Consiglio Nazionale delle Ricerche

XML SCHEMA

Best Practices

A. Marchetti, M. Tesconi, T. Bacci, M. Rosella

IIT B4-02/2006

Nota Interna

Maggio 2006

Istituto di Informatica e Telematica

XML Schema
Best Practices

Andrea Marchetti, Maurizio Tesconi, Tatiana Bacci, Marco Rosella

[1]CNR, IIT Department, Via Moruzzi 1, I-56124, Pisa, Italy
{andrea.marchetti, maurizio.tesconi}@iit.cnr.it

keywords: xml, xml schema

acm classification: Markup languages

Abstract

An XML Schema is a particular XML document allowing to validate the content and the structure of XML data.
This paper aims to describe a series of “Best Practices” for designing XML Schemas. The guidelines explain pros and cons about some design issues in order to help a schema designer to make the right decisions.
The first section presents the base concepts about XML Schema, exploring the main components of a schema: namespaces, simple and complex elements, simple and complex type, attributes and other components.
The second section shows a series of best practices related to some design issues like: the use of namespaces; the hiding or exposing of namespaces; about a declaration of element or type the choice between global or local way; about the declaration of an item the choice between element and type; the use of zero, one or more namespaces; the creation of an element that has a variable content; about the design of a schema the choice between to build type hierarchies (design by subclassing) or aggregate components (design by composition); the creation of extensible content models; the extension of schemas; the use between URL and URN in namespaces; the versioning of XML Schema and the hierarchy of substitutionGroup elements.

Introduzione

XML (Extensible Markup Language) è un linguaggio di markup aperto e basato su testo che fornisce informazioni di tipo strutturale e semantico relative a dei dati veri e propri.
Un documento XML consente di separare e gestire indipendentemente contenuto, struttura logica e presentazione attraverso la creazione di una struttura di markup che identifichi i diversi dati.
Un XML Schema è un particolare tipo di documento XML che permette e convalidare il contenuto e la struttura dei dati XML.
Il primo metodo che è stato utilizzato per questo scopo è stato DTD (Document Ttpe Definition) con il passare del tempo ha presentato alcune limitazioni:

- DTD non è estensibile come un linguaggio XML.
- DTD non supporta i data types.
- C'è una difficile coesistenza tra DTD e namespaces.
- Non è possibile imporre vincoli al contenuto testuale, in particolare agli attributi.

Uno schema XML consente di definire e descrivere determinati tipi di dati XML mediante il linguaggio XSD (XML Schema Definition).
Gli elementi degli schemi XML, ovvero elementi, attributi, tipi e gruppi, vengono utilizzati per definire la struttura valida, il contenuto dei dati valido e le relazioni di determinati tipi di dati XML. Essi defiscono quindi:

- quali elementi ed attributi possono apparire in un documento
- i data-type degli elementi ed attributi
- i valori predefiniti per elementi ed attributi
- le circostanze in cui un elemento è vuoto o può includere testo
- quali elementi sono elementi child (figlio)
- il numero e l'ordine degli elementi child

Di seguito sono riportati alcuni dei vantaggi offerti dagli schemi XML rispetto alle tecnologie precedenti, ad esempio i DTD:

- Negli schemi XML viene utilizzata la sintassi XML, per cui non è necessario apprendere una nuova sintassi per definire la struttura dei dati.
- Negli schemi XML sono supportati tipi riutilizzabili ed è consentita la creazione di nuovi tipi mediante l'ereditarietà.
- Gli schemi XML consentono di raggruppare elementi per controllare la ricorrenza di elementi e attributi.
- In XML Schema, i namespaces sono supportati.

Il presente documento verrà diviso in due parti.
Nella prima parte verranno analizzati i concetti di base su XML Schema, esplorando le componenti principali riguardanti la struttura di uno schema.
Nella seconda, invece, verranno enunciate una serie di "Best Practices"(soluzioni migliori) adottate nella progettazione di un documento XML Schema. L'obiettivo principale non è quindi quello di dettare delle regole, ma di creare un serie di linee guida che facciano luce sui pro ed i contro di ogni problema di progettazione e che aiutino quindi un progettista di schemi a compiere decisioni intelligenti nella costruzione degli stessi.

1. Concetti di base su XML Schema

In questa parte verrà descritta la struttura di un documento XML Schema.
Descriveremo brevemente alcuni concetti di base tra cui l'**elemento radice**, i **namespace**, gli **element**, i **complexType** e i **simpleType**, i **simpleContent** e i **complexContent**, gli **attributi**.

1.1. Elemento radice

Uno schema è costituito da un insieme di componenti; le più interessanti sono: **element**, **complexType**, **simpleType**, **complexContent**, **simpleContent** e **annotation**.

Dopo la dichiarazione XML (`<?xml version="1.0"?>`), un documento XML Schema presenta al suo interno un elemento radice chiamato `<xsd:schema/>`, come vediamo nell'esempio:

```
<?xml version="1.0"?>
<xsd:schema>
       ...
       ...
</xsd:schema>
```

Un concetto importante legato all'elemento radice è il campo di visibilità: le dichiarazioni degli elementi o gli attributi posti all'interno dell'elemento radice hanno **visibilità globale**, mentre gli elementi inseriti all'interno di un tag `complexType` (che vedremo in seguito) hanno **visibilità locale**. La visibilità globale indica una definizione in maniera assoluta equindi permette il riutilizzo dei componenti perché visibili a tutto lo schema; al contrario i componenti locali sono invisibili al resto dello schema e ad altri schemi.

1.2. Namespace

Prima di procedere con l'analisi del documento è necessaria una breve introduzione al concetto di "namespace XML".
Un Namespace XML, secondo la versione del W3C, è "una raccolta di nomi identificata da un riferimento URI; tali nomi vengono utilizzati nei documenti XML come tipi di elementi e nomi di attributi". Il motivo principale che porta a definire un Namespace XML è il fatto di evitare conflitti di nomi quando si utilizzano e ri-utilizzano più vocabolari.
I namespace sono il meccanismo creato appositamente per definire un identificativo univoco dei vari tag di markup.
In generale, un namespace viene dichiarato in questo modo:

```
<pfx:root xmlns:pfx="http://www.foo.com" />
```

Dove xmlns è la parola utilizzata per dichiarare un namespace. In questo caso ogni elemento che si riferisce a quel namespace dovrà avere il prefisso pfx. Per evitare di utilizzare il prefisso è possibile usare un namespace di default (chiaramente dovrà essere l'unico namespace di default). Rivediamo l'esempio precedente:

```
<root xmlns="http://www.foo.com" />
```

Diversamente dal caso precedente l'elemento root non ha il prefisso pfx, perché è dichiarato in modo da essere associato al namespace `http://www.foo.com,` che in questo caso sarà il namespace di default.
Sebbene la specifica su XML-Schema non imponga l'utilizzo dei namespaces è bene farne uso per garantire una buona elaborazione dello schema ed evitare eventuali collisioni con i nomi dei tag utilizzati.
Ogni schema, in generale, fa uso di due namespaces: il namespace XML Schema e il targetNamespace per gli elementi descritti dallo schema.
Aggiungiamo al nostro schema XML di esempio le relative dichiarazioni namespace, dichiarati attraverso gli attributi dell'elemento radice.:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
    targetNamespace="http://www.miur.it/anagrafeStudenti"
    xmlns="http://www.miur.it/anagrafeStudenti"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">
  …
  … …
</xs:schema>
```

Come possiamo vedere dall' esempio, l'elemento radice ha come attributi targetNamespace, xmlns, xmlns:xsd, elementFormDefault ed attributeFormDefault.

L'attributo "targetNamespace" permette di dichiarare il namespace dei componenti creati con lo schema e accetta come valore un URI. Questo attributo non è obbligatorio perché non è obbligatorio che sia presente un namespace degli elementi che vengono dichiarati nello schema. Infatti, nel caso in cui non sia presente il targetnNamespace, gli elementi del documento verranno validati come elementi non qualificati da un namespace. Anche se la specifica di XML Schema prevede che la dichiarazione di targetNamespace sia obbligatoria quando sono presenti nomi di tag che prevengono da più namespace.

L'attributo "xmlns" indica il namespace di default al quale appartengono tutti i componenti dello schema che sono privi di prefisso.

L'attributo "xmlns:xsd" introduce il prefisso xsd per individuare uno spazio dei nomi al quale appartengono i componenti di Xml Schema individuati dalla specifica.

L'attributo "elementFormDefault" permette di indicare se gli elementi, definiti nello schema, debbano avere o meno un prefisso di default qualora il loro uso dovesse avvenire all'interno di un istanza. In questo caso l'attributo ha come valore "qualified" e questo comporta che gli elementi nell'istanza debbano avere un prefisso.

Allo stesso modo, l'attribute "attributeFormDefault" permette di indicare se gli attributi, definiti nello schema, debbano avere o meno un prefisso di default qualora il loro uso dovesse avvenire

all'interno di un istanza. In questo caso l'attributo ha come valore "unqualified" e questo comporta che gli attributi nell'istanza non debbano avere un prefisso.

Riferimento a uno schema in un'istanza di documento

Consideriamo invece il caso in cui l'istanza non utilizza nessun namespace. Questo può essere fatto in due step:

1. dichiarazione del namespace:

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

2. indicazione della locazione dello schema:

```
xsi:noNamespaceSchemaLocation = "C:\My Writing\XML
SchemaEssentials\Ch01\Book.xsd"
```

Il primo attributo permette di associare il prefisso `xsi` all'URI mostrato, mentre il secondo consente di indicare la directory in cui è stato salvato lo schema che vogliamo associare all'istanza. Vediamo un esempio di documento XML:

```
<?xml version="1.0"?>
<Book pubCountry="USA"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="C:\My Writing\XML Schema
  Essentials\Ch01\Book.xsd">
  <Title>XML Schema Essentials</Title>
  <Authors>
    <Author>R. Allen Wyke</Author>
    <Author>Andrew Watt</Author>
  </Authors>
  <Publisher>John Wiley</Publisher>
</Book>
```

Possiamo vedere che `xmlns` e `noNamespaceSchemaLocation` sono entrambi attributi dell'elemento radice del documento XML, che nel caso specifico è `<Book>`. Chiaramente l'attributo `xsi:noNamespaceSchemaLocation` può essere utilizzato solo dopo che il prefisso `xsi` è stato dichiarato.

1.3. Dichiarazione di elementi e definizione di tipi

In XML Schema è molto importante la differenza che esiste tra *dichiarare* un elemento e *definire* un nuovo tipo. Un elemento può essere semplice (*simple element*) o complesso (*complex element*) così come allo stesso modo un nuovo tipo di dato può essere semplice (*simple type)* o complesso *(complex type).*

Definizione di *Simple type*

La definizione di *simple type* è associata ai particolari elementi che possono contenere solo testo e non hanno quindi né elementi child (figli) né attributi. Questi elementi prendono il nome di *simple element*.
La sintassi di definizione di un simple element è la seguente:

```
    <xsd:element name="xxx" type="yyy"/>
```

L'attributo `name` rappresenta il nome dell'elemento, mentre l'attributo `type` prende il nome di *data type.* I simple type possono essere *predefiniti*, quindi propri del linguaggio XML Schema, oppure *personalizzati* dall'autore dello schema.
I principali data type predefiniti sono i seguenti: `xsd:string, xsd:decimal, xsd:integer, xsd:boolean, xsd:date, xsd:time`.
Ad esempio, se vogliamo rappresentare nello schema questi elementi XML:

```
    <name>Dante Alighieri</name>
    <born>12-7-1261</born>
    <age>40</age>
```

possiamo utilizzare le seguenti definizioni:

```
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="born" type="xsd:date"/>
    <xsd:element name="age" type="xsd:integer"/>
```

I tipi personalizzati sono invece una derivazione o meglio una restrizione (*restriction*) di quelli predefiniti. Nell'esempio seguente definiamo un elemento di nome `friends` dove all'interno del tag cheindica che si tratta di un tipo semplice (`simpleType`) viene definita una particolare restrizione (tramite il tag `restriction`) in cui sono presenti - e quindi consentiti - solo quattro valori di tipo stringa:

```
    <xsd:element name="friends">
    <xsd:simpleType>
      <xsd:restriction base="xs:string">
        <xsd:enumeration value="Virgilio"/>
        <xsd:enumeration value="Beatrice"/>
        <xsd:enumeration value="Guido"/>
        <xsd:enumeration value="Lapo"/>
      </xsd:restriction>
    </xsd:simpleType>
</xsd:element>
```

Definizione di Complex Type

I tipi di dato complesso, *complex type*, sono riferiti ad elementi *complex element* contenenti uno o più dichiarazioni di elementi, riferimenti ad altri elementi e dichiarazioni di uno o più attributi.

Consideriamo di nuovo la seguente istanza:

```
<?xml version="1.0"?>
<Book pubCountry="USA"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="C:\My Writing\XML Schema
  Essentials\Ch01\Book.xsd">
  <Title>La Divina Commedia</Title>
  <Author>Dante Alighieri</Author>
</Book>
```

L'elemento `<Book>` ha come figli gli elementi `<Title>` e `<Author>`. La dichiarazione dell'elemento `<Book>` innesterà quindi al suo interno la definizione di un complex type in cui verranno definiti gli altri due elementi in una sequenza:

```
<?xml version="1.0"?>
  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="Book">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="Title"/>
          <xsd:element name="Author"/>
        <xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  <xsd:schema>
```

Spesso nasce l'esigenza di utilizzare un complex type più di una volta all'interno dello stesso documento XML: XML Schema permette di creare complex type riutilizzabili, e questo può essere fatto semplicemente come possiamo vedere nell'esempio:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:complexType name="BookPublication">
       <xsd:sequence>
         <xsd:element name="Title"/>
         <xsd:element name="Author"/>
       <xsd:sequence>
       <xsd:attribute name="kind" type="xsd:string"/>
   </xsd:complexType>
   <xsd:element name="Book" type="BookPublication"/>
<xsd:schema>
```

L'elemento `<xsd:complexType>` contiene le informazioni che definiscono il complex type "BookPublication". Questo tipo sarà definito come una sequenza di elementi ognuno dei quali ha

un contenuto simple type, e sarà utilizzato dall'elemento di nome `book` e qualcuno altro elemento ne indichi il nome nell'attributo `type`.
Adesso cerchiamo di capire come dichiarare, in XML Schema, un elemento che può contenere sia elementi figli che caratteri. Questo è possibile grazie all'attributo "mixed" dell'elemento complexType, che vediamo in un esempio:

```
<?xml version=”1.0”?>
<xsd:schema xmlns:xsd=“http://www.w3.org/2001/XMLSchema“>
  <xsd:element name=“Book“>
  <xsd:complexType mixed=”true”>
       <xsd:sequence>
         <xsd:element name=”Title”/>
         <xsd:element name=”Author”/>
       <xsd:sequence>
   </xsd:complexType>
   <xsd:element name=“Book“ type=”BookPublication”
<xsd:schema>
```

In questo modo l'elemento Book potrà avere degli elementi figli `title` e `author`, ma contenere anche dei caratteri. Quindi nell'istanza avremo:

```
<?xml version="1.0"?>
<Book xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="Book.xsd">
  <Title>La Divina Commedia</Title>
  <Author>Dante Alighieri</Author>
La Divina Commedia consta di più di quattordicimila
(esattamente 14.223) endecasillabi, distribuiti in cento
canti di oscillante ampiezza (da un minimo di 115 a un
massimo di 160 versi), raggruppati in tre cantiche:
Inferno; Paradiso di 33 canti con 4758 versi in tutto.
</Book>
```

Dichiarazione di elementi vuoti

In alcuni casi potrebbe essere necessario dichiarare degli elementi vuoti, cioè che non hanno nessun contenuto (*empty element*). Gli elementi vuoti sono complex type perché in generale possono avere degli attributi. Supponiamo di voler creare un elemento che contiene tutte le sue informazioni negli attributi e quindi non ha contenuto:

```
 <rect id=”window” x=”60px” y=”237” width=”50” height=”50”/>
```

L'elemento sarà definito in questo modo:

```
       <xsd:element name="rect">
         <xsd:complexType>
           <xsd:attribute name="id" type="xsd:string "/>
           <xsd:attribute name="x" type="xsd:positiveInteger"/>
           <xsd:attribute name="y" type="xsd:positiveInteger"/>
```

```
            <xsd:attribute name="width" type="xsd:positiveInteger"/>
            <xsd:attribute name="height" type="xsd:positiveInteger"/>
          </xsd:complexType>
        </xsd:element>
```

Per definire le informazioni negli attributi vediamo quindi l'utilizzo dell'elemento attribute, che esamineremo in seguito più in dettaglio.

