6.- Castor

6.1.-Breve introducción

Castor es un framework de enlace de datos Open Source para Java. Es el camino más corto entre objetos Java, documentos XML y tablas relacionales. Castor proporciona una vinculación de Java a XML, persistencia Java a SQL, y más.
El proyecto de Castor ha sido desarrollado pensando en la necesidad de usar las cosas ya hechas y escribir código reutilizable. Comenzó a desarrollarse por Keith Visco y Assaf Arkin de Intalio, pero ha recibido múltiples contribuciones.
Utiliza software, como Xerces, Yakarta ORO, y Yakarta Regexp, desarrollado por Apache Software Foundation. Castor también incluye JUnit testing framework.
Algunas características son:

- Castor XML: Transforma objetos Java a/desde XML. Genera código fuente de un XML Schema, introspección y archivos mapping para la existencia de modelos de objetos.
- Castor JDO: Persistencia de objetos Java a RDBMS. No es lo mismo ni compatible con JDO de Sun.
- El uso de archivos XML especifica los enlaces existentes entre el modelo de objetos.
- Soporte para schema.
- En memoria caché reduce las operaciones JDBC.
- Transacciones, rollback y detección de bloqueos.
- Permite la persistencia con EJB container mediante OpenEJB.
- Permite crear un mapa de clases Java existentes.
- Permite crear un XML Schema de un documento XML de entrada.

Para la funcionalidad que necesitamos nos centraremos en Castor XML.

6.2.-Castor XML

6.2.1.- Introducción

Castor XML es un framework de enlace de datos de XML. A diferencia de las APIs XML, DOM (Document Object Model) y SAX (Simple API for XML) que tratan con la estructura de un documento XML, Castor permite un trato con los datos definidos en un documento XML y un modelo de objetos que representan dichos datos.
Castor permite transformar casi cualquier objeto Java en un documento XML, y viceversa. En la mayoría de los casos el framework de transformación usa un conjunto de descriptores de clases y de campos (ClassDescriptors y FieldDescriptors) para describir como transformar de un documento XML a objeto.
Se denomina marshal al acto de transformar un objeto a un stream (secuencia de bytes), en nuestro caso un objeto Java a un documento XML; el proceso contrario de convertir un stream a un objeto se conoce como unmarshal.

6.2.2.- El framework marshalling

El framework marshalling, como su nombre implica, es responsable de hacer las conversiones entre Java y XML. El framework consta de dos clases, **org.exolab.castor.xml.Marshaller and org.exolab.castor.xml.Unmarshaller.**

Su uso es simple. Dado un objeto de tipo bean (ej. person), y **FileWriter** (ej. writer) sobre el archivo XML que queremos obtener, realizamos la siguiente llamada:

```
Marshaller.marshal(person, writer);   // Convierte de Java a documento XML.
```

Para el proceso contrario, necesitamos un **FileReader** (ej. reader) sobre el archivo XML que contiene los datos del objeto:

```
Person person=(Person)Unmarshaller.unmarshal(Person.class, reader);
        //Convierte un documento XML en objeto.
```

Si queremos además usar un mapping (cuya funcionalidad se explicará más adelante) se debe crear y cargar antes de las llamadas anteriores.

```
//Se crea el manejador del mapa y se accede al archivo que lo contiene.
Mapping mapping = new Mapping();
mapping.loadMapping("mapping.xml");

Unmarshaller unmarshaller = new Unmarshaller(Person.class);
//Se carga el mapa en el transformador
unmarshaller.setMapping(mapping);
Person person = (Person)unmarshaller.unmarshal(reader);
```

6.2.3.- Usando clases y objetos existentes.

Castor puede transformar casi cualquier objeto a XML y viceversa. Cuando los descriptores no están disponibles usa la reflexión para obtener la información sobre el objeto. Si están disponibles, Castor los utiliza en la transformación.
Existen restricciones sobre los objetos que se pueden transformar. Es necesario que posean:

- Un constructor público por defecto (sin argumentos de entrada).
- Métodos “get” y “set” para todas las propiedades que se vayan a transformar.

