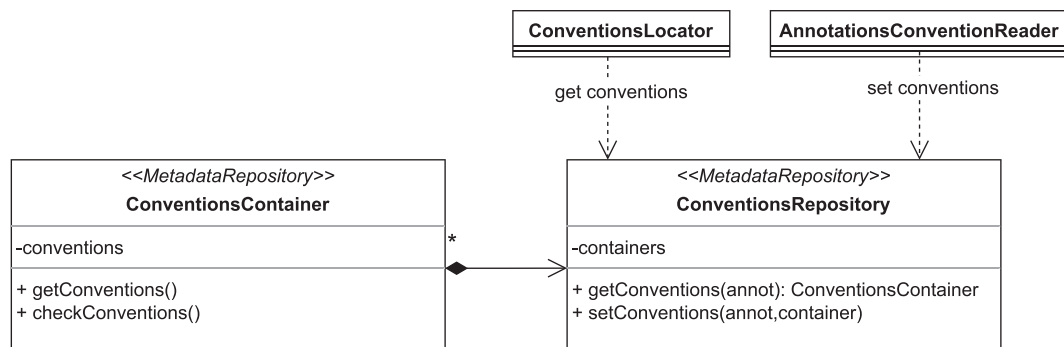


FIGURE 2 | Getting conventions from the annotation definition.

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 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.

4 | Esfinge Metadata Support for Conventions Over Configuration

The Esfinge Metadata was developed to support the process of reading and validating metadata [17, 18]. Unlike Java Reflection API, which only provides methods for retrieving annotations from code elements, Esfinge Metadata API uses Metadata Mapping [58] to map information on class annotations to attributes of an object. This information is helpful to keep this metadata at runtime [19]. Its structure guides the developers to implement metadata-based framework patterns and best practices [12, 13]. 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 [19]. 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 newly implemented functionality 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 [18, 19].

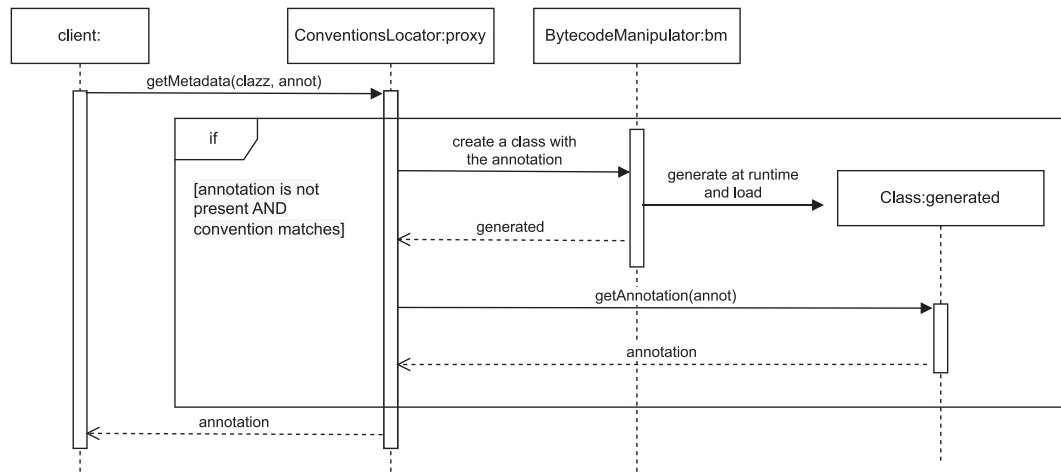


FIGURE 3 | Generating a class at runtime to get the annotation targeted by the convention.

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 and “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 has 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.

```

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.

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 the element matches the convention. Consider Listing 5 where we present the implementation of the `SuffixConventionVerifier`. 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 returns with the value set in the respective attribute. Consider the class `ConfigValues` declared in 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 `AttributeConventionValueGenerator` 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 have their retention policy, which must be runtime (line 1), and the target element must be a method (line 2). Each attribute convention has its generator associated (line 3). Then, the value `FixedStringGenerator.class` is a concrete implementation of the generator for the `@FixedStringValue` annotation.

```
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.

Listing 9 presents the implementation of the `FixedStringGenerator` class. The interface `AttributeConventionVerifier` has one single method (`generateValue`) responsible for generating the value for the attribute convention. In this case, the generator must return the name of the element for which the convention matches.

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

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.

When a code element matches the convention, returning an instance of the respective target annotation is necessary. 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

```
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.

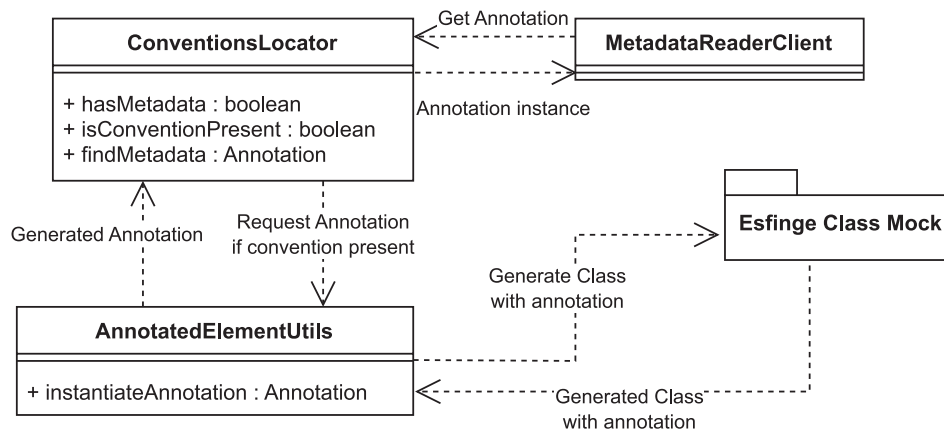


FIGURE 4 | Getting generated annotations at runtime.

framework annotation. If it matches, the locator requests the class `AnnotatedElementUtils` to generate the bytecode of a class with target annotation used as a source at runtime. The `AnnotatedElementUtils` uses the `Esfinge ClassMock` [59], a framework that generates classes at runtime using bytecode manipulation, to generate a class configured with the target annotation. After getting the generated class from `Esfinge ClassMock`, the `AnnotatedElementUtils` retrieves the respective annotation instance and returns it to the conventions locator.

5 | Esfinge Comparison Case Study

`Esfinge Comparison` [12] 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 for which we added conventions are listed below.

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

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 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.

TABLE 1 | Conventions used for comparison annotations.

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 <i>double</i> will be considered as having the @Tolerance target annotation, with a default value of 0.01.

```

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 complete source code of this case study is available at the Esfinge Comparison framework Github repository¹, at the branch "conventions," where all changes and tests can be accessed.

