Architectural Patterns for Metadata-based Frameworks Usage

EDUARDO GUERRA AND CLOVIS FERNANDES, Aeronautical Institute of Technology (ITA)

FÁBIO FAGUNDES SILVEIRA, Federal University of São Paulo (UNIFESP)

The usage of metadata-based frameworks is becoming popular for some kinds of software, such as Web and enterprise applications. However, it is not clear for which kinds of problems this approach can be applied. This paper presents a study that investigated the metadata usage in existing frameworks and documented recurrent solutions as architectural patterns. As a result, software architects might use such approaches for similar problems, being aware of their benefits and drawbacks in each scenario.

Categories and Subject Descriptors: **D.1.5** **[Programming Techniques]**: Object-oriented Programming — *Java* ; **D.2.11** **[Software Architectures]**: Patterns — *Metadata-based Frameworks*; **D.2.10** **[Design]**: Methodologies — *Frameworks*

General Terms: Design, Algorithms

Additional Key Words and Phrases: Metadata, Frameworks, Software Architecture, Patterns

ACM Reference Format:

Guerra, E., Fernandes, C. and Silveira, F. 2010. Architectural Patterns for Metadata-based Frameworks Usage. 17th Conference on Pattern Languages of Programs (PLoP), Reno, Nevada, USA (October 16-18 2010) 25 pages.

1. CONTEXT

A framework can be considered incomplete software system with some points that can be specialized to add application-specific behavior, consisting in a set of classes that represents an abstract design for a family of related problems. It is more than well written class libraries, which are more application independent and provide functionality that can be directly invoked. A framework provides a set of abstractions that must be extended and composed with others to create a concrete and executable application. Those classes can be application-specific or taken from a class library, usually built-in with the framework (Johnson and Foote 1988).

Another important characteristic of frameworks is their use of inversion of control (Johnson and Foote 1988). A framework's runtime architecture enables the definition of processing steps that can call applications handlers. This allows the framework to determine which set of application methods should be called in response to an external event. The common execution flow is an application to invoke the functionality on an external piece of software and not the opposite. Using inversion of control, the framework, and not the application, is responsible for the main execution flow. This is also known as the Hollywood Principle: “don't call us, we'll call you”.

Reflection usage in a framework can make it more adaptable to different contexts and applications (Foote and Yoder 1996). If an application class needs to implement a framework's interface or extend one of its superclasses, changes in the framework can break the application's code (Beck 2007). A reduced coupling facilitates the framework to evolve independently from the application and vice versa. The reflection usage enables the abstraction of pieces of code which deals with application specific classes, allowing them to become a framework responsibility.

When a framework uses reflection to access the class elements to execute its functionalities, sometimes the class intrinsic information is not enough. If framework behavior should differ for distinct classes, methods or attributes, it is necessary to add more specific information to enable differentiation. For some domains, it is possible to use marking interfaces, like Serializable in Java Platform, or naming conventions (Chen 2006), like in Ruby on Rails (Ruby et al. 2009). But those strategies can be used only for a limited amount of information and are not suitable for situations where more data is necessary, since it has a limited expressiveness.

Metadata is an overloaded term in computer science and can be interpreted differently according to the context. In the context of object-oriented programming, metadata is information about the program structure itself such as classes, methods and attributes. A class, for example, has intrinsic metadata like its name, superclass, interfaces, methods and attributes (Forman and Forman 2005). In metadata-based frameworks, the developer also must define some additional application-specific or domain-specific metadata.

Metadata-based frameworks can be defined as frameworks that process their logic based on the metadata of the application classes (Guerra et al. 2009). In those classes the developer must define additional domain-specific or application-specific metadata to be consumed and processed by the framework.

The use of metadata changes the way frameworks are built and how they are used by software developers. In an interview motivated by the fifteen years of the book “Design Patterns: Elements of Reusable Object-Oriented Software” (Gamma et al. 1994), when asked about how the metadata approach replace or complement the patterns in the book, Erich Gamma answered the following (O’Brien 2009):

> *“While they complement the patterns in DP (referring to the patterns in the book) it can indeed be the case that meta-programming can replace the design pattern used in a design. The evolution of JUnit 3 to JUnit 4 comes to mind. JUnit 3 was a small framework that used several patterns like Composite, Template Method and Command. JUnit 4 leverages the Annotations meta-programming facilities introduced in J2SE 5.0. The use of the patterns disappeared and the framework evolved into a small set of annotations plus a test runner infrastructure that executes the annotated Java code.”*

The developer's perspective in the use of those frameworks has a stronger interaction with metadata configuration than in method invocation or class specialization. In traditional frameworks, the developer must extend its classes, implement its interfaces and create hook classes for the behavior adaptation. He/she also has to create instances of those classes, setting information and hook class instances. Using metadata-based frameworks, programming focus is on declarative metadata configuration and the method invocation in framework classes is lesser and localized.

The metadata consumed by the framework can be defined in different ways. Naming conventions (Chen 2006) uses rules to create the name of classes and methods that have a special meaning for the framework. As an example, it is possible to cite the Java Beans specification (JavaBeans 1997), which uses method names beginning with 'get' and 'set', and the JUnit 3 (Massol and Husted 2003), which interprets methods beginning with 'test' as test cases implementations. Ruby on Rails (Ruby et al. 2009) is an example of a framework known by the naming conventions usage.

Conventions usage can save a lot of configurations but it has a limited expressiveness. For some scenarios the metadata needed are more complex and naming conventions are not enough. An alternative can be setting the information in a programmatically way in the framework, but it is not used in practice in the majority of the frameworks.

Another option is the metadata definition in external sources, like XML files and databases. The possibility to modify the metadata at deploy-time or even at runtime without recompiling the code is an advantage of this type of definition. However, the definition is more verbose because it has to reference and identify program elements. Furthermore, since the class definition and metadata configuration are in separate files, the distance between them is increased, which may be non-intuitive for some developers.

Another alternative that is becoming popular in the software community is the use of code annotations, which is supported by some programming languages like Java (JSR175 2003) and C#

(Miller 2003). Using this technique the developer can add custom metadata elements directly into the class source code, keeping this definition less verbose and closer to the source code. The use of code annotations is called attribute-oriented programing (Schwarz 2004).

This paper presents a study that has analyzed some existent metadata-based frameworks and abstracted the solution which they provide. These patterns abstract what sort of problems can be solved in an architecture by metadata-based framework. They present for each situation the benefits and drawbacks that the use of metadata can bring to an application. The lack of study about this kind of framework reduces the opportunities of using metadata to solve problems where it is appropriate. Architectural patterns are useful for the identification of these situations in an application architecture. As a result, based on the consequences resulting from using metadata-based frameworks in each scenario, an architect should be able to take a more conscious decision about the use or the creation of a framework with that approach for a given architectural component.

2. PATTERN CATALOG DESCRIPTION

The scenarios where metadata-based frameworks can be used were abstracted and documented in the collection of architectural patterns presented in this section. As part of this research, current frameworks were observed, and also new ones were created to experiment the use of metadata for the same purpose of the pre-existent ones, but in different domains. It is important to observe that some of the metadata usage on the observed frameworks could not be abstracted into patterns. Therefore, this collection does not mean to cover all possibilities.

The observations have found out four usages of metadata to be recurrent in frameworks. They are documented in the following patterns:

- **Entity Mapping** proposes the use of metadata to map an application class to another representation of it.
- **Configured Handler** Method proposes the use of metadata to mark methods that should handle events.
- **Crosscutting Metadata Configuration** proposes the use of metadata to configure variations on a crosscutting behavior.
- **Metadata-based Graphical Component** proposes the use of metadata to generate graphical components based on application classes and to lead the interaction with them.

The pattern format adopted uses some standard pattern elements like *Motivation, Problem, Forces, Solution, Consequences, Known Uses, Example* and *Related Patterns*. The element *Metadata Definition* is introduced to present some implementation details about how the metadata can be defined in each pattern context. It is also important to highlight that the frameworks presented in the *Example* element are either developed by one of the authors or by a student advised by him.

3. ENTITY MAPPING

Also known as Metadata Mapping, **Representation Mapping**, **Metadata Converter**

Motivation

The same entity in the application is often represented by using different schemes and paradigms. For instance, a business entity, which is represented in the system as a class, can be persisted in a database or sent by the network serialized in a XML document. Likewise, business processes, which are represented in the system as methods, can be also available as web services or adapted to another signature compatible with a legacy API.

The translation between those representations, which should be provided by the application, can be a complex task. For instance, due to differences between two paradigms, the granularity between the entities can be different, which means that one instance of an entity in a representation can be many in another. As a further example, attributes and relationships can also have distinct names and be represented in different formats. A well-known instance of this issue is the differences between the object-oriented and relational paradigm (Fowler 2002).

In a system with a large number of entities that should be translated, this task can become a bottleneck in the team productivity. The created translation code is usually coupled with both representation schemes, being sensible to changes in both sides. Reflection (Buschmann et al. 1996) can be used when the translation rules are uniform, even though usually there are particularities and exceptions for each entity.

Problem

How to map an application class to a different representation of the same entity reducing the coupling and the code amount?

Forces

– Different representation paradigms can have huge conceptual differences both in entity and relationship representations.
– Manual translation between entities is flexible since it can handle any translation rule. Anyway, given that it should be created for many system entities, it could become a hard-working task.
– Using **Reflection** a general translation routine can be created, but it should use the same algorithm for all entities, which could not be flexible enough for the system particularities.
– Naming similarities are common among the representations and can be used in the mapping. However, exceptions to this rule must also be handled.
– A source code that deals with entities on two paradigms is sensible to changes in both representations.
– When one representation is external to the system, it is usually accessed using a specific API, which can disable the use of a direct transformation.

Solution

Entity Mapping uses additional metadata in the application class in order to configure the translation rules to the other representation. A metadata-based framework introspects the application class structure and reads the translation metadata to configure internally how it should be performed. Also, a reflective algorithm that implements the translation rules interacts dynamically with the application class and invokes methods on an API used to access the other representation. Figure 1 illustrates this structure.

