Design Patterns for Annotation-based APIs

EDUARDO GUERRA, Instituto Nacional de Pesquisas Espaciais

With the introduction of code annotations in popular languages like Java and C#, several frameworks and platforms adopted a metadata-based API (Application Programming Interface). By using this approach, instead of extending classes, implementing interfaces or invoking configuration methods, the API demand its users to add metadata through annotations in their classes. This approach can bring several benefits, such as decoupling between API and application classes and even a gain in productivity. However, this approach uses a different paradigm from traditional object-oriented programming, and existing patterns cannot be used to define such APIs. Thus, the present work aims to document patterns that describe recurrent solutions in the usage of code annotations for the creation of APIs. Based on such patterns, it is expected that API designers will have a broader range of tools to model the interaction between an application and a programming interface.

Categories and Subject Descriptors: D.1.5 [**Programming Techniques**] Object-oriented Programming; D.2.11 [**Software Architectures**] Patterns

General Terms: Application Programming Interfaces

Additional Key Words and Phrases: Metadata, Frameworks, Code Annotations, APIs

ACM Reference Format:

Guerra, E. 2016. Design Patterns for Annotation-based APIs. jn 2, 3, Article 1 (May 2010), 13 pages.

1. INTRODUCTION

Code annotation, sometimes called as attributes or simply annotations, is a language feature that allows the addition of custom metadata into programming elements, such as a class or a method. These annotations can be used, for instance, for source code processing tools, load-time instrumenting and reflective frameworks. The custom metadata defined by annotations provide domain-specific data that can be used to infer information about the annotated programming element.

With the addition of code annotations on leading programming languages on industry, such as C# and Java [JSR175 2003], the APIs of several frameworks and platforms evolved to be based on such language feature. Phyton has a language feature called Decorators, that can be used similarly to code annotations, and is also used on several frameworks and APIs. An instance of the popularization of annotations is the Enterprise Java Beans specification [JSR220 2006] , which is part of the Java EE and migrated from a model where the beans needed to implement interfaces and extend API classes followed by huge XML descriptors, to a design where they are simple classes with annotations. Several other APIs and frameworks followed the same direction [JSR311 2009; JSR303 2009; JSR299 2009].

The design of APIs based on annotations and metadata uses different practices that are not part of the traditional object-oriented design. Because of this change in paradigm it is harder for developers to create such kind of API. Common questions are *"How to use the annotations?"*, and *"When annotations should be used?"*. However,

This work is supported by CNPq (grant 445562/2014-5) and FAPESP (grant 2015/16487-1)
Author's emails: E. Guerra, eduardo.guerra@inpe.br;

several mature annotation-based APIs learned with experience and were refined through time, and common practices used in such APIs could be used as a reference to API developers.

The goal of this paper is to present recurrent practices used to define annotation-based APIs. Each pattern presents an API definition problem and documents a solution for such problem. They will also compare the annotation-based approach with pure object-oriented alternatives.

The target audience for these patterns are developers that intend to define APIs on languages with support to code annotations. It assumes that the developer understands the annotation feature in a programming language. In this paper the following terms are used: **API** refer to the way that an application interact with a component; **API Implementation** is a concrete component that implements the API; **Application** is a software that uses the API; **API creator** is a developer that creates the API; **API User** is a developer that creates source code that uses the API.

This paper is organized as follows: section 2 presents different approaches for creating APIs; section 3 presents common consequences of the usage of an annotation-based API that apply to all the presented patterns; section 4 presents other pattern collections and pattern languages related to the usage of code annotations; and, finally, each of the following sections presents a patterns documented by this work. If you are familiar with code annotations, you might prefer to jump directly to the patterns and read the initial sections afterwards.

2. UNDERSTANDING ANNOTATION-BASED APIS

This section presents different approaches for creating APIs, to illustrate how an annotation-based API works. The example used considers an API for representing application objects in a georeferenced map. The class should also be able to provide callback methods to be invoked when it is selected and displayed. The code examples are in Java.

