A Pattern Language for Metadata-based Frameworks

Eduardo M. Guerra[1], Jerffeson T. de Souza[2], Clovis T. Fernandes[1]

[1] Aeronautical Institute of Technology, Praça Marechal Eduardo Gomes, 50
Vila das Acácias - CEP 12.228-900 – São José dos Campos – SP, Brazil

[2] Software Patterns Group (GPS.UECE), State University of Ceará (UECE)
Av. Paranjana, 1700 , Campus do Itaperi - CEP 60.740-903, Fortaleza – CE, Brazil

guerra@ita.br, jeff@larces.uece.br, clovistf@uol.com.br

Abstract. *Metadata-based frameworks are those that process their logic based on the metadata of the classes whose instances they are working with. Many recent frameworks use this approach to get a higher reuse level and to be more suitably adapted to the application needs. However, there is not yet a complete best practices documentation or reference architecture for the development of frameworks by using the metadata approach. As a result, this paper presents a pattern language that addresses preliminarily the internal structure of metadata-based frameworks, helping in the understanding and development of such kind of framework.*

1. Context

According to (Beck, 2007), there are three styles of use that a framework can support: instantiation, implementation and configuration. Instantiation is the simplest style of use, where the client instantiate a framework class and use it invoking its methods. In implementation, the client implements a framework interface or extends the framework class to include logic during its execution. In configuration, the client invokes framework methods and passes his own objects to be called at predetermined times.

The implementation style is the one that has the most potential to restrict future design decisions in the framework and the client classes also became tight coupled with the framework structure. Many recent frameworks, in order to avoid these drawbacks, use the configuration style by defining metadata of application's classes. Metadata-based frameworks are frameworks that use class metadata in runtime to process their logic (Guerra and Fernandes, 2008).

Many metadata-based frameworks use attribute-oriented programming, that 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 (Wada and Suzuki, 2005). In Java platform, this programming style has become popular with the native support to code annotations (JSR 175, 2003). But in these frameworks, the metadata can be stored not only by using annotations, but also by using external files (usually as XML documents), databases or programmatically. The metadata can also be configured implicitly using name conventions (Dov, 2006).

Many mature frameworks and APIs used in the industry nowadays are based on metadata, such as Hibernate (Bauer and King, 2004), EJB 3 (JSR 220, 2006) and JUnit

(JUnit, 2008; Beck, 2007). However, some of these frameworks have a few flexibility problems that can difficult or prevent their use in some applications. For instance, the metadata reading mechanism of some frameworks cannot be extended, which makes it difficult to retrieve metadata from other sources or formats. This lack of flexibility can create the need of annotation refactoring (Tansey and Tilevich, 2008), to make easy to migrate among versions or frameworks. Examples of other design concerns are the following: how to make metadata extensible, how the framework logic can be adapted according to class metadata and how the metadata can be shared among frameworks or components.

This paper presents a study that included an analysis in the internal structure of many existing open source metadata-based frameworks. Other frameworks are developed by the author's research group to experiment existent and alternative solutions in different contexts. The objective is to identify best practices both in the understanding and in the development of metadata-based frameworks and to consolidate this design knowledge. This study document the identified design patterns and structure them in a pattern language.

In Guerra et al. (2008b) some isolated patterns for metadata-based components were identified and documented. It classifies the patterns in two categories: structural patterns and application patterns. The structural patterns are related to the internal structure of the component and the application patterns are related to situations in which this kind of solution can be applied. This paper is a continuation of this work, but its scope is limited to structural patterns and the focus is only on frameworks. Some of the structural patterns showed in Guerra et al. (2008b) are split in new patterns of the pattern language proposed in this work.

The main objective of the Pattern Language for Metadata-based Frameworks is to serve as a guide to framework developers that want to use the metadata-based approach. It can also be used as a base to define reference architectures for this kind of frameworks (Buschmann et al., 2007). The running example of the pattern language also presents a complete and detailed example of how to refactor an existent framework to implement the patterns.

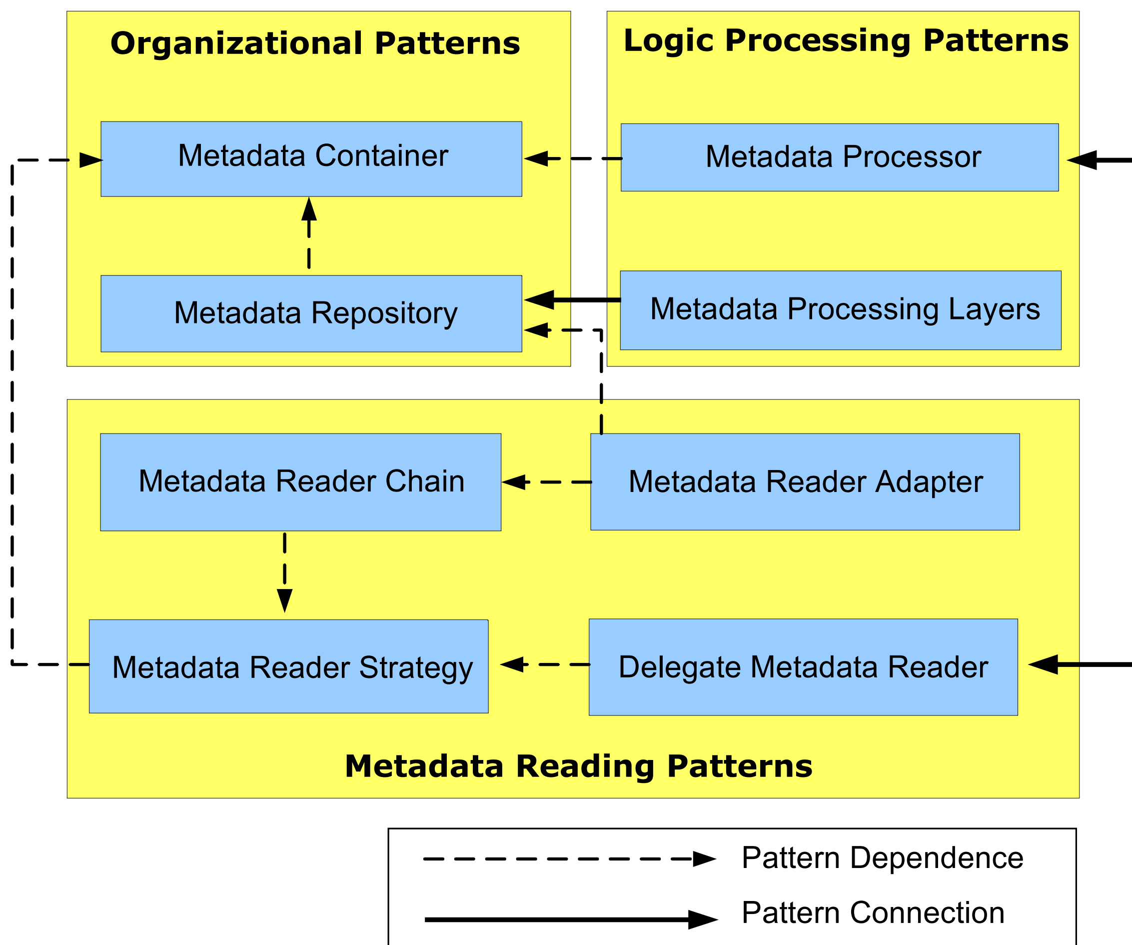
2. Pattern Language Description

This section presents the proposed Pattern Language for Metadata-based Frameworks. Picture 1 illustrates the pattern language structure. The dashed arrows represent pattern dependences, meaning that for the implementation of the dependent pattern, the other must also be implemented. For instance, Metadata Reader Strategy depends on the Metadata Container. The solid arrow represents that there is a connection between the patterns and they are often used together, but they can also be applied independently. For instance, Metadata Processing Layers is connected to Metadata Repository.

The patterns are classified in the following tree different categories:

- **Organizational Patterns -** These patterns document best practices about how to structure internally the classes of a metadata-based framework. They can be considered as the base of the pattern language, since many other patterns depend on their implementation. They are described in section 3.

- **Metadata Reading Patterns -** These patterns document recurrent solutions about the reading of metadata by the framework. They present solutions to improve the flexibility in the reading process, allowing metadata sharing and metadata extension. They are described in section 4.
- **Logic Processing Patterns -** These patterns document solutions in the design of the classes that process the main logic of the framework. They allow the logic processing that is based on the class metadata to be extended and modified. They are described in section 5.



Picture 1 – Pattern Language Structure

It is not in the scope of this pattern language to describe how metadata-based frameworks can be attached to the application architecture. The simplest case is when the framework entry point class is instantiated and used by the application, but it can also be transparent for clients in an application class proxy (Gamma et al, 1994), embedded in a container like in the EJB specification (JSR 220, 2006) or even weaved with an aspect (Guerra et al. , 2008c). How the framework is invoked does not change the best practices in its internal structure. For simplicity, the pattern language will consider, without loss of generality, that the client directly invokes the framework's methods.

2.1. Patterns Overview

The objective of this section is to give an initial idea of the pattern language and how each pattern fits in its context. The following gives a small description of each pattern:

- **Metadata Container -** This pattern decouples the metadata reading from the framework's logic creating a container to represent metadata at runtime. A metadata reader populates the instance of this class, which is used by the framework.
- **Metadata Repository -** This pattern creates a repository to store metadata in runtime, avoiding unnecessary metadata readings. The repository manages the metadata reading and is accessed by the framework to retrieve the metadata container.
- **Metadata Reader Strategy -** This pattern creates an abstraction of the metadata reading algorithm allowing it to have different implementations. It allows the framework to read metadata from different kind of sources.
- **Metadata Reader Chain -** This pattern allows the use of more than one source at the same time to get metadata from classes. It uses a composite metadata reader that invokes other readers to compose the reading algorithm.
- **Metadata Reader Adapter -** This pattern uses the metadata repository of another framework to get the existent metadata and set information in the metadata container. It allows metadata sharing among frameworks.
- **Delegate Metadata Reader -** In this pattern, the metadata reader delegate to other classes the responsibility to interpret pieces of metadata defined in annotations or XML elements. It allows the metadata schema to be extended by the application.
- **Metadata Processor -** In this pattern, the metadata container is composed of classes that maintain logic to process pieces of metadata. By implementing these class abstractions, it is possible to extend the framework behavior.
- **Metadata Processing Layers -** In this pattern, the logic processing is composed of many different layers with different responsibilities. It allows each layer to evolve independently and also the framework functionality to be extended by adding new layers.

2.2. Running Example

The pattern language is illustrated in this work by means of a running example featuring the Comparison Component, a framework that compare instances of the same class and returns a list of differences among them. It will compare all the application class properties searching for differences. The metadata will be used to configure characteristics in the comparison algorithm for each class, such as properties to ignore, numeric tolerance and objects to go deep into the comparison.

The comparison component provides tree annotations to configure how the classes should be compared: *@Ignore* annotates properties that should not be included in the

comparison; *@Tolerance* annotates numeric properties that should have a numeric tolerance to be considered; *@DeepComparison* annotates compound properties that should be compared using the comparison component itself. Listing 1 presents an example of a class with these annotations.

```
public class Person{
    private String name;
    private double weight;
    private int age;
    private Address address;
    public Person(String name, float weight, int age) {
        this.name = name;
        this.weight = weight;
        this.age = age;
    }
    @DeepComparison
    public Address getAddress() {
        return address;
    }
    public String getName() {
        return name;
    }
    @Tolerance(0.1)
    public double getWeight() {
        return weight;
    }
    @Ignore
        public int getAge() {
        return age;
    }

    //setter methods omitted
}
```

Listing 1 – Example of a class with Comparison Component's annotations.

The framework entry point, represented by the class *ComparisonComponent*, has only one public method called *compare()*. This method receives two instances of the same class, considering that they are different versions of the same business entity. This method also returns an instance list of *Difference*, which is a data class that has three properties representing the property name, the property value in the new instance and the property value in the old instance. Listing 2 presents a sample code that compare two instances and prints the differences in the console.

Listing 3 presents the initial implementation of the Comparison Component without any pattern implementation. In each pattern, this implementation will be refactored to increase its flexibility and, in some cases, enabling the creation of new functionalities.

```
Person p1 = new Person("John",70.5f,20);
Address e1 = new Address("Street Road","50");
p1.setAddress(e1);
Person p2 = new Person("John B.",70.6f,21);
Address e2 = new Address("Street Road","55");
p2.setAddress(e2);
ComparisonComponent c = new ComparisonComponent();
List<Difference> difs = c.compare(p2, p1);
for(Difference d : difs){
    System.out.println(d.getProperty() +":" +d.getOldValue() +"/" +d.getNewValue());
}
```

Listing 2 – The use of Comparison Component.

```
public class ComparisonComponent {
    public List<Difference> compare(Object oldObj, Object newObj)
            throws CompareException {
        List<Difference> difs = new ArrayList<Difference>();
        if (!newObj.getClass().isAssignableFrom(oldObj.getClass())) {
            throw new CompareException("Not compatible types");
        }
        Class clazz = newObj.getClass();
        for (Method method : clazz.getMethods()) {
            try {
                boolean isGetter = method.getName().startsWith("get");
                boolean noParameters = (method.getParameterTypes().length == 0);
                boolean notGetClass = !method.getName().equals("getClass");
                boolean noIgnore = !method.isAnnotationPresent(Ignore.class);
                if (isGetter && noParameters && notGetClass && noIgnore) {
                    Object oldValue = method.invoke(oldObj);
                    Object newValue = method.invoke(newObj);
                    String propName = method.getName().substring(3, 4).toLowerCase()
                            + method.getName().substring(4);

                    if (method.isAnnotationPresent(Tolerance.class)) {
                        Tolerance tolerance = method.getAnnotation(Tolerance.class);
                        compareWithTolerance(difs, tolerance.value(),
                                newValue, oldValue, propName);
                    } else if (method.isAnnotationPresent(DeepComparison.class)
                            && newValue != null && oldValue != null) {
                        List<Difference> difsProp = compare(newValue, oldValue);
                        for (Difference d : difsProp) {
                            d.setProperty(propName + "." + d.getProperty());
                            difs.add(d);
                        }
                    } else {
                        compareRegular(difs, propName, newValue, oldValue);
                    }
                }
            } catch (Exception e) {
                throw new CompareException("Error retrieving property", e);
            }
        }
        return difs;
    }
    private void compareWithTolerance(List<Difference> difs, double tolerance,
            Object newValue, Object oldValue, String prop) {
        double dif = Math.abs(((Double) newValue) - ((Double) oldValue));
        if (dif > tolerance) {
            difs.add(new Difference(prop, newValue, oldValue));
        }
    }
    private void compareRegular(List<Difference> difs, String prop,
            Object newValue, Object oldValue) {
        if (newValue == null) {
            if (oldValue != null) {
                difs.add(new Difference(prop, newValue, oldValue));
            }
        } else if (!newValue.equals(oldValue)) {
            difs.add(new Difference(prop, newValue, oldValue));
        }
    }
}
```

Listing 3 – The initial source code for ComparisonComponent.

