XML to Annotations Mapping Patterns

Milan Nosáľ and Jaroslav Porubän

Department of Computers and Informatics,
Faculty of Electrical Engineering and Informatics,
Technical University of Košice
Letná 9, 042 00, Košice, Slovakia
milan.nosal@tuke.sk,jaroslav.poruban@tuke.sk

Abstract

Configuration languages based on XML and source code annotations are very popular in the industry. There are situations in which there are reasons to move configuration languages from one format to the other, or to support multiple configuration languages. In such cases mappings between languages based on these formats have to be defined. Mapping can be used to support multiple configuration languages or to seamlessly move configurations from annotations to XML or vice versa. In this paper, we present XML to annotations mapping patterns that can be used to map languages from one format to the other.

1998 ACM Subject Classification D.3.4 Processors – Parsing

Keywords and phrases Mapping Patterns, Language Design, Annotations, Attribute-oriented Programming, XML

Digital Object Identifier 10.4230/OASIcs.SLATE.2013.97

1 Introduction

Our paper concerns software system configuration metadata formats. We will use the term *software system metadata* for the total sum (everything) of what one can say about any program element, in a machine or human understandable representation. This definition is in compliance with the understanding of software system metadata in Guerra et al. [6] or Schult et al. [12].

By the association model (Duval et al. [3]) there are two types of metadata.

- **Embedded** (or internal) **metadata** are metadata that share the source file with the target data. Embedded metadata use in-place binding, they are associated with the target data by their position.
- **External metadata** are metadata that are stored in different source files than the target data. They use navigational binding where metadata include references to the target data.

Attribute-oriented programming (@OP) is a program level marking technique. This definition shared by many works in the field [8, 11] is a basis for classifying @OP as a form of embedded metadata. An annotation is a concrete mark annotating (marking) a program element.

XML on the other hand is a classic form of external metadata [4, 15, 9]. XML allows structuring metadata and storing them externally to the source code. From the point of view of the language theory, XML is a generic language that can be used to host concrete domain-specific languages [1].

These two are both widely used metadata formats used as notations for configuration languages in professional frameworks. Java EE uses both formats in many technologies

2nd Symposium on Languages, Applications and Technologies (SLATE'13).
Editors: José Paulo Leal, Ricardo Rocha, Alberto Simões; pp. 97–113

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

such as Java Persistence API (JPA) or Enterprise Java Beans (EJB). The .NET framework extensively uses both XML formats (e.g., in the Enterprise Library) and .NET attributes (.NET attributes can be used for annotating, for example as in MyBatis.NET or Windows Communication Foundation (WCF)).

Most of the frameworks started with supporting XML as a notation for a configuration language, but after the introduction of annotations extended the configuration apparatus to support the annotations as well. Designing a good new notation for a configuration language with the same expression power in terms of supported configuration can be difficult. Mapping patterns, which can be used as a basis for designing (and thus for implementing as well) a new notation, may significantly reduce this effort.

2 Annotations and XML

In this paper, we want to present mapping patterns between this two formats. But why would anyone want to map a language in one format to the other? There are characteristics of these formats that make them complement each other. In some situations, annotations are better; in other the XML documents are advantageous.

Fernandes et al. [4] present a case study that compares three forms of configuration metadata – annotations, databases and XML. They compare these formats according to three criteria: the ability to change metadata during runtime of a system; the ability to use multiple configurations for the same program elements; and support for the definition of custom metadata. Tilevich et al. [15] compares annotations and XML in few aspects such as programmability, reusability, maintainability and understandability.

Annotations' association model and their native support in a language is the reason why Tansey et al. [14] talk about annotations as a tool for more robust and less verbose software system metadata. The XML navigational binding is more fragile during refactoring and evolution of the program than in-place binding [13, 15]. Annotations' compactness and simplicity is a consequence of native support in a language, that lowers the redundancy of structural information [10]. Since the annotations are a part of a host language, changes in annotations need recompilation. If runtime changes of configurations are a requirement, external metadata are a solution [6, 9].

The fact that annotations are scattered in the code puts a programmer into a situation when he/she needs to search whole source code to understand configuration [15, 9]. On the other hand, when examining only one program component a programmer can see the component code and configuration in one place [15].

All these arguments show that usages of both annotations and XML have their sense and meaning. Therefore, we think that there are situations when one language format is no longer the better and there is a need to port the configuration language to the other. If it is expected that there will be situations in which annotations are more adequate and also situations in which XML is better, then it is useful to support two notations for a configuration language, one annotation-based and one XML-based. This we consider the main motivation for using our mapping patterns. An interesting related work that deals with finding mapping between XML and other data format is a comprehensive discussion of XML to objects mapping by Lämmel and Meijer [7].

2.1 XML to @OP Mapping Patterns

We were dealing with the abstraction of multiple configuration sources in our previous work [9]. We have designed and implemented a tool called *Bridge To Equalia* (BTE) that facilitates unified access to @OP-based and XML-based notations for one configuration language by combining the sources into a complete configuration. One of the problems we had to deal with was finding a default mapping that would be common in existing frameworks. In this paper, we present XML to annotations mapping patterns that we identified while working on experiments with BTE.