The first approach presented in Figure 1 is based on inheritance. The application class, in order to be a georeferenced object, should extend the class GeoPoint and implement the abstract methods that are used for callback. Based on polymorphism, any class that extends GeoPoint can be accepted by the API.

```
1  public abstract class GeoPoint{
2
3     private String latitude;
4     private String longitude;
5     public void setLatitude(String l){
6        latitude = l;
7     }
8     public void setLongitude(String l){
9        longitude = l;
10    }
11    public String getLatitude(){
12       return latitude;
13    }
14    public String getLongitude(){
15       return longitude;
16    }
17
18    public abstract void onSelect();
19    public abstract void onDisplay();
20 }
```

Fig. 1. Class to be extended in a inheritance-based API.

The class presented in Figure 2 is an example of how the class GeoPoint is extended to use the API. The class Measurement can add additional information and methods, and need to implement the abstract methods, even when it does not have an associated logic.

```
public class Measurement extends GeoPoint{

   //additional info
   private double measure;
   private String unit;

   public void onSelect(){
      //show measurement value
   }
   public void onDisplay(){
      //do nothing
   }
}
```

Fig. 2. Usage of an inheritance-based API.

The second approach presented in Figure 3 is based on composition. Instead of extending the GeoPoint, the applications should compose it with its classes. Implementations of Listener interfaces might be added to handle the callbacks and the field info can receive any application object with additional data.

```
public class GeoPoint{

   private String latitude;
   private String longitude;

   public Object info;

   public List<Listener> listeners;

   public void onSelect(){
      for(Listener l:listeners){
         l.onSelect(this);
      }
   }

  //getters and setters ommited
}
```

Fig. 3. Class to be used and composed on a composition-based API.

In Figure 4 there is an example of how a class is composed in this kind of API. The application creates classes that has additional information and/or that handles the object events. Then, the GeoPoint class is not extended by an application class, but composed by them.

Instead of using polymorphism, an API may use reflection to invoke methods and locate code elements of interest. To enable this location, this kind of API defines code conventions that should be followed by the application classes. This practice is documented by the pattern Convention over Configuration [Chen 2006]. Figure 5 presents an example of a class with the conventions. The latitude and longitude fields are located by their names and

```
// Definition of application classes
public class MeasurementInfo{
   private double measure;
   private String unit;
}
public class ShowMeasurement implements Listener{
   public void onSelect(GeoPoint g){
      ////show measurement value
   }
}

//Creating a GeoPoint with measurements
GeoPoint g = new GeoPoint();
g.setInfo(new MeasurementInfo(...));
g.addListener(new ShowMeasurement());
```

Fig. 4. Example of a composition of the GeoPoint class.

methods started by "select" are invoked as callbacks for the selection event. As can be seen, this class does not depend on any class of the API.

```
public class Measurement{

   private String latitude;
   private String longitude;

   public double measure;
   public String unit;

   public void selectMeasurement(){
      //show measurement
   }

  //getters and setters ommited
}
```

Fig. 5. Class with conventions of a conventions-based API.

The last approach presented is the annotation-based API. The API implementation still use reflection to identify the elements, but the search for them can be driven by the code annotations. As can be seen on the code example of Figure 6, additional metadata might be added to allow the processing, such as in which layer the object should be displayed, the format used by latitude and longitude, and how many clicks should be done to invoke the callback method.

It is important to highlight that it was not presented a code example that uses the reflective and annotation-based API because it is the simple usage of the presented classes. These classes can be passed as parameters to the API methods and would be understood by them.

The next section highlights the benefits and drawbacks from annotation-based APIs comparing to the other approaches.

```
@Layer("INFO")
public class Measurement{

    @Latitude(UTM)
    private String lat;

    @Longitude(UTM)
    private String long;

    @Value
    public double measure;

    @Unit
    public String unit;

    @OnSelect(clicks=2)
    public void show(){
        //show measurement
    }

   //getters and setters ommited
}
```

Fig. 6. Class following an annotation-based API.