6.2.4.- Class Descriptor

La clase Descriptor del framework Castor le proporciona la información necesaria para que las clases java a las que se refieren sean transformadas apropiadamente. La clase Descriptor puede dividirse entre JDO y XML frameworks.
XML Class descriptors proporciona el framework marshalling con la información necesaria sobre una clase en orden para transformar a XML y viceversa. Esta en **org.exolab.castor.xml.XMLClassDescriptor**.
XMLClassDescriptor son creados de 4 maneras. Dos de ellas son básicamente en tiempo de ejecución, y las otras son en tiempo de compilación.

6.2.4.1.- Descriptores en tiempo de compilación.

Para usar clases de descriptores en tiempo de compilación, uno puede implementar la interfaz org.exolab.castor.xml.XMLClassDescriptor para cada clase que necesite ser descrita, o bien, tener el Source Code Generator que crea los oportunos descriptores.
La principal ventaja de los descriptores en tiempo de compilación es que son más rápidos que los de tiempo de ejecución.

6.2.4.2.- Descriptores en tiempo de ejecución.

Para usar clases de descriptores en tiempo de ejecución, puedes o bien usar la introspección de clases que proporciona Castor, o bien proporcionarle en archivo mapping, o bien la combinación de ambas.
La introspección es el proceso de descubrir las características de una clase Java en tiempo de diseño por uno de estos dos métodos.

- Reflexión de bajo nivel, que utiliza patrones de diseño para descubrir las características.
- Examinando una clase asociada que describe explícitamente las características.

Para trabajar con la introspección por defecto, la clase debe tener métodos set/get para cada campo que quiera ser transformado. Si no posee dichos métodos, Castor puede acceder a los campos que sean públicos. Aún así, no puede hacer ambas cosas al mismo tiempo, por lo que si la clase tiene algún método set/get, no se podrá tener acceso al campo directamente.
No hay que hacer nada para que ocurra la introspección por defecto; si no se encuentra un descriptor para nuestra clase, la introspección ocurre automáticamente.
Algunos comportamientos de la introspección pueden ser controlados poniendo las propiedades adecuadas en el archivo castor.properties. Este comportamiento consiste en cambio de convenciones de nombres, y si los tipos primitivos son tratados como atributos o como elementos. Por cambios de convenciones de nombres entendemos que Castor, en la transformación, buscará los nombres de las clases y atributos Java según una serie de reglas prefijadas.
También podemos usar un archivo mapping para describir las clases que se transformarán. El mapa se carga antes de que el proceso de transformación (marshalling/unmarshalling) tenga lugar.
Se necesita org.exolab.castor.mapping.Mapping.
La principal ventaja de descriptores en tiempo de ejecución es que necesitan menos esfuerzo para empezar a trabajar.

6.3.-Castor XML Mapping

6.3.1.- Introducción

Castor XML Mapping es un modo de simplificar el enlace de clases Java a documentos XML. Permite transformar los datos contenidos en un modelo de objetos java a un documento XML, y viceversa.
También es posible confiar en el comportamiento por defecto de Castor para el proceso de transformación de los objetos a documentos XML, pero puede ser necesario ejercer más control sobre su funcionamiento. Por ejemplo, si el modelo de objetos Java ya existe, Castor XML Mapping puede usarse de puente entre el documento XML y dicho modelo, relacionando las clases y atributos Java con los elementos y atributos de un documento XML. El archivo mapping da información explícita a Castor sobre cómo dado un documento XML y dado un conjunto de objetos Java relacionarlos entre sí. Seguiremos profundizando en este tema en los siguientes apartados.
Un archivo mapping de Castor es un buen método para disociar los cambios en la estructura de un modelo de objetos de Java de los cambios correspondientes al formato del documento XML.

6.3.2.- Visión de conjunto

La información de un mapping es especificado por un documento XML. Este documento está escrito desde el punto de vista del modelo de objetos Java y describe cómo las propiedades de los objetos tienen que ser trasladadas a XML. Una restricción para el archivo mapping es que Castor no puede permitir la ambigüedad de cómo un atributo/elemento XML dado debe ser trasladado al modelo de objetos durante la transformación.
El mapping describe para cada objeto cómo cada uno de sus campos tiene que ser mapeado a XML. Un campo es una abstracción de una propiedad de un objeto. Puede corresponder directamente a una variable pública de la clase o indirectamente a una propiedad por un método de acceso (set/get).
Es posible usar el mapping y el comportamiento por defecto de Castor en conjunción: cuando Castor tiene que encargarse de un objeto o de un dato XML pero no puede encontrar información sobre ellos en el mapping, se confiará en el comportamiento por defecto. Castor usará el API de reflexión de Java para determinar la introspección de los objetos Java.[1]

6.3.2.1.- Comportamiento de la transformación de Java a documento XML

Para Castor, una clase Java se mapea en un elemento XML. Cuando Castor transforma un objeto, será:

- Usando la información del mapa, si hay, encuentra el nombre del elemento para crearlo, o
- Por defecto, crea un nombre usando el nombre de la clase.

