Quality Improvement in Annotated Code

Diego A. A. Correia
Aeronautical Institute of Technology (ITA),
São José dos Campos - SP, Brazil,
dalvac@gmail.com

Eduardo M. Guerra
Aeronautical Institute of Technology (ITA),
São José dos Campos - SP, Brazil,
guerraem@gmail.com

Fabio F. Silveira
Federal University of São Paulo (UNIFESP),
São José dos Campos - SP, Brazil,
fsilveira@unifesp.br

and

Clovis T. Fernandes
Aeronautical Institute of Technology (ITA),
São José dos Campos - SP, Brazil,
clovistf@uol.com.br

Abstract

In order to customize their behavior at runtime, a wide sort of modern frameworks do use code annotations at the applications' classes as metadata configuration. However, despite its popularity, this type of metadata definition inserts complexity and semantic coupling that is ignored by traditional software metrics. This paper presents identified bad smells in annotated code and defines new metrics that help in their detection by enabling a quantitative assessment of complexity and coupling in this type of code. Moreover, it proposes some strategies to detect those bad smells by using the defined metrics and introduces an open-source tool created to automate the process of bad smell discovery on annotated code.

Keywords: Metadata, Attribute-Oriented Programming, Code Annotations, Metrics, Bad Smells, Refactoring.

1. INTRODUCTION

Attribute-oriented programming is a program-level marking technique used to mark program elements, such as classes, methods and attributes, to indicate that they maintain application-specific or domain-specific semantics. It is a type of metadata definition that is placed in the source code of the application's classes to be consumed by a metadata-based framework [1] at runtime, or statically by an annotation processor. In the Java platform it has become popular with the native support for code annotations. In computer science, metadata can be used for many purposes. Examples are source code generation, compile-time verifications and class transformation. The metadata-based components consume metadata at runtime and use it for framework processing.

The metadata consumed by a framework or a tool can be defined in different ways. It can be defined inside the application class, by using code annotations or code conventions. Moreover, it can be defined by external sources, through the use of XML documents or databases. A framework can support the use of more than one form of metadata definition. Each of these definitions has its advantages and drawbacks [2], which makes it necessary to evaluate properly which one to use on a case-by-case basis.

Many mature metadata-based frameworks and Application Programming Interfaces (APIs) currently used in the industry utilize code annotations, such as Hibernate, EJB 3 and Spring Framework. Code annotations are easier to configure and more readable than metadata in external sources. However, the metadata configuration sometimes is a complex task and usually the application's behavior turns out to be tightly coupled with the framework metadata schema and semantic, which results in modularity loss in the application and puts the maintainability, testing and evolution of the software into risk. Furthermore, the heavy use of metadata may deteriorate code readability and raise its complexity. Maintaining software often means evolving the software, and modifying the existing code is a large part of modern software engineering. Therefore, code readability clearly is closely related to its maintainability, and is thus a critical factor in overall software quality [3].

Despite these serious consequences of misusing metadata in applications, the current metrics about complexity and coupling do not consider the metadata definition in any measurement, giving rise to open discussion to the applicability of traditional metrics to evaluate complexity and coupling in the usage of metadata-based frameworks [2].

Focusing on these problems, this paper identifies bad smells [4] in annotated code and defines new metrics that help in their identification by enabling a quantitative assessment of complexity and coupling in applications containing this type of code. Furthermore, it proposes some strategies to detect those bad smells by using the defined metrics and introduces an open-source tool created to automate the process of bad smell discovery on annotated code.

2. USING METRICS TO DETECT BAD SMELLS

A metric is the mapping of a particular characteristic of a measured entity to a numerical value [5]. An entity can be anything, e.g. a car. The characteristic also can be anything, e.g. its weight.

Metrics help with summarizing particular aspects of things and detecting outliers in large amounts of data. They provide a strong basis for summing up the many details of software. Metrics are very important in software engineering, since they enable engineers to keep control of complexity, making them aware of abnormal growth of certain parts of the system and/or bad quality of code. However, in order to effectively take advantage of metrics, they must be used in such a way that they provide real information and not just numerical values.

2.1. Thresholds

Regardless of the metric being used, some reference points must be known, i.e. what is too high or too low, too much or too little. There is a need to bind numerical values to useful semantics, and for this purpose thresholds are used. They divide the metric value into regions, assigning semantic meaning to each different region. Defining thresholds for a metric is a hard task. How do we know that the correct minimum grade to call a student "good" is 7.5? As a matter of fact, there is no perfect threshold. The best approach is to choose thresholds based on reasonable arguments. They don't need to be perfect, but should be useful in practice. This makes them good enough for most purposes, like assessing software artefacts.

In Table 2.1, what M. Lanza and R. Marinescu [5] came up by measuring a large number of Java and C++ systems with respect to 3 metrics is presented: Average Number of Methods (NOM) per class, Average Lines of Code (LOC) per method and Average Cyclomatic Number (CYCLO) [6] per line of code.

Table 2.1: Statistical thresholds computed from 45 Java and 37 C++ systems

Java				
Metric	**Low**	**Average**	**High**	**Very High**
CYCLO/ Line of Code	0.16	0.20	0.24	0.36
LOC/Method	7	10	13	19.5
NOM/Class	4	7	10	15
C++				
CYCLO/ Line of Code	0.20	0.25	0.30	0.45
LOC/Method	5	10	16	24
NOM/Class	4	9	15	22.5

2.2. Detection Strategy

Marinescu [5] [7] proposed a mechanism called detection strategy, for formulating metric-based rules that capture deviations from good design principles and heuristics, i.e. a technique to evaluate the design of object-oriented systems and to detect structural bad smells [4]. Technically, it is a composed logical condition, based on metrics, that identifies the design fragments that match the condition. Thus, it provides a list of suspect design fragments, i.e. all entities in the system conforming to the applied detection strategy.