3. COMMON CONSEQUENCES OF ANNOTATION-BASED APIS

There are some general positive and negative consequences related to the usage of code annotations on APIs. Since the patterns present practices to the definition of such typo of APIs, these consequences apply to all of them. In order to avoid the repetition of such consequences on every pattern, the following presents a list of these general consequences:

(+) Coupling. Annotation-based APIs reduce the coupling of the classes that implement them, comparing with alternatives that demand the implementation of interfaces or extension of classes, since the classes only receive annotations in their elements [Guerra and Fernandes 2013].

(+) Declarative Definition. Annotations define configurations related to a class using a declarative approach. That approach can be more intuitive comparing to a programmatic (imperative) definition of such configurations [Guerra and Fernandes 2013].

(+) Productivity. An experiment [Guerra and Fernandes 2013] revealed that the usage of an annotation-based API have the potential to increase the developers productivity in software development.

(-) Performance. Invocation of methods in an annotation-based API are done using reflection and because of that takes more time than a regular polymorphic invocation, used in APIs based on interfaces [Guerra 2014].

(-) Debugging. Since the method with an annotation is invoked indirectly by a component or framework, it is hard to find errors in such definitions [Guerra and Fernandes 2013].

(-) Code Legibility. An overuse of annotations in the same class can make the code hard to be read [Correia et al. 2010].

4. PATTERNS RELATED TO CODE ANNOTATIONS

The author of this paper already worked on other patterns related to the usage of metadata, specially defined in the form of annotations. This section aims to describe briefly such works and make clear what are the differences from that patterns to the ones described in this paper.

A pattern language for the internal structure of metadata-based frameworks [Guerra et al. 2013] was described composed by eight patterns. This pattern language aimed to define recurrent solutions to structure a component or a framework that needs to handle metadata, including metadata reading and processing. These patterns considered several approaches for defining metadata, which included code annotations, but also considered external definitions and code conventions. A reference architecture for metadata-based frameworks was developed based on this pattern language [Guerra et al. 2013]. The difference of the patterns described in this paper is that they focus on the intent of the annotation definition, while the pattern language on the structure of the component that will process it.

Another set of patterns focused on architectural problems that can be solved by using frameworks based on metadata [Guerra et al. 2010]. These patterns focus on recurrent architectural problems that can be solved by a metadata-based solution. Since they focus on architectural problems, they have a broader context than the ones presented in this paper.

Finally, the most related work to this one are the documentation of idioms (language-specific patterns) for code annotations in the Java language [Guerra et al. 2010]. The idioms focused most on efficient ways to represent metadata, while in this paper the patterns are most related to the metadata purpose. Some idioms were related to limitations existing in Java, such as *Vectorial Annotation* that solves the problem of not being allowed more then one annotation of the same type on a single code element. Because of being focused on the purpose of metadata for creating APIs, the patterns in this paper can be considered language-independent.

5. CLASS DIFFERENTIATION

Also known as Class Stamp, Marking Classes

Some APIs need to differentiate classes, allowing the API user to define the kind of the class that he is defining. For instance, an API might want to know if a given class is supposed to me persisted on a database or send to through the network. The API implementation might threat differently classes with different characteristics.

How to allow the API users to differentiate classes with different roles?

APIs usually receive classes from the application to process and sometimes, based on some criteria, only some kind of classes should be accepted. There can also be more than one kind of class that should be processed in different ways by the API implementation.

Other information in the class, such as its name, package or implemented interfaces, might be used as a criteria for differentiating the classes. Despite the API documentation can make that clear, this kind of configuration might not be clear for who reads the source code.

Therefore:

Create an annotation for the API user to add in a class to configure that it plays a given role in the API context.

Using the object's reference passed as a parameter to the API, it is possible to retrieve a representation of its class. From this class it is possible to verify if it has the presence of some annotation. Based on the presence of that annotation, the implementation can drive the behavior to different directions. If more than two kinds of behavior are possible, it can be used more than one annotation or an annotation attribute to differentiate between them.

