Questioning traditional metrics for applications which uses metadata-based frameworks

3 AUTHORS, INCLUDING:

Eduardo Guerra
National Institute for Space Research, Brazil
28 PUBLICATIONS **46** CITATIONS

SEE PROFILE

Fábio F. Silveira
Universidade Federal de São Paulo
22 PUBLICATIONS **38** CITATIONS

SEE PROFILE

Questioning Traditional Metrics for Applications Which Uses Metadata-based Frameworks

Eduardo M. Guerra

Aeronautical Institute of Technology

guerraem@gmail.com

Fábio F.Silveira

Federal University of São Paulo

fsilveira@unifesp.com.br

Clóvis T. Fernandes

Aeronautical Institute of Technology

clovistf@uol.com.br

Abstract

Many recent frameworks use the metadata configuration in the application classes to customize the behavior at runtime. This metadata can be defined using external sources, like XML files and databases, by using code annotations or code conventions. This metadata definition inserts complexity and semantic coupling that is ignored by traditional metrics. This paper tackles examples in real frameworks where the use of current metrics, especially the ones regarding complexity and coupling, are not appropriate. It also presents challenges that must be considered in the definition of new metrics that considers metadata definition.

General Terms: Measurement, Design.

Keywords *Metadata; Code Annotations; Frameworks; Metrics;*

1.   Introduction

Metadata-based frameworks [1] are frameworks that process their logic based on classes´ metadata whose instances they are working with. Many mature frameworks and APIs popular in the industry use this approach, such as Hibernate [2], EJB 3 [3], and Spring Framework [4]. In these, developers configure metadata relative to the application classes which is consumed by the framework at runtime. This specific metadata about each class allows the framework to customize its behavior for each one.

One alternative for metadata definition is the use of at-tribute-oriented programming, which is a program-level marking technique that allows developers to mark programming elements, such as classes and methods, to indicate application-specific or domain-specific semantics [5]. In the Java platform, this programming style appeared with XDoclet [6], but it has become popular with the native support to code annotations in the Java Language [7]. Other alternatives for metadata definition are external sources, such as both XML files and databases, or even by code conventions [8].

The metadata usage to configure rules for frameworks substitutes imperative programming for a more declarative approach. The metadata configuration sometimes is a complex task and usually the application behavior became tightly coupled to the framework metadata schema. However, current metrics about complexity and coupling do not consider the metadata definition in any measurement.

This paper questions the applicability of the current metrics to evaluate complexity and coupling in applications that use metadata-based frameworks. It also defines some challenges for the definition of metrics considering the behavior that is programmed declaratively by using metadata.

This paper is organized as follows. Section 2 presents the different types of metadata definition and their peculiarities. Section 3 presents some current metrics, focusing in the ones about coupling and complexity. Section 4 summarizes how metadata can add complexity. Section 5 shows how metadata create coupling based on real examples. Next, in Section 6, some challenges for measurements that consider metadata are presented, and concluding remarks are given in the Section 7.

2.   Metadata Definition

There is more than one way to define metadata for a framework to consume. It can be defined inside the applica-

tion 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. The "Metadata Reader Strategy" comprises a design pattern that allows the framework to consume metadata from different sources, whilst the Metadata Reader Chain allows the combination of different sources at the same time [1].

The metadata defined in external sources are read and interpreted by the framework at runtime. The use of XML descriptors is more usual, but other formats of files and databases are also possible alternatives. By nature, these forms of metadata definition keeps a distance for the source code, allowing it to be changed at deploy time or even at runtime. This distance also makes this approach more verbose and less intuitive.

Attribute-oriented programming is a program level marking technique that allows developers to mark programming elements to indicate specific semantics [5]. It begins in Java with XDoclet [6], where annotations are defined inside comments in the application classes and used for generating usually source code and XML descriptors.

The JSR 175 [7] has created the native support of code annotations to the Java language. The main APIs from enterprise development in Java EE platform, like EJB 3 and JPA [2], are based in code annotations. The annotations are easier to configure and more readable than metadata in external sources because they are closer to the application class. This proximity also brings drawbacks. For example, annotations can not be changed after the compilation activity, and creates dependence between the class and the framework metadata schema.

Another alternative for metadata definition is the use of code conventions [8]. Rules for class and method names can be used for the framework to infer a specific semantic for it. A common example is the use of methods with "get" and "set" prefix in the Java Bean Specification [9], which means methods for getting and setting information. Ruby on Rails [10] is an example of a framework that uses a lot of conventions for metadata definition. Code conventions are more limited than other approaches and can not be used when a more complex metadata schema is necessary.

