Does It Make Sense to Have Application-Specific Code Conventions as a Complementary Approach to Code Annotations?

Rodrigo Teixeira
National Institute for Space Research (INPE)
São José dos Campos, São Paulo
Brazil
contato@rodrigomalk.com.br

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

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

Paulo Meirelles
Department of Health Informatics
Federal University of São Paulo (UNIFESP)
São Paulo, São Paulo, Brazil
paulo.meirelles@unifesp.br

Fabio Kon
Department of Computer Science
University of São Paulo (USP)
São Paulo, São Paulo, Brazil
fabio.kon@ime.usp.br

Abstract

Code annotations are extensively used by Java developers, especially in enterprise frameworks and APIs such as Spring and Java EE. Different code elements that frequently share some similarities, sometimes, repeat the annotations. The goal of this paper is to evaluate whether a real-world system could use code conventions as a means to avoid replicating code annotations. We report on a study on the software used for the EMBRACE Space Weather program at INPE to search for similarities.

CCS Concepts • **Software and its engineering → Application specific development environments**;

Keywords metadata, code annotation, code convention

ACM Reference Format:
Rodrigo Teixeira, Eduardo Guerra, Phyllipe Lima, Paulo Meirelles, and Fabio Kon. 2018. 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 (META '18), November 5, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3281074.3281078

META '18, November 5, 2018, Boston, MA, USA

ACM ISBN 978-1-4503-6068-5/18/11…$15.00
https://doi.org/10.1145/3281074.3281078

1 Introduction

A framework is an "incomplete application" that provides functionality and extension points that developers use to create their applications with reduced effort. Some frameworks offer these features in the form of metadata configuration, known as metadata-based frameworks [4]. In the Java language, code annotations provide these features. Since the introduction of annotations in Java version 5, they have been extensively used in enterprise frameworks and APIs to configure custom metadata [7]. For instance, the JAXB API [8] uses annotations to map Java Beans to XML documents, and the JPA API [9] uses annotations to perform an object-relational mapping. In a previous study [10], our group showed that out of 25 real-world projects, the one with the least amount of annotations had 10% of the classes annotated and the project with higher annotation usage had 68%. On average, all projects contained 50% of their classes with at least one annotation.

Frameworks based on metadata usually provide an annotation schema, which is a group of related annotations that belongs to their APIs and represent the set of metadata necessary for their usage. Each framework consumes these annotations and executes routines accordingly. If different locations in the application code require the same metadata configuration, the current approach is to repeat the annotations on these code elements. These repetitions cause several annotation duplications throughout the code, which can harm code maintenance and readability. For instance, we found a class with more than 700 annotations in its methods [10], most of them repeated. This clearly contradicts the well-established idea in software development of avoiding code repetitions, popularized with the DRY acronym, meaning "Don't Repeat Yourself". An approach to reduce the

number of annotations can be to identify common characteristics in code elements that receive a single configuration and adopt some code conventions to represent them.

Code conventions are a set of recommendations, usually for a specific programming language, to aid developers in programming style and good practices during the coding process [1]. Conventions cover a wide range of aspects of coding, such as naming, indentation, comments, declarations, and overall structure. A team of developers may also have their code conventions that best suit their needs. Following these recommendations improve software maintenance, readability, and evolution. For instance, the Java Beans standard defines *getters* and *setters* naming convention for methods that read and write class attributes, respectively.

A code convention can also be used by metadata-based frameworks to represent a configuration [1]. These frameworks are based on the "convention over configuration" design paradigm, which states that if developers follow the conventions, they need less specific configurations throughout the code. For instance, JUnit 3 uses the convention that methods with the prefix "test" represented test methods. The usage of conventions can potentially reduce the number of annotations, especially repetitive ones. For instance, the JPA API uses several code conventions, such as using the name of the class and the fields, respectively for the names of tables and columns as a default. However, several frameworks avoid the adoption of code conventions because they might not be suitable for all applications that instantiate it.

This paper defines as a "framework convention" the code conventions defined by the framework that applies to all applications that instantiate it. Additionally, we also define a new concept called "application-specific convention" that is a convention defined by an application to define metadata for a framework that it uses. A framework might provide an extension point to enable the implementation of this approach that allows the definition of new conventions by the application. For frameworks without this kind of support, an application-specific convention might be implemented by manipulating the classes bytecode at compile or load time to introduce the corresponding annotations. However, to the best of our knowledge, we do not know any concrete implementation of this technique.