Examples of this API are JDBC for the access to databases or DOM to access XML documents. This API can be a simple class with getter and setter methods from which the application needs to become decoupled. The name “Other Representation” is used to refer to entities not represented in the object-oriented paradigm.

Figure 2 represents a high-level abstraction of the sequences used by the framework to execute the translations. When the information should be translated from the other representation to the application class schema, the framework must invoke the methods from the API. So it sets information based on the translation rules by using introspection in the application class instance. As a result, if the translation should be in the inverse direction, the framework must access the application class instance information and invoke the appropriate methods on the API.

This structure decouples the application from the specific API to access the other representations, since it is encapsulated by the framework. The application also had a more localized contact with the framework API, since the main rules are defined using the metadata.

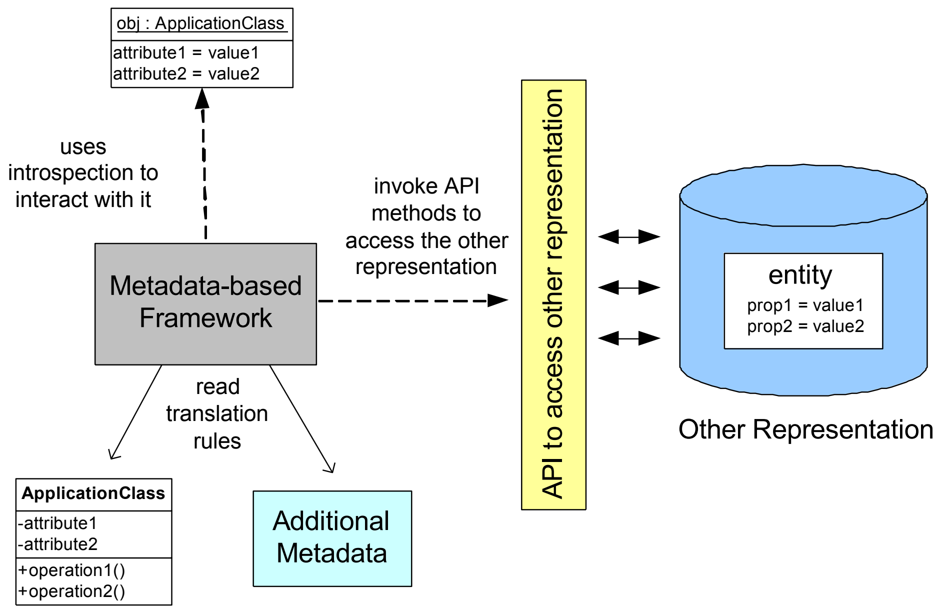

Fig. 1. Entity mapping using metadata.

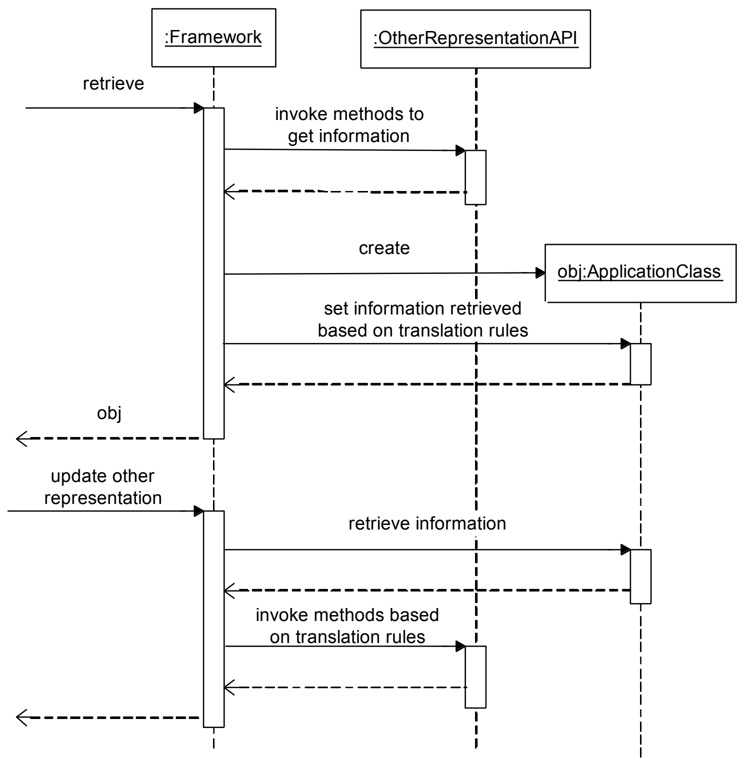
:Framework
:OtherRepresentationAPI
retrieve
invoke methods to get information
create
obj:ApplicationClass
set information retrieved based on translation rules
obj
update other representation
retrieve information
invoke methods based on translation rules

Fig. 2. Sequence for metadata-based entity.

The translation executed by the framework can be invoked directly from an application client or be triggered transparently in application class invocations, by using a **Proxy** (Gamma et al. 1994) or an aspect (Kiczales et al. 1997). Using the second approach, the class representation is monitored for modifications and when a method that modifies information is invoked, the framework updates it in the other representation as well.

However, when the entity translated is a method invocation, this structure should be a little different. In this situation, the framework acts like an **Adapter** (Gamma et al. 1994), which translates the method calls, the parameters and the return based on metadata. The sequence of the method called translation is presented in Figure 3. Although the dynamics are different, the metadata purpose remains for the translation among program entities.

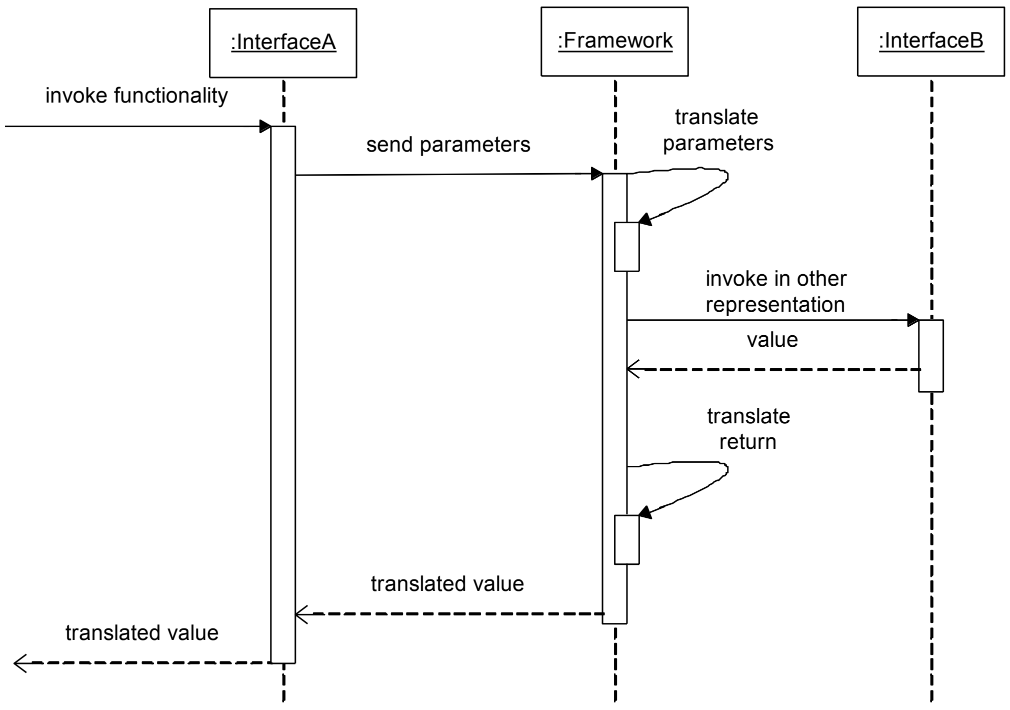

Fig. 3. Translation of method invocations.

Consequences

- ⊕ Changes in one representation can be easily handled since it demands modifications only in the metadata mapping.
- ⊕ The declarative metadata definition is usually easier to develop and maintain than specific translation logic.
- ⊕ The application classes become decoupled with the specific APIs to access the other representation.
- ⊕ When both representations are being developed, it is possible to have a tool that generates one of them based on the other representation plus metadata.
- ⊕ The translation rules can be adapted for each class using the additional metadata, which increases the framework reuse potential.
- ⊖ In situations that the framework does not support the translation requirements directly, to create a framework adaptation or to work around it can be more hard-working than to create the translation code directly.

– ⛔ The class and property in the mapping are referenced by using their names and little typing mistakes can cause defects hard to locate. This can be minimized with a tool support and descriptive error messages.

Example

This architectural pattern was explored in the implementation of the **MentalLink** framework (Guerra et al. 2008a), which uses annotations to map between classes and ontology instances (Davies et al. 2003). It uses the API of the Jena framework (Carroll et al. 2003) to interact with the ontology through the class `Model`. This framework was developed by one of the present work's author during the research. In turn, it is an academic implementation and does not support the translation of all the ontology concepts. However it can be considered a proof of concept that those entities can be mapped by using metadata.

The framework entry point is the class `ObjectModel` that contains the two main methods used to retrieve and set information in the ontology. Further, the method `addObject()` is used to add the application object's information into the ontology. The method `getObject()` retrieves an ontology instance as an application object.

The framework provides mapping annotations to be used on the application classes. The `@Resource` annotation maps a class to an ontology entity and the `@ResourceName` annotation indicates which property is used to store the ontology instance name, used for its identification. Additionally, the `@Property` annotation is used to map entity properties, including primitive values and other ontology instances. Since the inheritance is more granular in ontologies and used for classification, the annotation `@Classification` is used to map ontology classes to attributes values in the object.