1.4. Attributi

Gli attributi, in XML, rappresentano una funzione importante perché permettono di aggiungere informazioni agli elementi. Supponiamo per esempio di avere un documento XML la cui radice è l'elemento `<person>`. Vogliamo inoltre specificare che esistono diversi tipi di persone:

- Customer
- Employee
- Family
- Friend

Infine, vogliamo dire che una persona di un certo tipo ha un nome, un indirizzo, un numero di telefono e un indirizzo email.
Ci sono due modi per realizzare lo schema relativo: o considerare i diversi tipi di persona come child dell'elemento radice, oppure definire un attributo "type" dell'elemento persona i cui valori potranno essere quelli dell'elenco precedente.
Nel primo caso lo schema risulterebbe molto verboso, mentre nel secondo caso lo schema diventa più leggibile e snello. Vediamo un esempio:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsi="http://www.w3.org/2001/XMLSchema">
 <xsd:element name=”person”>
  <complexType>
   <xsd:sequence>
    <xsd:element name=”name” type=”xsd:string”
     minOccurs=”1” minOccurs=”1”/>
    <xsd:element name=”address” type=”xsd:string”
     minOccurs=”1” maxOccurs=”1”/>
    <xsd:element name=”phone” type=”xsd:string”
     minOccurs=”1” maxOccurs=”unbounded”/>
    <xsd:element name=”email” type=”xsd:string”
     minOccurs=”1” maxOccurs=”unbounded”/>
   </xsd:sequence>
   <xsd:attribute name=”type” use=”optional” type=”xsd:string”/>
  </complexType>
 </xsd:element>
</xsd:schema>
```

L'attributo "use" dell'elemento `<attribute>` permette di specificare se un determinato attributo è opzionale o obbligatorio per l'elemento relativo.
Utilizzando l'elemento `<xsd:restriction>` è possibile elencare la lista dei valori che l'attributo può assumere. Nel nostro caso possiamo restringere la scelta ai valori *customer*, *employee*, *friend*, *family* in questo modo:

```
<xsd:attribute name=”type” use=”optional>
 <xsd:simpleType>
       <xsd:restriction base=”xsd:string”>
             <xsd:enumeration value=”customer”/>
```

```
            <xsd:enumeration value=”employee”/>
            <xsd:enumeration value=”friend”/>
            <xsd:enumeration value=”family”/>
        </xsd:restriction>
  </xsd:simpleType>
</xsd:attribute>
```

1.5. Complex Content

Un altro elemento molto importante in XML Schema è l’elemento `<xsd:complexContent>` che può essere utilizzato come child dell’elemento `<xsd:complexType>`. Questo elemento potrà contenere soltanto altri elementi e non solo del testo. In generale `complexContent` viene utilizzato per restringere o estendere il modello del contenuto di un `complexType`.
Diversamente, un simple content può contenere character data e attributi. Gli elementi `<xsd:restriction>` e `<xsd:extension>` sono utilizzati spesso come child di questo elemento**.**
Cerchiamo di capire bene il concetto che sta dietro questi due elementi.
L’elemento `restriction` (lo stesso che abbiamo visto nell’esempio all’interno del sottocapitolo dedicato al simple type) consente di limitare il tipo base di un elemento. Esso può avere in generale due attributi: l’attributo “id”, che rappresenta l’identificatore univoco, e l’attributo “base” che rappresenta l’elemento sul quale si desidera sia basata la restrizione. Il valore di quest’ultimo attributo che può avere o non avere un prefisso namespace: la presenza o meno del prefisso dipende se l’elemento si trova o non in un altro schema.
Con l’elemento `<xsd:extension>`, `al contrario`, è possibile creare un’estensione dell’elemento base. Per esempio, consideriamo il seguente elemento

```
    <routerid color="blue">121</routerid>
```

Nello schema XML potrebbe essere dichiarato un complexType di nome "shoesize", con un contenuto è definito come un data type integer e un attributo di nome “color” definito come una stringa.

```
<xsd:element name="routerid">
  <xsd:complexType>
    <xsd:simpleContent>
      <xsd:extension base="xsd:integer">
        <xsd:attribute name="color" type="xsd:string" />
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:element>
```

1.6. Altri componenti

In XML Schema esistono inoltre dei particolari elementi che specificano un determinato utilizzato degli elementi all’interno di un documento. Possiamo dividerli in *componenti di ordine*, *componenti di occorrenza* e *componenti di gruppo*.

Componenti di ordine

Gli indicatori di ordine servono ad indicare in che modo un elemento può apparire nell'istanza del documento.

`<choice>` specifica che in un elemento può apparire solo uno degli elementi figlio:

```
<xsd:element name="boyfriend">
  <xsd:complexType>
    <xsd:choice>
      <xsd:element name="John" type="xsd:string"/>
      <xsd:element name="Gavin" type="xsd:string"/>
    </xsd:choice>
  </xsd:complexType>
</xsd:element>
```

`<sequence>` specifica invece quali gli elementi figlio debbano apparire in un certo ordine:

```
<xs:element name="movie">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="title" type="xsd:string"/>
      <xs:element name="subtitle" type="xsd:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

`<all>`, infine, serve a specificare che l'elemento figlio può apparire in qualunque ordine e che lo stesso elemento può apparire una ed una sola volta:

```
<xsd:element name="book">
  <xsd:complexType>
    <xsd:all>
      <xsd:element name="editor" type="xsd:string"/>
      <xsd:element name="pubdate" type="xsd:string"/>
    </xsd:all>
  </xsd:complexType>
</xsd:element>
```

Componenti di occorrenza

Il controllo della occorrenze permette di stabilire quante volte un elemento può apparire nell'istanza del documento. Utilizzando DTD è possibile stabilire soltanto se un elemento può apparire zero volte, o più di zero o uno, ma non permette di imporre un numero preciso di ricorrenze.
Infatti con la DTD è impossibile, facciamo un esempio, imporre che un elemento `<list>` abbia un minimo di due elementi `<item>` e un massimo di quattro.
Con XML Schema, invece, questo è possibile farlo grazie ai due attributi "minOccurs" e "maxOccurs" dell'elemento `<xsd:element>`.
Supponiamo di voler realizzare uno schema per una rubrica. L'istanza del documento sarà la seguente:

```
<?xml version="1.0"?>
<Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="Rubrica.xsd">
  <Name>Mario Rossi</Name>
  <Address>via Aurelia, 127</Address>
  <Phone>3752615455</Phone>
  <Phone>3226643261</Phone>
</Person>
```

In questo caso vogliamo imporre che il numero minimo di elementi `<address>` sia 1 così come il numero massimo, mentre per quanto riguarda l'elemento `<phone>` si impone un minimo di 1 e un massimo di 4. Vediamo come questo verrà realizzato in XML Schema:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsi="http://www.w3.org/2001/XMLSchema">
 <xsd:element name=”Person”>
  <complexType>
   <xsd:sequence>
    <xsd:element name=”Name” type=”xsd:string”
     minOccurs=”1” minOccurs=”1”/>
    <xsd:element name=”Address” type=”xsd:string”
     minOccurs=”1” maxOccurs=”1”/>
    <xsd:element name=”Phone” type=”xsd:string”
     minOccurs=”1” maxOccurs=”4”/>
   </xsd:sequence>
  </complexType>
 </xsd:element>
</xsd:schema>
```

Se non si vuole imporre un numero massimo di elementi perchè non possiamo decidere a priori quante volte un elemento può comparire, e possibile assegnare all'attributo "maxOccurs" il valore "unbounded", che indicherà quindi un numero infinito.

Componenti di gruppo

Uno schema può nominare un gruppo di dichiarazioni di elementi che possono essere inseriti come definizioni di tipi complesso.

```
<xsd:group name="persongroup">
 <xsd:sequence>
  <xsd:element name="firstname" type="xsd:string"/>
  <xsd:element name="lastname" type="xsd:string"/>
  <xsd:element name="birthday" type="xsd:date"/>
 </xsd:sequence>
</xsd:group>
```

Dopo la definizione, è possibile riferirsi ad un altro gruppo o definizione di complex type:

```
<xsd:group name="persongroup">
 <xsd:sequence>
  <xsd:element name="firstname" type="xs:string"/>
  <xsd:element name="lastname" type="xs:string"/>
  <xsd:element name="birthday" type="xs:date"/>
 </xsd:sequence>
</xsd:group>
<xsd:element name="person" type="personinfo"/>
<xsd:complexType name="personinfo">
<xsd:sequence>
 <xsd:group ref="persongroup"/>
 <xsd:element name="country" type="xs:string"/>
</xsd:sequence>
</xsd:complexType>
```

Uno schema può inoltre nominare un gruppo di dichiarazioni di attributi in modo che possano essere incorporati come un gruppo in una definizione di un tipo complesso. Essi sono definiti con una dichiarazione come questa:

```
 <xsd:attributeGroup name="personattrgroup">
  <xsd:attribute name="firstname" type="xsd:string"/>
  <xsd:attribute name="lastname" type="xsd:string"/>
  <xsd:attribute name="birthday" type="xsd:date"/>
</xsd:attributeGroup>
```

Come nel caso dei gruppi di elementi, dopo la definizione è possibile riferirsi ad un altro gruppo o definizione di complex type:

```
 <xsd:attributeGroup name="personattrgroup">
   <xsd:attribute name="firstname" type="xsd:string"/>
   <xsd:attribute name="lastname" type="xsd:string"/>
   <xs:attribute name="birthday" type="xsd:date"/>
 </xsd:attributeGroup>
 <xsd:element name="person">
  <xsd:complexType>
    <xsd:attributeGroup ref="personattrgroup"/>
  </xsd:complexType>
 </xsd:element>
```

2. Best practices

2.1. Uso dei Namespaces

Problema

Uno dei primi passi nella realizzazione di uno schema è la scelta dell'utilizzo del namespace di defaul tra il targetNamespace o il namespace XML Schema (http://www.w3.org/2001/XMLSchema).

Introduzione

Eccetto gli schemi che non prevedono al loro intero dei namespace, ogni schema XML utilizza almeno due namespace: targetNamespace e il namespace XMLSchema (http://www.w3.org/2001/XMLSchema).
Cerchiamo di capire come bisogna scegliere il namespace di default quando si realizza uno schema.
Ci sono tre modi per realizzare lo schema Library supponendo di voler utilizzare i due namespace dell'esempio precedente:

Metodo1. Utilizzare come namespace di default XML Schema e qualificare esplicitamente con un prefisso tutti i componenti del targetNamespace.

Metodo 2. Vice versa – utilizzare come namespace di default il targetNamespace e qualificare esplicitamente con un prefisso tutti i componenti dal namespace XMLSchema.

Metodo 3. Non usare un namespace di default – qualificare esplicitamente con un prefisso sia i componenti del targetNamespace che quelli dell'XML Schema.

Vediamo ogni soluzione nel dettaglio. Consideriamo una situazione alcuni elementi dichiarati in uno schema possano essere riutilizzati da altri schemi:

Metodo 1

Di seguito riportiamo lo schema Library realizzato con questo primo metodo. Con l'elemento <include> è possibile includere all'interno dello schema Library un altro schema, in questo caso BookCatalogue, dove viene definito l'elemento Book. L'elemento <include> ha come attributo schemaLocation che permette di localizzare precisamente lo schema che si desidera includere. Lo schema Library fa riferimento ("ref"s) all'elemento Book.

```
<?xml version="1.0"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
          targetNamespace="http://www.library.org"
          xmlns:lib="http://www.library.org"
          elementFormDefault="qualified">
       <include schemaLocation="BookCatalogue.xsd"/>
```

```
   <element name="Library">
    <complexType>
               <sequence>
      <element name="BookCatalogue">
          <complexType>
                <sequence>
                  <element ref="lib:Book" maxOccurs="unbounded"/>
                </sequence>
             </complexType>
           </element>
          </sequence>
        </complexType>
     </element>
</schema>
```

Notare che il namespace di default è XML Schema, di conseguenza tutti gli elementi utilizzati per costruire lo schema - element, include, complexType, sequence, schema, etc – non devono avere nessun prefisso per essere qualificati.
Il prefisso namespace lib permette di far riferimento (utilizzando l'attributo "ref") a tutti i componenti del targetNamespace (Library, BookCatalogue, Book, etc). Quindi, tutti gli elementi del targetNamespace dovranno essere esplicitamanete qualificati utilizzando il prefisso lib (nell'esempio precedente troviamo un riferimento, ref, a lib:Book).

Vantaggi:

Se il tuo schema include componenti che derivano da più namespaces questo è il modo più consistente che ti permette di fare dei riferimenti ai vari componenti (cioè, si qualifica sempre il riferimento).

Svantaggi:

Possono esserci dei casi in cui lo schema che vogliamo realizzare non ha un targetNamespace.
In questi casi gli schemi che non hanno un targetNamespace devono essere realizzati in modo che i componenti XMLSchema (element, complexType, sequence, etc) siano qualificati. Se si utilizza questo approccio per progettare gli schema in alcuni di questi si dovranno qualificare i componenti XMLSchema e in altri no. Chiaramente il fatto di cambiare continuamente il modo di realizzare uno schema può non essere produttivo.

Metodo 2

In questo caso si utilizza come namespace di default il targetNamespace, quindi saranno tutti i componenti del namespace XML Schema a dover essere qualificati attraverso un prefisso:

```
     <?xml version="1.0"?>
     <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
          targetNamespace="http://www.library.org"
          xmlns="http://www.library.org"
          elementFormDefault="qualified">
       <xsd:include schemaLocation="BookCatalogue.xsd"/>
       <xsd:element name="Library">
     <xsd:complexType>
          <xsd:sequence>
           <xsd:element name="BookCatalogue">
```

```
            <xsd:complexType>
         <xsd:sequence>
          <xsd:element ref="Book" maxOccurs="unbounded"/>
         </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
</xsd:schema
```

Questo modo di realizzare lo schema fa sì che tutti i componenti utilizzati per costruire uno schema siano qualificati con il prefisso xsd:.
Infatti in questo caso il namespace di default è il targetNamespace, quindi tutti gli elementi che fanno parte di quel namespace non avranno bisogno di un prefisso per essere qualificati.
Vantaggi**:**
In questo modo tutti gli elementi del namespace XML Schema saranno sempre qualificati indipendentemente dalla presenza o meno del targetNamespace.
Svantaggi**:**

Utilizzando questo metodo, nel caso in cui siano stati importati o inseriti altri schemi, è molto semplice che nascano dei conflitti tra le varie dichiarazioni.

Metodo 3

Questo modo di realizzare lo schema prevede che non ci sia nessun namespace di default, quindi sia gli elementi appartenenti al namespace targetNamespace che quelli appartenenti al namespace XML Schema devono essere qualificati:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.library.org"
  xmlns:lib="http://www.library.org"
  elementFormDefault="qualified">
   <xsd:include schemaLocation="BookCatalogue.xsd"/>
   <xsd:element name="Library">
    <xsd:complexType>
     <xsd:sequence>
      <xsd:element name="BookCatalogue">
        <xsd:complexType>
         <xsd:sequence>
          <xsd:element ref="lib:Book" maxOccurs="unbounded"/>
         </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
     </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

Vantaggi

In questo modo, anche nel caso in cui siano importati o inseriti altri schemi, non potrà crearsi un conflitto tra i vari elementi che vengono dichiarati nello schema.

Svantaggi

Chiaramente il fatto che tutti gli elementi siano qualificati attraverso un prefisso può rendre illeggibile il documento, o quantomeno pesante.

Best Practice

Non c'è un best practice relativo a questo problema. È sicuramente una scelta personale.

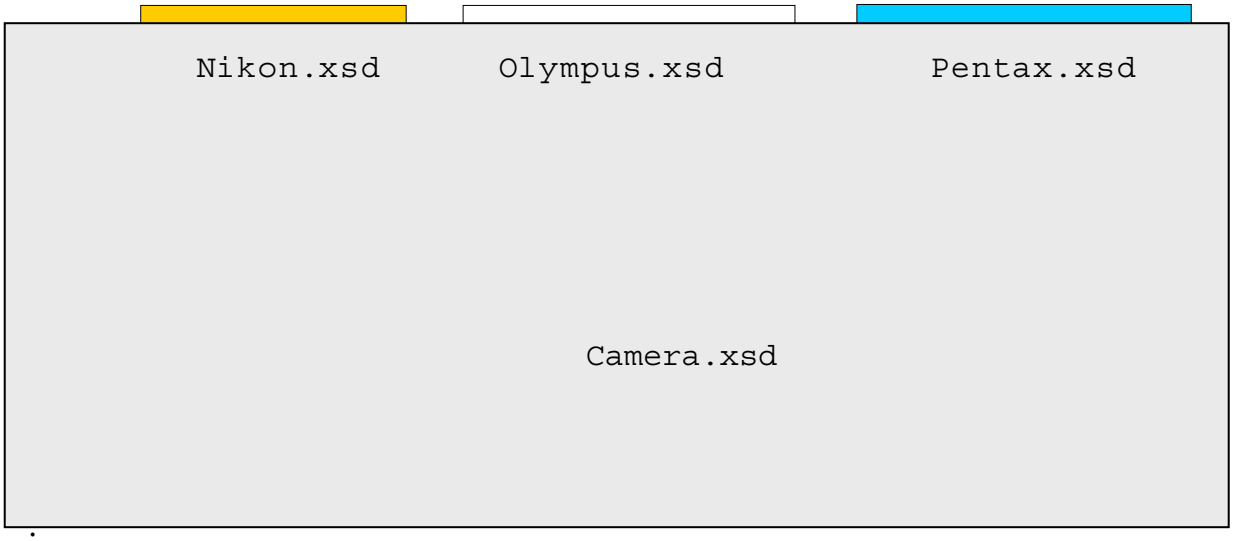
2.2. Nascondere o rende visibili i namespace

Problema

Una volta realizzato lo schema è possibile rendere visibili i namespaces anche nell'istanza oppure scegliere di non farlo. In questo paragrafo cercheremo di capire quali sono le situazioni in cui è meglio fare una cosa o l'altra.

Introduzione

È inoltre possibile importare o inserire uno schema all'interno di un altro schema. Questo implica inoltre che nello stesso schema si utilizzino più namespaces. C'è la possibilità di fare in modo che nell'istanza del documento sia visibile o meno questo fatto. Uno schema, inoltre, potrebbe includere elementi che derivano da più namespace. È possibile decidere se i vari namespaces siano visibili o meno nell'istanza del documento. Facciamo un esempio per capire bene quello che intendiamo dire quando si parla di rendere visibili o meno i namespace nell'istanza, ma anche per vedere come si realizza lo schema nei due casi.
Supponiamo di voler realizzare uno schema XML (camera.xsd) per descrivere una macchina fotografica. Di seguito riportiamo lo schema per descrviere una macchina fotografica, e che questo schema utilizzi elementi e tipi di elmenti che sono stati definiti in un altri schemi. Per esempio l'elemento <body> è di tipo nikon:body, che chiaramente come si capisce dal presfisso è stato definito nello schema nikon.xsd, così accade anche per altri elementi



.

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd=http://www.w3.org/2001/XMLSchema
targetNamespace="http://www.camera.org"
xmlns:nikon="http://www.nikon.com"
xmlns:olympus="http://www.olympus.com"
xmlns:pentax="http://www.pentax.com"
elementFormDefault="unqualified">
<xsd:import namespace="http://www.nikon.com"
                schemaLocation="Nikon.xsd"/>
  <xsd:import namespace="http://www.olympus.com"
      schemaLocation="Olympus.xsd"/>
  <xsd:import namespace="http://www.pentax.com"
     schemaLocation="Pentax.xsd"/>
  <xsd:element name="camera">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="body" type="nikon:body_type"/>
        <xsd:element name="lens" type="olympus:lens_type"/>
<xsd:elementname="manual_adapter"
type="pentax:manual_adapter_type"/>
     </xsd:sequence>
   </xsd:complexType>
</xsd:element>
</xsd:schema>
```