6 | Evaluation Study Design

To evaluate our model proposed in Section 3, we conducted an experiment addressing the following Research Question (RQ):

RQ: How does the proposed model for Conventions over Configuration implementation impact the development of a metadata-based framework? To answer the question, first, we implemented the model presented in Section 3 on the Esfinge Metadata API. We designed programming tasks for the participants to implement conventions in an existing subject framework. We designed four tasks, meaning we had four different conventions but with similar behavior. 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. After completing the tasks, the participants answered an evaluation questionnaire. We then analyzed their solution, observing the changes made to the code.

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.

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, referencing the exact specification used in another study [19]. 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.

We executed the experiment in a class of 28 students who tried to execute the tasks and answered a questionnaire. We divided the students into four groups of 7 students. The groups are described in Table 2. The groups classify the students relative to the implemented convention and the approach used. The tasks are detailed in Table 3. After implementing its tasks, the participants answered an online 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 specifications for the correctness analysis. We then evaluated the changes made by the participants, considering only the students who managed to implement the tasks correctly. The last step was to evaluate the questionnaire responses based on participants' experiences with difficulties, advantages, and disadvantages while implementing the conventions with both approaches.

TABLE 2 | 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.

TABLE 3 | 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 <code>@IsParameterPresent</code> 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 <code>@ParameterPrecision</code> with the parameter decimalPlaces equals to 1.
B	1	Any field ending in "Text" should be considered as having the framework annotation <code>@TextValue</code> with the parameter value equal to what comes before this suffix.
B	2	Any field of non-primitive type with the <code>@OneToOne</code> should be considered as having the framework annotation <code>@CompositeParameter</code> .

To avoid any participant bias, we used the crossover design [60]. This design suggests that the participants perform their tasks in different orders, reducing the variability of the results and improving the result's confirmability. In our study, different participants had to implement two different conventions using two different approaches, the Reflection API and the Esfinge Metadata API. However, since we followed the crossover design,

some participants were asked first to implement conventions using the Reflection API. In contrast, other participants first implemented their conventions with the Esfinge Metadata API. The participants were asked to implement the tasks in different frameworks. For instance, some participants implemented conventions A and B using the Reflection API and conventions C and D using the Esfinge Metadata API. On the other hand, some participants implemented conventions A and B using the Esfinge Metadata API and conventions C and D using the Reflection API.

Carrying out experiments using students is a feasible approach and a common option to advance the software engineering field via empirical studies. Salman et al. [61] showed that developers and students performed similarly when experimenting with new approaches. In our experiment, we provided the participants with a complete implementation of the subject framework, with a suite of tests whose two failed tests. The failing tests are related to the conventions that must be implemented. In other words, the participants experimented with a novel approach.

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 using only Reflection API. The exact versions were developed for a previous study that compared the usage of these two APIs [19], 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 and mapped to a JavaBean class. This mapping is done through metadata configurations using the framework annotations.

This subject framework implements the Metadata Container pattern [12], 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:

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.

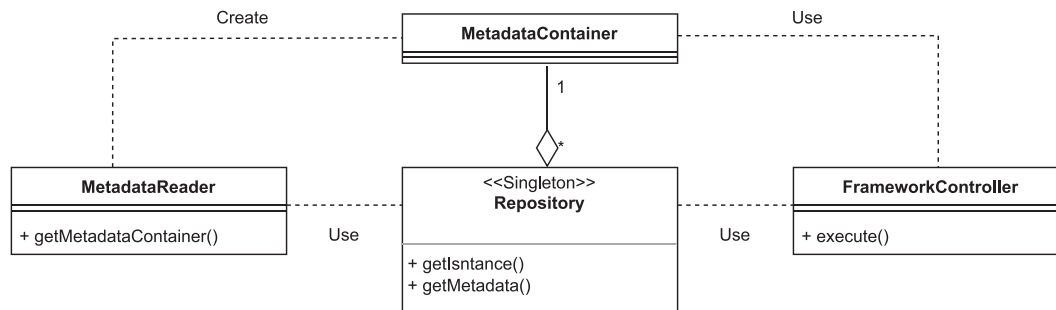


FIGURE 5 | Basic structure of the Metadata Container pattern.

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.

each scenario. We divided the tasks into two groups **A** and **B** according to Table 3. Each group of tasks, A and B, has two tasks. The column “Convention to be Implemented” presents the description of the task that the student will perform. The students had to improve the `readParameters` method, checking if the class attributes match a specific convention. For instance, if a participant had to implement Task 1 of Group A, then this par-

```

1 public class ParamMapper {
2     public <E> E map(String[] args, Class<E> paramClass){
3         ParamMapperContainer c = readMetadata(paramClass);
4         E paramInstance = (E) readParameters(args, c);
5         return paramInstance;
6     }
7     private ParamMapperContainer readMetadata(Class<?> paramClass){
8         //create an instance of ParamMapperContainer
9         //add information based on the annotations
10        //return it
11    }
12    private Object readParameters(String[] args, ParamMapperContainer c){
13        //Create an instance from the parameter class
14        //Use metadata container to know what should be in
15        //each class field.
16        //add information based on the annotations
17        //return it
18    }
19 }
  
```

LISTING 13: Subject framework internal structure.

We provided a concrete implementation of the `ParamMapper` class to the participants. We omitted the code since it is extensive. Listing 13 presents the `paramMapper` class. The class `ParamMapper` is responsible for mapping the parameters received from the command line to the instance of a class instance. The main method, `map`, returns the instance of the class. The `readMetadata` method creates the container with the parameters. The method `readParameters` is responsible for processing the parameter container and using its information to create the class instance. The `ParamMapperContainer` class was also provided to the students. This class is a container with the parameters received by the `paramMapper` 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

participant had to write the code to check if the class field type is boolean and if its name ends with “Present”. If the class attribute matches the convention, then the boolean field matching the convention must be instantiated with the parameter value.

6.3 | Code Repositories

The repositories for each participant, one containing the implementation using the Esfinge Metadata API and another using the Reflection API, belong to a GitHub organization.² Each participant had access only to their private repositories. For instance, the repository names are like *participantOne-metadata* (Esfinge Metadata repository) and *participantOne-reflection* (Reflection repository), where *participantOne* is the participant's GitHub username. The repositories from other participants were invisible to *participantOne*. The participants had access to commit and pull the source code, and their solutions had to be pushed to the correct repository. For instance, the code present in *participantOne-reflection* had to contain the implementation of the tasks performed by *participantOne* using the Reflection API. We used the code present in these repositories to evaluate the participants' solutions.

TABLE 4 | Questionnaire questions.