XML to @OP mapping patterns are patterns that specify ways how an XML document or parts of it can be mapped to its counterpart in annotations (and vice versa). The following catalogue contains patterns that we have identified in experiments we have performed so far. The patterns should be found useful in the following scenarios:

- *Rewriting an existing system from one configuration format to another* – when a system author decides to change configuration notation from XML to annotation or vice versa, the patterns can help him/her to do better decisions in designing a new configuration language.
- *Adding a new configuration notation to a system* – in this situation a system supports XML or annotations but not both. A framework author wants to support a new configuration notation in order to gain benefits of supporting multiple configuration formats.
- *Building a new framework supporting multiple configuration notations* – mapping patterns can be used even when the two configuration notations are designed simultaneously from scratch.
- *Designing a mapping apparatus for a configuration abstraction tool* – in an abstraction tool there has to be way to define a mapping between supported notations; there has to be a mapping language. In this context, mapping patterns are perfect candidates for the concepts of such a language.

These situations can be abstracted and summarised by Figure 1. In short, there is a need to define mappings between annotations and XML to define configurations both with XML and annotations. If the case is that configuration is moving from one format to another, then mappings are needed to reuse domain knowledge from the old configuration language.

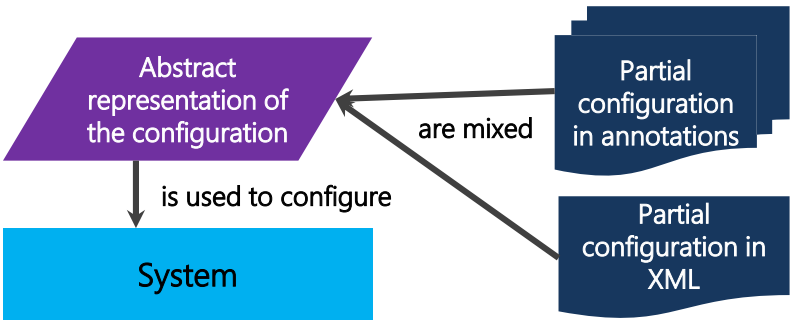

Figure 1 Supporting XML and annotation-based configuration.

3 Pattern Catalogue

In the next sections we introduce a catalogue of XML to annotations mapping patterns. We have divided these patterns into two groups according to their roots.

The following describes the pattern's description elements (inspired by [5]):

- The *Motivation* presents a problem context in which the pattern's solution is suitable.
- The *Problem* states the question to which a pattern provides an answer.
- *Forces* lists conflicting forces that the pattern should help balance.
- The *Solution* describes the mapping pattern.
- *Consequences* lists the possible positive and negative consequences in a pattern.
- *Known Uses* lists known uses of the described pattern.
- The *Example* illustrates the pattern usage.
- *Related Patterns* describes how the pattern interacts with the other ones from this catalog.

3.1 Structural Mapping Patterns

The first group of the patterns are simple structural patterns. They aim to show the fundamental mappings between XML and annotations. They do not deal with the relationship of a configuration to the program structure, merely with the structural mappings between the languages.

An example when these patterns can be sufficient may be global configurations that are not configurations of some program element but of a system as a whole. In case of configurations through annotations this means that the binding of configuration annotations to program elements does not have to be significant. Probably the best case would be if they could just annotate a system as a whole. In .NET framework there is an option to annotate assemblies. However, currently the Java annotations do not support annotating a whole system. Package annotations are usually used for this purpose. Another solution may be annotating an arbitrary class or a class that is somehow significant for configuration. Interesting example of such usage are configuration classes in the Spring framework.

3.1.1 Direct Mapping Pattern

Motivation. The first and basic problem when mapping a language from XML to annotation (or vice versa) is the question of how to represent language constructs from one language in another. E.g., if there is an annotation-based configuration language and the new language is supposed to be built on XML, there has to be a simple and direct way to match constructs from the first language to the second to have a starting point.

Problem. What is the simplest way to map annotations' constructs to XML and vice versa?

Forces.

- An annotation-based and an XML-based language need to be able to represent the same configuration information.
- Both languages do not have to conform to any special rules.

Solution. The most common and straightforward way is to use the Direct Mapping pattern. By default, the Direct Mapping pattern proposes to map annotation types to XML elements with the same name. Annotation type parameters are mapped to elements with the same name, too. This simple mapping is usually sufficient. From the point of view of XML an XML element is by default mapped to an annotation that has its simple name identical to

the corresponding XML element's name. XML attributes are mapped to annotation type parameters of the same name.

This pattern can be parameterized with the name mapping and XML attribute/element mapping choice. Naming parameterization allows different names (in this context we can speak of keywords) for mapped constructs in both languages. It allows keeping naming conventions in both formats, identifiers starting with uppercase for Java annotation types and identifiers starting with lowercase in XML. For some language constructs in annotation-based languages it might be interesting to use XML attributes instead of elements. This concerns only marker annotations (see [16]) and annotation members that have as return type a primitive, string or a class (if its canonical name is sufficient in XML). For example, arrays are excluded since XML attributes are of simple type and there cannot be more than one attribute with the same name.

Consequences.

[+] Simple XML structures can be mapped to annotations and vice versa.
[+] Naming parameterization allows different names in annotations and in XML.
[+] In some cases element/attribute choice parameterization allows more convenient notation in XML, because attributes are less verbose than elements.
[-] More complex mappings cannot be realised merely using this pattern.

Known Uses.

- JAX-WS's mapping of the `serviceName` parameter of the `@WebService` annotation to `wsdl:service` element of WSDL language.
- JSF's mapping of the `@ManagedBean` annotation mapped to `managed-bean` element.
- JPA maps the `@Table` annotation to the `table` XML element and its `name` parameter to the XML attribute `name`.

