UNIVERSITÀ DEGLI STUDI DI MILANO

Facoltà di scienze Matematiche, Fisiche e Naturali

Dipartimento di Informatica e Comunicazione

Corso di Laurea in Scienze e Tecnologie della Comunicazione Musicale

SVILUPPO DI UN PARSER PER SCHEMI XSD NECESSARIO PER LA CREAZIONE DI UN'INTERFACCIA DINAMICA ATTA ALLA COMPILAZIONE GUIDATA DI XML

Relatore: Prof. Luca Andrea Ludovico

Correlatore: Matteo Perelli

Elaborato di:

Massimiliano SALINA

Matricola: 709065

Anno Accademico 2008 – 2009

Indice

1 – Introduzione

La presente tesi si può collocare all'interno di un contesto lavorativo molto preciso. Riguarda infatti quella che definiamo come la gestione e la realizzazione delle attività di digitalizzazione di risorse culturali.

Questi sono progetti che ci permettono di risolvere e di avvicinarci alle problematiche della conservazione del patrimonio culturale e artistico.

Sappiamo infatti che le opere a disposizione nel dominio digitale, introducono una serie di vantaggi inerenti alla gestione dell'informazione. Si dispone ad esempio di ampie possibilità di messa a disposizione del pubblico, piuttosto che capacità maggiori per la conservazione a lungo a termine. Bisogna notare che questo aspetto riguarda sia le opere antiche, magari disponibili solo su supporti obsoleti con tutte le problematiche inerenti oppure supporti cartacei non accessibili al pubblico, sia le opere che vengono prodotte oggi.

È importante anche tenere presente che l'attività di digitalizzazione non riguarda soltanto il processo in sé di conversione dell'informazione, ma coinvolge anche una notevole quantità di lavoro dal punto di vista della gestione dei documenti digitali ottenuti. Questo in applicazioni quali ad esempio l'inserimento in un database, piuttosto che la fruizione o una descrizione tramite metadati.

1.1 – Il progetto e l'obiettivo

L'obiettivo del presente progetto sarà quello di sviluppare il core per un'interfaccia grafica dinamica, atta alla compilazione guidata di XML.

Da sottolineare il concetto di dinamicità conferito a tale interfaccia: questa infatti dovrà essere generata a partire da un qualunque schema XSD.

Il contesto è inizialmente descrivibile tramite le tre principali fasi di lavoro:

1. gestione delle informazioni di input
2. organizzazione del core dei dati
3. realizzazione della parte grafica dell’interfaccia

Lo schema XSD è il punto di partenza per il *parser*. Esso descrive la struttura che un documento XML dovrà avere. Leggendo uno schema XSD diventa possibile selezionare le informazioni che si ritengono utili ai fini dell'obiettivo, ossia le informazioni di input (punto 1).

Queste informazioni ottenute andranno di conseguenza organizzate in una *struttura dati* (punto 2). Essa dovrà quindi essere rappresentativa dei dati prelevati dallo schema.

Nel momento in cui si dispone di tale struttura dati sarà poi possibile utilizzarla per comporre i campi dell'interfaccia grafica (punto 3). Questi saranno compilati dall'utente il quale creerà, a quel punto, il proprio documento XML.

Il primo e il secondo punto saranno l'obiettivo principale della presente tesi: determinare quali sono i dati rilevanti per tale scopo presenti nello schema XSD e trovare un modo ordinato ed intelligente di organizzarli per costruire una struttura dati che li rappresenti.

Uno degli aspetti cruciali per la realizzazione sarà proprio la reperibilità in qualunque momento di qualunque informazione ritenuta utile. Nel momento in cui anche soltanto una di esse dovesse divenire irraggiungibile sarà assolutamente necessario riorganizzare il sistema.

Questo progetto è a supporto della piattaforma Codex[ml] sviluppata presso il CILEA, la piattaforma integrata che segue l'intero processo di creazione del dato digitale, la sua archiviazione, la gestione, la fruizione e la sua conservazione a lungo termine.

1.2 – I motivi

In questo ambiente viviamo oggigiorno nel mezzo di una crescente esigenza di gestione delle varie fasi di tale attività. Si vuole infatti contrastare il rischio di perdere definitivamente la memoria sui beni culturali.

L'effetto pratico della realizzazione di tale progetto sarà quello di permettere ad utenti non specializzati di creare XML, organizzando così l'attività di digitalizzazione delle risorse culturali.

Essere in grado di generare documenti XML significa poter utilizzare un metalinguaggio capace di descrivere documenti strutturati. Il vantaggio quindi di poter organizzare così le informazioni si rispecchia in quelli che sono i vantaggi intrinseci dell'XML: come discusso in [B3 e S3] troviamo la flessibilità, cioè poter gestire il sistema dei *marcatori* (i "tag") a seconda delle proprie esigenze e

senza limitazioni, la compatibilità, cioè rendere possibile la comunicazione tra applicazioni molto differenti tra di loro, e la portabilità, grazie proprio alla compatibilità dell'XML la nostra applicazione diventa molto più portabile in quanto si basa su un linguaggio che ha un livello di astrazione superiore rispetto agli altri.

1.3 – Risultati attesi

Come già specificato l'obiettivo di questa tesi, data la complessità e il grado di ricerca coinvolti, è soprattutto giungere al controllo dei dati in input e della successiva costruzione di una struttura dati precisa e ordinata.
La realizzazione di questa parte equivale a generare fondamentalmente il cuore dell'intero lavoro.
In tutto questo c'è da notare che per come si presentano le informazioni all'interno di uno schema XSD è necessario, al fine dell'interfaccia grafica, averle ordinate e strutturate in modo diverso rispetto a come si presentano. Questo implica che il documento sia sottoposto a più analisi sintattiche e su più livelli, in grado di riconoscere tutti gli elementi che incontrano per poterli riorganizzare nel modo più funzionale.

1.4 – Il CILEA e Codex[ml]

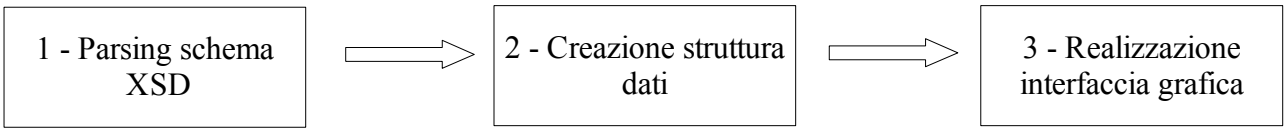
Il CILEA (Consorzio Interuniversitario Lombardo per l'Elaborazione Automatica) è il consorzio che ha permesso la realizzazione di tale idea.

Fondato nel 1974 nasce con lo scopo di poter offrire alle università consorziate potenza elaborativa a supporto della ricerca e relativi servizi, offrendo gli stessi anche a sostegno della didattica. Col passare del tempo le attività si diversificano e si ampliano anche verso altre direzioni: non si occupa più soltanto di mettere a disposizione l'utilizzo delle proprie macchine, ma si concentra sullo sviluppo e sulla creazione di altri tipi di servizio. Più precisamente riesce ad operare in settori eterogenei quali il calcolo ad alte prestazioni, le biblioteche–editoria elettronica–digital library o lo sviluppo di software.
Il CILEA si presenta ora come un consorzio presente in tutto il territorio nazionale ed è noto

soprattutto per i suoi servizi alle università e agli enti di ricerca, oltre che al ministero.
Codex[ml] è la piattaforma nella quale si contestualizza concretamente il presente progetto.
Il sistema integrato Codex[ml] V2 è stato sviluppato dal CILEA per la gestione, conservazione e fruizione via web delle risorse digitali, come evoluzione del precedente sistema Codex[ml]. La piattaforma consente ad ogni istituzione culturale, biblioteca o archivio che possieda informazioni cartacee digitalizzate, la relativa fruizione attraverso la navigazione via web, sfruttando l'analisi dei diversi standard di metadati, in linguaggio XML. Il sistema si presenta modulare e scalabile: ogni modulo è un elemento indipendente che interagisce con gli altri per garantire la possibilità di creare, conservare, distribuire immagini corredate da metadati. L'utente può navigare all'interno della risorsa digitale visualizzandola con un'interfaccia web, sfruttando la struttura descritta nei metadati, codificati in linguaggio XML, secondo le specifiche previste dagli standard METS, MAG, ecc., ma predisposta anche per nuovi standard non ancora scritti.
Il sistema è stato sviluppato interamente in ottica OO, con pHp 5.2.0-8. Codex[ml] V2 è affiancato da IBM DB2 9, un database ibrido di nuova generazione con gestione ottimizzata di dati relazionali e XML. Mediante la tecnologia 'Pure XML', DB2 fornisce l'abilità di salvare informazioni in campi di tipo XML, favorendo la gestione di applicazioni che interagiscono con documenti XML e permettendo ad esempio, l'interrogazione diretta del database tramite XQuery. Codex[ml] V2 viene installato solamente su sistemi operativi Linux / Unix.
Da un punto di vista tecnico Codex[ml] V2 può essere considerato un insieme di applicativi collegati, la cui base è rappresentata da un framework, in questo caso lo Zend Framework. Il Framework è un insieme di moduli e classi cooperanti atte a realizzare un infrastruttura riutilizzabile e personalizzabile all'interno della web application che racchiude tutte le funzionalità di base condivise. Esso però non si interfaccia direttamente con i vari moduli del sistema, ma viene visto da essi attraverso un sistema di risorse condiviso, chiamato Maestrale.

2 – Realizzare l'idea

Richiamando i tre punti elencati nel capitolo 1.1, possiamo riassumere schematizzando le principali fasi di lavoro come segue:



Oggetto di questa tesi è la messa a punto per concretizzare i primi due blocchi di questo schema.
Per rendere la descrizione più precisa occorre, prima di spiegarli in dettaglio, specificare l'esposizione della parte teorica, ossia dove viene stabilita a tavolino una guida da seguire per raggiungere l'obiettivo, e quella della parte più pratica in cui, seguendo appunto la linea guida, si traduce l'idea in un'azione reale.

2.1 - Pensare all'obiettivo: la progettazione inversa

Come in tutti i progetti esistenti prima di cominciare a lavorare è necessario dedicare del tempo a pensare a come, dato un preciso traguardo, si può cominciare a muoversi per raggiungerlo. Bisogna conoscere bene contro cosa ci si sta scontrando, in modo da prevedere nel modo più preciso possibile quali saranno i potenziali ostacoli.
La *progettazione inversa* è una metodologia di approccio al lavoro: partendo da quello che dovrà essere il risultato finale si percorrono a ritroso e si disegnano le tappe da cui passare per la realizzazione, fino a raggiungere di conseguenza l'inizio, cioè quello che sarà poi lo "start" per la parte pratica del lavoro.
Si è detto che si vuole realizzare un sistema in grado, partendo da un schema XSD, di generare automaticamente uno strumento (l'interfaccia grafica) per creare documenti XML. Si può per ora distinguere quali sono i tre protagonisti principali del progetto:

1 – Un **linguaggio di programmazione** per la manipolazione dei dati;

2 – **Schema XSD**;

3 – **Documento XML**;

In questo che quindi è un percorso mentale seguito al contrario rispetto alle reali fasi di lavoro, il primo elemento da prendere in considerazione è il documento XML. Bisogna porsi delle domande:

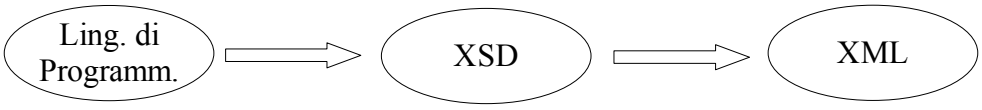
- cosa caratterizza un XML?
- come si definisce e sulla base di cosa si costruisce un XML?
- quali dati servono per creare correttamente un XML?