ID	Question
Overall framework experience	
Q1	How many tasks have you accomplished?
Q2	How much time did you spend 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 need 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	What are the advantages of using Esfinge Metadata API for creating Code Conventions compared to the other approach?
Q2	What do you consider the disadvantages of Esfinge Metadata API for creating Code Conventions compared to the other approach?
Q3	What are the advantages of using standard Java reflection API for creating Code Conventions compared to the other approach?
Q4	What are 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

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 task 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 and search for other information sources to help implement the tasks. We considered reading the documentation and implementing the conventions as the total time necessary for concluding the tasks.

6.5 | Questionnaire Design

The questionnaire is composed of two parts. The first part asks for the participant's familiarity with annotations and their primary programming language. In the second part, the participants are asked about their overall experience and the approaches to evaluation. Table 4 present the questions of the second part of the questionnaire. The questions Q1–Q9 are asked

twice, once for the experience implementing with the Reflection API and one time for the Esfinge Metadata API. Lastly, a set of 6 questions is asked to the participants, where we ask them to evaluate each approach.

The first set of questions asked the participant 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

(Continues)

TABLE 5 | (Continued)

ID	Question
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 volunteered participants, one did not accept the repository invitation. Of the 27 participants who 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 who participated in the experiment, we had 12 participants who correctly implemented the tasks in both frameworks. We had five participants who 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 implemented with the Esfinge Metadata API with Java Reflection API.

We can see that the number of participants who failed to accomplish any reflection task (five) is higher than the number of participants who failed to implement the Esfinge Metadata

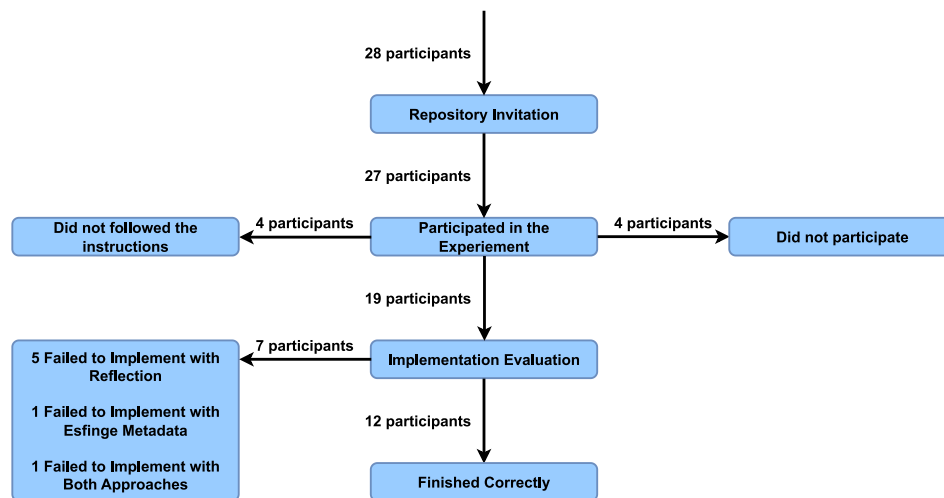


FIGURE 6 | Total of participants for each experiment step.

TABLE 6 | Correctness according to participants' familiarity with Java and code annotations.

Correctness by participants' primary language		
Language	Succeed	Failed
Java	7	7
Other	5	4

Correctness by participants' familiarity with code annotations		
Code annotation familiarity	Succeed	Failed
1	0	1
2	4	2
3	3	4
4	4	4
5	1	0

tasks (one), which means that participants had more difficulty implementing the tasks with reflection in this 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 who 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 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 spent much time reading the documentation, he could not accomplish the tasks on 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 the boilerplate code was frustrating, and it was hard to understand the framework's core, 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.

We further investigate if the primary programming language and familiarity with code annotations affected the correctness of the participants. Table 6 presents the distribution of participants according to their primary programming language and familiarity with code annotations. We compared if participants who declared as their primary language Java had more success implementing the tasks than participants who preferred another programming language. As we can see, 7 participants who declared Java as their primary language succeeded in implementing the tasks, and 7 participants who also declared Java as their primary programming language failed to implement the tasks. Five participants who are more familiar with other programming languages succeeded in implementing the tasks, while 4 others failed.

Accounting for the code annotation familiarity, participants declared their familiarity on a scale from 1 to 5, with 1 not familiar and 5 being very familiar with code annotations. The only one participant that declared its familiarity as 1 ailed in

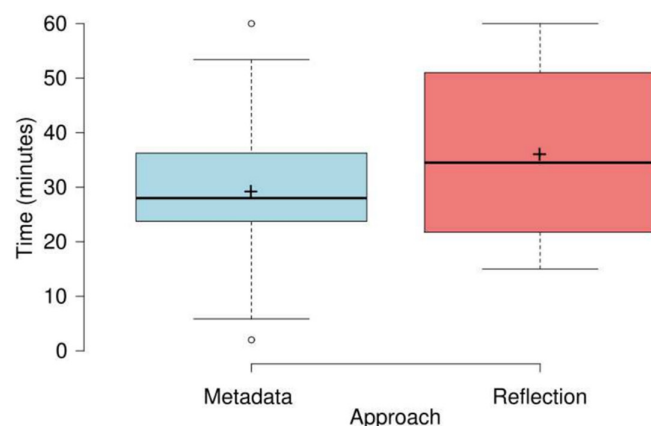


FIGURE 7 | Boxplot for implementation time for both approaches.

the experiment. Four participants that are not so familiar with code annotations succeeded in the experiment, while 2 failed. Three code annotations familiar participants succeeded in the tasks, and 4 failed. Four participants who have a good familiarity with code annotations succeeded, and 4 other participants who are also familiar with code annotations failed to implement the tasks. Only one participant who is very familiar with code annotations succeeded in implementing the tasks.

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 2 min, and for the Reflection API, the minimum was 15 min. We notice that 25% of the values for the Esfinge Metadata API are lower than 23.75 min against 21.75 for the Reflection API. The median for the Reflection API was 34.5 min, and the Esfinge Metadata API had a median of 28 min. Seventy-five percent of the Esfinge Metadata API sample had times lower than 36.25 and 51 min 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.

7.4 | Modified Lines of Code

Having access to the code repositories, we inspected the source code produced by the participants. Table 7 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 8, we can see that the Reflection lines of code have more elements than the Esfinge Metadata API. For instance, participant P5, who 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.

TABLE 7 | 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 8 | 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

```

1  this.hasIsParameterPresent = f.isAnnotationPresent(IsParameterPresent.class) ||
2      f.getName().endsWith("Present"); // modified code
3  if (...) { // existing code
4      IsParameterPresent pv = f.getAnnotation(IsParameterPresent.class); // inserted code
5      //omitted existing code
6      if (...) // 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 14: 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  public boolean hasIsPresentSuffixPresent() { // //getter for the created field as added code
9      return hasPresentSuffix; //added code
10 }

```

LISTING 15: Code solution of participant P3 for convention to @IsParameterPresent.