L’elemento <import> permette di importare gli elementi definiti in altri schemi (Nikon, Olympus e Pentax). L’attributo elementFormDefault di <schema> è stato settato al valore di unqualified e questo significa che gli elementi definiti in questo schema non dovranno essere qualificati attraverso un prefisso nell’istanza del documento. Di conseguenza i namespaces non saranno visibili nell’istanza stessa.

```
 <?xml version="1.0"?>
  <my:camera xmlns:my="http://www.camera.org"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.camera.org
                     Camera.xsd">
<body>
  <description>Ergonomically designed casing for easy handling
  </description>
</body>
<lens>
  <zoom>300mm</zoom>
  <f-stop>1.2</f-stop>
</lens>
<manual_adapter>
  <speed>1/10,000 sec to 100 sec</speed>
</manual_adapter>
</my:camera>
```

In questo esempio vediamo l'istanza del documento relativa allo schema precedente. Appare subito evidentemente che la persona che va a leggere questo documento non si accorgerà minimamente che alcuni elementi provengono da namespaces diversi da "http://www.camera.org". Soltanto l'elemento radice <camera> è qualificato tramite il prefisso.
Se, invece, il valore dell'attributo elementFormDefault è settato a "qualified" ogni elemento deve essere qualificato esplicitamente attraverso il prefisso e questo implica che i namespaces devono essere visibili nell'istanza.
Riportiamo l'istanza dell'esempio precednete in cui ogni elemento è qualificato attraverso il prefisso e abbiamo una dichiarazione per ogni namespace:

```
   <?xml version="1.0"?>
   <c:camera xmlns:c="http://www.camera.org"
    xmlns:nikon="http://www.nikon.com"
    xmlns:olympus="http://www.olympus.com"
    xmlns:pentax="http://www.pentax.com"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation=
   "http://www.camera.org
    Camera.xsd>
    <c:body>
    <  <nikon:description>Ergonomically  designed  casing  for  easy
   handling</nikon:description>
    </c:body>
    <c:lens>
    < <olympus:zoom>300mm</olympus:zoom>
    < <olympus:f-stop>1.2</olympus:f-stop>
    </c:lens>
    <c:manual_adapter>
    <    <pentax:speed>1/10,000 sec to 100 sec</pentax:speed>
    </c:manual_adapter>
   </c:camera>
```

In questo caso tutti gli schemi coinvolti devono avere un valore consistente dell'attributo elementFormDefault. Facciamo un esempio per capire che cosa succede quando gli schemi

coinvolti non hanno lo stesso valore per questo attributo. Consideriamo sempre lo schema Camera.xsd, in particolare supponiamo che gli schemi Camera.xsd e Olympus.xsd abbiano elementFormDefault=”unqualified”, mentra Nikon.xsd e Pentax.xsd abbiano elementFormDefault=”qualified”.
Di seguito riportiamo la relativa istanza:

```
<?xml version="1.0"?>
<my:camera xmlns:my="http://www.camera.org"
 xmlns:nikon="http://www.nikon.com"
 xmlns:pentax="http://www.pentax.com"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation=
"http://www.camera.org
 Camera.xsd>
<body>
<nikon:description>Ergonomically designed casing for easy
handling</nikon:description>
</body>
<lens>
 <zoom>300mm</zoom>
 <f-stop>1.2</f-stop>
</lens>
<manual_adapter>
<pentax:speed>1/10,000 sec to 100 sec</pentax:speed>
</manual_adapter>
</my:camera>
```

Dall’esempio sopra possiamo vedere che alcuni elementi sono qualificati attraverso un prefisso (pentax: e nikon:), mentra altri non hanno nessun prefisso.
Abbiamo già detto che se settiamo il valore dell’attributo elementFormDefault a “unqualified” nell’istanza del documento non saranno visibili i namespaces utilizzati. Per fare questo, però, dobbiamo anche fare in modo che l’elemento non sia dichiarato globalmente all’interno dello schema, cioè non deve essere child diretto della radice <xsd:schema>, come in questo caso:

```
<?xml version=“1.0”?>
<xsd:schema ...>
  <xsd:element name=“foo”>
   ...
 </xsd:schema ...>
```

L’elemento foo dovrà avere il namespace visibile nell’istanza del documento indipendentemente dal valore dell’attributo elementFormDefault. L’elemento foo è un elemento globale (cioè, figlio dell’elemento radice) e quindi dovrà essere sempre qualificato. Soltanto gli elementi non globali possono avere il proprio namespace non visibile nell’istanza del docuemnto.

Best Practice

Per questo tipo di problema non c'è un Best Practice ben preciso per realizzare lo schema. Ci sono dei casi in cui è meglio nascondere i namespaces mentre in altri è meglio renderli visibili. Di seguito riportiamo alcuni accorgimenti/consigli legati a questo problema:

Ogni volta che realizzi uno schema fanne due copie. In uno avrai l'attributo elementFormDefault="qualified" e nell'altro elementFormDefault="unqualified". Così le persone che useranno il tuo schema potranno decidere se rendere o meno visibili i namespaces nell'istanza del documento.
Minimizzare l'uso di attributi ed elementi globali, comunque questo aspetto verrrà analizzato in dettaglio più avanti.

Vantaggio nel non-rendere visibili i namespaces nell'istanza

L'istanza del documento è semplice. È facile da leggere e da capire. E questo fa sì che non sia necessario conoscere quali sono i namespaces utilizzati nello schema, altrimenti si andrebbe a complicare e quindi a rendere meno leggibile l'istanza.
Realizzare lo schema in modo che i namespaces non siano visibili nell'istanza del documento è consigliabile:

quando è più importante scrivere un'istanza semplice, leggibile e capibile e non sono necessarie le informazioni aggiuntive dei namespaces;
quando è necessario poter modificare lo schema senza che questo possa incidere sull'istanza.

Vantaggi di rendere visibili i Namespaces nell'Istanza del Documento

Può divenatre importante rendere visibili i namespaces nell'istanza quando si vuole ottenere un riconoscimento per aver realizzato un determinato schema.
In altri casi è consigliabile rendere visibili i namespaces quando si processa l'istanza del documento. Quindi si rende necessario conoscere il namespace per capire come deve essere processato un elemento. Se i namespaces sono localizzati nello schema si rende necessario fare una ricerca nello schema per ogni elemento e questo può rallentare molto il prcesso.
È consigliabile esporre i namespaces nell'istanza del documento

quando la provenienza degli elementi è importante per gli utenti (per esempio per scopi di copyright).
quando ci sono più elementi che hanno lo stesso nome ma un diverso significato. In questo caso può essere utile qualificare gli elementi per poterli differenziare (per esempio, publisher:body e human:body).
quando il processo dell'istanza da parte di un'applicazione dipende dalla conoscenza dei namespaces degli.

2.3. Global versus Local

Problema

Quando un elemento o tipo deve essere dichiarato globale e quando locale?

Introduzione

Una definizione si dice globale se è posta all'interno del tag <schema>. In questo caso l'elemento o l'attributo è definito in maniera assoluta. L'elemento può essere un elemento radice del documento.
Una definizione si dice locale se è inserita all'interno di un tag <complexType>. In questo caso l'elemento o l'attributo esiste solo se esiste un'istanza di quel tipo, e l'elemento non può essere un elemento radice del documento.
Cerchiamo di capire qual è la soluzione migliore, prendendo in considerazione questa parte di istanza:

```
<Book>
 <Title>Illusions</Title>
 <Author>Richard Bach</Author>
</Book>
```

Vediamo alcuni approcci per realizzare lo schema relativo a questa parte di istanza.

Russian Doll Design

Questo stile di progettazione rispecchia la struttura dell'istanza del documento.

```
<xsd:element name="Book">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="Title" type="xsd:string"/>
      <xsd:element name="Author" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</element>
```

In questo modo gli elementi sono dichiarati e contemporaneamente si stabilisce una gerarchia tra di essi.

Salami Slice Design

La caratteristica di questa soluzione è quella di dichiarare tanti elementi e successivamente stabilire la gerarchia tra di essi. Questo viene reso possibile dall'utilizzo dell'attributo "ref", come possiamo vedere nell'esempio:

```
<xsd:element name="Title" type="xsd:string"/>
<xsd:element name="Author" type="xsd:string"/>
<xsd:element name="Book">
  <xsd:complexType>
    <xsd:sequence>
```

```
        <xsd:element ref="Title"/>
        <xsd:element ref="Author"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
```

gli elementi Title e Author vengono dichiarati e successivamente una volta che andiamo a definire l'elemento Book attraverso l'attributo "ref" di <xsd:element> stabiliamo che I due elementi definiti precedentemente sono child dell'elemento Book.
Per capire meglio questi due metodi per realizzare uno schema, cerchiamo di pensare in termini di "scatole", dove una "scatola" rappresenta un elemento o un tipo.

Il Russian Doll design equivale ad avere una singola "scatola" che contiene al proprio interno altre scatole come le classiche Matrioske.
Il Salami Slice design invece equivale ad avere tante scatole separate che possono essere assemblate successivamente.

Esaminiamo le cararreristiche di questi due metodi di lavoro.

Russian Doll Design

Il contenuto di Book è nascosto agli altri schemi o ad altre parti dello stesso schema. Gli elementi ed i tipi non sono riutilizzabili.

La regione dello schema dove la dichiarazione di Title e Author sono applicabili è localizzata dentro l'elemento Book. Quindi se lo schema ha come attributo elementFormDefault="unqualified" i namespaces di Title e Author sono nascosti (localized) dentro lo schema.

Tutto è contenuto in una semplice, singola unità.

Ogni componente è self-contained (non interagisce con altri componenti). Quindi cambiare un componente ha un impatto limitato.

Salami Slice Design Characteristics

Tutti gli elementi e i tipi child di Book sono visibili agli altri schemi e ad altre parti dello stesso schema. In questo modo tutti i tipi e gli elementi che sono dentro Book sono riusabili.

Tutti i componenti sono globali. Quindi, indipendentemente dal volore dell'attributo elementFormDefault i namespaces di Title e Author saranno visibili nell'istanza del documento.

Ogni cosa è chiaramente visibile.

Nell'esempio che abbiamo fatto è facile intuire che l'elemento Book dipende dagli elementi Title e Author, quindi nel momento in cui uno dei due viene modificato anche l'elemento Book ne risente. Questo modo di realizzare lo schema comporta che ci siano dei legami tra i vari elementi.

Con questo approccio di progettazione tutti i dati correlati sono inseriti in gruppi di componenti auto-descrittivi. In questo modo i componenti sono coesivi.

Questi due modi di realizzare lo schema differiscono tra di loro per due motivi principali:

Il primo metodo (Russian Doll) non permette che i namespaces siano visibili all'interno del dell'istanza e questo va ad aumetnare la complessità dello schema, mentre nel secondo caso possiamo rendere visibili i namespaces.

Il metodo Salami Slice permette il riuso dei componenti, mentra l'altro no.

A questo punto viene spontaneo chiedersi se esiste un modo per nascondere i namespaces e allo stesso tempo permettere il riuso dei componenti. Un modo esiste.
Consideriamo di nuovo l'esempio dell'elemento Book. Un'alternativa può essere quella di creare la definizione di un tipo globale che ha il suo interno la dichiarazione degli elementi Title e Author, e poi dichiarare l'elemento Book di tipo "Publication":

```
<xsd:complexType name="Publication">
  <xsd:sequence>
    <xsd:element name="Title" type="xsd:string"/>
    <xsd:element name="Author" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:element name="Book" type="Publication"/>
```

Così riusciamo a raggiungere il nostro obiettivo, e cioè:

- a nascondere il namespace di Title a Author e

- ad avere un tipo Publication riusabile.

Venetian Blind Design

Con questo metodo riusciamo a realizzare quello che riuscivamo a fare con il secondo metodo. Però, invece di dichiarare gli elementi e poi utilizzare l'attributo "ref" si definisconi i tipi. Vediamo un esempio:

```
<xsd:simpleType name="Title">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Mr."/>
    <xsd:enumeration value="Mrs."/>
    <xsd:enumeration value="Dr."/>
  </xsd:restriction>
</xsd:simpleType>
<xsd:simpleType name="Name">
  <xsd:restriction base="xsd:string">
    <xsd:minLength value="1"/>
  </xsd:restriction>
</xsd:simpleType>
<xsd:complexType name="Publication">
  <xsd:sequence>
    <xsd:element name="Title" type="Title"/>
    <xsd:element name="Author" type="Name"/>
  </xsd:sequence>
```

```
</xsd:complexType>
<xsd:element name=“Book” type=“Publication”/>
```

Così siamo riusciti ad:
avere quattro componenti riusabili – il tipo Title, il tipo Name, il tipo Publication, e l’elemento Book e contemporaneamente

la possibilità di poter nascondere i namespaces. Chiaramente questo dipenderà dal valore dell’attributo dell’ ElementFormDefault

Gli obiettivi che si prefigge questo metodo per realizzare lo schema sono i seguenti:

- possibilità di non rendere visibili i namespaces nell’istanza del documento, semplicemente andando a modificare il valore dell’attributo elementFormDefault a seconda delle esigenze;
- avere degli elementi e dei tipi riusabili;
- annidare le dichiarazioni degli elementi all’interno delle definizioni dei tipi.

A questo punto facciamo un confronto tra l’ultimo metodo di realizzazione presentato e il precedente:

Salami Slice Design:

```
<xsd:element name=“Title" type=“xsd:string"/>
<xsd:element name=“Author" type=“xsd:string"/>
<xsd:element name=“Book">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref=“Title"/>
      <xsd:element ref=“Author" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Questo stile di progetto permette quindi di creare componenti riusabili, però non permette di non rendere visibili i namespaces.
Però ci si potrebbe chiedere:”Se però voglio realizzare uno schema ed avere i namespaces visibili nell’istanza del documento, questa è una buona soluzione?”
Potrebbe essere una buona soluzione fino al momento in cui non si rende necessario nascondere i namespaces. A questo punto si dovrebbe riscrivere lo schema.
È sicuramente meglio utilizzare la soluzione Venetian Blind che permette di gestire la visibilità dei namespaces semplicemente modificando il valore dell’attributo ElementFormDefault, senza chiaramente dover riscrivere completamente lo schema.

Caratteristiche del Venetian Blind Design:

1. Massima riusabilità;
2. potenzialità di nascondere i namespaces o meno semplicemente andando a modificare il valore dell’attributo elementFormDefault;

3. gli elementi che vengono dichiarati all'interno di uno schema realizzato con questo metodo sono interconnessi l'uno all'altro.
4. come negli altri casi, tutti i dati correlati sono inseriti in gruppi in componenti auto-descrittivi. In questo modo i componenti sono coesivi.

Best Practice

Il metodo Venetian Blind diventa la scelta più adatta quando si vuole scrivere uno schema in cui ci sia la flessibilità di decidere se i namespaces si vogliono rendere visibili o meno nell'istanza. Senza, chiaramente, ogni volta dover modificare completamente lo schema stesso;

mentre quando chi scrive lo schema vuole permettere agli autori dell'istanza di usare l'elemento substitution è migliore l'approccio Salami Slice;

infine, quando l'obiettivo principale è quello che lo schema sia piccolo e non ci sia una dipendenza tra i vari elementi allora la scelta ricade sul metodo Russian Doll.

2.4. Elementi o Tipi?

Problema

Quando un item dovrebbe essere dichiarato come elemento e quando invece dovrebbe essere definito come tipo?

Introduzione

Questo problema si discute bene con un esempio:

Esempio

"Warranty" dichiarato come elemento:

```
<xsd:element name="Warranty">
  ...
</xsd:element>
```

o come tipo:

```
<xsd:complexType name="Warranty">
        ...
</xsd:complexType>
```

Best Practice

Nel dubbio, è consigliabile dichiararlo come tipo. Questo perché è molto semplice creare un elemento partendo dal tipo che abbiamo definito precedentemente. In questo modo altri elementi possono riusare questo tipo.
Supponiamo di scrivere uno schema in cui inizilamente definiamo Warranty come tipo:

```
<xsd:complexType name="Warranty">
   ...
</xsd:complexType>
```

e poi in un secondo tempo nasce l'esigenza di dover dichiarare l'elemento Warranty. Diventa tutto molto semplice perché basta dichiarare un elemento con nome Warranty e di tipo Warranty:

```
<xsd:element name="Warranty" type="Warranty"/>
```

Se l'item deve essere un elemento nell'istanza allora va dichirato come elemento, altrimenti possiamo definirlo come comlexType. Questo vale anche quando il contenuto dell'item deve essere riusabile. Supponiamo che nel nostro caso ci siano degli elementi che hanno necessità di usare il contenuto di Warranty allora è consigliabile definirlo come complexType:

```
<xsd:complexType name="Warranty">
   ...
</xsd:complexType>
...
<xsd:element name="PromissoryNote" type="Warranty"/>
<xsd:element name="AutoCertificate" type="Warranty"/>
```

L'esempio mostra che i due elementi PromissoryNote e AutoCertificate riusano il tipo Warranty.
Adesso, invece consideriamo il caso in cui l'item debba essere usato come elemento nell'istanza e contemporaneamente debba essere a volte nullo e a volte no.
Proviamo a creare l'elemento Warranty:

```
<xsd:element name="Warranty">
  ...
</xsd:element>
```

L'elemento Warranty può essere *riusato* tramite un riferimento

```
<xsd:element ref="Warranty"/>
```

Supponiamo di aver bisogno di una versione dell'elemento che supporti il valore *nil*, istinitvamente potrebbe nascere l'idea di farlo in questo modo:

```
<xsd:element ref="Warranty" nillable="true"/>
```

Questo non è valido. Questa possibilità dinamica di morphing (cioè di riutilizzare una dichiarazione di elemento aggiungendo l'attributo nillable) non può essere realizzata utilizzando gli elementi. Questo perché il riferimento e gli attributi nillable sono mutuamente esclusivi. Potete usare il riferimento o potete usare nillable, ma non entrambi.
Si può realizzare il morphing dinamico solo definendo la Warranty come tipo:

```
<xsd:complexType name="Warranty">
...
</xsd:complexType>
```

E riutilizzando il tipo:

```
<xsd:element name="Warranty" nillable="true" type="Warranty"/>
...
<xsd:element name="Warranty" type="Warranty"/>
```

Se l'item è pensato per essere usato come elemento nelle istanze di documenti ed altri elementi devono poter sostituire l'elemento, in questo caso deve essere dichiarato come elemento.
Supponiamo di voler realizzare lo schema in modo che i vocaboli Warranty, Guarantee o Promise (come nome dei tag) siano interscambiabili nell'istanza, cioè:

```
<xsd:Warranty>
   ...
</xsd:Warranty>
   ...
<xsd:Guarantee>
   ...
</xsd:Guarantee>
   ...
<xsd:Promise>
     ...
</xsd:Promise>
```

Per abilitare questa capacità di sostituire i nomi dei tag, Warranty, Guarantee e Promise devono essere dichiarati come elementi e assegnati ad un substitutionGroup:

```
<xsd:element name="Warranty">
   ...
</xsd:element>
<xsd:element name="Guarantee" substitutionGroup="Warranty"/>
<xsd:element name="Promise" substitutionGroup="Warranty"/>
```

2.5. Zero, uno o più Namespaces

In questa parte affrontiamo il problema di decidere se non utilizzare i namespaces oppure se utilizzarne solo uno oppure utilizzare più namespaces. Per capire al meglio questo problema mettiamoci nel caso in cui si desideri progettare uno schema XML per documenti che hanno una struttra molto complessa. In questi casi si usa in genere scomporre il documento in più parti che vengono definite ciascuna in modo indipendente. Successivamente si esegue un merger per ottenere lo schema generale.
In questo ambito vengono discusse tre possibili soluzioni per quanto riguarda i namespaces da utilizzare:

Heterogeneous Namespace Design: associa un diverso targetNamespace ad ogni schema

Homogeneous Namespace Design: associa lo stesso targetNamespace a tutti gli schemi

Chameleon Namespace Design: associa un targetNamespace allo schema principale mentre nessun targetNamespace agli altri schemi "di supporto.

Ognuna di queste tecniche presenta sia vantaggi che svantaggi e la scelta di quale utilizzare dipende dalle caratteristiche delle varie parti di cui è composto il documento globale. L'uso della soluzione Heterogeneous viene racoomandata quando si usa uno schema in cui si esegue l'import di altri schema definiti da altri. In particolare viene consigliato di usare <import> e di non copiare lo schema all'interno di quello globale, in modo da avere lo schema globale automaticamente aggiornato.
Quando tutti i componenti dello schema vengono definiti dalla stessa persona o da uno stesso gruppo i modelli Heterogeneous e più in particolare Chameleon risultano più potenti e flessibili.

L'uso della soluzione Chamaleon Namespace è consigliata quando:
gli schemi includono componenti che non sono legati tra di loro a livello semantico;
gli schemi contengono dei componenti che hanno un legame semantico soltanto nel contesto dello schema che include tutti gli altri;
when you don't want to hardcode a namespace to a schema, rather you want <include>ing schemas to be able to provide their own application-specific namespace to the schema

L'uso della soluzione Homogeneous Namespace è consigliata quando:
• tutti gli schemi sono concettualmente legati,
• non è necessario identificare nell'istanza del documento l'origine di ogni elemento e attributo. In questo modo non è possibile capire dall'istanza in quale schema è stato definito un determinato elemento, visto che tutti i componenti provengono dallo stesso namespace.

L'uso della soluzione Heterogeneous Namespace è consigliata quando:
• sono presenti più elementi con lo stesso nome. (Questo permette di evitare la collisione di nomi)
• è necessario poter identificare dall'istanza del documento l'origine di ogni elemento e attributo.

Collisione di nomi.

Nei casi Homogeneous e Chameleon si possono verificare collisioni tra i nomi, cioè quando si tenta di definire due elementi con lo stesso nome in due schemi diversi.
Il problema delle collisioni può essere affrontato mediante l'uso di due tecniche:
Uso dell'elemento <redefine>.
Proxy Schemas.

Elemento <redefine>

Questo elemento può essere usato solo nei casi in cui si utilizzi come soluzione Homogeneous Namespace o Chamaleon Namespace. L'elemento permette di accedere ai componenti in altri schemi e contemporaneamente dà la possibilità di modificare zero o più componenti. Quindi ha una doppia funzionalità:
Implicitamente è come se si utilizzasse l'elemento <include>, quindi permette di accedere a tutti i componenti dello schema riferito;
Consente di estendere o restringere i componenti nello schema riferito

Proviamo a capire meglio l'uso di questo elemento con un esempio, consideriamo i seguenti schemi:

Product.xsd

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespac e="http://www.company.org"
  xmlns="http://www.product.org"
  elementFormDefault="qualified">
    <xsd:complexType name="ProductType">
 <xsd:sequence>
    <xsd:element     name="Type"     type="xsd:string"     minOccurs="1"
maxOccurs="1"/>
 </xsd:sequence>
    </xsd:complexType >
</xsd:schema>
```

Person.xsd

```
<?xml version="1.0"?>
      <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                  targetNamespace="http://www.company.org"
   xmlns="http://www.person.org"
   elementFormDefault="qualified">
        <xsd:complexType name="PersonType">
          <xsd:sequence>
             <xsd:element name="Name" type="xsd:string"/>
<xsd:element name="SSN" type="xsd:string"/>
          </xsd:sequence>
       </xsd:complexType>
      </xsd:schema>
```

Company.xsd

```
<?xml version="1.0"?>
 <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                  targetNamespace="http://www.company.org"
                  xmlns="http://www.company.org"
                  eleme ntFormDefault="qua lified">
         <xsd:include schemaLocation="Person.xsd"/>
        <xsd:include schemaLocation="Product.xsd"/>
        <xsd:element name="Company">
         <xsd:complexType>
           <xsd:sequence>
 <xsd:element name ="Person" type="PersonType"   maxOccurs="unbounded"/>
 <xsd:element name ="Product" type="ProductType" maxOccurs="unbounded"/>
           </xsd:se quence>
         </xsd:complexType>
       </xsd:element>
      </xsd:schema>
```

Supponiamo che si desideri utilizzare ProductType nello schema Product.xsd e che tale elemento vogliamo estenderlo per includere un elemento child ID. Di seguito riportiamo un esempio di come possiamo realizzare questo usando l'elemento <redefine>:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
         targetNamespace="http://www.company.org"
         xmlns="http://www.compa ny.org"
         elementFormDefault="qualified">
 <xsd:include schemaLocation="Person.xsd"/>
 <xsd:redefine schemaLoc ation="Product.xsd">
     <xsd:complexType name="ProductType">
         <xsd:complexContent>
   <xsd:extension base="ProductType">
       <xsd:sequence>
<xsd:element name ="ID" type="xsd:ID"/>
       </xsd:se quence>
   </xsd:extension>
         </xsd:comple xContent>
     </xsd:complexType>
 </xsd:redefine>
 <xsd:element name ="Company">
     <xsd:complexType>
         <xsd:sequence >
<xsd:element name="Person" type="PersonType"  maxOccurs="unbounded"/>
<xsd:element name="Product" type="ProductType"  maxOccurs="unbounded"/>
         </xsd:sequence>
     </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

In questo modo l'elemento <Product> nell'istanza del documento includerà l'elemento `<Type>` e `<ID>`, inoltre vediamo che abbiamo con l'utilizzo dell'elemento `<redefine>` abbiamo potuto definire un complexType con lo stesso nome del complexType che vogliamo estendere.

Proxy Schema

Una possibile soluzione al problema della collisione dei nomi è quella di creare per ogni schema che non ha namespace uno schema "associato" ("proxy schema") che include (<include>) lo schema che non ha namespace. Successivamente sarà lo schema principale che importa (<import>) lo schema proxy.
Supponiamo, infatti di avere due schemi, Q.xsd e R.xsd che non hanno namespace, e quindi di creare i relativi Proxy Schema, Q-Proxy.xsd e R-Proxy.xsd. Quindi avremo:

Q-Proxy.xsd

```
<xsd:schema …
 targetNamespace="Z1">
  <xsd:include schemaLocation="Q.xsd"/>
```

R-Proxy.xsd

```
<xsd:schema …
 targetNamespace="Z2">
  <xsd:include schemaLocation="r.xsd"/>
```

Lo schema principale importerà i due schemi precedenti:

Z.xsd

```
<xsd:schema …    targetNamespace="Z">
 <xsd:import namespace="Z1" schemaLocation="Q-Proxy.xsd"/>
 <xsd:import namespace="Z2" schemaLocation="R-Proxy.xsd"/>
…
</xsd:schema>
```

Best Practice

Adesso cerchiamo di capire qual è la soluzione migliore per realizzare uno schema per risolvere il problema che abbiamo esposto in questo paragrafo.
In generale, quando si ri-utilizzano schemi che altri hanno realizzato è consigliabile importare quelli schemi (<import>), e quindi utilizzare la soluzione Heterogeneous Namespace. Non è assolutamente una buona idea copiare i componenti di questi schemi all'inerno del nuovo nalespace per due semplici ragioni:
le copie locali non saranno sincronizzate con gli altri schemi;
si perde l'interoperabilità con le applicazioni esistenti che processano i componenti degli altri schemi.
Il caso che ci interessa analizzare più in dettaglio è quello di capire come trattare i namespaces in un insieme di schemi che si realizzano. Vediamo per ogni soluzione che abbiamo proposto quali sono gli aspetti positivi e quelli negativi.

È consigliabile utilizzare la soluzione **Chamaleon** quando:
gli schemi che contengono i componenti non hanno nessun legame semantico tra di loro;
gli schemi che contengono i componenti hanno una relazione semantica con lo schema che li include;
si vuol fare in modo che il namespace non sia specificato all'interno dello schema principale ma si vuole che lo schema principale utilizzi i namespaces degli schemi inclusi!
La soluzione **Homogenous Namespace** è consigliabile utilizzarla quando:
tutti i componenti dei vari schemi in gioco sono legati concettualmente;
non c'è la necessità di rendere visibile all'interno dell'istanza l'origine di ogni elemento/attributo. In pratica, tutti i componenti provengono dallo stesso namespace quindi non è possibile sapere semplicemente dall'istanza del documento da quale schema proviene un determinato elemento/attributo.
Infine si preferisce scegliere la soluzione **Heterogeneous Namespace** quando:
ci sono più elementi con lo stesso nome e quindi questo è un modo per evitare la collisione tra nomi;
c'è la necessità di poter identificare nell'istanza l'origine di ogni elemento/attributo.

2.6. Creazione di un elemento che ha contenuto variabile

Problema

Creazione di un elemento che ha un contenuto variabile

Introduzione

Cerchiamo di capire meglio il problema con un esempio: l'elemento <Catalogue> è l'elemento che può includere contenuto variabile come gli elementi <Book> e <Magazine>:
Dobbiamo porci alcune domande:

- Possiamo permettere che un elemento contenga elementi dissimili, indipendenti o più in generale non strettamente legati tra di loro?
- Come possiamo creare un elemento il cui contenuto è variabile ma che sia anche estendibile?

Esempio

Consideriamo un elemento il cui è contenuto è costituito da una collezione di elementi, dove ogni elemento è variabile.
Consideriamo l'esempio dell'elemento <Catalogue> i cui elementi child sono <Book> e <Magazine> che sono di tipi diverso:

```
    <Catalogue>
        <Book> ... </Book>
        <Magazine> ... </Magazine>
        <Book> ... </Book>
      </Catalogue>
```

Descriviamo quattro modi per definire questo elemento.

Metodo 1: utilizzo di un elemento astratto e di un elemento subsitution

Prima di andare ad analizzare questo metodo ci sono alcuni concetti che devono essere ben chiari!

un elemento può essere dichiarato astratto.

gli elementi astratti non possono essere instanziati nell'istanza.

nell'istanza del documento l'elemento astratto deve essere sostituito da elementi concreti
l'elemento astratto e tutti gli elementi nel suo substitutionGroup devono essere dichiarati come elementi globali.

Implementazione:
Dichiarare un elemento astratto (Publication):

```
<xsd:element name="Publication" abstract="true"
       type="PublicationType"/>
```

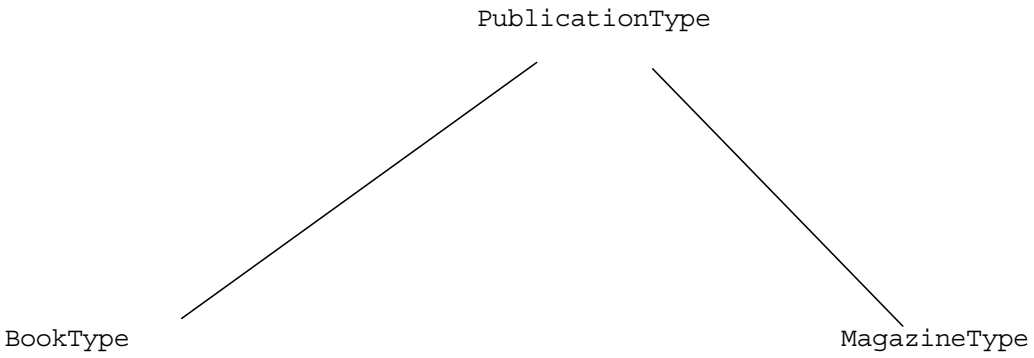
Dichiarare un elemento a contenuto variabile (Catalogue) che ha come contenuto l'elemento astratto appena definito (facciamo un riferimento, "ref", alla dichiarazione dell'elemento astratto):

```
<xsd:element name= "Catalogue">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="Publication"
             maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Notare che maxOccurs="unbounded", in questo modo l'elemento Catalogue può contenere uno o più elementi Publication.
Dichiarare gli elementi concreti (Book and Magazine) che sono inclusi nell'elemento contenitore e dichiararli in modo che possano essere nel substitutionGroup con l'elemento astratto:

```
<xsd:element name="Book" substitutionGroup="Publication"
       type="BookType"/>
<xsd:element name="Magazine" substitutionGroup="Publication"
       type="MagazineType"/>
```

Per fare in modo che Book e Magazine possano sostituire Publication, il loto tipi (BookType e MagazineType) devono derivare da PublicationType.



Vediamo, quindi, come si definiscono i tipi:

PublicationType – il tipo base:

```
<xsd:complexType name="PublicationType">
  <xsd:sequence>
    <xsd:element name="Title" type="xsd:string"/>
    <xsd:element name="Author" type="xsd:string"
           minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="Date" type="xsd:gYear"/>
  </xsd:sequence>
</xsd:complexType>
```

BookType – estensione del tipo Publication: sono aggiunti i due elemnti ISBN e Publisher:

```
<xsd:complexType name="BookType">
  <xsd:complexContent>
    <xsd:extension base="PublicationType">
      <xsd:sequence>
        <xsd:element name="ISBN" type="xsd:string"/>
        <xsd:element name="Publisher" type="xsd:string"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

MagazineType – restrizione del tipo Publication, viene eliminato l'elemento Author:

```
<xsd:complexType name="MagazineType">
  <xsd:complexContent>
    <xsd:restriction base="PublicationType">
      <xsd:sequence>
        <xsd:element name="Title" type="xsd:string"/>
        <xsd:element name="Author" type="xsd:string"
               minOccurs="0" maxOccurs="0"/>
        <xsd:element name="Date" type="xsd:gYear"/>
      </xsd:sequence>
    </xsd:restriction>
  </xsd:complexContent>
```

```
</xsd:complexType>
```

Di seguito riportiamo un esempio di istanza che fa riferimento ad uno schema in cui sono presenti le definizioni appena viste:

```
<?xml version="1.0"?>
        <Catalogue xmlns="http://www.catalogue.org"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.catalogue.org
       Catalogue.xsd">
  <Book>
<Title>Illusions The Adventures of a Reluctant   Messiah</Title>
    <Author>Richard Bach</Author>
    <Date>1977</Date>
    <ISBN>0-440-34319-4</ISBN>
    <Publisher>Dell Publishing Co.</Publisher>
  </Book>
  <Magazine>
    <Title>Natural Health</Title>
    <Date>1999</Date>
  </Magazine>
  <Book>
    <Title>The First and Last Freedom</Title>
    <Author>J. Krishnamurti</Author>
    <Date>1954</Date>
    <ISBN>0-06-064831-7</ISBN>
    <Publisher>Harper &amp;amp; Row</Publisher>
  </Book>
  </Catalogue>
```

Vantaggi:

- **Estensibilità:** in questo modo è possibile inserire altri elementi nell'elemento che ha contenuto variabile, che nel nostro caso è <Catalogue>, anche se non si può controllare direttamente lo schema in cui abbiamo definito l'elemento Catalogue. Infatti, supponiamo che non sia possibile modificare lo schema Catalogue. Attualmente l'elemento Catalogue può contenere soltanto gli elementi Book e Magazine. Ma supponiamo che si desideri inserire anche l'elemento <CD>, come vediamo nell'esempio:

```
    <Catalogue>
        <Book> ... </Book>
        <CD> ... </CD>
        <Magazine> ... </Magazine>
        <Book> ... </Book>
      </Catalogue>
```

Cerchiamo di capire con un esempio come sia possibile estendere l'elemento Catalogue senza modificare lo schema. Prima di tutto si deve creare uno schema separato che contiene la dichiarazione dell'elemento CD (con un tipo, CDType, che estende PublicationType definito nello schema Catalogue), e quindi fare in modo che l'elemento CD faccia parte del substitutionGroup Publication:

```
  <xsd:include schemaLocation="Catalogue.xsd"/>
```

```
<xsd:complexType name="CDType">
  <xsd:complexContent>
    <xsd:extension base="PublicationType">
      <xsd:sequence>
        <xsd:element name="RecordingCompany"
               type="xsd:string"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="CD" substitutionGroup="Publication"
       type="CDType"/>
```

Quindi gli elementi Book, Magazine e CD possono essere inseriti nell'elemento Catalogue, come possiamo vedere nell'esempio seguente:

```
<?xml version="1.0"?>
 <Catalogue xmlns="http://www.catalogue.org"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.catalogue.org
                           CD.xsd">
   <Book>
    <Title>Illusions The Adventures of a Reluc tant Messiah</Title>
    <Author>Richard Bach</Author>
    <Date>1977</Date>
    <ISBN>0-440-34319-4</ISBN>
    <Publisher>Dell Publishing Co.</Publisher>
  </Book>
  <CD>
    <Title>Timeless Serenity</Title>
    <Author>Dyveke Spino</Author>
    <Date>1984</Date>
    <RecordingCompany>Dyveke Spino Productions</RecordingCompany>
  </CD>
  ...
 </Catalogue>
```

- **Coesione Semantica**: tutti gli elementi, child dell'elemento a contenuto variabile, discendono gerarchicamente dallo stesso tipo (in questo caso PublicationType). Questo tipo li lega insieme gerarchicamente, dando in questo modo una coerenza strutturale , e quindi semantica, a tutti gli elementi che possono essere contenuti nell'elemento a contenuto variabile.