Per trovare le risposte a tutte e tre le domande dobbiamo procedere a ritroso. Lo schema XSD ci viene dunque incontro, essendo lui a stabilire le regole di formazione corrette per scrivere un documento XML.

- come si presenta uno schema XSD?
- come si possono selezionare le informazioni utili?
- quali sono le informazioni utili?

Ecco che entrano in gioco i linguaggi di programmazione: linguaggi formali dotati di sintassi, semantica e lessico definiti e che si utilizzano per il controllo del comportamento di un dato sistema che si vuole costruire.
Occorre quindi scrivere del codice che sia in grado di leggere uno schema XSD, e che contemporaneamente sappia riconoscere, selezionare ed usare ciò che si ritiene un'*informazione utile*.
L' approccio mentale sinora descritto, si potrebbe riassumere e schematizzare così:

2.2 - Concretizzare l'obiettivo



La parte pratica, vista la progettazione inversa, vuole quindi che quest'ultima sia presa e rovesciata. A questo punto è quindi noto come nella realizzazione i tre protagonisti andranno seguiti.
Come sarà meglio descritto avanti, il lavoro da svolgere prevede che partendo dalla programmazione si scrivano una serie di metodi che, incastrati correttamente tra loro, siano in grado per prima cosa di "leggere" lo schema XSD; consecutivamente occorre che proprio durante questa lettura, le informazioni utili siano estrapolate e che vadano a formare quello che abbiamo già definito come il cuore dell'intero lavoro: la struttura dati rappresentativa dello schema XSD.

2.2.1 – Scelta delle tecnologie

Come già accennato nell'introduzione questo elaborato coinvolge un certo grado di ricerca. Questo aspetto di conseguenza si rifletterà in tutte le decisioni e le scelte che saranno da intraprendere, sia nella parte logica ed intellettuale, quella più astratta, che in quella più pratica per l'effettiva realizzazione.
La scelta delle tecnologie e il modo in cui impiegarle e combinarle non sarà quindi estranea da tale osservazione, anzi, scegliere uno strumento da usare significa conoscere bene le sue possibilità e i suoi limiti. Di conseguenza, nel momento in cui si conosce il proprio obiettivo, diventa di cruciale importanza munirsi dei mezzi più appropriati.

2.2.2 - PHP: Hypertext Preprocessor

Per la realizzazione del progetto è stato utilizzato PHP (PHP: Hypertext Preprocessor), linguaggio di scripting interpretato con licenza *open source*, e più precisamente *Object Oriented PHP 5.x*. Questa scelta è stata principalmente manovrata dalle seguenti motivazioni:

- elevate performance
- elevata portabilità
- presenza di molte estensioni
- vasta libreria di funzioni
- grande disponibilità di classi, funzioni e applicazioni open source
- costi bassi

2.2.3 – Document Object Model (DOM)

Più noto semplicemente come DOM, questa è un'estensione che consente a PHP di operare in un certo modo sui documenti XML. Questo tramite l'API DOM con PHP 5. Per ulteriori approfondimenti si veda [B5, B6 e B7].
DOM è stato di sicuro lo strumento più importante adottato. Inoltre si è anche dimostrato la scelta giusta per riuscire nel lavoro. Infatti, per il modo in cui occorre qui lavorare con PHP per manipolare documenti XML, esiste infatti un'altra metodologia di *parsing*: il SAX (Simple API for XML). Inizialmente è stata tentata questa via, ma per le esigenze che si sono presentate durante l'evoluzione del lavoro non è più risultata efficiente e sufficiente.
Per cominciare a dare un'idea più precisa del tipo di approccio necessario, bisogna sapere che il SAX opera sulla base di un evento che si verifica, come "trovato nuovo tag", e richiede che questi tipi di eventi siano manipolati. In questo modo SAX è costretto ad operare analizzando riga per riga il documento in questione.
Il DOM, invece, carica l'intero documento XML in memoria ed è in grado di generare un "albero" rappresentativo dei dati XML. Questo albero può poi essere attraversato in una moltitudine di modi e si leggono quindi i singoli dati così organizzati.
Da notare, molto importante, c'è un altro aspetto fondamentale dal punto di vista della programmazione e che è intuibile dallo stesso acronimo (Document Object Model): usando il DOM per caricare il documento XML quest'ultimo diventa un *oggetto,* e come tale noi lo possiamo trattare mentre programmiamo usufruendo di tutti i vantaggi impliciti che la programmazione orientata agli oggetti porta con sé [B10].
Altri vantaggi del DOM sono:

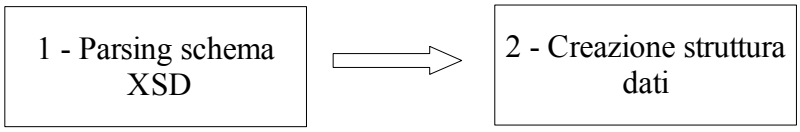
- alta velocità per piccoli documenti
- accesso a qualunque dato in qualunque momento
- semplice interfaccia PHP

2.2.4 - Combinare le tecnologie

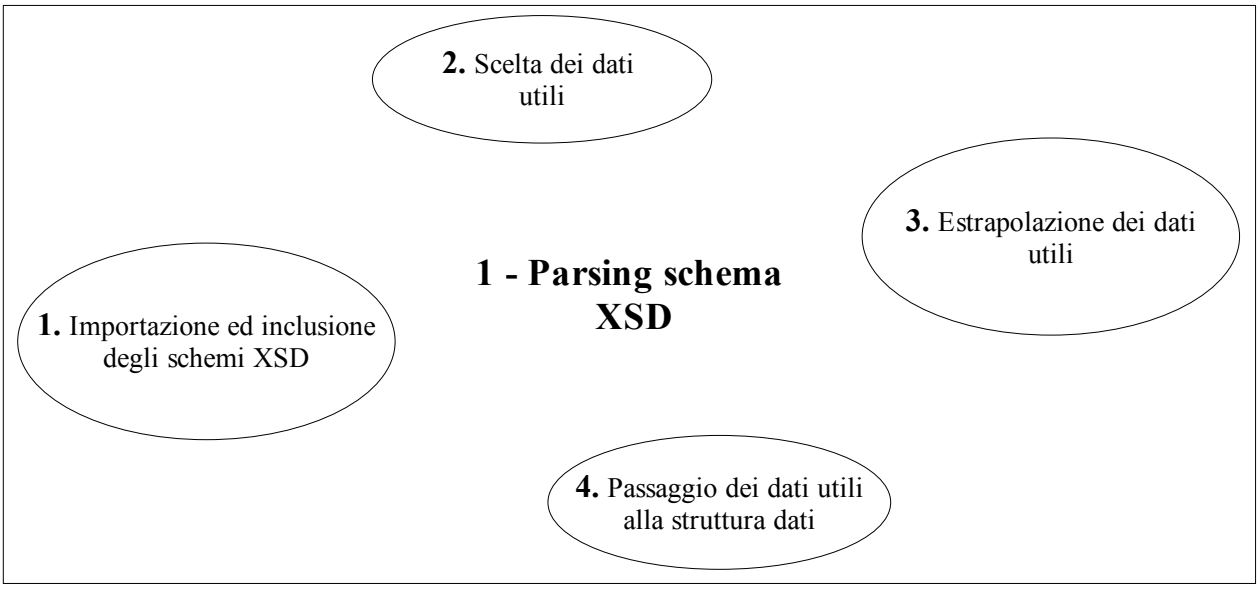
Si capisce dunque come grazie a questa estensione di PHP, sia ora facile riuscire ad interagire tra il linguaggio di programmazione e il documento XML.
All'interno dell'ambito DOM, come si può vedere in [B5, B6 e S1], PHP mette a disposizione

un'ottima libreria di classi, metodi e proprietà, che sono di fondamentale importanza per riuscire a sbizzarrirsi quanto più possibile in fase di programmazione. Sarà infatti il modo con cui verranno intrecciati tali a strumenti a determinare l'efficacia, l'economicità e la velocità dell'intero sistema.
Tutte queste potenzialità dipendono molto dalla vastità che la libreria di PHP in ambito DOM offre. Gli strumenti sono parecchi e permettono di avere il controllo su tutti i dati dell'XML (attributi, elementi, nodi, ecc.), tutto sta a preparare, nel caso di questo progetto, un terreno favorevole per poter reperire e avere a disposizione qualunque informazione e in qualunque momento presente nel documento XSD. Questo concetto sarà maggiormente chiarito nei capitoli successivi.

3 – Xsd2Gui



Giungere alla costruzione della struttura dati, si è già sottolineato come equivalga alla realizzazione del cuore dell'intero lavoro.
Per cominciare a descrivere meglio questa azione però, occorre specificare quali sono le linee guida da seguire. Il primo grosso capitolo di cui occuparsi è definire i parametri per regolarizzare l'operazione di *parsing*. All'interno di questa operazione si può distinguere un ciclo di passaggi che hanno portato alla realizzazione del parser stesso.

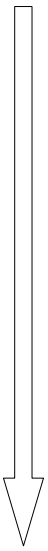


3.1 - Parsing schema XSD

Un *parser* lo possiamo immaginare come un lettore di testi, capace di riconoscere la sintassi di un linguaggio. Dato che viene utilizzata l'estensione DOM di PHP per leggere gli schemi XSD, il modo migliore di sfruttarlo è quello di dare determinate istruzioni al codice di programmazione

affinché il sistema possa operare riconoscendo i nodi dell'albero incontrati. In base a quello che sono si svolgeranno determinate operazioni.
Questo è infatti a grandi linee il principio adottato: costruire un parser per schemi XSD necessario per la costruzione di un'interfaccia grafica. Ecco un frammento di schema XSD riportato come esempio:

```
<xsd:complexType name="audio">
  <xsd:sequence>
      <xsd:element name="sequence_number" type="xsd:positiveInteger"/>
      <xsd:element name="nomenclature" type="xsd:string"/>
      <xsd:element name="proxies" type="audioproxy">
      <xsd:element name="note" type="xsd:string" minOccurs="0"/>
  </xsd:sequence>
  <xsd:attribute name="holdingsID" type="xsd:IDREF" use="optional"/>
  <xsd:attribute name="audiogroupID" type="xsd:IDREF" use="optional"/>
</xsd:complexType>
```



La grande freccia rappresenta il parser. Legge dall'inizio alla fine lo schema XSD e il compito è quello di istruirlo, attraverso la programmazione, a selezionare dall'albero costruito i nodi ritenuti importanti per andare a formare la struttura dati.

3.1.1 - XML e schemi XSD

Come già discusso in [B4, B7 e S3], XML (acronimo di eXtensible Markup Language) è un meta-linguaggio di markup cioè un linguaggio che permette di definire altri linguaggi di markup, non ha tag predefiniti e non serve per definire pagine Web né per programmare. Esso serve esclusivamente per definire altri linguaggi. È un insieme di specifiche che definiscono le modalità secondo cui è possibile crearsi un proprio linguaggio di markup.
Il *tag* è la cellula che permette di costruire questi documenti: è una parola, o meglio, una **keyword** che si associa ad un contenuto, ad esempio un immagine, un articolo o un video. Come già detto nella definizione precedente l'XML non ha tag predefiniti, essi possono essere scelti a seconda delle proprie esigenze.