To exemplify the framework mapping, the ontology illustrated in Figure 4 is mapped to the class whose source code is presented in Figure 5. In the ontology, the entity `Person` is extended by the classes Man, `Woman`, `Parent` and `Child`. The property `parentOf` of `Parent` connects it to a `Child`. Further, the entity `Person` is mapped to the class `People`, whose property name receives as a value the ontology instance name. The attribute `sex` receives the value `'Male'` if the `Person` is an instance of `Man` and `'Female'` if it is an instance of `Woman`. The attribute `children` receives a list of `People` which the instance is `parentOf`, that are recursively loaded from the ontology.

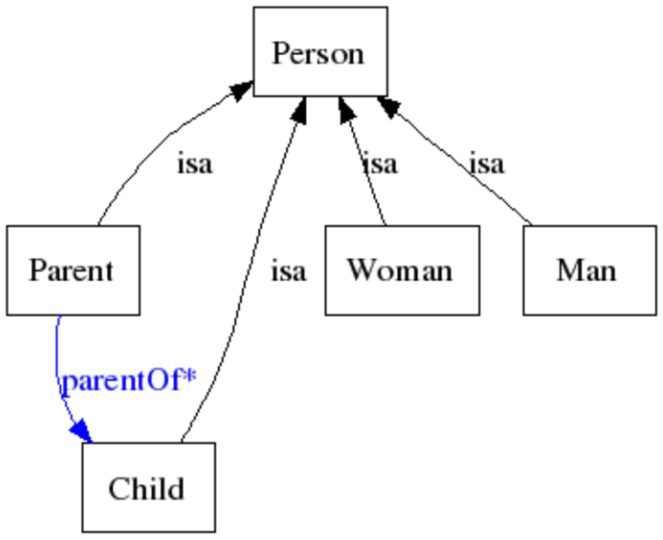

Fig. 4. Graphical representation of the family ontology.

```
package org.mentallink.example.ontology;

@Resource(instanceURI="http://family-instances/",
       ontClass="http://family/Person")
public class People{

    private String name;
    private String sex;
    private List<People> children =
       new ArrayList<People>();

    @Property(value="parentOf",
           isResource=true,
           baseURI="http://family/",
           instanceClass=Pessoa.class)
    public List<People> getChildren() {
        return children;
    }
    public void setChildren(List<People> children) {
        this. children = children;
    }
    public void addChildren(People child){
        children.add(child);
    }
    @ResourceName
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    @Classification(baseURI="http://family/",
           values={"Male","Female"},
           ontClasses={"Man","Woman"})
    public String getSex() {
        return sex;
    }
    public void setSex(String sex) {
        this.sex = sex;
    }
}
```

Fig. 5. Example of an application class mapped to an ontology entity.

Known Uses

Hibernate (Bauer and King 2006) is a framework which uses metadata for object-relational mapping. **JPA** (JSR220 2006) is a standard API for the same purpose that is implemented by **Hibernate**. Both of them support mapping of the object-oriented paradigm concepts, such as inheritance and composition, to a relational database.

JAXB (JSR222 2006) is a standard API for mapping Java classes to XML entities. Classes are mapped to a XML schema by using annotations. It also provide a tool that generate the annotated classes based on a XML Schema.

It can also be taken as example the **JAX-RS** (JSR314 2009) that is the standard Java API for RESTfull web services (Fielding 2000), which uses annotations to map methods on a class to web services. It is an example for the mapping between functionalities. The JAXB (JSR222 2006) can be used to annotate types in parameters and return to configure their conversion to XML.

Metadata Definition

The use of code conventions is a good choice for saving configurations since the representations often use the same name for entities and its properties. Nevertheless, it is also important for the framework to support an additional metadata definition that is able to represent more complex information.

The use of code annotations is common in frameworks with metadata mapping. Some developers consider that the code readability is improved since the mapping information is close to the application class definition. Because almost all of the properties must be mapped, a less verbose metadata definition strategy also helps in productivity. When complicated translation rules should be defined, then annotations end up polluting the source code sometimes. This source code pollution can also occur if the same class was mapped for two different representations by using different annotation schemes.

In scenarios where the framework needs to map classes for two different representations or legacy classes need to be mapped, annotations are not the best choice. For these requirements, the use of external metadata definition, such as XML files, are more appropriate. This kind of definition also brings the advantage to allow mapping modifications without recompiling the source code. Accordingly, another good alternative is the combination of XML files and annotations.

In these frameworks is very common to have tools that generates one representation definition based on the other representation plus metadata. When this is possible, it helps to improve even more the team productivity (Wada and Junior 2009).

Related Patterns

This pattern is related to **Adapter** (Gamma et al. 1994) which is used to translate calls between two different interfaces. This pattern proposes a more flexible and general solution that uses the metadata mapping to configure the translation rules, allowing a framework to be a reusable solution between two different paradigms. Accordingly, the framework can be used in the creation of adapters among these interfaces.

Metadata Mapping (Fowler 2002) can be considered a specialization of this pattern, since it documents this solution only for object-relational mapping. Moreover, mapping frameworks usually allow the mapped classes to have one or more **Configured Handler Method** to be invoked at some determined points in the translation process..

4. CONFIGURED HANDLER METHOD

Also known as **Metadata-based API**, **Metadata-based Invoker**

Motivation

Observer (Gamma et al. 1994) defines a method to receive events or notifications generated by the observed subject. It usually overrides this method from an interface that the class must implement or from a superclass that the class must extend, coupling the handler class with one of them. Besides, those events usually come with information for the handler method to verify if it should be handled.

Some applications have a coarse grained event handling providing only few methods to handle many event types. The class interested in only a subset of those events usually contains conditional logic to select the desired ones. If that class must trigger processes related to more than one concern, these behaviors should be implemented in the same method, reducing class cohesion.

On the other hand, if the event handling is fine grained, the interface has a large number of methods that distinguish the events to the handler class. In this scenario, the classes usually need to handle only a few event types, leaving many methods empty in the class implementation. Notwithstanding, sometimes the event data must be used to determine if it should be handled, which is too fine grained to be defined in an interface.

Problem

How to enable fine grained event handling without proliferating empty methods or reducing the class cohesion?

Forces

– A handler method that overrides methods from classes or interfaces, becomes coupled with them and the types received as parameters.
– If an interface defines many methods for a fine grained event handling, a class that needs only a few ends up with a lot of empty methods.
– If the interface defines one method for a coarse grained event handling, a class must use conditional logic to select only the desired ones to handle.
– If the interface defines one method for a coarse grained event handling, a class that handles events for two distinct concerns in the same method reduces its cohesion.
– An interface cannot define granular handling methods that distinguish the events by their information.
– Handling methods usually receives all the event information even if it does not need all of them.

Solution

Configured Handler Method uses additional metadata in a method to enable its invocation only for events that it should handle. The metadata also can be used to identify what information should be passed as parameters to the method. In brief, the metadata-based framework generates or receives an event and invokes all methods on the registered classes whose metadata match the event information. This structure is presented in Figure 6.

Figure 7 presents a sequence diagram illustrating how the framework invokes the handler methods. When an event is received or generated by the framework, for each handler method registered to receive callbacks, it is verified if the metadata matches the event information. In positive cases, the event data is adapted to the parameter types and the method is invoked.

Each method's metadata describes which event type it should handle. It can also describe conditions based on the event information, by providing a fine grained API to restrict the handled events. In addition, the method parameters can also receive metadata to indicate what event information should be passed to the method in that parameter. As a result, it avoids the method to receive information that it would not use in its processing.

This pattern can be used for the event handling strategy of a larger metadata-based framework. Methods in the application classes received can handle events that are generated by the framework itself as callbacks steps of its processing. However, another possibility is the framework to implement an interface to receive external events and call the **Configured Handler Method**s based on metadata. In this scenario, the framework acts like a **Mediator** (Gamma et al. 1994), receiving calls in the standard API format and invoking methods configured with metadata.

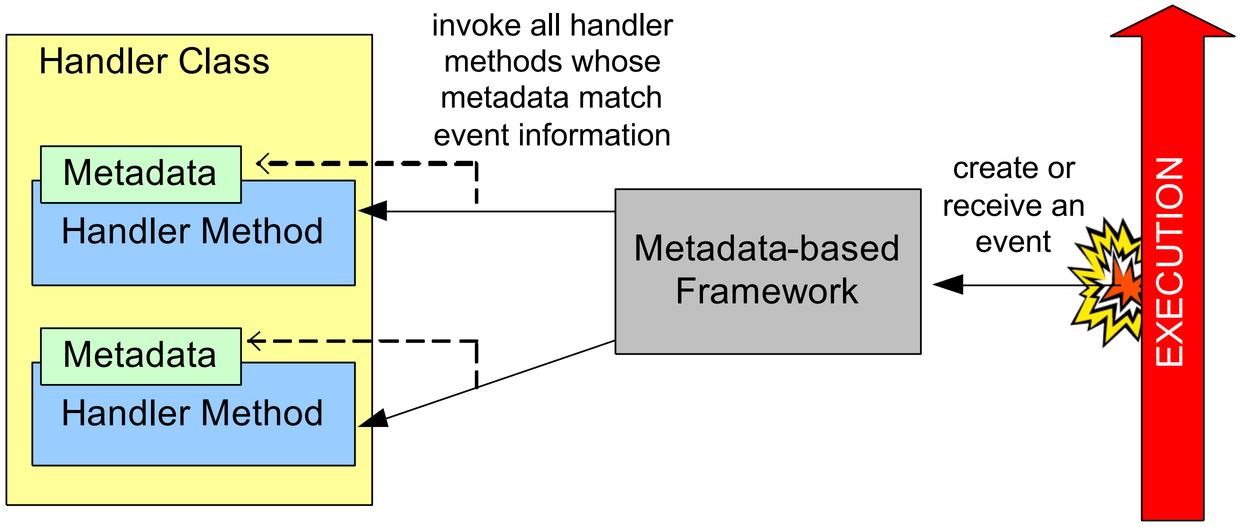
Handler Class
Metadata
Handler Method
Metadata
Handler Method
invoke all handler methods whose metadata match event information
Metadata-based Framework
create or receive an event
EXECUTION

Fig. 6. Representation of the invocation of a Configured Handler Method.