The focus of this paper is not on the implementation of the support for application-specific conventions, but instead on verifying whether these conventions do exist naturally in real software. In the study presented in this paper we focus on identifying conventions that appear on specific applications. If they were explicit and appeared in every application using a given framework, then it might indicate that the framework itself should implement these conventions as a means to configure metadata.

This paper presents a study that involved a set of Java applications from the Brazilian EMBRACE Space Weather program to investigate if a convention could define similar annotation configurations based on their respective code elements. We searched all annotations used throughout the code to verify how many of them a convention could define. After retrieving information from all annotations, we verified if annotations of the same type are used in code elements that share similarities. The goal is to search for patterns in annotated code and evaluate if application-specific code conventions are a valid complementary approach for metadata configuration.

2 Metadata Configuration

In the context of object-oriented programming, metadata is data about code elements, such as classes, methods, and attributes. For instance, a method's metadata is composed of its name, signature, parameters, and return type. Software applications such as tools and frameworks can consume metadata to execute routines based on code elements structure. It can be used for source code generation [2], compile-time verification [3, 12], class transformation [11], and framework adaptation [4]. However, it may be necessary to configure additional custom metadata to inform how code elements structure a software that consumes metadata should interpret.

2.1 Code Conventions

Some frameworks, such as Ruby on Rails [13] and CakePHP, use code conventions to define custom metadata [1]. These are specific guidelines that developers use when creating a code such as naming pattern, return type, implementation of an interface, and so forth. Some frameworks or tools are capable of identifying these conventions and execute specific behavior.

For instance, when using CakePHP, a controller class must use camel case and end with the word `Controller`. The framework recognizes, using this convention, the class as a controller class without the need of additional configurations. This technique can be very productive in some contexts. However, it has limited expressiveness and cannot be used to define more complex metadata. In the Java language, the Java Beans standard defined *getters* and *setters* naming convention for methods that read and write class attributes, respectively.

Although this choice can be very productive in some contexts, code conventions have limited expressiveness and cannot be used to define more complex metadata. For instance, a code convention could be used to define a method as a test method as in JUnit 3. However, it could not be used to define a valid range of a numeric property as in Bean Validation API. Another drawback is that the metadata are actually implicit in the source code, hidden behind the conventions. As such, an unwary developer might alter a method's name without knowing that this actually is part of a convention, as well as, some frameworks rely on this to execute a specific behavior.

2.2 Code Annotations

The Java programming language allows custom metadata to be configured directly on the source code, through the use of code annotations [6]. This feature became available when the Java version 5 was released and has since been used extensively by developers. It is also possible to define custom metadata via external files, such as XML. However, using annotations has the advantage of not having to provide a path to the code element, since the metadata is configured directly on it. Some authors call this technique as attribute-oriented programming [15], which is defined as a programming technique used to mark software elements with annotations to indicate application-specific or domain-specific semantics [16].

```
1  import javax.persistence.Entity;
2  import javax.persistence.Id;
3
4  @Entity
5  public class Table {
6    @Id
7    private Long id;
8    ...
9    //getters and setters omitted
10 }
```

Figure 1. Example of the usage of Code Annotation

Figure 1 contains an example of a code annotation: The `Table` class is annotated with `@Entity`. The JPA API will consume this annotation and will consider that this class must be mapped into a table in the database, named "Table". Also the `id` member is annotated with `@Id`. Again, the JPA API will consume this and map this member into a column marked as primary key.

We cannot say which of these two approaches is the best overall solution for custom metadata configuration. One needs to analyze the trade-off between them. Using annotations allows more complex metadata to be configured, since through the Java Reflection API they can be easily recovered at runtime. On the other hand, using code conventions requires the framework to implement algorithms capable of reading these conventions, thus limiting the amount of metadata that can be configured.

Using code conventions frees the developers to manually configuring the metadata as it is implicitly configured amid the code. This also can be a problem, since failing to use the convention will not cause any compile-time errors but will break the code as the framework will not be able to correctly read the metadata.

