

XML E XSLT PER LA GENERAZIONE AUTOMATICA DI CODICE

Autore : ANDREA FRASCA

Quando si parla di XSLT (o della sua sottoclasse XSL) si pensa normalmente alla trasformazione di un file XML in HTML o altri formati XML. Tuttavia è possibile trasformare un XML in molti altri formati : ex. rtf, pdf, testo semplice. In questo breve tutorial vedremo un esempio di come usare XSLT per generare un programma java (l'output quindi è testo semplice) a partire da un file XML.

Una volta appreso come usare alcuni costrutti fondamentali XSLT, si avranno tutte le conoscenze necessarie per generare file di testo in qualsiasi altro formato e quindi usarli per generare codice per i nostri otter, mace, nuSmv ecc.

I componenti fondamentali per la generazione automatica di codice sono :

1. **Meta-dati** : usati per descrivere le caratteristiche del programma da generare. Essi possono essere rappresentati in diversi formati, noi useremo la rappresentazione tramite file XML.
2. **Modello di programma**: descrive la struttura sulla quale basarsi per generare il file di output rispetto ai meta-dati (XML) . Nel nostro caso il modello è il file XSL.

Il processo di generazione di codice usa dunque i "meta-dati" più il "modello di programma" per generare un nuovo programma. I programmi generati, condividono le caratteristiche del "modello di programma" in input quindi, se il modello è sintatticamente corretto e ben formato, risulterà corretto e ben formato anche file di output.

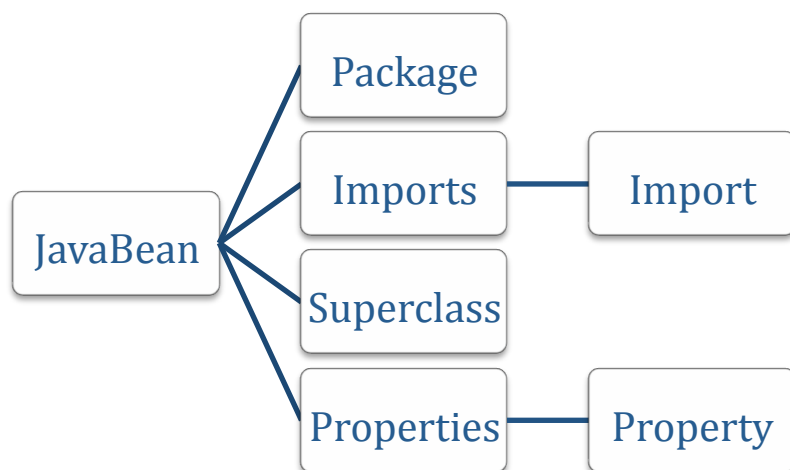
Esempio esplicativo : Generazione di un JavaBean tramite XSL

Vediamo adesso più in dettaglio come applicare il processo di generazione di codice ad un esempio pratico.

Genereremo di seguito tramite XML e XSL il codice di un semplice JavaBean.

Una volta apprese i principi generali che ci sono dietro, si riusciranno a generare molti programmi usando questa tecnica.

1. **Si analizzano le caratteristiche del file da generare**
Un JavaBean per definizione è una classe con un costruttore nullo e alcune proprietà con relativi metodi *get* e *set*.
2. **Si progetta il file XML contenente i meta-dati della nostra classe JavaBean.**
Un JavaBean è una classe java e deve contenere un package, una serie di import, la superclasse e le sue proprietà specifiche.



L'elemento Imports può contenere da 0 a N elementi Import.

L'elemento Properties può contenere da 1 a N elementi Property.

3. **Si realizza il file XML del nostro JavaBean specifico basato sulla struttura progettata nel punto 2**
Un esempio di realizzazione è il seguente :

```
<?xml version="1.0" encoding="UTF-8"?>
```

```

<JavaBean nome="UtenteBean">

    <Package>progetto.beans</Package>

    <Imports>
        <Import>java.util.Date</Import>
    </Imports>

    <Superclass nome="Object"/>

    <Properties>
        <Property name="nome" type="String"/>
        <Property name="cognome" type="String"/>
        <Property name="indirizzo" type="String"/>
        <Property name="dataNascita" type="Date"/>
    </Properties>

</JavaBean>

```

La prima riga specifica la versione del nostro file xml e la codifica utilizzata.