Keep in mind that metrics give only evidence of bad design and not imperative information about that. For this reason, once the list of suspect design fragments is available, a human must go and analyze them to decide which actions should be taken in each situation. By using detection strategies an engineer can directly find code affected by a particular design flaw, rather than having to infer the real design problem from a large set of abnormal metric values.

The main principle of detection strategies is based on the fact that, alone, a metric cannot help answer all concerns about the design of a system. Metrics must be combined to provide relevant information. The mechanisms of filtering and composition are the bases of this technique.

Table 2.2: Classification of Data Filters

Type of Data Filter	Limit Specifiers		Filter Example
Marginal	Semantical	Relative	TopValues(10), BottomValues(5%)
		Absolute	> 20(HigherThan(20)), < 60 (LowerThan(60))
	Statistical		Box-Plot
Interval	Composition of two marginal filters, with semantical limit specifies of opposite polarities		(20,30) := Higher than(20) ^ Lower than(30)

A filter is a predicate that flags metric values of interest. The purpose of filtering is to reduce the initial data set by keeping only those design fragments that have special properties captured by the metric. In Table 2.2 by Marinescu [7], some samples of filters and their classification are presented.

In addition to the filtering mechanism, a composition mechanism to support a correlated interpretation of multiple result sets is needed. Further in this paper, in table 5.1, some examples of detection strategies created to detect bad smells on annotated code are presented.

3. ANNOTATIONS' BAD SMELLS

To help developers with the task of writing a well-structured annotated code, the authors of this paper are not going to define configuration principles, rules and heuristics, but instead are going to proceed like Fowler in [4], presenting situations where the configuration must be structurally improved. These situations refer to symptoms of bad configuration – bad smells – often found in annotated codes. They are presented in table 3.1 and can be divided into the following three categories: (a) class smells, related to a class overall, (b) declaration smells, referring to a single declaration of a class, method or attribute, and (c) annotation smells, referring to a single annotation.

From here on in this paper, for comparison purposes, it is assumed that two annotations are equal if they are of the same type and declare attributes with the same values.

Table 3.1: Annotations' Bad Smells

Bad Smell Description	Sample Code

Crowded Party
It is a *class smell* that refers to a class implementation that contains too many annotations overall. In the sample code, the source code uses a large number of JPA annotations in comparison to actual code lines. This bad smell correlates with low code readability and high complexity.
	`@NamedQuery(name = "getCrowdedClass",`
	`  Query = "from Classes where`
	`  n > :veryhigh")`
	`@Entity`
	`@Table(name = "Crowded")`
	`@Inheritance(strategy = SINGLE_TABLE)`
	`@DiscriminatorColumn(name = "_type_",`
	`  discriminatorType = STRING)`
	`@DiscriminatorValue("crowded")`
	`@Cache(usage = NONSTRICT_READ_WRITE)`
	`public class Crowded {`
	`  private Integer id;`
	
	`  @Id`
	`  @GeneratedValue(strategy=SEQUENCE,`
	`  generator = "hibseq")`
	`  @SequenceGenerator(name =”myGenerator",`
	`  sequenceName = "common_seq")`
	`  public Integer getId(){`
	`  return this.id;`
	`  }`
	`}`

Annotation Drowning Unlike the overall aspect of the *Crowded Party* definition, the *Annotation Drowning* is a *declaration smell* and refers to single elements with too much configuration in their declaration. It could be identified in an element with a high amount of annotations in its declaration, or in an element with a moderate number of annotations that together contain a high number of declared attributes. In the sample code, the class declaration definitely presents *Annotation Drowning*.	

Fat Annotation

This is an *annotation smell* referring to annotations containing too many configured parameters. Within the sample code, in the JAXB *@XMLElements* annotation are defined two annotations *@XMLElement*, each containing six attributes: 'name', 'nillable', 'required', 'namespace', 'defaultValue' and 'type'.

The need of configuring the annotation that much, which already is a configuration itself, increases complexity and also impacts code readability. This might indicate that a single annotation is encapsulating more configurations than it should.

```
@XMLElements({
@XMLElement(name = "fat",
         nillable = true,
         required = true,
         namespace =
            http://www.fat.com/xhtml",
         defaultValue = "Fat",
         type = FatClass.class),
@XMLElement(name = "fatter",
         nillable = true,
         required = true,
         namespace =
            http://www.fat.com/xhtml",
         defaultValue = "Fatter",
         type = FatterClass.class),
})
public Object getData(){
   return this.data;
}
```

Popular Configuration

This is an *annotation smell* that deals with the problem of having the same annotation repeated over the code many times. This practice makes maintenance harder in the case one needs to apply changes over the repeated element. The problem faced here is similar to the *Shotgun Surgery* bad smell [4]. In the sample code, the EJB 3 annotation *@TransactionAttribute* is repeated several times in the declared Bean.