The implementation of the *compare()* method in Listing 3 first verify if both instances are from the same class. After that, it retrieves all getter methods from the class and compare the values retrieved from both instances based on the metadata. The private methods *compareRegular()* and *compareWithTolerance()* are used by the *compare()* method respectively for the regular comparison and for the comparison using tolerance.

3. Organizational Patterns

3.1. Metadata Container

Some frameworks that use metadata read this information during the logic execution. It difficult changes in the metadata schema and definition form. It also disable the sharing of the obtained information with other classes.

Problem

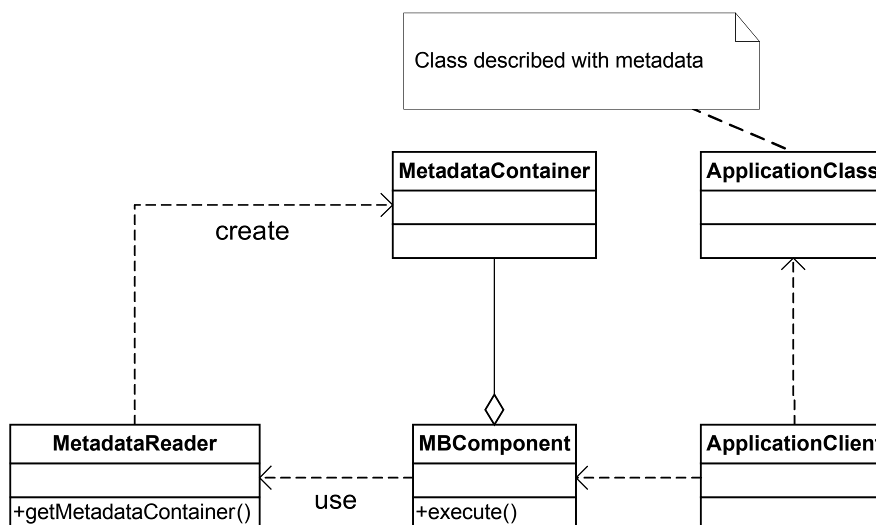
How to decouple the metadata reading from the framework's main logic?

Forces

- The reading of metadata mixed with the framework main logic sometimes is easier, but it makes it difficult the use of metadata from other sources.
- The reading of metadata in all framework invocations can make it stateless, but it can become a performance bottleneck.
- The existence of an internal representation of the metadata can consume more runtime memory, that is a problem specially in embedded applications, but it can enable the metadata sharing among frameworks.
- The metadata format coupled with the framework's main logic is easier to implement, but can disable the application of other patterns that can make flexible the metadata reading and enable the extension of the metadata schema.

Structure

In **Metadata Container**, the application class metadata is stored in an instance that represents it at runtime. Picture 2 represents the pattern structure. The *MetadataContainer*, that is the class that represents the metadata structure, is the main interface among a class that reads the metadata wherever it is defined, the *MetadataReader*, and the class that contains the main logic, the *MBComponent*.

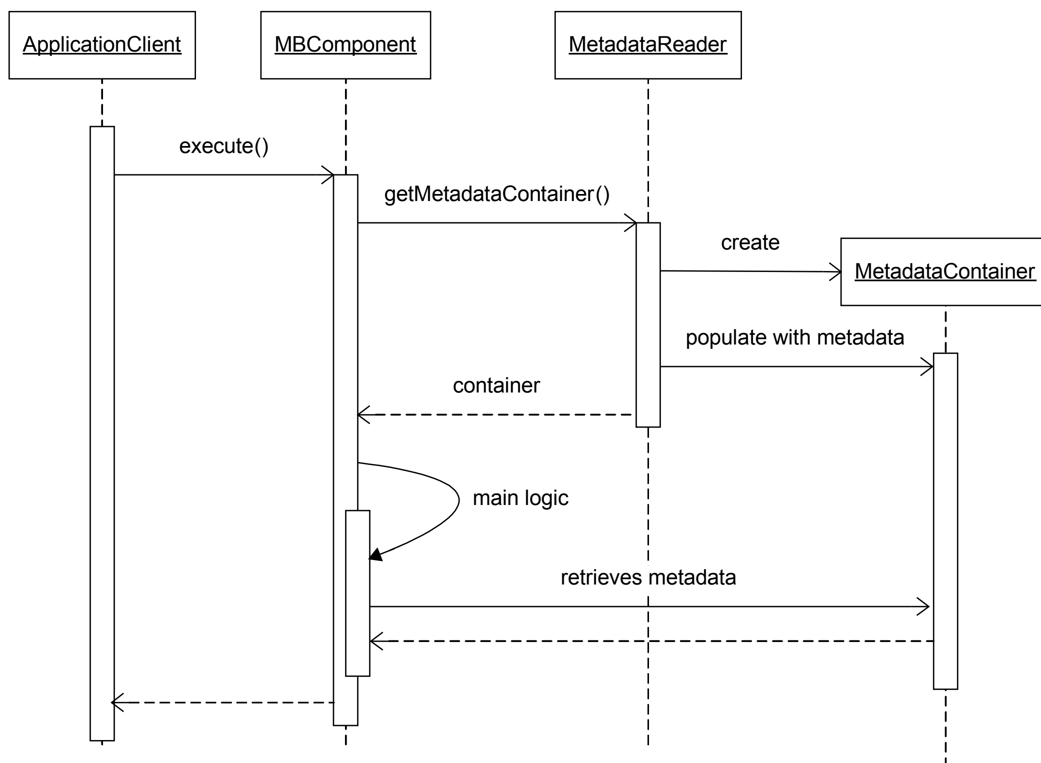


Picture 2 – Metadata Container structure

The *MBComponent* can store the *MetadataContainer* instance internally as represented in the diagram. This is usually done when the class with the metadata is received by the *MBComponent* in the constructor. Alternatively, the *MBComponent* can be stateless and accept invocation with any class, invoking the *MetadataReader* in each one.

When the application client accesses the entry point of the metadata-based framework, it invokes the *MetadataReader* that reads the metadata and returns an instance of the *MetadataContainer* populated with the class metadata. In the execution of the main logic, the *MBComponent* accesses the required metadata by using the *MetadataContainer* instance. The correspondent sequence diagram is represented in Picture 3.

As an alternative implementation of this pattern, the *ApplicationClient* can create the *MetadataContainer* instance explicitly. Using this approach, the client uses the *MetadataReader* to create the *MetadataContainer* and passes it to the *MBContainer* in its constructor or as a parameter in a method.



Picture 3 – Metadata Container sequence diagram

Participants

- **MetadataReader -** It is in charge of reading the metadata wherever it is defined. It is used by *MBComponent* to retrieve an instance of *MetadataContainer* representing the metadata of an *ApplicationClass*.

- **MetadataContainer -** It is responsible for representing the metadata of an *ApplicationClass* at runtime. It is the main interface among *MetadataReader* and *MBComponent*.
- **MBComponent -** It is the framework entry point. It is responsible for executing the main logic and for being a controller of the other classes. It uses the data of *MetadataContainer*, retrieved from *MetadataReade*r, to execute its logic.
- **ApplicationClient -** It represents the application class that invokes, implicitly or explicitly, *MBComponent*.
- **ApplicationClass -** It represents the application class that is described by the metadata used by *MBComponent* in its main logic.

Consequences

- The metadata can be shared with other frameworks by the MetadataContainer instance, enabling metadata sharing.
- The framework became more testable, enabling metadata reading and the logic processing to be tested separately.
- The MBComponent can reuse the same MetadataContainer instance, avoiding unnecessary metadata reading and improving the application performance.
- As to frameworks that use a small amount of metadata, the use of this pattern can complicate unnecessarily its structure.

Known Uses

Hibernate is a framework that uses metadata for object-relational mapping for implementing a persistent layer (Bauer and King, 2004). The metadata can be defined in a XML file or using annotations. It uses, as a *MetadataContainer*, an implementation of the interface *ClassMetadata* that can be retrieved from a *SessionFactory* instance. The *SessionFactory* is the class responsible for creating instances of *Session*, which is the class that the application uses to interact with the framework. *ClassMetadata* contains all the information about a persistent class that is used by the framework.

The SwingBean framework provides graphical components for creating forms and tables for a Swing desktop application in Java (SwingBean, 2009). These components use the metadata of an application domain class defined in a XML document to configure how the information should be retrieved or presented. The class *XMLDescriptorFactory* provides a **Facade** (Gamma et al, 1994) for the client to retrieve the metadata, represented by the *FieldDescriptor* interface. The client uses this instance to create the graphical components, passing it as an argument in the constructor.

The JAXB API (JSR 222, 2006) is a Java standard for XML binding, which uses annotations on application classes to map them to the target XML Schema. The API itself does not provide an interface that allows the client to retrieve the metadata information, but by examining the internal structure of its reference implementation, the structure of a **Metadata Container** can be identified. The interfaces *ClassInfo* and *Element* represent the metadata at runtime and are used by the internal logic of the component.

Running Example

To increase its flexibility, the Comparison Component, defined earlier in the Running Example (section 2.1), is refactored to decouple the metadata reading from the comparison logic. This step is necessary to allow the implementation of metadata reading from other sources and the extension of the components logic.

Listings 4 and 5 represents respectively the classes *PropertyDescriptor* and *ComparisonDescriptor*, which are the metadata containers for this component. The *PropertyDescriptor* provides information about one property, such as property name, its tolerance and whether it should be compared "deeply". The class *ComparisonDescriptor* has a map with the properties that should be included in the comparison and the respective *PropertyDescriptor* instances.

```
public class PropertyDescriptor {

    private String name;
    private double tolerance;
    private boolean deepComparison;

    //getters and setters omitted
}
```

Listing 4 – Source code of PropertyDescriptor.

```
public class ComparisonDescriptor {

    private Map<String,PropertyDescriptor> properties =
        new HashMap<String, PropertyDescriptor>();

    public void addPropertyDescriptor(PropertyDescriptor descProp){
        properties.put(descProp.getName(), descProp);
    }
    public PropertyDescriptor getPropertyDescriptor(String prop){
        return properties.get(prop);
    }
    public Set<String> getProperties(){
        return properties.keySet();
    }
}
```

Listing 5 – Source code of ComparisonDescriptor.

The class *ComparisonMetadataReader* is represented in Listing 6. The *createContainer()* receives a class as a parameter and returns the respective instance of *ComparisonDescriptor* created using the annotations in the properties getter methods. When a property presents the annotation *@Ignore*, it is not included in the *ComparisonDescriptor* instance. The information provided by other annotations are obtained and stored in the descriptor.

Listing 7 illustrates the *ComparisonComponent*, which is the class that actually has the comparison logic. It retrieves the *ComparisonDescriptor* from the *ComparisonMetadataReader* and compares the two instances based on the metadata. It uses the Reflection API (Forman and Forman, 2004) to retrieve the properties´ values and, based on the information contained on the respective *PropertyDescriptor*, delegates the comparison to one of the methods *compareWithTolerance()*, *compareRegular()* or the method *compare()*, in case of deep comparison.

```
public class ComparisonMetadataReader {
    public ComparisonDescriptor createContainer(Class c) {
        ComparisonDescriptor descr = new ComparisonDescriptor();
        for (Method method : c.getMethods()) {
            boolean isGetter = method.getName().startsWith("get");
            boolean noParameters = (method.getParameterTypes().length == 0);
            boolean notGetClass = !method.getName().equals("getClass");
            boolean noIgnore = !method.isAnnotationPresent(Ignore.class);
            if (isGetter && noParameters && notGetClass && noIgnore) {
                PropertyDescriptor prop = new PropertyDescriptor();
                String getter = method.getName();
                String propName = getter.substring(3, 4).toLowerCase()
                + getter.substring(4);
                prop.setName(propName);
                prop.setDeepComparison(method
                        .isAnnotationPresent(DeepComparison.class));
                if (method.isAnnotationPresent(Tolerance.class)) {
                    Tolerance t = method.getAnnotation(Tolerance.class);
                    prop.setTolerance(t.value());
                }
                descr.addPropertyDescriptor(prop);
            }
        }
        return descr;
    }
}
```

Listing 6 – Source code of ComparisonDescriptorReader.

```
public class ComparisonComponent {
    protected ComparisonMetadataReader reader;
    public ComparisonComponent() {
        this.reader = new ComparisonMetadataReader();
    }
    public List<Difference> compare(Object oldObj, Object newObj)
    throws CompareException {

        List<Difference> difs = new ArrayList<Difference>();

        if (!newObj.getClass().isAssignableFrom(oldObj.getClass()))
            throw new CompareException("Not compatible types");
        ComparisonDescriptor descr = reader.createContainer(newObj.getClass());

        for (String prop : descr.getProperties()) {
            try {
                String getterName = "get" + prop.substring(0, 1).toUpperCase()
                    + prop.substring(1);
                Method method = newObj.getClass().getMethod(getterName);
                Object oldValue = method.invoke(oldObj);
                Object newValue = method.invoke(newObj);
                PropertyDescriptor descProp = descr.getPropertyDescriptor(prop);

                if (descProp.getTolerance() != 0) {
                    compareWithTolerance(difs, descProp.getTolerance(),
                                         newValue, oldValue, prop);
                } else if (descProp.isDeepComparison() && newValue != null
                        && oldValue != null) {
                    List<Difference> difsProp = compare(newValue, oldValue);
                    for (Difference d : difsProp) {
                        d.setProperty(prop + "." + d.getProperty());
                        difs.add(d);
                    }
                } else {
                    compareRegular(difs, prop, newValue, oldValue);
                }
            } catch (Exception e) {
                throw new CompareException("Error retrieving property", e);
            }
        }
        return difs;
    }
    //compareWithTolerance and compareRegular had the same implementation of Listing 3
}
```

Listing 7 – Source code of ComparisonComponent.

Related Patterns

Metadata Container is the base of the proposed pattern language. In Picture 1, one can observe that all patterns depend directly or indirectly on it. The rationale behind this pattern is that the decoupling between the logic and the metadata reading provides a structure that allows each part to evolve independently.