2.3 Application Specific Code Conventions

Consider that all classes in packages ended with `*.domain` should have the same annotation `@Entity`. A definition of an application-specific convention could avoid several annotation configurations, as shown in Figure 2. Since this convention relies on the package name, it cannot be generalized by the framework, because it does not apply to all applications.

```
1  package com.myapp.module.domain;
2
3  import javax.persistence.Entity;
4  import javax.persistence.Id;
5
6  //does not need to configure @Entity
7  //this class is covered by the convention
8  public class Table {
9    @Id
10   private Long id;
11   ...
12   //getters and setters omitted
13 }
```

Figure 2. Example of an Application-Specific Code Annotation

As another example, consider an application using the Bean Validation API [5] that has several fields for defining percentage values that should be between 0 and 100. The annotation `@Range(min=0,max=100)` must be used explicitly on each of these fields. Since they share some similarities, such as being of type Integer and having the `Percentage` suffix, a code convention could be used to avoid repeating this configuration in this application. However, if the convention does not match, it is always possible to use the annotation as before. Figure 3 presents an example of this scenario.

```
1  public class Candidate {
2
3    //validation configured based on convention
4    //no need to add an annotation
5    private Integer votesPercentage;
6
7    //since the convention does not match
8    //the metadata should be explicitly defined
9    @Range(min=0,max=100)
10   private Long rejectionRate;
11 }
```

Figure 3. Application-Specific Code Annotation for Validation Metadata

3 Target Application

To perform our analysis, we used the web applications source code developed for EMBRACE (Brazilian Studies and Monitoring of Space Weather), a program maintained by Brazil's National Institute for Space Research since 2008. Its goal is to become an information and forecasting center for space weather. With specialized sensors and instruments, EMBRACE performs real-time monitoring and generates data publicly available on its website. Most of the information is available in a graphical format that can also be downloaded.

Developed using the Java language, EMBRACE has to cope with a vast amount of data received, process it, and make it available. It is composed of several products that follow the same reference architecture [14] based on Java Enterprise APIs. The source code is divided into 94 components archived in independent deployment units.

We used the EMBRACE program software as a case study since we were investigating application-specific code conventions used by our internal development team, which is related to the software architecture as well as the team guideline. Using multiple applications for this study would span several application-specific code convention, thus making diverting from any concrete conclusion at this stage of the research.

4 Annotation Analysis

This section presents the study performed in the EMBRACE source code to search for patterns in annotation configuration that could be replaced by an application-specific code convention. We do not aim to judge if a convention found should be generalized by the framework or should be application-specific, and we considered any pattern found as a potential convention.

4.1 Research Question

We argue that developing Java software using code conventions specific for applications could reduce the number of annotations repeated throughout the source code. In other words, we might find useful to use a hybrid approach to configure metadata. We can manually insert annotations where complex metadata configuration is required, and in other parts of the code we use conventions. We also consider this convention to be specific to the application since it might not be possible for a metadata-based framework to generalize such conventions for all applications. To guide this study, we propose the following research question:

> **RQ:** Does It Make Sense to have Application-specific Code Conventions as a Complementary Approach to Code Annotations?

To answer this question, we looked into the source code of a set of Java applications from the EMBRACE project, searching for annotations. We then related the information of the code elements that shared similar annotation configuration looking for similarities. Guided by the results, we performed a qualitative analysis trying to find possible code conventions that raised naturally in the applications. For this analysis, we also took into consideration the context and meaning of the annotation type in its API.

4.2 Methodology

This section describes the steps performed to answer the research questions.

Step 1 - Obtaining the dataset
We obtained and accessed to the EMBRACE program Java source code.

Step 2 - Extracting code elements metadata
With the source code available for analysis, we developed a tool, named Annotation Localizer (ALocalizer)[1], which extracts each code element and reports all of its metadata. For instance, for a field code element, the tool reports its name, access modifier, type, class and package it belongs to, as well as every annotation along with its attributes.

Step 3 - Creating a database
With the report available from the ALocalizer, the next step was to create a database to organize the collected annotation data.

Step 4 - Executing and analyzing queries
We used the database to execute queries that relate and group annotations based on their code elements information. For classes, we considered its interfaces, packages, and superclasses; for methods, we considered its return type and name; for fields, we considered its type and name.

Step 5 - Identifying potential code convention usage
We performed a qualitative analysis in the query results considering the annotation type context and meaning inside its API. We searched for scenarios where a code convention could be used to replace a group of annotations.