Prolix Configuration

It is similar to the *Popular Configuration*, but as a *class smell*. It indicates that a class contains too many repeated annotations overall. This situation has similar implications to the ones discussed when considering the *Popular Configuration* bad smell.

```
@Stateless
public class PopularBean {...

   @TransactionAttribute(SUPPORTS)
   public String popularMethod(){...}

   @TransactionAttribute(SUPPORTS)
   public String getPopularity(){...}

   @TransactionAttribute(SUPPORTS)
   public String showPopularity(){...}

   @TransactionAttribute(SUPPORTS)
   public String hidePopularity(){...}
   ...
}
```

Digging Too Deep

It is an *annotation smell* and refers to an overuse of nested annotations to define an attribute. In the sample code, the JPA annotation *@AssociationOverrides* uses the annotation *@AssociationOverride*, which uses the annotation *@JoinColumn* to define the *joinColumns* attribute.

The approach of using nested annotations is quite tricky, since while helpful, it may dramatically increase complexity, as usually happens in any nested code where a high nesting level is reached. This practice makes harder to apply changes in the code and to promote code refactorings.

```
@Entity
public class DiggingTooDeep {
   @AssociationOverrides({
      @AssociationOverride(name="digging",
         joinColumns =
            @JoinColumn(name="DIG_ID")),
      @AssociationOverride(name="tooDeep",
         joinColumns =
            @JoinColumn(name="DEEP_ID"))
   })
   public void goDeeper(){
      ...
   }
}
```

Bag Over-Stamped

A *class smell* that refers to annotated code containing annotations from many different annotation schemas.

In the sample code there are annotations from five different annotation schemas: EJB 3 (*@Entity* and *@Id*), XapMap [8] (*@XMap*), JAXB (*@XMLRootElement* and *@XMLAttribute*), Java Core (*@SuppressWarnings*) and a domain annotation [9] (*@DomainAnnotation*) defined in the application itself.

The use of framework annotations in an application couples its classes to that metadata schema. This sort of coupling is considered a semantic or indirect coupling [2] [9] between the application class and the framework.

```
@Entity
@XMap(mappingTo = Example.class)
@XMLRootElement
@DomainAnnotation
public class OverStamped {

   @Id
   @XMap(field = "key")
   @XmlAttribute("id")
   @SuppressWarnings("unused")
   public Integer getId() {
      return this.id;
   }
}
```

4. ANNOTATIONS' METRICS

In this section, the authors of this paper are going to propose metrics to be used to assess quality and complexity of annotated codes. They were chosen in the context of detection strategies and meant to be useful for this purpose. It is important to notice that this is a work in progress and that the authors still do not have statistical data related to these metrics to define statistical thresholds. For this reason, in this paper they are going to limit the estimation of thresholds to an approach based solely on their experience. Since the authors of this paper developed a tool that automates the calculation of these metrics, which enables the evaluation of them in different projects, a future work is a statistical analysis aiming to correct and validate the values of the thresholds used here. This tool is introduced in section 7.

The number three is used as a reference for a reasonable number of information that someone can easily understand in a single element, being set as the AVERAGE threshold. The number seven is used for the HIGH threshold, since it was proven to be the upper limit of the human short-term memory [5]. The LOW threshold does not make much sense for annotations, since its usage is optional and a class with few annotations or none does not incur in any bad practice. Find in table 4.1 and table 4.2 the catalog of proposed annotation metrics:

Table 4.1: Elementary Annotations' Metrics

Elementary Metrics
ASC: *Annotation Schemas in Class* Number of different annotation schemas being used or referenced in a class. The schemas are distinguished by the packages containing the definition of the annotations in use. Annotations defined in the same package belong to the same annotation-schema.
AED: *Annotations in Element Declaration* Number of annotations present in the declaration of a single element of a class (i.e. an attribute, a method or a class). Nested annotations, as well as the annotations present in the method's parameters, are considered.
AA: *Attributes in Annotation* Number of attributes defined in an annotation. The attributes of nested annotations are also considered.
LOCAD: *LOC in Annotation Declaration* Number of lines of code (LOC) in an annotation declaration.
ANL: *Annotation Nesting Level* Maximum nesting level of annotations in an annotation declaration. The nested annotations used to define the annotation's attributes are considered as well.
UAC: *Unique Annotations in Class* Number of unique annotations in a class. Notice that this metric can easily be extended for packages.
ARC: *Annotation Repetition in Class* Number of annotations in a class that are equal to a specific annotation declared in the same class.
ARP: *Annotation Repetition in Package* Number of annotations in use at the classes of a package that are equal to a specific annotation declared in a class of the package.