Interessa capire però cosa c'è "dietro" questo tipo di documento e capire quali regole bisogna seguire per crearlo.
Ed è qui che entrano in gioco gli schemi XSD...
Come definito in [B8, B9, S2 e S3], uno schema XSD altro non è che un documento XML che utilizza un insieme di tag speciali per definire la struttura di un documento XML.
Significa che per definire la composizione di un documento XML che si vuole creare, è necessario seguire delle regole imposte da un altro documento XML. Ecco perché viene chiamato *schema*. Se devo scrivere un documento, l'ordine con cui dispongo i dati, le gerarchie con cui si presentano e quali o come si possono assegnare i valori agli elementi, sono tutte disposizioni date dallo schema XSD.
Infatti da questa definizione emerge un'altra regola importante: viene utilizzato un documento XML (lo schema XSD) per validare un altro documento XML [S3]. Quando un documento XML viene creato è infatti necessario verificare che esso sia corretto, ossia validarlo. La *validazione* serve proprio a ciò. Il documento XML una volta terminato viene confrontato (e quindi validato) nei confronti del documento XSD. Se nell'XML non si dovesse rispettare anche solo un unico punto di quanto imposto dallo schema XSD, esso non potrà essere ritenuto valido. Per approfondimenti si veda anche [B8 e B9].

Questi schemi sono per il presente lavoro il punto di partenza, il "luogo" dove poter cercare e selezionare quelle che saranno le informazioni di input per creare la struttura dati. Questi dati quindi sono effettivamente i *campi* che costituiranno l'XML finale. Estrapolarli e poterli presentare dallo schema in un'interfaccia grafica, significa permettere all'utente finale di dover solo assegnare il valore desiderato al campo specifico per creare il proprio documento.
Dato che lo schema porta con sé le informazioni di come si deve organizzare un documento XML, estrarre a partire da esso tali dati significa permettere di creare correttamente i documenti basati sul dato schema. L'utente non dovrà quindi preoccuparsi neanche di questo aspetto.

3.1.2 – Cosa estrapolare, perché conoscere il documento XML/XSD

Come già specificato precedentemente, la prima cosa da fare è capire quali informazioni dallo schema XSD occorrono per riuscire a creare la struttura dati, e permettere poi di formare correttamente il documento XML.

È stato necessario trascorre del tempo a studiare questi tipi di documento. In particolare occorreva analizzarli per studiare e raccogliere principalmente due informazioni: quali sono i tag principali che esistono, per questo si può servirsi anche di [S2], e come essi si possono comportare in determinate condizioni.

Conoscere quali e quanti tipi di tag esistono è utile non solo per conoscere a fondo tutti quelli esistenti e saper gestire la maggior parte dei casi possibile, ma anche per individuare quali effettivamente sono da considerarsi utili ai fini (la costruzione della struttura dati), quali invece si possono tranquillamente ignorare e, infine, quali necessitano di essere considerati per il loro significato o comportamento ma che non saranno comunque visualizzati nel risultato finale. Quest'ultimo punto si può usare per introdurre la spiegazione del perché bisogna conoscere il comportamento che i tag possono assumere. Tale discussione nei confronti di questo progetto è estremamente importante ed è stata infatti oggetto delle maggiori attenzioni dovute al lavoro.

Questa argomentazione si riflette nella pratica sulla tipologia di struttura che l'insieme totale dei tag dà all'intero schema XSD.

3.1.3 - Struttura “simil-matrioska”, struttura referenziata e l'attributo “type”

Scrivendo un documento XML qualunque, quindi anche uno schema XSD, i tag si distribuiscono disegnando una struttura gerarchica. Possiamo inizialmente vederli proprio come statuette a forma di matrioska in cui una contiene l'altra ma con una sostanziale differenza: mentre una statuetta vera può contenere al più un'altra statua e così via, il tag può contenere più tag contemporaneamente che vengono quindi a trovarsi sullo stesso livello (ecco perché la chiamo “simil-matrioska”). Possiamo pensarli come padri e/o figli di altri tag presenti; due o più di essi che si trovano sotto lo stesso padre e quindi sullo stesso livello potremmo quindi vederli come fratelli.

Ecco un breve esempio tratto da uno schema XSD:

```
<xsd:simpleType name="access_rights">          // livello 1
  <xsd:restriction base="xsd:integer">         // livello 2
    <xsd:enumeration value="0">                // livello 3
      <xsd:annotation>                         // livello 4
        <xsd:documentation>uso riservato...    // livello 5
        </xsd:documentation>                   // livello 5
      </xsd:annotation>                        // livello 4
    </xsd:enumeration>                         // livello 3
    <xsd:enumeration value="1">                // livello 3
      <xsd:annotation>                         // ...
        <xsd:documentation>uso pubblico
        </xsd:documentation>
      </xsd:annotation>
    </xsd:enumeration>
  </xsd:restriction>
</xsd:simpleType>
```

Quanto sottolineato sono i tag, più precisamente il loro nome. Ad ogni apertura di uno nuovo deve corrispondere la sua chiusura, sintatticamente distinguibile dal fatto che porta lo stesso nome del relativo tag di apertura ma preceduto dal carattere "/".

Si può quindi vedere che aprendo un tag, al suo interno è possibile scriverne altri prima di chiuderlo: è così che possiamo immaginare un padre che contiene uno o più figli, i quali possono essere padri a loro volta.

Adesso, per comodità nei confronti di questo progetto, definiamo una struttura "simil-matrioska" una struttura che presenta tutti i tag dall'inizio alla fine che si comportano solamente così. Ovvero in cui si susseguono padri e figli senza alcun tipo di diversificazione.

Dobbiamo tener conto che tutti i documenti XML per loro natura funzionano così.

Questa definizione di struttura "simil-matrioska" quindi ci serve non perché ne esiste un tipo diverso che vogliamo distinguere, ma perché dobbiamo differenziare quei casi in cui, per le caratteristiche che un tag può portare con sé, la stessa struttura del documento così lineare come è stata presentata può essere scombinata. Possiamo opporre quella che per il presente elaborato definiamo una struttura *referenziata*.

In questo caso i tag si presentano tutti distribuiti nel solito disegno gerarchico. Essi però possono ***fare riferimento*** (referenziarsi) ad un altro tag presente in un altro punto del documento, se non addirittura trovarsi in un altro schema. Ed è qui che ci si ricollega all'importanza di studiare il comportamento e il significato di questi elementi.

Ma come si può e cosa vuol dire ***fare riferimento*** ad un altro tag?

Per capire meglio questa situazione occorre specificare *cosa* può caratterizzare un tag. Prendiamone

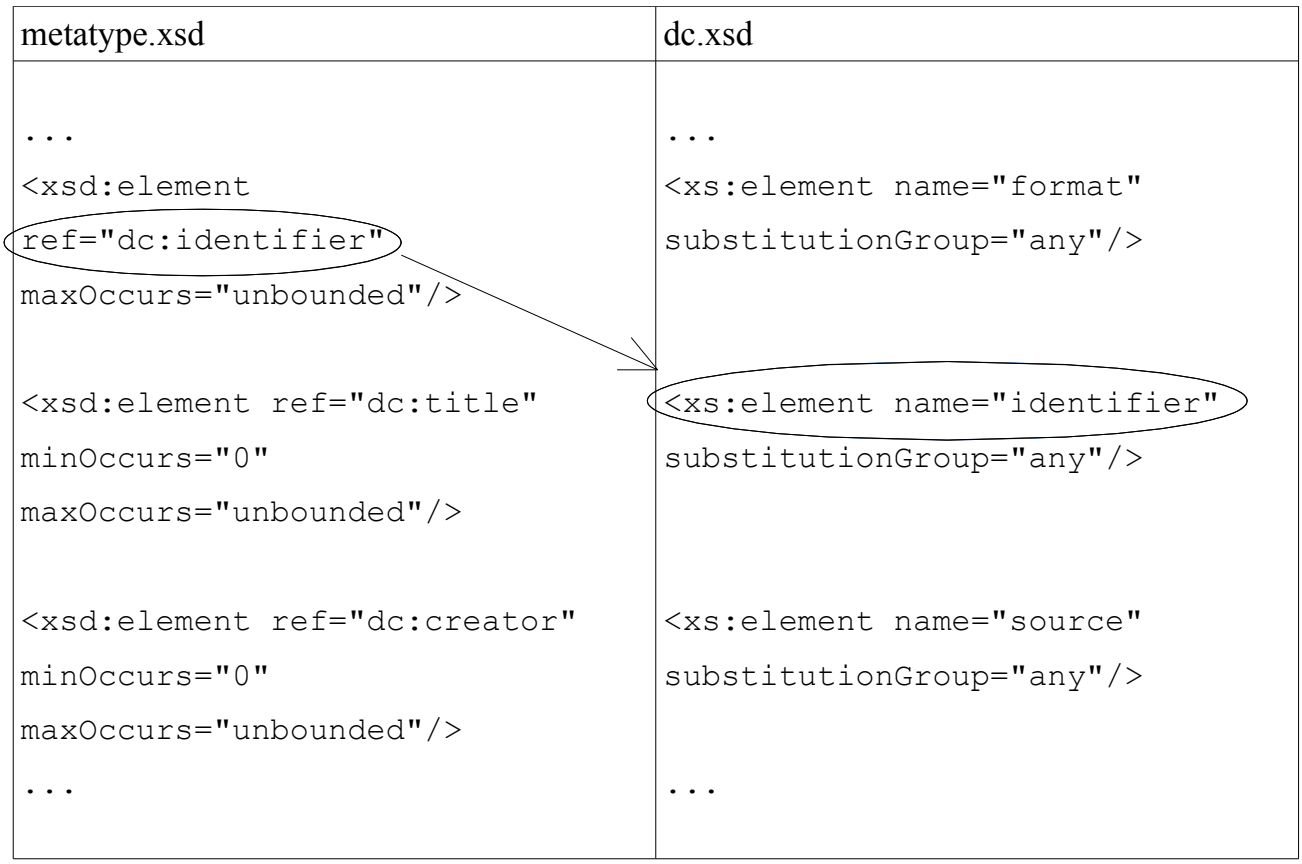
in esempio uno singolo:

```
<xsd:simpleType name="BICI">
</xsd:simpleType>
```

Quello che interessa sapere è che un tag è quasi sempre caratterizzato da uno o più ***attributi***.
Sono proprio questi che determinano la descrizione e, come tra poco si vedrà, il comportamento di un tag per specifici casi.
L'attributo si definisce come una coppia *nome-valore*, separata da un segno di uguale *(=)*, che può essere presente all'interno del tag di apertura di un elemento, dopo il nome dell'elemento stesso.
In questo piccolo esempio l'attributo è definito dal nome "name" a cui è associato il valore "BICI".
Adesso che è stato chiarito il concetto di attributo è possibile spiegare il significato dell'attributo ***REF***.
Esempio:

```
<xsd:element ref="dc:identifier" maxOccurs="unbounded"/>
<xsd:element ref="dc:title" minOccurs="0" maxOccurs="unbounded"/>
<xsd:element ref="dc:creator" minOccurs="0"/>
```

Quando si verifica uno scenario del genere ci si trova di fronte a quella che è stata chiamata una struttura referenziata, in cui, come accennato poco fa, il tag con tale attributo indica che in quel preciso punto dello schema XSD quell'elemento sta facendo riferimento ad un altro elemento. Più precisamente si sta riferendo a ciò che è specificato dal valore dello stesso attributo.
Significa che se si sta creando un documento XML sulla base di uno schema XSD con tale attributo all'interno di un tag, in quel preciso punto bisogna andare a vedere l'elemento a cui si sta riferendo. Sarà proprio quest'ultimo a dettare cosa si deve scrivere in quel preciso momento.