Example. Figure 2 shows an example of the Direct Mapping pattern. A simple sentence states that there is a book *Heaven Has No Favorites* in a library. The `book` element is mapped to the `@Book` annotation, its attributes (`in-library`) and child elements (`title` and `author`) are mapped to corresponding parameters of the `@Book` annotation. The naming parameterization can be seen in mapping the `@Book` annotation to the `book` element (blue colour), where the name "book" has been capitalized in the annotation name. The attribute parameterization is present in mapping between `in-library` attribute and `inLibrary` annotation parameter (green colour).

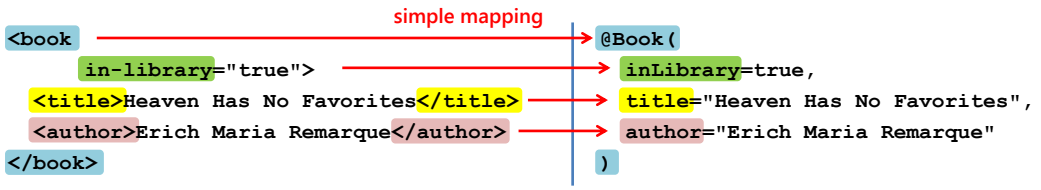

Figure 2 An example of the Direct Mapping pattern.

Related Patterns. The Direct Mapping pattern proposes only mappings of keywords one to one. Usually, it is combined at least with the Nested Annotations pattern to define the simplest mappings between annotations and XML.

3.1.2 Nested Annotations Pattern

Motivation. Another structural problem of mapping is handling the XML tree structure in annotations. A tree structure of the XML is usually used to model some language property and therefore it is significant.

Problem. How to preserve XML tree structure in annotation-based languages?

Forces.

- XML allows to structure configuration information to trees of element nodes and their attributes.
- Meaning of the tree structure is significant and therefore it has to be preserved in some form in annotations.

Solution. The Nested Annotations pattern proposes to nest annotations in order to model a tree structure in annotations. The root of the tree in XML is modelled by an annotation type and its direct descendants (in XPath the children axis of the XML element) are modelled by the parameters of the annotation type. If one of the descendants has children itself, its type will be an annotation type too. This annotation type defines the children by its parameters.

Consequences.

[+] XML tree structures can be mapped to annotations.

[-] Currently annotations' implementations do not support cyclic nesting, so if the XML language has an element that can have itself as a descendant, this pattern cannot be used. XML allows modelling arbitrary trees, e.g., there can be a `node` XML element with child elements of the same type. The same is not allowed in annotations (at least not in Java or .NET attributes).

[-] The sequence of annotation members is not preserved during the compilation; therefore if the order is significant, this approach will fall short. The only way of preserving the order of the elements in annotations is usage of an array as an annotation parameter type.

[-] In some programming languages, that do not support annotation nesting at all (e.g. .NET), the pattern is not applicable.

[-] All nested annotations inherit the target program element from the root annotation. Therefore, the modelled XML tree has to apply to the same target program element.

Known Uses.

- In JPA there is a `@Table` annotation with the parameter `uniqueConstraints` that is of the type array of `@Unique` annotations. The `@Table` is mapped to `table` element and `uniqueConstraints` are mapped to the `unique-constraint` element with its own children.
- EJB and its `@MessageDriven` annotation with the parameter `activationConfig` that is of type array of `@ActivationConfigProperty` annotations. This models a branch from XML, where the `@MessageDriven` annotation is mapped to the `message-driven` element and the parameter `activationConfig` to the `activation-config-property` element.
- Java Servlets technology uses nested annotations pattern in the `@WebFilter` annotation. Its parameter `initParams` is an array of `@WebInitParam` annotations. The `@WebFilter` is mapped to the `filter` XML element and the configuration branch represented by the `initParams` annotation parameter is mapped to an XML subtree `init-param` (or in fact multiple subtrees, since the `initParams` parameter is an array).

Example. Figure 3 shows an example of the Nested Annotations pattern. The library language from the example from the Direct Mapping pattern has been slightly changed. For simplification we have omitted the flag that specifies whether the book is in the library and decided to split the name of the author into first name and surname. In XML this is easy to do using child elements (or possibly attributes). This splitting created a new subtree (branch). In annotations it is mapped using the Nested Annotations pattern to `@Book`'s parameter `author` (yellow colour), that holds another annotation called `@Author`. `@Author` annotation deals with structuring the author's name using its own parameters.

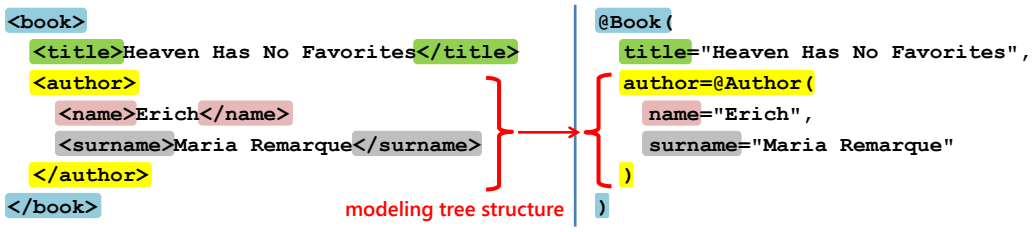

Figure 3 An example of the Nested Annotations pattern.

Related Patterns. A closely related pattern is the Parent pattern that can be used as an alternative for mapping XML tree structures to annotations. The Parent pattern provides more expressiveness than the Nested Annotations pattern, as we argue in its description.