3.   Traditional Metrics

Size-based metrics are direct measurements that consider in a quantitative way the size of the software. Examples are number of classes, number of operations, number of packages, and lines of code [11]. Computed proportions of those metrics are also important for the evaluation. For example class structuring can be evaluated by using number of operations per class and operation structuring, calculating the number of lines of code per operation. In general, a large

number of these values indicate that the software is more error prone and complex, although it is open to interpretation and depends on the individual circumstances.

Complexity-based metrics are intended to evaluate the difficulty to understand, develop and maintain the software. The cyclomatic complexity counts the number of possible logic paths in a piece of code [12]. It considers the possible flows that an execution can follow that impact in the code readability and maintainability, as well in the number of tests cases for a good coverage. The Halstead Effort [13] is a metric that considers the number of operands and operators to evaluate the effort and the volume of written code.

Coupling metrics evaluate how intensive and how dispersed is the coupling in the system [11]. The coupling intensity can be measured by using the number of operation calls per operation, and the coupling dispersion can be measured as the number of called classes per operation call. A more recent approach for the coupling measurement, called dynamic coupling, counts the number of calls at runtime and not statically [14].

None of these metrics consider metadata elements inside the source code, like code annotations and code conventions. They also do not analyze external sources of information capable of changing the application behavior, like XML descriptors and information stored in databases. In applications which use metadata-based frameworks the metadata definition has a great influence in the application behavior. The next sections present examples about how metadata can add complexity and creates coupling.

4.   Metadata Add Complexity

The metadata definition uses a declarative approach inside an imperative language and the current metrics usually ignores this information. Configure annotations in the code elements and create XML descriptors are complex activities and their existence in the application impacts in the code readability and maintainability.

The use of code conventions also adds complexity, because the use of a different name for a method or a class can change the application behavior. The implicit metadata carried by a code element name is used by frameworks and bugs can be originated by identifiers that do not follow correctly the naming conventions.

JPA [2] is a standard API for object-relational mapping that use annotations, code conventions or XML documents for metadata configuration. By using this approach, the persistent entities are represented by annotated classes, which basically contain attributes, getters and setters methods. Analyzing the complexity by using the current metrics, those classes are considered with low complexity, because there are not many possible logic paths in the execution and only a few operations are performed in their methods.

The Figure 1 and Figure 2 represent, respectively, a piece of the classes *Paper* and *Author* annotated with JPA metadata, in which there is a mapped relationship between them. The metadata used to map the relationship columns are in the class *Author* in the attribute *paper*. In the instance variable *authors* in the class *Paper*, there are other configurations like: (a) the relation should be cascaded in all operations; (b) the list of authors should be retrieved in the first access to variable; and (c) the list of authors should be ordered by the *order* attribute. This metadata configuration is simpler than implementing this functionality directly connecting to the database, but it is also not trivial and brings a lot of complexity to the class.

```
@Entity
public class Paper implements Serializable {

  @Id
  @GeneratedValue(strategy=SEQUENCE)
  private Integer idPaper;

  @OneToMany(cascade=ALL,
             fetch= LAZY,
             mappedBy="paper")
  @OrderBy("order")
  private List<Author> authors;

  …
}
```

Figure 1. The class Paper mapped with JPA metadata

```
@Entity
public class Author implements Serializable {

  @Id
  @GeneratedValue(strategy=SEQUENCE)
  private Integer idAuthor;

  @Column(name = "order")
  private Integer order;

  @JoinColumn(name = "id_paper",
              referencedColumnName = "id_paper")
  @ManyToOne
  private Paper paper;

  …
}
```

Figure 2. The class Author mapped with JPA metadata

In frameworks that use metadata for event handling management, the presence of an annotation can influence the execution flow in the application. JColtrane [15] is a metadata-based framework for parsing XML documents based on SAX events. The methods of the handler class are annotated with information that defines conditions about when it should be invoked. Parameters are also annotated with information about what information should be passed there by the framework.

Figure 3 presents an example of an annotated method of the framework JColtrane. The annotation *@EndElement*

configures this method to be invoked in the end of the tag <**name**> which contains the attribute **lang** equals to **pt**. The annotation *@BeforeElement* configures the framework to invoke it if only to process a tag <**name**> that is inside of a tag <**paper**>. The method parameters are also configured by annotations and the framework will pass the values based on them. In the Figure 3 example, the parameter *name* will receive the body of the tag <**name**>.

```
@EndElement(
  tag="name",
  attributes={
    @ContainsAttribute(name="lang",value="pt")
  }
)
@BeforeElement(tag="paper")
public void handlePtPaperName(@Body String name){
  …
}
```

Figure 3. Example of an annotated method for XML handling in JColtrane framework.

