An Annotation-Based API for Supporting Runtime Code Annotation Reading

Phyllipe Lima
National Institute for Space Research - INPE
São José dos Campos, São Paulo
Brazil
phyllipe_slf@yahoo.com.br

Eduardo Guerra
National Institute for Space Research - INPE
São José dos Campos, São Paulo
Brazil
eduardo.guerra@inpe.br

Marco Nardes
National Institute for Space Research - INPE
São José dos Campos, São Paulo
Brazil
marconardes@gmail.com

Andrea Mocci
REVEAL - Faculty of Informatics - University of Lugano
Switzerland
andrea.mocci@usi.ch

Gabriele Bavota
REVEAL - Faculty of Informatics - University of Lugano
Switzerland
gabriele.bavota@usi.ch

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

Abstract

Code annotations are the core of the main APIs and frameworks for enterprise development, and are widely used on several applications. However, despite these APIs and frameworks made advanced uses of annotations, the language API for annotation reading is far from their needs. In particular, annotation reading is still a relatively complex task, that can consume a lot of development time and that can couple the framework internal structure to its annotations. This paper proposes an annotation-based API to retrieve metadata from code annotations and populate an instance with meta-information ready to be used by the framework. The proposed API is based on best practices and approaches for metadata definition documented on patterns, and has been implemented by a framework named Esfinge Metadata. We evaluated the approach by refactoring an existing framework to use it through Esfinge Metadata. The original and the refactored versions are compared using several code assessment techniques, such as software metrics, and bad smells detection, followed by a qualitative analysis based on source code inspection. As a result, the case study revealed that the usage of the proposed API can reduce the coupling between the metadata reading code and the annotations.

CCS Concepts • **Software and its engineering** → *Object oriented development*;

Keywords metadata, code annotation, framework development

Meta'17, October 22, 2017, Vancouver, Canada

ACM ISBN 978-1-4503-5523-0/17/10…$15.00
https://doi.org/10.1145/3141517.3141856

ACM Reference Format:
Phyllipe Lima, Eduardo Guerra, Marco Nardes, Andrea Mocci, Gabriele Bavota, and Michele Lanza. 2017. An Annotation-Based API for Supporting Runtime Code Annotation Reading. In *Proceedings of ACM SIGPLAN International Workshop on Meta-Programming Techniques and Reflection (Meta'17).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3141517.3141856

1 Introduction

Enterprise applications usually execute in middlewares that gather information from the software components, providing the services they need. In the past, most of the information needed by the application server was defined in external descriptors. This information was mostly metadata about the application classes and methods. For large applications, those descriptors became artifacts really hard to maintain. An example of such kind of platform was J2EE with EJB 2.1 [15]. Later, code annotations were introduced as a native feature in the Java language [16] as a solution for metadata configuration. Nowadays, several official Java APIs [17–19] and popular frameworks [1, 20], not only in enterprise applications domain, have code annotations at their core.

Developers are used to annotate their own code using annotations provided by frameworks. However, creating their own solutions based on annotations is much less common. Based on the popularity of annotations-based frameworks, empirical evidences [24, 25, 28] and experimental studies [12], one can say they might be losing an opportunity to use a more suitable solution to some kinds of problems, such as entity mapping, creation of callback methods and configuring a crosscutting concern [13]. The native Java API only provides methods to retrieve annotations directly from a code element, which is far from what is needed by a framework that should retrieve information from an entire annotation schema[1].

[1] A group of related annotations that represents framework domain metadata

The goal of the present work is to propose an API to read annotations at runtime, suitable for the needs of metadata-based frameworks. It is part of a long term effort to improve the design of such kind of software. The initial steps were the documentation of patterns for annotation configuration [7, 10] and for metadata-based frameworks [11]. Aiming to give a better support to implement these practices, we propose in this paper this API with its respective implementation, called Esfinge Metadata[2]. Metadata validation was the first feature implemented by Esfinge Metadata [2], but is out out this paper's scope. This paper focuses on the features for annotation reading and sections 3 and 4 are original contributions.

The usage of the new API was evaluated by refactoring an existing metadata-based framework for class instances comparison. This framework was chosen for using advanced metadata-reading techniques, for instance, that allow an extension in the annotation schema. A comparison between the original version and the refactored version was performed using several techniques, such as object-oriented metrics [21] and bad smell detection[27].