This pattern is also related to **Data Accessor** (Nock, 2003), which encapsulates physical data access details in a single component, decoupling data access responsibilities. **Metadata Container** uses the same principle to decouple the metadata reading, but not necessarily from physical data.

3.2. Metadata Repository

The main class of some frameworks are instantiated many times in the same application and is not desirable that the same metadata was read more than one time. The applications also need a way to control the moment for the metadata reading, choosing a lazy loading or a pre-loading strategy.

Problem

How to provide a central place to store metadata and manage the metadata reading?

Forces

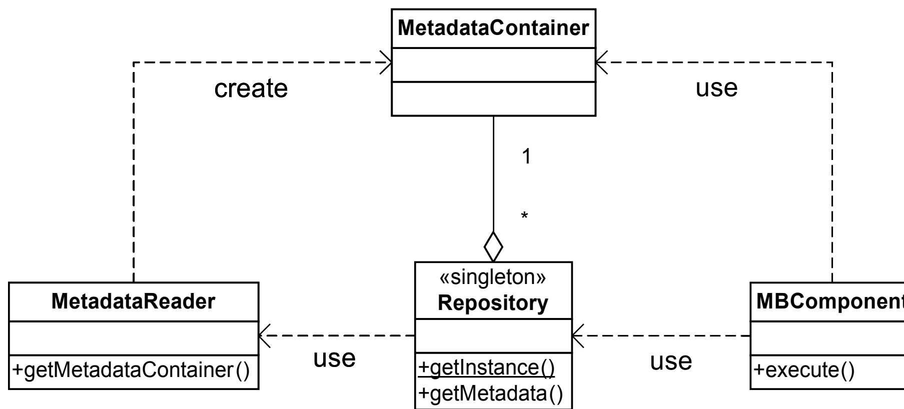
- Having a singleton instance of the framework entry point can be easier in some architectures, but for other circumstances, like when the class is a proxy, it is not possible.
- Having a central place to retrieve metadata can ease the metadata sharing, but it consumes more runtime memory.
- Lazy loading the metadata can avoid unnecessary metadata readings and can slow the execution when the metadata is retrieved, but pre-loading the metadata can accelerate the execution and can slow down the application initialization.

Structure

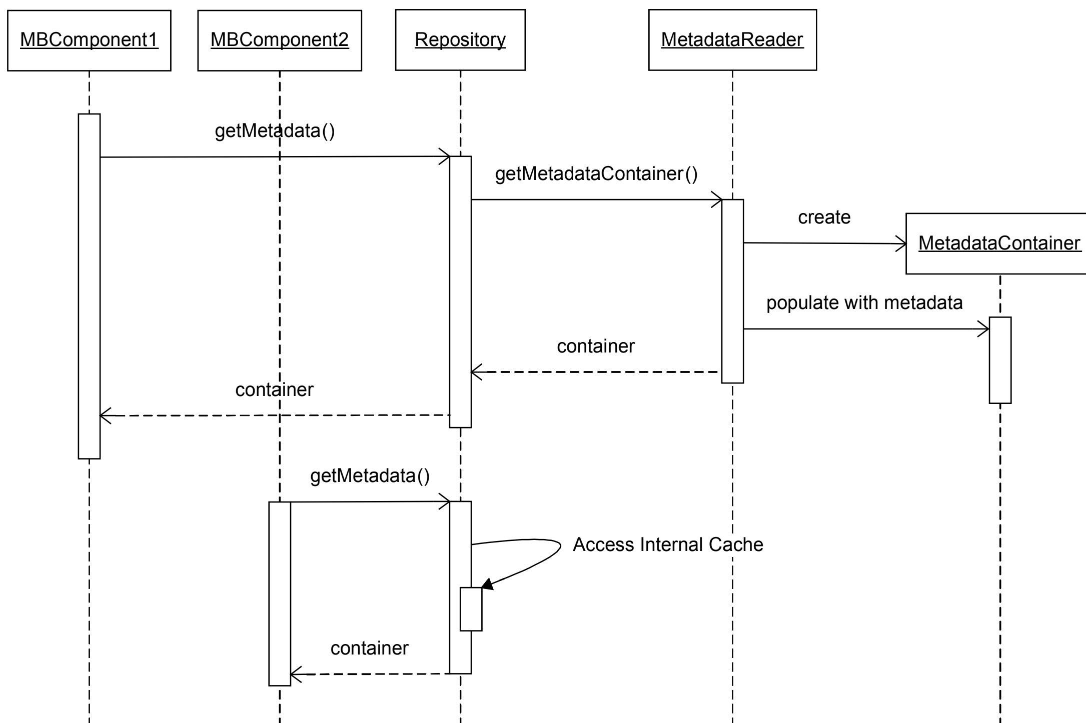
In **Metadata Repository**, a **Singleton** class (Gamma et al, 1994) is responsible for managing the metadata reading and storing internally the class metadata. In the first reading of metadata, the *Repository* caches the information and makes it available to any other component that needs it.

In Picture 4, a class diagram of the pattern is presented. In this structure, *MBComponent* does not access directly *MetadataReader*. All of the metadata accesses occurs via *Repository*, which has a common base of metadata, represented by instances of *MetadataContainer*, which is shared between all *MBComponen*t instances.

The sequence diagram represented in Picture 5 shows the first access to metadata of one class, and then the access from another component to the metadata of the same class. In the first access, *Repository* collaborates with *MetadataReader* to retrieve the metadata represented by an instance of *MetadataContainer*. In the second access, the metadata is already stored inside *Repository* and is returned without an additional reading.



Picture 4 – Metadata Repository structure



Picture 5 – Sequence diagram representing two instances of MBComponent accessing the repository to retrieve metadata of the same class.

Participants

- **MetadataReader -** It is responsible for reading the metadata wherever it is defined. It is used by *Repository* to retrieve instances of *MetadataContainer* and store it internally.
- **MetadataContainer -** It is responsible for representing the metadata of an application class at runtime. It is stored internally by using *Repository*.
- **MBComponent -** It is the framework entry point. It is responsible for executing the main logic and for being a controller of the other classes. It uses *Repository* to retrieve metadata represented in instances of *MetadataContainer*.

- **Repository -** It is responsible for managing the access to *MetadataReader* and storing internally the instances of *MetadataContainer*. It is a singleton and provides metadata to all instances of *MBComponent*.

Consequences

- The unnecessary metadata readings can be avoided, improving the application performance.
- The application can use the repository to load all metadata when the application starts or to load it only when the application needs them.
- The metadata has a central point to be accessed by other frameworks, facilitating to share metadata.
- For a component that has only one instance shared by all the application, the creation of a repository may be unnecessary.
- For applications that can change the metadata at runtime, it is necessary to control the access in the *MetadataContainer* for concurrent modification.
- The storage of metadata can increase the use of runtime memory by the application.

Known Uses

The Hibernate framework provides the class *SessionFactory*, which is responsible for creating *Session* instances (Bauer and King, 2004). The *Session* class provides an API for the application to persist and retrieve entities from the database. The *SessionFactory* has a repository encapsulated and provides methods to retrieve metadata. Each instance of *Session* created receives a reference to *SessionFactory*, a central place to retrieve the metadata for all instances.

The SwingBean framework provides static methods to encapsulate the metadata reading in the class *XMLDescriptorFactory* (SwingBean, 2009). The methods of this class encapsulates an access to a singleton map that acts as a repository for the framework. This structure allows the framework to have a mechanism to load in a low priority background thread the metadata for the graphical components creation, to improve the application performance.

XapMap, which stands for Cross Application Mapping, is a framework that maps entities of the same domain but implemented in different class structures (XapMap, 2009). It provides a component that creates an instance of one schema based on an instance of the other schema and also provides a proxy that encapsulates the access of an instance of one schema based on the other schema's API. Both components of the framework access the metadata through a singleton repository, which guaranties that the metadata will not be read again for the same class.

Running Example

The application that uses the Comparison Component creates an instance of *ComparisonComponent* and then calls the *compare()* method to get the differences between two instances of the same class. According to the implementation presented in Listing 7, whenever the *compare()* method is called, a new metadata reading occur.

To avoid the probably lost of performance that unnecessary metadata readings can cause, the **Metadata Repository** pattern is applied to the Comparison Component. Listing 8 represents the *Repository* class, which is a singleton, and provides metadata to all *ComparisonComponent* instances. Listing 9 represents changes in the *MBComponent* to use the class *Repository* instead of accessing directly the *ComparisonMetadataReader*.

```
public class Repository {

    private static Repository instance;

    public static Repository getInstance(){
        if(instance == null){
            instance = new Repository();
        }
        return instance;
    }

    private ComparisonMetadataReader reader;
    private Map<Class, ComparisonDescriptor> cache;

    private Repository(){
        reader = new ComparisonMetadataReader();
        cache = new HashMap<Class, ComparisonDescriptor>();
    }
    public ComparisonDescriptor getMetadata(Class clazz){
        if(cache.containsKey(clazz)){
            return cache.get(clazz);
        }
        ComparisonDescriptor cd = reader.createContainer(clazz);
        cache.put(clazz, cd);
        return cd;
    }
}
```

Listing 8 – Source code of Repository.

```
public class ComparisonComponent {

    public List<Difference> compare(Object oldObj, Object newObj)
    throws CompareException {

        List<Difference> difs = new ArrayList<Difference>();

        if (!newObj.getClass().isAssignableFrom(oldObj.getClass()))
            throw new CompareException("Not compatible types");
                ComparisonDescriptor descr = Repository.getInstance().
                  getMetadata(newObj.getClass());

        //the same as Listing 4
    }

    //the same as Listing 4

}
```

Listing 9 – Changes in MBComponent source code.

Related Patterns

Metadata Repository is an important pattern to allow metadata sharing. The **Metadata Reader Adapter** uses the repository of another component to read and adapt the metadata format. The **Metadata Processing Layers** divides the logic of the metadata-

based component in many processing layers and the use of **Metadata Repository** is recommended to provides only one place to access metadata by each layer.

This pattern is also related to the patterns **Cache Accessor** and **Demand Cache** (Nock, 2003), which provides a structure for caching data retrieved from a data source. The **Metadata Repository** can be considered a specialization of those patterns to the context of a metadata-based component. Other **Cache Patterns** can also be applied in the implementation of a metadata repository.

4. Metadata Reading Patterns

4.1. Metadata Reader Strategy

Different applications that uses the framework may have different needs about the metadata definition. Code annotations, XML, code conventions and database are some options to define metadata.

Problem

How to provide a structure to allow metadata reading from different sources?

Forces

- The framework usually provides only one way to define metadata, but the application can already have the same information in other sources.
- The use of external sources of metadata definition can make it modifiable at deployment time, but the use of code annotations make the definition closer to the class.
- The use of annotations may be easier in application classes, but sometimes it is unfeasible to use them in legacy classes.
- As to some applications, one metadata definition approach may be appropriate for a set of classes, but other ways to define metadata might be preferable for other ones.

Structure

The **Metadata Reader Strategy** is a specialization of the **Strategy** (Gamma et al., 1994). An interface is used to abstract the reading of metadata and different classes can implement it to read metadata from different sources. A **Singleton** (Gamma et al., 1994) is used to retrieve the correct metadata reader instance.

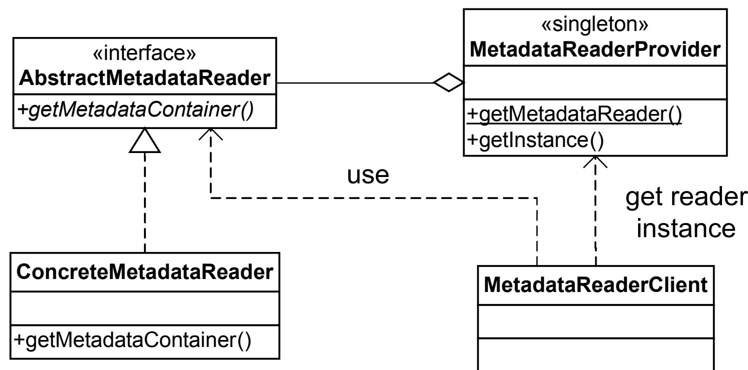
The structure of **Metadata Reader Strategy** is presented in Picture 6. The interface *AbstractMetadataReader* abstracts the reading of metadata and is implemented by any *ConcreteMetadataReader*. The *MetadataReaderClient* represents the class that needs to read metadata. If the component uses the structure of **Metadata Container** the client will be *MBComponent*. Otherwise, if it uses **Metadata Repository** structure the client will be *Repository*.

The client uses *MetadataReaderProvider* to retrieve the correct instance of *AbstractMetadataReader*. This class can have some logic that returns different readers

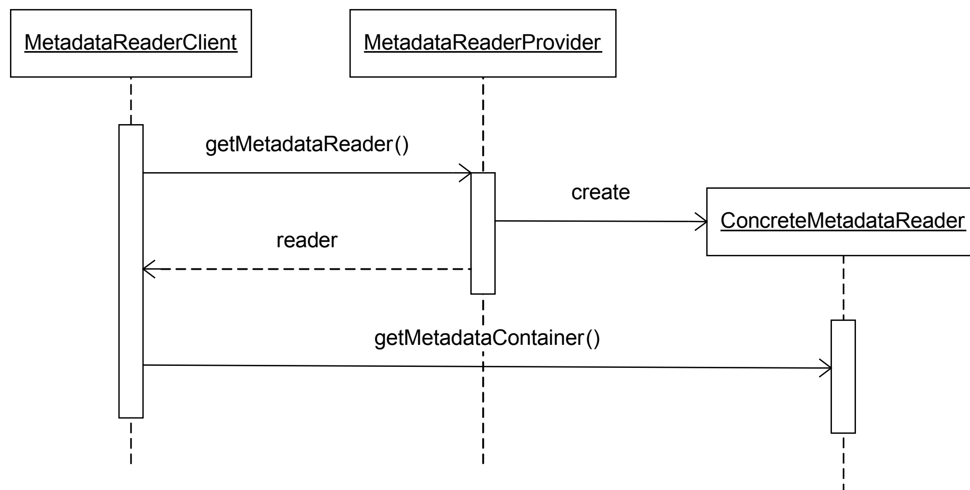
to read metadata from different classes. It is very useful if different sets of classes in the application requires different readers.

As an alternative implementation, the *MetadataReaderProvider* is not necessary if the client passes the *ConcreteMetadataReader* instance as a parameter. The need for the *MetadataReaderProvider* can also be eliminated by the use of dependence injection (Fowler, 2004) on the *MetadataReaderClient*.

Picture 7 shows a sequence diagram that represents how the *MetadataReaderClient* retrieves the *MetadataContainer* in this structure.