Come accennato prima il tag a cui si fa riferimento si può trovare sia in un altro punto del documento stesso, sia in un altro schema. È proprio questo il caso dell'esempio qui sopra riportato.
Per ora basta sapere che uno schema XSD è in grado di importare e di includere (più avanti sarà visto come sono due azioni con importanti differenze) altri schemi, come se si creasse una catena di fogli XSD tutti collegati tra loro e tutti necessari per la creazione dei relativi XML.
metatype.xsd e *dc.xsd* sono due schemi (due file) separati, in cui il primo importa il secondo. Si vede come il tag `<xsd:element ref="dc:identifier"...>` porta con sé l'attributo *ref*. Il relativo valore indica `"dc:identifier"`. Come si può capire dall'esempio qui sopra, "dc:identifier" è un elemento che risiede nello schema (nel file) *dc.xsd* mentre lo stiamo chiamando da *metatype.xsd*; quindi se si sta scrivendo un documento XML seguendo tale schema, quando si arriva di fronte a questo tag è necessario andare nell'altro schema (*dc.xsd*) per sapere come proseguire nella scrittura del documento, e poi per proseguire ritornare a quel punto in cui si era "interrotto". Notare che l'elemento chiamato dal primo file, che sarebbe appunto "identifier", è preceduto non a casa dal prefisso "dc:", che indica poi il Namespace, più avanti ne sarà specificato il significato, del file stesso in cui risiede effettivamente il dato in questione ("dc:identifier").

Questa notazione sui prefissi serve ad introdurre un altro scenario su un altro comportamento da

non sottovalutare: il valore che può assumere l'attributo “type” il quale serve ad indicare il tipo di dato dell'elemento cui si riferisce. Esistono dei tipi di dato che sono detti semplici o nativi, essi principalmente sono: *ID, string, decimal, integer, positiveInteger, boolean, date, time, dateTime, anyURI* e *unsignedLong*. Vengono chiamati semplici o nativi proprio perché sono praticamente i tipi più comuni, che rappresentano la maggior parte delle informazioni basilari.
Quello che interessa adesso far notare è come si definiscono in un XML: sono seguiti da un prefisso preciso, o “xsd:” o “xs:”. Breve esempio:

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

Però spesso si può imbattersi in casi come il seguente:

```
<xsd:element name="format"   type="niso:format"/>
<xsd:element name="scanning" type="niso:image_creation"/>
```

oppure come questo:

```
<xsd:element name="audio_metrics" type="audio_spatialmetrics"/>
<xsd:element name="format"        type="audio_format"/>
```

Il significato è semplice. Quando si trova, come nel primo caso, un prefisso che sia diverso da quello dei tipi semplici, esso sta specificando che il tipo di dato dell'elemento in questione lo si può trovare nel file indicato dal prefisso stesso, e in particolare è l'elemento indicato dopo di esso. Per seguire l'esempio riportato sopra, il *type* dell'elemento *format* è specificato nel file dell'XSD **niso** dall'elemento con nome “format” (questo si trova in **niso**). Nel secondo caso il meccanismo è lo stesso, cambia che non è precisato alcun prefisso e significa che semplicemente non si sta cercando un elemento presente su un altro file ma nel documento stesso in cui ci si trova.

Nel seguente capitolo è spiegato come la struttura referenziata vada gestita dal parser, premettendo quali sono le problematiche inerenti.

3.1.4 - Il problema della struttura referenziata. Perché una struttura dati?

Effettivamente l'unico grosso inconveniente dal punto di vista del parser che si introduce, è il fatto di perdere la linearità con cui si presentano i tag padri e figli annidati l'uno dentro l'altro. Questo è anche il motivo per cui bisogna costruire una struttura dati capace di riflettere la struttura dello schema XSD con cui si sta lavorando.
Come dovrebbe comportarsi il parser di fronte ad un *ref* o ad un *type* come nei casi riportati precedentemente? Come fa a memorizzare le posizioni e sapere quando considerare un elemento?
Se queste informazioni vengono salvate in una struttura dati, è possibile far muovere il parser da un punto all'altro dello schema XSD e anche da uno schema all'altro senza problemi. I dati utili man mano che il parser procede nella sua analisi vengono immagazzinati in modo ordinato e distribuito. Si rende necessaria però una caratteristica in più: la *ricorsività*. I metodi che costituiscono il parser sono sviluppati per generare tale ricorsività, la classe richiama se stessa generando una sequenza di oggetti che ha termine al verificarsi di una condizione particolare, la *condizione di terminazione* [S4].
Alla fine di tutto questa struttura dati sarà in grado di riflettere la struttura dello stesso schema XSD, con la differenza che sarà ordinata in un modo completamente diverso.

3.1.5 - Cosa estrapolare

La cosa necessaria da fare immediatamente è stata una scrematura di tutti i tipi ti tag che si possono incontrare scorrendo un XML.
È importante sottolineare quale documento è stato utilizzato per studiare e testare l'intero sistema. Dovendo essere questo applicabile a qualunque tipo di schema (per soddisfare il requisito di dinamicità richiesto) la scelta è ricaduta sul ***MAG*** (*metadati* gestionali-amministrativi), il Gruppo MAG ha prodotto uno schema XSD ed ha predisposto un set minimo di metadati gestionali al fine di una loro applicazione nei progetti di digitalizzazione. Questa scelta perché tale schema è distribuito su ben sette fogli e quindi ci si aspetta che al suo interno ci sia un gran numero diversificato di tag, inoltre introduce un'altra questione molto delicata quanto importante da gestire; ovvero l'importazione e l'inclusione di più schemi (anche per quanto discusso nel capitolo sulla struttura referenziata). I fogli XSD in questione sono:

- metadigit.xsd
- metatype.xsd
- audio-mag.xsd
- video-mad.xsd
- dc.xsd
- xlink.xsd
- niso.xsd

In questo modo si è creata la condizione di dover fronteggiare una casistica maggiore, mettendosi davanti alla possibilità di capire come gestire la maggior parte dei casi che si possono riscontrare.

Ma cosa sono quindi questi casi?

Sono appunto tutti i tag ritenuti informazioni utili e che una volta estrapolati saranno le *informazioni di input* per l'intero sistema, per i quali bisogna scegliere cosa fare:

Schema, Import, Include, Element, complexType, Attribute, attributeGroup, Choice, Sequence, simpleType, simpleContent, complexContent, Extension.

Questo è l'elenco dei tag utili. Dopo un'attenta analisi e per come viene gestita l'intera operazione di parsing, questa lista di tag mostra quali sono i dati utili utilizzati in questo progetto per arrivare alla struttura dati desiderata.

- **Schema**: questo tag definisce l'elemento *root* di uno schema XSD, cioè l'elemento radice, il primo della sorgente.

```
<xsd:schema
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:niso="http://www.niso.org/pdfs/DataDict.pdf"
    ...
targetNamespace="http://www.iccu.sbn.it/metaAG1.pdf"
elementFormDefault="qualified">
    ...
</xsd:schema>
```

- **Import**: viene utilizzato per aggiungere, più precisamente per importare, altri schemi XSD, i

quali però hanno un *Namespace* diverso da quello del documento stesso in cui si trova la sua chiamata.

```
<xsd:import namespace="http://purl.org/dc/elements/1.1/"
schemaLocation="./dc.xsd"/>
```

Ciò che si legge alla fine di questo valore (`schemaLocation="./dc.xsd"/`) indica il nome del file che sarà importato, in questo caso dal file *metatype.xsd* si sta importando lo schema *dc.xsd.*

- **Include**: viene utilizzato anch'esso per aggiungere altri schemi XSD, più precisamente per includerli, questa volta però il Namespace del nuovo schema è lo stesso di quello del documento in cui si trova la relativa chiamata.

```
<xsd:include schemaLocation="./audio-mag.xsd"/>
```

Si può notare, rispetto all'esempio precedente sull'Import, che è presente un unico attributo: "schemaLocation". Prima era necessario specificare anche il nuovo *Namespace.* *audio-mag.xsd* condivide lo stesso Namespace di *metatype.xsd* che è dove viene chiamata l'inclusione del nuovo schema.

- **Element**: definisce un elemento. Questo è il tag che descrive l'esistenza di un contenuto vero e proprio, quale ad esempio il nome o il cognome di un autore, un immagine, un video e così via.

```
<xsd:element name="file" type="link" minOccurs="0"/>
```

- **complexType**: determina quello che si chiama un "tipo complesso". Un elemento di tipo complesso è un elemento che *contiene* sicuramente altri elementi e/o altri attributi, oltreché ovviamente, eventuali restrizioni legate agli stessi.

```
<xsd:complexType>
 <xsd:sequence>
   <xsd:element name="library" type="xsd:string" minOccurs="0"/>
   <xsd:element name="inventory_number" type="xsd:string"/>
   <xsd:element name="shelfmark"minOccurs="0"...>
       <xsd:complexType mixed="true">
         <xsd:attribute name="type" type="xsd:string"/>
       </xsd:complexType>
   </xsd:element>
 </xsd:sequence>
 <xsd:attribute name="ID" type="xsd:ID"/>
</xsd:complexType>
```

- **Attribute**: definisce un attributo. Lo schema XSD si è detto che deve fornire tutte le specifiche per la creazione di un documento XML. In questo caso le specifiche riguardano quali sono e in che modo devono essere gli attributi dell'elemento a cui si riferiscono.

```
 ...
<xsd:attribute name="creation" type="xsd:dateTime" use="optional"/>
```

In questo esempio riportato, anche se non si vede, la definizione dell'attributo riguarda un complexType, quindi è legato a tutti gli elementi in esso contenuto. Sta specificando che a questi elementi è possibile conferire l'attributo "creation". Si dice che è possibile perché è specificato il campo *use="optional*", quindi non è obbligatorio usarlo. E questo attributo se usato dovrà avere formato dateTime (es. 11-06-1987).

- **attributrGroup**: viene utilizzato per raggruppare una serie di dichiarazioni di attributi, in modo che possano essere inseriti come gruppo all'interno delle definizioni di un tipo complesso.

```
<xs:attributeGroup name="personattr">
   <xs:attribute name="attr1" type="string"/>
   <xs:attribute name="attr2" type="integer"/>
</xs:attributeGroup>

<xs:complexType name="person">
   <xs:attributeGroup ref="personattr"/>
</xs:complexType>
```

- **Sequence**: appartiene alla famiglia delle restrizioni ed estensioni. Esso lo si trova dopo un elemento complesso e serve a dare una regola precisa per gli elementi presenti al suo interno: durante la scrittura del documento XML, se presente un *sequence* la presentazione degli elementi stessi deve rispettare *l'ordine di sequenza* con cui si presentano qui nello schema.

```
...
<xs:complexType>
  <xs:sequence>
    <xs:element name="firstname" type="xs:string"/>
    <xs:element name="lastname" type="xs:string"/>
     <xs:element name="address" type="xs:string"/>
    <xs:element name="city" type="xs:string"/>
    <xs:element name="country" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

Quando si compilerà il documento XML e quando il parser porterà gli elementi nella struttura dati, sarà necessario far scrivere in ordine: nome, cognome, indirizzo, città e paese.

- **Choice**: appartiene anch'esso alla famiglia delle restrizioni ed estensioni. Detto anche "elemento di scelta" si presente sempre dopo la dichiarazione di un tipo complesso. Data quindi una lista di elementi, esso consente, sempre al momento della scrittura del documento XML nonché condizione da rispettare nella costruzione della struttura dati, di poter usufruire e quindi di scegliere solamente uno degli elementi presenti nell'elenco.
  Lo possiamo trovare anche accoppiato ad un *sequence*. In questo caso la scelta sarà tra un blocco e l'altro di elementi compresi nel *sequence* stesso.