Svantaggi:

- **Nessun elemento indipendente:** gli elementi che possono far parte dell'elemento a contenuto variabile devono essere di un tipo derivato dal tipo dell'elemento astratto (nel nostro esempio PublicationType). Inoltre, gli elementi devono far parte del substitutionGroup insieme all'elemento astratto. Quindi l'elemento "contenitore" non

può includere i tipi che non derivano dal tipo dell'elemento astratto, o che non sono nel substitutionGroup con l'elemento. Per esempio, se in un altro schema viene definito l'elemento "Newspaper" il cui tipo, però, non deriva da PublicationType, l'elemento <Catalogue> non può contenere l'elemento <Newspaper>.

- **Variabilità strutturale limitata:** uno schema può cambiare nel tempo, di conseguenza può aumentare il numero di elementi che un elemento può includere. I nuovi elementi potrebbero essere concettualmente collegati agli elementi già presenti, ma non strutturalmente. Quindi il tipo base potrebbe essere giusto per gli elementi originali ma non per quelli nuovi che hanno una struttura diversa. Quindi ci si trova di fronte a un tradeoff: si crea un tipo base semplice che permetta di supportare molte strutture che differiscono tra di loro, oppure si crea un tipo base meno semplice che però non ci permette di inserire elementi che di tipo differente.
- **Elaborazione Non scalabile:** Consideriamo uno stylesheet che processa ogni elemento child dell'elemento <Catalogue>:

```
<xsl:if test="Book">
    -- process Book --
</xsl:if>
<xsl:if test="Magazine">
    -- process Magazine --
</xsl:if>
```

Questo frammento di stylesheet si blocca non appena viene aggiunto un nuovo elemento in <Catalogue>. Se gli elementi child di <Catalogue> sono elementi che sono sostituti dell'elemento astratto Publication, ogni elemento può essere processato uniformemente in questo modo:

```
<xsl:for-each select="Catalogue/*">
 -- process the element --
</xsl:for-each>
```

Così si processsa ogni elemento dentro <Catalogue> indipendentemente dal nome, e non ci sono problemi di scalabilità. Nascono dei problemi di scalabilità quando Catalogue contiene più elementi astratti:

```
<xsd:element name="Catalogue">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="Publication"
             maxOccurs="unbounded"/>
      <xsd:element ref="Retailer"
             maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Supponiamo che sia Publication che Retailer siano elementi astratti, e all'interno di Catalogue ci può essere un numero qualsiasi di elementi di ogni tipo. Vediamo una semplice istanza:

```

```

```
<Catalogue>
  <Book> ... </Book>
  <Magazine> ... </Magazine>
  <Book> ... </Book>
  <MarketBasket> ... </MarketBasket>
  <Macys> ... </Macys>
</Catalogue>
```

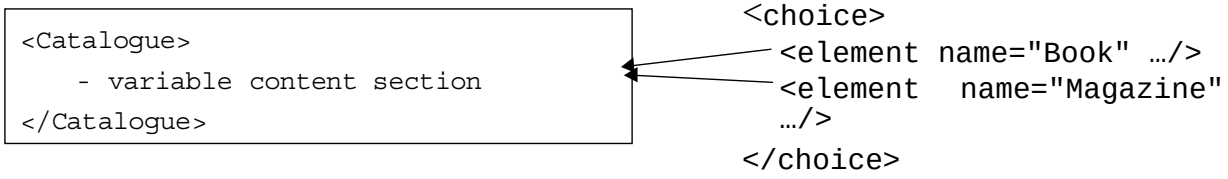
Per processare soltanto gli elementi Publication sarà necessario scrivere un codice speciale come abbiamo fatto per il primo caso, e questo non è scalabile. Ogni volta che viene inserito un nuovo elemento si deve aggiornare il codice e questo è molto pesante.

- **Nessun controllo sull'esposizione dei namespaces:** questo metodo richiede che tutti gli elementi che sono usati all'interno dell'elemento contenitore si trovino in un sobstitutionGroup con l'elemento astratto. Un vincolo nell'uso di substitionGroup è che tutti gli elementi devono essere dichiarati globalmente. I namespaces degli elementi globali non possono essere mai nascosti nell'istanza del documento. Di conseguenza non c'è nessun modo per poter nascondere i namespaces degli elementi utilizzati nel contenitore.

Metodo 2: utilizzo dell'elemento choice

Descrizione:

Questo metodo, semplicemente, elenca dentro l'elemento <choice> tutti gli elementi che possono apparire nell'elemento.



Implementazione:

Per prima cosa bisogna dichiarare dentro un elemento <choice> tutti glie elementi (come Book, Magazine) che possono essere usato all'interno dell'elemento "contenitore", <Catalogue> nel nostro caso:

```
<element name="Catalogue">
  <complexType>
    <choice maxOccurs="unbounded">
      <element name="Book" type="BookType"/>
      <element name="Magazine" type="MagazineType"/>
    </choice>
  </complexType>
</element>
```

Vantaggi:

Elementi indipendenti: gli elementi contenuti in Catalogue, che nel nostro caso è l'elemento contenitore, non hanno bisogno di avere il tipo in comune. Non devono essere legati in alcun modo.

Svantaggi:

Non è estensibile: consideriamo il caso in cui non sia possibile modificare lo schema Catalogue. Supponiamo che ci sia l'esigenza di inserire come elemento child di <Catalogue> l'elemento CD. Vediamo l'istanza in questo caso:

```
<Catalogue>
  <Book> ... </Book>
  <CD> ... </CD>
  <Magazine> ... </Magazine>
  <Book> ... </Book>
</Catalogue>
```

Questo metodo richiede necessariamente di poter modificare l'elemento <choice> nello schema in modo da poter includere anche l'elemento CD. Quindi nel caso in cui non sia possibile modificare lo schema questo metodo comporta delle restrizioni molto forti a livello di estensibilità.

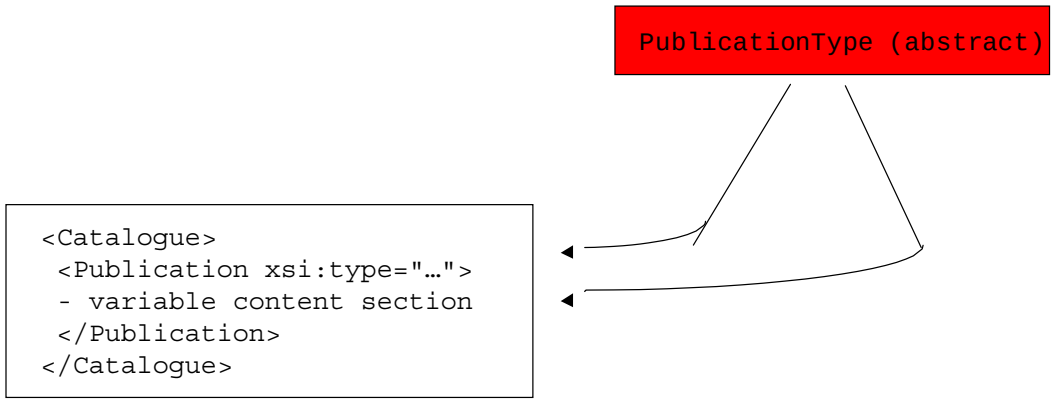
Nessuna coerenza semantica: l'elemento <choice> permette di raggruppare insieme elementi non simili. Anche se questo può essere un vantaggio contemporaneamente può essere visto come uno svantaggio. Infatti in questo modo non è possibile legare insieme gli elementi gerarchicamente, stabilire una struttura, etc quindi se si desidera processare un'istanza non è possibile fare alcuna assunzione sulla struttura degli elementi.

Metodo 3: utilizzo di un tipo astratto e di un tipo substitutionGroup

Descrizione:

Prima di analizzare questo metodo vediamo alcuni concetti importanti:

- Un complexType può essere dichiarato astratto;
- Un elemento che viene dichiarato per essere un tipo astratto non può avere il suo tipo istanziato nell'istanza;
- Nell'istanza un elemento di tipo astratto deve avere il suo contenuto sostituito da un contenuto di tipo non astratto che derivi dal tipo. Questo è chiamato tipo substitution.



BookType MagazineType

Implementazione:

Definire un tipo di base astratto (PublicationType)

```
<xsd:complexType name="PublicationType" abstract="true">
  <xsd:sequence>
    <xsd:element name="Title" type="xsd:string"/>
    <xsd:element name="Author" type="xsd:string"
           minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="Date" type="xsd:gYear"/>
  </xsd:sequence>
</xsd:complexType>
```

Dichiarare un elemento "contenitore" (Catalogue) per contenere un elemento(Publication) di tipo astratto:

```
<xsd:element name="Catalogue">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="Publication"

              type="PublicationType"
              maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
```

```
</xsd:element>
```

Come abbiamo già detto, nell'istanza il contenuto di Publication può essere soltanto di un tipo concreto che derivi da PublicationType come abbiamo visto per il metodo 1.
With this method instance documents will look different than we saw with the above two methods. Nominamente, con questo metodo l'elemento <Catalogue> non avrà contenuto variabile ma conterrà sempre lo stesso elemento (Publication). Namely, <Catalogue> will not contain variable content. Instead, it will always contain the same element (Publication).

```
<?xml version="1.0"?>
 <Catalogue xmlns="http://www.catalogue.org"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.catalogue.org
Catalogue.xsd">
 <Publication xsi:type="BookType">
<Title>Illusions The Adventures of a Reluctant Messiah</Title>
<Author>Richard Bach</Author>
<Date>1977</Date>
<ISBN>0-440-34319-4</ISBN>
<Publisher>Dell Publishing Co.</Publisher>
 </Publication>
 <Publication xsi:type="MagazineType">
<Title>Natural Health</Title>
<Date>1999</Date>
 </Publication>
 ...
</Catalogue>
```

Vantaggi:

Estensibile: questo metodo permettere di estendere facilmente l'insieme degli elementi che possono essere usati come contenuto dell'elemento "contenitore" creando due nuovi tipi che derivano da quello astratto:

```
<include schemaLocation="Catalogue.xsd"/>
<complexType name="CDType">
   <complexContent>
     <extension base="PublicationType" >
      <sequence>
        <element name="RecordingCompany" type="string"/>
      </sequence>
     </extension>
   </complexContent>
</complexType>
```

Quindi si può estendere lo schema Catalogue senza modificarlo, e questo è possibile andando a creare un nuovo schema in cui includiamo lo schema principale. Di seguito riportiamo un esempio dell'istanza con il nuovo elemento <CD>:

```
<?xml version="1.0"?>
<Cata logue xmlns="http://www.catalogue.org"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation=
     "http://www.catalogue.org
     CD.xsd">
   <Publication xsi:type="BookType">
   <Title >Illusions The Adventures of a Reluctant Messiah</Title>
   <Author>Richard Bach</Author>
   <Date>1977</Date>
   <ISBN>0-440-34319-4</ISBN>
   <Publisher>Dell Publishing Co.</Publisher>
   </Publication>
<Publication xsi:type="CDType">
   <Title>Timeless Serenity</Title>
   <Author>Dyveke Spino</Author>
   <Date>1984</Date>
   <RecordingCompany>Dyveke Spino Produc tions</RecordingCompany>
   </Publication>
              ...
</Catalogue>
```

Dipendenze minime: per estendere l'insieme di elementi che posssono apparire nell'elemento "contenitore" bisogna accedere soltanto al tipo astratto.

Processo scalabile: supponiamo di voler processare ogni elemento Publication con il seguento frammento di stylesheet:

```
<xsl:for-each select="Publication">
 -- do something --
</xsl:for-each>
```

Anche se vengono creati altri tipi, come CDType, non c'è sbisogno di modificare il codice.

Coesione Semantica: tutti gli elementi derivano dallo stesso tipo a livello gerarchico quindi.

Controllo sui namespace: la parte variabile sono le dichiarazioni degli elementi che sono all'interno delle definizioni dei tipi. Di conseguenza è possibile decidere se rendere visibili i namespaces o meno nell'istanza.

Svantaggi:

Nessuna indipendenza degli elementi: come per il metodo 1, tutti i tipi devono derivare dal tipo astratto. Questo, chiaramente, implica che ci siano delle proibizioni nell'uso dei tipi che

non derivano dal tipo astratto. Un caso concreto potrebbe essere quello in cui il tipo è in un altro schema.

Metodo 4: utilizzo di un tipo “ciondolante”(dangling)

Motivazione:

Supponiamo di voler creare un elemento a contenuto variabile che possa maneggiare anche simple content, i metodi visti finora non ci permettono di farlo. Abbiamo bisogno di un metodo che ci permetta di creare un elemento “contenitore” simpleType.
Un concetto chiave che dobbiamo tener presente è il seguente:

- Con l’elemento <import> l’attributo schemaLocation è opzionale.

Descrizione:

Consideriamo un esempio. Supponiamo di voler avere un elemento <sensor> che ha come contenuto il nome del sensore di una stazione metereologica:

```
<sensor>Barometric Pressure</sensor>
```

Ci sono due cose da notare:

1. Questo elementomaneggia un simpleType
2. Ogni stazione metereologica potrebbe avere dei sensori che sono unici. Quindi lo schema deve essere realizzato in modo che l’elemento sensor possa essere personalizzato da ogni stazione.

Implementazione:

Per prima cosa dichiarimao l’elemento sensor all’interno dello schema:

```
<xsd:element name=“sensor” type=“s:sensortype”/>
```

Da notare che l’elemento sensor è di tipo sensortype, che oltretutto proviene da un namespace diverso:

```
xmlns:s=“http://www.sensor.org”
```

Il prossimo passo è la chiave di tutto. Quando si importa questo namespace non bisogna fornire un valore per l’attributo schemaLocation. Per esempio:

```
<xsd:import namespace=“http://www.sensor.org”/>
```

Successivamente l’istanza deve identificare uno schema che implementi sensor_type. Quindi nel momento della validazione si deve far incontrare il riferimento a sensor_type con l’implementazione di sensor_type. Vediamo un esempio di istanza:

```
xsi:schemaLocation="http://www.weather-station.org weather-
station.xsd
http://www.sensor.org boston-sensors.xsd"
```

In questa istanza l'attributo schemaLocation sta identificando uno schema, boston-sensor.xsd, che è quello che fornisce l'implementazione di sensor_type. Riportiamo lo schema principale, che contiene il tipo dangling:

weather-station.xsd

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://www.weather-station.org"
 xmlns="http://www.weather-station.org"
 xmlns:s="http://www.sensor.org"
elementFormDefault="qualified">
 <xsd:import namespace="http://www.sensor.org"/>
 <xsd:element name="weather-station">
 <xsd:complexType>
<xsd:sequence>
  <xsd:element name="sensor" type="s:sensor_type"
 maxOccurs="unbounded"/>
</xsd:sequence>
 </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

L'elemento <import> non ha l'attributo schemaLocation per identificare un particolare schema che implementa sensor_type.
E questo è lo schema che implementa sensor_type:

boston-sensors.xsd

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://www.sensor.org"
 xmlns="http://www.sensor.org"
 elementFormDefault="qualified">
    <xsd:simpleType name="sensor_type">
<xsd:restriction base="xsd:string">
    <xsd:enumeration value="barometer"/>
    <xsd:enumeration value="thermometer"/>
    <xsd:enumeration value="anenometer"/>
</xsd:restriction>
    </xsd:simpleType>
</xsd:schema>
```

Adesso un'istanza può essere conforme allo schema weather-station.xsd e usare boston-sensors.xsd come l'implementazione di sensor_type:

boston-weather-station.xml

```
<?xml version="1.0"?>
<weather-station xmlns="http://www.weather-station.org"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation=
  "http://www.weather-station.org weather-station.xsd
http://www.sensor.org boston-sensors.xsd">
  <sensor>thermometer</sensor>
  <sensor>barometer</sensor>
  <sensor>anenometer</sensor>
 </weather-station>
```

Vediamo adesso di fare un esempio pratico. Supponiamo che la stazione metereologica di Londra abbia gli stessi sensori di quella di Boston, più alcuni che sono presenti sono nella stazione di Londra. Quindi la stazione di Londra creerà la sua propria implementazione di sensor_type:

london-sensors.xsd

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
         targetNamespace="http://www.sensor.org"
         xmlns="http://www.sensor.org"
         elementFormDefault="qualified">
  <xsd:simpleType name="sensor_type">
     <xsd:restriction base="xsd:string">
         <xsd:enumeration value="barometer"/>
         <xsd:enumeration value="thermometer"/>
         <xsd:enumeration value="anenometer"/>
         <xsd:enumeration value="hygrometer"/>
     </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

L'istanza della stazione di Londra sarà conforme ai due schemi boston-sensor.xsd e london-sensor.xsd:

london-weather-station.xml

```
<?xml version="1.0"?>
<weather-station xmlns="http://www.weather-station.org"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation=
   "http://www.weather-station.org weather-station.xsd
http://www.sensor.org london-sensors.xsd">
<sensor>thermometer</sensor>
<sensor>barometer</sensor>
<sensor>hygrometer</sensor>
<sensor>anenometer</sensor>
</weather-station>
```

La stazione metereologica di Londra, in questo modo, è in grado di personalizzare il contenuto sell'elemento <sensor> utilizzando lo schema london-sensor.xsd, che definisce sensor_type appositamente per la stazione di Londra.

Sommario:

Questo metodo è molto potente, i punti chiave da ricordare sono:

1. Quando si dichiara l'elemento a contenuto variabile è meglio dichiararlo di un tipo che deriva da un altro namespace, per esempio come nel nostro caso s:sensor_type;
2. Quando si importa il namespace non bisogna dare un valore all'attributo schemaLocation, cioè <xsd:import namespace="http://www.sensors.org"/>;
3. Creare un numero qualsiasi di implementazioni del tipo dangling, per esempio:
   - boston-sensors.xsd
   - london-sensors.xsd
4. Nell'istanza bisogna identificare lo schema che si desidera utilizzare per implementare il tipo dangling, come nel nostro caso:

```
xsi:schemaLocation=
  "http://www.weather-station.org weather-station.xsd
http://www.sensor.org london-sensors.xsd"
```

NOTA: nel nostro esempio abbiamo implementato il tipo dangling come un simpleType ma questo non è una costrizione, infatti potrebbe essere definito anche come un comlexType.