3.1.3 Enumeration Pattern

Motivation. Sometimes there is a piece of configuration information represented by a set of mutually exclusive marker annotations. Designing an XML language with "marker" XML elements might seem a little too verbose and it increases the complexity of XML instance documents.

Problem. How to elegantly map a set of mutually exclusive marker annotations to XML?

Forces.

- There is a configuration property that is in annotation represented by a set of mutually exclusive marker annotations.
- The Direct Mapping pattern makes the XML language too complex.

Solution. The Enumeration pattern proposes to map a set of mutually exclusive marker annotations to XML element values. Each of the marker annotations is mapped to an enumeration value.

This pattern can be extended for those that are not mutually exclusive merely by allowing more occurrences of the enumeration element. Modifications may include allowing one annotation with parameter (such as in case of JSF in the Known Uses paragraph) that is mapped to XML element's values that are not a part of the enumeration.

Consequences.

[+] XML language can be more compact and comprehensible.
[+] At the same time marker annotations are more compact regarding readability.
[−] Such an indirect mapping may make mixing configurations from XML and annotations more complex.

Known Uses.

- JSF technology uses this pattern to specify the scope of their managed beans. The annotations `@ApplicationScoped`, `@RequestScoped`, `@SessionScoped` and other scoping annotations from the `javax.faces.bean` package are mapped to single XML element – the `managed-bean-scope` element. It is modified pattern, because the `@CustomScoped` annotation has a parameter that is mapped to the `managed-bean-scope` value that differs from the enumeration values. While the standard cases are handled by marker annotations, specific cases are handled by using custom values.
- EJB use `@Stateless`, `@Stateful` and `@Singleton` annotations to configure session beans. These three annotations are mapped to the value of the `session-type` XML element in the deployment descriptor.

Example. Figure 4 presents a simple example of the Enumeration pattern. The example is based on the JSF managed beans technology. Annotations are mapped to the value of the `managed-bean-scope` XML element and not to the element itself.

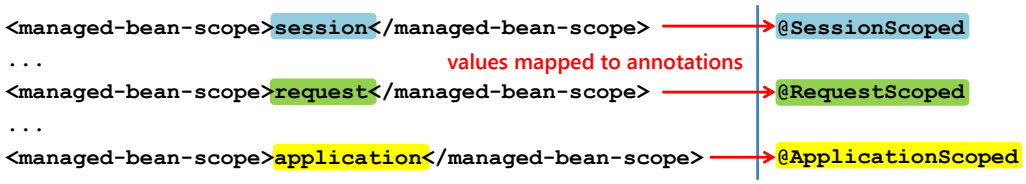

Figure 4 An example of the Enumeration pattern.

Related Patterns. Enumeration pattern is an alternative to the Direct Mapping pattern, but the Enumeration pattern is applicable only in special cases. The Enumeration pattern can improve code and configuration readability in a situation when in an annotation there is a set of mutually exclusive marker annotations, or vice versa, in XML there is an XML element that has values of an enumeration type.

3.1.4 Wrapper Pattern

Motivation. In some situations there may be an implicit grouping property in annotations that has to be mapped to XML. An example may be a grouping of some pieces of configuration information according to their bound target elements. In annotations, this structuring is implicit according to their usage, for example they annotate the members of the same class.

Problem. How to represent grouping in XML that is based on some implicit property of annotations?

Forces.

- Annotations use some implicit grouping property that does not have appropriate language construct that could be mapped to XML.
- Grouping is closed on branches and depth – grouping is defined strictly on one branch in a tree and it groups merely constructs on the same level in the tree.

Solution. The Wrapper pattern proposes to map implicit groupings from annotations to so called wrapper XML elements. Wrapper XML element is an element, that groups together elements according to some property. A wrapper XML element does not have its counterpart in annotations, at least not an explicit language construct like an annotation or an annotation parameter. The Wrapper pattern is usually just a notation enhancement, for

example, binding to the members of the same class can be recovered from target program element specifications. The Wrapper pattern simply structurally enhances the notation in XML, for example, for design reasons.

Consequences.

[+] XML can model some implicit properties of the annotations or the target program.
[+] The Wrapper pattern may increase readability of the configuration in the XML language.
[+] @OP Wrapper pattern is used to overcome the problem of annotating the program element with more annotations of the same type (so called Vectorial Annotation idiom [5]).
[-] Since the Wrapper pattern proposes a use of a language construct in one language that does not have a corresponding counterpart in the other, this may increase complexity of mixing the configuration.

Known Uses.

- JPA uses the `attributes` XML element to group column definitions in the `entity` element. The `attributes` element does not have an @OP equivalent in the code.
- JPA also uses vectorial annotation `@NamedQueries` that is a Wrapper for an array of `@NamedQuery` annotations. In XML, the `@NamedQueries` does not have an equivalent, the `named-query` XML elements (that are equivalents to `@NamedQuery`) are directly situated in the root `entity-mappings` element.
- The `@MessageDriven` annotation from EJB has a parameter that is of type array of `@ActivationConfigProperty`. While using the nested annotations pattern this would be mapped to a set of `activation-config-property` XML elements that would be direct descendants of the `message-driven` XML element, EJB uses the Wrapper pattern to wrap the `activation-config-property` elements into the `activation-config` element.

Example. Figure 5 presents an instance of the Wrapper pattern. The binding to program elements is done with the Target pattern combined with the Parent pattern (introduced in Section 3.2.2), mapping of the elements is highlighted with colours (to keep it simple the binding to program elements is not considered). The important thing here is the `attributes` XML element that has no direct equivalent in annotation-based language and is only used to wrap `column` elements.