JColtrane was developed as the case study of a work that has evaluated the use of metadata for granular event handling [16]. For the validation process, a handler for a XML document where created using two different approaches: pure SAX and the framework annotations. Many metrics were used for the comparison, but there is not a metric that consider the complexity added with the metadata.

The metadata used for adding information to a class, do not had influence in that class behavior, but had influence in the frameworks behavior when it received such a class. Metadata can be used for changing business rules, non-functional behavior and even to determine conditions for a method execution. This implicit complexity bought by the metadata should be considered for a realistic measurement of how hard is to maintain, develop or read a source code.

5.   Metadata Create Coupling

The use of metadata can couple the application behavior to a certain framework and it can impact in some maintenance issues. Especially when the metadata is defined externally to the source code, it is not considered in the evaluation of application coupling or modularity.

The use of framework annotations in an application couples its classes to that metadata schema. Tansey & Tilevich [17] report experiences in annotation refactoring for the migration between the frameworks JUnit 3 (which uses code conventions), JUnit 4 and TestNG (which uses annotations). The test class created for one framework do not execute in the other one because the test class is coupled with the metadata schema and the framework can only understand some types of metadata. Figure 4 represents the de-

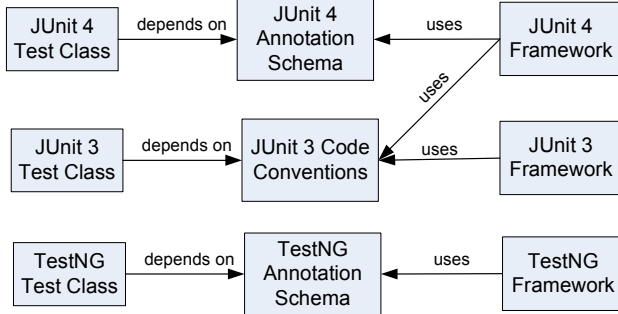pendences among test classes, frameworks, and metadata schema.

Figure 4. Dependences in metadata for testing frameworks

Neto et al. [18] classified dependencies into two different types: syntactic and semantic coupling. Syntactic coupling occurs when one component contains a direct reference to some other component, such as inheritance, method calls, composition and so on. Semantic coupling is a dependency that is not syntactically defined in the code, so that there is no direct reference between the components.

The use of name conventions is a good example of a semantic coupling, because the final application behavior depends on the use of it. Annotations do not influence directly on the functionality of the annotated class and is only additional information attached to it, but it does influence frameworks that had the behavior dependent on that configuration. Thus, the dependence between the annotated class and the framework are also considered by this work as a kind of semantic coupling.

Some metadata described in XML files also can create dependences and an indirect coupling. An example of how it can cause maintenance problems is the development of applications based on the EJB 2.1 specification [19]. In this context, a vendor provides the EJB Container which is responsible to execute and provide functionalities to the EJBs based on metadata configured on two XML descriptors. One of the main objectives of a standard is to provide vendor independence, allowing an EJB application to be deployed on any EJB container that follows the standard.

One of the XML descriptors is defined based on the specification and the other one is different for each container. The descriptors contain metadata relative to the application components (EJBs) for concerns like transaction management and security. Figure 4 represents the dependences among specifications, implementations, and descriptors. The resulting application behavior depends on the interactions among all those elements.
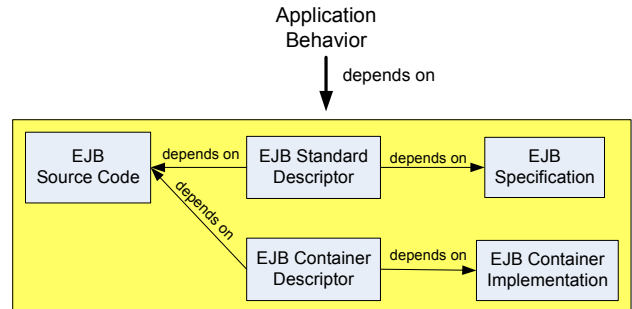

Figure 5. Dependence in EJB 2.1 applications

To migrate between containers there is no need to change the EJB source code, but the metadata format shall be changed. This kind of migration for the EJB 2.1 specification in practice is really difficult because there is a large amount of metadata defined in the container's descriptor that should be migrated to the new format. The recurrence of this problem results in many modifications for the metadata definition in the version 3.0 of this specification [3].

These examples presented show how the metadata definition can create dependences between components in an application. The kind of coupling is semantic and is not represented as an explicit reference in the source code, but it may difficult maintenance and changes among implementations.