Subito dopo si comincia con l'elemento `JavaBean`, il suo attributo `nome` rappresenta il nome della classe.

L'elemento `Package` rappresenta il package della classe.

Anche se possiamo avere più elementi `Import` al di sotto dell'elemento `Imports` in questo caso né consideriamo uno solo.

La superclasse del nostro bean è specificata nell'attributo `nome` dell'elemento `Superclass`.

Ci sono quattro elementi `Property` sotto l'elemento `Properties`.

Gli elementi `Property` saranno usati per generare le variabili di istanza e i metodi `get` e `set` del nostro bean.

4. Si realizza il file XSL che strutturerà il nostro file di output

La prima parte di un possibile esempio di file XSL per processare l'XML definito precedentemente è :
(in rosso testo semplice (in questo caso direttive java) per strutturare il file di testo di output!)

```

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:output method="text"/>

<xsl:template match="/">

    package <xsl:value-of select="//Package"/>;

    <xsl:apply-templates select="//Import"/>

    public class <xsl:value-of select="JavaBean/@name"/> extends
    <xsl:value-of select="//Superclass/@name"/> {

    <xsl:apply-templates select="//Property" mode="var"/>

    public <xsl:value-of select="JavaBean/@name"/> ()
    {
    }

    <xsl:apply-templates select="//Property" mode="get"/>

    <xsl:apply-templates select="//Property" mode="set"/>

    }

</xsl:template>

<!-- fine prima parte -->

```

Cominciano a spiegare questa prima parte di documento XSL(T).

La prima riga è la dichiarazione di quale versione di XSL stiamo usando in accordo con le specifiche W3C XSLT.

```

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

```

Nella seconda riga useremo l'espressione `<xsl:output method="text"/>` indicando che noi vogliamo come output del "testo semplice" e non un formato specifico (rtf, pdf, html, xml...).

Dopo l'inizializzazione, si comincia ad usare uno degli elementi fondamentali per i nostri scopi: `xsl:template`. Esso viene usato per costruire dei template relativi agli elementi.

Il tag `xsl:template` ha tre attributi principali: `match`, `name` e `mode`.

- L'attributo `match` è usato per associare un template ad un elemento del file XML di input.
Ad esempio se nel nostro XML abbiamo un elemento `<nome>Andrea</nome>` e nel nostro file XSL specifichiamo il template `<xsl:template match="nome">` allora esso specificherà nel suo corpo cosa dovremo "fare" quando incontreremo l'elemento `nome` nel file XML per tutte le sue occorrenze.
N.B. Le espressioni contenute nell'attributo `match` sono relative sempre al nodo corrente. Questo perché un template può essere richiamato all'interno di un altro template e il riferimento di nodo corrente cambia. Quindi il loro valore deve essere usato a seconda del nodo corrente in cui ci troviamo.
 - Con `<xsl:template match="/">` indichiamo il match con l'elemento root del nostro file XML.
 - Con `<xsl:template match="nodop/nodof">` indichiamo il match con l'elemento `nodof` figlio del nodo `nodop` che è figlio del nodo corrente che stiamo processando.
 - Con `<xsl:template match="//nodo">` indichiamo il match con l'elemento `nodo` del nostro file XML in qualsiasi posizione annidata esso si trovi.
- L'attributo `mode` indica la modalità di processing del nostro template.
Questo perché magari vogliamo usare più template diversi per lo stesso `match`.
In questo modo si possono avere due o più template che hanno la stessa espressione di `match` (ovvero si riferiscono agli stessi nodi) ma che vengono utilizzati in casi distinti.
La modalità viene specificato dall'elemento `xsl:apply-templates`.
Vedremo in seguito un'applicazione esplicativa nel nostro esempio specifico.
- L'attributo `name` definisce il nome del template di modo da poterlo richiamare esplicitamente nel documento.
Usabile con chiamata diretta usando l'elemento `xsl:call-template`
Generalmente parametrizzato usando `xsl:with-param`
Anche in questo caso vedremo in seguito un'applicazione esplicativa nel nostro esempio specifico.