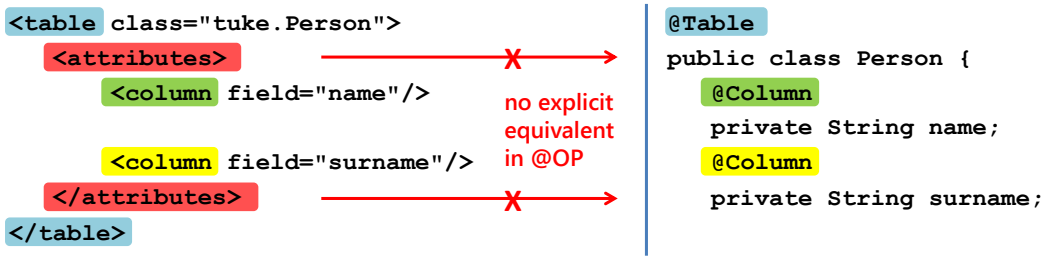

Figure 5 An example of the Wrapper pattern.

Related Patterns. The Parent and the Nested Annotations patterns are related to this pattern. The Parent and the Nested Annotations are used to group and structure configuration information too, but they have direct and explicit representations in both languages. They carry significant configuration information. The Wrapper pattern is more of a design decision (e.g., to make the language more readable).

3.1.5 Distribution Pattern

Motivation. Sometimes the same configuration information is supposed to be distributed in one configuration language differently than in the other. There may be some configuration information that in XML is due to design reasons separated from its logical tree structure, but it still has to be somehow associated together as a logic unit. An example may be dealing with so called Fat Annotation annotations' bad smell (Correia et al. in [2]). While an XML element with many children elements might be good, overparameterized annotation might increase code complexity and reduce code readability.

Problem. How to handle different distribution of configuration information in XML and annotations?

Forces.

- Due to design decisions one or more constructs in the first language are mapped to one or more constructs in the second language, while the mapping is not straightforward.
- Distributed constructs need to be tied together to form a logical unit.
- Logical units in both languages need to be unambiguously mapped to their counterparts.

Solution. The Distribution pattern proposes to map distributed constructs by sharing a unique identifier. This identifier is unique in the configuration. The complete model for the corresponding configuration information is built up from all the constructs with the same identifier. This unique identifier may even be a target program element, as in case of mapping one XML element to more simple annotations. The unique identifier is shared between both languages to serve for mapping between logical units.

Consequences.

[+] More complicated tree structures of configuration languages are possible that do not have to strictly follow each other, and therefore there is more space to adapt the languages to good design.

[-] Such differences in the languages may increase the complexity and readability of the mapping itself and therefore of the configuration as well.

Known Uses.

- Java Servlets technology is a nice example (since it is quite simple, we used it in the example section). The `@WebServlet` annotation annotates a servlet implementation and through parameters specifies its name and its URL mappings. XML configuration distributes these pieces of information to the `servlet` XML element, that specifies the servlet and binds it to its implementation, and to possibly multiple `servlet-mapping` XML elements (one for each URL mapping in annotations). The `servlet-mapping`, the `servlet` elements are tied together by the servlet's name, which is servlet's identifier.
- Java Servlets use the same approach with the `@WebFilter` annotations.

Example. Figure 6 shows an example of the Distribution pattern. In this example the `PersonServ` servlet configuration information is represented by one construct in the annotations – the `@WebServlet` annotation. In the XML language the same annotation is distributed to two XML elements – the `servlet` and the `servlet-mapping` elements. These elements are bound through the servlet name in `servlet-name` elements – "PersonServlet" (highlighted in red). In the example there is again used the Target pattern (yellow colour), that is explained in Section 3.2.1.

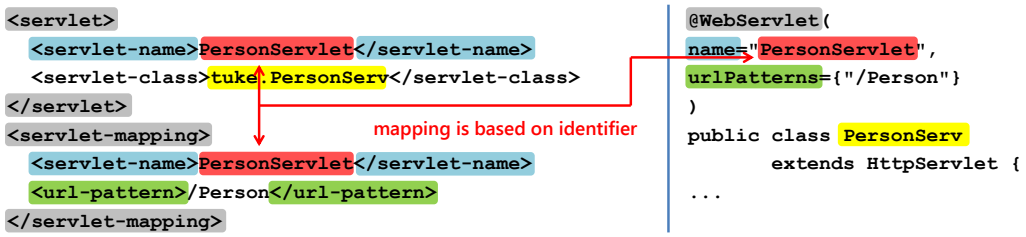

Figure 6 An example of the Distribution pattern.

Related Patterns. The Target pattern binds the XML constructs to target program elements. The Distribution pattern can be used to indirectly bind scattered XML elements to program elements. The Target pattern can be used to bind one of the distributed constructs to target element, then the Distribution pattern that ties distributed constructs together propagates this binding to the rest of them.

3.2 Program Elements Binding Patterns

The structural patterns are fundamental for mapping between annotations and XML. However, in practice they are combined with program elements binding patterns. That is because most of the configuration usually directly concerns program elements. Annotations deal with this easily, because annotations by definition annotate program elements. They are a form of embedded metadata and they bind themselves to program elements using in-place binding. Annotations are written before (annotate) program elements. Metadata of an annotated program element are enriched by the metadata represented by its annotation. This of course raises a question of how to take into account this binding to program structure in an XML-based language. This group of patterns deals with this issue and shows how XML structures can be bound to their target program elements.