```
<xsd:complexType>
  <xsd:choice minOccurs="0">
    <xsd:sequence>                                          blocco 1
      <xsd:element name="year" type="xsd:string"/>
      <xsd:element name="issue" type="xsd:string"/>
      <xsd:element name="stpiece_per" type="SICI".../>
    </xsd:sequence>
    <xsd:sequence>                                          blocco 2
      <xsd:element name="part_number" type="..."/>
      <xsd:element name="part_name" type="xsd:string"/>
      <xsd:element name="stpiece_vol" type="BICI" minOccurs="0"/>
    </xsd:sequence>
  </xsd:choice>
</xsd:complexType>
```

- **simpleType**: determina quello che si chiama un "tipo semplice". Può specificare vincoli e informazioni sui valori degli attributi.

```
<xsd:simpleType name="access_rights">
  <xsd:restriction base="xsd:integer">
    <xsd:enumeration value="0">
      <xsd:annotation>
        ...
```

- **simpleContent**: questo tipo di tag non contiene elementi, ma ben si restrizioni o estensioni per i valori di tipo testo di un *complexType* o di un *simpleType*.

```
<xs:element name="shoesize">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:integer">
        <xs:attribute name="country" type="xs:string" />
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
```

In questo esempio viene dichiarato un *complexType,* “shoesize”, con il suo contenuto che dovrà essere di tipo *integer* e con l'attributo “country”.

- **complexContent**: determina anch'esso quelle che possono essere restrizioni o estensioni, ma solo per un complexType che contiene o contenuto di tipo misto o solo elementi.

```
<xs:element name="employee" type="fullpersoninfo"/>

<xs:complexType name="personinfo">
   <xs:sequence>
      <xs:element name="firstname" type="xs:string"/>
      <xs:element name="lastname" type="xs:string"/>
   </xs:sequence>
</xs:complexType>


<xs:complexType name="fullpersoninfo">
   <xs:complexContent>
      <xs:extension base="personinfo">
         <xs:sequence>
            <xs:element name="address" type="xs:string"/>
            <xs:element name="city" type="xs:string"/>
              <xs:element name="country" type="xs:string"/>
         </xs:sequence>
      </xs:extension>
   </xs:complexContent>
</xs:complexType>
```

- **Extension**: serve a descrivere le estensioni per gli elementi di tipo complesso o di tipo semplice.

```
<xs:simpleType name="size">
   <xs:restriction base="xs:string">
      <xs:enumeration value="small" />
      <xs:enumeration value="medium" />
      <xs:enumeration value="large" />
   </xs:restriction>
</xs:simpleType>
```

```
<xs:complexType name="jeans">
  <xs:simpleContent>
    <xs:extension base="size">
      <xs:attribute name="sex">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="male" />
              ...
```

In questo caso per l'elemento complesso “jeans” viene effettuata un'estensione con *simpleContent* che richiama le restrizioni dell'elemento semplice “size”.

I tipi di tag che non appaiono all'interno di questo elenco, fanno parte di quelli che si può permettersi di ignorare dal punto di vista delle operazioni da compiere per ognuno. Tutti i tag tranne `<xsd:documentation>` e `<xsd:annotation>` sono informazioni da considerare per una corretta lettura dello schema, ma questi invece descritti nelle pagine precedenti sono quelli gestiti per creare le informazioni di input da inserire nella struttura dati.
I comportamenti descritti, quali la struttura referenziata piuttosto che le varie restrizioni, sono tutti condizioni da rispettare affinché si possa permettere la creazione di documenti XML corretti.

3.1.6 - Istanziazione delle classi

Con la versione 5 di PHP è stato completamente rivoluzionato il supporto alla programmazione ad oggetti, si veda anche [B10]. Nella versione 4 era possibile programmare ad oggetti, ma le enormi possibilità della classica programmazione ad oggetti erano fortemente limitate dalla mancanza di un supporto adeguato da parte del linguaggio. Questo perché il supporto alla programmazione ad oggetti è entrato a far parte di PHP come qualcosa di aggiuntivo, non nato assieme al linguaggio, che era stato inizialmente pensato per essere solamente procedurale [S3].
Le classi, nella programmazione ad oggetti, sono un elemento fondamentale. Come è possibile vedere in [B1 e B2] esse possono essere definite come una collezione di **proprietà** e **metodi** che utilizzano quanto definito all'interno della classe stessa per svolgere determinate azioni.

- **proprietà**: è la definizione nella programmazione ad oggetti di quella che normalmente si indica come variabile, una porzione di memoria destinata a contenere dei dati.
- **metodi**: è la definizione nella programmazione ad oggetti di quella che si chiama *funzione*, ossia una combinazione di operazioni logiche da compiere con i valori delle variabili.

La differenza è che nella programmazione ad oggetti le proprietà ed i metodi di una classe appartengono esclusivamente a questa classe ed eventualmente alle classe che ereditano [S3].

La classe la si può immaginare come un contenitore nel quale mettere tutti i dati e tutte le azioni che con essi si desidera svolgere.

Ecco un esempio di codice semplificato per definire una classe in PHP:

```
<?php
  class Pluto
  {
    $elementoXSD = '...'; // DEFINIZIONE DI UNA PROPRIETÀ
    ...
    function Programmare ($propr.1, $propr.2, ...) // DEFINIZIONE METODI
      {
         ... // OPERAZIONI DEL METODO
      }
  }
?>
```

Questa che si vede è proprio la sintassi da adottare per definire una classe.
Una possibilità importantissima permessa dalle classi è quella di crearne una che sia un'estensione di un'altra, che sarà quindi dichiarata in un altro punto del codice:

```
<?php
  class Paperino extends Pluto
  {
   $informazione = '...';  // DEFINIZIONE PROPRIETÀ LOCALE
   $informazione   =   $this->elementoXSD;   /*   LA      PROPRIETÀ   LOCALE
      "$INFORMAZIONE" VIENE EGUAGLIATA AL VALORE DI UNA PROPRIETÀ   CHE SI
      TROVA NELLA      CLASSE "PLUTO", "elementoXSD" */
   $this->Programmare($informazione) { /* METODO DELLA CLASSE PLUTO AL QUALE
      PASSO COME PARAMETRO LA PROPRIETÀ APPENA DEFINITA "$INFORMAZIONE".
      QUEST'ULTIMA ORA SARÀ QUANTO PROCESSATO DAL METODO "PROGRAMMARE" */

      ... // QUI SI SCRIVONO LE OPERAZIONI DEL METODO
     }
  }
?>
```

L'effetto pratico è che la classe estesa erediterà il contenuto della classe genitore. Ora la classe ***Paperino*** potrà utilizzare le proprietà e i metodi della classe ***Pluto***, e lo farà grazie alla *keyword* "$this->".

Dal punto di vista della programmazione è un enorme vantaggio usare le classi in questo modo, permette infatti di tener separate le informazioni a proprio piacimento ma, contemporaneamente, di incrociarle all'occasione giusta.

Questa breve introduzione alle classi serve ovviamente per spiegare che ruolo hanno nel presente progetto.

Dato che la classe è una collezione di proprietà e metodi e dato che ci sono da gestire una serie di tag XML, l'approccio è stato quello di unire le due cose: **istanziare per ogni tag ritenuto utile una propria classe**. Questa scelta è stata operata per i vantaggi propri nell'utilizzo delle classi e delle classi estese: ogni tag richiede di essere gestito in modo indipendente ma è necessario prevedere delle operazioni comuni a tutti, delle operazioni comuni a tutti i tag.

La risposta possibile per fronteggiare queste esigenze è la seguente:

```
   class Xsd2GuiTag{
      ...
    }
```

Questa è la classe principale del progetto (non a caso si chiama Xsd2Gui**Tag**). Per ogni tag

dell'elenco riportato al capitolo precedente sarà istanziata una propria classe, la quale però, sarà un'estensione di quella principale:

```
class Xsd2Gui Schema extends Xsd2GuiTag{...}

class Xsd2Gui Import extends Xsd2GuiTag{...}

class Xsd2Gui Element extends Xsd2GuiTag{...}

 etc etc...
```

In questo modo all'interno della classe principale si possono raccogliere i metodi e le proprietà che interessa condividere con tutte le altre classi estese a partire da questa. All'occorrenza da ogni classe relativa ad un tag si potrà richiamare lo stesso metodo o la stessa proprietà senza doverlo riscrivere ogni volta. Ecco che si separa quanto serve avere a disposizione sempre da quanto invece serve gestire in modo indipendente.
Per rispecchiare ancor più questo concetto il codice di programmazione è stato scritto su tre file diversi:

- ***main.php*** che è fondamentalmente lo start del programma, dove viene stabilito qual'è lo schema XSD di partenza e poi passato ai vari metodi;
- ***DOMXsd2GUIBaseLib.php*** è dove risiede la classe appena discussa `class Xsd2GuiTag{...}` e altre, quindi è la libreria dove si trovano metodi e proprietà da avere sempre a disposizione per l'uso (non a caso il file è stato nominato con la parola *BaseLib*).
- Infine ***DOMXsd2GUILib.php*** è la libreria nella quale vengono istanziate tutte le classi relative ai tag più altre, e quindi dove si trovano metodi e proprietà locali ossia relativi alla gestione del singolo elemento in questione.

Un'altra classe estremamente importante creata e che si trova in *DOMXsd2GUIBaseLib.php* è `class Xsd2GuiStatics{...}`. In essa sono state raccolte e quindi dichiarate le proprietà statiche, ossia quelle proprietà che si vogliono rendere visibili a qualunque oggetto e per l'intera esecuzione del programma. Per distinguerla nella definizione va preceduta dalla *keyword* **static**.

```
static public $debug = FALSE;
```

Per richiamare da un punto esterno alla classe stessa in cui si trova questo tipo di variabile è

necessario utilizzare un prefisso particolare. Esso è costituito dal nome della classe in cui si trova la variabile seguito dal doppio ":".

```
Xsd2GuiStatics::$schemaCounter;
```

Per le proprietà e per i metodi scritti all'interno di una classe bisogna poi specificare un'altra importante caratteristica: la loro ***visibilità.*** La visibilità di una proprietà o di un metodo può essere di tipo *public*, *protected* o *private*, e serve a definire fondamentalmente la loro accessibilità. Se è tipo *public* sarà visibile, e quindi accessibile, da qualunque punto del codice, se *protected* sia dalla classe stessa in cui si trova che da quelle estese dalla stessa, se *private* invece solo dalla classe che la definisce. Per la definizione è necessario anteporre la giusta *keyword*.