4.3 Data Extraction

EMBRACE software spans across several web applications, which in turn make up a total of 94 components. Since this is proprietary software, the source code cannot be made publicly available. Table 1 contains additional information about the analyzed source code.

Table 1. Analyzed Source Code Information

Number of Classes	1314
Number of Annotated Classes	837
Total Number of Annotations	5206
Number of Unparsed Classes	5

[1] http://github.com/phillima/alocalizer

To collect annotation information from the source code, we developed an open source tool named Annotation Localizer (ALocalizer). It scans Java projects and outputs an XML file with metadata of code elements in the source code. For instance, for a method code element, the report gives its name, class it belongs to, package, signature, return type, annotations present, and their attributes. We executed the ALocalizer for all 94 components, and the outputted XML files were persisted in a database.

4.4 Data Analysis

We analyzed the source code of 94 software components and organized the dataset in a MySQL database. The study took into consideration the similarities on code elements that shared the same annotation type.

There were 91 annotation types, and several code elements use most of them throughout the projects. Table 2 presents the annotation types that have the most number of occurrences. Despite some annotations from the Java core API, such as @Override and @SupressWarnings, one can find annotations from Java Persistence API (JPA), Enterprise Java Beans (EJB), and Java Server Faces (JSF).

The chart in Figure 4 presents the number of annotation occurrences that have values below 150 occurrences. We organized this information in decreasing order from left to right. It is possible to see that a considerable number of annotation types occur more than 40 times in the analyzed projects.

After capturing this data, we performed queries that combine the annotation types with other element characteristics to identify potential code conventions. The results of such queries were submitted to a qualitative analysis that also considered the annotation type context. For instance, the annotation @Column is the second in the number of occurrences, but it receives attributes that make it hard to be generalized in a convention[2].

After the qualitative analysis, the following characteristics end up being more relevant for the code conventions: implemented interfaces, naming patterns, package names, return types, and field types. Table 3 describes some of the code conventions we identified. For that, it presents the annotation itself, the element type, and the number of occurrences (#). Based on our analysis, we found a total of 908 annotations (21.42% of the total) that could be replaced by 17 conventions. In some cases, we found that the same annotation could be replaced by multiple code conventions, or by a stronger convention that englobed them all.

5 Discussion

With this analysis, we concluded that it is valid to complement annotations with application-specific code conventions as a strategy for metadata configuration. As shown, a significant amount of code elements that share the same annotation type often are added in the same package, implement the same interface, or have naming similarities. Hence, a hybrid approach with annotations and code-conventions might be used to reduce the burden of the development team of manually writing code annotations. This approach is presented not as the solution but as an evidence of the existence of application-specific code convention for metadata configuration.

It is important to highlight that these conventions were naturally present in the source code due to team practices, thus reinforcing that these are application-specific code conventions, and might not apply to other applications or even other development teams. If there were previously defined code conventions, the number could be even higher.

We did not evaluate whether these conventions should belong to the application or generalized directly into the framework. Generally, conventions related to packages and application types are more suitable as application-specific conventions. However, based on standard conventions used by applications, the framework development team might consider adding such convention as a framework convention.

6 Threats to Validity

The tool used to fetch annotations, ALocalizer, had no other equivalent found to perform a comparison. As such, we checked its accuracy manually. We performed this evaluation through a manual comparative analysis, taking into consideration the code annotation and the similarities between them and their target code elements.

The analysis focused on the applications from the EMBRACE program only. Further studies using other projects from other environments should be performed to confirm the generalization of our findings. n

7 Conclusion

This study aimed at identifying patterns in annotated code and evaluating whether it would be a valid approach to use application-specific code conventions as a complementary alternative to code annotations. In other words, the code style adopted in the context of a software project could also be used for metadata configuration. To the best of our knowledge, no work proposing application-specific conventions exists in the literature; therefore this research introduces a novel idea and can be further explored by the software engineering community.

Our analysis showed the existence of specific patterns between annotations and their target code elements. For instance, it is common to keep similarly annotated methods and classes in the same package. Classes implementing the same interface often contain the same annotations, as well

[2]The relation between attribute values and the code elements names were not considered.