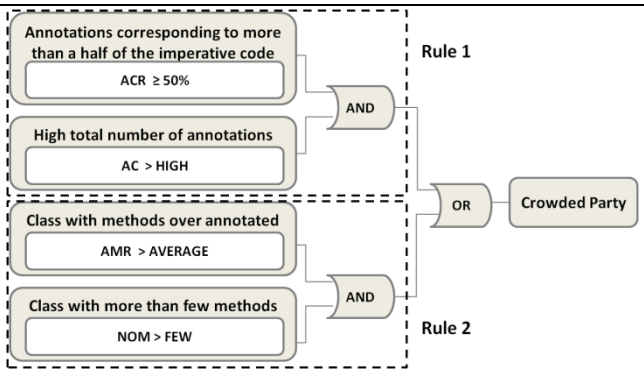
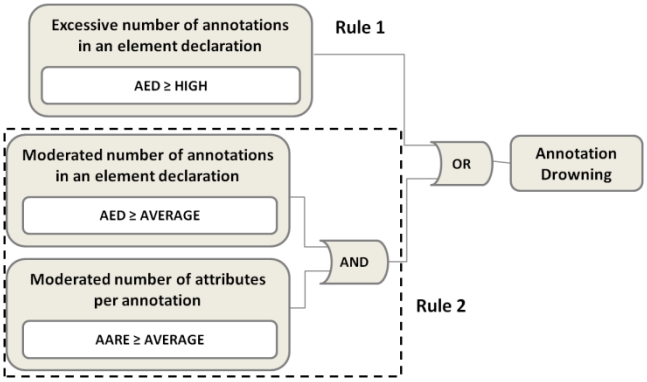
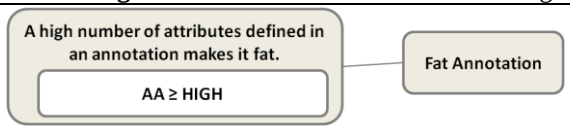
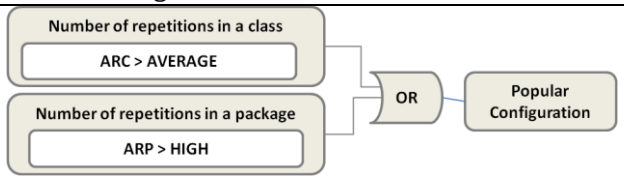
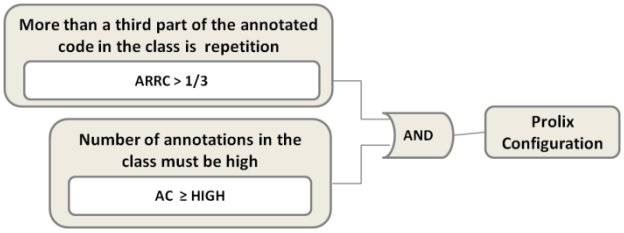
Table 4.2: Non-Elementary Annotations' Metrics

Non-Elementary Metrics (with formulas)
AC: *Annotations in Class* Total number of annotations present in a class. It is the sum of the AED metrics for each element of the class. $$AC = \sum_{\substack{each\ class \\ element}} AED$$
AMR: *Annotations per Method Ratio* Average number of annotations present in the declaration of a method in a class. Nested annotations are considered in the computation, as well as the annotations present in the method's parameters. $$AMR = \frac{\sum_{M \in Methods} AED(M)}{NOM}$$
AARE: *Attributes per Annotation Ratio in Element* Average number of attributes contained in the annotations present in an element. Attributes of nested annotations are considered in the computation. $$AARE = \frac{\sum_{\substack{each\ annotation \\ in\ the\ element}} AA}{AED}$$
ACR: *Annotations Codification Ratio* Number of lines of code with annotations divided by the total number of lines of code in a class. Line of code is being considered as any non-blank line in a source file. A line of code is considered to be with an annotation if it is part of an annotation declaration. $$ACR = \frac{\sum_{\substack{each \\ annotation}} LOCAD}{LOC}$$
ARRC: *Annotation Repetition Ratio in Class* Percentage of repeated annotations in a class. It is the number of repeated annotations divided by the number of total annotations present in the class. $$ARRC = 1 - \frac{UAC}{AC}$$

5. DETECTING ANNOTATION BAD SMELLS

Based on the annotations' metrics definitions, it is possible to detect bad smells on annotated codes following the principles of detection strategy. Table 5.1 presents the detection strategies designed for the bad smells stated section 3, representing the detection strategies by using schematic diagrams, as shown in figures 5.1 to 5.7.

Table 5.1: Detection Strategies for Annotations' Bad Smells

Bad Smell and Detection Strategy Description	Detection Strategy Schema (DSS) or Formula
Crowded Party As seen before, it is a *class smell*. The first rule intends to filter classes with a high number of annotations overall (AC > HIGH avoids considering classes too small). The second rule considers classes containing methods over annotated (NOM > FEW assure the analysis only for classes with a moderate number of methods). Notice that the same analysis done for methods could be done for attributes as well. It was intentionally omitted to keep the representation simple.	 **Figure 5.1:** DSS for Crowded Party
Annotation Drowning As discussed before, it is a *declaration smell* and refers to single elements with too much configuration in their declaration. It can be identified by verifying the number of annotations in an element declaration, as in the first rule. Moreover, the second rule deals with another possibility: filtering the elements having a moderated number of annotations, and such that they contain a moderated number of declared attributes. Summing up, the element ends up with a high number of attributes, which determines too much configuration in its declaration.	 **Figure 5.2:** DSS for Annotation Drowning
Fat Annotation Remember that it is an *annotation smell*, so, the filtering is done for each annotation. The rule of detection searches for annotations with a HIGH number of attributes.	 **Figure 5.3:** DSS for Fat Annotation
Popular Configuration As seen before, it is an *annotation smell*. Its detection strategy is straightforward, considering the definitions of the metrics ARC and ARP. It searches for annotations that are repeated over classes and packages.	 **Figure 5.4:** DSS for Popular Configuration
Prolix Configuration Remember that it is a *class smell*. Its detection strategy is straightforward, considering the definition of the metric ARRC. However, as the value of ARRC is a rate, an additional filter is put together to avoid classes with few annotations. Notice that this strategy can be easily extended for packages.	 **Figure 5.5:** DSS for Prolix Configuration

Digging Too Deep

As discussed before, it is an *annotation smell*. Its detection strategy is straightforward, considering the definition of the metric ANL. It searches for annotations that do not have a LOW nesting level.