Listing 14 presents the convention's implementation for the @IsParameterPresent framework annotation, and Listing 15 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 are standard.

We present on listing 16 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 solutions 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 17. The participants P1, P3, P5, P7, P8, P9, and P10 implemented this convention using the Reflection API. Their solution was overall very similar. Listing 18 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 @SuffixConvention(value = "Present") //added code
2 public @interface IsParameterPresent {
3     @ElementNameBeforeSuffix(suffix = "Present") //added code
4     String name();
5 }

```

LISTING 16: Code solution for convention to @IsParameterPresent framework annotation using Esfinge Metadata API. appointed difficulties by the participants. Observing the par-

```

1 @HaveAnnotationOnElementConvention(annotationClass = OneToOne.class) // added code
2 public @interface CompositeParameter {
3 }

```

LISTING 17: 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 18: Code solution for convention to @CompositeParameter framework annotation using Esfinge Metadata API.

7.5 | Difficulties and Disadvantages

We collected the reported difficulties and disadvantages that each participant felt about implementing tasks with each approach from the questionnaire. 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

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

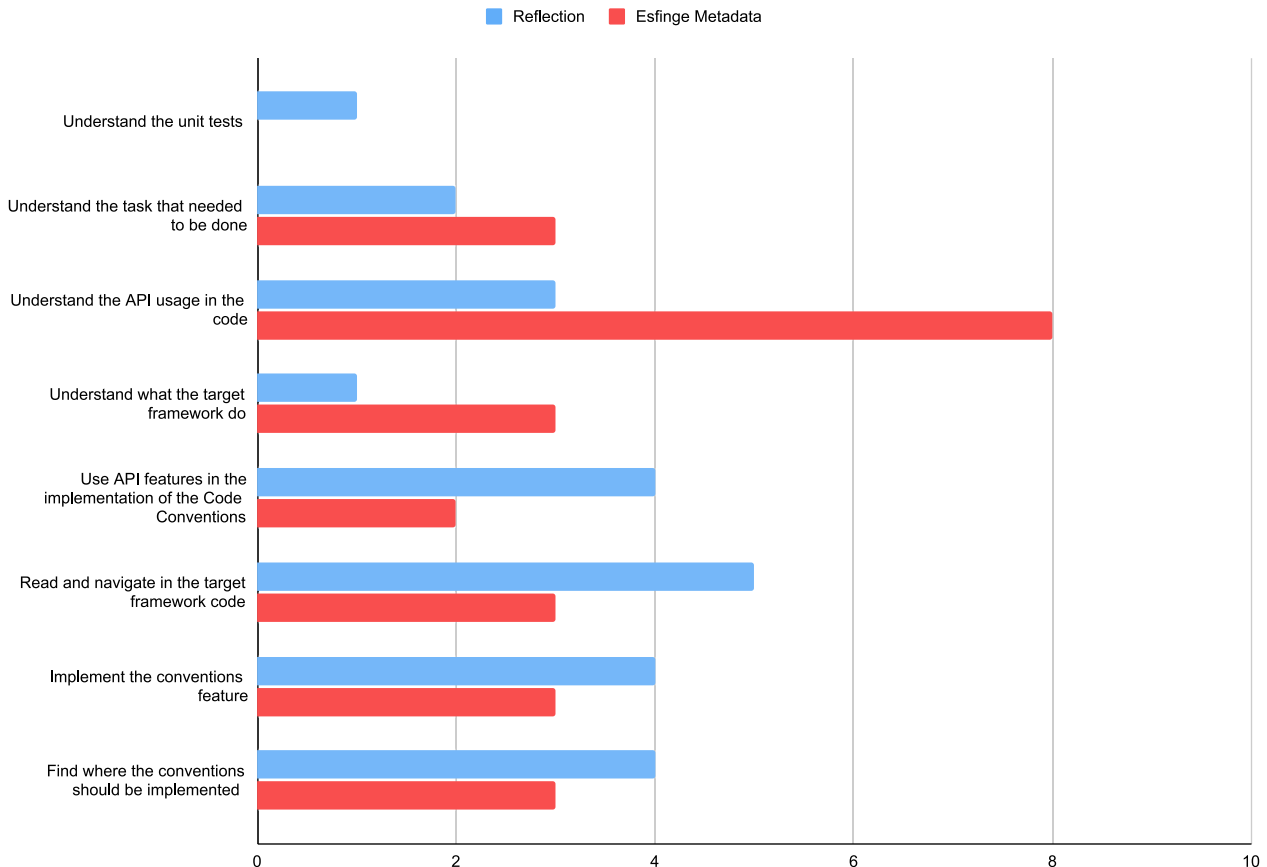


FIGURE 8 | Difficulties for both approaches.

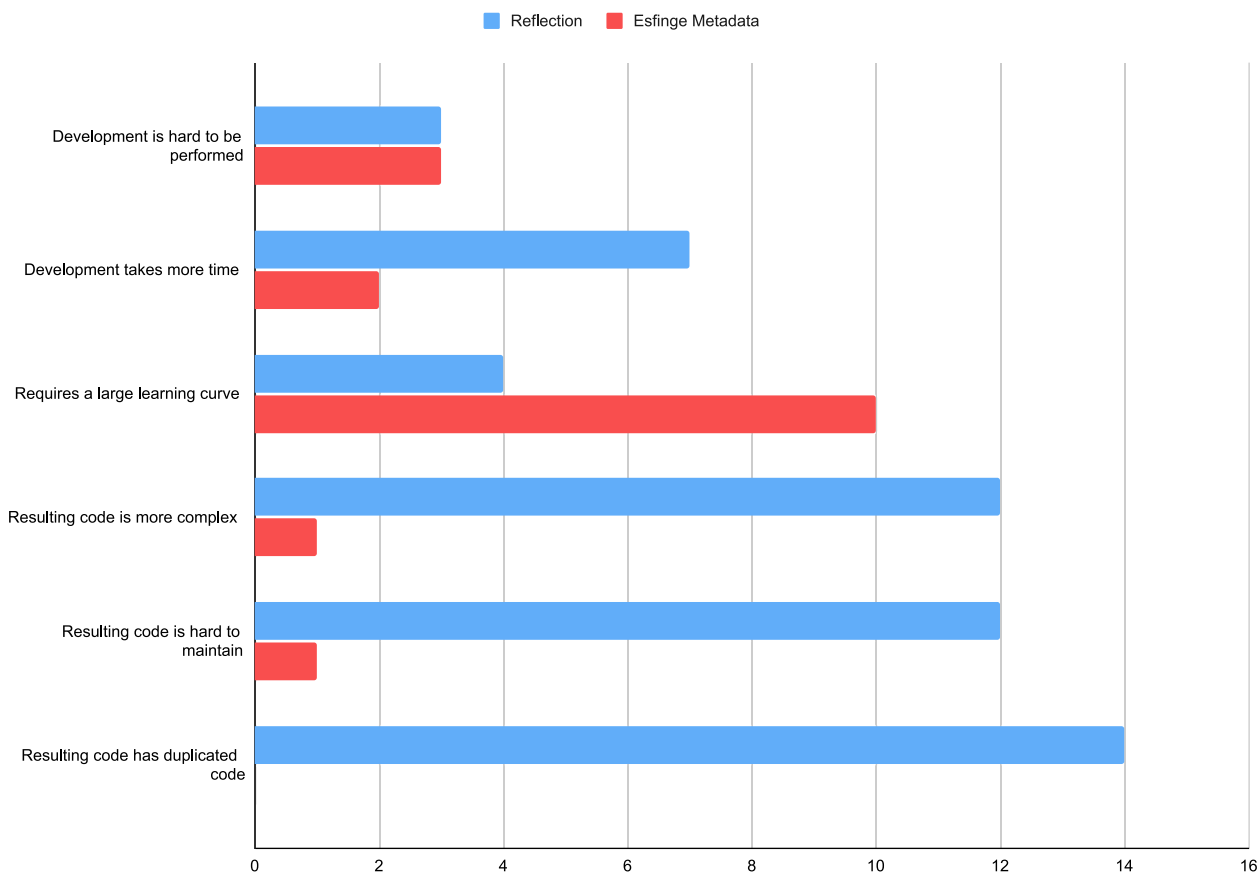


FIGURE 9 | Disadvantages for both approaches.

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.

7.6 | Advantages With Esfinge Metadata API Versus Reflection API

Figure 10 presents 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 (10) 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. Nine participants also stated that the Esfinge Metadata API produces a code that is easy to change, 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. According to Figure 11, a total of 11 participants preferred the Esfinge Metadata API, and one preferred the Reflection API. Most participants considered the learning curve necessary for using the Esfinge Metadata API difficult. 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. 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 although it is challenging at the early development stages, the framework outperforms Reflection regarding development time. Participants P3, P6, P7, P9, and P12 stated that even with

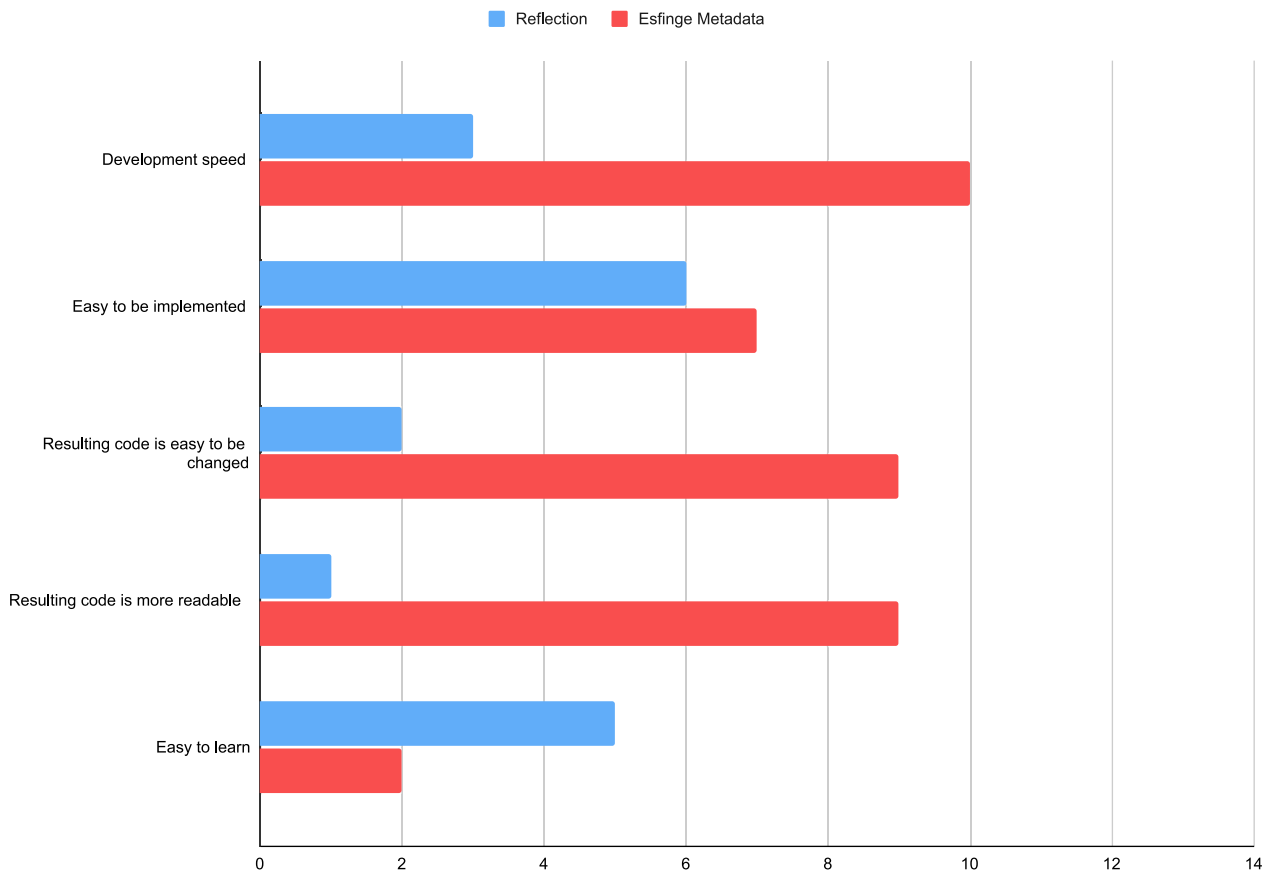


FIGURE 10 | Advantages for both approaches.

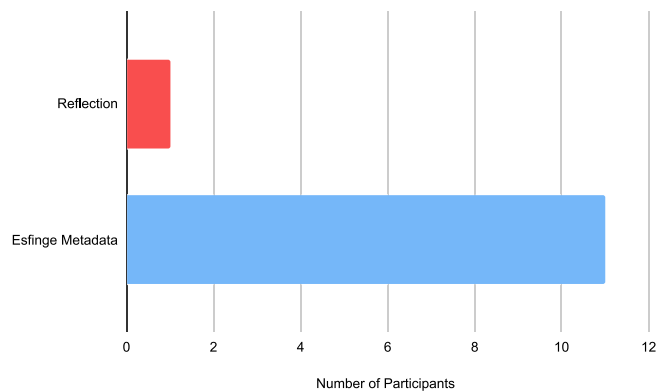


FIGURE 11 | Participants preferred approach.

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 implementation of the code conventions was correct. We excluded from the analysis the participants who could not implement the conventions in time or produced code that did not pass all unit tests. However, we expected some participants to be unable to implement some 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 who failed to implement the code conventions. Only one participant failed to implement the code conventions using the Esfinge Metadata API, but the tasks were successfully implemented using the Reflection API. The total of participants who 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 identified by the six participants. The participant who 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. After resolving his confusion with the code repositories, the participant stated that implementing one of the tasks with Esfinge Metadata API was quick. However, spending so much time fixing the repository 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 accomplishing the task. This participant successfully implemented both tasks with Reflection API but stated that going deeper into the code was necessary. 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 who 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 who 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 unit tests are still failing.

Investigating if the primary programming language or experience with code annotation affected our results, we rely on the data presented in Table 6. The participants that had Java as their primary programming language had equal success and failure rates. The participants who declared another programming language as more familiar had similar results; five participants succeeded, while 4 failed. Since our sample is small, we cannot generalize that the programming language does not affect our conclusions. However, the results show no evidence that it affected positively or negatively.

The participants' experience with code annotations takes place in the correctness analysis. As we can see from Table 6, we had the majority of participants as being familiar with annotations but not experts or were completely new to this language feature. As we can see from our results, the participants had a similar success rate for the intermediary groups, from "not so familiar" to "very familiar." Similar to the discussion over programming

language, we cannot find evidence that the expertise with code annotations influenced our results.