Picture 6 – The structure of Metadata Reader Strategy.



Picture 7 – Sequence diagram representing how MetadataReaderClient retrieves a MetadataContainer from a reader.

Participants

- **AbstractMetadataReader -** It represents an abstraction of a metadata reader. It is implemented by any *ConcreteMetadataReader*.
- **ConcreteMetadataReader -** It is responsible for reading metadata from one source and implements the interface of *AbstractMetadataReader*.

- **MetadataReaderClient -** It represents a *Repository* or a *MBComponent*, depending on the component's structure. It is the class that needs to directly retrieve metadata from a reader. It is the class that is coupled only with *AbstractMetadataReader*, not with one specific implementation.
- **MetadataReaderProvider -** It is responsible for providing the correct *ConcreteMetadataReader* instance that the *MetadataReaderClient* should use for each application class.

Consequences

- The framework can provide more than one alternative for metadata definition for the applications to use.
- The application can create new approaches for metadata definition, extending the framework's metadata reading strategy.
- The *MetadataReaderProvider* can return different readers for different sets of classes in one application.
- In cases in which one approach for metadata definition is enough, the use of this pattern can over-design the component.

Known Uses

Early versions of Hibernate framework support metadata definition only using XML (Bauer and King, 2004). The class *Configuration* is used to setup the configuration files and create the *SessionFactory* instance. The project Hibernate Annotations is a separated release that gives to the framework support to annotations. The class *AnnotationConfiguration*, which extends *Configuration*, creates the same *SessionFactory* instance by using instead annotations as the metadata source.

The ACE framework uses metadata to map functionalities of a web application to mobile applications (Costa and Figueredo, 2009). This framework supports this mapping for new applications, using annotations, and for legacy applications, using XML files. As a result, it uses an interface to abstract the metadata reading and has implementations for get it both from annotations and from XML files.

MentalLink is a framework for mapping between instances of an ontology and objects (MentalLink, 2008; Guerra et al., 2008c). It supports only annotations, but it uses the structure of this pattern to allow in the future the extension of the metadata reading by the application.

Running Example

The application that uses the Comparison Component now needs to compare classes from a legacy application in which the developers do not have access to modify the source code. One solution to this problem is to provide the Comparison Component with a facility to define metadata by using XML files.

Before implementing the reading from XML files, the framework is refactored to implement the **Metadata Reader Strategy**. The *ComparisonMetadataReader* class is renamed to *AnnotationComparisonMetadataReader* and an interface, named

ComparisonMetadataReader, is extracted to generalize the concept of metadata reading. The interface extracted is represented in Listing 10.

Listing 11 presents the *MetadataReaderProvider* class. It is a **Singleton** class that returns an instance of the configured *ComparisonMetadatReader*. It has two static methods that encapsulate the access to the singleton instance to set and to get the metadata reader instance. The access to the metadata factory also should be refactored in the *Repository* or in the *ComparisonComponent*.

```
public interface ComparisonMetadataReader {

    public abstract ComparisonDescriptor createContainer(Class c);

}
```

Listing 10 – The ComparisonMetatadaReader interface.

```
public class MetadataReaderProvider {

    private static MetadataReaderProvider provider;

    public static MetadataReaderProvider getProvider(){
        if(provider == null){
            provider = new MetadataReaderProvider();
        }
        return provider;
    }

    private ComparisonMetadataReader reader;

    private MetadataReaderProvider(){
        //set the default implementation
        reader = new AnnotationComparisonMetadataReader();
    }
    public void setReader(ComparisonMetadataReader reader){
        this.reader = reader;
    }
    public ComparisonMetadataReader getReader(){
        return reader;
    }

    //ease the access to configured reader
    public static void set(ComparisonMetadataReader reader){
        getProvider().setReader(reader);
    }
    public static ComparisonMetadataReader get(){
        return getProvider().getReader();
    }
}
```

Listing 11 – MetadataReaderProvider source code.

After refactoring, the metadata reading using XML files can be implemented and used by the Comparison Component. Listing 12 presents an example of a XML document for defining metadata for a class. For simplicity, the metadata of a class is considered to be stored in a XML file with the same name of the class.

The *XMLComparisonMetadataReader* source code is presented in Listing 13. It uses JColtrane framework (JColtrane, 2009), which is based in SAX parsing (SAX, 2004), to read the XML files. JColtrane uses annotations for the SAX event management. Listing 14 presents the XML handler that reads metadata and populates the *ComparisonDescriptor*.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<comparison>
   <prop name="name"/>
   <prop name="weight" tolerance="0.1" />
   <prop name="address" deep="true" />
</comparison>
```

Listing 12 – An example of the metadata definition in XML.

```java
public class XMLComparisonMetadataReader implements ComparisonMetadataReader{
    @Override
    public ComparisonDescriptor createContainer(Class c) {
        try {
            SAXParser parser= SAXParserFactory.newInstance().newSAXParser();
            File file=new File(c.getSimpleName()+".xml");
            ComparisonXMLHandler handler = new ComparisonXMLHandler();
            parser.parse(file,new JColtraneXMLHandler(handler));
            return handler.getDescriptor();
        } catch (Exception e) {
            throw new RuntimeException("Can't read metadata",e);
        }
    }
}
```

Listing 13 – The implementation of XMLComparisonMetatadaReader.

```java
public class ComparisonXMLHandler {

    private ComparisonDescriptor descriptor;

    @StartDocument
    public void init(){
        descriptor = new ComparisonDescriptor();
    }

    @StartElement(tag="prop")
    public void addProperty(
               @Attribute("name") String name,
               @Attribute("tolerance") Float tolerance,
               @Attribute("deep") Boolean deep){
        PropertyDescriptor pd = new PropertyDescriptor();
        pd.setName(name);
        if(tolerance != null)
            pd.setTolerance(tolerance);
        if(deep != null)
            pd.setDeepComparison(deep);
            descriptor.addPropertyDescriptor(pd);
    }

    public ComparisonDescriptor getDescriptor() {
        return descriptor;
    }
}
```

Listing 14 – The handler to interpret XML files using JColtrane framework.

Related Patterns

Metadata Reader Strategy provides the flexibility in the achievement of metadata required by other patterns in this pattern language. **Metadata Reader Chain** and **Metadata Reader Adapter** are patterns based on the fact that it is possible to change the algorithm for reading metadata.

This pattern is a specialization of **Strategy** (Gamma at al, 1994) to help to solve the specific problem of metadata reading. It is also related to **Data Access Object**, also known as **DAO** (Alur at al., 2003), which is used to encapsulate data access and

manipulation in a separate layers. Using **DAO**, it is possible to have different implementations for retrieving data from different data sources.

4.2. Metadata Reader Chain

The annotations is an easy way to define metadata but the use of XML can enable adjusts in the metadata at deploy time. The use of more than one source of metadata at the same time by the framework can enable benefits that are not possible with the use of only one.

Problem

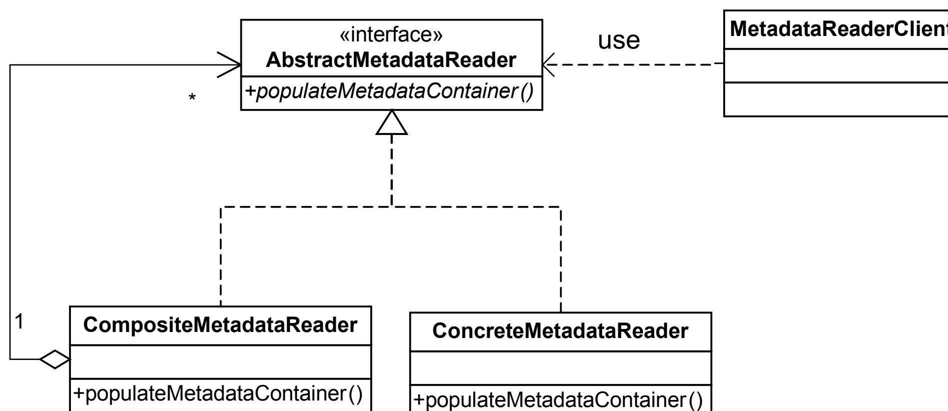
How to provide a structure to allow metadata to be read from more than one source?

Forces

- Each kind of metadata definition has its benefits and drawbacks, but some applications crave to take advantage of the benefits from more than one definition type.
- Some metadata can be retrieved partially from alternative sources, but it must be complemented with more information.
- An order can be defined for reading metadata from more than one source, but some applications may need to use a different order.
- Code annotations are useful and makes it easy to define metadata, but it can be complemented by a XML document that can be changed at deployment time.

Structure

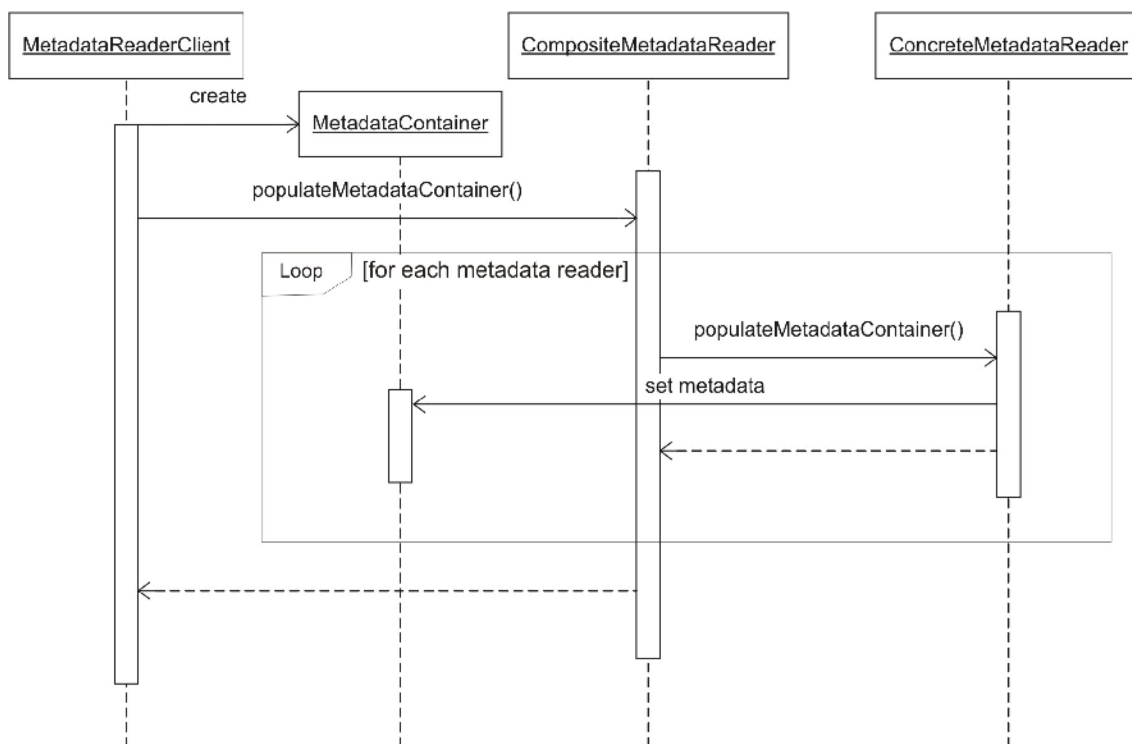
The **Metadata Reader Chain** provides a structure that allows the class metadata to be read from more than one source. Picture 8 presents the class diagram that represents the basic structure of this pattern. The *CompositeMetadataReader* is a class that uses other readers to compose the metadata reading algorithm. At first, it populates a *MetadataContainer* by using the first reader; then the information is complemented by using other readers.



Picture 8 – Structure of Metadata Reader Chain.

In this pattern, the *AbstractMetadataReader* implementations must have a different implementation for reading metadata. The method *populateMetadataContainer()* does not return an instance of *MetadataContainer*, but receives it as a parameter and populates it. The reading algorithm must not assume that *MetadataContainer* instance is empty and must consider that another reader may have already populated it.

Picture 9 represents how *CompositeMetadataReader* uses other readers to compose the information in *MetadataContainer*. The *MetadataReaderClient* is responsible for creating a *MetadataContainer* instance, which is to be passed as a parameter. The *CompositeMetadataReader* delegates for each *ConcreteMetadataReader* instance the responsibility to read metadata and populate the *MetadataContainer*.



Picture 9 – Sequence diagram for a composite metadata reading.

Participants

- **AbstractMetadataReader -** It represents an abstraction of a metadata reader. Any *ConcreteMetadataReader* must implement it.
- **ConcreteMetadataReader -** It is responsible for reading metadata from one source and implements the interface of *AbstractMetadataReader*. It should consider that the *MetadataContainer* might have already been populated by another source.
- **MetadataReaderClient -** It represents a *Repository* or a *MBComponent*, depending on the component's structure. It is the class that needs to directly retrieve metadata from a reader.

- **CompositeMetadataReader -** It represents the reader that uses other *AbstractMetadataReader* to compose the metadata reading algorithm.
- **MetadataContainer -** It is responsible for representing the metadata of an application class at runtime. It is created by the MetadataReaderClient and populated by metadata readers.

Consequences

- The framework can use simultaneously more than one source of metadata.
- Alternative metadata sources that contains only a part of information can be used to complement the reading of metadata.
- The metadata defined in one source can be inconsistent with other sources and the readers must consider that situation, dealing with it appropriately.
- The order that the readers retrieve metadata can be configured.
- The use of more than one source for metadata definition by the framework can slow down its reading.

Known Uses

The ACE framework supports the metadata definition by using annotations and XML documents (Costa and Figueredo, 2009). The XML-based definition overrides the annotation-based definition allowing the configurations to be changed at deployment time. This pattern is used internally to allow this reading from multiple sources.

MetadataSharing is an implementation that allow metadata sharing among frameworks (MetadataSharing, 2009). A repository reads the metadata from many sources and then provides an API for the frameworks to retrieve the information that they need. The metadata reading in organized in a composite reader with a configurable reading order.

Hibernate Validator, release 4.0 alpha, is framework that checks in-memory instances of a class for constraint violations (Hibernate Validator, 2009) and implements the specification Bean Validation (JSR 303, 2009). It can override the metadata defined in annotations by the metadata in XML files, but by the specification this order cannot be changed. With an element in the XML file, it is possible to ignore the all the validation annotations defined in the class.

Running Example

After the Comparison Component is deployed with the application, some comparison rules needed to be changed. As to this kind of requirement, the definition of metadata in XML files is the most appropriate solution, but it is easier and more productive configure the comparison metadata using annotations too. Based on that, the Comparison Component is refactored to support the definition of metadata using annotations and be overridden by definitions conveyed by a XML document.

The first modification to be made is to change the signature of the method that reads metadata, to receive the metadata container as a parameter. Listing 15 shows this modification in the *ComparisonMetadataReader* interface. The method is renamed from *createContainer()* to *populateContainer()* to best describe what it really does.

Other modification that should be made in the existing metadata readers is to consider that metadata information may already exist in the descriptor. An example is presented in Listing 16. In this piece of code, it first tries to retrieve the *PropertyDescriptor* to verify if it is already in the descriptor and creates a new one only if it is not. The *Repository* also needs to be changed to create an empty instance of *ComparisonDescriptor* to pass it to the *populateContainer()* method.