3.2.1 Target Pattern

Motivation. @OP is a form of embedded metadata. Annotations are always bound to program elements. The same way the configuration author wants the XML elements to be bound to program elements.

Problem. How to bind XML elements and/or attributes to program elements?

Forces.

- XML elements/attributes have to be bound to the same program elements as their corresponding annotation constructs.
- XML elements/attributes are a form of external metadata so they do not provide in-place binding of embedded metadata.

Solution. The Target pattern proposes to use a special dedicated XML element or attribute to set a target program element for an XML element. An attribute is preferred to make the notation more compact. The name of the dedicated element/attribute has to be unique in a given context for the node to be distinguishable from other nodes used to carry configuration information. By default the generic name is used, like "class" for binding elements to class definitions. By means of parameterization a special name can be used, that may suit better for the language. Another reason may be preventing name clashes if the language already

defines a node with the same name that is used to represent configuration information. To make the notation shorter, the target program element is inherited in XML branch, the same way as XML namespaces are. Thus, by default a whole branch is bound to the same target program element.

A more structured approach can be used as well. The reference can be structured to more XML elements/attributes. For example, a canonical name of the class member can be split to the class name and to the simple name of the member. This can be useful when there is a need to use both of these pieces of configuration information. Then the canonical name does not have to be parsed every time configuration is read.

Consequences.

[+] XML elements and attributes can be bound to target program elements.

[-] Navigational binding using canonical name of the target program element is error-prone due to the absence of code refactoring that would take into account the composition of languages. If the programmer changes the name of a method, thanks to in-place binding, an annotation will be still valid while XML will need manual refactoring (or implementing a tool that would be able to automate it).

Known Uses.

- JSF technology uses the Target pattern for example in the `managed-bean` element to bind it to the program element, to which its counterpart – the `@ManagedBean` annotation, is bound. The `managed-bean` XML element has a child of name `managed-bean-class` that states the target program element canonical name.
- Servlets have the XML `servlet` element with a child `servlet-class` element that binds the servlet to its implementation.
- The JPA `entity` XML element is bound to its class using the `class` attribute.
- EJB uses the `resource-ref` XML element as an equivalent to the `@Resource` annotation. The `resource-ref` element is bound to its target program element by `injection-target` element, that has two descendants, `injection-target-class` specifying the target class, and `injection-target-name` identifies the actual field to be injected.

Example. Figure 7 shows an example of the Target pattern. The `table` XML element is navigationally bound to class `tuke.Person` exactly as its corresponding annotation `@Table`. The binding uses the canonical name of the class (highlighted in yellow), the pattern uses the generic name for binding the XML attribute (highlighted in green).

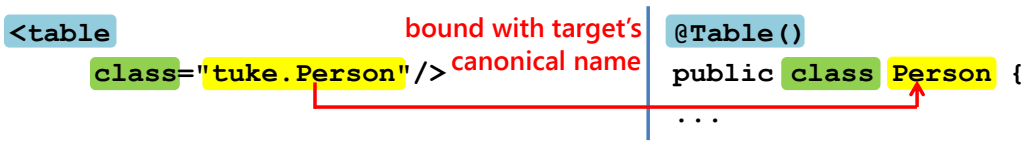

Figure 7 An example of the Target pattern.

Related Patterns. The Target pattern is related to the Distribution pattern. The Target pattern proposes a way to define bindings between XML and program elements by using a program element's identifier. The Distribution pattern proposes a way to bind distributed language structures together using a custom identifier. The Distribution pattern can be used to indirectly bind distributed XML constructs to program elements by binding constructs without explicit target program elements to a construct that is already bound to a target program element.

3.2.2 Parent Pattern

Motivation. Sometimes the Nested Annotations pattern is not suitable for modelling the XML tree. In the ORM the columns of the table belong to the table. In XML this is modelled by putting the column element into the table element. Using the Nested Annotations pattern this would be modelled by a single annotation `@Table` with a member which would have the type of array of `@Column` annotations. But these annotations have a different target program element type than the @Table that annotates the class. They enhance metadata of the class members. According to the Target element pattern, in the Nested Annotations pattern, the parameters of annotations are bound to the same target program element as the annotations themselves.

Problem. How to model XML tree structure in annotations when the descendants of some element belong to a different target program element then their parent does?

Forces.

- XML allows to group elements by their meaning and to create trees of element nodes.
- Meaning of the structure is significant and therefore it has to be preserved in some form in annotations.
- Some of the XML elements or attributes in the tree belong to different target program element than the root.

Solution. The Parent pattern proposes to define parent-child relationships between annotation types. This relationship defines two roles, parent and its children. Parent-child relationships can be used to define logical tree structures consisting of annotations. Metadata carried by annotations in the children role are considered to be on the same level as the parent's members.

The matching of child annotations with their parents has to be unambiguous. Usually, a matching based on program structure axes is sufficient. For example, in the Java programming language a program structure is built from packages, classes and class members. Commonly used axis is the descendant axis, where the children annotations annotate descendants of the program element annotated by the parent annotation. A concrete example of using this axis can be found in the example section of this pattern. Another example can be the self axis. In this case all children annotations annotate the same program element as their parent. In the XML tree the new target program element of some of the descendants can be specified both using absolute name – full canonical name of the program element; or using relative name, specifying merely the identifier relative to the current target element context[1].

This pattern allows overriding the inherited target program element. In XML that is easy, just by explicitly specifying a new target element. However, annotations have to use this logical relationship to preserve the desired structure and to still properly annotate different target program elements.