Después se usa la información de los campos del archivo mapping para determinar como, para una propiedad dada de un objeto, se traslada a uno y sólo uno de los siguientes:

- Un atributo
- Un elemento
- Un texto de contenido
- Nada, si elegimos ignorar ese campo en particular.

Este proceso es recursivo: si Castor encuentra una propiedad que tiene un tipo especificado de clase en otra parte en el mapping, se usará la información para transformar el objeto.
Por defecto, si Castor encuentra que no hay información para una clase dada en el mapping, se producirá la introspección en la clase y aplicará un conjunto de reglas para suponer los campos y hacerles la conversión. Las reglas por defecto son las siguientes:

- Todos los tipos primitivos, incluidos los tipos primitivos envueltos (Bolean, Short, etc.) serán transformados a atributos.
- Todos los otros objetos serán transformados a elementos con contenido de texto o de elemento.

6.3.2.2.- Comportamiento de la transformación de un documento XML a objeto Java.

Cuando Castor encuentra un elemento en un proceso de transformación de un documento, intentará usar la información del mapping para determinar qué objeto instanciar. Si no hay información en el mapping, Castor usará el nombre del

[1] Castor puede no manejar todos los mapping posibles. En algunos casos complejos, pede ser necesario confiar en una transformación XSL en conjunto con Castor para adaptar el documento XML a un formato más amigable.

elemento para intentar adivinar el nombre de la clase e instanciarla (por ejemplo, para un elemento llamado 'test-element', Castor intentará instanciar una clase llamada 'TestElement'). Castor intentará entonces usar la información del campo del mapping para manejar el contenido del elemento.
Si la clase no esta descrita en el Mapping, Castor intentará meterse en la clase usando la API de reflexión de Java para determinar si hay funciones de la forma getXxxYyy()/setXxxYyy(<type> x). Estos accesos están asociados con elementos/atributos XML llamdo xxx-yyy. En el futuro, se proporcionará un modo de no hacer caso del comportamiento por defecto.
Castor sólo entrará en las variables de los objetos usando el método de acceso directo si no existen métodos get/set en la clase. En ese caso Castor buscará variables públicas de la forma:

public <type> xxxYyy;

y espera un elemento/atributo llamado 'xxx-yyy'. Las únicas colecciones que maneja como <type> son java.lang.Vector y array (versión superior a 0.8.10).
Para <type> primitivos, Castor buscará primero un atributo y luego un elemento. Si <type> no es un tipo primitivo, Castor buscará primero un elemento y luego un atributo.

6.3.3.- El archivo Mapping

6.3.3.1.- El elemento <mapping>

El elemento <mapping> es el elemento raíz del archivo mapping.

<!ELEMENT mapping (description?, include, class*, key-generator*)>*

Contiene:
- Una descripción opcional.
- Cero o más <include> que facilitan la reutilización de los mapping.
- Cero o más descriptores de <class>: uno por cada clase de la que se va a dar información en el mapa.
- Cero o más <key-generator>: no se usa para XML mapping.

Estructura típica:

```
<?xml version="1.0"?>
<!DOCTYPE mapping PUBLIC "-//EXOLAB/Castor Mapping DTD Version 1.0//EN"
                         "http://castor.org/mapping.dtd">
<mapping>
    <description>Description of the mapping</description>
    <include href="other_mapping_file.xml"/>
    <class name="A">
        .........
    </class>
    <class name="B">
        .........
    </class>
</mapping>
```

6.3.3.2.- El elemento <class>

El elemento <class> contiene toda la información usada para mapear una clase Java en un documento XML. El contenido de <class> es principalmente usado para describir los campos que serán mapeados.

```
<!ELEMENT class ( description?, cache-type?, map-to?, field+ )>
    <!ATTLIST class
        name            ID      #REQUIRED
        extends         IDREF   #IMPLIED
        depends         IDREF   #IMPLIED
        auto-complete   ( true |false ) "false"
        identity        CDATA   #IMPLIED
        access          ( read-only | shared | exclusive | db-locked )  "shared"
        key-generator   IDREF   #IMPLIED >
```