```
public interface ComparisonMetadataReader {

    public abstract void populateContainer(Class c, ComparisonDescriptor descriptor);

}
```

Listing 15 – Source code of ComparisonMetadataReader: new signature for the method that reads metadata.

```
PropertyDescriptor pd = descriptor.getPropertyDescriptor(name);
if(pd == null){
    pd = new PropertyDescriptor();
    pd.setName(name);
    descriptor.addPropertyDescriptor(pd);
}
```

Listing 16 – Example of change in the metadata readers: getting or creating the PropertyDescriptor.

With these modifications, a composite metadata reader is possible to be implemented. The class *ChainComparisonMetadataReader* is presented in Listing 17. It receives a list of *ComparisonMetadataReader* as a parameter in the constructor and invokes all of them in the same order to populate the *ComparisonDescriptor* instance.

```
public class ChainComparisonMetatataReader implements ComparisonMetadataReader {

    private List<ComparisonMetadataReader> readers;

    public ChainComparisonMetatataReader(ComparisonMetadataReader... readers) {
        this.readers = new ArrayList<ComparisonMetadataReader>();
        for(ComparisonMetadataReader reader : readers){
            this.readers.add(reader);
        }
    }
    @Override
    public void populateContainer(Class c, ComparisonDescriptor descriptor) {
        for(ComparisonMetadataReader reader : readers){
            reader.populateContainer(c, descriptor);
        }
    }
}
```

Listing 17 – Source code of ChainComparisonMetadataReader.

Related Patterns

Metadata Reader Chain is important to enable metadata reading from sources that contains only a part of the necessary metadata. It is important for the implementation of **Metadata Reader Adapter**, which uses metadata already obtained from other frameworks to populate the **Metadata Container**.

Metadata Reader Chain uses **Composite** (Gamma at al, 1994) to create a metadata reader composed by other readers. An alternative implementation of this pattern could use the **Chain of Responsibility** (Gamma at al, 1994), where each reader could represent a handler in the processing chain. In this implementation, after reading metadata each reader would invoke the next reader in the chain.

4.3. Metadata Reader Adapter

Some metadata are useful for more that one framework. For example, information defined for object-relational mapping may be useful for the configuration of user interface components. It is desirable that the metadata can be read only once and do not need to be defined twice, witch can lead to inconsistencies.

Problem

How to allow one framework to retrieve metadata already obtained from another one?

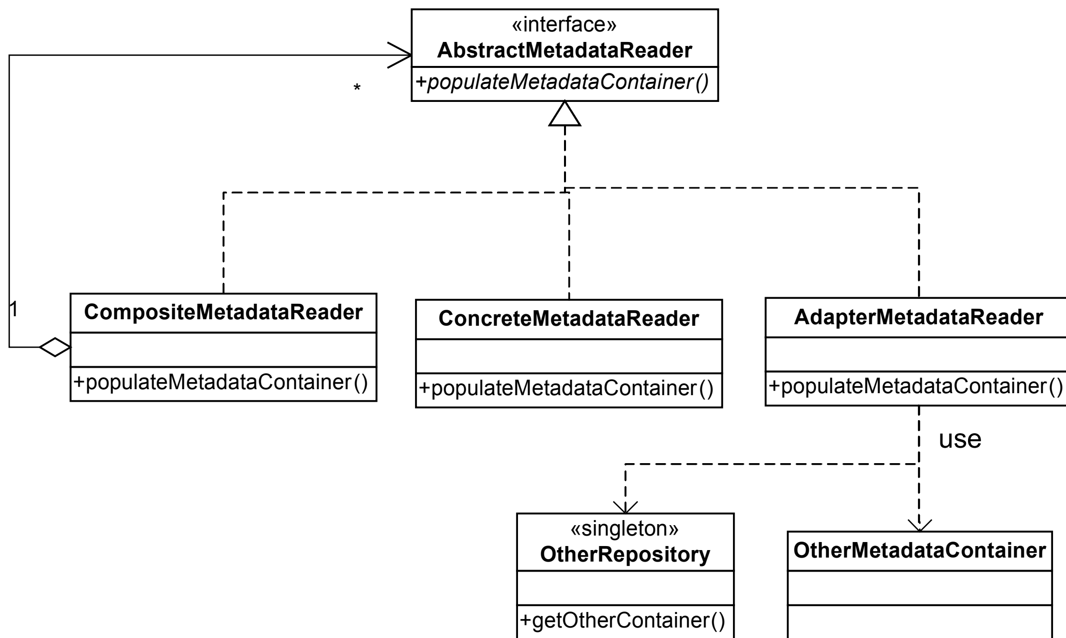
Forces

- A framework can read the metadata of another one directly, but the performance can degrade with two readings of the same information.
- It is easy to locate metadata of frameworks that support only one source, but that is not true for frameworks that support more than one source or an extensible metadata reading.
- The same information can be defined in the metadata formats of both frameworks, but it can lead to inconsistencies and may reduce the productivity.

Structure

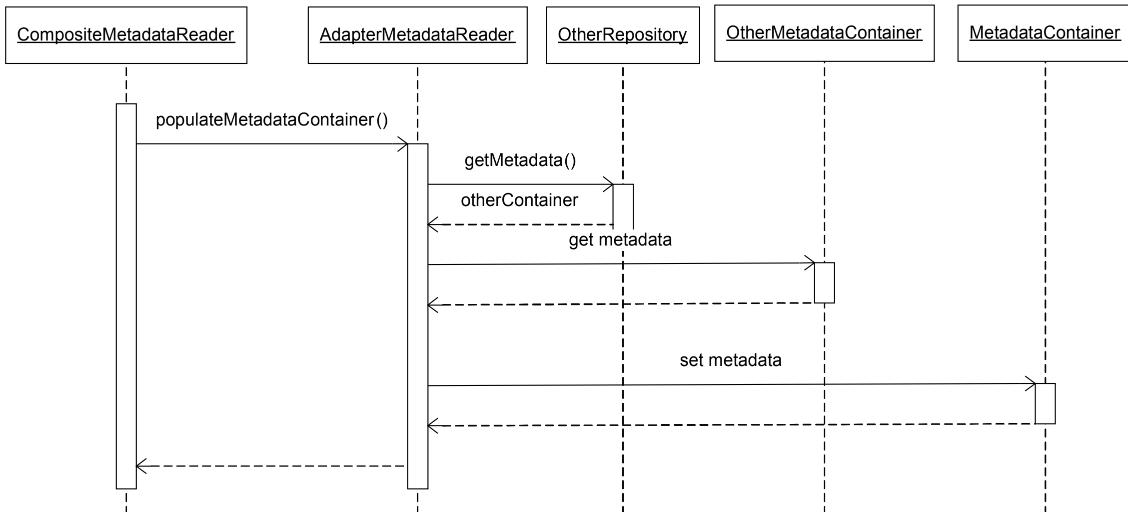
In **Metadata Reader Adapter** there is a metadata reader that accesses the **Metadata Repository** of another framework to get information to populate its own **Metadata Container**. It is difficult to get from other component all the metadata needed, so this pattern considers the use of the shared information to compose a **Metadata Reading Chain**.

Picture 10 presents the structure of this pattern. The class *AdapterMetadataReader* uses a repository from another framework, represented by the class *OtherRepository*, to retrieve its metadata container, represented by the class *OtherMetadataContainer*.



Picture 10 – Metadata Reader Adapter structure.

Picture 11 represents the sequence diagram for the method *populateMetadataContainer()* of the class *AdapterMetadataReader*. After retrieving the metadata from the other framework, the adapter sets the information on its own metadata container. Alternatively, the adapter can retrieve the metadata directly from a metadata reader, when a repository is not available in the other framework.



Picture 11 – Sequence diagram representing how the AdapterMetadataReader retrieves metadata from the repository of another framework.

Participants

- **AbstractMetadataReader -** It represents an abstraction of a metadata reader.

- **ConcreteMetadataReader -** It is responsible for reading metadata from one source and implements the interface of *AbstractMetadataReader*.
- **CompositeMetadataReader -** It represents the reader that uses other *AbstractMetadataReader* to compose the metadata reading algorithm.
- **AdapterMetadataReader -** It represents the metadata reader that accesses the *OtherRepository* to retrieve instances of *OtherMetadataContainer*. It converts also the information obtained to the *MetadataContainer* format.
- **MetadataContainer -** It is responsible for representing the metadata of an application class needed by the framework.
- **OtherRepository -** It represents the metadata repository of another framework.
- **OtherMetadataContainer -** It is responsible for representing the metadata of an application class needed by another framework.

Consequences

- The possibility of inconsistencies that might occur with the definition of the same information in two-metadata schema is reduced.
- The amount of metadata that should be defined for a class is reduced.
- This solution is sensitive to changes in the **Metadata Container** interface of the other component.
- This solution is only viable when the other component have an API that allows an application to get its **Metadata Container**.

Known Uses

MetadataSharing has central repository is used to store metadata read from the configured sources (MetadataSharing, 2009). The components must use a **Metadata Reader Adapter** to get it from the repository and put it in a format of its **Metadata Container**.

Hispagnol (2009) proposed a model to unify the models OLTP (Online Transaction Processing) and OLAP (Online Analytical Processing). In the proposed implementation, the OLTP metadata is retrieved and then adapted and complemented to compose the metadata necessary for OLAP.

In Nardon and Silva (2007), tips, tricks and new design patterns are presented in the context of a Java EE application using the EJB 3 specification (JSR 220, 2006). One of the practices described is the use of object-relational metadata retrieved from Hibernate SessionFactory (Bauer and King, 2004) for its use inside some kinds of EJB components.

Running Example

Some applications that use the Comparison Component, may also use the Hibernate (Bauer and King, 2004) in the persistence layer. When a property of a persistent class is also a persistent class, that property is a composed object. For the comparison domain, that means that this property must be deep compared. To avoid duplicate configurations

and inconsistencies, a Metadata Reader Adapter is created to get this information directly from Hibernate and is presented in the Listing 18.

```
public class AdapterComparisonMetadataReader implements ComparisonMetadataReader {
    private SessionFactory sessionFactory;

    public AdapterComparisonMetadataReader(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }