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

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

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 mixed with the framework processing logic, which requires the developer to inspect more source code than implementing tasks 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 min, almost equal to the average time for the Reflection approach, which was 36 min. 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 the Esfinge Metadata API was lower than with the Reflection API, where the former had an average of 29 min and the latter 36 min. The highest value for both approaches was 60 min. However, for the Esfinge Metadata API, this value was considered one outlier since only one participant needed 60 min 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, when using the Reflection API, the participants needed more time navigating the framework and understanding where the metadata had to be processed to implement 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 it was 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. This result regarding the learning curve is consistent with other studies that compared solutions based on metadata [55] with others. Corroborating this, another study shows that it is difficult to debug code with annotations due to its indirection [54], which reveals that adding behavior through annotations configuration might not be intuitive at first. However, this initial difficulty can be reduced once the developer is more used to the API, as shown in this longer experiment, in which the code structure was more consistent throughout a sequence of tasks when using a metadata-based API [19].

We also consider 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 should 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 8 that the Reflection API code conventions had more lines of code. Also, the participants who implemented 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. [19], 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 15, which presented the solution of the convention for the framework annotation `@IsParameterPresent` using Reflection API. This solution was implemented by participant P1 and Listing 16, 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

8.4.1 | Correctness Threat

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. To mitigate some of these errors, we used source code comparing tools to verify the differences between the original source code and the code provided by the students.

8.4.2 | Execution Threat

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. Additionally, the fact that the participants measured the time themselves could have introduced some errors.

8.4.3 | Sample Threat

Another threat to our validation is that we only had undergraduate students, meaning some 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 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 sample size can also affect our conclusions and the result's generalizability. Since we had a small number of participants, we cannot generalize our results for a large population.

However, with this small sample, we managed to have a first impression of the potential of this solution. In this case study, the novel approach boosted the development time and produced code that was easier to maintain and read. To generalize this result, we would need to reproduce this experiment with a larger sample of students. It is important to highlight that this first study was focused on evaluating how this novel approach could be used for convention over configuration implementation, and we believe that this first study produced interesting results.

8.4.4 | Knowledge Threat

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.

8.5 | Implications and Impacts

The proposed approach has an impact on both developers and researchers. First, developers can use this approach to define their conventions. Implementing convention over configuration using our approach produced interesting results since the study participants quickly implemented conventions for the target annotations. Also, developers can implement more conventions for their frameworks since implementing new conventions is easy using this novel approach. This approach can significantly reduce the configurations necessary in metadata-based frameworks. Having fewer configurations, developers may commit fewer mistakes when configuring annotations or even forget to configure a target annotation if conventions are implemented. Finally, we identified that the development community is divided about annotation usage, meaning that this approach can be suitable for developers who dislike annotation usage and can implement their frameworks using conventions instead of annotations.

The framework users may benefit from this feature by creating their conventions in external files. Today, it is not possible for users to configure their metadata conventions. Since framework developers can implement more conventions, metadata-based frameworks may provide more conventions to their final users. With more conventions, the framework users can have a better development experience since they will spend less time configuring metadata by adopting conventions; this will allow framework users to focus more on their framework's logic than on metadata configuration.