2 Patterns for Code Annotations

The authors have documented in previous work recurrent solutions for metadata-based framework internal structure, focusing on metadata reading and processing[11]. A pattern called Metadata Container is the core pattern of this language, introducing a class whose instances represent metadata at runtime, as shown in Fig. 1. The class `Metadata Container` is responsible to store metadata read from the annotations at runtime. The `FrameworkController` asks the repository for the metadata for a given class. If the metadata of that class was not retrieved, it invokes the `MetadataReader`, responsible to get the metadata wherever it is. All peculiarities of the annotation schema and strategies for metadata definition should be handled by the `MetadataReader`, that should return the `MetadataContainer` with the metadata ready to be used by the `FrameworkController`. Since this is a recurrent practice in several frameworks[11], we assume that storing metadata in a regular class when reading a complex annotation schema is a general good practice. Therefore Metadata Container pattern is central for the proposed API.

The same pattern language also has the patterns Delegate Metadata Reader and Metadata Processor that supports the creation of an extensible annotation schema. For each annotation to be processed there should be a respective implementation of the `ReaderDelegate` interface that knows how to read and interpret its information. Each Delegate Metadata Reader should create one or more implementations of `MetadataProcessor` and add them in the `MetadataContainer`. The `MetadataProcessor` is the abstraction that represents the framework behavior associated with

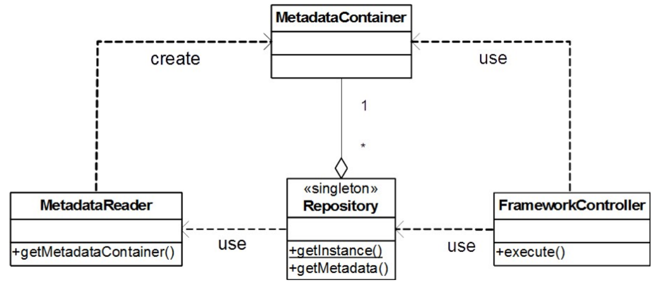

Figure 1. Basic structure of a metadata-based framework using Metadata Container pattern

each annotation. Based on that, the framework should retrieve each processor and invoke them when processing the logic associated with its respective code element.

New annotations should be associated with their respective implementation of a Metadata Reader Delegate. This binding is usually done by a framework annotation that configures the processor class in the custom annotation. Based on that, the framework should search in all annotations of a given element, for the ones that are annotated with its binding annotation. For each one found, it should instantiate the configured class, invoke it to interpret the metadata from that annotation and add its respective processors to the metadata container.

Another set of language-dependent patterns, called idioms, documented a set of practices to represent metadata as annotations [10]. In particular, there are two idioms that focus on a more efficient metadata definition: General Configuration and Annotation Mapping. Using a General Configuration it is possible to define an annotation in a more general context, such as a class, applying the metadata definition to elements contained by it, such as its methods. Annotation Mapping allows the creation of a new annotation to represent a group of annotations, which can be used to create domain annotations [4, 23].

Finally, another set of patterns documented fundamental practices for annotation-based APIs [7] that are used by several frameworks. For instance, pattern Class Stamp documents the practice of adding an annotation to a class to differentiate its processing from the others, and pattern Metadata Parametrization introduces attributes in annotations to allow more granular and specific definitions.

3 Annotation-Based API to Consume Code Annotations

This section presents the new general purpose API to read code annotations. The usage of the proposed API is not limited to frameworks that already use the patterns, but it is designed to guide the developers towards the best practices. It is similar to MVC web frameworks that direct the development to a good separation of concerns. It has a **Simple**

[2]github.com/EsfingeFramework/metadata

API for individual metadata retrieval, and a **Mapping API** to retrieve metadata into a Metadata Container [11], following the pattern presented in section 2. Also, the annotation schema from the API itself can be extended. If the framework needs to retrieve a metadata from the class that is not supported by the native API annotations, new metadata reading APIs might be defined.

The **Simple API** has methods that are equivalent to the Java current API, which retrieves annotations from single elements. The difference is that it takes in consideration configurations that can indicate that the target annotation might be defined in other elements. These configurations are detailed in subsection 3.1. The **Mapping API** retrieves information from an annotation schema and populates a metadata container instance with them. The class `AnnotationReader` has a method called `readingAnnotationsTo()` that receives as parameters the class with the annotations that should be read and the class that represents the metadata container. Fig. 2 presents examples from both approaches.

```
//Simple API usage
List<Annotation> annotList = AnnotationFinder.findAnnotation
        (codeElement, MyAnnotation.class);
//Mapping API usage
AnnotationReader annotationReader = new AnnotationReader();
MetadataContainer container = annotationReader
  .readingAnnotationsTo(AnnotatedClass.class,
   MetadataContainer.class);
```

Figure 2. Simple and Mapping API usage.

To use the mapping API the metadata container class should contain annotations that maps each of its attributes to meta-information that should be retrieved from the target class. The subsections 3.2, 3.3 and 3.4 detail the mapping annotations.