    public void populateContainer(Class c, ComparisonDescriptor descriptor) {
        ClassMetadata metadata = sessionFactory.getClassMetadata(c);
        if (metadata != null) {
            for (String prop : metadata.getPropertyNames()) {
                if (metadata.getPropertyType(prop).isEntityType()) {
                    PropertyDescriptor pd = descriptor.getPropertyDescriptor(prop);
                    if (pd == null) {
                        pd = new PropertyDescriptor();
                        pd.setName(prop);
                        descriptor.addPropertyDescriptor(pd);
                    }
                    pd.setDeepComparison(true);
                }
            }
        }
    }
}
```

Listing 18 – The AdapterComparisonMetadataReader source code.

The class *AdapterComparisonMetadataReader*, presented in Listing 18, receives in the constructor an instance of *SessionFactory*, namely a Hibernate class with a method that allows the application to retrieve the class metadata. Based on class metadata, in the method *populateContainer()* searches for properties that are also entities and sets the *deepComparison* to *true* in the *PropertyDescriptor*.

Related Patterns

Metadata Reader Adapter assumes that the framework uses **Metadata Reader Chain** because rarely the metadata from the other framework would have all the information needed. If that is not true, the **Metadata Reader Chain** does not need to be implemented. The **Metadata Repository** also needs to be implemented but in the other framework. If it is not implemented, an alternative is to retrieve the metadata directly from a metadata reader.

This pattern obviously is related to the **Adapter** (Gamma at al, 1994). In **Metadata Reader Adapter**, it is not the functionality provided that is adapted to another API, but the information provided by the other framework that is used and interpreted in another context.

4.4. Delegate Metadata Reader

The framework usually provides a standard format to define its metadata, but in some applications and for some frameworks domains it is important to allow the extension of the metadata schema. A prerequisite for this is to enable in the framework extensions to read pieces of metadata.

Problem

How to allow the extension of the metadata schema, delegating to configurable classes the responsibility to interpret pieces of metadata?

Forces

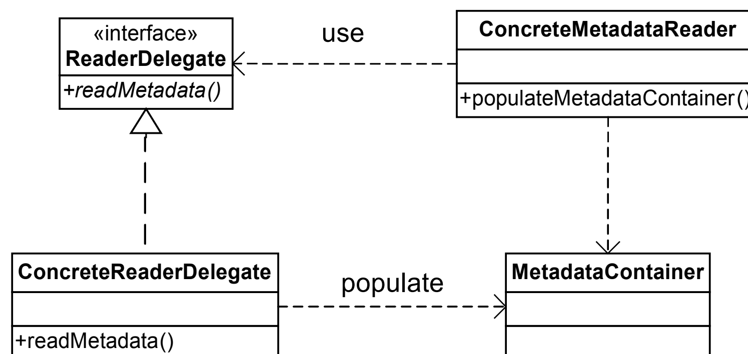
- The framework provides a general schema to represent metadata, but an application can express it closer to its domain.
- It is possible to read metadata from different schemas using the Metadata Reader Strategy, but sometimes it is important to extend the metadata from one of them.
- The frameworks provide metadata of general use, but for some specific domains, such as validation, some applications will probably need more specific metadata.

Structure

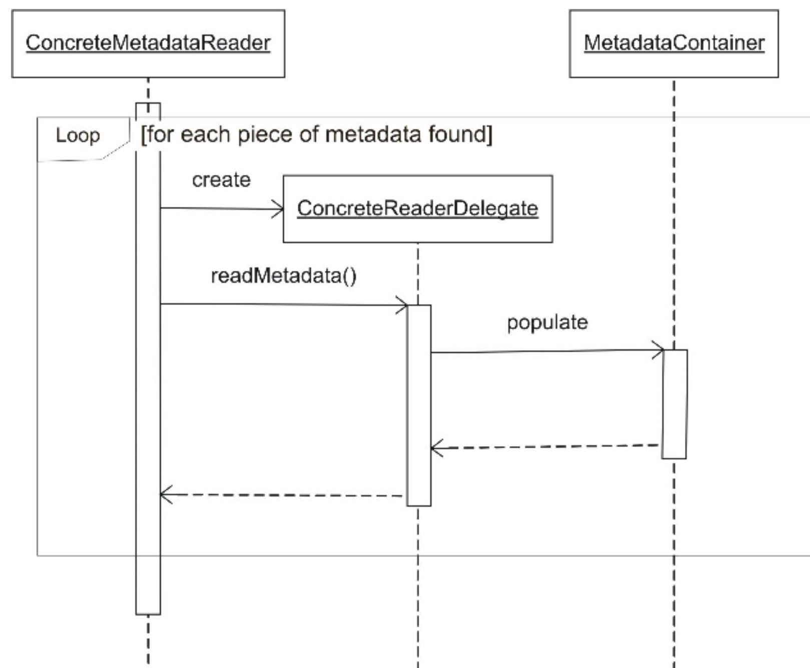
In **Delegate Metadata Reader** the metadata reader delegates to other classes the reading and interpretation of metadata. Each piece of metadata can have related classes with which the reading should be delegated. Using this structure, the metadata schema can be extended by creating classes that can read and interpret this extension.

To use this pattern, it is important to define what is a piece of metadata for the framework. It can be an annotation or, if it uses XML documents, an element or an attribute. The framework should also create a configurable mapping to relate each piece of metadata to the class that should read it.

Picture 12 represents the pattern structure. The interface *ReaderDelegate* is an abstraction of the classes that receives a piece of metadata and populates the *MetadataContainer* based on its information. Picture 13 presents a sequence diagram representing the reading of metadata using the *ReaderDelegate*. For each piece of metadata found, it creates the respective *ConcreteReaderDelegate* and delegates the reading of that piece to it. This *readMetadata()* method receives the *MetadataContainer* as a parameter and populates it with the metadata contained in that piece.



Picture 12 – The structure of Metadata Reader Delegate



Picture 13 – Sequence diagram representing the metadata reading process using Metadata Reader Delegate.

Participants

- **ConcreteMetadataReader -** It represents a class that reads metadata and delegates part of its logic to implementations of *ReaderDelegate*. It is responsible for instantiating the appropriate *ConcreteReaderDelegate* for each piece of metadata.
- **ReaderDelegate -** It is an abstraction of the classes that reads and interprets a piece of metadata.
- **ConcreteReaderDelegate -** It represents a concrete class that implements *ReaderDelegate* and interprets a specific piece of metadata.
- **MetadataContainer -** It is responsible for representing the metadata of an application class needed to the framework.

Consequences

- The metadata schema can be changed without affecting the concrete class responsible for reading that type of metadata definition.
- The metadata schema of the framework can be extended by the application.
- Depending on the number of possible different pieces of metadata, there will be many different implementations of *ReaderDelegate* and usually they are small classes.
- In some frameworks, it is difficult to divide metadata in pieces, which makes unfeasible the implementation of this pattern.

Known Uses

Hibernate Validator, release 3.1, is framework that checks in-memory instances of a class for constraint violations based on annotations (Hibernate Validator, 2009). New annotations can be created to validate more specific constraints associated to an application domain. Each framework annotation has the annotation *@ValidatorClass* that receives as a parameter a class that implements the interface *Validator*. This interface has the method *initialize()* that is used to interpret the annotation.

JColtrane (JColtrane, 2009) is a framework to parse XML files based on SAX (SAX, 2004) events. It uses annotations to define conditions for each method to be executed. New conditions can be added by creating annotations annotated with *@ConditionFactoryAnnotation*. This annotation receives the class responsible for reading that annotation.

XapMap, which stands for Cross Application Mapping, is a framework that maps entities of the same domain but implemented in different class structures (XapMap, 2009). It provides annotations for converting data among different types and a mechanism for the application to define its own convertion annotations. The mechanism is similar to the ones used in Hibernate Validator and JColtrane.

Running Example

The Comparison Component in this step is refactored to support annotations created by the application. As a consequence, the application is able to create annotations related to its domain, which can also be used by other frameworks and components.

Listing 19 presents the interface *AnnotationReader*, used to abstract the reading of an annotation. The method *readAnnotation()* receives as parameters the annotation to be interpreted and the *PropertyDescriptor* associated to the property where the annotation is found. The generic parameter in this interface will represent in the subclasses, the specific annotation that it should receive to interpret. Listing 20 presents the annotation that will be used in the annotations to define its respective *AnnotationReader*.

```
public interface AnnotationReader<A extends Annotation> {

   public void readAnnotation(A annotation, PropertyDescriptor descriptor);

}
```

Listing 19 – The interface AnnotationReader.

```
@Target({ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface DelegateReader {
   Class<? extends AnnotationReader> value();
}
```

Listing 20 – The annotation used to configure the AnnotationReader for each annotation.

The class *AnnotationComparisonMetadataReader* must be refactored to look for annotations with *@DelegateReader*, then to create the configured *AnnotationReader* and subsequently to invoke the *readAnnotation()* method. Listing 21 illustrates the new

version of this class. The specific code to read the *@Tolerance* and *@DeepComparison* is removed and substituted by a loop that searches for annotations that have the *@DelegateReader* annotation.

After the modification in the class that reads comparison metadata using annotations, the *@DelegateReader* must be inserted into the current framework annotations. Listing 22 presents this insertion into the *@Tolerance* and Listing 23 shows the class responsible for reading this annotation and inserting its information in the respective property descriptor. The same must be done for *@DeepComparison*.

```
public class AnnotationComparisonMetadataReader implements ComparisonMetadataReader {

   public void populateContainer(Class c, ComparisonDescriptor descr){
      for (Method method : c.getMethods()) {
         boolean isGetter = method.getName().startsWith("get");
         boolean noParameters = (method.getParameterTypes().length == 0);
         boolean notGetClass = !method.getName().equals("getClass");
         boolean noIgnore = !method.isAnnotationPresent(Ignore.class);
         if (isGetter && noParameters && notGetClass && noIgnore) {
            String getter = method.getName();
            String propName = getter.substring(3,4).toLowerCase()+getter.substring(4);
            PropertyDescriptor prop = descr.getPropertyDescriptor(propName);
            if(prop == null){
               prop = new PropertyDescriptor();
               prop.setName(propName);
               descr.addPropertyDescriptor(prop);
            }
            for(Annotation an :method.getAnnotations()){
               Class anType = an.annotationType();
               if(anType.isAnnotationPresent(DelegateReader.class)){
                  DelegateReader reader = (DelegateReader) anType
                     .getAnnotation(DelegateReader.class);
                  Class<? extends AnnotationReader> readerClass = reader.value();
                  try {
                     AnnotationReader anReader = readerClass.newInstance();
                     anReader.readAnnotation(an, prop);
                  } catch (Exception e) {
                     throw new RuntimeException("cannot instanciate reader",e);
                  }
               }
            }
         }
      }
   }
}
```

Listing 21 – The interface AnnotationReader.

```
@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@DelegateReader(ToleranceReader.class)
public @interface Tolerance {
   double value();
}
```

Listing 22 – The definition of @Tolerance annotation with @DelegateReader.

```
public class ToleranceReader implements AnnotationReader<Tolerance>{

   @Override
   public void readAnnotation(Tolerance annotation, PropertyDescriptor descriptor){
      descriptor.setTolerance(annotation.value());
   }
}
```

Listing 23 – The class responsible to read the @Tolerance annotation.

As an example of the extension of the metadata schema, one can assume that the application has fields in many different classes that represent a weight, and for these values the tolerance must always be '0.1'. Listing 24 has the definition of the *@Weight* annotation. The class *WeightComparisonReader*, represented in Listing 25, is configured to read this annotation. It eases a change in the tolerance for all the properties that represent a weight and can also be used by other frameworks for other purposes, such as validation with Hibernate Validator.

```
@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@DelegateReader(WeightComparisonReader.class)
public @interface Weight {
}
```

Listing 24 – The definition of the custom annotation @Weight.

```
public class WeightComparisonReader implements AnnotationReader<Weight> {
   @Override
   public void readAnnotation(Weight annotation, PropertyDescriptor descriptor) {
      descriptor.setTolerance(0.1);
   }
}
```

Listing 25 – The class responsible for interpreting the annotation @Weight.

Related Patterns

Delegate Metadata Reader is often used with **Metadata Processor**, allowing the extension of the metadata schema with the extension of the framework logic. Both patterns can be used independently, but it is more plausible to imagine them being used together.

5. Logic Processing Patterns

5.1. Metadata Processor

To extend the metadata schema increasing the framework functionality for new pieces of metadata, is not enough the extension of the metadata reading mechanism. It is necessary to enable in the framework the definition of classes to extend this processing logic for each new piece of metadata.

Problem

How to allow the framework functionalities to be extended, with the creation of classes that process pieces of metadata?

Forces

- Metadata provided by the framework usually cover a more general domain, but some applications may need some more specific functionalities.
- Applications can use parallel solutions to implement functionalities not covered by the framework, but the architecture will have two components with the same responsibility.

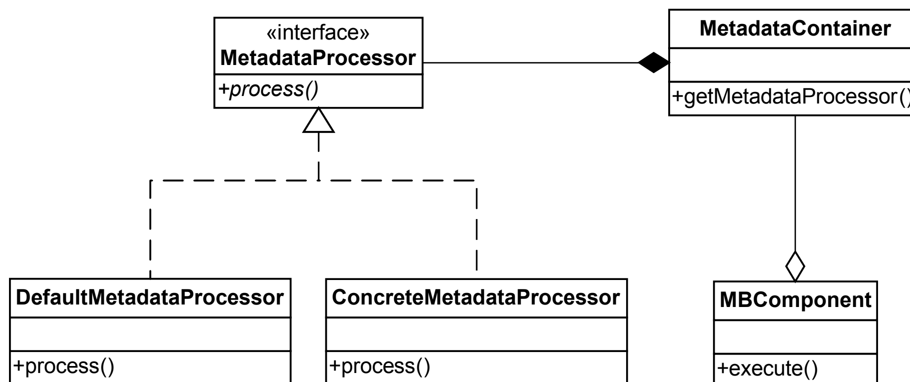
- The application developers may change the source code of an open-source framework to add functionality, but it might make unfeasible to take advantage of its future versions.
- Some framework domains like object-relational mapping do not need much metadata extension, but other domains, like instance validation that are closer to the application domain, do.

Structure

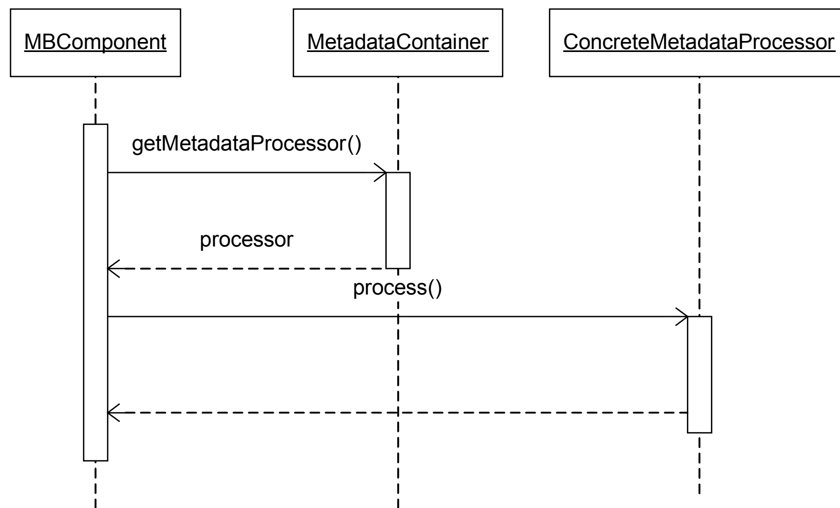
In **Metadata Processor,** part of the main functionality of the framework is delegated to other classes. These classes compose the **Metadata Container** and are created during the phase of metadata reading. The controller class of the framework retrieves processor instance from the container and calls its methods as part of the execution.

Picture 14 shows the class diagram for this pattern. The interface *MetadataProcessor* abstracts the concept of processing a piece of metadata. The *DefaultMetadataProcessor* represents a default implementation for processing and the *ConcreteMetadataProcessor* represents other implementations. The *MetadataProcessor* implementations can have instance variables to represent part of the metadata obtained during its reading.

A sequence diagram that represents the use of the processor by the framework is presented in Picture 15. The *MBComponent* retrieves the *MetadataProcessor* from the *MetadataContainer* and invokes its methods. It is possible to have more than one kind of processor in a framework, depending of the different tasks that it executes and the need to make it extensible.



Picture 14 – The structure of Metadata Processor.



Picture 15 – Sequence diagram representing the use of the metadata rocessor by the framework.

An alternative implementation is to store the metadata in the *MetadataContainer* in a more flexible way, for example using attribute maps, then use this information in the *MBComponent* to create the processor. It is recommended when the processor must have heavyweight objects or it cannot be shared between more than one framework entry point instance.

Participants

- **MBComponent -** It is the framework entry point. It is responsible for executing of the main logic and for being a controller of the other classes. It retrieves an implementation of the *MetadataProcessor* from the *MetadataContainer* and executes it as part of the framework logic.
- **MetadataContainer -** It is responsible for representing the metadata of an application class needed to the framework. It is composed of instances of *MetadataProcessor* implementations.
- **MetadataProcessor -** It is an abstraction of the classes that compose the *MetadataContainer* and is invoked as part of the framework's logic by the *MBComponent*.
- **DefaultMetadataProcessor -** It represents a default implementation of the *MetadataProcessor*.
- **ConcreteMetadataProcessor -** It represents a concrete implementation of the *MetadataProcessor*.

Consequences

- It is possible to extend the framework functionalities by creating more *MetadataProcessor* implementations.
- The *MetadataContainer* uses a more flexible structure, allowing the addition of different information easily.

- Allow the application to extend the metadata schema by adding functionality relative to its domain.
- To implement this pattern, the metadata must be divided into pieces that can be processed separately and in some frameworks domains that is not possible.
- The use of processors in the metadata container structure, may difficult the metadata interpretation when it is retrieved by other frameworks.

Known Uses

In Hibernate Validator framework (Hibernate Validator, 2009), release 3.1, one can define new annotations by using the annotation *@ValidatorClass* to reference a class that implements the interface *Validator*, that is simultaneously the **Metadata Reader Delegate** and the **Metadata Processor**. Other frameworks, like Stella (Caelum Stella, 2009), extends the validation annotations to a more specific domain.

JColtrane (JColtrane, 2009) uses annotations to define conditions for each method to be executed based on SAX events. The method *getConditions()* of the interface *ConditionFactory* returns a list of *Condition*, that is the **Metadata Processor** for this framework. The *Condition* interface has the method *verify()*, which based on information of the parsing event, returns true if that method should be invoked.

The SwingBean framework (SwingBean, 2009) is an example of a framework that uses the **Metadata Processor** and not the **Delegate Metadata Reader**. The **Metadata Container** in SwingBean uses maps to store metadata that were read in XML files. The processor in SwingBean correspond to a set of wrappers for graphical components that are created based on the descriptor. Wrappers for new graphical components can be created and mapped for different values of the 'type' attribute for the 'property' element in the XML descriptor.

Running Example

The Comparison Component deals with a domain that can have many rules that are specific to the application. It is important for the component to allow the application developers to add new types of comparison and associate them with new pieces of metadata. In this section, the Comparison Component is refactored and a new kind of comparison is added.

The interface *ComparisonProcessor* is presented in Listing 26. The method *compare()* receives the property name and the values to be compared and return *null* if they can be considered the same or the respective *Difference* instance. Listing 27 presents the new *PropertyDescriptor* class, which has as an instance variable the respective *ComparisonProcessor*. The method *getProcessor()* returns an instance of the class *RegularProcessor*, presented in Listing 28, if the processor attribute is *null*.