Since framework users cannot define their conventions from external files, there is an opportunity for research in this aspect. Our approach does not allow users to implement their

conventions in external files. In the future, we plan to extend our model, including this feature. Another aspect is that convention over configuration was poorly explored in the academic literature. In this case, it would be suitable to investigate the state of the practice and evaluate the current approaches that implement convention over configuration. Such a study may improve developers' knowledge when designing metadata-based frameworks.

Another characteristic of the convention over configuration usage is that conventions are usually not visible to the developers. Although this approach reduces the amount of configurations, it is sometimes not apparent to the developer what is being processed in the background. Using code augmentation to present the developer when a target annotation is added based on a convention. This way, the framework annotation is not presented in the source code, but the developer can still see that a convention replaces it. With this approach, we understand that configurations made by conventions are more transparent for the developer, reducing the time spent trying to comprehend which configurations are being replaced by conventions and improving code comprehension. Based on the difficulties in using conventions reported by practitioners, a visualization approach can be designed as the goal of this task. This visualization approach must help developers determine how conventions are affecting a target element visually and which code elements are affected by the convention. Based on the software visualization approach, we can, for instance, implement a tool prototype integrated with an existing IDE.

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 of 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 convention annotations to the framework annotations, substituting their usage 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.

The proposed approach can impact framework developers, users, and researchers. First, developers using this approach can implement more conventions for their frameworks. Our proposed model allows the easy extension of the existing conventions, and adding it to frameworks can be done with a few lines of code, as presented in Section 5 when we conducted a case study and by the participant's time necessary for implementing conventions using our approach. Although the time spent was very similar, the students' feedback shows evidence that more time was spent reading the documentation than implementing the feature. The users of frameworks might benefit from frameworks with more conventions. Considering that frameworks have more conventions, users can improve their development time. Framework users can spend more time focusing on their application logic than configuring their metadata if frameworks implement more conventions. Finally, this work presents challenges to researchers. First, the studies on convention over configuration can be advanced, which, to this date, has been poorly explored in the academic literature. This novel approach can be extended, allowing conventions to be defined in external files. Practitioners reported that conventions are often concealed in the source code, meaning that visualization tools could help developers understand which conventions are configuring their metadata.

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. Lastly, implementing visualization tools to support developers in understanding new frameworks that implement convention over configuration.

Acknowledgments

We want to thank the support granted by FAPESP (São Paulo Research Foundation, grant 2019/12743-4 and 2021/00071-1). The Article Processing Charge for the publication of this research was funded by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) (ROR identifier: 00x0ma614).

Data Availability Statement

The data that support the findings of this study are available on request from the corresponding author. The data are not publicly available due to privacy or ethical restrictions.

Endnotes

¹ <https://github.com/EsfingeFramework/comparison>

² <https://github.com/Metadata-Experiment>

References

1. E. M. Guerra, J. T. de Souza, and C. T. Fernandes, "A Pattern Language for Metadata-Based Frameworks," in *Proceedings of the 16th Conference on Pattern Languages of Programs* (ACM, 2010), 1–29, <http://doi.acm.org/10.1145/1943226.1943230>.
2. P. Lima, E. Guerra, M. Nardes, A. Mocci, G. Bavota, and M. Lanza, "An Annotation-Based API for Supporting Runtime Code Annotation Reading," in *Proceedings of the 2nd ACM SIGPLAN International Workshop on Meta-Programming Techniques and Reflection* (2017), 6–14.
3. Z. Yu, C. Bai, L. Seinturier, and M. Monperrus, "Characterizing the Usage, Evolution and Impact of Java Annotations in Practice," *IEEE Transactions on Software Engineering* 47, no. 5 (2021): 969–986.