3.1 Metadata Search

There are several patterns where annotations can be defined outside the target element, like on the enclosing code element or inside other annotations. Because of that, the API provides a way to configure an annotation about the places where it can be defined. The following are the annotations provided by the API that can be added in the annotation definition to enable its search in other places:

- `@SearchInsideAnnotations` - This annotation configures when an annotation can be defined inside another one. When an annotation with this configuration is searched, the API implementation should look for it inside each of the target element annotations;
- `@SearchOnEnclosingElements` - This annotation configures when an annotation can be defined in the scope of its enclosing element. When an annotation with this configuration is searched in a method, if not found, the API implementation should verify if it is present in its class;
- `@SearchOnAbstractions` - This annotation configures when an annotation can be defined in abstractions, such as superclasses and interfaces[3]. When an annotation with this configuration is searched in types, the API implementation should verify its superclasses and interfaces. When it is searched on a method, it should search on a method that it overrides from the superclass or that it implements in interfaces.

Those configurations can be freely combined. For instance, for an annotation with `@SearchOnAbstractions` and `@SearchInsideAnnotations`, it can be defined in an abstraction and inside other annotations. That means that for a given class, this annotation could be found inside another annotation which is defined in its superclass.

The API defines an extension point that allows the introduction of new strategies to locate metadata. That could be used, for instance, to define code conventions that can be an alternative to annotations for metadata representation. In order to define a new metadata search annotation, it should receive the annotation `@Locator` with the class that implements the interface `MetadataLocator`.

3.2 Mapping Simple Metadata

The class that represents a metadata container should have annotations to map the class metadata to its attributes. This subsection presents the fundamental annotations of the API, that maps information that can be directly retrieved from the target element. It is important to highlight that all annotations for metadata search presented in subsection 3.1 are considered for the mapping.

The initial configuration that a metadata container needs to have is the `@ContainerFor` annotation. It defines the kind of code element that this container is for. The allowed values are TYPE (for classes, interfaces, enums and annotations), METHOD, FIELD or ALL (if all kinds of elements are allowed). Fig. 3 presents the example of a class with a simple mapping.

```
@ContainerFor(ContainerTarget.TYPE)
public class ContainerClass {
   @ElementName private String elementName;
   @ContainsAnnotation(Element.class) private boolean element;
   @ReflectionReference private Class<?> clazz;
   @AnnotationProperty(annotation = Table.class,
              property ="value")
   private String tableName;
}
```

Figure 3. Example of an annotated metadata container.

The attributes can receive annotations that maps them to information on the target class metadata. They can refer

[3]The annotation `@Inherited` from the regular Java API configures that an annotation defined in a class is inherited by its subclasses. However, it does not work on methods and for implemented interfaces.

to intrinsic reflective metadata, such as element name, or to custom metadata on annotations. Table 1 presents the descriptions of the annotations used on the code example in Fig. 3.

Table 1. Simple API Annotations

Annotation	Description
`@ElementName`	Receives the name of the target code element.
`@ContainsAnnotation`	Maps to a boolean value that states if the annotation is present or not.
`@ReflectionReference`	Receives the instance from the Reflection API that represents the target code element.
`@AnnotationProperty`	Receives the value of an annotation property.

3.3 Cascade Method and Attribute Mapping

When the metadata of a class is read, usually it is necessary to retrieve metadata from its methods or attributes. To support this requirement, the API has annotations that allows the mapping of methods or fields metadata to collections of their respective metadata containers. In other words, a metadata container to represent class metadata can have a collection of metadata containers to represent its methods metadata.

Fig. 4 presents an example of how each method is mapped to its own container. The annotation `@ProcessMethods` can annotate an attribute whose type is a collection of containers for methods metadata. This class can contain mapping annotations from the API to retrieve metadata from each method.

```
//class metadata container
@ContainerFor(ContainerTarget.TYPE)
public class ContainerClass {
   @ProcessMethods
   private List<ContainerMethod> methodContainers;
}
//method metadata container
@ContainerFor(ContainerTarget.METHOD)
public class ContainerMethod {
   @AnnotationProperty(annotation = Column.class,
               property ="value")
   private String columnName;
}
```

Figure 4. Mapping a list of method containters.

This mapping allows a metadata container that represents a class to have collections of containers from its internal elements, such as methods and fields. Other annotations from the API enable criteria that can include only or exclude from the list methods with a certain annotation.

3.4 Mapping Metadata Processors

The support for an extensible annotation schema is an important feature provided by the API, as it is a functionality that previously needed to be manually implemented by the framework. Each new annotation introduced should be associated with a class responsible for its interpretation. What should be added in the metadata container is the metadata processor that results from the execution of the annotation reading method for the custom annotation.