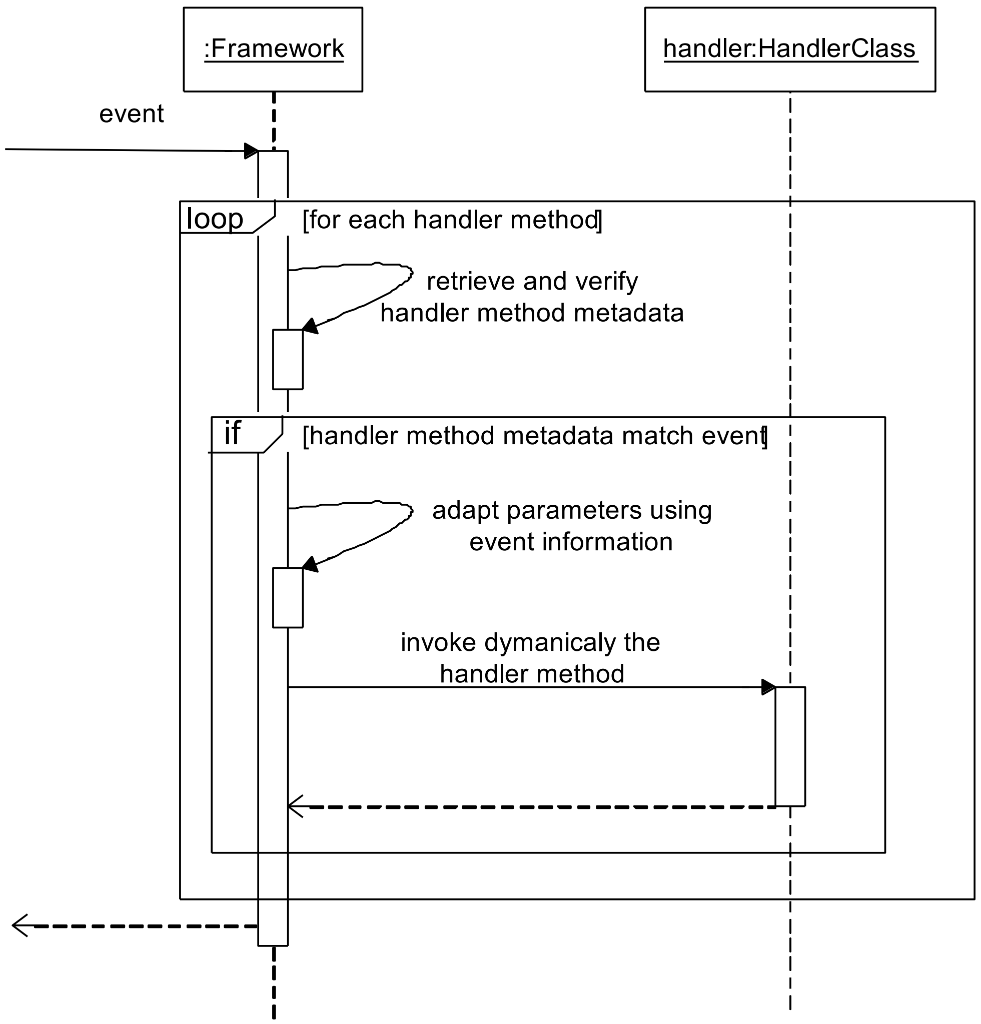
:Framework
handler:HandlerClass
event
loop
[for each handler method]
retrieve and verify handler method metadata
if
[handler method metadata match event]
adapt parameters using event information
invoke dymanicaly the handler method

Fig. 7. Sequence of how the framework invoke the handler methods.

Consequences

- ⊕ More than one method can handle the same event in the same class, allowing the separation of concerns between them and improving class cohesion.
- ⊕ The application class does not need to implement interfaces or extend classes from the event provider to handle its events, increasing the decoupling.
- ⊕ The handler methods can receive only the parameters needed in the processing, avoiding the declaration of unnecessary parameters.
- ⊕ The conditional logic inside handler methods can be reduced by the metadata usage to define the invocation rules for each method.
- ⊕ APIs based on **Configured Method Handler**s can add new operations without breaking the existing classes.
- ⊖ The performance can be reduced since the methods are invoked using introspection, which is a problem when the number of concurrent events is high.
- ⊖ Complicated rules that require many metadata configurations can be more complicated to develop and maintain than the conditional logic.
- ⊖ Rules to restrict the events must be predicted by the framework and incorporated to the metadata schema, which can make exceptional situations hard to develop.

Example

The framework **JColtrane** (Nuccitelli et al. 2010) uses this pattern to execute XML parsing based on SAX events (SAX 2004). Using SAX, a handler class must implement an interface that receives events like the document start, an element start, an element end, the document end and characters found inside an element as well. Those methods receive all events generated and the source code must use conditional logic to distinguish them.

JColtrane receives those SAX events and call method handlers whose annotations define conditions for what XML piece should be handled. It is an open source framework developed as part of a graduation final work advised by the present work's author. Such a framework won in 2008 the programming contest of *“The Search for the XML Superstar”*, sponsored by IBM. There are some reports of people already using it for XML parsing in production applications.

Figure 8 presents an example of how **JColtrane** can be used to define a fine grained event handling. In the annotation `@EndElement`, the attribute `tag` determines that it should only handle the tag name and the attributes that this tag must have an attribute `lang` equals to `pt`. The annotation `@BeforeElement` configures a condition that the handled tag name must be inside a tag `<paper>`. The method parameters also receive annotations to configure what information each one should receive. In the example it receives the text inside the handled tag.

JColtrane helps in the parser code organization, enabling each tag to be handled in a different method. The annotations allow each method to handle only the tags of interest and to receive only relevant information in parameters. As consequences, the conditional logic reduces inside the methods and the method's cohesion increases. **JColtrane** also supports the creation of custom annotations that can define new conditions for the method execution.

```
@EndElement(tag=”name”, attributes={@ContainsAttribute(name=“lang”,value=”pt”)})
@BeforeElement(tag=“paper”)
public void handlePtPaperName(@Body String name){
   //method implementation
}
```

Fig. 8. Example of an annotated XML handler using JColtrane.

Known Uses

Based on **EJB 3** and **JPA** specifications (JSR220 2006) it is possible to add methods in the EJBs or in persistent classes to handle life cycle events. Those methods receive a simple annotation and are called by the EJB container when that life cycle event happens. They can also be defined in a separate class, which is also configured using metadata.

Genesis (Genesis 2010) is a framework that includes functionalities for binding actions of graphical components to controller class methods. Configurations are guided by annotations on the controller class. For instance, by using the annotation `@CallWhen` it is possible to define conditions for execution based on values and states from other graphical components.

VRaptor (VRaptor 2010) is an action-based controller framework for web applications. It supports the REST style (Fielding 2000) to build web applications, in which the URL represents the resources and the HTTP method the operation. In the handler class, each method contains an annotation that configures the path and the HTTP method that it should handle. Furthermore, those annotations can have information about how the method parameters should be populated..

Metadata Definition

For event handling purposes, the metadata proximity to the source code should be as much as possible. Accordingly, arguments for using an external metadata source are unlikely in the event handling scenario, such as deploy time configurations and usage in legacy code. For instance, it is unlikely for a legacy class method to need to receive an event that it does not receive yet. Another example is that it is not common to find requirements that demands changes at deploy time for this kind of metadata.

For those reasons, the use of annotations and code conventions is indicated for metadata definition in the context of event handling. However, XML definition can also be found in some frameworks, especially those that have born before the annotations support on the Java language. The conventions can be used to define more coarse grained information while annotations can refine it with more specific conditions. In Java language the parameter names are not available at run time through the native Reflection API, which prevents the use of naming conventions for parameters metadata..

Related Patterns

In the scenario of the framework receiving external events, this pattern is related to **Adapter** (Gamma et al. 1994) since it adapts from a fixed interface to annotated methods. In this case it can also be related to **Mediator** (Gamma et al. 1994) because the framework mediates between components that generates the events and their handlers.

This pattern can be used in conjunction with the other patterns in this collection. The **Configured Handler Method** can be located in the application classes received by the framework to handle events at specific execution points. It allows the creation of new hook methods in those classes that can be executed as part of the framework's processing logic.

5. CROSSCUTTING METADATA CONFIGURATION

Also known as Metadata Interception Configuration, Crosscutting Behavior Configuration

Motivation

Some patterns can be used to add functionality or substitute an existent class transparently in an application. For instance, **Decorator** (Gamma et al. 1994) attaches additional responsibilities to an object dynamically encapsulating its access through a class that implements the same interface, executes additional responsibilities and delegates the execution to the original object. Another example is **Proxy** (Gamma et al. 1994) that transparently provides a placeholder for another object and controls the access to it. Those usually implement some functionality that crosscuts the original class behavior.

There are some reflection techniques that allow the creation of dynamic proxies, in other words, classes that can implement an interface at runtime and encapsulate the access as a Proxy to any object. For instance, Java Reflection API (Forman and Forman 2005) provides a virtual machine native functionality for dynamic proxy creation and CGLib (CGLib 2010) is a framework that provides the functionality to manipulate bytecode and create the proxy class at runtime. Another alternative is the use of aspect-orientation to intercept a method execution and execute additional functionality.

In all those dynamic alternatives it is necessary to create a more general source code that intercepts all method executions. It receives as parameters the methods being invoked with their intrinsic metadata and the parameters passed to it. This is the only information that can be used to differentiate the execution between them. If the processing needs to be differentiated between method groups, different proxy classes must me created and associated with the respective classes and methods. Furthermore, in aspect-oriented frameworks, an abstract main aspect must be extended to implement the specific behavior for each possible variation (Guerra et al. 2008b).

The problems arrive in this scenario when the variations are numerous and fine grained. One of the problems is the number of main behavior specializations for different methods and other problem is to manage what behavior should be executed in each method.