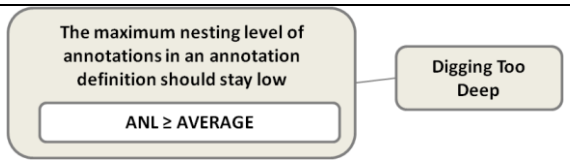


Figure 5.6: DSS for Digging Too Deep

Bag Over-Stamped

As seen previously, it is a *class smell*. Its detection strategy is straightforward, considering the definitions of the metric ASC. It looks for classes semantically coupled with more than a LOW number of annotation schemas.

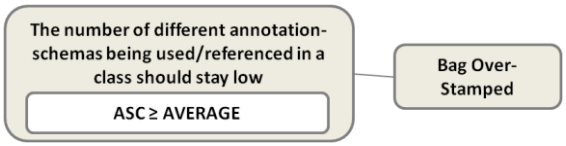


Figure 5.7: DSS for Bag Over-Stamped

6. ANNOTATION REFACTORINGS

The previous sections of this paper presented techniques and definitions on how to detect bad smells on annotated code. The next step is to use techniques and best practices to refactor the source code and eliminate these bad smells.

Refactoring annotated code may be quite tricky. The annotation bad smell in the application code may be a result of misusing a set of annotations. Also, it may be a design issue on the annotation definition, such as when an important attribute definition is missing in an annotation and when a single annotation is responsible for too much configuration. Furthermore, it may be a consequence of a design flaw in the application code itself, such as when an annotation is duplicated in different classes and when a class with too many responsibilities receives annotations from many different frameworks. So, when considering a refactoring on an annotated code there is a primarily need to identify if the problem is only in the application's code, or if it is also a consequence of the annotation schema provided by the metadata-based framework.

A general approach for refactoring annotated code is to search for a solution that uses the set of annotations more adequate for the application. The refactorings exposes guidelines for the application to reorganize the metadata definition based on the annotation schema provided by the adopted frameworks. They also indicate changes in the metadata schema adopted by the framework. The following subsections present some techniques that might be the target of refactorings and that can be used to eliminate some of the annotation bad smells.

6.1. External Metadata Definition

Annotations are not always the best metadata definition strategy for all situations [11]. For instance, when a class can be configured with more than one metadata set in different contexts, the use of annotations would probably generate code smells that indicate poor readability. Another example is when a class needs to be reused in a context where such metadata is not needed and the dependence with the framework annotations is undesirable.

An alternative to avoid the use of annotations is to use an external definition to metadata, usually defined in XML documents. When the framework does not support this alternative, it should be refactored to implement the pattern *Metadata Reader Strategy* [1]. This pattern provides a structure in which the framework can provide more alternatives for metadata reading. In this case, the application can even provide a customized metadata reader which fits best its needs.

The use of external metadata definition for some information may reduce the number of annotations in the classes, improving its readability and reducing the coupling with a given annotation schema. Using this approach, the class definition and the metadata from a schema can be accessed separately. However, this decoupling has the negative effect of increasing the distance between the class and its metadata definition.

This refactoring may be especially useful to reduce the number of annotations, helping in the elimination of bad smells like *Crowded Party*, *Annotation Drowning* and *Digging Too Deep*. It also decouples the application classes from one metadata schema, helping to reduce the *Bag Over-Stamped*.

6.2. Code Conventions

Another alternative to annotations is the use of code conventions. By using conventions, the framework can infer the metadata from the intrinsic metainformation of the element, like its name or its type. It is usually combined with the usage of annotations to reduce the number of configurations needed for a class. The use of this approach may save the creation of many annotations, improving the code readability.

If the framework does not support the use of conventions, the metadata reading mechanism should be refactored in order to start supporting. In this case, an analysis needs to be performed to identify naming rules and other characteristics that can be used to expose a metadata value used by the framework. For example, if a class needs to

define an annotation that configures a respective validation class for it, a naming convention strategy is to search for a class named *class_name + "Validator"*. In the cases where the rule does not apply, the annotation is still an alternative.
Moreover, if the framework already supports conventions, a possible refactoring in the application could be to rename and adapt elements to follow the framework code conventions. After that, the annotations could be removed. Considering the example given in the previous paragraph, a possible refactoring would be to rename the validation class in order to remove the annotation in the class that it validates.
However, this approach also has the drawback to not explicitly indicate the metadata, since the information would still be within the code, but implicitly defined. It is not clear in the code that, for instance, changing a class or a method name, the framework behavior when dealing with that class could be changed. Because of that, it is important to communicate the conventions used to all the development team.
Using code conventions can reduce the number of annotations or annotations attributes, helping to remove bad smells like *Crowded Party*, *Annotation Drowning*, *Fat Annotation and Digging Too Deep*. Conventions are also effective to capture recurrent configurations, being a good option when facing occurrences of *Popular Configuration* and/or *Prolix Configuration* bad smells.