Fig. 5 presents an example of a mapping of metadata processors. The attribute from the `@CustomReader` annotation receives the class from the annotation used to configure the delegate metadata reader. In another words, in the given example, the API implementation should search in the target element for annotations that are annotated with `@ReaderConfigAnnotation`. The presence of this annotation is used to configure that it is a custom annotation from the framework.

```
@ContainerFor(ContainerTarget.TYPE)
public class ContainerClass {
   @CustomReader(value=ReaderConfigAnnotation.class,
      type=ProcessorType.READER_IS_PROCESSOR)
   private List<ProcessorInterface> processors;
}
```

Figure 5. Mapping a list of metadata processors.

Despite the `@CustomReader` annotation that get processors from the target element, other similar annotations named `@MethodProcessors` and `@FieldProcessors` retrieves the processors respectively from each method and field of a class. These other annotations are mapped to attributes from the type `Map` whose key is respectively from the class `Method` and `Field`.

Fig. 6 presents the example of an annotation that the framework can define to configure new annotations. This is the annotation that should be the one referenced in `@CustomReader`. This annotation should receive as the `value` attribute, a class that implements the processor interface. In order to extend the framework adding a new annotation, it should receive this annotation with its respective processor.

```
//Custom annotation definition
@Target(ElementType.ANNOTATION_TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface ReaderConfigAnnotation {
   Class<? extends ProcessorInterface> value();
}f

//Annotation processor interface definition
public interface ReaderInterface {
   @ExecuteProcessor
   public void readAnnotation(Annotation ann,
              AnnotatedElement ael);
}
```

Figure 6. Defining a framework annotation with its respective processor.

Fig. 6 also presents the last piece to create the extensible metadata reading mechanism based on the API, the interface for reading the annotation. This interface should be implemented by the classes that receive the custom annotations to retrieve the data necessary to the annotation processor. This interface should have a method with the annotation `@ExecuteProcessor`.

The API supports 3 different approaches to implement the processors. Those possibilities were based on implementations of existing frameworks. The desired approach is defined in the `type` attribute of the `@CustomReader` annotation. The following are the available options:

- `READER_IS_PROCESSOR` - By using this approach, the class that reads the annotation also have methods for processing it. The API implementation should add in the container the instance itself that reads the annotation;
- `READER_RETURNS_PROCESSOR` - By using this approach, the class that reads the annotation should return the processor in the method with the `@InitProcessor` annotation. The API implementation should add in the container the instance returned by this method;
- `READER_ADDS_METADATA` - By using this approach, a processor is not really required, since the reader receives the metadata container instance and is responsible to add the metadata on it. The API implementation should just pass the container as a parameter to the method with the `@InitProcessor` annotation.

4 Esfinge Metadata - API Implementation

The Esfinge project[4] is an initiative to create innovative open-source metadata-based frameworks. It is defined as an organization on GitHub[5] which contains the source code for all of its projects. Examples of the initiative frameworks are Esfinge Guardian [26], for flexible access control, and Esfinge AOM Role Mapper [8], for adaptive object models implementation. Esfinge Comparison, which is used for the case study described in section 5, is also part of this initiative. It is important to state that despite they are part of the same project, they are all independent softwares.

Esfinge Metadata is a framework that aims to provide functionality to facilitate metadata reading, specially defined by annotations. Considering the whole context of Esfinge project, the idea is that Metadata becomes a meta-framework used by other projects as well as by external frameworks developers. It implements the API described in section 3. The goal of this section is to present some of its implementation details.

[4]http://esfinge.sourceforge.net/ - in portuguese
[5]https://github.com/EsfingeFramework

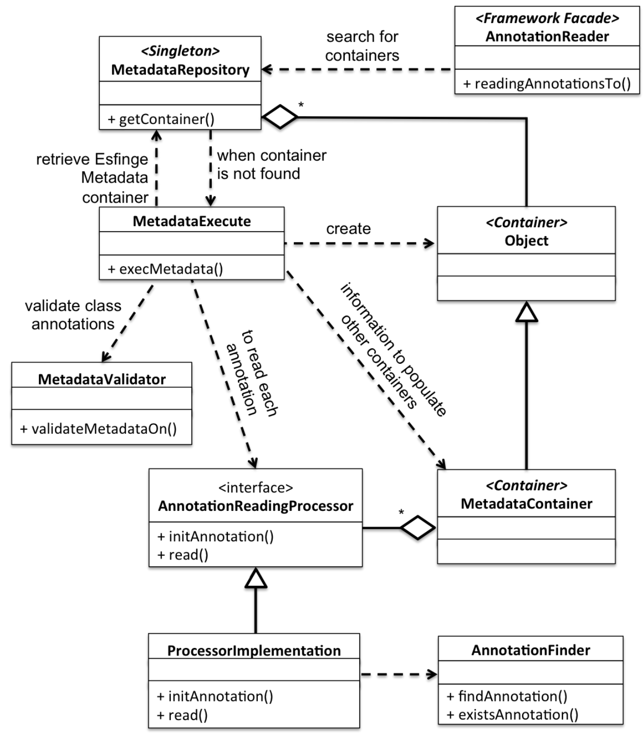

Figure 7. Basic structure of a metadata-based framework using Metadata Container pattern