Descripción de los atributos:

- name: nombre de la clase Java que queremos mapear. Debe aparecer como 'miPaquete.miClase'.
- extends: debe usarse sólo si la clase extiende de otra clase de la cual el mapa proporciona información. No se usará si la clase de la que extiende no es usada en el mapa.
- depends: afecta a JDO.
- auto-complete: si es verdadero, la clase tendrá introspección para determinar sus campos y los campos especificados en el mapping serán usados para no hacer caso a los campos encontrados durante la introspección.
- identity: afecta a JDO.
- access: afecta a JDO.
- key-generator: afecta a JDO.

El atributo auto-complete es interesante para permitir un grado de control más fino de la introspección: es posible especificar sólo los campos para los que el comportamiento por defecto de Castor en la introspección no se ajuste a nuestras necesidades.

Descripción del contenido:

- descripción: opcional <description>.
- cache-type: usado sólo para Castor JDO.
- Un opcional <map-to>: usado si el nombre del elemento no es el mismo que el de la clase. Por defecto, Castor intentará inferir el nombre del elemento a ser mapeado del nombre de la clase: una clase Java que se llame 'XxxYyy' se transformará en 'xxx-yyy'. Si no quieres que Castor genere el nombre, debes usar <map-to> para especificar el nombre que quieres usar para el elemento raíz.
- field: cero o más <field> que serán usados para describir propiedades de los objetos Java.

6.3.3.3.- El elemento <map-to>

Es usado para especificar el nombre del elemento que debe ser asociado con la clase dada, y se usa sólo para el elemento raíz. Si esta información no está presente, Castor intentará:

- Para la transformación de Java a XML, una clase Java llamada 'XxxYyy' se transformará en 'xxx-yyy'.
- Para la transformación de XML a Java, si un elemento es llamado 'test-element', se intentará usar una clase llamada 'TestElement'.

```
<!ELEMENT map-to EMPTY>
 <!ATTLIST map-to
        table              NMTOKEN  #IMPLIED
        xml                NMTOKEN  #IMPLIED
        ns-uri             NMTOKEN  #IMPLIED
        ns-prefix          NMTOKEN  #IMPLIED
        ldap-dn            NMTOKEN  #IMPLIED
        element-definition  (true|false) "false"
        ldap-oc            NMTOKEN  #IMPLIED>
```

Descripción de los atributos:

- xml: nombre del elemento que está asociado a la clase.
- ns-uri: espacio de nombre URI.
- ns-prefix: espacio de nombre deseado.
- element-definition: verdadero si el descriptor es creado de un schema que era el tipo de elemento (en oposición de <complexType>). Esto sólo se usa en contexto de source code generation.
- ldap-dn: no se usa en XML.
- ldap-oc: no se usa en XML.

6.3.3.4.- El elemento <field>

```
<!ELEMENT field ( description?, sql?, bind-xml?, ldap? )>
<!ATTLIST field
        name          NMTOKEN  #REQUIRED
        type          NMTOKEN  #IMPLIED
        handler       NMTOKEN  #IMPLIED
        required      ( true | false )  "false"
        direct        ( true | false )  "false"
        lazy          ( true | false )  "false"
        transient     ( true | false )  "false"
        get-method    NMTOKEN  #IMPLIED
        set-method    NMTOKEN  #IMPLIED
        create-method NMTOKEN  #IMPLIED
        collection    ( array | vector | hashtable | collection | set | map )
        #IMPLIED>
```

Se usa para describir una propiedad de un objeto Java que queremos transformar. Se da:

- Su identidad (‘name’).
- Su tipo (inferido de ‘type’ y ‘collection’).
- Su método de acceso (inferido directamente, por método get o método set).

De esta información Castor permite el acceso a la propiedad dada de la clase Java. Para determinar las formas que Castor espera, hay dos reglas fáciles para aplicar.