Problem

How to differentiate and configure for each method the execution of a transparent crosscutting behavior?

Forces

- Traditional proxies and decorators are suitable for a set of classes that implements the same interface, but when the interfaces are different, many classes must be created and their source codes are usually similar.
- Using dynamic proxies it is possible to create proxies that implements any interface at runtime, but many proxy classes should be created when the behavior should vary for different methods.
- In aspect-oriented frameworks it is necessary to create a new aspect that extends an abstract one for each behavior variation, which is a problem especially when many variations are possible.
- When creating many proxy specializations it is hard to manage what implementation is attached in each method or class since it is only visible in the line of code that the class and the proxy are associated.
- In proxy functionalities which different behaviors can be combined through a recursive composition, the number of different behaviors grows exponentially with the number of possible variations (Guerra et al. 2008b).

Solution

Crosscutting Metadata Configuration uses additional metadata to differentiate the behavior of software components that add a crosscutting behavior transparently to an application class. A dynamic proxy encapsulates the access to this class and invokes a metadata-based framework in each execution. In succession, the framework uses the application class metadata to define what behavior should be executed. Also, the dynamic proxy's role can be performed as well by an aspect. This structure is elaborated in Figure 9.

To create the proxy encapsulating the application class transparently, a good practice is to use a factory to create its instances. Figure 10 illustrates the sequence of the application class instance creation. It is visible that the factory creates the application object setting it in the dynamic proxy, which is returned as an instance of the class interface type. Nevertheless, for aspect-oriented frameworks weaved at load-time or at compile-time this step is not necessary.

Figure 11 introduces a sequence diagram representing the metadata-based framework being invoked by the proxy. The application class client invokes a method in the proxy, which is working as a placeholder for the application class. Then the proxy invokes the framework by passing invocation information, such as method and parameters, and after that it executes the original method. Depending on the domain, the framework logic can also be executed after the method invocation or the application class can be called from the framework itself. In an aspect-oriented framework, those possibilities are represented by the advice being before, around or after.

The access to method metadata allows the framework to customize its behavior for each one of them. Class metadata can be used for default metadata values or for general configurations. Parameters can also receive metadata to provide a special meaning for the framework, which can even change them for the application class invocation. Besides, for some framework domains, such as security, it can also prevent the application method to be executed.

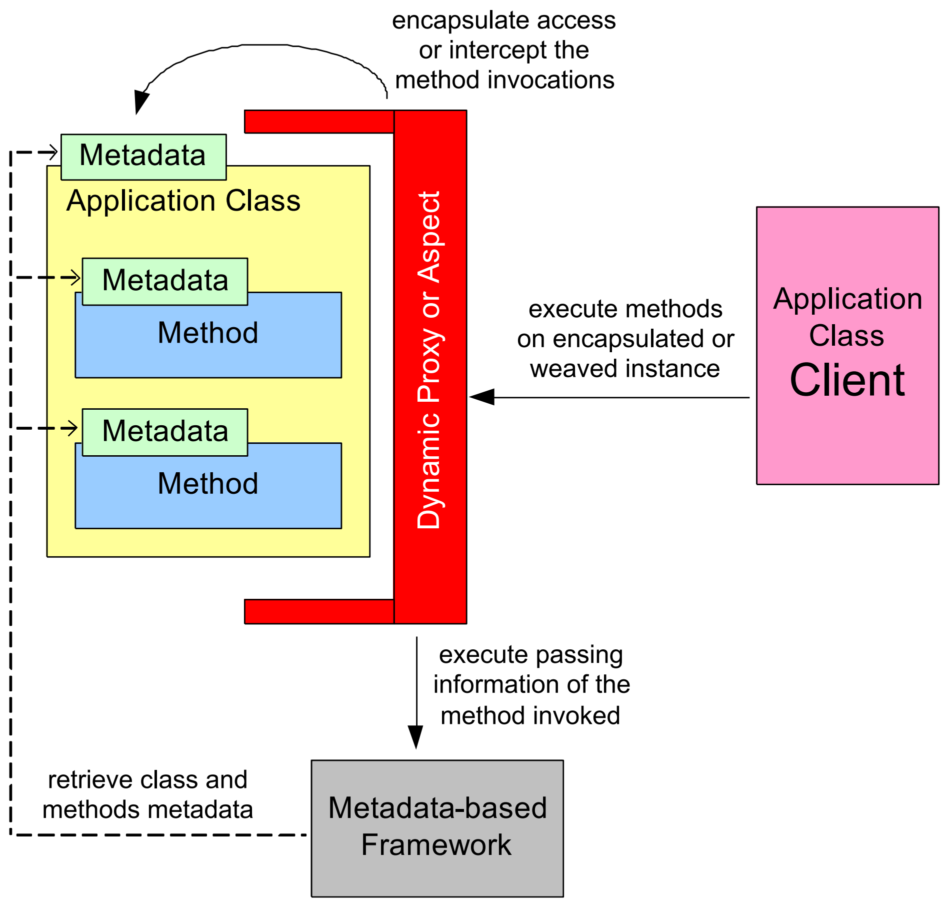
encapsulate access or intercept the method invocations
Metadata
Application Class
Metadata
Method
Metadata
Method
Dynamic Proxy or Aspect
execute methods on encapsulated or weaved instance
Application Class Client
execute passing information of the method invoked
retrieve class and methods metadata
Metadata-based Framework

Fig. 9. The representation of Crosscutting Metadata Configuration.

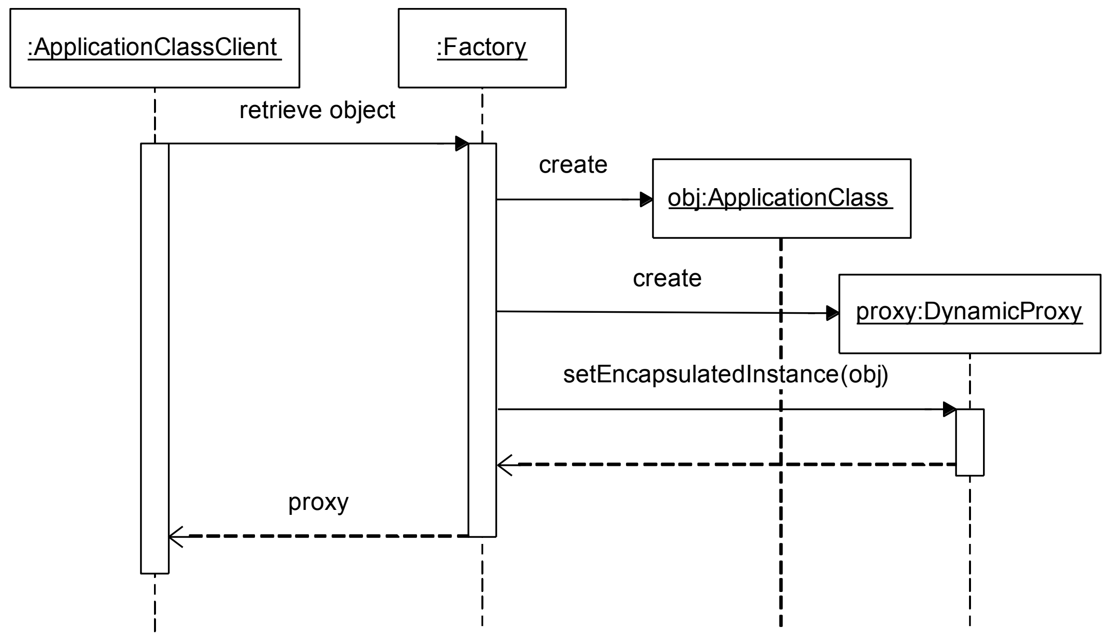

Fig. 10. Sequence of proxy creation using a factory.

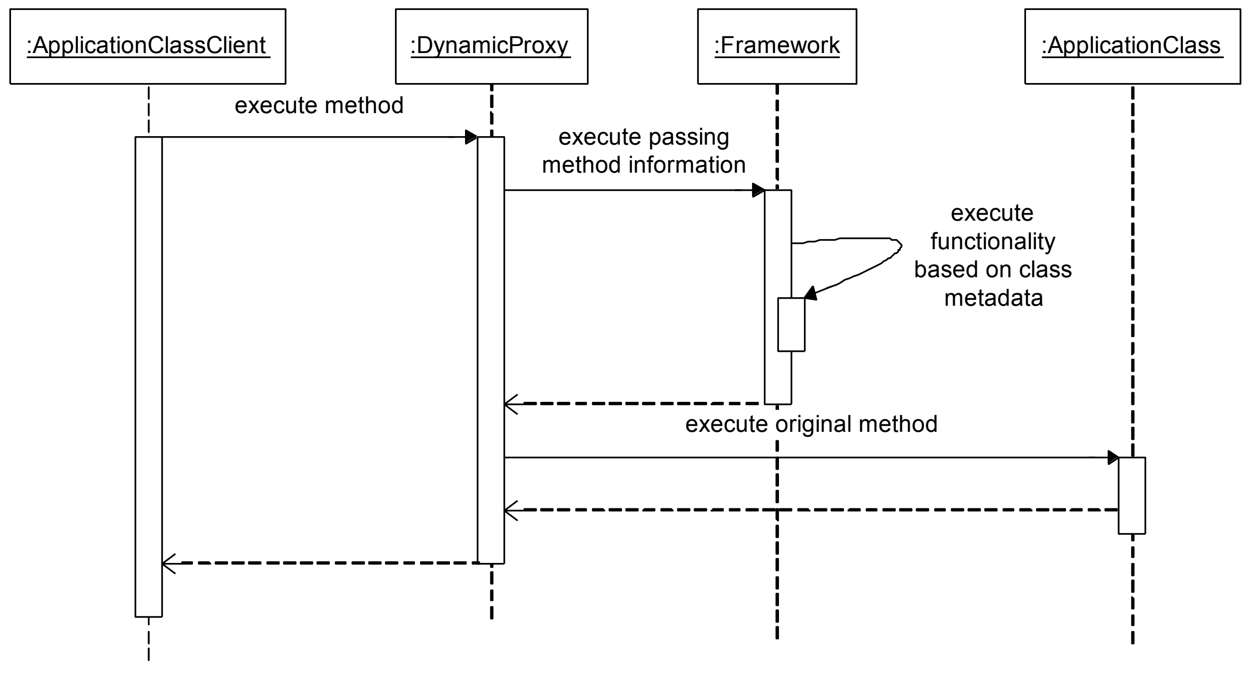

Fig. 11. Sequence of the proxy invoking the metadata-based framework.