6. Metadata Metrics Challenges

The definition of metrics that consider the use of metadata that had influence in the application behavior is important in this context, but it is not an easy task. There are many issues that should be considered for a more realistic evaluation. The following describes some challenges for metadata metrics:

- **Evaluating Metadata Semantic**: Metadata can be used for many purposes in metadata based frameworks, such as for mapping, domain information, and to define when a method should be invoked. Depending on the metadata semantic, different kinds of coupling can be created. The complexity for metadata configuration can also depend on its semantic.
- **Identifying Naming Conventions:** Metadata can be identified directly into annotations and in XML documents, but when naming conventions are used it is harder to identify. The identification of metadata defined in naming conventions is important for the evaluation of coupling between a class and a framework naming rules.
- **Finding Metadata in XML Descriptors:** Frameworks that use XML files to define metadata use different strategies to reference pieces of the source

code, such as classes and methods. This way, the identification of those elements inside the file to count or identify the relations between that artifact and source code is a hard task. The reference of a piece of code in an external file means that the modification of it can impact in this resource.

- **Considering more than one kind of metadata definition.** As described before in this paper, metadata can be defined using different strategies. The measurements about metadata can be used to evaluate the strategy adopted. So, the same metrics should be applied in this evaluation. It is a hard task to think about a common metric to measure metadata defined in XML files, annotations and code conventions.

7. Conclusion

This paper analyzes the complexity and coupling that the usage of metadata-based frameworks brings to an application. The current metrics do not consider metadata definition in the measurements and it can lead to wrong conclusions. Some examples based on frameworks used in the industry shows how the complexity can be increased and how the components can became coupled.

This paper highlights the importance of future works to define metrics that consider the metadata definition. Some issues and challenges for those metrics are also addressed in this work and should be taken into account in assessing applications that uses metadata-based frameworks.

References

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

[2] Bauer, C.; King, G. "Hibernate in Action". Manning Publications, 2004.

[3] "JSR 220: Enterprise JavaBeans 3.0", 2006. Available on http://www.jcp.org/en/jsr/detail?id=220.

[4] "Spring Framework", Available at http://www.springframework.org/, 2009.

[5] Wada, H.; Suzuki, J. "Modeling Turnpike Frontend System: a Model-Driven Development Framework Leveraging UML Metamodeling and Attribute-Oriented Programming". In Proc. of the 8th ACM/IEEE International Conference on Model Driven Engineering Languages and Sytems (MoDELS/UML 2005), 2005.

[6] "XDoclet: Attribute Oriented Programming". Available at http://xdoclet.sourceforge.net/xdoclet/index.html, 2005.

[7] "JSR 175 - A Metadata Facility for the Java Programming Language", 2003. Available at: http://www.jcp.org/en/jsr/detail?id=175.

[8] "Convention over Configuration", Available at: http://softwareengineering.vazexqi.com/files/pattern.html

[9] "JavaBeans Specification 1.01 Final Release". Available at http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html, 1997.

[10] "Ruby on Rails". Available at http://www.rubyonrails.org/, 2009.

[11] Lanza, M.; Marinescu, R. "Object-oriented Metrics in Practice – Using software metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systens". Springer, 2007.

[12] McCabe, T.J. "A measure of complexity". IEEE Transactions on Software Engineering, 2(4):308-320, December, 1976.

[13] Halsted, M. H. "Elements of Software Science". New York, Elsevier Science, 1977.

[14] Arisholm, E.; Briand, L.C.; Foyen, A. "Dynamic coupling measurement for object-oriented software", IEEE Transactions on Software Engineering, Volume 30, Issue 8, Pages 491 –506, Aug. 2004.

[15] "JColtrane – Better than SAX Alone", Available on http://jcoltrane.sf.net, 2009.

[16] Nucitelli, R. "Java Event Handling using Metadata.", Technical Report, Aeronautical Institute of Technology, 2008.[in Portuguese]

[17] Tansey, W.; Tilevich, E. "Annotation Refactoring: Inferring Upgrade Transformations for Legacy Applications", The International Conference on Object Oriented Programming, Systems, Languages and Applications - OOPSLA 2008, Nashville, USA, 2008.

[18] Neto, A. C., de Medeiros Ribeiro, M., D´osea, M., Bonifácio, R., Borba, P., and Soares, S.. "Semantic Dependencies and Modularity of Aspect-Oriented Software". In 1st Workshop on Assessment of Contemporary Modularization Techniques (ACoM'07), in conjunction with the 29th International Conference on Software Engineering, 2007

[19] "JSR 153 – Enterprise JavaBeans 2.1", 2003. Available at: http://www.jcp.org/en/jsr/detail?id=153.