Vantaggi:

Dinamico: uno schema che contiene un tipo dangling è molto dinamico. Infatti non obbliga

l’identificazione stretta dello schema per implementare il tipo, anzi è l’autore dell’istanza che può scegliere lo schema da utilizzare per implementare il tipo dangling.

Svantaggi:

Namespace Diversi: l’implementazione del tipo dangling deve essere in un namespace diverso da quello dell’elemento “contenitore”.

Best Practice

Cerchiamo di capire quale metodo è consigliabile usare in deterinate situazioni:

Usare il metodo 1 quando:

- Tutti gli elementi discendono da un tipo comune;
- Si ha la necessità di estendere l’insieme di elementi che possono essere nel contenuto dell’elemento “contenitore” senza modificare lo schema;
- Si vuole fare in modo che tutti i namespaces siano visibili nell’istanza del documento;

Usare il metodo 2 quando:

Nasce l’esigenza che gli elementi contenuti nell’elemento “contenitore” siano tutti dissimili e indipendenti;
La scelta degli elementi, e quindi anche l’inserimento di nuovi, è fatta preventivamente;

Usare il metodo 3 quando:

Tutti gli elementi sono dello stesso tipo o derivano dallo stesso tipo;
È possibile dare a tutti gli elementi un nome uniforme;
Il numero degli elementi potrebbe crescere indipendentemente dallo schema principale;
C’è la necessità di non rendere visibili i namespaces nell’istanza;
C’è bisogno di supportare un processo scalabile;

Usare il metodo 4 quando:

- C’è bisogno di definire l’elemento “contenitore” come un simpleType;
- C’è la necessità di estendere un simpleType;
- Si vuole creare un contenuto personalizzabile e dinamico.

Best Practice:

Il Metodo 4 è lontano dall’essere l’approccio più flessibile.

2.7. Composition contro Subclassing

Problema

Il problema che affrontiamo in questa parte è quello di capire se è meglio realizzare uno schema per costruire gerarchie di tipo o per aggregare componenti.

Introduzione

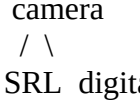
Affrontiamo il problema supponendo di dover realizzare uno schema per una macchina fotografica 35mm. Di seguito riportiamo un semplice esempio di un'istanza di documento:

```
<camera>
    <lens>
        <length>35mm</length>
        <f-stop>1.8</f-stop>
        <shutter-speed>
             <min>1/30 sec</min>
             <max>2 sec</max>
        </shutter-speed>
    </lens>
    <housing>
        <description>Ergonomically designed casing for easy
handling</description>
    </housing>
    <film>
        <ASA>200</ASA>
        <exposures>24</exposures>
        <brand>Kodak</brand>
    </film>
</camera>
```

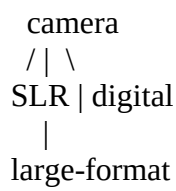
Un possibile schema potrebbe essere il seguente:

```
<xsd:element name="camera">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="lens" type="lens-type"/>
            <xsd:element name="housing" type="housing-type"/>
            <xsd:element name="film" type="film-type"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

Supponiamo però che successivamente ci sia la necessità di descrivere con lo schema anche macchine fotografiche digitali. Una possibile soluzione è quella di creare una gerarchia di tipo e quindi dire che le macchine fotografiche si suddividono in digitali e SLR (SLR - single lens reflex):

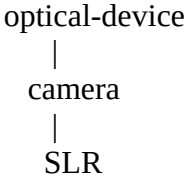


Ma non è detto che poi non si voglia aggiungere un altro sottoinsinsieme di macchina fotografica. Quindi seguendo lo stesso criterio del caso precedente si ha:
Inizio modulo



Questo modo di procedere in un certo senso permette di costruire una gerarchia, però si espande solo nel momento in cui si vuole aggiungere un nuovo componente. Vediamo quali sono gli svantaggi:

Complessità: supponiamo che camera derivi da optical-device.



Nel momento in cui la catena gerarchica diventa molto lunga bisogna sempre risalire all'origine della catena.

Elementi legati alla radice: cerchiamo di spiegare questo svantaggio riferendoci al nostro esempio. Tutti i tipi di macchina fotografica sono innestati dentro la stessa radice. Quindi nel momento in cui la radice viene modificata tutto quello che è al suo interno si modifica automaticamente. Di conseguenza questo aspetto potrebbe impedire di modificare lo schema.
Un approccio migliore potrebbe essere quello di definire un elemento astratto <recording-medium>:

```
<xsd:element name="recording-medium"
abstract="true" type="recording-medium-type"/>
```

e di dichiarare l'elemento <camera> in questo modo:

```
<xsd:element name="camera">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="lens" type="lens-type"/>
            <xsd:element name="housing" type="housing-type"/>
            <xsd:element ref="recording-medium"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

Facendo un confronto con lo schema precedente notiamo che l'elemento <film> è stato sostituito dal riferimento a un elemento astratto <recording-medium>. I componenti 35mm, disk e 3x5 sono dichiarati come elementi stabndalone che possono essere usati come sostituti dell'elemento recording-medium:

```
<xsd:element    name="35mm"    substitutionGroup="recording-medium"
type="35mm-type"/>
<xsd:element    name="disk"    substitutionGroup="recording-medium"
type="disk-type"/>
<xsd:element     name="3x5"    substitutionGroup="recording-medium"
type="3x5"/>
```

Un vantaggio molto importante di questo approccio è la **semplicità**: tutti i componenti sono standalone. Quindi possono essere modificati indipendentemente da tutti gli altri.
Consideriamo altri modi di procedere:
Contenimento diretto: la forma più semplice è quella di inserire la dichiarazione dell'elemento come abbiamo fatto nell'esempio precedente per lens e housing: "camera contiene lens e housing".
Contenimento per riferimento: in questo caso si inserisce un elemento vuoto con un attibuto IDREF. L'attributo fa riferimento a un elemento che contiene un attributo ID. in this approach we embed an empty element with an IDREF attribute. The attribute references an element containing an ID attribute.
Vediamo come un'applicazione processa un elemento che è stato realizzato utilizzando Composition. Gli elementi sono organizzati in questo modo:
camera
- lens
- housing
- recording-medium(abstract)
    ^
    |
    ---------- 35mm or disk or 3x5 etc

Fine modulo
Uno stylesheet può processare l'elemento camera delegando il tutto ad ogni sottoelemento:
Inizio modulo

```
<xsl:template match="camera">
    <!-- delegate to lens element -->
    <xsl:apply-templates select="lens"/>
    <!-- delegate to housing element -->
    <xsl:apply-templates select="housing"/>
    <!-- delegate to whatever recording-medium component is present -->
    <xsl:apply-templates select="*[last()]"/>
</xsl:template>
```

Nel momento in cui vengono richiesti nuovi tipi di componenti recording-medium basterà semplicemente applicare una regola di template come nell'esempio seguente:
Inizio modulo

```
<xsl:template match="35mm">
  ... process 35mm recording-medium ...
</xsl:template>

<xsl:template match="disk">
  ... process disk recording-medium ...
</xsl:template>
```

```
<xsl:template match="3x5">
  ... process 3x5 recording-medium ...
</xsl:template>
```

Tutti I componenti o sono dello stesso tipo o comunuqe derivano dallo stesso tipo quindi è anche possibile avere una singola regola di template per tutti i componenti recording-medium.

```
<xsl:template match="*[type() = 'recording-medium-type']">
  ... do some processing common to all recording-medium components ...
</xsl:template>
```

Best Practice

Sicuramente la composition è una soluzione migliore, perchè è più semplice, più robusta, facilmente modificabile e ha un design plug-and-play. Nel caso in cui si preferisca utilizzare subclassing è consigliabile seguire alcune semplici regole:
Evitare di usare la restrizione: quando si utilizza la restrizione il tipo derivato deve ripetere le dichiarazioni nel tipo parent, cioè:

```
<xsd:complexType name="T1">
    <xsd:sequence>
        <xsd:element name="E1" type="E1_type"/>
        <xsd:element name="E2" type="E2_type" maxOccurs="unbounded"/>
        <xsd:element name="E3" type="E3_type"/>
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="T2">
    <xsd:complexContent>
        <xsd:restriction base="T1">
            <xsd:sequence>
                <xsd:element name="E1" type="E1_type"/>
                <xsd:element name="E2" type="E2_type"/>
                <xsd:element name="E3" type="E3_type"/>
            </xsd:sequence>
        </xsd:restriction>
    </xsd:complexContent>
</xsd:complexType>
```

In questo esempio T2 è una restrizione di T1, in particolare in T2 l'elemento E2 non può apparine infinite volte, ma al massimo una. Però in T2 vengono ripetuti anche gli altri sottoelementi anche se non vengono modificati. Oltretutto qualsiasi modifica che viene fatta a T1 automaticamente anche T2 viene modificato, così come verranno modifcati tutti i tipi che derivano da T1. se invece T2 viene derivato da T1 come estensione, allora in quel caso le modifiche di T1 non si ripercuoteranno su T2.
La gerarchia deve essere limitata a un massimo di tre livelli: questo chiaramente non è un obbligo ma è solo un consiglio per rendere la struttra dello schema più flessibile.

2.8. Creazione di modelli a contenuto estensibile

Problema

Il problema che affrontiamo in questa sezione è quello di capire come realizzare elementi di cui possiamo modificare l'estensione nell'istanza con sottoelementi che non sono stati definiti nello schema.

Introduzione

```
<xsd:element name= "Book">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="Title" type="string"/>
      <xsd:element name="Author" type="string"/>
      <xsd:element name="Date" type="string"/>
      <xsd:element name="ISBN" type="string"/>
      <xsd:element name="Publisher" type="string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Lo schema sopra dichiara che nell'istanza l'elemento <Book> deve sempre avere come elementi child gli elementi <Title>, <Author>, <Date>, <ISBN>, e <Publisher>. Quindi un esempio di istanza potrebbe essere il seguente:

```
<Book>
    <Title>The First and Last Freedom</TItle>
    <Author>J. Krishnamurti</Author>
    <Date>1954</Date>
    <ISBN>0-06-0064831-7</ISBN>
    <Publisher>Harper &amp;amp; Row</Publisher>
</Book>
```

Quindi il contenuto dell'elemento deve essere rigidamente conforme alle specifiche dello schema. Ci saranno casi in cui questa rigidità è necessaria, e altri in cui sarebbe preferibile poter avere un'istanza più flessibile.

Di seguito riportiamo due modi che permettono di fare in modo che il contenuto di un elemento sia estensibile

Metodo 1: estensibilità utilizzando il tipo Substitution

Consideriamo lo schema precedente in cui però il contenuto dell'elemento Book è stato definito utilizzando il type definition:

```
<xsd:complexType name="BookType">
  <xsd:sequence>
    <xsd:element name="Title" type="xsd:string"/>
    <xsd:element name="Author" type="xsd:string"/>
    <xsd:element name="Date" type="xsd:string"/>
    <xsd:element name="ISBN" type="xsd:string"/>
    <xsd:element name="Publisher" type="xsd:string" />
```

```
    </xsd:sequence>
  </xsd:complexType>
  <xsd:element name=“BookCatalogue”>
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name=“Book” type=“BookType”
               maxOccurs=“unbounded”/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
```

Adesso definiamo un tipo che deriva da BookType:

```
  <xsd:complexType name=“BookTypePlusReviewer”>
    <xsd:complexContent>
      <xsd:extension base=“BookType” >
        <xsd:sequence>
          <xsd:element name=“Reviewer” type=“xsd:string”/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
```

in questo modo nell’istanza posso creare un elemento Book che contiene un elemento <Reviewer> seguito dagli altri cinque elementi:

```
  <Book xsi:type=“BookTypePlusReviewer”>
     <Title>My Life and Times</Title>
     <Author>Paul McCartney</Author>
     <Date>1998</Date>
     <ISBN>94303-12021-43892</ISBN>
     <Publisher>McMillin Publishing</Publisher>
     <Reviewer>Roger Costello</Reviewer>
  </Book>
```

Così siamo riusciti ad estendere il contenuto dell’elemento Book con un nuovo elemento!
In questo esempio non abbiamo modificato lo schema iniziale. Basta realizzare un altro schema in cui si importa o si include (import/include) lo schema iniziale e lì definire tipi che derivano da BookType.

MyTypeDefinitions.xsd

```
  <xsd xmlns =" http://www.publishing.org"
  <xsd:include schemaLocation="BookCatalogue.xsd"/>
  <xsd:complexType nam e="BookT ypePlusReviewer">
   <xsd:com plexContent>
   <xsd:extension bas e="BookType" >
  <xsd:sequence>
  <xsd:element name="Reviewer" type=" xsd:string"/>
  </xsd:sequence>
   </xsd:extens ion>
```

```
 </xsd:complexContent>
</xsd:complexType>
```

BookCatalogue.xsd

```
xmlns =" http://www.publishing.org"
<xsd:com plexType nam e="BookType">
 <xsd:sequence>
 <xsd:elem ent name="T itle" type="xsd:string"/>
 <xsd:elem ent name="Author"  type="xsd:string"/>
 <xsd:elem ent name="Date" type=" xsd:year"/>
 <xsd:elem ent name="ISBN" type=" xsd:string" />
 <xsd:elem ent name="P ublis her" type=" xsd:string" />
 </xsd:s equence>
</xsd:complexT ype>
<xsd:elem ent Book type="BookT ype"/>
```

Vediamo un esempio di istanza:

```
 xmlns="http://www.publishing.org"
 xsi:schemaLocation="http://www.publishing.org
  MyTypeDefinitions.xsd"
<Book xsi:type="BookTypePlusReviewer">
 <Title>The First and Last Freedom</Title>
 <Author>J. Krishnamurti</Author>
 <Date>1954</Date>
 <ISBN>0-06-064831-7</ISBN>
 <Publisher>Harper &amp;amp; Row</Publisher>
 <Reviewer>Roger L. Costello</Reviewer>
</Book>
```

Svantaggi:

Location Restricted Extensibility: gli elementi che vengono aggiunti sono posizionati alla fine del modello di contenuto (nel nostro esempio dopo l'elemento <Publisher>). Se per esempio si vuole estendere l'elemento <Book> aggiungendo degli elementi all'inizio o in qualsiasi altra posizione che non sia alla fine, non possiamo utilizzare questo metodo.

Estensibilità imprevista: consideriamo la definizione dell'elemento <Book>:

```
<xsd:element name="Book" type="BookType"
       maxOccurs="unbounded"/>
```

e del tipo BookType:

```
<xsd:complexType name="BookType">
  <xsd:sequence>
    <xsd:element name="Title" type="xsd:string"/>
    <xsd:element name="Author" type="xsd:string"/>
    <xsd:element name="Date" type="xsd:gYear"/>
```

```
    <xsd:element name=“ISBN” type=“xsd:string”/>
    <xsd:element name=“Publisher” type=“xsd:string”/>
  </xsd:sequence>
</xsd:complexType>
```

guardando le due definizioni viene naturale pensare che l’elemento <Book> dovrà contenere sempre gli elementi <Title>, <Author>, <Date>, <ISBN>, e <Publisher>. Però allo stesso tempo può accadere che qualcuno estenda il contenuro utillizzando l’elemento substitution. Quindi, si deve tenre conto del fatto che l’elemento <Book> può contenere più di cinque elementi child.

Metodo 2: estensibilità utilizzando l’elemento <any>

L’elemento <any>, se inserito in un content model, permette di inserire gli elementi nell’istanza del documento. Di seguito riportiamo un esempio sull’utilizzo dell’elemento <any>:

```
<xsd:element name= “Book”>
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name=“Title” type=“string”/>
      <xsd:element name=“Author” type=“string”/>
      <xsd:element name=“Date” type=“string”/>
      <xsd:element name=“ISBN” type=“string”/>
      <xsd:element name=“Publisher” type=“string”/>
      <xsd:any namespace=”##any” minOccurs=”0”/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

In questo modo l’elemento <Book> può contenere gli elementi <Title>, <Author>, <Date>, <ISBN>, <Publisher> e più altri elementi opzionali da qualsiasi altro namespaces.
L’elemento <any> può essere inserito in qualsiasi punto del content model.
Con questo metodo l’autore dell’istanza può inserire un elemento <Reviewer>, definito in un altro schema, dopo i cinque elementi che sono stati definiti nello schema originale. Sotto riportiamo la definizione dell’elemento <Reviewer>:

```
<xsd:element name="Reviewer">
 <xsd:complexType>
 <xsd:sequence>
 <xsd:element name="Name">
 <xsd:com plexType>
 <xsd:sequence>
 <xsd:elem ent name="F irst" type="xsd:string"/>
 <xsd:elem ent name="L ast" type="xsd:s tring"/>
 </xsd:sequence>
 </xsd:complexT ype>
 </xsd:element>
 </xsd:sequence>
 </xsd:complexType>
</xsd:element>
```

Poichè nello schema abbiamo previsto l'estensibilità dell'elemento <Book>, nell'istanza è lecito utilizzare l'elemento <Reviewer> all'interno dell'elemento <Book>:

```
<Book>
<Title>T he First and Last Freedom </Title>
<Author>J. Krishnam urti</Author>
<Date>1954</Date>
<ISBN>0-06-064831-7</ISBN>
<Publisher>Harper &amp;amp; Row</Publisher>
<rev:Reviewer>
 < rev:Name>
<rev:Last>Costello</rev:Last>
<rev:First>Roger</rev:First>
 < /rev:Name>
</rev:Reviewer>
</Book>
```

Un'alternativa è quella di definire un tipo BookType in cui andiamo ad inserire l'elemento <any>:

```
<xsd:complexType name="BookType">
 <xsd:sequence>
<xsd:element name="Title" type="xsd:string"/>
<xsd:element name="Author" type="xsd:string"/>
<xsd:element name="Date" type="xsd:year"/>
<xsd:element name="ISBN" type="xsd:string"/>
<xsd:element name="Publisher" type="xsd:string"/>
<xsd:any namespace="##any" minOccurs="0"/>
 </xsd:sequence>
</xsd:element>
```

e quindi di dichiarare Book di tipo BookType:

```
<xsd:element Book type="BookType"/>
```

Così dopo l'elemento <Publication> all'interno dell'elemento <Book> può essere presente qualsiasi cosa.
Inoltre è possibile controllare l'estensibilità utilizzando un tipo. Infatti basta utilizzare l'attributo block quando dichiari che l'elemento Book è di tipo BookType:

```
<xsd:element Book type="BookType" block="#all"/>
```

Con questo secondo metodo riusciamo a controllare dove posizionare l'estensibilità e quanto vogliamo rendere estensibile un elemento. Per esempio, nello schema possiamo dire che al massimo si vogliono inserire due nuovi elementi nell'elemento <Book>, vediamo come:

```
<xsd:complexType name="Book">
 <xsd:sequence>
 <xsd:any namespace="##other" minOccurs="0" maxOccurs="2"/>
 <xsd:element name="Title" type="xsd:string"/>
 <xsd:element name="Author" type="xsd:string"/>
 <xsd:element name="Date" type="xsd:string"/
 <xsd:element name="ISBN" type="xsd:string"/>
 <xsd:element name="Publisher" type="xsd:string"/>
 </xsd:sequence>
</xsd:complexType>
```

Riassumendo:

- l'elemento <any> deve essere messo nel punto in cui vogliamo l'estensibilità;
- nel caso in cui si desideri l'estenbilità in più punti è possibile inserire più elementi <any>;
- con l'attributo maxOccurs si può specificare quanti nuovi elementi al massimo sia possibile aggiungere.