Consequences

– ⊕ The same proxy can have a custom behavior for each method invoked.

- ⊕ The number of proxy classes or advices can be reduced to one, since their behavior can vary through metadata configuration.
- ⊕ The management of what is the framework behavior for each application method is facilitated since it is centralized on metadata.
- ⊖ The use of annotations in application classes can break part of their obliviousness since it explicit adds crosscutting information to the class.
- ⊖ External metadata definition for a large number of methods can be hard-working for a verbose definition format.
- ⊖ Changes in configurable metadata at runtime must be synchronized with the framework executing processes.

Example

An example of this pattern usage is the framework **Metadata-based Logger** (Guerra et al. 2008b), which is an aspect-oriented framework that uses metadata to configure the logging strategy for each method. The metadata is used to configure where the information should be registered, such as in a file or in a database, and also what should be logged, like method name, parameters, return and exceptions. It is important to highlight that this is an academic framework developed by a student advised by present work's author to explore the development of metadata-based aspect-oriented frameworks.

This framework supports metadata definition using annotations and XML files, as presented respectively in figures 12 and 13. The information to be logged can be combined, creating a large number of possibilities. The log location can also be configured and combined with new locations that can be added in the framework. To create those possibilities with traditional aspect-oriented framework techniques, many advices must be created.

```
@LogMarker(
  logInfo = {LogData.METHOD, LogData.EXCEPTION,
             LogData.RETURN_TYPE},
  logLocation = {LogData.FILE}
)
public Parent testLog(Integer i)
               throws NullPointerException, IOException {
   ...
}
```

Fig. 12. Logging metadata definition using annotations.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<logmarker>
   <method signature="Parent logging.LoggerApplication.testLog(Integer)">
      <info>
         <value>METHOD</value>
         <value>PARAMETER</value>
         <value>RETURN_TYPE</value>
         <value>EXCEPTION</value>
      </info>
      <type>
         <value>DATABASE</value>
      </type>
   </method>
</logmarker>
```

Fig. 13. Logging metadata definition using XML files.

Known Uses

In the **EJB 3** specification (JSR220 2006), the EJB container encapsulates the access to EJB components. The container uses annotations and configurations on a XML descriptor to determine how it should behave for each method. Transaction management and access control can be considered examples of characteristics that can be configured using metadata.

Hibernate (Bauer and King 2006) uses proxies in the persistent classes to synchronize their state with the persistent data in the database. For example, properties with 'lazy-loading' metadata are retrieved from the database when they are first accessed by the application. The proxy interacts with other **Hibernate** components in order to execute its functionalities.

SwingBean (SwingBean 2008) provides a feature for binding the application classes to forms and tables. The application class access is encapsulated by a proxy, which sets information in the graphical components by the invocation of setter methods and then retrieves information from them each time a getter method is invoked. This dynamic proxy accesses the **SwingBean** components which use metadata to get, set and convert information to the type expected by the application class.

Metadata Definition

For crosscutting functionalities, it is often a requirement to enable metadata configuration at run time. Facing this requirement, annotations can only be used associated with other external metadata definition approach. If the metadata can be altered by the application itself, a good choice is to use database storage. However, if it would be changed only by developers, a configuration file probably should be enough.

In applications that use crosscutting metadata-based frameworks, there are usually method groups that have the same behavior. Providing ways to configure default metadata values for the application or for a class can enable only the configuration of exceptions, saving a considerable amount of metadata lines of code.

In this scenario, the annotation usage in application classes breaks part of framework's obliviousness (Filman and Friedman 2000), since classes will contain explicit information about crosscutting concerns. An alternative in this situation is to use domain annotations (Doernenburg 2008), which express domain-related metadata and can be mapped to the framework annotations (Perillo et al. 2009). As a result. annotations can be retrieved by the framework through the domain annotations directly or by a tool, such as Daileon (Perillo et al. 2009), which might be used to translate annotations at compile-time.

Related Patterns

This pattern is obviously related to **Proxy** and **Decorator** (Gamma et al. 1994), but it can be also used with any structure that adds functionality transparently in an application class, such as aspects and container-based components.

It can be used with **Entity Mapping** to encapsulate one representation with a proxy that updates automatically the other one. If the mapping is between methods and services, this pattern can be used to add functionality to the method in the adaptation. It can also be used in **Metadata-based Graphical Components** to synchronize the application class with the graphical component.

6. METADATA-BASED GRAPHICAL COMPONENT

Also known as Metadata-based GUI

Motivation

Graphical components usually represent application classes for the final user to visualize and to interact with them. Since each application class has its own attributes and presentation particularities, this piece of the graphical interface is hard to be reused for different classes even in the same application. Therefore, it includes not only the code to create the graphical interface but also to retrieve and set information from it. For classes with a large number of attributes, this implementation can become a bottleneck for the development team productivity.

More than one screens from the application interface usually share a similar internal logic, such as when a button is pressed a form information should be retrieved and persisted. Due to the difficulty to reuse code that interacts with specific application classes, it is also hard to create a graphical interface that can be reused for more than one application class.

A common approach is to use a drawing tool to generate the graphical interface source code. Nevertheless, the drawbacks of this practice are: the screen still has to be hand-made; one screen cannot reuse characteristics from another; the tool may not recognize generated code from another tool or if it has been altered manually; and it do not helps in the interaction with the application classes. Also there are available tools that generate the graphical interface based on its static description, using for example a XML document. In this alternative approach the problems with application class interaction and the lack of reuse remains.