[1] E.g., if the context is `tuke.Person` and a subtree has as target program element the `surname` field of the `tuke.Person` class (the `tuke.Person.surname` field), its relative name in this context is merely `surname`.

Consequences.

[+] Tree structures of XML can be mapped to annotations.

[+] Different target program elements of the configuration information in the tree are preserved in the annotation's concrete syntax.

[+] In programming languages that do not support annotation nesting this pattern can substitute the nested annotations pattern (using the self axis).

[-] Using unnatural or complicated matching algorithms can make annotation languages difficult to understand and use. We believe using the direct descendant axis is easiest to understand.

Known Uses.

- JPA provides a classical example of the Parent pattern on the direct descendant program axis. The `@Entity` annotation annotates a class. The annotations `@Id`, `@Basic` and `@Column` annotate members members of the same class. In XML, their equivalents are logically structured as descendants of the `entity` element, the equivalent of the `@Entity` annotation.
- JSF uses the Parent pattern on the self axis with annotations `@ManagedBean` and scope annotations, such as `@RequestScoped` or `@SessionScoped`. Scope annotations are logical descendants of the `@ManagedBean` annotation.

Example. Figure 8 shows an example of the Parent pattern. The `@Column` annotation is on the descendant program axis of the `@Table`'s target program element. Class members are descendants of the class itself in the program structure in Java. The `@Table` is in this example a parent of the `@Column` annotation. This relationship models the structure in XML, where the `table` element is the parent of the `column` element (while the `column` have different target program elements than the `table`). The `table`'s simple descendants are mapped to `@Table`'s members using the Direct Mapping pattern.

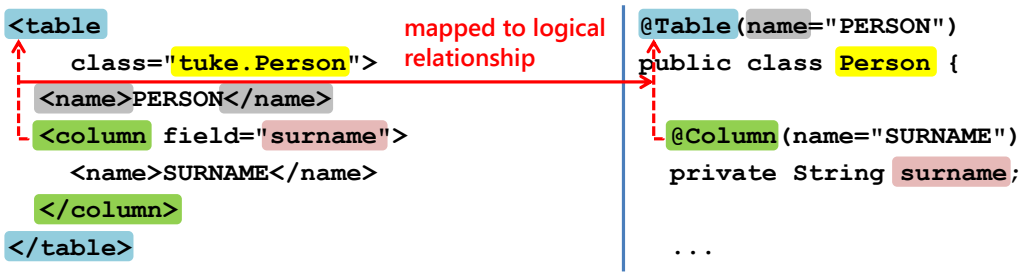

Figure 8 An example of the Parent pattern (using relative naming).

Related Patterns. Related pattern is the Nested Annotations parent. Both of these patterns are used to model the tree structure of XML languages, but the Parent pattern is more flexible.

3.2.3 Mixing Point Pattern

Motivation. If the system supports both XML and annotations, there has to be the mechanism to recognize whether some configuration information is in both annotations and XML or merely in one of the formats. The common situation may be using an annotation-based configuration language to define a default configuration and an XML-based language to override the default configuration. In such a situation processing only one configuration

format based on users choice is not sufficient, finer granularity in mixing is required than merely on the root construct level. The languages should not only be substitutable but rather be able to complement each other.

Problem. How to provide finer granularity for duplicity checking in XML and annotations configuration languages?

Forces.

- The annotation-based and XML-based configuration languages can both be used to define the configuration.
- Some pieces of the configuration information are duplicated in either XML and annotations.
- Both XML and annotations configuration are not complete, a complete configuration is a combination of both.

Solution. The Mixing Point pattern proposes to define so called mixing points in a tree structure of the languages. A node that is a mixing point has to be unambiguously identified, e.g., by its name and target program element (or another policy may be used, like merely by its name). These two properties are easily represented in both formats using the Direct Mapping pattern and the Target pattern. When a node in a tree is a mixing point, the checks for duplication on its level must be performed in both annotation and XML-based trees and the configuration information is mixed from both sources. If a node is not a mixing point, no mixing is performed and simply the configuration information is taken from the language with higher priority. The Mixing Point pattern is parameterized by the priority parameter that specifies which format has higher priority and thus overrides the duplicated configuration information. A root node is always by default a mixing point.

Consequences.

`[+]` A finer granularity of combining two partial configurations in XML and annotations.

`[-]` Sometimes even this granularity might be too coarse-grained. The same name and the target element can be found in array annotation members. So if there is a need to combine two arrays without duplicates, some more adequate duplication detection mechanisms must be used (for example based on values).

Known Uses.

- The Spring framework currently supports mixing of both annotations and XML configuration, an example may be using both approaches to configure dependency injection, where the XML configuration can be complemented by annotations (`@Autowired` can be considered one of the mixing points in this example). XML has the higher priority in the Spring framework.
- Hibernate (ORM framework) supports mixing too; it allows specifying mappings in both annotations and XML. However, as mixing points there are classes and not fields, it is possible to configure one class through annotations and one through XML, but it is not possible to configure one class by mixing both formats. The annotations have higher priority in Hibernate.

Example. Figure 9 shows an example of the Mixing Point pattern. The configuration information specifing column for the field `tuke.Person.name` is duplicated. The duplication is detected using the naming parameter of the Direct Mapping pattern (`@Column` to `column` XML element – blue colour) and using the target element identifier (green colour). If there is a configuration information that is not duplicated, this information is directly used in whole configuration (configuration information for `tuke.Person.id` and `tuke.Person.surname`).