6.3. General Configuration

An annotation is meant to add new information in only one code element. In situations where more than one code element should be configured with the same metadata, the same annotation should be present in all of them. It may increase the total number of annotations and difficult the maintainability due to the duplication.
The use of more general configurations may reduce this annotation repetition, making general changes in the metadata configuration easier. The framework should be prepared to read metadata in elements and interpret it as a piece of the metadata of their members. For instance, an annotation in the class can configure a general metadata for its methods and an annotation in a method can configure a general annotation for its parameters. More general configurations involving packages or a greater number of classes can be performed in external configurations.
An example of this practice can be found in the definition of interceptors in the EJB 3 API [12]. The annotation *@Interceptors* can be used at method level to define interceptors for that method and at the class level to define interceptors for all its methods. Since the definition is accumulative, the method interceptors are added to the ones defined in the class. For methods where the class interceptors should not be executed, the annotation *@ExcludeClassInterceptors* can be used. A more general configuration can also be made in the deployment descriptor, configuring an interceptor for all EJBs in that application. This configuration can be excluded as well from classes or methods using the annotation @ExcludeDefaultInterceptors. Figure 6.1 presents an example of this practice.

```
*** Definition of a default interceptor ***
<assembly-descriptor>
   <interceptor-binding>
      <ejb-name>*</ejb-name>
      <interceptor-class>org.example.DefaultInterceptor</interceptor-class>
   </interceptor-binding>
   ...
</assembly-descriptor>

*** EJB Definition ***
@Stateless
@Interceptors ({ClassInterceptor.class})
public class ExampleBean implements ExampleRemote {

   //Has default and class interceptors
   public void method1(){...}

   //Has default and class interceptors plus the one defined in the method
   @Interceptors ({MethodInterceptor.class})
   public void method2(){...}

   //Has only the default interceptor since it excluded the one defined in class
   @ExcludeClassInterceptors
    public void method3(){...}
}
```

Figure 6.1: Example of general configurations for EJB 3 Interceptors

If the used framework provides the possibility of general configurations, the application could use it to reduce the number of annotations and to avoid a high number of duplications. The developer should search for common

configurations and identify if it can be generalized in a class or in a group of classes. This technique is not recommended without following a well-defined rule when the duplications are spread along the application classes.
Since general configurations can reduce the number of annotations and annotation attributes, it can help to eliminate *Crowded Party*, *Annotation Drowning and Fat Annotation*. However, the main intent of this practice is to avoid the annotation duplication that can be detected by *Popular Configuration* and *Prolix Configuration* bad smells.

6.4. Default Values

For some framework domains, the values of some annotation attributes are mandatory for its processing. However in some situations there is a value that is configured most of the times. The definition of a default value for an attribute may avoid unnecessary configurations, reducing the amount of code used for annotations. It is also compliant with the pattern Gentle Learning Curve [13], since with default values the number of necessary configurations for a simple framework instantiation should be reduced.
For frameworks that do not provide default values, an analysis of the most common value should be made for the default configuration. If the most common value can depend on the application, the framework can provide an alternative for the default value to be configured externally. In the application, the annotation attributes which receives default values can simply be removed.
The definition of default values can reduce the number of attributes and even eliminate the use of certain annotations, being important in order to eliminate *Fat Annotation* and *Digging Too Deep* bad smells.

6.5. Structural Changes

There are many different ways to structure the same metadata using annotations. The framework developer should design the annotations to be intuitive to create and to not impact negatively in the code readability. These refactorings focus on alternatives to restructure the annotation schema.
Since the annotations need to be changed, the framework component that reads the annotation should be also adapted to them. The applications that use the framework also needs to be modified. To enable backward compatibility, the framework can maintain the old annotation schema and select the appropriate metadata reader using the pattern *Metadata Reader Strategy* [1].
The following are examples of structural refactorings in annotations: join annotations; split annotation; join attributes; split attribute; group attributes in a composite annotation; use well-formed expression; and structure expression in attributes. Figure 6.2 presents an example of how an annotation could be refactored. In the original version, the annotation *@Conditions* is composed by other annotations that structure the information. In the refactored version, the annotation condition receives a well-formed expression that is interpreted by the framework. Based on this example, it is clear how a change in the annotations structure can lead to a better metadata organization improving the code readability.
This kind of refactoring does not remove information from the annotations, but only restructure them. So, it can be used to avoid frequent occurrences of *Fat Annotation* and/or *Digging Too Deep* bad smells.

```
@Conditions({
   @Condition(prop="age",value="18",oper=">"),
   @Condition(prop="sex",value="male",oper="=")
})
public void onlyForAdultMans() { … }
```

```
@Condition("age>18 && sex='male'")
public void onlyForAdultMans() { … }
```

Figure 6.2: Example of use of well-formed expressions in annotations

6.6. Domain Annotations

Repeated configurations using annotations usually arises in an application for a special reason. Methods and classes with the same annotations normally have something in common related to the application domain. The domain annotations are a representation of domain-specific metadata [14]. Using this approach the application uses metadata related to its domain instead of framework-specific metadata. These annotations can be mapped dynamically or statically to the framework annotations [9].
To support a dynamic reading of domain annotations, the framework annotation reading mechanism should search for its annotations not only in the code elements but also inside their other annotations. Using this technique, the application developer is able to create its own annotations and configure it with the framework annotations. Then, the framework should consider the elements annotated with the new annotation as if they were directly annotated with its annotations.
Figure 6.3 presents an example of a domain annotation definition. The application annotation @Administrative provides a metadata for a method that indicates that it is meant for the system administration. The annotations in bold configures the following characteristics for these methods: they should be executed in a transaction; they can only be

accessed by administrators; and they should be logged in the database after its execution. Further, as seen in figure 6.4, the administrative methods do not need to use all of these annotations, only @Administrative, which is related to the application domain.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@Transaction(REQUIRED)
@SecurityConstraints(role="admin")
@LoggingConfig(info={METHOD,RETURN}, when=AFTER, where=DATABASE}
public @interface Administrative{ ... }
```

Figure 6.3: Example of annotation definition for a domain annotation