```
public    $variabile_public = Array();
protected function metodo_protected(){...}
private   function metodo_private(){...}
```

Ora si può vedere come è stata programmata l'istanziazione delle classi. Più avanti si vedrà precisamente in quale punto del programma si colloca questo frammento di codice:

```
foreach($node->childNodes as $child){
   if($child->nodeType == XML_ELEMENT_NODE){
      $className = $this->_nameBuilder($child->tagName);
      if(class_exists($className, false)){
         new $className($child);
      }
   }
}
```

Il concetto è verificare per ogni nodo dell'albero DOM creato, se è presenta la relativa classe per la sua gestione nel codice scritto.
Dato un nodo (`$node`) si analizza uno ad uno ognuno dei suoi figli: `$node->childNodes as $child`. Quello che c'è da notare è che per ogni nodo si controlla se ha figli, così facendo siamo in grado di controllare tutti i nodi che costituiscono l'albero senza tralasciarne.
Il ciclo **foreach** passa nella proprietà **$child** ogni figlio e li fa analizzare tutti uno per volta attraverso le istruzioni che contiene. Per ogni nodo figlio viene prima verificato che sia realmente

un elemento XML `if($child->nodeType == XML_ELEMENT_NODE)`. Successivamente il nome del tag del singolo nodo viene passato al metodo `_nameBuilder`, egli restituisce una stringa che sarà il nome della classe. Ecco un esempio: se incontriamo il tag `<xsd:element...>` la scritta "xsd:element" viene passata al metodo `_namebuilder`. Esso costruisce questa stringa con due blocchi, nel primo inserisce il valore di una proprietà protetta che è sempre lo stesso (`$_classPrefix = 'Xsd2Gui'`) al quale concatena il secondo, il nome del tag privato del suo prefisso. Quindi da "xsd:element" si ottiene solo "element". Il risultato finale è "Xsd2GuiElement". Infine si chiede al codice di verificare se a questa stringa creata corrisponde l'esistenza di una classe e, se verificato, viene quindi istanziato un oggetto da essa.
Il metodo `_namebuilder` lavora grazie anche ad un espressione regolare, è necessaria per separare il prefisso (*xsd:*) dal nome del tag (*element*):

```
protected function _nameBuilder($nodeName){
 return $this->_classPrefix.(ucfirst
        (strtolower(preg_replace('/^xs.?:/i','', $nodeName))));
}
```

Come detto poco fa questa azione di controllo delle classi viene eseguita per ogni tag, quindi siamo in grado di scegliere e di controllare tutti i tag ritenuti utili tramite la creazione della relativa classe. I tag come `<xsd:documentation>` e `<xsd:annotation>` non sono importanti ai nostri fini, ecco quindi che non preparando alcuna classe per loro (es. Xsd2GuiDocumentation) si può ignorarli senza che interferiscano con il programma.
Notare che tutte queste classi relative ai tag risiedono appunto nel file *DOMXsd2GUILib.php*, quindi nella libreria dove sono presenti proprietà e metodi che riguardano la gestione della singola classe in questione.
Il metodo per la costruzione del nome della classe, così come la proprietà protetta e il ciclo *for* sopra indicati, risiedono invece nella libreria-base. Sono operazioni da compiere per ogni nodo che si presenta.

3.1.7 - Namespace, import e include

I *Namespace* vengono utilizzati per fornire ad un documento XML in modo univoco i nomi degli elementi e degli attributi da utilizzare. Rimanendo come esempio sul ***MAG***: dallo schema

metatype.xsd ne vengono importati altri tre (*dc.xsd, xlink.xsd e niso.xsd*). Sappiamo che quando si è di fronte ad un import è perché si sta coinvolgendo uno schema che utilizza un Namespace differente: il Namespace di *metatype.xsd* è differente da quello di dc.xsd, che è diverso dagli altri due e così via. Questo significa che i nomi degli elementi e degli attributi da utilizzare potranno anche ripetersi tra di loro in modo uguale tra i due schemi XSD, ma questo non creerà confusione in quanto fanno riferimento a due *Namespaces* differenti. È come creare dei vocabolari a cui fare riferimento per vedere se una parola esiste ed è di conseguenza utilizzabile. Essi sono definiti in [S2].

Vengono dichiarati usando l'attributo XML riservato *xmlns* all'interno del tag "schema", che si sa già come definisca l'elemento *root* di uno schema XSD.

```
<xsd:schema
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:niso="http://www.niso.org/pdfs/DataDict.pdf"
xmlns:xlink="http://www.w3.org/TR/xlink"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns="http://www.iccu.sbn.it/metaAG1.pdf"
xmlns:dc="http://purl.org/dc/elements/1.1/"
targetNamespace="http://www.iccu.sbn.it/metaAG1.pdf"
elementFormDefault="qualified">
 ...
</xsd:schema>
```

In questo modo quando gli altri schemi verranno importati il loro Namespace sarà già dichiarato.

Come già spiegato in [B4 e S3] anche se in teoria sarebbe possibile utilizzare qualsiasi identificatore per un Namespace, sarebbe opportuno utilizzare nomi univoci che evitino a loro volta omonimie tra Namespace. Per questo motivo **i Namespace vengono identificati tramite URI** (Uniform Resource Identifier); in particolare è prassi comune utilizzare un URL (che è un tipo particolare di URI) come identificatore di un Namespace, in modo che ci sia una corrispondenza biunivoca tra chi definisce un Namespace ed il rispettivo nome a dominio. L'uso di un URL come identificatore di Namespace può creare qualche problema di leggibilità, soprattutto quando vengono combinati insieme diversi linguaggi di markup. In questi casi è opportuno utilizzare dei prefissi come abbreviazione degli identificatori di Namespace.

```
xmlns:niso  = "http://www.niso.org/pdfs/DataDict.pdf"
xmlns:xlink = "http://www.w3.org/TR/xlink"
```

L'attributo `schemaLocation` del Namespace, come ad esempio quello per il file *xlink* `"http://www.w3.org/TR/xlink",` ci consente di associare un Namespace al rispettivo XML Schema (o schema XSD).

```
<xsd:import namespace = "http://www.w3.org/TR/xlink"
            schemaLocation = "./xlink.xsd"/>
```

Naturalmente in un documento XML si può soltanto fare riferimento ad un Namespace. Il Namespace vero e proprio viene definito nel relativo schema XSD che definisce la grammatica di un linguaggio XML-based. Per definire un Namespace occorre utilizzare l'attributo targetNamespace come nel seguente esempio:

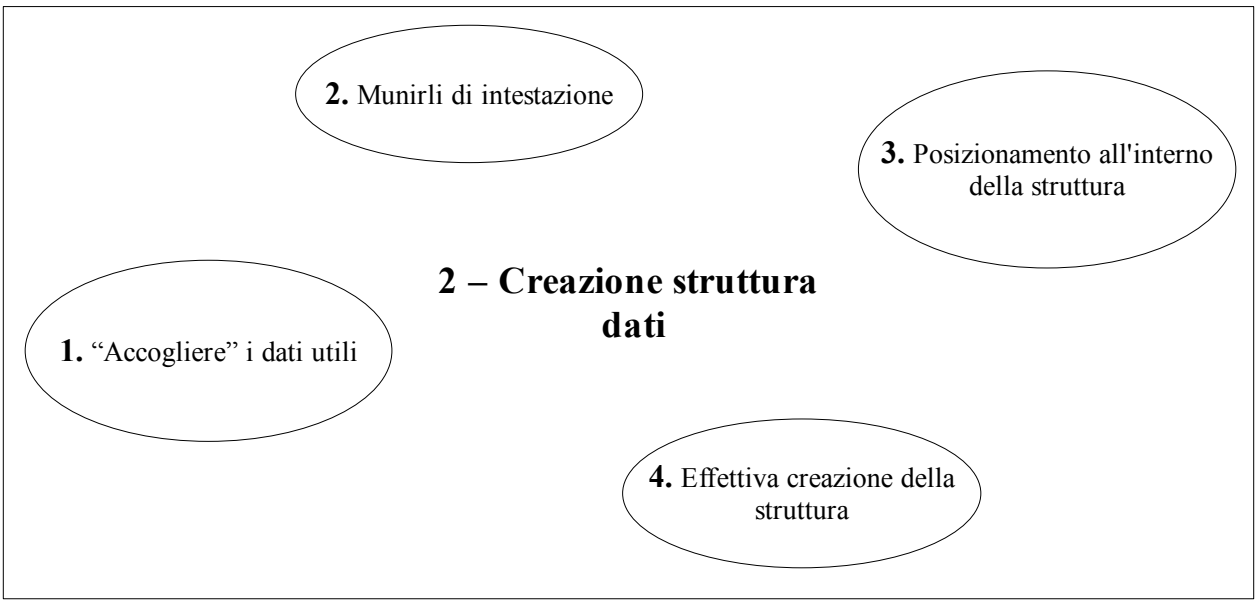
```
<xsd:schema xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      ...
      targetNamespace="http://www.iccu.sbn.it/metaAG1.pdf" ...>
```

In questo modo si specifica che i tag definiti nello schema corrente costituiranno gli elementi del Namespace indicato.

Adesso è giunto il momento di spiegare come tutti questi elementi si collocano nel presente progetto: occorre che il parser sia capace di gestire e riconoscere tutte queste situazioni e comportamenti, di manovrare le informazioni di input e disporle in modo ordinato.

3.2 - Creazione struttura dati

Anche in questo caso, come per il capitolo precedente, è possibile delineare un ciclo di passaggi che riassume le linee guida da seguire per capire cosa coinvolge la realizzazione della struttura dati.
Nel corso dei primi passaggi è come essere in una sorta di confine tra il lavoro svolto dal parser, in cui si leggono le informazioni e si selezionano, e il lavoro di costruzione della struttura, dove bisogna ordinare in modo intelligente quanto raccolto. Questo per dire che non si vedrà sin dall'inizio come si costruisce, ma si vedrà anche come si preparano attraverso l'incrocio di una serie di metodi le informazioni da collocare poi all'interno di tale struttura.



3.2.1 - Un Super-Array

Come definito in [B2 e S1] un array si descrive come una mappa ordinata, ossia una mappa capace di immagazzinare al suo interno un numero indefinito di valori che possono essere di diverso tipo tra di loro e tutti associati ad una chiave capace di identificarli. Esso inoltre può essere multidimensionale, nel senso che all'interno di un array è possibile inserire come dato un altro array, dentro il quale è possibile aggiungerne altri e così via.
Proprio questo tipo di "locazione" è stata adottata come impalcatura della struttura dati, un array multidimensionale. Diventa un "super-array" perché le informazioni da memorizzare sono molte, e ora della fine saranno diversi gli array memorizzati dentro altri array.
Un piccolo esempio:

```
[Array 1] // array principale
  (
    [chiave elemento 1] => elemento 1 // elemento dell'array 1
    [chiave elemento 2] => elemento 2 // elemento dell'array 1
    [chiave elemento 3] => Array 2    // array 2 dentro un array 1
       (
        [chiave elemento 1] => elemento 1 // elemento dell'array 2
        [chiave elemento 2] => elemento 2 // elemento dell'array 2
        [chiave elemento 3] => elemento 3 // elemento dell'array 2
       )
  )
```

Qualcuno potrebbe chiedersi come mai scegliere di usare gli array piuttosto che delle liste o altri sistemi ancora. La risposta può essere data dallo stesso PHP: per questo linguaggio di

programmazione manipolare tale tipo di dato risulta tanto economico quanto efficace. Inoltre, come d'altronde funziona per sua natura l'ambiente PHP, si dispone di un gran numero di funzioni che permettono di intervenire in ogni minimo dettaglio e di manovrare con molte soluzioni i valori contenuti.
Questo tipo di approccio multidimensionale è stato realmente adottato per descrivere ogni singola parte. Tutta la struttura fino ad arrivare ai singoli elementi saranno infatti rappresentati all'interno da altri array ancora, ognuno dei quali, a seconda di che elemento sia, porterà con sé tutti gli attributi che lo descrivono e delle informazioni preziose per la loro identificazione. Si vedrà come l'indirizzo o la strada per raggiungere ognuno di essi sarà data dalla lettura di array in array.

3.2.2 – 'main.php' e le librerie

Se diamo uno sguardo al file *main.php* si può vedere, non a caso, che la prima classe ad essere chiamata in causa ed il parametro che gli viene passato sono rispettivamente `class Xsd2GuiSchema extends Xsd2GuiTag`, quindi la classe istanziata per il tag riguardante l'elemento *root* dello schema, e '`metadigit.xsd`' che è il primo schema XSD da lanciare.