One of the most important features of Esfinge Metadata is the search for metadata in different places as defined for the API in Section 3.1. The framework implemented the pattern Chain of Responsibility [6] where each node is responsible to search the annotation following a given approach. When it is necessary to search for an annotation, a locator's chain is built for it. This chain is based on the locator's annotations that the target annotations have. The class `RegularLocator` that search normally for the annotation in the current element is always present in the chain as it last element.

Fig. 7 presents a class diagram with some of the main classes of the framework. It can be used as a reference for the following explanations. The main facade from the framework is the class `AnnotationReader` that is responsible to return a container with class metadata. It uses the class `MetadataRepository` to search for the container. When the container is not found in the repository, it uses the class `MetadataExecute` to create it. The repository has a map that stores containers from Esfinge Metadata itself and from other frameworks.

`MetadataExecute` is the class that orchestrates the process for metadata reading. Its responsibility is to create the target metadata container and populate it with the target class metadata. Its first step is to invoke the class `Metadata Validator` to verify if the annotations are used in the class according to the defined constraints. It is important to highlight that Esfinge Metadata uses itself to read its own annotations and create its own container. In other words, the same classes that use Esfinge Metadata to create and store the metadata containers from other frameworks are used to do the same for the framework itself. Due to this fact, `MetadataExecute` invokes `MetadataRepository` to retrieve the metadata reading container.

The class `MetadataContainer` is the Esfinge Metadata container and is composed by instances of `AnnotationReadingProcessor`. All Esfinge annotations that map attributes to metadata are associated to a processor. Each processor implementation uses the class `AnnotationFinder`, which uses the metadata locator chain to retrieve the annotations.

Despite the usage of Esfinge Metadata in the case study presented in section 5, it was also applied in the development of a gamification framework [14]. It was also used internally to read the Esfinge Metadata annotations, in other words, it uses itself to read its own annotations.

5 Case Study - Refactoring an Existing Framework

This section discusses the case study used to validate the metadata reading API, implemented by Esfinge Metadata. The framework used for the case study was the Esfinge Comparison[6]. The initial version of the framework was released on 2012 on Sourceforge platform[7], and it is already used in some real applications currently in production.

The Esfinge Comparison framework compares two instances of the same class and returns a list with the differences between them. It is heavily based on code annotations, since it allows applications using the framework to configure the comparison algorithm for each class property. For instance, the framework provides annotations to configure numeric tolerance or a property to be ignored.

Esfinge Comparison was chosen for this case study as it is using patterns for the internal structure of metadata-based frameworks [11]. It has features to enable the annotation schema extension, which is specially interesting to evaluate if the proposed solution supports this appropriately. Its internal design was evaluated as good in a previous study [9] that focused on the reference architecture it is based on. These facts support the claim that comparing the refactored version of Esfinge Comparison with its previous version can provide a case study in which it is possible to focus on the impact generated by the usage of the proposed API, without the interference of a poor previous design.

The goal of this evaluation is to compare two metadata-based frameworks in which the unique difference is the usage of the proposed API. Despite the case study is a refactoring of an existing software, the proposed API also aims to be introduced in the beginning of the framework development.

5.1 Study Design

The case study focuses on answering the following research question: ***How does the refactoring performed with the proposed API impact the internal structure of the target framework?***

After the refactoring process, the automated unit tests were executed to guarantee that Esfinge Comparison maintained its expected behavior. Different techniques were used to assess distinct characteristics from the source code of both versions.

The following are the approaches that were used in the study: (a) Object-oriented (OO) and annotation metrics: Extraction of metrics from the source code, including size, complexity and coupling metrics; (b) Bad smell detection: Analyze if bad smell instances appeared or were removed from the source code.

Since both versions of the framework implements the same functionalities, we can perform a direct comparison between them. This allows, for example, to analyze whether the code complexity increased or decreased in the refactored system, without the need for defining thresholds indicating high/low complexity levels.

5.2 Framework Refactoring

The refactoring focused on changing the annotation reading class to use EsfingeMetadata for annotation reading. Both the original[8] and the refactored[9] versions are available in GitHub. Table 2 presents a summary of the changes in the refactoring. These changes were extracted from the GitHub website using the feature to compare versions. Changes in configuration and project files were excluded.

Table 2. Changes for refactoring

Changes	Added	Removed	Changed
Classes	4	3	13
Lines of code	317	214	-

The test coverage of the original version measured with Intellij IDE[10] was 83%, and for the refactored version was 80%. A code inspection was performed in the parts of the code not covered by the tests. It was found that most of them involved default constructors and access methods.