4. R. Teixeira, E. Guerra, P. Lima, P. Meirelles, and F. Kon, "Does It Make Sense to Have Application-Specific Code Conventions as a Complementary Approach to Code Annotations?," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Meta-Programming Techniques and Reflection* (2018), 15–22.
5. Stackoverflow, "Arguments Against Annotations," (2009), <https://stackoverflow.com/questions/1675610/arguments-against-annotations>.
6. Y. Bugayenko, "Java Annotations Are a Big Mistake," (2016), Personal Blog: <https://www.yegor256.com/2016/04/12/java-annotations-are-evil.html>.
7. A. Warski, "The Case Against Annotations," (2017), SoftwareMill Tech Blog: <https://blog.softwaremill.com/the-case-against-annotation-s-4b2fb170ed67>.
8. T. Jungle, "Annotations and Its Benefits in Java," (2018), <http://technicaljungle.com/annotations-in-java-introbenefits-avoidance/>.
9. F. Deissenboeck and M. Pizka, "Concise and Consistent Naming," *Software Quality Journal* 14, no. 3 (2006): 261–282.
10. C. Fernandes, D. Ribeiro, E. Guerra, and E. Nakao, "XML, Annotations and Database: A Comparative Study of Metadata Definition Strategies for Frameworks," in *Proceedings of XATA 2010: XML, Associated Technologies and Applications* (2010), 115–126.
11. N. Chen, "Convention Over Configuration," (2018), <http://softwareengineering.vazexqi.com/files/pattern.html>.
12. E. Guerra, J. de Souza, and C. Fernandes, "Pattern Language for the Internal Structure of Metadata-Based Frameworks," *Transactions on Pattern Languages of Programming III* (Springer, 2013), 55–110.
13. E. Guerra, F. Alves, U. Kulesza, and C. Fernandes, "A Reference Architecture for Organizing the Internal Structure of Metadata-Based Frameworks," *Journal of Systems and Software* 86, no. 5 (2013): 1239–1256.
14. Stackoverflow, "Avoid Loading Associated Relationship Objects," (2012), <https://stackoverflow.com/questions/6403082/avoid-loading-associated-relationship-objects>.
15. Stackoverflow, "Start With Node.js as a Complete Server-Side Newbie?," (2012), <https://stackoverflow.com/questions/7947195/start-with-node-js-as-a-complete-server-side-newbie>.
16. Stackoverflow, "Start With Node.js as a Complete Server-Side Newbie?," (2012), <https://stackoverflow.com/questions/5147814/is-there-a-way-to-rename-activerecord-model-columns>.
17. J. L. de Siqueira, F. F. Silveira, and E. M. Guerra, "An Approach for Code Annotation Validation With Metadata Location Transparency," in *Computational Science and Its Applications–ICCSA 2016: 16th*

- International Conference, Beijing, China, July 4-7, 2016, Proceedings, Part IV 16* (Springer, 2016), 422–438.
18. P. Lima, E. Guerra, M. Nardes, A. Mocci, G. Bavota, and M. Lanza, “An Annotation-Based API for Supporting Runtime Code Annotation Reading,” in *Proceedings of the 2nd ACM SIGPLAN International Workshop on Meta-Programming Techniques and Reflection* (2017), 6–14, <https://doi.org/10.1145/3141517.3141856>.
 19. E. Guerra, P. Lima, J. Choma, et al., “A Metadata Handling API for Framework Development: A Comparative Study,” in *Proceedings of the 34th Brazilian Symposium on Software Engineering* (2020), 499–508.
 20. ECMA, “ECMA—334: C# Language Specification,” (2017), <https://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf>.
 21. JSR, “JSR 175: A Metadata Facility for the Java Programming Language,” (2004), <http://www.jcp.org/en/jsr/detail?id%3D175>.
 22. H. Wada and J. Suzuki, “Modeling Turnpike Frontend System: A Model-Driven Development Framework Leveraging UML Metamodeling and Attribute-Oriented Programming,” *Model Driven Engineering Languages and Systems* (2005): 584–600, <http://www.springerlink.com/index/116636337837142.pdf>.
 23. D. Schwarz, “Peeking Inside the Box: Attribute-Oriented Programming With Java 1.5, Part,” (2004), <http://archive.oreilly.com/pub/a/on-java/2004/06/30/insidebox1.html>.
 24. I. Damyanov and N. Holmes, “Metadata Driven Code Generation Using .Net Framework,” in *Proceedings of the 5th International Conference on Computer Systems and Technologies* (ACM, 2004), 1–6.
 25. M. D. Ernst, “Type Annotations Specification,” (2008), JSR 308: <http://types.cs.washington.edu/jsr308/>.
 26. J. Quinonez, M. Tschantz, and M. Ernst, “Inference of Reference Immutability,” in *ECOOP 2008—Object-Oriented Programming*, (2008): 616–641, <http://www.springerlink.com/index/6M5U5M330T81763T.pdf>.
 27. P. Lombok, “Project Lombok,” (2018), <https://projectlombok.org/>.
 28. I. Córdoba-Sánchez and J. de Lara, “Ann: A Domain-Specific Language for the Effective Design and Validation of Java Annotations,” *Computer Languages, Systems & Structures* 45 (2016): 164–190, <http://www.sciencedirect.com/science/article/pii/S1477842416300318>.
 29. D. Binkley, M. Davis, D. Lawrie, J. I. Maletic, C. Morrell, and B. Sharif, “The Impact of Identifier Style on Effort and Comprehension,” *Empirical Software Engineering* 18 (2013): 219–276.
 30. A. Hindle, M. W. Godfrey, and R. C. Holt, “Reading Beside the Lines: Using Indentation to Rank Revisions by Complexity,” *Science of Computer Programming* 74, no. 7 (2009): 414–429.
 31. B. Weinberger, C. Silverstein, G. Eitzmann, M. Mentovai, and T. Landray, “Google c++ style guide,” (2013), Section: Line Length, http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#Line_Length.
 32. “Gnome Developer Documentation,” <https://developer.gnome.org/documentation/guidelines/programming/coding-style.html#;> Accessed: 2023-01-24.