```
<?php
   require_once('DOMXsd2GUIBaseLib.php');
   require_once('DOMXsd2GUILib.php');
   print '<pre>';
   $schema = new Xsd2GuiSchema('metadigit.xsd');
   ...
```

Grazie al comando `require_once` è possibile coinvolgere gli altri due file, quindi le librerie create. La cosa importante da notare ora è che nella dichiarazione della proprietà `$schema` si sta istanziando la prima classe manualmente, `Xsd2GuiSchema`, alla quale tramite il valore indicato tra le parentesi si passa il file indicato. In questo caso `metadigit.xsd` è la radice dei sette fogli XSD, quella da cui vengono poi via via importati ed inclusi gli altri schemi.
Quello che è interessante vedere adesso è per prima cosa quanto succede all'interno della classe `Xsd2GuiSchema`, cioè vedere quali sono effettivamente le prime operazioni compiute da questo programma, e successivamente le classi relative ai tag *Import* ed *Include.*
Identifichiamo prima di tutto quali sono i metodi che caratterizzano `Xsd2GuiSchema` in questa

fase iniziale: `schemaConnect()`, `versionConnect()`, `domBuilder()`, `buildNameSpaces()`, `digger()`, `lostRecover()`, `schemaDisconnect()`, `versionDisconnect()`. Al fianco di questi metodi si ricorda la presenza delle variabili `protected $_localSchema` e `protected $_localVersion`.
Come prima illustrazione si può prendere il frammento di codice a riguardo. Da qui è possibile osservare quelle che sono le chiamate ai metodi e quale parametro viene ad essi passato.

```
public function __construct($xsdPath){
  if(empty(Xsd2GuiStatics::$absolutePath)){
    Xsd2GuiStatics::$absolutePath = dirname($xsdPath);
  }
  if(!Xsd2GuiStatics::$include){
    $this->_schemaConnect();
    $this->_versionConnect($xsdPath);
  }
  $root = $this->_domBuilder($xsdPath);
  $this->_buildNameSpaces($xsdPath);
  $this->_digger($root);
  if(!Xsd2GuiStatics::$include){
    $this->_lostRecover();
    $this->_schemaDisconnect();
    $this->_versionDisconnect();
  }
}
```

I metodi evidenziati sono quelli che permettono di raccogliere le primissime informazioni utili da inserire nel super-array.
Bisogna prima capire come organizzare le informazioni raccolte affinché sia possibile averle ordinate.

Gestione Namespace

La prima cosa da fare è appunto gestire i Namespaces, che come si è visto sono il vocabolario a cui ciascun elemento fa riferimento. È necessario che la struttura dati rifletta questa condizione.
Per questo motivo il primo livello del super-array è dettato dalla separazione dei sette fogli XSD coinvolti: per ogni foglio c'è un proprio array, all'interno di questi si troveranno gli array dei singoli

elementi con le informazioni giuste nei casi di struttura referenziata. Si ricorda come la struttura dati è definibile corretta e ordinata solo quando ogni dato sarà reperibile e identificabile.
I primi metodi in questione per la gestione di questa fase si chiamano `_schemaConnect()` e `_versionConnect($xsdPath)`, e si occupano in modo automatico di riconoscere e creare queste sezioni per i vari schemi.
Il funzionamento del primo è molto semplice: il concetto è che per ogni nuovo schema, cioè quando il parser incontra il tag "schema" e viene istanziata la relativa classe, viene banalmente incrementato un valore numerico presente in una proprietà, il quale sarà quindi univocamente rappresentativo dello schema attuale in cui ci si trova.

```
protected function _schemaConnect(){
 $this->_localSchema = Xsd2GuiStatics::$actualSchema;
 Xsd2GuiStatics::$schemaCounter++;
 Xsd2GuiStatics::$actualSchema = Xsd2GuiStatics::$schemaCounter;
}
```

La proprietà statica presente nella libreria *base* (`Xsd2GuiStatics::$actualSchema`) è già settata al valore 0, mentre la proprietà locale è quella che identifica il valore rappresentativo dello schema "locale" ossia quello in cui ci si trova, infatti si chiama `_localSchema`. Quest'ultima viene inizialmente eguagliata a `Xsd2GuiStatics::$actualSchema`. Successivamente però un'altra proprietà statica viene incrementata di 1 ovvero quella che si occupa di "tenere il conto" per ogni nuovo schema che si incontra, `Xsd2GuiStatics::$schemaCounter.` A questo punto la prima proprietà statica viene eguagliata al valore incrementato così da ottenere il nuovo identificativo per il prossimo nuovo schema, dato che la proprietà locale `_localSchema` sarà eguagliata a tale valore incrementato per il prossimo schema.
Il secondo metodo, `_versionConnect($xsdPath),` è invece utilizzato per creare un livello ancora più forte di univocità identificativa per ogni schema. Il risultato ottenuto ci permette di avere un valore che usiamo per definire la "versione" dello schema. Occorre però che tale valore sia davvero unico per ogni schema, perché si potrebbe avere versioni differenti del file dello stesso schema.
A tale scopo si sfrutta l'algoritmo crittografico di hashing MD5, per il quale PHP fornisce già una funzione che lo calcola, applicandolo direttamente al file dello schema XSD. Quando il metodo viene chiamato infatti si può vedere come tra le parentesi è specificato un parametro, la variabile `$xsdPath.` Con questa scrittura stiamo passando al metodo il path del file dello schema XSD.

L'algoritmo MD5 viene usato per soddisfare questo livello di univocità in quanto prende in input una stringa di lunghezza arbitraria e ne restituisce in uscita un'altra con lunghezza fissa di 32 valori esadecimali.

```
protected function _versionConnect($xsdPath){
  $this->_localVersion = Xsd2GuiStatics::$actualVersion;
  Xsd2GuiStatics::$actualVersion = md5_file($xsdPath);
}
```

In questo modo il valore contenuto nella proprietà statica `Xsd2GuiStatics::$actualVersion` viene usato per indicare la versione dello schema.

Creazione dell'*oggetto* DOM

Il prossimo metodo è quello che permette di utilizzare il "Document Object Model". Anche ad esso viene passato il path del file dello schema XSD e dato questo, grazie al comando `new domDocument()`, si rende il documento in questione un oggetto DOM. Viene anche effettuato un controllo sul corretto caricamento del file, se non dovesse essere soddisfatto verrà restituito e visualizzato un messaggio di errore.

```
protected function _domBuilder($xsdPath){
  $dom = new domDocument();
  if(@$dom->load($xsdPath) === FALSE){
     throw new Xsd2GuiException('XSD not loaded: '.$xsdPath);
  }
  return $dom->documentElement;
}
```

Namespace URI, import ed include

Bisogna finire di capire cosa vuol dire gestire i Namespace, questo perché quanto visto nel paragrafo precedente "Gestione dei Namespace" si spiegava solo come nella struttura dati vengano effettivamente posizionati. Viene costruito a parte, parallelamente alla struttura dati stessa, un array in cui si raccolgono le informazioni utili alla loro descrizione. Questo lo si può immaginare come un *array-contenitore* in cui diventa semplicissimo reperire le informazioni che si scrivono al suo

interno.

Questo array-contenitore è `Xsd2GuiStatics::$nameSpacesURI` e il metodo che coinvolge la sua creazione è **_buildNameSpaces**.

Inizialmente questo metodo grazie alle espressioni regolari distingue quali degli *attributeName* (nome degli attributi) del tag "Schema" indicano Namespaces di schemi che saranno importati o inclusi. Quando li trova comincia a costruire `nameSpacesURI`.

```
protected function _buildNameSpaces($xsdPath){
   $matches   = Array();
   $xsdString = file_get_contents($xsdPath);
   $xsdString = preg_replace('/<!--.*?-->/s', '', $xsdString);
   preg_match_all('/xmlns:[a-zA-Z]*=[\'"].*?[\'"]/', $xsdString,
                  $matches);
   unset($xsdString);
   foreach($matches[0] as $match){
     $arrayS    = explode('=', $match);
     $tag       = explode(':', $arrayS[0]);
     $arrayS[1] = preg_replace('[\'"]', '', $arrayS[1]);
     if($tag[1] !== 'xsi' && $tag[1] !== 'xs' && $tag[1] !==
                   'xsd'){
       Xsd2GuiStatics::$nameSpacesURI[Xsd2GuiStatics::$actualSchema]['ns']
                     [$tag[1]] = preg_replace('/[\'"]/', '', $arrayS[1]);
     }
   }
```

Quello che accade è che grazie alla funzione `file_get_contents` è possibile analizzare il contenuto del file indicato. Successivamente, come già detto, entrano in gioco le espressioni regolari che operano sul nome degli attributi di "Schema", selezionando dove il prefisso comprende la stringa "xmlns".

Quello che avviene nel ciclo *foreach* è che per ogni *attributeName* selezionato precedentemente viene memorizzato ciò che si trova dopo il carattere ":" nella stringa. Se ad esempio la stringa su cui si sta lavorando è "xmlns:dc" viene salvato solo "dc". A questo punto la costruzione di *Namespace URI* viene inizializzata solo se la stringa è diversa da "xsi", "xs" o "xsd", che non indicano l'importazione o l'inclusione di nuovi schemi XSD.

Namespace URI è costituito da tanti array quanti Namespace presenti tra gli schemi XSD. Ognuno di essi poi contiene altri due array: uno che descriva in quello schema quanti Namespace sono

presenti, questo avrà il nome "*ns*", l'altro, molto importante, si chiama "*array_path*" e serve a dare un indirizzo al singolo schema. Questo valore viene fornito principalmente dalla coppia di valori *actualSchema* e *actualVersion*.