The greatest difficulty in the mapping of the metadata container happened when it was necessary to retrieve an information from the field type that was not supported by the

[6] http://esfinge.sourceforge.net/Comparasion.html - available in portuguese.
[7] https://sourceforge.net/projects/esfinge/files/Esfinge2/Comparison%201.0/
[8] https://github.com/EsfingeFramework/comparison/releases/tag/1.1.0-SNAPSHOT
[9] https://github.com/EsfingeFramework/comparison/releases/tag/1.2.0-SNAPSHOT
[10] https://www.jetbrains.com/idea/

Table 3. Metrics changes

Metrics	{Original}	{Refactored}
LOC	703	729
CYCLO	189	182
NOM	104	106
NOC	35	36
NOP	8	9
CALL	295	266
FOUT	36	30

framework. For list comparison, it was necessary to retrieve the generic type from the collection, and the framework did not provide an annotation to map this. The feature for metadata extension was used to create a new custom metadata reading annotation called `@Associate`. Following this approach it was possible to map the desired information to the metadata container.

The class responsible for metadata reading was removed from the project, as well as the class `Repository`, which cached the metadata container instances relative to classes in which the annotations were already processed. In the `compare()` method, which orchestrates the framework functionality, a call to the removed classes was replaced to a call to the `AnnotationReader` class from Esfinge Metadata. The framework, based on the annotations added in the metadata container classes, was able to retrieve the meta-information from the classes.

In conclusion, the API was enough to fulfill the metadata reading requirements for the Esfinge Comparison framework. The feature for having an extensible metadata schema was supported by the API standard features. For the retrieve of a generic type parameter that was not implemented by Esfinge Metadata, a new reading annotation was created. That fact presents evidence that the API is flexible enough for metadata reading requirements that were not directly supported by the implementation.

5.3 Object-Oriented

The analysis was performed based on a suite of object oriented (OO) metrics proposed by [21]. The tool Infusion[11] was used to extract the OO metrics. Table 3 presents the modifications.

A first change is an increase on the Lines of Code (LOC) (703 -> 729) and a decrease on the Cyclomatic Complexity (CYCLO) (189 -> 182). It was expected a reduction in LOC value, since some code regarding metadata reading was removed. However, the addition of the API and the need for the framework extension compensated the lines of code removed. The removed lines are primarily concerned with metadata reading, previously done by code relying on the Java Reflection API. The added lines are annotations provided by the Esfinge Metadata and the imports associated to it.

The reduction in CYCLO is explained by the fact that several of the new lines of codes are annotations, which are not considered by this metric. The new classes introduced have small methods and their invocation are orchestrated by the proposed API, which eliminates much of the need for conditional expressions. The complexity removed were in the metadata reading classes, before the refactoring.

Looking at the coupling metrics, both Number of Operation Calls (CALL) (295 -> 266) and Number of Called Classes (FOUT) (36 -> 30) reduced. Annotation reading logic usually is coupled with several other types, specially the ones related to the annotations that are being retrieved. Inspecting the code searching were there was a reduction in coupling, we confirmed that it happened in the metadata reading classes. The great responsible for this decrease were the removed classes.

Despite most of the changes observed in the analysed quality metrics might look minor (e.g., CYCLO going from 189 to 182), it is worth highlighting that the number of classes impacted by the refactoring is quite limited (see Table 2). Thus, we cannot expect strong changes in the value of the considered metrics.

5.4 Bad Smells Elimination

A bad smell detection analysis was carried out to verify if some bad smell was removed or added by the refactoring. It is important to highlight that bad smells are not necessarily code bugs, but rather design flaws that might become troublesome in the long term. The bad smell detection tool used to perform the anaysis was JSpirit [27]. A total of 16 bad smells were found in the original version and 14 in the refactored version.

The refactoring removed two bad smells, one Intensive Coupling and one Dispersed Coupling. This follows what was already observed in the OO metrics, where we found evidence that the coupling has been reduced by the refactoring. This results reinforces what was described in the previous subsection, demonstrating that the cyclomatic complexity is associated to the metadata reading previously performed in the Esfinge Comparison.

5.5 Case Study Conclusions

This section presents an evaluation of the results facing the research question ***How does the refactoring performed with the proposed API impact the internal structure of the target framework?***

The proposed API removed the need for the major part of the metadata reading logic from the original version. Therefore we expected a decrease in the LOC value. However, the lines eliminated in that part were compensated by implementation of a framework extension and to the API annotations

[11] currently discontinued

added in the container class. Also, the API caused a reduction in code coupling. Evidences of this are seen by the decrease of CALL and FOUT metrics, and by the removal of two coupling bad smells.