An alternative to the annotations usage are using a naming convention or a marking interface, which is an interface without methods just to allow its identification. The annotation has the benefit over these solutions of

being a more explicit configuration. Specially a naming convention, might be changed by a developer that are not aware that it can change the behavior of a component that uses that class.

As an example, imagine that modifications in some classes should be logged in a system. In order to identify such classes, there can be defined an annotation that enables the addition of a metadata with this semantic. Figure 7 presents an example of a class with that annotation.

```
1 @RegisterModifications
2 public class ImportantStuff{
3     //class body ommited
4 }
```

Fig. 7. Identifying classes whose modifications should be registered.

* * *

The JPA API [JSR220 2006] uses the annotation *@Entity* to identify which class is a persistent class.

The EJB API [JSR220 2006] uses the annotation *@Remote* on an interface to identify that it should be used by a remote proxy the access the class, and the annotation *@Statefull* to let the application container know that it has some state that should be maintained between calls.

On the Spring framework [Ho and Harrop 2012], annotating a class with *@Configuration* indicates that the class can be used by the as a source of bean definitions by the container.

6. CALLBACK CONFIGURATION

Also known as Annotated Callback Method

Usually APIs receives application classes that allow its behavior extension. At some point, the API implementation invoke methods on that class. It is desirable to reduce the coupling between the application class and the API, however the API should identify which methods should be invoked.

How to identify methods for invocation minimizing the coupling with application classes?

If an application class need to implement an interface or extend a class from an API, it became coupled to that API. Changes in the interface or in the superclass can affect that class. It also prevent the reuse of that class in contexts where the API is not present.

An unknown method can be dynamically invoked using reflection. However to invoke a method, the API need to have information about the method of a given class that should be invoked. If that method does not come from an interface or a superclass, it should have other ways to identify it.

Therefore:

Create an annotation for the API user to add in a method to configure when it should be invoked.

From the object where the method should be invoked, it is possible to have access to its class representation and its respective methods. When an event occur in the API implementation that demands the invocation of a method in the application class, it can search among the methods for the ones that contains a given annotation. Based on the methods found, they are invoked.

This solution creates a lower coupling between the application classes and the API, since it will only have metadata annotating its methods. That makes the application classes less sensible to changes in the API.

Another option is to use a naming convention, such as a fixed name or a predefined prefix, to identify the methods that should be invoked. Comparing to this option, metadata has the advantage of being a more explicit definition. It might not be clear to a developer that refactoring a method name makes it ineligible to be invoked by an API.

Using this solution it is also possible to define more than one method to respond to the same event, or even to do not define anyone. That is a benefit of this pattern considering the use of interfaces, since it is mandatory to implement all its methods. On the one hand, if the class does not need to respond of an event handled by an interface method, it should implement the method and leave it empty. On the other hand, if it needs to handle the same event for different purposes, the implementation need to be on the same method. When this pattern is used this problem does no occur, because more than one method or none can receive the annotation. Event the same method can respond to different events.

As an example, consider that you want to handle button clicks in an application, but you don't want that class to implement any interface. In order to identify the methods that should be called, there can be defined an annotation that marks such methods allowing its easy identification. Figure 8 presents an example of a class with that annotation.

```
1 public class Handler{
2     @ButtonClick
3     public void handleClick(){
4         //method body omitted
5     }
6 }
```

Fig. 8. Adding metadata for callback methods.

* * *

JUnit API [Langr et al. 2015] has annotations that can be used to annotate methods in a test class that should be invoked after or before each test or after or before the entire test suite, namely *@Before*, *@After*, *@BeforeClass* and *@AfterClass*. There can be none or more than one method with such annotations based on the class needs.

JPA API [JSR220 2006] define annotations for callback methods based on the persistence lifecycle, such as *@PostLoad* or *@PrePersist*. A given method can receive more than one callback method annotation and be called in different moments.

A Java EE API called Common Annotations [JSR250 2006], define annotations such as *@PostConstruct* and *@PreDestroy*. Methods with such annotations are invoked for frameworks, like Spring, when it is creating or destroying a Java Bean.