```
public class ExampleClass {
   @Administrative
   public void method1(){...}
}
```

Figure 6.4: Domain annotation usage

If the framework does not support the domain annotation reading directly, an alternative is a static mapping among the annotations. Daileon [9] is an example of a tool which uses bytecode manipulation to map annotations statically.
Domain annotations can be used to reduce the number of annotations, being an alternative to eliminate the bad smells *Crowded Party* and *Annotation Drowning*. It is also a solution to reduce annotation repetition, specifically detected by the bad smell *Popular Configuration*. Differently from a general configuration, using domain annotation is possible to eliminate the duplications spread in the source code. An important characteristic of domain annotations is that they decouple the application class from the framework annotations, helping to eliminate the *Bag Over-Stamped* bad smell as well.

6.7. Refactoring Summary

Table 6.1 summarizes the correlation between the techniques presented and the documented annotation bad smells. It is important to notice that each bad smell had at least two techniques that can help in its elimination.

Table 6.1: Refactoring techniques applied to annotations' bad smells

	Crowded Party	**Annotation Drowning**	**Fat Annotation**	**Popular Config**	**Prolix Config**	**Digging Too Deep**	**Bag Over-Stamped**
External Metadata Definition	X	X				X	X
Code Conventions	X	X	X	X	X	X	
General Configurations	X	X	X	X	X		
Default Values			X			X	
Structural Changes in annotations			X			X	
Domain Annotations	X	X		X			X

7. ANNOTATION SNIFFER

In practice, would not be reasonable for someone to manually calculate the metrics presented here. Software is needed for this purpose. Thus, the authors of this paper developed an open-source tool called "Annotation Sniffer" [10] which automates this calculation. Evaluating these metrics in different projects will enable a statistical analysis aiming to correct and validate the values of the thresholds used in section 4.
Additionally, as a future work, the tool will implement the detection strategies shown in table 5.1 in order to be able to automatically detect annotations' bad smells.

7.1. Tool Elements

The tool, written in Java language, was developed as an Eclipse [15] plugin. This decision intended to reduce efforts on tasks such as getting project information and parsing Java source files, which can be accomplished by using the Java Model API and the Java Document API, provided by the Eclipse Java Development Tools (JDT) [16].
The Java Model API enables navigation of the Java element tree. The Java element tree defines a Java centric view of a project. It surfaces elements like package fragments, compilation units, binary classes, types, methods and fields. On the other hand, the Java Document Model API enables manipulation of a structured Java source document.

7.2. Tool Design

The tool was designed to be extensible regarding metrics, so that new metrics definitions can be added to the tool with no coding modifications in the core code.

For this purpose, the metrics to be used by the tool are declared in an external XML file, called *metrics.xml*. In this file, as shown in figure 7.1, the name of the metric and the qualified name of the Java class that implements the metric should be declared.