Metadata reading logic usually needs to be coupled to the annotation types and to the metadata container classes. That fact makes this logic sensible to changes on both. The use of the proposed API exchanged the imperative code that perform the metadata reading for a declarative mapping using annotations.

5.6 Threats to Validity

The main threat to validity of the results come from the fact that the case study was conducted based on a single framework. It shows a possible impact that the usage of the proposed API, but it cannot be generalized that all frameworks that adopt the API would have a similar impact. However, we can affirm that the reduction in the internal dependencies found in the results is possible to be achieved.

To mitigate the impact of using the refactoring of a single framework as the case study, a qualitative analysis based on code inspection was performed. This qualitative analysis aimed to investigate more deeply the causes of the differences between the two versions, understanding in terms of software design what happened to impact the metrics.

6 Related Work

The metadata-based frameworks widely used by industry still use the standard Java reflection API to retrieve annotations. Despite there are several works that explore the use of annotations, we found none that aims to focus on a general purpose API for reading annotations in runtime.

There are some alternative APIs for reflection in Java based on the concept of fluent APIs [5]. An example of such implementation is Mirror [12]. Despite it gives some support for annotation reading, it still retrieve each annotation individually.

There are some implementations that instead of searching annotations based on the target element, search for classes with a given annotation. Scannotation [13] and Extensible Component Scanner [14] are examples of such solutions. Despite these solutions can be considered for general purpose, they focus on searching classes and not on retrieving those classes metadata.

Checker Framework [3, 22] is a framework for compile time annotation processing. It aims to extend the Java's type system using type annotations to enable verifications that can detect bugs and bad practices in source code. Despite it can be used for more general purposes, it is designed to be used by compiler plugins and not by metadata-based frameworks.

7 Conclusion

This paper presented the proposal of a new API for reading code annotations. This API is based on documented patterns and aims to provide a better support for frameworks to adopt and implement such practices. The proposed API is based on metadata mapping, where a class receives annotations that provide information about what meta-information should be retrieved from the target class and attributed to each field.

The proposed API provides innovative features, such as: (a) support to search annotations in other code elements related to the target code element; (b) mapping for class metadata and annotation attributes; (c) chain processing of methods and field metadata; (d) support for implementation of an extensible metadata schema; (e) extension point that allows the creation of new metadata reading annotations. Esfinge Metadata is a framework that implemented the proposed API as a way to show the viability of its implementation.

A case study was conducted by refactoring an existing framework aiming to evaluate the impact in source code of using the proposed API. The most evident conclusion was the reduction in the number of dependencies in components that perform metadata reading. This was confirmed by the reduction in coupling metrics and the elimination of two coupling bad smells. The new version of the refactored framework also has an increase on its number of annotations as expected. However, an expected reduction in the number of lines of code does not happened, mainly because of an extension of the framework that was needed to retrieve a metadata that was not originally supported by it.

As a future work, there are some points in the API that could be improved in a further version. One of them is the support for reading metadata defined by other means, such as external files and code conventions. Another possibility is the mapping between equivalent annotations from different frameworks. Another point to be improved is to investigate the current structure of existing frameworks to search for additional possibilities on mapping for features that the framework currently implements. Another future work will aim to evaluate the usage of the current API by developers. Some points to be investigated are: (a) How is the impact of the API learning curve in development?; (b) Are developers more productive by using the API? To investigate this points a study involving a controlled experiment might be applied or its usage in the development of a new framework could be investigated.

Acknowledgements

This work is supported by CNPq (grant 445562/2014-5) and FAPESP (grant 2014/16236-6)

[12] http://projetos.vidageek.net/mirror/mirror/
[13] http://scannotation.sourceforge.net
[14] http://extcos.sourceforge.net

References

[1] J.A. Cassoli. 2016. *Web Application with Spring Annotation-Driven Configuration: Rapidly Develop Lightweight Java Web Applications Using Spring with Annotations.* CreateSpace Independent Publishing Platform. https://books.google.ch/books?id=QsUdvgAACAAJ

[2] José Lázaro de Siqueira, Fábio Fagundes Silveira, and Eduardo Martins Guerra. 2016. *An Approach for Code Annotation Validation with Metadata Location Transparency.* Springer International Publishing, Cham, 422–438. DOI: http://dx.doi.org/10.1007/978-3-319-42089-9_30

[3] Werner Dietl, Stephanie Dietzel, Michael D Ernst, Kivanç Muşlu, and Todd W Schiller. 2011. Building and using pluggable type-checkers. In *Proceedings of the 33rd International Conference on Software Engineering.* ACM, 681–690.

[4] Erick Doernenburg. 2008. Domain annotations. *The Thought-Works Anthology: Essays on Software Technology and Innovation* (2008).