7. METADATA PARAMETRIZATION

Also known as Detailed Metadata, Granular Identification

Sometimes, just the presence of an annotation in a class or in a method for its identification is not enough. For a class, you might need some additional information to know how it should be handled, and for a method, you might need to verify some condition that will determine if it should be invoked or not.

How to configure domain-specific detailed information about a code element?

Annotations might have attributes that can receive additional information. Some programming languages might limit the options to express data in the annotation attributes. In Java language, for instance, the annotation attributes are restricted to primitive types, Strings, instance of *Class*, an Enum, other annotation, or an array of any of these.

These is no limit to the number of annotations that a class can receive. It is possible to have an annotation that adds more information to another. However, a high number of annotations or annotation attributes in the same code element can reduce the code legibility.

Therefore:

Use annotation attributes or other annotations to specify details about how a code element should be processed.

While the API implementation search for the presence of an annotation in a class or in a method, when it find it, it can retrieve additional information from the annotation or from other ones present in the same element. This information can guide the execution introducing conditions or processing parameters.

From the API point of view, these attributes or complimentary annotations let the API user to parametrize the execution of an implementation that it does not have direct contact with. Some idioms to define annotations in Java language [Guerra et al. 2010] might help to define more complex data: **Composite Annotation** uses internal annotations to define more structured data; **Well-formed Expression** uses a String with an expression language to define data; and **Associative Annotation** receives a class as a parameter to configure a logic that should be executed by the API implementation. This last option provides a way to define an implementation to be executed as an extension point in the API.

The introduction of a high number of parameters might lead to a code hard to read and give maintenance. So, it is advisable to use default values to most common options to avoid a high number of mandatory configurations. Some patterns such as **Inferred Metadata** and **General Configuration** [Guerra et al. 2010] can be used to reduce the number of configurations.

To exemplify this pattern, consider the same example of methods to handle button clicks. Imagine that now you want to handle only events where a certain number of clicks were performed. The *@ButtonClick* can now receive a parameter to allow this more granular handling. Figure 9 presents an example where this parametrization is used.

```
1 public class Handler{
2     @ButtonClick(numberOfClicks=2)
3     public void handleTwoClicks(){
4         //method body omitted
5     }
6 }
```

Fig. 9. Using annotation attributes to allow a more granular event handling.

* * *

JUnit API [Langr et al. 2015] defines the *@Test* annotation that defines a method as a test case for the framework. It can receive an attribute *expected* that receives an exception class. When this attribute is configured, the JUnit framework expects that this test method will throw that exception.

JPA API [JSR220 2006] uses the *@Entity* annotation to define that a class should be mapped to a database table. However, this mapping might need to receive some additional parameters, such as when the the table name is different to the class name. In this case, you should define a *@Table* annotation with the database table name. Other parameters, such as for inheritance mapping, can also be added to the class.

JColtrane framework [Nuccitelli et al.] defines annotations to be added in methods that should handle events for XML parsing based on SAX. The annotation *@StartElement* can be added to a method that will be called every time an element starts in the XML. Additional parameters in that annotation can add restrictions for this method to be called, such as the element name or the presence of a given XML attribute.

8. PROXY PROCESSING CONFIGURATION

Also known as Wrapper Configuration, Additional Behavior Metadata

Proxy and Decorator are important patterns that are highly used by APIs and frameworks to aggregate behavior for existing classes and interfaces. However, sometimes, the methods execution should be intercepted and processed in different ways depending on the class and on the method.

How to change the behavior of proxies depending on the intercepted method?

Proxies and Decorators are usually created based on interfaces, which can have several implementations. Sometimes it is desirable to have different behaviors from the same proxy based on the wrapped class.

A dynamic proxy is a class that can intercept calls from any interface or, depending on the implementation, even any class. When this kind of proxy is used, the interception of all methods are handled by the same method from the dynamic proxy. In this scenario, it is even more important to have more information about the method that is being intercepted. For instance, the proxy behavior might apply only to some subset of methods.

The proxy class might receive some parameters in its creation that can guide its behavior. However, the wrapping of a class in a proxy is desirable to be done transparently. The parametrization in the proxy creation might prevent this transparency.