Andiamo dunque a specificare il nostro template di `match` con la root `<xsl:template match="/">`

Questo template costruisce il guscio del programma java che stiamo generando:

- Si comincia con il creare la riga in cui si specifica il package del nostro JavaBean.
L'elemento `<xsl:value-of>` è usato per prendere il dato appropriato dal documento XML, nel nostro caso usando `<xsl:value-of select="//Package"/>` preleveremo il valore del nodo `<Package>`.
- Per secondo dovremo scrivere tutti i nostri import, quindi dovremo definire un template specifico per gli elementi di Imports. L'elemento `<xsl:apply-templates select="//Import"/>` indica l'applicazione di un template esterno creato appositamente per processare tutti gli elementi `Import` da aggiungere al nostro file XLS.

```
<xsl:apply-templates select="//Import">
    import <xsl:value-of select=".">
</xsl:apply-templates>
```

Questo template verrà invocato una volta per ogni nodo `Import`. Usando un template specifico e annidandovi la stringa di specifica Java **import** essa non verrà scritta nel nostro file di output se non ci sono import per la nostra classe.

- Dopodiché per la dichiarazione di classe si prelevano con questa notazione gli attributi `JavaBean/@nome` e `SuperClass/@nome` dal file XML (notare la `@` per prelevare il valore degli attributi di un elemento!).
- Il costruttore della nostra classe Java viene creato allo stesso modo.
- Nota a parte per le righe di codice riguardanti le proprietà del nostro JavaBean:

```
<xsl:apply-templates select="//Property" mode="instanceVariable"/>
```

...

```
<xsl:apply-templates select="//Property" mode="get"/>
```

```
<xsl:apply-templates select="//Property" mode="set"/>
```

Visto che l'istanziamento delle variabili, i metodi get e i metodi set del nostro JavaBean prendono dati tutti quanti dagli elementi Property, è stato scelto di usare (ragionevolmente) uno stesso template (che matcha con l'elemento Property) con tre modalità diverse per scrivere :

1. La prima modalità scrive le variabili di istanza del JavaBean
2. La seconda modalità scrive i metodi "get"
3. La terza modalità scrive dei metodi "set"

- Il template con la prima modalità sarà :

```
<xsl:template match="Property" mode="var">
    private <xsl:value-of select="@tipo"/><xsl:text> </xsl:text><xsl:value-of
select="@nome"/>;
</xsl:template>
```

L'elemento `<xsl:text> </xsl:text>` è usato per inserire uno spazio nel file di output.

L'output ad esempio atteso è del tipo `:private String nome;`

- Per la seconda e terza modalità facciamo vedere come usare ulteriori costrutti XSL che ci potranno essere sempre utili in futuro in particolari template con parametri e funzioni.

Diciamo che abbiamo bisogno di una piccola routine di appoggio che ci permette di effettuare al seguente trasformazione spiegata con un esempio.

Avendo un elemento del tipo :

```
<Property nome="indirizzo" tipo="String"/>
```

noi vorremmo in output i corrispettivi `getIndirizzo()` e `setIndirizzo()`.

Poiché la semplice concatenazione porterebbe al risultato `getindirizzo()` e `setindirizzo()` e a noi non ci sta bene, dobbiamo trovare un modo per trasformare la prima lettera del valore dell'attributo `nome` di modo da portarla da lower-case ad upper-case.

Costruiremo quindi un template di appoggio che realizza questo compito usando il concetto di parametro.

```
<xsl:template name="initLowUp">
    <xsl:param name="x"/>
    <xsl:value-of select="translate(substring($x,1,1)
                                , 'abcdefghijklmnopqrstuvwxyz'
                                , 'ABCDEFGHIJKLMNOPQRSTUVWXYZ')"/>
    <xsl:value-of select="substring($x,2)"/>
</xsl:template>
```

Notiamo che in questo caso usiamo solo l'attributo `name` senza usare il `match` poiché noi useremo questo template solo per richieste dirette.

Questo template prende una variabile `x` come parametro : `<xsl:param name="x"/>`.