Table 2. Top 15 Annotation Types with Most Occurrences

Annotation	Annotation schema	Number of occurrences
@Override	JVM	1575
@Column	JPA	633
@Basic	JPA	348
@SupressWarnings	JVM	227
@Remote	EJB	139
@Entity	JPA	101
@Table	JPA	101
@NamedQueries	JPA	100
@Id	JPA	99
@ManagedBean	EJB	99
@Stateless	EJB	99
@JoinColunm	JPA	97
@NotNull	JVM	97
@PersistenceContext	JPA	94

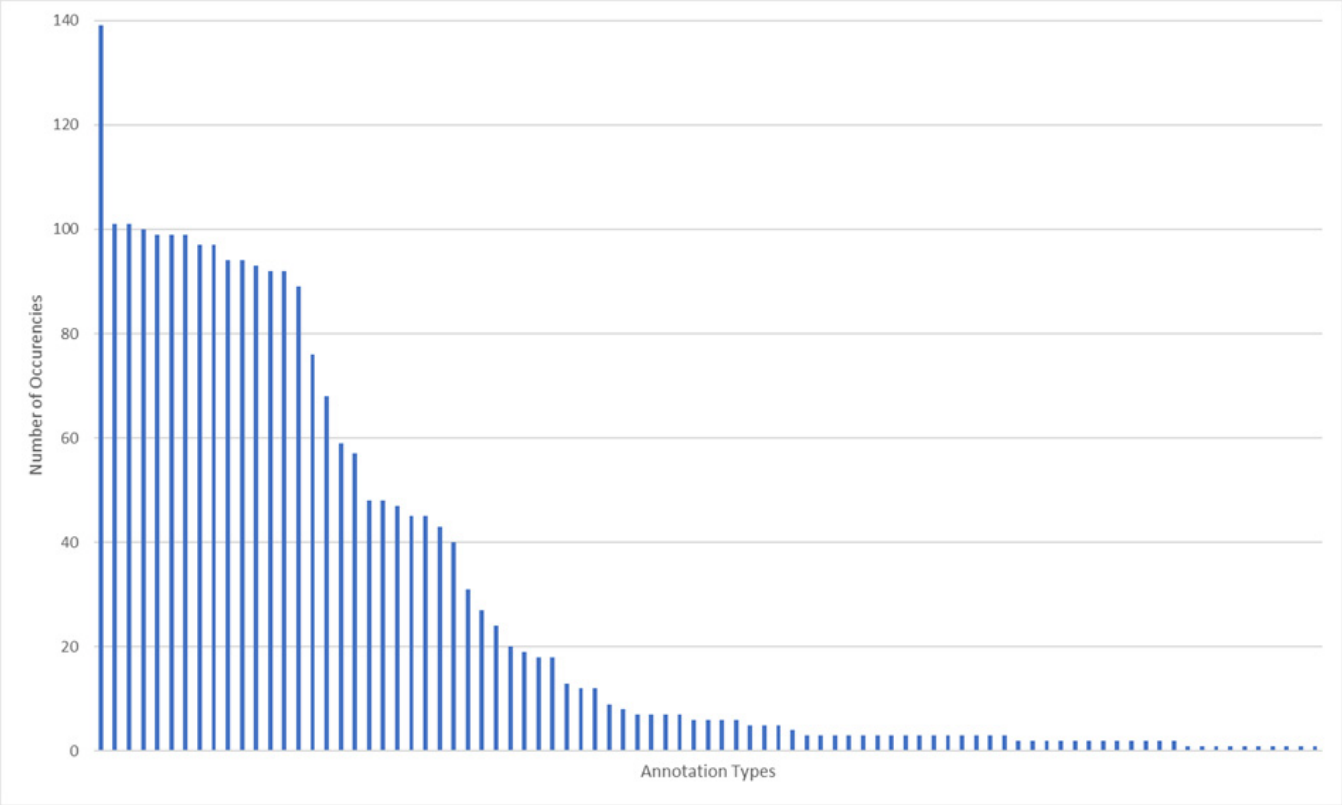

Figure 4. Number of Occurrences per Annotation Type

as inherited class members tend to use the same annotations among child classes.

With these results, we answer our research question, showing that it does make sense to use code conventions for metadata configuration, reducing the manual insertion of code annotations significantly. Additionally, the usage of conventions has the potential of reducing bugs derived from missed annotations, which might be hard to debug, and provides more structure to the code. As a result, this requires writing less source code, keeping it more readable, cleaner, and maintainable.