```
<?xml version="1.0" encoding="UTF-8"?>
<list>
    <net.sf.asniffer.util.MetricRepresentation>
        <name>Attributes in Annotation (AA)</name>
        <path>net.sf.asniffer.metrics.impl.AaMetric</path>
    </net.sf.asniffer.util.MetricRepresentation>
...
</list>
```

Figure 7.1: Sample snippet of a metrics.xml file

A metric implementation must inherit either the abstract class *SimpleMetric* or *MultiMetric*, depending on its nature. Additionally, in order to help in the task of implementing a metric, the class *UnitParser* provides a set of utility methods that may help the developer in tasks such as parsing Java source files.

The core of the application is the class *ClassMetrics.* It is responsible for calling the XML parser for the file *metrics.xml* and for evaluating each metric found in the parsed file. Finally, the class *MetricsXMLParser* executes the parsing of the XML file *metrics.xml* and creates one instance of each metric found in the file, by using Java Reflection [17].

To achieve the decoupling between the core of the application and the metrics implementation, the design pattern Strategy [18] was used. Figure 7.2 brings a class diagram of the tool.

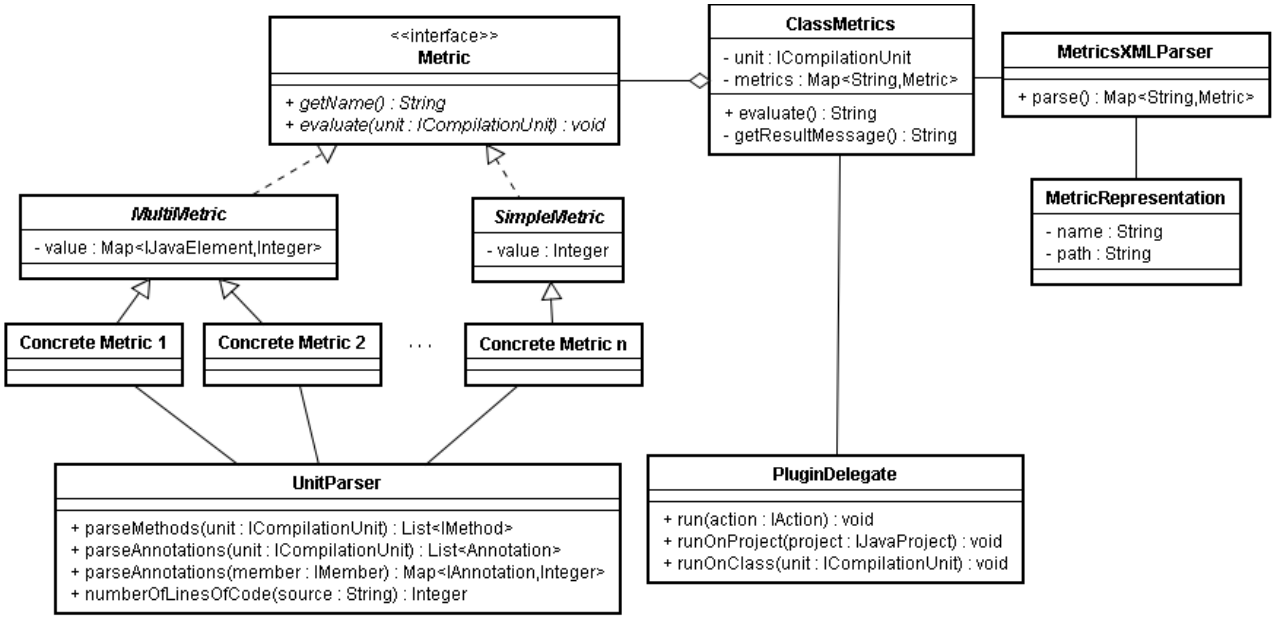


Figure 7.2: Class diagram of the tool "Annotation Sniffer"

8. CONCLUSION

In this paper the authors presented identified bad smells on annotated code, proposed definitions of metrics that help on their detection, created detection strategies that use the defined metrics to detect those bad smells, detailed refactoring mechanisms to eliminate them and finally introduced an open-source tool developed to automate the process of bad smell discovery in annotated code.

The authors consider the main contributions of this work to be: the identification of bad smells on annotated code, the definition of metrics that embrace the impact of metadata in the code of an application, the creation of detection strategies for the bad smells identified, the discussion and guidelines provided to help refactoring annotated code and

ultimately the creation of an open-source tool to automate the calculation of these metrics, which is the first step on enabling the implementation of an automated bad smell discovery process.
As a future contribution to this research, assess the defined metrics for diverse Java applications using the developed tool is certainly a main goal since it would enable a statistical study to define better thresholds for the metrics and possibly improve the detection strategies. Also, the implementation of the presented detection strategies in the tool is a target since it would enable the automation of the bad smell discovery process.
Also important is the validation of the created strategies in a real application. It would certainly help with improving the strategies and contribute to the study of refactoring mechanisms of annotated code.
This work benefits far beyond the application developers, also reaching to the metadata-based framework designers. Based on the characteristics and metrics of the applications that use the framework annotations, it is possible to improve the metadata schema definition of the framework and lead applications to have a source code more readable and less complex.

References

[1] Guerra, Eduardo; Souza, Jefferson; Fernandes, Clovis; *A Pattern Language for Metadata-based Frameworks*, 16th Conference on Pattern Languages of Programming, Chicago, August 2009.

[2] Guerra, Eduardo; Silveira, Fábio; Fernandes, Clovis; *Questioning Traditional Metrics for Applications Which Uses Metadata-based Frameworks*, 3rd Workshop on Assessment of Contemporary Modularization Techniques (ACoM.09), Florida, October 2009.

[3] Buse, Raymond; Weimer, Westley; *A Metric for Software Readability*, ISSTA'08, Washington, July 2008.

[4] Fowler, Martin; Beck, Kent; Brant, John; Opdyke, William; Roberts, Don; *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[5] Lanza, Michele; Marinescu, Radu; *Object-Oriented Metrics in Practice – Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer, 2006.

[6] McCabe, T. *A Complexity Measure*, IEEE Transactions on Software Engineering, December 1976.

[7] Marinescu, Radu; *Detection Strategies: Metrics-Based Rules for Detecting Design Flaws*. Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04), 2004.

[8] *Xapmap: Cross Application Mapping Framework*, 2009. Available at: http://xapmap.sourceforge.net.

[9] Perillo, José; Guerra, Eduardo; Silva, Jefferson; Silveira, Fábio; Fernandes, Clovis; *Metadata Modularization Using Domain Annotations*, 3rd Workshop on Assessment of Contemporary Modularization Techniques, Florida, October 2009.

[10] *Annotation Sniffer for Eclipse*, 2010. Available at: http://asniffer.sourceforge.net

[11] Fernandes, C.; Ribeiro, D.; Guerra, E.; Nakao, E; *XML, Annotations and Database: a Comparative Study of Metadata Definition Strategies for Frameworks*. In: XML: Aplicações e Tecnologias Associadas, 2010, Vila do Conde, Portugal.

[12] *JSR 220: Enterprise JavaBeans 3.0, 2006*. Available at: http://www.jcp.org/en/jsr/detail?id=220

[13] Foote, Brian; Yoder, Joseph. *The selfish class*. In: Vlissides, John M.; COPLIEN, James O.; KERTH, Norman L. *Pattern languages of program design 3*. Boston: Addison-Wesley Longman, 1997. Chap. 25, p. 451-470.

[14] E. Doernenburg. *Domain Annotations*. In: The Thought-Works Anthology: Essays on Software Technology and Innovation, chapter 10. Pragmatic Bookshelf, Raleigh, NC, USA, March 2008.

[15] *Eclipse Home*. Available at: http://www.eclipse.org

[16] *Eclipse Java Development Tools (JDT)*. Available at: http://www.eclipse.org/jdt/

[17] Forman, Ira R.; Forman, Nate; *Java Reflection in Action*. Manning Publications, October 2004.

[18] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John M.; *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, November 1994.