- Primero determinar el tipo
  - Si no hay atributo ‘collection’, el tipo será solamente un tipo específico de Java en <type-attribute>. Puede estar indicado con su nombre completo como ‘java.lang.String’, o por un nombre corto:

short name	Primitive type?	Java Class
other	N	java.lang.Object
string	N	java.lang.String

integer	Y	java.lang.Integer.TYPE
long	Y	java.lang.Long.TYPE
boolean	Y	java.lang.Boolean.TYPE
double	Y	java.lang.Double.TYPE
float	Y	java.lang.Float.TYPE
big-decimal	N	java.math.BigDecimal
byte	Y	java.lang.Byte.TYPE
date	N	java.util.Date
short	Y	java.lang.Short.TYPE
char	Y	java.lang.Character.TYPE
bytes	N	byte[]
chars	N	char[]
strings	N	String[]
locale	N	java.util.Locale

Tabla 6.1: Castor: nombres cortos de tipos no collection

- Si hay atributo 'collection', puedes usar la siguiente tabla:

name	<type>	default implementation
array	<type_attribute>[]	<type_attribute>[]
arraylist	java.util.ArrayList	java.util.Arraylist
vector	java.util.Vector	java.util.Vector
hashtable	java.util.Hashtable	java.util.Hashtable
collection	java.util.Collection	java.util.Arraylist
set	java.util.Set	java.util.Hashset
map	java.util.Map	java.util.Hashmap
sortedset	java.util.SortedSet	java.util.TreeSet

Tabla 6.2: Castor: tipos collection admitidos

Es necesario usar una colección cuando se espera más de un elemento del tipo especificado.

- Determinar la forma de la función
  - Si 'direct' está a verdadero, Castor espera encontrar una variable en la clase con la siguiente forma: public <type> <name>;
  - Si 'direct' está a falso o se omite, Castor esperará acceder a la propiedad mediante métodos de acceso. Castor determina la forma del acceso como sigue: si el 'método get' o 'método set' del atributo está suministrado, intentará encontrar una función con la siguiente forma:

    public <type> <get-method>();

public void <set-method>(<type> value);

Si el 'método get' o 'método set' del atributo no está, Castor intentará encontrar la siguiente función:

public <type> get<capitalized-name>();
public void set<capitalized-name>(<type> value);

<capitalized-name> significa que Castor toma el nombre del atributo y pone la inicial en mayúsculas sin modificar las otras letras.

El contenido de <field> es información sobre cómo mapear los campos dados a SQL, XML,...

Descripción de los atributos:

- name: es necesario incluso si no existe el campo en la clase. Si 'direct' es verdadero, 'name' será el nombre público de la variable que se mapea (el campo debe ser público, no static, y no transient). Si no tiene acceso directo y no hay métodos 'get/set' especificados, este nombre será usado para inferir el nombre del método de acceso.
- type: tipo Java del campo. Se usa para acceder al campo.
- required: el campo puede ser opcional o requerido.
- transient: si es verdadero, este campo será ignorado durante la transformación. Se usa cuando esta activa la opción auto-complete.
- direct: si es verdadero, Castor esperará una variable pública en el objeto para modificarlo directamente.
- collection: si se espera más de una ocurrencia del mismo elemento, es necesario especificar que colección de Castor usar. El tipo especificado se usa para definir el tipo del contenido de la colección.
- get-method: nombre opcional del método get que Castor debe usar. Si este atributo no posee dicho método, intentará adivinarlo por el método antes descrito.
- set-method: nombre opcional del método set que Castor debe usar. Si este atributo no posee dicho método, intentará adivinarlo por el método antes descrito.
- create-method: método de factoría para la instanciación de FieldHandler.

Descripción del contenido:
En el caso de mapping XML, el contenido de un elemento campo debe ser un y sólo un elemento, <xml> describiendo el campo como se mapea el documento XML.

6.3.3.5.- El elemento <bind-xml>

Se usa para describir cómo dado un campo Java éste debe aparecer en un documento XML. Se usa tanto en el proceso de marshalling como unmarshalling.

```
<!ELEMENT bind-xml (class?, property*)>
<!ATTLIST bind-xml
    name      NMTOKEN      #IMPLIED
    type      NMTOKEN      #IMPLIED
    location  CDATA        #IMPLIED
    matches   NMTOKENS     #IMPLIED
    QName-prefix NMTOKEN   #IMPLIED
    reference    ( true | false ) "false"
    node         ( attribute | element | text )    #IMPLIED
    auto-naming ( deriveByClass | deriveByField ) #IMPLIED
    transient    ( true | false ) "false">
```

Descripción de los atributos:

- name: nombre del elemento o atributo.
- auto-naming: si el nombre no es especificado, este atributo controla cómo Castor creará automáticamente un nombre para el campo. Normalmente el nombre es creado usando el nombre del campo, pero quizás sea necesario crear el nombre usando el tipo de la clase instanciada.
- type: tipo del XML Schema (del valor de este campo) que requiere especifico mantenimiento en el framework marshalling de Castor.
- location: (desde 0.9.4.4) permite el uso para especificar el "sub-path" por el cual el valor debe ser transformado en ambos sentidos. Esto es útil para envolver valores en elementos o para mapear valores que aparecen en subelementos al elemento actual representado por la clase mapping. Para más información ver Location attribute.
- QName-prefix: cuando un campo representa un valor QName, un prefijo puede ser proporcionado para usarse cuando se transforma como valor de tipo Uname.
- reference: indica si este campo tiene que ser tratado como una referencia para el unmarshalling. En orden de trabajo, tú debes especificar el tipo de nodo a 'attribute' y para ambos el 'id' y el 'reference'.
- matches: permite no hacer caso de las reglas para el nombre del elemento, y usar el del campo name.
- node: indica si el nombre correspondiente al atributo, al elemento, o al contenido del texto. Por defecto, los tipos primitivos son asumidos para ser un atributo y de otra manera el nodo es asumido para ser un elemento.
- transient: se tiene en cuenta para hacer este campo transitorio para XML. El valor por defecto es obtenido del elemento campo.
- Nested class mapping: (desde 0.9.5.3) el elemento bind-xml soporta una necesidad de clase mapping, que es a veces útil cuando necesitas especificar más de un mapa para una clase particular.

6.3.4.- xsi:type

Normalmente, un mapping sólo referenciará tipos que sean de clases concretas (no clases abstractas ni interfaces). Esta razón es que para la transformación de documento XML a Java, se requiere instanciar las clases para convertir los tipos y no se puede instanciar una interfaz. Quizás, en muchas situaciones reales, el modelo de objetos depende del uso de interfaces. Muchas propiedades de las clases son definidas para tener tipos de interfaces que soporten la habilidad de cambiar implementaciones. Este es el caso frecuente en frameworks.

El problema es que se debe usar un mapping diferente cada vez que el mismo modelo se use en la transformación con una implementación que use tipos concretos diferentes, y esto no es conveniente. El mapping debe representar el modelo y la especificación del tipo concreto usado para procesar un documento debe representarse en un parámetro de configuración; debe ser especificado en el documento instanciado para ser transformado.

Por ello, en el documento XML se debe indicar:

```
<nombreDeClase>
    <nombreElementoInterfazOAbstracto
    xsi:type="paquete.nombreDeLaInstanciaciónConcreta" />
</nombreDeClase>
```

6.3.5.- Location attribute

Aplicable desde la versión 0.9.5.

El atributo Location permite especificar un elemento envuelto por un campo dado. Elementos envueltos son simplemente elementos que aparecen en la instancia XML,

pero no tienen un mapeo directo a un objeto o campo dentro del modelo de objetos.
Por ejemplo para mapear una instancia de la siguiente clase:

```
public class Foo {
    private Bar bar = null;
    public Foo();
    public getBar() {
      return bar;
    }
    public void setBar(Bar
bar) {
      this.bar = bar;
    }
  }
```

En la siguiente instancia de XML:

```
<?xml version="1.0"?>
<foo>;
  <abc>
    <bar>...</bar>
  </abc>
</foo>
```

Podemos usar el siguiente mapping:

```
<?xml version="1.0"?>
  ...
  <class name="Foo">
    <field name="bar" type="Bar">
      <bind-xml          name="bar"
location="abc"/>
    </field>
  </class>
  ...
</mapping>
```

Nótese que en el atributo “location”, el valor de éste atributo es el nombre del elemento envuelto. Para usar más de un elemento envuelto, el nombre es separado por /:

```
<bind-xml name="bar" location="abc/xyz" />
```

Además, el nombre del elemento no es parte de su localización y la localización es siempre relativa a la clase en la que el campo es definido. Esto funciona para atributos también:

```
<bind-xml name="bar" location="abc" node="attribute" />
```

Producirá lo siguiente:

```
<?xml version="1.0"?>
<foo>
  <abc bar="..."/>;
</foo>
```