[5] Steve Freeman and Nat Pryce. 2006. Evolving an embedded domain-specific language in Java. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications.* ACM, 855–865.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[7] Eduardo Guerra. 2016. Design Patterns for Annotation-based APIs. In *Proceedings of the 11th Latin American Conference on Pattern Languages of Programs (SugarLoafPLoP '16).* ACM, New York, NY, USA.

[8] Eduardo Guerra and Ademar Aguiar. 2014. *Support for Refactoring an Application towards an Adaptive Object Model.* Springer International Publishing, Cham, 73–89. DOI: http://dx.doi.org/10.1007/978-3-319-09156-3_6

[9] Eduardo Guerra, Felipe Alves, Uirá Kulesza, and Clovis Fernandes. 2013. A Reference Architecture for Organizing the Internal Structure of Metadata-based Frameworks. *J. Syst. Softw.* 86, 5 (May 2013), 1239–1256. DOI: http://dx.doi.org/10.1016/j.jss.2012.12.024

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

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

[12] Eduardo Guerra and Clovis Fernandes. 2013. *A Qualitative and Quantitative Analysis on Metadata-Based Frameworks Usage.* Springer Berlin Heidelberg, Berlin, Heidelberg, 375–390. DOI: http://dx.doi.org/10.1007/978-3-642-39643-4_28

[13] Eduardo Guerra, Clovis Fernandes, and Fábio Fagundes Silveira. 2010. Architectural Patterns for Metadata-based Frameworks Usage. In *Proceedings of the 17th Conference on Pattern Languages of Programs (PLOP '10).* ACM, New York, NY, USA, Article 4, 25 pages. DOI: http://dx.doi.org/10.1145/2493288.2493292

[14] Eduardo M. Guerra, Gabriel Fornari, Wanderson S. Costa, Sandy M. Porto, Marcos P. L. Candia, and Tiago Silva da Silva. 2017. *An Approach for Modularizing Gamification Concerns.* Springer International Publishing, Cham, 635–651. DOI: http://dx.doi.org/10.1007/978-3-319-62404-4_47

[15] JSR. 2003. JSR 153: Enterprise JavaBeans 2.1. (Aug. 2003). http://www.jcp.org/en/jsr/detail?id=153

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

[17] JSR. 2013. JSR 338: JavaTM Persistence 2.1. (May 2013). http://www.jcp.org/en/jsr/detail?id=338

[18] JSR. 2013. JSR 344: JavaServerTM Faces 2.2. (May 2013). http://www.jcp.org/en/jsr/detail?id=344

[19] JSR. 2017. JSR 365: Contexts and Dependency Injection for JavaTM 2.0. (Jan. 2017). http://www.jcp.org/en/jsr/detail?id=365

[20] Jeff Langr, Andy Hunt, and Dave Thomas. 2015. *Pragmatic Unit Testing in Java 8 with JUnit* (1st ed.). Pragmatic Bookshelf.

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

[22] Matthew M Papi, Mahmood Ali, Telmo Luis Correa Jr, Jeff H Perkins, and Michael D Ernst. 2008. Practical pluggable types for Java. In *Proceedings of the 2008 international symposium on Software testing and analysis.* ACM, 201–212.

[23] José Perillo, Eduardo Guerra, Jefferson Silva, Fábio Silveira, and Clovis Fernandes. 2009. Metadata modularization using domain annotations. In *Workshop On Assessment Of Contemporary Modularization Techniques (ACoM. 09) at OOPSLA*, Vol. 3.

[24] Romain Rouvoy and Philippe Merle. 2006. Leveraging Component-Oriented Programming with Attribute-Oriented Programming. In *11th International ECOOP Workshop on Component-Oriented Programming (WCOP'06).* 10–18.

[25] Romain Rouvoy, Nicolas Pessemier, Renaud Pawlak, and Philippe Merle. 2006. Using Attribute-Oriented Programming to Leverage Fractal-Based Developments. In *5th International ECOOP Workshop on the Fractal Component Model (Fractal'06), Nantes, France.*

[26] Jefferson O. Silva, Eduardo M. Guerra, and Clovis T. Fernandes. 2013. *An Extensible and Decoupled Architectural Model for Authorization Frameworks.* Springer Berlin Heidelberg, Berlin, Heidelberg, 614–628. DOI: http://dx.doi.org/10.1007/978-3-642-39649-6_44

[27] S. Vidal, H. Vazquez, J. A. Diaz-Pace, C. Marcos, A. Garcia, and W. Oizumi. 2015. JSpIRIT: a flexible tool for the analysis of code smells. In *2015 34th International Conference of the Chilean Computer Science Society (SCCC).* 1–6. DOI: http://dx.doi.org/10.1109/SCCC.2015.7416572

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