Therefore:

Use annotations on methods and classes to parametrize the behavior of proxies that intercept it.

The proxy should search from annotations in the wrapped class, and on the intercepted methods. Based on the metadata found, it can decide if each method should be intercepted and how the behavior should be executed. The metadata can be read during proxy creation or when each method is intercepted.

This pattern might also be applied to aspect-oriented frameworks. Since aspects can also intercept methods, the capture of metadata from a given method might provide context to customize the aspect behavior based on the intercepted method. This kind of approach is also called metadata-based aspect-oriented framework [Guerra et al. 2013].

From the API point of view, the idea is to create the impression that the presence of the annotation is adding behavior to a method or to class. For instance, the addition of an annotation related to access control, seams that a security restriction is being added to the method behavior. However, it is just an information that is being consumed by a proxy that is transparently wrapping the annotated class.

The debugging is specially hard when the annotation is used to configure proxy behavior. In this case, when it is expected that a behavior executes when a class or a method have an annotation, it is hard to find out the reason if that behavior is not being executed as expected.

To exemplify this pattern, consider a proxy that should block the access to some methods when the user is not logged in. Instead of separating methods with different proxy behavior in different classes, an annotation could be used for a dynamic proxy to identify methods where the behavior should apply. The annotation *@LoggedOnly* presented in Figure 10 is how the API user would define it.

```
1 public class ServiceClass{
2     @LoggedOnly
3     public void serviceMethod(){
4         //method body omitted
5     }
6 }
```

Fig. 10. Differentiating proxy processing with annotation configuration.

* * *

EJB [JSR220 2006] is an API that defines some annotations that can be added on the Enterprise Java Beans methods in order to configure some behaviors. An example are the annotation *@TransactionAttribute*, to configure how transactions should be managed on this method, and *@RolesAllowed*, to configure access control restrictions.

The framework Esfinge Guardian [Guerra et al. 2013] is another example of an API that uses annotations for access control. The annotation schema is extensible and it works with aspects, dynamic proxies and other platform-specific approaches for method interception.

9. METHOD PARAMETER MAPPING

Also known as Variable Parameters, Used-only Parameters

Methods with different requirements, need different information to execute. By using interfaces, sometimes, several parameters need to be passed in order cover all possible needs. Some methods do not use all this data, making them unnecessary parameters. The addition of a new parameter creates a breaking change that affects all classes that implement that method.

How to make flexible the parameters that a method can receive?

Interfaces usually define methods receiving all possible parameters that it might need. However, frequently the methods do not use all these parameters.

When the information need for the method cannot be predicted, an alternative is to have a Map or a more flexible structure as a parameter. However that does not make explicit exactly what the method is receiving.

Addition of parameters into interface methods is a breaking change that demands the modification of code that uses or implement that method. However, this this new information passed to the method might be necessary to enable the addition of a new feature in the API.

Methods invoked by reflection based on a **Callback Configuration** might need to receive parameters. A convention is usually used to define standard parameters for these methods. However, that might lead to a runtime error when the method does not have the expected convention.

Therefore:

Use annotations on parameter methods to identify what information should be passed in each one.

The API should define annotations for each information that can be passed to the method. When a method should be invoked by the API, it reads the annotations in each parameter to prepare the values that should be passed in each one. Parameter types and names might be used to create convention and reduce the number of configurations. Based on the parameter metadata, the method is invoked with the appropriate values.

The introduction of new types of parameters would not affect the existing code, since each method only declares the parameters that it uses. The new methods would be capable to add a parameter of the new type, but it will not have impact on existing methods.

This pattern can also helps in the legibility of methods, since only relevant parameters will be declared. The method can declare only things that it will use, ignoring other data that it might receive.