Solutions that do not reuse logic can be easy to create or generate, but they are hard to maintain and evolve. Changes in the interface or in the application classes impacts in many different points, such as components creation, information setting and retrieving. A general change in the application screens is also hard to be made, since it had to be performed individually in every graphical interface.

Problem

How to create a graphical component that can be reused for interaction and representation of several application classes?

Forces

- Generated code for graphical interfaces can be easy to create, but modifications in the application classes structure demands changes in many locations.
- Screens created using drawing tools and interface descriptors are hard to be reused and extended.
- General changes in interfaces are hard to do when the screens do not share logic among them.
- The source code that exchanges data between application classes and graphical components are usually repetitive and, for a large number of attributes, can impact on the development team productivity.
- The piece of graphical interface that depends directly on the class structure, like forms and tables, are hard to be reused.
- Reflective solutions to generate the graphical interface can generate a basic interface but it rarely fulfills the usability requirements in real applications.
- Graphical interfaces usually implements many details about single fields, such as validation and formatting.

Solution

Metadata-based Graphical Component uses additional metadata to generate graphical components based on the application class structure. The metadata is also used to customize the graphical components behavior and visualization, besides to retrieve and set information into them. Additionally, the application interacts with the framework using its own classes, which eliminates the need for setting or getting attribute by attribute. Moreover, the framework usually encapsulates the access to an existent component or a combination of them. Figure 14 illustrates the structure of this pattern.

Figure 15 presents a sequence diagram of how the framework interacts with the application classes and the graphical components. When the application initiates the framework, it creates the encapsulated graphical components configuring and arranging them based on the metadata. Then, the application invocations to retrieve or set information on the framework components use its own classes as parameters. Further, the framework uses the data from the received classes and the metadata to update the components state appropriately.

Metadata in attribute or access methods can be used to generate the graphical component to represent the respective property at the user interface. It also can be used to handle other issues, like how the information is converted between the application class and the component and how the component should behave due to user interaction. Class metadata are used for more general purposes like to define the layout for each component. Methods can also receive metadata to handle events generated by the user interaction with the graphical components.

This pattern differentiates from solutions that describe graphical interfaces in external XML files. This pattern proposes to use the application class instances as the base to create and interact with the graphical components. XML files can be used to describe metadata, however they should describe information about the classes and not about the interface itself.

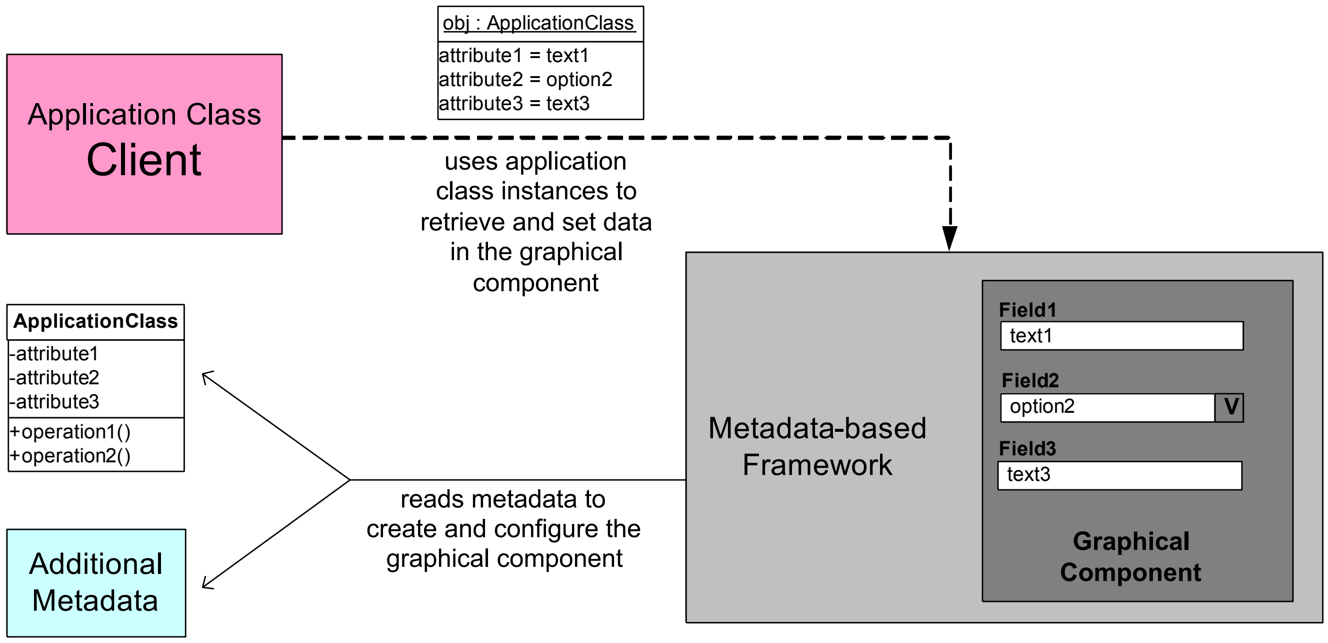

Fig. 14. Representation of Metadata-based Graphical Components..

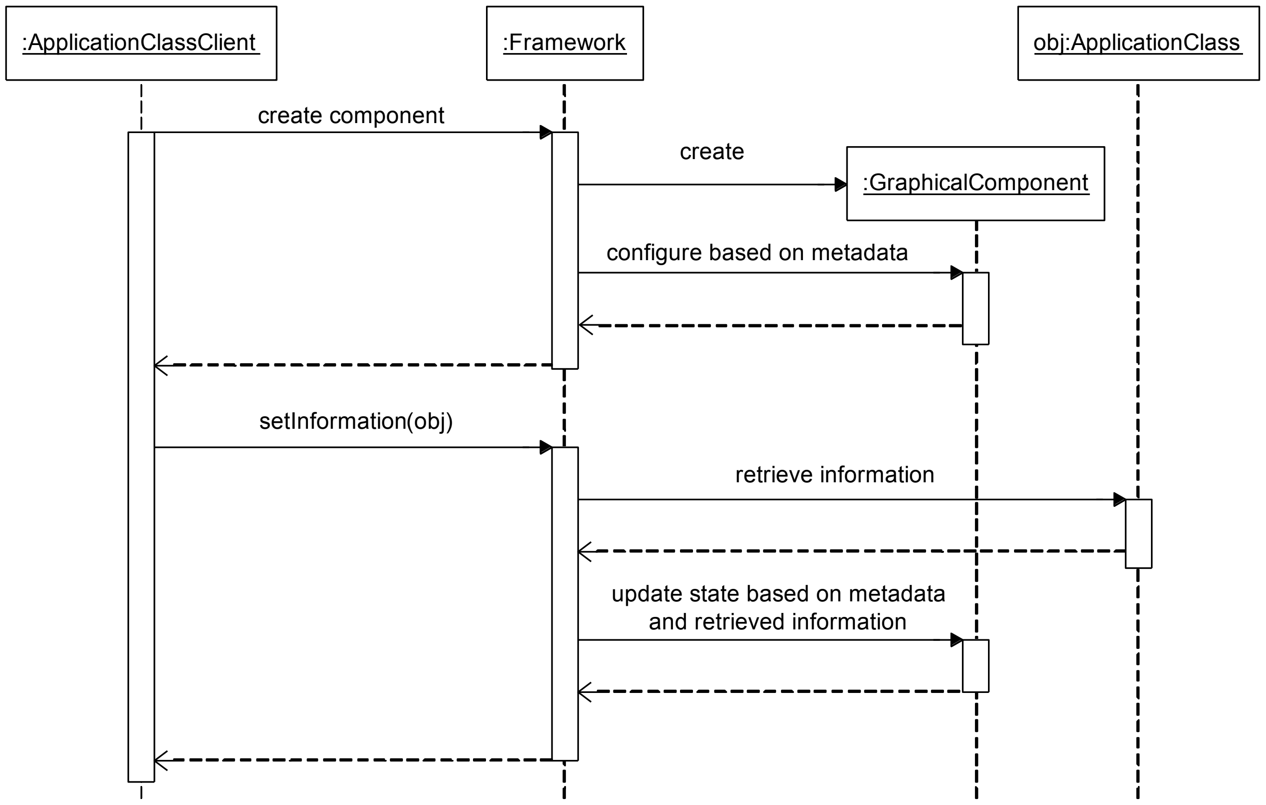

Fig. 15. Sequence of the interaction between the framework and graphical components.

Consequences

- ⊕ The same source code can be reused to generate screens that use different application classes, since the piece that is dependent on them can be based on class metadata.
- ⊕ The interaction of the application class with the graphical components can be performed internally by the framework, saving the creation of repetitive code.
- ⊕ Graphical components configuration details are encapsulated by the framework.
- ⊕ It is easier to standardize graphical interfaces if they use the same component to generate a piece of them or if they reuse the same class.
- ⊕ The graphical interface is composed by coarse grained components that represent entire application classes instead of fine grained ones that represent their attributes.
- ⊖ The pieces of the screen generated by the framework are restricted by the framework's generation rules.
- ⊖ Visual requirements sometimes cannot be fulfilled with an automated generation of a graphical component

Example

SwingBean (SwingBean 2008) is a framework that uses metadata defined in XML files to generate forms, tables and trees based on application class structure. This framework was developed by the present work's author before the beginning of this research and evolves during its progress. It became known by the brazilian software development community though two articles on a magazine (Guerra 2007; Guerra 2008) and it has already more than 4000 downloads by the time this work is written. Indeed, it has been already used successfully in more than one production applications. Thought the support of annotations was developed (Oliveira 2009), it is not yet available for download in production releases.

Figure 16 presents an example of **SwingBean** metadata descriptor to generate a form for the class Dog and Figure 17 displays the result. The `<line>` elements are used to define the organization of the class properties on the form and the `<property>` elements define the metadata for each one. The framework also uses default configurations based on property names and types to save some metadata definition. If the property is a `Date`, for instance, the `JCalendar` component is automatically used.

The form generation is performed by the class `JBeanPanel`, which uses application classes to set and retrieve information. The method `populateBean()` receives an application class as a parameter and populates it based on form information. Similarly, to retrieve information from an application class instance to be set on the form, the method `setBean()` should be used. Other methods are also provided for the application to interact with the component.

SwingBean also provides other components than forms, like tables, trees and tree-tables. It has an extension point that enables the addition of new components to represent properties, allowing them to receive new types of metadata defined in the XML file. To illustrate this, the following are examples of the framework's additional features: binding functionality to synchronize application classes with the components; validation of the information by each property's field in forms and tables; metadata-based filtering and ordering for information in tables; and general appearance customization.