 33. J. Johnson, S. Lubo, N. Yedla, J. Aponte, and B. Sharif, “An Empirical Study Assessing Source Code Readability in Comprehension,” in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (IEEE, 2019), 513–523.
 34. S. Gunawardena, E. Tempero, and K. Blincoe, “Concerns Identified in Code Review: A Fine-Grained, Faceted Classification,” *Information and Software Technology* 153 (2023): 107054, <https://www.sciencedirect.com/science/article/pii/S0950584922001653>.
 35. L. G. DeMichiel, L. U. Yalçınalp, and S. Krishnan, “Enterprise JavaBeans™ Specification, Version 2.0,” (2001), <http://java.sun.com/products/ejb/docs.html>, year 2001. (Cité page 57.)
 36. J. Miller, “Patterns in Practice—Convention Over Configuration,” *MSDN Magazine* 24 (2009): 39.
 37. S. Ruby, D. Thomas, and D. Hansson, *Agile Web Development With Rails*, 3rd ed. (Pragmatic Bookshelf, 2009).
 38. D. Marx, “Basic Java Persistence API Best Practices,” (2008), <https://www.oracle.com/technical-resources/articles/javaee/marx-jpa.html>.
 39. M. F. C. Nazário, E. Guerra, R. Bonifácio, and G. Pinto, “Detecting and Reporting Object-Relational Mapping Problems: An Industrial Report,” in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (2019), 1–6.
 40. E. Guerra, M. Cardoso, J. Silva, and C. Fernandes, “Idioms for Code Annotations in the Java Language,” in *Proceedings of the 8th Latin American Conference on Pattern Languages of Programs* (2010), 1–14.
 41. J. R. C. Perillo, E. M. Guerra, and C. T. Fernandes, “Daileon: A Tool for Enabling Domain Annotations,” in *Proceedings of the Workshop on AOP and Meta-Data for Software Evolution* (2009), <https://doi.org/10.1145/1562860.1562867>.
 42. S. Duncan, “The Thoughtworks® Anthology, Essays on Software Technology and Innovation,” *Software Quality Professional* 11, no. 2 (2009): 53, <https://www.proquest.com/scholarly-journals/thoughtworks-anthology-essays-on-software/docview/214067173/se-2>.
 43. E. Guerra, C. Fernandes, and F. F. Silveira, “Architectural Patterns for Metadata-Based Frameworks Usage,” in *Proceedings of the 17th Conference on Pattern Languages of Programs* (2010), 1–25.
 44. Hibernate, “Hibernate Validator 8.0,” (2022), <https://hibernate.org/validator/>.
 45. J. S. Pages, “Servlets and JSP Pages,” (2022), <https://www.oracle.com/technical-resources/articles/java/servlets-jsp.html>.
 46. S. Rcp, “The Spring Rich Client Project,” (2010), <http://www.springframework.org/spring-rcp>.
 47. G. Framework, “Genesis Framework,” (2010), <https://genesis.dev.java.net/>.
 48. L. Cavalcanti, *Vraptor: Desenvolvimento ágil para web com java*, (Editora Casa do Código, 2014).
 49. SwingBean, “Swingbean: aplicações swing a jato!,” (2009), <http://swingbean.sourceforge.net/>.
 50. H. Y. Yang, E. Tempero, and H. Melton, “An Empirical Study Into Use of Dependency Injection in Java,” 19th Australian Conference on Software Engineering (ASWEC 2008), 239–247.
 51. E. Guerra, “Design Patterns for Annotation-Based APIs,” *Proceedings of the 11th Latin-American Conference on Pattern Languages of Programming, SugarLoafPLOP 16* (2016): 9.
 52. E. Gamma, R. Helm, R. Johnson, J. Vlissides, and D. Patterns, “Elements of Reusable Object-Oriented Software,” *Design Patterns* 1 (1995): 417.
 53. P. Lima, E. Guerra, P. Meirelles, L. Kanashiro, H. Silva, and F. Silveira, “A Metrics Suite for Code Annotation Assessment,” *Journal of Systems and Software* 137 (2018): 163–183, <http://www.sciencedirect.com/science/article/pii/S016412121730273X>.
 54. E. Guerra and C. Fernandes, “A Qualitative and Quantitative Analysis on Metadata-Based Frameworks Usage,” in *International Conference on Computational Science and its applications* (Springer, 2013), 375–390.
 55. E. Guerra, A. D. O. Dias, L. G. D. O. Vêras, A. Aguiar, J. Choma, and T. S. Da Silva, “A Model to Enable the Reuse of Metadata-Based Frameworks in Adaptive Object Model Architectures,” *IEEE Access* 9 (2021): 85124–85143.
 56. P. Lima, J. Melegati, E. Gomes, N. S. Pereira, E. Guerra, and P. Meirelles, “CADV: A Software Visualization Approach for Code Annotations

Distribution,” *Information and Software Technology* 154 (2022): 107089, <https://www.sciencedirect.com/science/article/pii/S0950584922001987>.

57. E. Guerra, E. Gomes, J. Ferreira, et al., “How Does Annotations Affect Java Code Readability?,” (2022).

58. E. Guerra, C. Fernandes, and F. F. Silveira, “Architectural Patterns for Metadata-Based Frameworks Usage,” in *Proceedings of the 17th Conference on Pattern Languages of Programs* (2010), 1–25, <https://doi.org/10.1145/2493288.2493292>.

59. E. Guerra, F. Silveira, and C. Fernandes, “ClassMock: A Testing Tool for Reflective Classes Which Consume Code Annotations,” in *Proceedings of the Brazilian Workshop for Agile Methods (WBMA 2010)* (2010), 1–14.

60. S. Vegas, C. Apa, and N. Juristo, “Crossover Designs in Software Engineering Experiments: Benefits and Perils,” *IEEE Transactions on Software Engineering* 42, no. 2 (2015): 120–135.

61. I. Salman, A. T. Misirli, and N. Juristo, “Are Students Representatives of Professionals in Software Engineering Experiments?,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. (IEEE, 2015), 666–676.

Supporting Information

Additional supporting information can be found online in the Supporting Information section.