Quindi quando lo richiameremo gli passeremo questo parametro.

Il primo carattere di `x` è estratto con la funzione predefinita `substring(String s, int i1, int i2)` che suddivide la stringa da `i1` a `i2`.

Si sfrutta poi la funzione `"translate(String s1,String s2,String s3)"` che preso il primo argomento, ne verifica le occorrenze di carattere nel secondo argomento, le sostituisce con i caratteri corrispondenti nella stessa posizione del terzo argomento ridandoli in output.

Questa lettera è poi portata in output usando la `<xsl:value-of ... >`. I rimanenti caratteri della stringa sono estratti da `substring($x,2)` espressa poi in output anche essa con `<xsl:value-of ... >`.

Vediamo dunque adesso le due modalità del template per Property per scrivere i metodi "get" e "set" :

Modalità 2(scrive metodi get) :

```
<xsl:template match="Property" mode="get">
    public <xsl:value-of select="@tipo"/> get<xsl:call-template name="initLowUp">
```

```

        <xsl:with-param name="x" select="@nome"/>
    </xsl:call-template>() {
    return <xsl:value-of select="@nome"/>;
}
</xsl:template>

```

Modalità 3(scrive metodi set) :

```

<xsl:template match="Property" mode="set">
    public void set<xsl:call-template name="initLowUp">
        <xsl:with-param name="x" select="@nome"/>
    </xsl:call-template>(<xsl:value-of select="@tipo"/>
    param<xsl:call-template name="initLowUp">
        <xsl:with-param name="x" select="@nome"/>
    </xsl:call-template>() {
        <xsl:value-of select="@nome"/> = param<xsl:call-template
name="initLowUp">
        <xsl:with-param name="x" select="@nome"/>
    </xsl:call-template>;
    }
</xsl:template>

```

L'unica parte ignota di questi template sono le direttive per richiamare i template con parametri :

```

<xsl:call-template name="initLowUp">
    <xsl:with-param name="x" select="@nome"/>
</xsl:call-template>

```

con `xsl:call-template` viene richiamato il template per convertire la prima lettera dell'attributo da lower-case ad upper case; ovviamente dobbiamo passargli il parametro da processare, questo viene fatto con l'elemento `xsl:with-param` con nome del parametro e nella `select` il valore da convertire (in questo caso l'attributo `nome`).

Dall'uso dei nostri due file XML e XSL costruiti si arriva a questo output :

```
package progetto.beans;
```

```

import java.util.Date;

public class UtenteBean extends Object {

    private String nome;

    private String cognome;

    private String indirizzo;

    private Date numeroTelefono;

    public UtenteBean() {}

    public String getNome() {

        return nome;

    }

    public String getCognome() {

        return cognome;

    }

    public String getIndirizzo() {

        return indirizzo;

    }

    public Date getNumeroTelefono() {

        return numeroTelefono;

    }

    public void setNome(String paramNome) {

        nome = paramNome;

    }

    public void setCognome(String paramCognome) {

        cognome = paramCognome;

    }

    public void setIndirizzo(String paramIndirizzo) {

        indirizzo = paramIndirizzo;

    }

    public void setNumeroTelefono(Date paramNumeroTelefono) {

        numeroTelefono = paramNumeroTelefono;

    }

}

```

Abbiamo illustrato un semplice esempio di come generare codice automaticamente dato un file strutturato e specificandone il modo di processamento. Ovviamente questa è solo una piccola parte della vastità di cose che ci permette di fare l'unione XML/XSLT.

Per approfondimenti, guide e tutorial riguardo tutte le specifiche su XSLT si rimanda ai seguenti link (indicati dal collega Claudio DiCiccio) :

- <http://www.w3schools.com/xsl/default.asp>
- http://www.w3schools.com/xsl/xsl_w3celementref.asp
- <http://www.w3.org/TR/XSLT>
- <http://www.zvon.org/xxl/XSLTutorial/Books/Output/contents.html>
- <http://www.ibm.com/developerworks/xml/library/x-tiploop.html>
- http://www.oreillynnet.com/pub/a/oreilly/java/news/javaxslt_0801.html