Table 3. Possible code conventions found

Annotation	Type	#	Convention
@Id	Field	96	Fields where the name is "id"
@GeneratedValue	Field	93	Fields where the name is "id"
@PersistenceContext	Field	89	Fields where the name is "em"
@Temporal	Field	25	Fields where the name contains "*.date.*"
@MessageDriven	Class	7	Classes with the interface MessageListener
@GET	Method	43	Methods with the return type "Response"
@Produces	Method	37	Methods with the return type "Response"
@Type	Field	48	Fields type "DateTime"
@Temporal	Field	31	Fields type "Date"
@Lob	Field	27	Fields type "byte[]"
@EJB	Field	44	Fields type that the name has the "FacadeLocal" suffix
@Entity	Class	101	Classes in packages that the path matches "*.data"
@NamedQuerries	Class	100	Classes in packages that the path matches "*.data"
@SessionScoped	Class	90	Classes in packages that the path matches "*.viewer.*" and where the name contains "*.Controller.*"
@ManagedBean	Class	92	Classes in packages that the path matches "*.viewer.*" and where the name contains "*.Controller.*"
@Remote	Class	96	Classes with an interface that the name has the "Remote" suffix and in a package which the path matches "*.facade" and where the name contains "*.Facade.*"
@Stateless	Class	96	Classes with an interface that the name has the "Remote" suffix and in a package which the path matches "*.facade" and where the name contains "*.Facade.*"

Finally, as future work, we intend to implement tools and frameworks that allow the introduction of application-specific code conventions. As an alternative for existing frameworks and APIs, it is possible to create a tool that processes the source code introducing the annotations at compile or load time. Following this approach, the bytecode would be searched for code elements that match the convention introducing the respective annotation there before being loaded by the application.

Another possible approach for new frameworks is to create an API for consuming code annotations that provide extension points to introduce new code convention definitions. When the framework searches for an annotation, it would also verify if there is an object configured by the application to verify if the code convention exists. When this object is present, it can verify if the target element matches the convention and return a metadata that is equivalent to the annotation configuration. With this alternative, the annotation itself does not need to be introduced in the element and it is an extensible metadata reading mechanism that identifies and interprets the convention.

References

[1] N. Chen. 2006. Convention over configuration. http://softwareengineering.vazexqi.com/files/pattern.html

[2] Ivo Damyanov and Nick Holmes. 2004. Metadata Driven Code Generation Using .NET Framework. In *Proceedings of the 5th international conference on Computer systems and technologies*. ACM, 1–6.

[3] Michael D Ernst. 2008. Type annotations specification (JSR 308).

[4] Eduardo M. Guerra, Jerffeson T. de Souza, and Clovis T. Fernandes. 2010. A Pattern Language for Metadata-based Frameworks. In *Proceedings of the 16th Conference on Pattern Languages of Programs (PLoP '09)*. ACM, New York, NY, USA, Article 3, 29 pages. https://doi.org/10.1145/1943226.1943230

[5] Red Hat. 2009. JSR 303: Bean Validation. http://beanvalidation.org/1.0/spec/

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

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

[8] JSR. 2017. JSR 222: Java Architecture for XML Binding (JAXB) 2.0. https://jcp.org/en/jsr/detail?id=222

[9] JSR. 2017. JSR 338: Java Persistence 2.2. https://jcp.org/en/jsr/detail?id=338

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

[11] Project Lombok. 2016. http://projectlombok.org/.

[12] Jaime Quinonez, Matthew Tschantz, and Michael Ernst. 2008. Inference of reference immutability. *ECOOP 2008–Object-Oriented Programming* (2008), 616–641. http://www.springerlink.com/index/6M5U5M330T81763T.pdf

[13] Sam Ruby, Dave Thomas, and David Hansson. 2009. *Agile Web Development with Rails, Third Edition* (3rd ed.). Pragmatic Bookshelf.

[14] Nilson SantÁnna, Eduardo Guerra, André Ivo, Fernando Pereira, Marcos Moraes, Vitor Gomes, and Luiz Gustavo Veras. 2014. Modelo Arquitetural para Coleta, Processamento e Visualização de Informações de Clima Espacial. *Simpósio Brasileiro de Sistemas de Informação* (2014), 125 – 136. http://www.lbd.dcc.ufmg.br/colecoes/sbsi/2014/0010.pdf

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

[16] 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