```
public interface ComparisonProcessor {

   public Difference compare(String prop, Object oldValue, Object newValue);

}
```

Listing 26 – The ComparisonProcessor interface.

```
public class PropertyDescriptor {

   private String name;
   private ComparisonProcessor processor;
   private boolean deepComparison;

   public ComparisonProcessor getProcessor() {
      if(processor == null)
         processor = new RegularProcessor();
      return processor;
   }
   public void setProcessor(ComparisonProcessor processor) {
      this.processor = processor;
   }

   //other getters and setters omitted
}
```

Listing 27 – The source code of the refactored PropertyDescriptor.

```
public class RegularProcessor implements ComparisonProcessor {
    @Override
    public Difference compare(String prop, Object oldValue, Object newValue) {
        if (newValue == null) {
            if (oldValue != null) {
                return new Difference(prop, newValue, oldValue);
            }
        } else if (!newValue.equals(oldValue)) {
            return new Difference(prop, newValue, oldValue);
        }
        return null;
    }
}
```

Listing 28 – The source code of RegularProcessor.

The tolerance, that are stored as an attribute in the *PropertyDescriptor*, now is an instance variable of the *ToleranceProcessor*, presented in Listing 29. The class responsible for reading the *@Tolerance* annotation, represented in Listing 30, is also changed to create the instance of *ToleranceProcessor* with the correct tolerance value.

```
public class ToleranceProcessor implements ComparisonProcessor {
    private double tolerance;
    public ToleranceProcessor(double tolerance) {
        this.tolerance = tolerance;
    }
    @Override
    public Difference compare(String prop, Object oldValue, Object newValue) {
        double dif = Math.abs(((Double) newValue) - ((Double) oldValue));
        if (dif > tolerance) {
            return new Difference(prop, newValue, oldValue);
        }
        return null;
    }
}
```

Listing 29– The class ToleranceProcessor that processes the tolerance metadata.

```
public class ToleranceReader implements AnnotationReader<Tolerance>{
    @Override
    public void readAnnotation(Tolerance annotation,
            PropertyDescriptor descriptor){
        double tolerance = annotation.value();
        ToleranceProcessor processor = new ToleranceProcessor(tolerance);
        descriptor.setProcessor(processor);
    }
}
```

Listing 30 – The class ToleranceReader that creates the ToleranceProcessor instance.

The new implementation of the class *ComparisonComponent* is presented in Listing 31. When the property *deepComparison* is *false*, the *ComparisonProcessor* is retrieved from the *PropertyDescriptor* and is used to make the comparison.

```
public class ComparisonComponent {

    public List<Difference> compare(Object oldObj, Object newObj)
                                              throws CompareException {

        List<Difference> difs = new ArrayList<Difference>();

        if (!newObj.getClass().isAssignableFrom(oldObj.getClass()))
            throw new CompareException("Not compatible types");
        ComparisonDescriptor descr = Repository.getInstance().
                getMetadata(newObj.getClass());

        for (String prop : descr.getProperties()) {
            try {
                String getterName = "get" + prop.substring(0, 1).toUpperCase()
                    + prop.substring(1);
                Method method = newObj.getClass().getMethod(getterName);
                Object oldValue = method.invoke(oldObj);
                Object newValue = method.invoke(newObj);
                PropertyDescriptor descProp = descr.getPropertyDescriptor(prop);

                if (descProp.isDeepComparison() && newValue != null
                        && oldValue != null) {
                    List<Difference> difsProp = compare(newValue, oldValue);
                    for (Difference d : difsProp) {
                        d.setProperty(prop + "." + d.getProperty());
                        difs.add(d);
                    }
                } else {
                    ComparisonProcessor processor = descProp.getProcessor();
                    Difference dif = processor.compare(prop, oldValue, newValue);
                    if(dif != null)
                        difs.add(dif);
                }
            } catch (Exception e) {
                throw new CompareException("Error retrieving property", e);
            }
        }
        return difs;
    }
}
```

Listing 31 – The class ComparisonComponent refactored to support ComparisonProcessor.

To illustrate how the metadata can be extended conveying with new framework functionalities, an annotation for comparing property substrings is created. Listing 32 creates the annotation *@CompareSubstring*, that has attributes to configure the beginning and the end of the String. This annotation is mapped to the delegate reader presented in Listing 33, the *SubstringComparisonReader*. This class creates the

SubstringProcessor, presented in Listing 34, and puts it in the *PropertyDescriptor* to be used by the framework.

```
@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@DelegateReader(SubstringComparisonReader.class)
public @interface CompareSubstring {
   int begin() default 0;
   int end() default Integer.MAX_VALUE;
}
```

Listing 32 – The definition of @CompareSubstring annotation.

```
public class SubstringComparisonReader implements AnnotationReader<CompareSubstring> {
    @Override
    public void readAnnotation(CompareSubstring annotation,
                               PropertyDescriptor descriptor) {
        int begin = annotation.begin();
        int end = annotation.end();
        SubstringProcessor p = new SubstringProcessor(begin,end);
        descriptor.setProcessor(p);
    }
}
```

Listing 33 – The class SubstringComparisonReader that reads the @CompareSubstring annotation.

```
public class SubstringProcessor implements ComparisonProcessor {
    private int begin;
    private int end;
    public SubstringProcessor(int begin, int end) {
        this.begin = begin;
        this.end = end;
    }
    @Override
    public Difference compare(String prop, Object oldValue, Object newValue) {
        if (newValue == null) {
            if (oldValue != null) {
                return new Difference(prop, newValue, oldValue);
            }
        } else{
            String oldString, newString;
            if(end == Integer.MAX_VALUE){
                oldString = oldValue.toString().substring(begin);
                newString = newValue.toString().substring(begin);
            }else{
                oldString = oldValue.toString().substring(begin, end);
                newString = newValue.toString().substring(begin, end);
            }
            if(!oldString.equals(newString))
                return new Difference(prop, newValue, oldValue);
        }
        return null;
    }
}
```

Listing 34 – The class SubstringProcessor that makes the comparison of substrings.

Related Patterns

Metadata Processor is often used in conjunction with **Delegate Metadata Reader**, allowing the extension of the metadata schema conveying the extension of the framework logic. This pattern can also be used in conjunction with a **Metadata Container** with a flexible structure, where the processors are created based on the information contained in the metadata container.

A **Metadata Processor** is similar to the **Command** (Gamma et al., 1994), but it is related to a piece of metadata. It is also related to **Strategy** (Gamma et al., 1994), because each processor can be considered a strategy for executing one piece of metadata.

5.2. Metadata Processing Layers

Sometimes it is not possible to divide the processing of a metadata-based framework by pieces of metadata, because it can have more than one responsibility that uses all the metadata schema. To enable the addition of new responsibilities, the framework must support the addition of new processing logic layers.

Problem

How to allow the framework functionalities to be extended, with the creation of layers with different responsibilities?

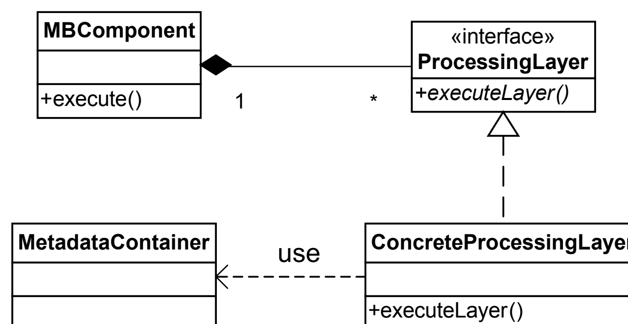
Forces

- Some metadata-based frameworks have a single and well-defined responsibility, but others execute different tasks based on the same metadata.
- The **Metadata Processor** enable extension when it is possible to separate the logic by pieces of metadata, but when the framework has more than one responsibility it is harder to make this division.
- The framework may have well defined responsibilities, but the application may need to add other ones that can be executed using the same metadata.

Structure

In **Metadata Processing Layers**, the main logic of the framework is divided in more than one layer of execution. This allows each layer to evolve independently and enable the extension by the addition of other layers.

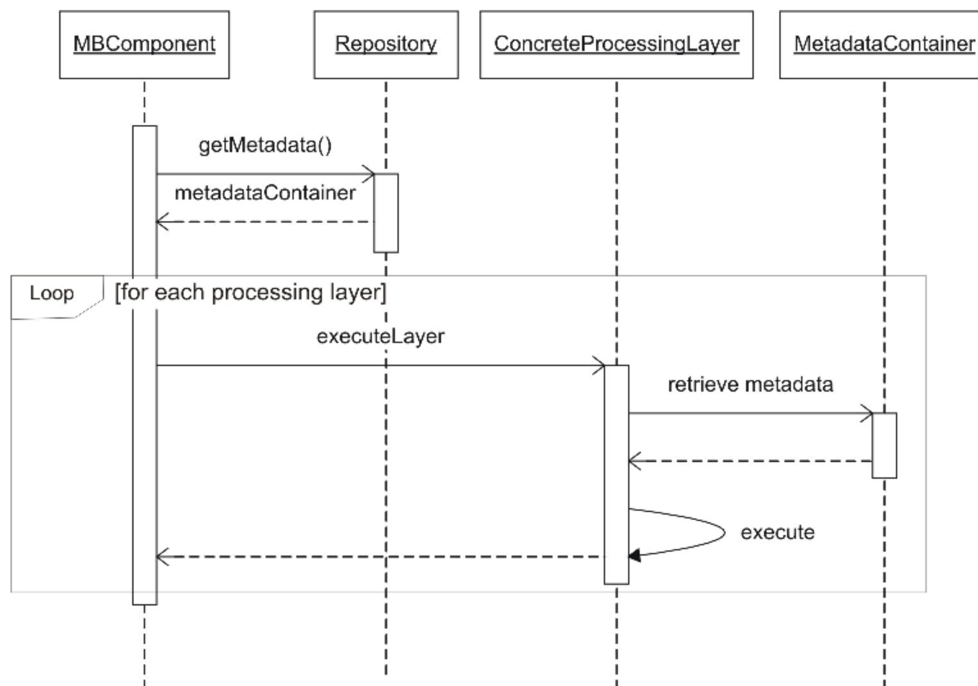
Picture 16 presents the pattern structure. The *MBComponent* is composed by many processing layers with different responsibilities. The *MBComponent* is responsible for defining when the layers should be invoked. Each ConcreteProcessingLayer can access the information in MetadataContainer to use the metadata as the base for its logic.



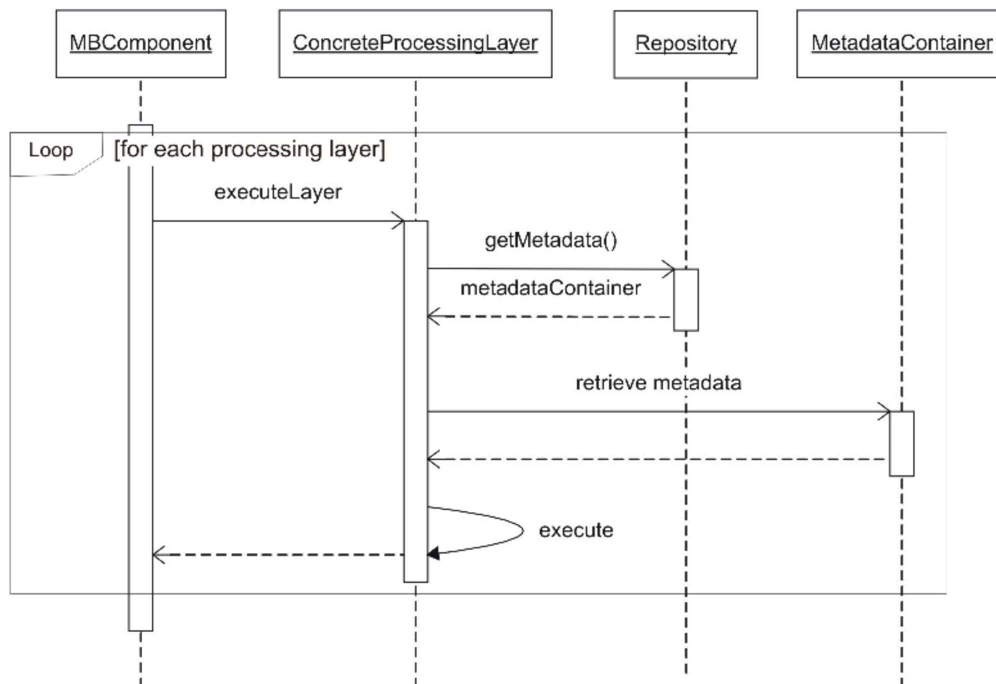
Picture 16 – Structure of Metadata Processing Layer.

There are two alternatives for each layer to access the metadata. The *MBComponent* can retrieve the *MetadataContainer* and pass it as a parameter in each layer, as represented in the sequence diagram of Picture 17. The other solution is each layer to access the *Repository* separately to retrieve the *MetadataContainer*, as represented in Picture 18. The use of the *Repository* is not mandatory, but it is particularly important when each layer retrieves metadata independently.

As an alternative implementation, the layers can also be implemented using **Chain of Responsibility** (Gamma et al., 1994). In this implementation, each layer would be responsible for invoking or not the next one. This way, the *MBComponent* does not control the layers invocation and only call the first one.



Picture 17– Sequence diagram for the alternative that the metadata container is passed as a parameter to the layers.



Picture 18– Sequence diagram for the alternative that the metadata container is retrieved from the repository for each layer.

Participants

- **MBComponent -** It is the framework entry point. It is responsible for executing the main logic and for being a controller of the other classes. It contains a list of *ProcessingLayer* implementations and invokes them in the right order.
- **MetadataContainer -** It is responsible for representing the metadata of an application class needed to the framework. Each *ProcessingLayer* can use it during the logic processing.
- **ProcessingLayer -** It is an abstraction of the classes that represents a processing layer of the framework.
- **ConcreteProcessingLayer -** It represents a concrete implementation of the *ProcessingLayer*. It uses the *MetadataContainer* to execute part of the framework's logic.
- **Repository -** It is responsible for managing the metadata reading and storing internally the instances of MetadataContainer. It is a singleton and provides metadata for *MBComponent* or for each *ConcreteProcessingLayer*.

Consequences

- It is possible to extend the framework functionalities by creating more *ProcessingLayer* implementations.
- The order of layers execution can be customized by the application.
- Implementations of *ProcessingLayer* can be added, substituted and removed for each *MBComponent* instance, enabling different behaviors for the framework in the same application.

- The creation of layers can over-design the framework if it has a well-defined responsibility that rarely can be extended.

Known Uses

JBoss Application Server 5 (JBoss, 2009) is an implementation of an application server that supports the EJB 3 specification (JSR 220, 2006). The specification defines that an EJB container must execute many responsibilities, such as transaction management, access control and exception handling. These functionalities are executed based on class metadata implemented in many aspects, using JBoss AOP (JBoss AOP, 2009). Each aspect advice can be considered a processing layer in this context.

The SwingBean framework implements many responsibilities such as validation, form and table creation and customization of each graphical component (SwingBean, 2009). Each responsibility is implemented in a different class, which receive a FieldDescriptor instance with the class metadata.

Esfinge Framework is a framework for the business layer of a corporative application (Esfinge, 2007). It provides a layered structure that allows layers to be easily created and inserted. Each layer can use the entity class metadata to customize its behavior. There are layers implemented for logging, remote access, remote notification and access control.

Running Example

The Comparison Component has some different responsibilities in terms of comparison. The functionality to be delegated to the layers is the comparison of object properties. Examples of these responsibilities are the comparison of *null* values, the deep comparison and the comparison of values using the *ComparisonProcessor*. The application may need to add another comparison layers for more complex data structures such as lists, sets, maps and trees.

Listing 35 presents the class *ComparisonLayer* that abstracts a comparison processing layer for each property. The method *compare()* receives the values to be compared, the list of *Difference* and the respective *PropertyDescriptor*. This implementation chose to pass the metadata as a parameter, because only a part of the class metadata is necessary. The boolean value returned by the *compare()* method indicates whether the comparison was already performed in that layer. Each layer also receives the reference to the own *ComparisonComponent*.