Non-Determinism and the <any> element

Se nell'esempio precedente, in cui abbiamo inserito l'elemento <any> all'inizio del content model, avessimo settato il valore dell'attributo namespace ad "##any" si sarebbe potuto verificare un errore "non determinisi content model" al momento della validazione dello schema. Cerchiamo di capirne il motivo.
Supponiamo che effettivamente il valore dell'attributo namespace sia "##any" e di avere un'istanza del documento come nell'esempio seguente:

```
<Book>
 <Title>The First and Last Freedom</Title>
 <Author>J. Krishnamurti</Author>
 <Date>1954</Date>
 <ISBN>0-06-064831-7</ISBN>
 <Publisher>Harper &amp;amp; Row</Publisher>
</Book>
```

Quando il validatore dello schema si trova di fronte l'elemento <Title> non riesce a capire se l'elemento Title proviene dall'elemento <any> o dalla dichiarazione <xsd:element name="Title" .../> nello schema. Questo è un content model non-deterministico perché il validatore non riesce a determinare univocamente la provenienza di quell'elemento. Gli schemi non-deterministici non sono permessi.
La soluzione è quella di settare il valore del namespace a "##other", perché in questo modo il validatore sa che gli elementi aggiunti provengono da namespace diversi dal targetNamespace. Per esempio:

```
<?xml version="1.0"?>
<xsd:schema ...
 xmlns:bk="http://www.books.com" …>
<xsd:import namespace="http://www.books.com"
 schemaLocation="Books.xsd"/>
<xsd:complexType name="Book">
 <xsd:sequence>
 <xsd:any namespace="##other" minOccurs="0"/>
 <xsd:element ref="bk:Title"/>
 <xsd:element name="A uthor" type="xsd:string"/>
 <xsd:element name="D ate" type="xsd:string"/>
 <xsd:element name="ISBN" type="xsd:string"/>
 <xsd:element name="Publisher" type="xsd:string"/>
 </xsd:sequence>
</xsd:complexType>
</xsd:schema>
```

Adesso consideriamo questa istanza di documento:

```
<Book>
    <bk:Title>The First and Last Freedom</bk:TItle>
    <Author>J. Krishnamurti</Author>
    <Date>1954</Date>
    <ISBN>0-06-0064831-7</ISBN>
    <Publisher>Harper &amp;amp; Row</Publisher>
</Book>
```

Quando un validatore incontra bk:Title proveà a validarlo con la dichiarazione appropriata nello schema. Ma nasce un'ambiguità: Title è quello del namespace http:// www.books.com oppure deriva dall'uso dell'elemento any? Di conseguenza questo schema è ambiguo e quindi non può essere considerato corretto?
A questo punto dobbiamo trovare una soluzione per fare in modo di poter estendere un elemento. Basta mettere un elemento opzionale <other> e mettere il suo contenuto <any>. Riportaimo un esempio:

```
<xsd:complexType name="Book">
 <xsd:sequence>
 <xsd:element name="other" minOccurs="0">
 <xsd:any namespace="##any" maxOccurs="2"/>
</xsd:element>
 <xsd:element name="Title" type="xsd:string"/>
 <xsd:element name="Author" type="xsd:string"/>
 <xsd:element name="Date" type="xsd:string"/
 <xsd:element name="ISBN" type="xsd:string"/>
 <xsd:element name="Publisher" type="xsd:string"/>
 </xsd:sequence>
</xsd:complexType>
```

L'autore dell'istanza deve inserire nell'istanza un elemento "contenitore" (<other>) nel quale poter mettere tutti gli elementi che vengono aggiunti. Non è questa certo la soluzione migliore, ma è l'unica soluzione.

Best Practice

L'uso dell'elemento <any> è una buona soluzione, anche perché rende lo schema dinamico flessibile e semplice. Soprattutto dà molta libertà all'autore dell'istanza, infatti gli permmette di inserire liberamente i dati che gli sembrano più significativi.
Ma allostesso tempo dobbiamo sottolineare quali sono gli svantaggi di questa soluzione per colui che crea lo schema. L'autore dello schema non può prevedere tutte le varietà di dati che l'autore dell'istanza potrebbe aver bisogno di creare.

2.9. Extendere gli schemi XML

Problema

Controllo dei vincoli nelle istanza che non possono essere espressi con XML Schema

Introduzione

XML Schema è uno strumento molto potente ma che non permette di esprimere alcuni vincoli. Riportiamo alcuni esempi:

- Assicurare che il valore dell'elemento <Elevation> sia più grande del valore dell'elemento <Height>.
- Assicurare che: se il valore dell'attributo mode è "water" allora il valore dell'elemento <Transportation> può essere o airplane o hot-air balloon.
- se il valore dell'attributo mode è "air" allora il valore dell'elemento <Transportation> può essere boat o hovercraft.
- se il valore dell'attributo mode è "ground" il valore dell'elemento <Transportation> può essere car o bicycle.
- Assicurare che il valore dei due elementi <PaymentReceived> e <PaymentDue> sia uguale anche se i due elementi sono in due documenti diversi!

Per poter controllare tutti questi vioncoli è necessario utilizzare XML Schema insieme ad altri tool. Consideriamo la seguente istanza:

```
<?xml version="1.0"?>
<Demo xmlns="http://www.demo.org"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.demo.org demo.xsd">
 <A>10</A>
 <B>20</B>
</Demo>
```

Con XML Schema possiamo imporre i seguenti vincoli:

- l'elemento Demo (radice) contiene una sequenza di elementi, A seguito da B;
- l'elemento A contiene un intero;
- l'elemento B contiene un intero;

Di seguito, riportiamo uno schema che implementa questi vincoli:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.demo.org"
  xmlns="http://www.demo.org"
  elementFormDefault="qualified">
 <xsd:element name="Demo">
  <xsd:complexType>
  <xsd:sequence>
  <xsd:element name="A" type="xsd:integer"/>
  <xsd:element name="B" type="xsd:integer"/>
  </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

Con XML Schema non possiamo imporre che il valore di A sia maggiore del valore di B, quindi dobbiamo trovare un modo per poter fare questo. Ci sono tre possibili soluzioni per estendere XML Schema: 1) l' integrazione con altri linguaggi di schema; 2) la scrittura di un codice per l'espressione di costrizioni aggiunte; 3) l'espressione di vincoli con uno stylesheet XSLT/XPath

Integrazione con altri linguaggi di schema:

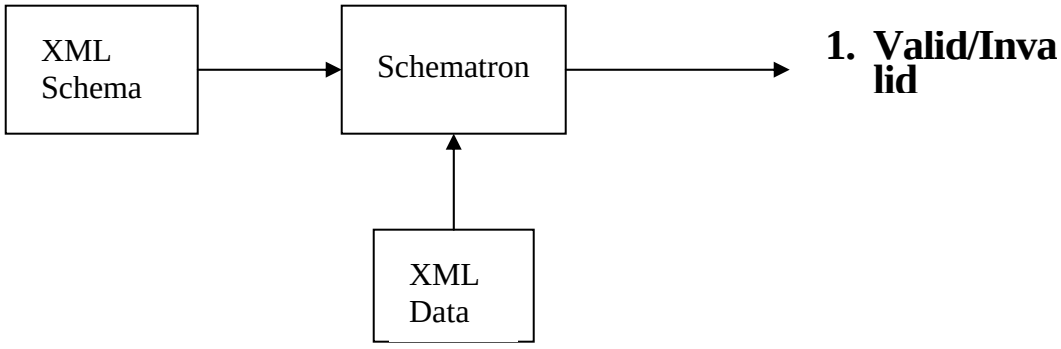
Ci sono altri linguaggi di schema oltre a XML Schema:

- Schematron
- TREX

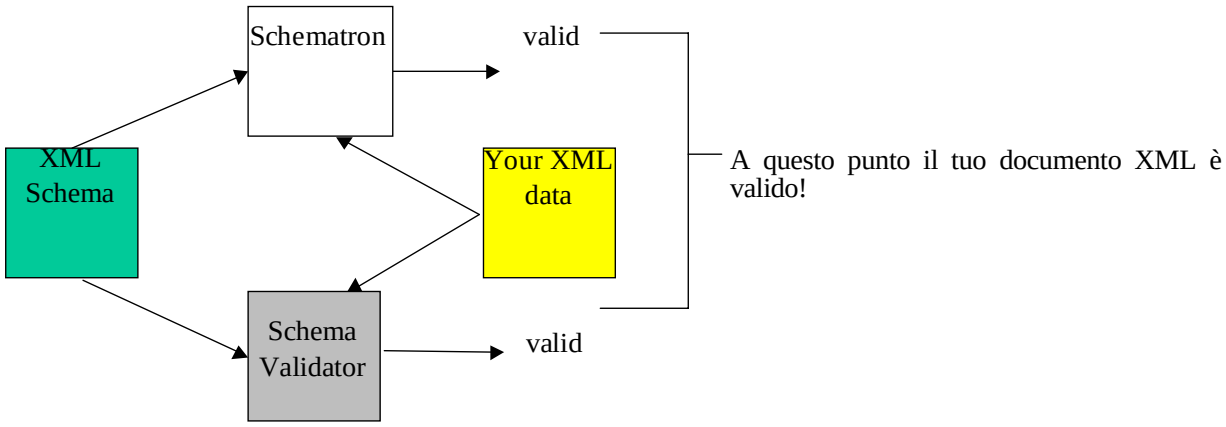
- RELAX
- SOX
- XDR
- HOOK
- DSD
- Assertion Grammars
- xlinkit

Quindi, una prima soluzione è quella di utilizzare uno o più di questi linguaggi per riuscire ad implementare i vincoli detti precedentemente. Soffermiamoci su uno di questi linguaggi: Schematron.

Using Schematron you embed the additional constraints (as “assertions”) within the schema document (within <appinfo> elements). A Schematron engine will then extract the assertions and validate the instance document against the assertions.



Questa è l’architettura che determina se i dati rispettano tutti i vincoli:



In Schematron:

– puoi dichiarare un vincolo utilizzando l'elemento <assert>
– l'elemento <rule> ~~contiene uno o più e~~lementi <assert>
– l'elemento <pattern> contiene uno o più elementi <rule>.

```
<pattern>
  <rule>
        <assert> … </assert>
        <assert> … </assert>
    </rule>
</pattern>
```

L'elemento <pattern> è innestato dentro l'elemento <appinfo> di XML Schema.
Vediamo un esempio di un'asserzione:

<assert test="d:A > d:B">A should be greater than B</assert>

Espressione XPAth Descrizione del vincolo imposto

L'elemento <assert> ha l'attributo test che ha come valore un'espressione Xpath ed è questa che permette di implementare il vincolo. In più, come valore dell'elemento si ha l'affermazione della costrizione nel linguaggio normale.

L'elemento <rule> è usato per specificare il contesto dell'elemento <assert>:

```
<rule context="d:Demo">
    <assert test="d:A &gt; d:B">A should be greater than B</assert>
</rule>
```

Queste due righe devono essere lette in questo modo:"All'interno del contesto dell'elemento Demo, si afferma che l'elemento A dovrebbe essere più grande dell'elemento B".

È possibile associare l'elemento <diagnostic> all'elemento <assert>. L'elemento <diagnostic> permmette di dare un messaggio di errore quando il documento XML non rispetta un determinato vincolo. Questo elemento deve essere annidato dentro l'elemento <diagnostics>

che segue immediatamente l'elemento <pattern>.

```
<pattern name="Check A greater than B">
      <rule context="d:Demo">
            <assert test="d:A &gt; d:B" diagnostics="lessThan"> A
should be greater than B
            </assert>
      </rule>
</pattern>
<diagnostics>
      <diagnostic id="lessThan">
            Error! A is less than B
A = <value-of select="d:A"/>
            B = <value-of select="d:B"/>
      </diagnostic>

</diagnostics>
```

Per identificare gli elementi di schematron, questi devono essere identificati dal prefisso sch. Vediamo come viene modificato lo schema:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSc hema"
   targetNamespac e="http://www.demo.org"
   xmlns="http://www.demo.org"
   xmlns:sch="http://www.ascc.net/xml/Schematron"
   eleme ntFormDefault="qua lified">
<xsd:annotation>
   <xsd:appinfo>
   <sc h:title>Schematr on validation</sch:title>
   <sc h:ns prefix="d" uri="http://www.demo.or g"/>
   </xsd:appinfo>
    </xsd:annotation>
    <xsd:ele ment name="Demo">
<xsd:annotation>
   <xsd:appinfo>
  <sc h:patter n name="Check A greate r than B">
 <sch:rule c ontext="d:Demo">
<sch:assert test="d:A &gt; d:B" diagnostics="lessThan">A should be
greater than B</sch:assert>
 </sch:rule>
  </sch:pattern>
  <sc h:diagnostics>
 <sch:diagnostic id="lessThan">
Error! A is less than B. A = <sch:value-of select="d:A"/>   B = <
sch:value-of se lect="d:B"/>
 </sch:diagnostic>
  </sch:diagnostics>
   </xsd:appinfo>
   </xsd:annotation>
   <xsd:complexType>
   <xsd:sequence >
```

```
  <xsd:element name="A" type="xsd:integer" minOccurs="1"
maxOccurs="1"/>
  <xsd:element name="B" type="xsd:integer" minOccurs="1"
maxOccurs="1"/>
   </xsd:sequence>
   </xsd:comple xType>
    </xsd:element>
</xsd:schema>
```

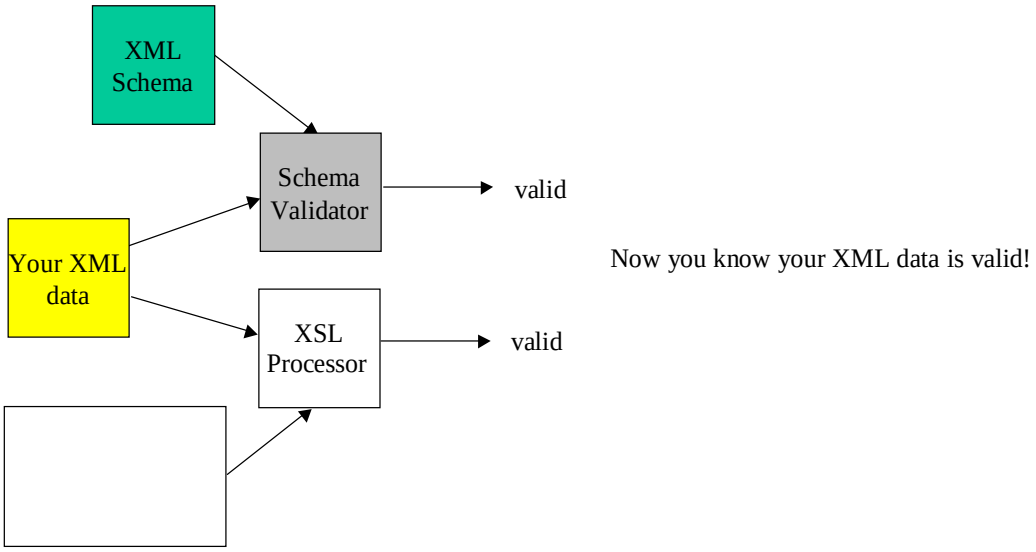
Schematron estrae le istruzioni dallo schema per poter creare lo schema Schematron. In seguito Schematron validerà l'istanza del documento con lo schema Schematron.

Scrivere un codice per esprimere le costrizioni aggiunte

La seconda soluzione prevede di scrivere un codice in Java, Perl, C++ per controllare i vincoli aggiuntivi.

Esprimere i vincoli con uno stylesheet XSLT/XPath

La terza soluzione prevde di scrivere uno stylesheet per implementare i voncoli.



Per esempio, il seguente stylesheet controlla l'istanza del documento per vedere se il contenuto dell'elemento A è maggiore dell'elemento B:

```
<?xml version="1.0"?>
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:d="http://www.demo.org"
  version="1.0">

<xsl:output method="text"/>

<xsl:template match="/">
<xsl:if test="/d:Demo/d:A &lt; /d:Demo/d:B">
<xsl:text>Error! A is less than B</xsl:text>
<xsl:text>&#xD;&#xA;</xsl:text>  <!-- carriage return -->
<xsl:text>A = </xsl:text><xsl:value-of select="/d:Demo/d:A"/>
<xsl:text>&#xD;&#xA;</xsl:text>   <!-- carriage return -->
<xsl:text>B = </xsl:text><xsl:value-of select="/d:Demo/d:B"/>
 </xsl:if>
 <xsl:if test="/d:Demo/d:A &gt;= /d:Demo/d:B">
<xsl:text>Instance document is valid</xsl:text>
 </xsl:if>
</xsl:template>
</xsl:stylesheet>
```

Se "validiamo" il documento XML precedente, verrà visualizzato il seguente messaggio di errore:

```
Error! A is less than B. A = 10, B = 20
```

Quindi, il modo di procedere per questa terza soluzione è la seguente:

– implementare I vincoli che è possibile con XML Schema;

– per tutte le altre scrivere uno stylesheet che ti permette di controllare se il documento XML è valido e se il processore XSL genera un'uscita corretta.

Vantaggi/Svantaggi delle tre soluzioni

-Integrazione con altri linguaggi di schema:

Vantaggi

- **Posizione delle costrizioni:** abbiamo visto che con Schematron è possibile inserire utleriori vincoli all'interno dello schema setsso. Questa è una caratteristica molto importante di Schematron. Questa possibilità non si ha con gli altri linguaggi.

- **Semplicità:** molti linguaggi sono stati creati in reazione alla complessità di XML Schema e alle sue limitazioni, per questo motivo sono molto semplici da imparare e da usare.

Svantaggi

- **Potrebbero essere richiesti più linguaggi:** ogni linguaggio ha le proprie caratteristiche e i propri limiti, quindi potrebbe nascere la necessità di utilizzare più linguaggi per poter implementare tutti i vincoli.
- **Ancora un altro vocabolario (YAV):** ogni linguaggio ha il proprio vocabolario e la propria semantica.
- **Il supporto sarà a lungo termine**

- Scrivere un codice per esprimere le costrizioni aggiunte

Vantaggi

- **Utilizzo si un solo linguaggio di programmazione:** in questo modo con un solo linguaggio di programmazione tu puoi esprimere tutti i vincoli.

Svantaggi

- **Non si utilizzano tecnologie XML.**

- Esprimere i vincoli con uno stylesheet XSLT/XPath

Vantaggi

- **Vincoli specifici per ogni applicazione:** ogni applicazione può creare un proprio stylesheet per verificare i vincoli che sono proprio di quell'applicazione! È possibile accrescere lo schema senza modificarlo.
- **Core Technology:** XSLT/XPath è una "core technology" che è ben supportata, ben capita e su cui è stato molto materiale.
- **Potenza espressiva:** XSLT/XPath è un linguaggio potente. Si può esprimere tutti i vincoli con questo linguaggio senza doverne imparare altri.

- **Supporto a lungo termine**

Svantaggi

- **Documenti separati:** questa soluzione prevede che tu scriva un tuo documento XML Schema, e poi un documento separato in XSLT/Xpath per esprimere I vincoli che non hai potuto espriemere con XMl Schema. Quindi bisogna che i due documenti siano sempre coerenti.

2.10. XML Namespace Name: URN o URL?

Problema

Scelta tra formulare il namespace XML Schema come URN o come URL?

Esempio:

urn:publishing:book ⟶ URN

http://www.publishing.com/book ⟶ URL

Che cos'è un nome del namespace XML Schema?

- I nomi del namespace sono valori unici.
- I nomi del namespace sono solo delle etichette.
- Non c'è nessuna esigenza di richiedere il namespace a una risorsa online.
- Il XML Schema Part 0: Primer (http://www.w3.org/TR/xmlschema-0) afferma che i namespaces target ci permettono di distinguere tra definizioni e dichiarazioni da vocabolari diversi.

Che cos'è un Uniform Resource Identifier (URI)?

- URI Generic Syntax (RFC 2396 – http://www.ietf.org/rfc/rfc2396.txt) definisce il seguente:
- Identificatore: un identificatore è un oggetto che può agire come un riferimento a qualcosa che ha identificato.
- Un URI può essere classificato al massimo come un locator, un nome o entrambi.
- Il termine "Uniform Resource Locator" (URL) si riferisce al sottoinsieme di URI che identificano risorse attraverso il meccanismo di accesso primario ad essa, piuttosto che attraverso il nome o altri attributi della risorsa.
- Il termine "Uniform Resource Name" (URN) si riferisce al sottoinsieme di URI che sono indipendenti dalla localizzazione della risorsa e che persistono anche quando la risorsa non esiste più o diventa indisponibile.

Perchè usare l'URN?

- Con l'URN è più semplice concettualizzare un nome che non una locazione. Quindi, utilizzando l'URN i namespaces identificano univocamente qualcosa e non la locazione di qualcosa.
- Gli utenti non si devono aspettare che l'URL localizzi una risorsa come accade per gli URL.

Perchè usare l'URL

- Gli URLs sono parte integrante del World Wide Web (www). Quando si utilizza un l'URL, *potenzialmente* è presente una risorsa associata. Questa risorsa potrebbe contenere della documentazione (uno schema, un puntatore allo schema). Se in un futuro il W3C decidesse di utilizzare il nome del namespace per puntare ad una risorsa, la sintassi appropriata sarà già in uso ei nomi dei namespaces non dovranno essere modificati.
- La sintassi dell'URL è familiare agli utenti del web.
- I nomi URL degli schemi sono già gestiti, quindi è molto più semplice assicurare che i nomi dei namespaces siano unici.

2.11. XML Schema Versioning

Problema

Qual è il modo migliore per stabilire la versione di XML Schema?

Introduzione

È abbastanza evidente il fatto che XML Schema con il passare del tempo può evolvere e quindi diventa importante stabilire qual è la sua versione.

Cambiamenti dello schema – Due casi

Consideriamo due casi di cambiamento dello schema:

Caso 1. Il nuovo schema cambia il modo di interpretare alcuni elementi.
Per esempio, un elemento che era valido per il precedente schema non è validato dal nuovo schema.

Caso 2. Il nuovo schema estende il namespace, per esempio aggiungendo nuovi elementi, ma non invlilida i documenti precedentemente validati.

Vediamo alcune soluzioni per identificare qual è la versione dello schema:

1. Cambiare l'attibuto version dell'elemento schema
2. Creare un attributo schemaVersion nell'elemento radice.
3. Cambiare il targetNamespace dello schema.
4. Cambiare il nome/locazione dello schema.

Soluzione 1: modificare l'attributo version di schema.

Un modo per risolvere il problema è quello di modificare il numero dell'attributo opzionale all'inizio dello schema. Per esempio nel codice seguente si potrebbe mettere version="1.1" invece che version="1.0"

```
       <xs:schema             xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" attributeFormDefault="unqualified"
version="1.0">
```

Vantaggi:

- Facile. Fa parte delle specifiche dello schema.
- Le istanze dei documenti possono non cambiare se rimangono validi anche con la nuova versione dello schema.
- Lo schema contiene le informazioni che possono informare l'applicazione che ci sono state delle modifiche. Quindi l'applicazione potrebbe interrogare l'attributo version e rendersi conto che quella è una nuova versione dello schema e agire di conseguenza.

Svantaggi:

- Il validatore ignora l'attributo version.

Soluzione 2: creare un attributo schemaVersion nell'elemento radice.

Optando per questa soluzione, si inlude nell'elemento che introduce il namespace un attributo. Questa soluzione può essere usata in due modi:

A: Primo, come nella soluzione 1 questo attributo potrebbe essere utilizzato per capire qual è la versione dello schema. Quindi, si potrebbe fare in modo che l'attributo sia obbligatorio e di valore fisso. Di conseguenza, ogni istanza che utilizza quello schema dovrebbe settare il valore dell'attributo al valore utilizzato nello schema.

```
    <xs:schema                          xmlns="http://www.exampleSchema"
    targetNamespace="http://www.exampleSchema"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified" attributeFormDefault="unqualified">
    <xs:element name="Example">
    <xs:complexType>
    ….
    <xs:attribute name="schemaVersion" type="xs:decimal" use="required"
    fixed="1.0"/>
    </xs:complexType>
    </xs:element>
```

Vantaggi:

- L'attributo schemaVersion è obbligatorio, quindi le istanze che non lo stsso numero della versione non sono validate.

Svantaggi:

- Il valore dell'attributo schemaVersion deve essere esattamente lo stesso nell'istanza. Quindi se si utilizzano più versioni dello schema l'istanza in alcuni casi potrebbe essere non valida.

B: in questo secondo caso si utilizza l'attributo schemaVersion in un modo completamente diverso. Non ha più un valore fisso nello schema, ma piuttosto è utilizzato nell'istanza per dichiarare la versione dello schema con il quale l'istanza è compatibile. Si potrebbe anche stabilire una convenzione con la quale si dichiara come questo attributo deve essere usato. Quindi si potrebbe stabilire che l'attributo schemaVersion dichiara qual è l'utlima versione con la quale l'istanza è compatibile. Vediamo un esempio:

Esempio di schema (la versione è 1.3)

```
<xs:schema                          xmlns="http://www.exampleSchema"
targetNamespace="http://www.exampleSchema"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified"    attributeFormDefault="unqualified"
version="1.3">
<xs:element name="Example">
<xs:complexType>
….
<xs:attribute         name="schemaVersion"         type="xs:decimal"
use="required"/>
</xs:complexType>
</xs:element>
```

Esempio di istanza (dichiara che la versione con cui è compatibile è 1.2
(oppure con la versione 1.2 o altre secondo qual è la convenzione))

```
<Example schemaVersion="1.2"
xmlns="http://www.example"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.example MyLocation\Example.xsd">
```

Vantaggi:

- L'istanza può non essere modificata se rimane valida anche con l'ultima versione dello schema.
- Come nel primo caso, l'applicazione potrebbe ricevere un'indicazione che lo schema è stato modificato.

Svantaggi:

- Richiede processi extra da parte dell'applicazione

Soluzione 3: cambiare il targetNamespace dello schema.

Il targetNamespace dello schema potrebbe essere modificato per indicare che esiste una nuova versione dello schema. Un modo per fare questo potrebbero essere quello di includere

nel valore del targetnamespace il numero della versione dello schema, come abbiamo fatto nel seguente esempio:

```
<xs:schema                              xmlns="http://www.exampleSchemaV1.0"
targetNamespace="http://www.exampleSchemaV1.0"
xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified">
```

Vantaggi:

- Le applicazioni sono avvertite del cambiamento dello schema.
- Ci deve essere un'azione che permette di assicurare che non ci siamo delle incompatibilità con il nuovo schema. Sia l'istanza che usa lo schema, che lo stesso schema devono cambiare il riferimento al nuovo targetNamespace. Questo però è sia un vantaggio che uno svantaggio.

Svantaggi:

- In questo modo i documenti non saranno validi fino a quando non sarà modificato il valore del targeNamespace.
- Qualsiasi schema che include questo schema deve essere modifcato, perché il targetNamespace dei componenti inclusi deve essere lo stesso dello schema incluso.

Soluzione 4: Modifica del nome/locazione dello schema.

Questa soluzione prevede di modificare il nome del file o la locazione dello schema.

Vantaggi:

Svantaggi:

- implica la modifica dell'istanza anche se il cambiamento dello schema non va a coinvolgere l'istanza.
- qualsiasi schema che importa lo schema deve modificare il nome o la locazione dello schema importato.
- le applicazioni non ricevono nessuna informazione sul cambiamento.
- l'attributo schemaLocation non è obbligatorio.

Best Practices

[1] Cattura la versione dello schema in qualunque area dell' XML schema.

[2] Identificare nell'istanza del documento qual è la versione dello schema con la quale l'istnza è compatibile.

[3] Rendere disponibili le versioni precedenti dello schema.

In questo modo si permette alle applicazioni di poter utilizzare anche le versioni meno recenti dello schema. Per fare questo si potrebbero avere delle applicazioni che pre-parsano l'istanza e quindi scelgono lo schema appropriato basandosi sul numero della versione. Per esempio, si potrebbe avere l'URI dello schemaLocation che punta a un documento che include una lista delle locazioni delle versioni disponibili dello schema. E quindi, utilizzare un tool per stabilire qual è la versione corretta dello schema. Questo approccio ha, però, lo svantaggio che l'istanza deve essere pre-parsata e poi validata.

[4] Quando le modifiche apportate allo schema sono soltanto delle estensioni, si potrebbe anche non invalidare nuovamente le istanze già esistenti.
Per esempio, se vengono aggiunti nuovi elementi si potrebbe pensare di renderli opzionali.
Un possibile approccio è il seguente:
Cambiare il numero della versione dello schema all'interno dello schema;
Registrare i cambiamenti nello schema;
Rendere disponili la nuova versione e le precedenti dello schema.

[5] Un altro modo di procedere potrebbe essere quello di modificare il targetNamespace dove lo schema modifica l'interpretazione di alcuni elementi. In questo caso i cambiamenti da eseguire sono gli stessi del caso precedente, ma con qualcosa in più:
Modificare il targetNamespace;
Aggiornare le istanze con il relativo targetNamespace;
Confermare che non ci siano problemi di compatibilità con il nuovo schema;
Cambiare l'attributo che identifica la versione dello schema che è compatibile con l'istanza;
Aggiornare il nome/locazione dello schema se necessario.

2.12. Implementazione della gerarchia degli elementi in un SubstitutionGroup

Problema

Dare una gerarchia agli elementi che appartengono a un Substitution Group

Introduzione

Nella sezione in cui abbiamo discusso dei metodi per implementare elementi a contenuto variabile, ne abbiamo visto uno che prevede di creare un elemento astratto, e quindi creare gli elementi che possono sostituirlo. Facciamo un esempio:

```
<xsd:element name="Publication" abstract="true"
             type="PublicationType"/>

<xsd:element name="Book" substitutionGroup="Publication"
             type="BookType"/>
```

```
<xsd:element name="Magazine" substitutionGroup="Publication"
             type="MagazineType"/>
```

dove BookType e MagazineType derivano entrambi da PublicationType:

```
 PublicationType
  /      \
BookType  MagazineType
```

L'elemento a contenuto variabile è dichiarato per contenere l'elemento astratto:

```
<xsd:element name="Catalogue">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="Publication" maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

Quindi, l'elemento <Catalogue> può contenere qualsiasi elemento che fa parte del substitution Group con Publication:

```
<Catalogue>
    <Book>...</Book>
    <Magazine>...</Magazine>
    <Book>...</Book>
</Catalogue>
```

L'alternativa è quella di creare un tipo con il nome type:

```
<xsd:complexType name="PublicationContainer">
    <xsd:sequence>
        <xsd:element ref="Publication" maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:element name="Catalogue" type="PublicationContainer"/>
```

Supponiamo di voler dichiarare un elemento che mantiene gli elementi di <Book>:

```
<xsd:element name="BookCatalogue">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="Book" maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

Oppure come nel caso precedente possiamo creare un tipo con un nome:

```
<xsd:complexType name="BookContainer">
    <xsd:sequence>
        <xsd:element ref="Book" maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>
```

```
<xsd:element name="BookCatalogue" type="BookContainer"/>
```

Ugualmente, per dichiarare un elemento che mantiene l'elemento <Magazine> dobbiamo fare un set di dichiarazioni simili:

```
<xsd:complexType name="MagazineContainer">
    <xsd:sequence>
        <xsd:element ref="Magazine" maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:element name="MagazineCatalogue" type="MagazineContainer"/>
```

Riassumendo:

La gerarchia originale è la seguente:

```
    PublicationType
     /     \
BookType   MagazineType
```

Si ha un substitution Group di cui fanno parte tre elementi:

{Publication, Book, Magazine}

E abbiamo tre tipi che contengono gli elementi del substitution Group:

PublicationContainer, BookContainer, MagazineContainer

Questi tre tipi di "contenitori" sono totalmente incorrelati.

Relazioni tra i tipi degli elementi contenitori

Sappiamo benissimo che ci sono dei benefici nel creare delle gerarchie di tipo. Teoricamente è possibile dichiarare un elemento in modo che sia del tipo radice della gerarchia, e che il contenuto dell'elemento possa essere sostituito da qualsiasi tipo derivato.
Possiamo fare in questo modo:

```
    PublicationContainer
     /       \
BookContainer    MagazineContainer
```

```
<xsd:complexType name="BookContainer">
    <xsd:complexContent>
        <xsd:restriction base="PublicationContainer">
            <xsd:sequence>
                <xsd:element ref="Book" maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:restriction>
    </xsd:complexContent>
</xsd:complexType>
```

```
<xsd:complexType name="MagazineContainer">
    <xsd:complexContent>
        <xsd:restriction base="PublicationContainer">
            <xsd:sequence>
                <xsd:element ref="Magazine" maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:restriction>
    </xsd:complexContent>
</xsd:complexType>
```

Come vediamo l'esempio, il tipo base contiene un elemento Publication e l'elemento derivato contiene un elemento Book. Questo è permesso,infatti basta pensare substitution group come choice. Infatti nel tipo base PublicationContainer c'è una scelta tra Publication, Book e Magazine, nel tipo derivato non si fa altro che restriingere la scelta a Book.

Sommario

Per prima cosa abbiamo dischiarato l'elemento astratto e i suoi elementi substitution Group:

```
    <xsd:element name="Publication" abstract="true"
                 type="PublicationType"/>

    <xsd:element name="Book" substitutionGroup="Publication"
                 type="BookType"/>
    <xsd:element name="Magazine" substitutionGroup="Publication"
                 type="MagazineType"/>
```

Poi si dichiara un tipo "contenitore" per ogni elemento, e tale tipo è la radice della gerarchia di tipo:

```
    <xsd:complexType name="PublicationContainer">
        <xsd:sequence>
            <xsd:element ref="Publication" maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="BookContainer">
        <xsd:complexContent>
            <xsd:restriction base="PublicationContainer">
                <xsd:sequence>
                    <xsd:element ref="Book" maxOccurs="unbounded"/>
                </xsd:sequence>
            </xsd:restriction>
        </xsd:complexContent>
    </xsd:complexType>

    <xsd:complexType name="MagazineContainer">
        <xsd:complexContent>
            <xsd:restriction base="PublicationContainer">
                <xsd:sequence>
                    <xsd:element ref="Magazine" maxOccurs="unbounded"/>
                </xsd:sequence>
            </xsd:restriction>
        </xsd:complexContent>
    </xsd:complexType>
```

Dichiariamo <Catalogue> in modo che sia di tipo PublicationContainer:

```
<xsd:element name="Catalogue" type="PublicationContainer"/>
```

Questa è una possibile istanza:

```
<Catalogue>
    <Book>...</Book>
    <Magazine>...</Magazine>
    <Book>...</Book>
</Catalogue>
```

PublicationContainer contiene un elmento astratto (Publication), così <Catalogue> deve contenere soltanto elementi che sono nel substitution group con Publication, come abbiamo fatto nel nostro esempio.
In questo modo possiamo tranquillamente sostituire PublicationContainer con qualsiasi tipo derivato come in questo caso:

```
<Catalogue xsi:type="BookContainer">
    <Book>...</Book>
    <Book>...</Book>
    <Book>...</Book>
</Catalogue>
```

Best Practice

Questo approccio permette di sostituire gli elementi e i tipi. Nel nostro esempio abbiamo visto che il contenuto dell'elemento <Catalogue> può essere qualsiasi elemento del substitution Group con Publication. Inoltre è anche possibile restringere il contenuto di <catalogue> a un particolare elemento del substitution Group.