As an example, consider an API for a class that handle events for user login and logout. The pattern Callback Configuration can be used to configure the method, however it can receive different parameters. Method Parameter Mapping can be used to identify the semantic of each parameter, so who is invoking the method can identify it. The class in Figure 11 has this pattern applied on the method *execute()*.

```
1 public class LogoutRegister{
2    @WhenLogout
3    public void execute(@Login String login, @LoginTime Date d1, @LogoutTime Date d2){
4       //method body omitted
5    }
6 }
```

Fig. 11. Identifying method parameter semantics with annotations.

* * *

The API specification for JAX-WS [JSR224 2006] uses the annotation *@WebParam* to identify web services parameters. Other annotations are used in the class to map its elements to the XML schema.

The framework Spring MVC [Ho and Harrop 2012] defines as part of its API annotations that can be used on controller methods parameters to receive information from the web request. Examples of such annotations are *@RequestParam*, *@RequestHeader*, and *@PathVariable*. This way, the method only receives the information that it needs.

The framework JColtrane [Nuccitelli et al.], that aims to parse XML documents based on SAX, uses parameter annotations to define which information about the XML element the method need to receive. Example of its parameter annotations are *@CurrentBranch*, *@Attribute("name")*, *@AttributeMap*, *@Tag* and *@Body*.

REFERENCES

CHEN, N. 2006. Convention over configuration.

CORREIA, D. A. A., GUERRA, E. M., SILVEIRA, F. F., AND FERNANDES, C. T. 2010. Quality improvement in annotated code. *CLEI Electron. J. 13,* 2.

GUERRA, E. 2014. *Componentes Reutilizáveis em Java com Reflexão e Anotações*. Casa do Código.

GUERRA, E., ALVES, F., KULESZA, U., AND FERNANDES, C. 2013. A reference architecture for organizing the internal structure of metadata-based frameworks. *Journal of Systems and Software 86,* 5, 1239 – 1256.

GUERRA, E., ANDJEFFERSON SILVA, M. C., AND FERNANDES, C. 2010. Idioms for code annotations in the java language. *In Proceedings of the 17th Latin-American Conference on Pattern Languages of Programs, SugarLoafPLoPe*, 1–14.

GUERRA, E., BUARQUE, E., FERNANDES, C., AND SILVEIRA, F. 2013. *A Flexible Model for Crosscutting Metadata-Based Frameworks*. Springer Berlin Heidelberg, Berlin, Heidelberg, 391–407.

GUERRA, E., DE SOUZA, J., AND FERNANDES, C. 2013. *Pattern Language for the Internal Structure of Metadata-Based Frameworks*. Springer Berlin Heidelberg, Berlin, Heidelberg, 55–110.

GUERRA, E. AND FERNANDES, C. 2013. A qualitative and quantitative analysis on metadata-based frameworks usage. *Computational Science and Its Applications âĂŞ ICCSA 2013 Lecture Notes in Computer Science 7972*, 375–390.

GUERRA, E., FERNANDES, C., AND SILVEIRA, F. F. 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, 4:1–4:25.

HO, C. AND HARROP, R. 2012. *Pro Spring 3* 1st Ed. Apress, Berkely, CA, USA.

JSR175. 2003. Jsr 175: a metadata facility for the java programming language.

JSR220. 2006. Jsr 220: Enterprise javabeans 3.0.

JSR224. 2006. Jsr 224: Javatm api for xml-based web services (jax-ws) 2.0.

JSR250. 2006. Jsr 250: Common annotations for the javatm platform.

JSR299. 2009. Jsr 299: Contexts and dependency injection for the javatm ee platform.

JSR303. 2009. Jsr 303: Bean validation.

JSR311. 2009. Jsr 311: Jax-rs: The javatm api for restful web services.

LANGR, J., HUNT, A., AND THOMAS, D. 2015. *Pragmatic Unit Testing in Java 8 with JUnit* 1st Ed. Pragmatic Bookshelf.

NUCCITELLI, R., GUERRA, E., AND FERNANDES, C. Parsing xml documents in java using annotations. In *XML: APLICAĂĞÃŢES E TECNOLOGIAS ASSOCIADAS (XATA 2010)*.

SugarLoafPLoP'16, November 16-18, Buenos Aires, Argentina. ACM 978-1-4503-0107-7