```
<beanDescriptor>
   <line>
      <property name='name' size='20' uppercase='true' />
   </line>
   <line>
      <property name='race' label='Race'
                comboList='Yorkshire;Beagle;Basset;Poodle'
                colspan='3' />
      <property name='kennelNumber' />
      <property name='phone' mask='(##)####-####' colspan='2' />
   </line>
   <line>
      <property name='birthDate' format='dd/MM/yyyy' />
      <property name='hasPedigree' dlu='55' />
   </line>
</beanDescriptor>
```

Fig. 16. SwingBean XML metadata descriptor.

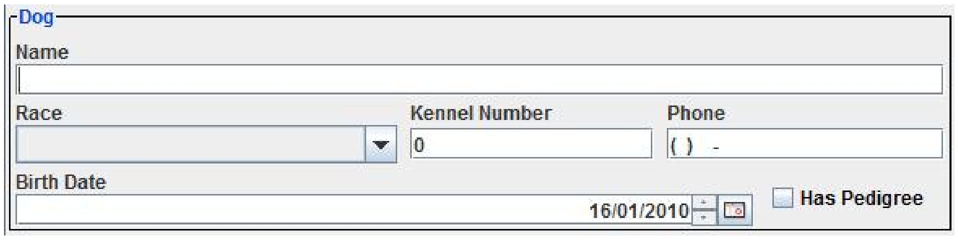

Fig. 17. Form generated using SwingBean.

Known Uses

JSP Bean (Farro et al. 2009) is a framework that generates HTML forms and tables based on a domain class and its annotations. The framework also provides features to populate the application classes based on request parameters sent by the generated form. The metadata is also used to generate client-side functions in Javascript to validate input formats.

ACE Framework (Costa and Figueredo 2009) uses metadata defined in annotations and XML files to map application services to be accessed by mobile devices in WML. The form to access the service is created based on the metadata present in the parameter classes. Then the framework provides an extension point that allows the creation of new generated graphical interface formats.

Spring Rich Client Project (SpringRCP 2010) is a framework whose goal is to provide a solution to create high-quality Swing applications quickly. Part of the metadata used in to generate a form for a class is configured programmatically using the class FormBuilder, in which each property should be added with its metadata. In addition, other information, like the label of each field, can be configured in property files to allow internationalization.

Metadata Definition

An important requirement for metadata definition in Metadata-based Graphical Components is that more than one component can be generated for the same application class. Because of this, sometimes is necessary to create more than one group of metadata for the same class. According to (Fernandes et al. 2010), annotations are not a good choice facing those requirements, unless they are combined with other alternatives.

When external metadata definition is used, it is possible to create distinct files that define two different groups of metadata for the same class. Besides, annotations can be used to complement it with more general information which is valid for all the graphical components, such as for validation and formatting. To configure the graphical components, metadata defined in other frameworks can be quite useful; therefore the Metadata Adapter can be a good pattern to be used in this scenario.

Related Patterns

In **Model View Controller** (Fowler 2002), a **Metadata-based Graphical Component** can be used in the View layer to make the interaction with the Model layer easier. **Transform View** (Fowler 2002) is a program that looks at domain-oriented data and converts it to HTML. In this context, the presented pattern can be considered a general **Transform View** that uses metadata to obtain knowledge about the domain. This pattern also complements **Template View** (Fowler 2002) or eliminates completely its necessity, since a piece of the screen that depends directly on the application class structure can be generated automatically by the framework.

The patterns **Property Renderer** and **Entity View** (Welicki et al. 2007) provide solutions to render the UI code respectively for instance properties and for an entire entity in adaptive object models. Since entity and property metadata is used to create the graphical component, these patterns can be considered specializations of **Metadata-based Graphical Component**.

This pattern can be considered a specialization of **Entity Mapping**, since the user interface can be considered a representation of the class objects. It is documented in a separate pattern because graphical components also have behavior and other particularities, and besides the forces and consequences for it are different. A **Configured Method Handler** can be used to handle events from the components and **Crosscutting Metadata Configuration** can be used to synchronize the application classes with components.

7. SUMMARY

This paper presented four architectural patterns which abstracted scenarios where the metadata usage are suitable for, also based on the analysis of preexisting frameworks and on the development of new ones. Each pattern presents a framework example whose development had the author's participation. These architectural patterns help software architects to visualize scenarios where metadata-based frameworks can be used to encapsulate functionalities and promote decoupling between software components from different layers. Framework engineers can also use the presented patterns in other to decide for a metadata-based approach in a framework.

ACKNOWLEDGMENTS

We would like to thank our shepherd Hugo Ferreira for all the valuable suggestions that aggregate value in these patterns during the course of this work. We also would like to acknowledge all the ones which participated on the writer's workshop given a precious feedback.

REFERENCES

Bauer, Christian; King, Gavin. Java persistence with hibernate. Greenwich: Manning Publ., 2006.

Beck, Kent. Implementation patterns. Upper Saddle River: Addison-Wesley Professional, 2007.

Buschmann, Frank et al. Pattern-oriented software architecture: a system of patterns. [S.l.]: Wiley, 1996. v.1

Carroll, Jeremy et al. Jena: implementing the semantic web recommendations. In: International World Wide Web Conference, 13., 2004, Proceedings... [S.l.: s.n], 2004. p. 74-83 .

CGLIB. Code Generation Library - CGLIB. Available at http://cglib.sourceforge.net/ accessed in 2010-01-31.

Chen, Nicholas. Convention over configuration. Local, 2006. Available at: <http://softwareengineering.vazexqi.com/files/pattern.html> Accessed on: 17 dec. 2009.

Costa, Bruno; Figueredo, Leandro. Uma arquitetura baseada em metadados para integração entre aplicações Web e plataformas móveis. 2009. 81 p. Trabalho de Curso (Engenharia de Software) - Curso de Especialização em Tecnologia da Informação, Instituto Tecnológico de Aeronáutica, São José dos Campos.

Davies, John; Fensel, Dieter; Harmelen, Frank. Towards the semantic web: ontology-driven knowledge management. S.l.: John Wiley & Sons, 2003.

Doernenburg, Erick. Domain Annotations. In: The Thoughtworks anthology: essays on software technology and innovation. Raleigh: Pragmatic Bookshelf, Mar.2008. Chapter 10, p. 121-141.

Farro, Carlos; Santos, Davyd; Firmo, Felipe; Jacomelli, Leandro. Um modelo para reuso de interface baseado em metadados de classes de domínio. 58 p. 2009.Trabalho de Curso (Engenharia de Software) - Curso de Especialização em Tecnologia da Informação, Instituto Tecnológico de Aeronáutica, São José dos Campos.

Fernandes, Clovis; Ribeiro, Douglas; Guerra, Eduardo; Nakao, Emil. XML, Annotations and Database: a Comparative Study of Metadata Definition Strategies for Frameworks. In: XML: Aplicações e Tecnologias Associadas, 2010, Vila do Conde, Portugal.

Fielding, Roy. Architectural styles and the design of network-based software architectures. 2000. Thesis (PhD in Information and Computer Science) - University of California, Irvine.

Filman, Robert; Friedman, Daniel. Aspect-oriented programming is quantication and obliviousness. In: Workshop On Advanced Separation Of Concerns At OOPSLA, Oct. 2000. Proceedings… Mountain View: RIACS, 2000. (RIACS Technical Report, 01.12).

Foote, Brian; Yoder, Joseph. Evolution, architecture, and metamorphosis. In: Pattern languages of program design 2. Boston: Addison-Wesley Longman, 1996. Chap. 13, p. 295-314.

Forman, Ira ; Forman, Nate. Java reflection in action. Greenwich: Manning Publ., 2005.

Fowler, Martin. Patterns of enterprise application architecture. Boston: Addison-Wesley Professional, 2002.

Gamma, Erich et al.. Design patterns: elements of reusable object-oriented software. Reading, MA: Addison-Wesley, 1994.

Genesis Framework. Available at <https://genesis.dev.java.net/> accessed in 2010-01-31.

Guerra, Eduardo ; Parente, José Maria ; Fernandes, Clovis. Mapeando Objetos para Entidades de uma Ontologia Utilizando Metadados. In: SIGE - Defense Operational Applications Symposium, 10., 2008, São José dos Campos, 2008. Anais... [S.l.: s.n], 2008.

Guerra, Eduardo ; Silva, Jefferson ; Silveira, Fábio ; Fernandes, Clovis. Using metadata in aspect-oriented frameworks. In: Workshop On Assessment Of Contemporary Modularization Techniques, 2., 2008. Nashville. Proceedings…[S.l.: s.n], 2008.

Guerra, Eduardo ; Souza, Jerffeson; Fernandes, Clovis. A pattern language for metadata-based frameworks. In: Conference On Pattern Languages Of Programs, 16., 2009, Chicago. Proceedings…[S.l.: s.n], 2009.

Guerra, Eduardo. Binding, componentes customizados e templates de telas no SwingBean 1.2 . Revista MundoJava, Curitiba, n.29, 2008.

Guerra, Eduardo. SwingBean : produtividade para interfaces desktop. Revista MundoJava, Curitiba, n. 21., 2007.

JavaBeans(TM) specification 1.01 Final release. 1997. Available at: <http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html> Accessed on: 27 dec. 2009.

Johnson, Ralph; Foote, Brian. Designing reusable classes. In: Journal Of Object-Oriented Programming, v.1, n. 2, p. 22-35, Jun./Jul. 1988.

JSR 175: a metadata facility for the java programming language. 2003. Available at: <http://www.jcp.org/en/jsr/detail?id=175>. Accessed on:17 dec. 2009.

JSR 220: Enterprise JavaBeans 3.0. 2006. Available at: <http://www.jcp.org/en/jsr/detail?id=220>. Accessed on: 17 dec. 2009.

JSR 222: Java Architecture for XML Binding (JAXB) 2.0. 2006. Available at: <http://jcp.org/en/jsr/detail?id=222>. Accessed on: 17 dec. 2009.

JSR 314: JAX-RS: The Java API for RESTful Web Services. 2009. Available at: <http://www.jcp.org/en/jsr/detail?id=314>. Accessed on: 10 jan. 2010.

Kiczales, Gregor et al. Aspect-oriented programming. In: European Conference On Object-Oriented Programming, 1997. Proceedings… [S.l:s.n.], 1997. p.220–242.
Massol, Vincent; Husted, Ted. JUnit in action. Greenwich: Manning Publ., 2003.
Miller, James. Common language infrastructure annotated standard. Boston: Addison-Wesley, 2003.
Nuccitelli, Renzo; Guerra, Eduardo; Fernandes, Clovis. Parsing XML Documents in Java Using Annotations. In: XML: Aplicações e Tecnologias Associadas, 2010, Vila do Conde, Portugal.
O'Brien, Larry. Design patterns 15 years later: an interview with Erich Gamma, Richard Helm and Ralph Johnson. Indianápolis: InformIT, Oct. 22, 2009. Available at: <http://www.informit.com/articles/article.aspx?p=1404056>. Accessed on: 26 dec. 2009.
Oliveira, Júlio César. Flexibilização do uso de metadados no framework swingbean. 2009. 51 f. Trabalho de Graduação (Ciência da Computação,) - Instituto Tecnológico de Aeronáutica, São José dos Campos.
Perillo, Roberto et al. Metadata modularization using domain annotations. In: Workshop On Assessment Of Contemporary Modularization Techniques (ACoM.09) at OOPSLA, 3., 2009 Orlando. Proceedings…[S.l:s.n.], 2009.
Ruby, Sam et al. Agile web development with Rails. 3. ed. [S.l.]: Pragmatic Bookshelf, 2009.
SAX Project. 2004. Available at:<http://www.saxproject.org/>. Accessed on: 03 jan. 2010.
Schwarz, Don. Peeking inside the box: attribute-oriented programming with Java 1.5. [S.n.t.], 2004. Available at: <http://missingmanuals.com/pub/a/onjava/2004/06/30/insidebox1.html>. Accessed on: 17. dec. 2009.
Spring RCP. The spring rich client project. Available at: <http://www.springsource.org/spring-rcp> Accessed on:06 jan. 2010.
SwingBean: aplicações Swing a Jato!. 2008. Available at:<http://swingbean.sourceforge.net/> Accessed on: 17 dec. 2009.
VRaptor: Alta Produtividade no Desenvolvimento Web em Java. Available at: <http://vraptor.caelum.com.br/>. Accessed on: 10 jan. 2010.
Wada, Marcos; Junior, Salomão. Estudo Comparativo de Ferramentas de Apoio ao Uso de Frameworks Baseados em Metadados, 2009. . Trabalho de Curso (Engenharia de Software) - Curso de Especialização em Tecnologia da Informação, Instituto Tecnológico de Aeronáutica, São José dos Campos.
Welicki, León; Yoder, Joseph; Wirfs-Brock, Rebecca. Rendering Patterns for Adaptive Object-Models, In: Conference On Pattern Languages Of Programs, 14., 2007, Monticello. Proceedings…[S.l.: s.n], 2007.