Otherwise the information from the language with higher priority is used, in our example it may be XML overriding the column value "NAME" to "FIRSTNAME".

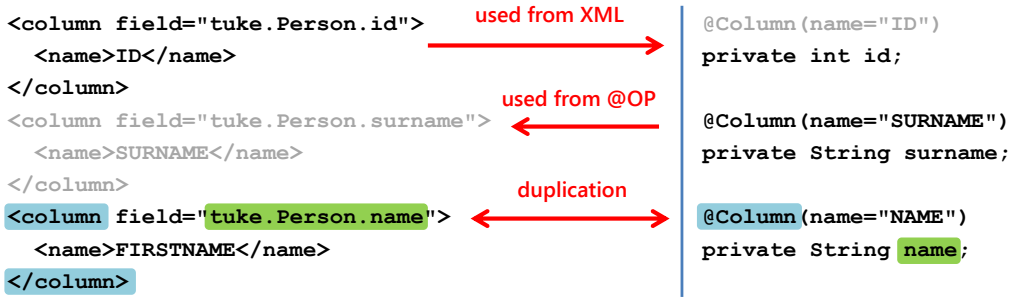

Figure 9 An example of the Mixing Point pattern.

Related Patterns. The Distribution pattern is related to the Mixing Point pattern. While the Mixing Point pattern uses the unique identifier to find conflicts in configuration to prevent duplications, the Distribution pattern uses the identifier of the configuration information to logically bind pieces of information together.

4 Conclusion

This pattern catalogue presents common mapping patterns between XML and @OP. These patterns are the basis for designing new annotation-based or XML-based configuration languages that are based on existing configuration languages. Its contribution lies in recognizing and formalizing mapping patterns from practice. Future work can address mappings between other configuration formats – YAML, .properties, INI files or even proprietary domain-specific languages, if they are built upon generic languages.

Acknowledgements This work was supported by VEGA Grant No. 1/0305/11 Co-evolution of the Artifacts Written in Domain-specific Languages Driven by Language Evolution.

References

1 Sergej Chodarev and Ján Kollár. Language development based on the extensible host language. In *Proceedings of CSE 2012 International Scientific Conference on Computer Science and Engineering*, pages 55–62. EQUILIBRIA, s.r.o., 2012.

2 Diego A. A. Correia, Eduardo M. Guerra, Fábio F. Silveira, and Clovis T. Fernandes. Quality improvement in annotated code. *CLEI Electron. J.*, 13(2), 2010.

3 Erik Duval, Wayne Hodgins, Stuart A. Sutton, and Stuart Weibel. Metadata principles and practicalities. *D-Lib Magazine*, 8(4), 2002.

4 Clovis Fernandes, Douglas Ribeiro, Eduardo Guerra, and Emil Nakao. Xml, annotations and database: a comparative study of metadata definition strategies for frameworks. In *Proceedings of XATA 2010: XML, Associated Technologies and Applications*, XATA 2010, pages 115–126, 2010.

5 Eduardo Guerra, Menanes Cardoso, Jefferson Silva, and Clovis Fernandes. Idioms for code annotations in the java language. In *Proceedings of the 17th Latin-American Conference on Pattern Languages of Programs*, SugarLoafPLoP, pages 1–14, 2010.

6 Eduardo Guerra, Clovis Fernandes, and Fábio Fagundes Silveira. Architectural patterns for metadata-based frameworks usage. In *Proceedings of the 17th Conference on Pattern Languages of Programs*, PLoP2010, pages 1–14, 2010.

7 Ralf Lämmel and Erik Meijer. Revealing the x/o impedance mismatch: changing lead into gold. In *Proceedings of the 2006 international conference on Datatype-generic programming*, SSDGP'06, pages 285–367, Berlin, Heidelberg, 2007. Springer-Verlag.

8 Carlos Noguera and Renaud Pawlak. Aval: an extensible attribute-oriented programming validator for java: Research articles. *J. Softw. Maint. Evol.*, 19(4):253–275, July 2007.

9 Milan Nosáľ and Jaroslav Porubän. Supporting multiple configuration sources using abstraction. *Central European Journal of Computer Science*, 2(3):283–299, October 2012.

10 Renaud Pawlak. Spoon: Compile-time annotation processing for middleware. *IEEE Distributed Systems Online*, 7(11):1–, November 2006.

11 Romain Rouvoy and Philippe Merle. Leveraging component-oriented programming with attribute-oriented programming. In *Proceedings of the 11th International ECOOP Workshop on Component-Oriented Programming*, WCOP'06. Karlsruhe University, July 2006.

12 Wolfgang Schult and Andreas Polze. Aspect-oriented programming with c# and .net. In *Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, ISORC '02, pages 241–. IEEE Computer Society, 2002.

13 Myoungkyu Song and Eli Tilevich. Metadata invariants: checking and inferring metadata coding conventions. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 694–704, Piscataway, NJ, USA, 2012. IEEE Press.

14 Wesley Tansey and Eli Tilevich. Annotation refactoring: inferring upgrade transformations for legacy applications. *SIGPLAN Not.*, 43(10):295–312, October 2008.

15 Eli Tilevich and Myoungkyu Song. Reusable enterprise metadata with pattern-based structural expressions. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, AOSD '10, pages 25–36, New York, NY, USA, 2010. ACM.

16 Hiroshi Wada and Shingo Takada. Leveraging metamodeling and attribute-oriented programming to build a model-driven framework for domain specific languages. In *Proc. of the 8th JSSST Conference on Systems Programming and its Applications*, 2005.