```
public abstract class ComparisonLayer {

   private ComparisonComponent component;

   public abstract boolean compare(Object oldValue, Object newValue,
        List<Difference> difs, PropertyDescriptor descProp) throws CompareException ;

   public ComparisonComponent getComponent() {
      return component;
   }
   public void setComponent(ComparisonComponent component) {
      this.component = component;
   }
}
```

Listing 35 – The ComparisonLayer abstract class.

The new implementation of the *ComparisonComponent* is presented in Listing 36. The attribute *layers* stores the list of the configured *ComparisonLayer* instances. The class provides a constructor that receives a list of *ComparisonLayer*. It also has a constructor without parameters that defines tree default constructors. In the *compare()* method, the comparison of each property uses the comparison of each layer until one returns *true*, meaning that the comparison is already completed.

Listings 37, 38 and 39 presents respectively the classes *NullComparisonLayer*, *DeepComparisonLayer* and *ValueComparisonLayer*. They present the implementation of comparison layers whose functionalities are already included in the earlier version of the Comparison Component.

```
public class ComparisonComponent {

    private List<ComparisonLayer> layers;

    public ComparisonComponent(ComparisonLayer... layers){
        this.layers = new ArrayList<ComparisonLayer>();
        for(ComparisonLayer layer : layers){
            layer.setComponent(this);
            this.layers.add(layer);
        }
    }
    public ComparisonComponent(){
        this(new NullComparisonLayer(),
                new DeepComparisonLayer(),
                new ValueComparisonLayer());
    }
    public List<Difference> compare(Object oldObj, Object newObj)
    throws CompareException {

        List<Difference> difs = new ArrayList<Difference>();

        if (!newObj.getClass().isAssignableFrom(oldObj.getClass()))
            throw new CompareException("Not compatible types");
        ComparisonDescriptor descr = Repository.getInstance().
                getMetadata(newObj.getClass());

        for (String prop : descr.getProperties()) {
            try {
                String getterName = "get" + prop.substring(0, 1).toUpperCase()
                + prop.substring(1);
                Method method = newObj.getClass().getMethod(getterName);
                Object oldValue = method.invoke(oldObj);
                Object newValue = method.invoke(newObj);
                PropertyDescriptor descProp = descr.getPropertyDescriptor(prop);
                boolean compared = false;
                for(int i=0; i<layers.size() && !compared; i++){
                    ComparisonLayer layer = layers.get(i);
                    compared = layer.compare(oldValue, newValue, difs, descProp);
                }
            } catch (Exception e) {
                throw new CompareException("Error retrieving property", e);
            }
        }
        return difs;
    }
}
```

Listing 36 – The ComparisonComponent refactored to use the ComparisonLayer.

```
public class NullComparisonLayer extends ComparisonLayer {
    @Override
    public boolean compare(Object oldValue, Object newValue,
            List<Difference> difs, PropertyDescriptor descProp)
            throws CompareException {
        if ((oldValue == null && newValue != null)
             || (oldValue != null && newValue == null)) {
            Difference dif = new Difference(descProp.getName(), oldValue,newValue);
            difs.add(dif);
            return true;
        }
        if(oldValue == null && newValue == null){
            return true;
        }
        return false;
    }
}
```

Listing 37 – NullComparisonLayer, responsible for comparison when null values are involved.

```
public class DeepComparisonLayer extends ComparisonLayer {
    @Override
    public boolean compare(Object oldValue, Object newValue,
            List<Difference> difs, PropertyDescriptor descProp)
            throws CompareException {
        if (descProp.isDeepComparison()) {
            List<Difference> difsProp = getComponent().compare(newValue, oldValue);
            for (Difference d : difsProp) {
                d.setProperty(descProp.getName() + "." + d.getProperty());
                difs.add(d);
            }
            return true;
        }
        return false;
    }
}
```

Listing 38 – DeepComparisonLayer, responsible for deep comparisons.

```
public class ValueComparisonLayer extends ComparisonLayer {
    @Override
    public boolean compare(Object oldValue, Object newValue,
            List<Difference> difs, PropertyDescriptor descProp)
            throws CompareException {
        ComparisonProcessor processor = descProp.getProcessor();
        Difference dif = processor.compare(descProp.getName(), oldValue, newValue);
        if(dif != null)
            difs.add(dif);
        return true;
    }
}
```

Listing 39 – ValueComparisonLayer, responsible for comparisons using the ComparisonProcessor.

Related Patterns

Metadata Processing Layer can be combined with **Metadata Processor** to extend the framework logic in different ways. The use of this pattern in conjunction with **Metadata Repository** is recommended, since the metadata can be retrieved independently in each layer.

This pattern can be implemented using the structure of the **Chain of Responsibility** (Gamma et al., 1994), in which one layer is responsible for invoking the next one. In frameworks that implement crosscutting concerns, each layer can be implemented as a **Proxy** or **Decorator** pattern(Gamma et al., 1994).

6. Conclusion

This paper presents the Pattern Language for Metadata-based Frameworks that document design best practices for this kind of framework. Many existing frameworks used to develop applications apply these concepts and this work can help in the development of new frameworks and in the refactoring of existent ones. Despite the fact that all code examples are given in Java, the patterns presented here can be implemented in any object-oriented programming language.

The following are the main contributions of this work:

- The study and investigation of the internal structural solutions of existing open source metadata-based frameworks.
- The documentation of the best practices found, in the form of a pattern language that includes solutions for the structure, metadata reading and logic processing of metadata-based frameworks.
- The creation of a detailed running example that illustrates how to refactor a metadata-based framework to implement each pattern of the presented pattern language.

Observing the Comparison Component, that is functional before implementing none of the patterns, it is possible to verify how it can be more flexible and extensible by using the best practices documented in the presented pattern language. The consolidation of this design knowledge about metadata-based frameworks is important for the generation of more mature solutions in the development of this kind of software. The metadata sharing among different frameworks, for example, is an objective hard to be accomplished in an architecture with the structure of many existent frameworks.

In this work, the design patterns are applied to the Comparison Component through refactoring, but in a real framework development, the requirements should make some to be implemented in the first place. A suggestion for a future work is the definition of a methodology for the development of metadata-based frameworks, that should include not only the framework design, but also activities like metadata modeling.

This pattern language addresses only solutions regarding the internal structure of a framework. It is also important, as a future work, to identify architectural patterns that capture what roles can a metadata-based framework perform in a software architecture. This work will facilitate the identification of situations where these frameworks can be used successfully.

Applications that uses Adaptive Object Models (Yoder et al, 1998), that uses metadata to define a more dynamic and flexible domain model, may also benefit from these patterns. But, as the authors do not investigate any examples of their use in this scenario, that analysis is left as a future work.

References

Alur, D. ; Malks, D.; Crupi, J. “Core J2EE Patterns: Best Practices and Design Strategies”. Prentice Hall PTR, 2nd Edition, 2003.

Bauer, C.; King, G. “Hibernate in Action”. Manning Publications, 2004.

Beck, K. “Implementation Patterns”, Addison-Wesley Professional, 1st Edition, 2007.

Buschmann, F.; Henney, K.; Schmidt, D. C. “Pattern Oriented Software Architecture Volume 5: On Patterns and Pattern Languages”. Wiley, 2007.

Caelum Stella. “Stella – Simplifying the Software Development in Brazil”, Available on http://stella.caelum.com.br/, 2009. [in Portuguese]

Costa, B. C.; Figueredo, L. P. “A Metadata-based Architecture for the Integration of Web and Mobile Applications”, Technical Report, Aeronautical Institute of Technology, 2009. [in Portuguese]

Dov, A. B. "Convention vs. Configuration". Available on http://www.javalobby.org/java/forums/t65305.html, 2006.

Esfinge, “Esfinge Framework”, Available on http://esfinge.sourceforge.net/, 2007.

Forman, I. R.; Forman, N. “Java Reflection in Action”. Manning Publications, 2004.

Fowler, M. “Inversion of Control Containers and the Dependency Injection pattern”. [S.n.t.]. Available on http://www.martinfowler.com/articles/injection.html, 2004.

Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. “Design Patterns: Elements of Reusable Object-Oriented Software”, Addison-Wesley, 1994.

Guerra, E. M. ; Fernandes, C. T. “A Metadata-Based Components Model”. In: Doctoral Symposium at 22nd European Conference on Object Oriented Programming (ECOOP 08), 2008, Pathos - Cyprus.

Guerra, E. M. ; Parente, J. M. ; Fernandes, C. T. . “Mapping Objects to Ontology Entities Using Metadata” In: X SIGE - Defense Operational Applications Symposium, São José dos Campos, SP, Brazil, 2008. [in Portuguese]

Guerra, E. M. ; Pavão, F.; Fernandes, C. T. “Design Patterns for Metadata-based Components and Frameworks ”. In: 7ª Latin American Conference on Pattern Languages of Programming – SugarLoafPLoP 2008, 2008, Fortaleza - Brazil. [in Portuguese]

Guerra, E. M. ; Silva J. ; Silveira, F. ; Fernandes, C. T. . “Using Metadata in Aspect-Oriented Frameworks”. In: 2nd Workshop on Assessment of Contemporary Modularization Techniques (ACoM.08) at OOPSLA 2008 - ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2008, Nashville - EUA.

Hibernate Validator. “Hibernate Validator”, Available on http://www.hibernate.org/412.html, 2009.

Hispagnol, G. M. “Unified Multidimensional Model: Integrating OLAP and OLTP Domains.”, Technical Report, Aeronautical Institute of Technology, 2009.[in Portuguese]

JBoss. “JBoss Application Server”, Available on http://www.jboss.org/jbossas/, 2009.

JBoss AOP. “JBoss AOP”, Available on http://www.jboss.org/jbossaop/, 2009.

JColtrane. “JColtrane – Better than SAX Alone”, Available on http://jcoltrane.sf.net, 2009.

JSR 175. “JSR 175: A Metadata Facility for the Java Programming Language”. Available on http://www.jcp.org/en/jsr/detail?id=175, 2003.

JSR 220. “JSR 220: Enterprise JavaBeans 3.0”. Available on http://www.jcp.org/en/jsr/detail?id=220, 2006.

JSR 222. “JSR 222: Java Architecture for XML Binding (JAXB) 2.0”. Available on http://jcp.org/en/jsr/detail?id=222, 2006.

JSR 303. “JSR 303: Bean Validation”. Available on http://jcp.org/en/jsr/detail?id=303, 2009.

JUnit. “JUnit - Testing Resources for Extreme Programming”. Available on http://www.junit.org/, 2008.

MentalLink. “MentalLink” Available on http://sourceforge.net/projects/mentallink/, 2008.

MetadataSharing. “SourceForge.net MetadataSharing”. Available on http://sourceforge.net/projects/metadatasharing/, 2009.

Nardon, F.; Silva, E. “Implementing Java EE Applications Using Enterprise JavaBeans (EJB) 3 Technology: Real World Tips, Tricks, and New Design Patterns”, JavaOne 2007, Session TS-4721, Avaliable on http://developers.sun.com/learning/javaoneonline/2007/pdf/TS-4721.pdf, 2007.

Nock, C. “Data Access Patterns: Database Interactions in Object-Oriented Applications”, Addison-Wesley Professional, 2003.

SAX. “SAX Project” Available on http://www.saxproject.org/, 2004.

SwingBean. “SwingBean”, Available on http://swingbean.sourceforge.net/, March, 2009.

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.

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.

Yoder, J. W.; Foote, B.; Riehle, D.; Tilman, M. “Metadata and Active Object-Models”, Fifth Conference on Patterns Languages of Programs (PLoP '98), Monticello, Illinois, August 1998.

XapMap. “XapMap - Cross Application Mapping Framework”, Available on http://xapmap.sourceforge.net, March, 2009.