```
if(!Xsd2GuiStatics::$include){
   if(Xsd2GuiStatics::$actualNameSpace == 'default'){
      $localSchema = Xsd2GuiStatics::$actualSchema;
   }else{
      $localSchema = $this->_localSchema;
   }
  Xsd2GuiStatics::$nameSpacesURI[$localSchema]['array_path'][Xsd2GuiStatics:
                :$actualNameSpace] =
      Xsd2GuiStatics::$actualNameSpace.'#'.Xsd2GuiStatics:$actualSchema.'#'.
      Xsd2GuiStatics::$actualVersion;
 Xsd2GuiStatics::$nameSpacesURI[Xsd2GuiStatics::$actualSchema]['array_path']
                [Xsd2GuiStatics::$actualNameSpace] =
      Xsd2GuiStatics:$actualNameSpace.'#'.Xsd2GuiStatics::$actualSchema.'#'.
      Xsd2GuiStatics::$actualVersion;
   }

   ...
```

Quanto qui di seguito riportato è il risultato finale, ossia `Xsd2GuiStatics::$nameSpacesURI` costruito:

```
Array
(
    [1] => Array
        (
            [ns] => Array
                (
                    [dc] => http://purl.org/dc/elements/1.1/
                    [niso] => http://www.niso.org/pdfs/DataDict.pdf
                    [xlink] => http://www.w3.org/TR/xlink
                )

            [array_path] => Array
                (
                    [default] => default#1#5f6f9400fa7bffcfd33d88...
                    [dc] => dc#2#d9e336d39d7f8280e64fc853a2a7d476
                    [xlink] => xlink#3#ad49878bba62905816cd8ec12a8122a9
                    [niso] => niso#4#93c750eaec89df46a1e737b0d0ad153a
                )

        )

    [2] => Array
        (
            [array_path] => Array
                (
                    [dc] => dc#2#d9e336d39d7f8280e64fc853a2a7d476
                )

        )

    [3] => Array
        (
            [ns] => Array
                (
                    [xlink] => http://www.w3.org/TR/xlink
                )

            [array_path] => Array
                (
                    [xlink] => xlink#3#ad49878bba62905816cd8ec12a8122a9
                )

        )

    [4] => Array
        (
            [array_path] => Array
                (
                    [niso] => niso#4#93c750eaec89df46a1e737b0d0ad153a
                )

        )

)
```

I tag “Import” ed “Include” sono importanti e da gestire con attenzione.
Per quanto riguarda la classe di **Import** essa contiene i metodi utili alla trattazione del nuovo Namespace per il relativo schema che si sta importando. In particolare c'è da notare come viene creato un nuovo oggetto quando sono verificate due importanti condizioni: l'esistenza di un nuovo *schemaLocation* e di un nuovo *Namespace*.

```
public function __construct(DOMElement $node){
   $attribute = $node->attributes->getNamedItem('namespace');
   $doSchema  = FALSE;
   if(!is_null($attribute)){
     $doSchema = $this->_nameSpacesConnect($attribute->value);
   }
   $attribute = $node->attributes->getNamedItem('schemaLocation');
   if(!is_null($attribute) && $doSchema){
     $oldInclude              = Xsd2GuiStatics::$include;
     Xsd2GuiStatics::$include = FALSE;
     $this->_callNewSchema($attribute);
     Xsd2GuiStatics::$include = $oldInclude;
     $this->_defaultNameSpace();
    }
   $this->_digger($node);
}
```

La parte effettivamente più importante da notare è la chiamata al metodo `_nameSpacesConnect`. Esso ricerca il nuovo Namespace su cui lavorare utilizzando il Namespace URI passato (utilizza la proprietà statica `Xsd2GuiStatics::$nameSpacesURI`). Restituisce TRUE se trova il Namespace, FALSE altrimenti.
Successivamente se esiste l'attributo *schemaLocation* e contemporaneamente c'è un nuovo *nameSpace*, questo nodo viene passato a `_callNewSchema` che istanzia il nuovo oggetto schema e a sua volta richiamerà `_newSchema` per fare del nuovo schema un nuovo oggetto DOM. `_defaultNameSpace()` serve ad indicare che adesso ci si trova in nuovo schema e lo fa ripristinando il valore attuale (`Xsd2GuiStatics::$actualNameSpace`) con il valore locale (`_localNameSpace`).

```
protected function _defaultNameSpace(){
     if(!empty($this->_localNameSpace)){
        Xsd2GuiStatics::$actualNameSpace = $this->_localNameSpace;
     }
   }
```

Notare bene che alla fine di tutto è stato invocato il metodo `_digger`.

Per quanto riguarda invece la classe di **Include**, la sua trattazione è meno complessa. Questo perché si tratta di gestire l'inclusione di un nuovo schema che condivide lo stesso Namespace dello schema che lo sta includendo.

```
class Xsd2GuiInclude extends Xsd2GuiImport{
 public function __construct(DOMElement $node){
   $attribute = $node->attributes->getNamedItem('schemaLocation');
   if(!is_null($attribute)){
      $oldInclude              = Xsd2GuiStatics::$include;
      Xsd2GuiStatics::$include = TRUE;
      $this->_callNewSchema($attribute);
      Xsd2GuiStatics::$include = $oldInclude;
   }
   $this->_digger($node);
 }
}
```

Il metodo "_digger"

Uno dei metodi maggiormente utilizzato è sicuramente `_digger`. Il suo uso così largo lascia intuire il suo scopo e la sua importanza: "scavare" all'interno dello schema XSD per poter lavorare con il nodo che viene analizzato.
Effettivamente è proprio questo il lavoro compiuto. Ogni volta che si ha tra le mani un elemento da posizionare all'interno del super-array, `_digger` si occupa primariamente di chiamare il metodo per scriverlo effettivamente all'interno del super-array e successivamente esegue il ciclo di controllo sui figli visto nel capitolo 3.1.6 sull'istanziazione delle classi e qui riportato:

```
foreach($node->childNodes as $child){
  if($child->nodeType == XML_ELEMENT_NODE){
     $className = $this->_nameBuilder($child->tagName);
     if(class_exists($className, false)){
        new $className($child);
        ...
```

In questo modo il sistema è anche ricorsivo: a `_digger` vengono sempre passati elementi DOM, o nodi, e lui controlla se essi hanno figli e se esiste la relativa classe. Quando queste condizioni sono soddisfatte ci saranno una serie di operazioni per il nuovo nodo e quindi nella nuova classe, ed essa stessa una volta compiute queste operazioni dovrà ripassare l'elemento a `_digger` per dargli un posto nella struttura dati. Egli ricontrollerà se il nodo ha figli e il ciclo ricomincia.
É proprio questo il punto di confine tra la lettura e la selezione dei dati presenti nello schema XSD e il posizionamento di essi nella struttura dati. Questo metodo è utilizzato da ogni nodo a cui interessa scavare posizioni nella struttura finale.
Dal frammento di codice ricopiato che segue si può vedere come tra le prime parentesi sia indicato il parametro passato al metodo ossia l'elemento DOM, il nodo, che qui si trova nella variabile `$node`. Notare che la scritta `DOMElement` significa che viene svolto un controllo per verificare che il parametro passato sia effettivamente un elemento DOM.
Ecco il metodo `_digger`:

```
protected function _digger(DOMElement $node){
  $this->_debug($node);
  $this->_libraryBuilder($node, $returnPath, $is_attribute);
  $this->_childAppend($returnPath, $is_attribute);
  if($node->hasChildNodes()){
    $this->_father($node, TRUE);
    if($node->localName == 'element'                &&
       Xsd2GuiStatics::$actualNameSpace == 'default' &&
       ((Xsd2GuiStatics::$rootLevel == 0)             ||
       (Xsd2GuiStatics::$level < Xsd2GuiStatics::$rootLevel))){
          Xsd2GuiStatics::$rootLevel   = Xsd2GuiStatics::$level;
          Xsd2GuiStatics::$rootAddress = Xsd2GuiStatics::$father;
    }
    Xsd2GuiStatics::$level++;
    ...
```

Un altro metodo da qui invocato e molto importante per la definizione dei singoli elementi è **`_father.`** Se si vuole conoscere la posizione di un nodo è ovviamente importante conoscere e gestire chi è il nodo dentro cui si trova annidato, e sapere che egli stesso può essere padre di altri nodi. Il metodo sa controllare e distinguere questo caso tenendo traccia di quando un elemento è sia figlio che padre.
Le informazioni utili a tal fine da reperire e registrare sono il nome del Namespace attuale, lo schema attuale, la versione, il nome del nodo in questione e l'MD5 calcolato sempre per l'xpath.

```
protected function _father($node, $flag){
 $name = $node->getAttribute('name');
 switch($name){
   case   '': $name = $node->getAttribute('ref');
              break;
 }
 if($name != ''){
   if($flag){
     $this->_oldFather       = Xsd2GuiStatics::$father;
     Xsd2GuiStatics::$father = Xsd2GuiStatics::$actualNameSpace.
                               '#'.Xsd2GuiStatics::$actualSchema.
                               '#'.Xsd2GuiStatics::$actualVersion.
                               '#'.$name.
                               '#'.md5(Xsd2GuiStatics::$xpath);
   }else{
   Xsd2GuiStatics::$father = $this->_oldFather;
   }
 }
}
```

class Xsd2GuiElement

Osservare la classe per il tag “element” è un esempio ed è il modo migliore di capire il livello più basso del super-array, ovvero dove effettivamente risiedono gli elementi e gli attributi che costituiranno il documento XML finale.
La prima importante osservazione da fare è notare la posizione che queste classi hanno nel codice. Abbiamo visto come la classe `Xsd2GuiTag` rimane quella principale dalla quale molte altre si estendono. Per le classi di cui si parla ora invece, esiste un ulteriore suddivisione.

Le classi per i tag `element`, `complexType`, `simpleType` sono in realtà estensioni della classe `Xsd2GuiPseudoElement` che a sua volta è un estensione della stessa `Xsd2GuiTag`. Questo perché quei tre tag richiedono un trattamento leggermente diverso.

Ecco la classe `Xsd2GuiPseudoElement`:

```
class Xsd2GuiPseudoElement extends Xsd2GuiTag{
  protected function _checkElement(DOMElement $node){
    $name  = $node->getAttribute('name');
    switch($name){
      case   '': $name = $node->getAttribute('ref');
                              break;
    }
     if(Xsd2GuiStatics::$choiceId === TRUE &&
        Xsd2GuiStatics::$choiceSuppressor === FALSE){
           Xsd2GuiStatics::$nodeData[$name]['choice_group'] =
             Xsd2GuiStatics::$choiceCounter;
    }
    if(empty($name)){
       $this->_digger($node);
    }else{
    //##--> Add Default Data
    Xsd2GuiStatics::$nodeData[$name]['is_attribute'] = FALSE;
    Xsd2GuiStatics::$nodeData[$name] +=
                                  $this->_nodeSimpleData($node);
    //##--> Add XPath
    Xsd2GuiStatics::$xpath .= '/'.$name;
    Xsd2GuiStatics::$nodeData[$name]['xpath'] = Xsd2GuiStatics::$xpath;
    //##--> Add Node Type
   Xsd2GuiStatics::$nodeData[$name] += $this->_type($node, $name);
    //##--> Add to Library array
    $this->_digger($node);
    //##--> Xpath recover
    Xsd2GuiStatics::$xpath = dirname(Xsd2GuiStatics::$xpath);
    }
  }
}
```

Quando la classe “element” viene istanziata, come fondamentalmente per “complexType” e “simpleType”, quello che succede è chiamare il metodo `_checkElement` presente nella classe qui sopra passandogli il nodo. Questo perché tale metodo decide praticamente la sorte di chi lo invoca.

```
class Xsd2GuiElement extends Xsd2GuiPseudoElement{
   public function __construct(DOMElement $node){
      $this->_checkElement($node);
  }
}
```

Come già anticipato precedentemente anche questi elementi nella struttura dati sono rappresentati tramite array, i quali devono portare con sé tutte le informazioni utili a descrivere il nodo in questione. E questo deve avvenire in maniere diversificata e secondo che si stia trattando un elemento, un attributo, un attributeGroup o altro. Questa fase è cruciale per il funzionamento dell'intero sistema. Le informazioni collegate al singolo nodo saranno quelle che determinano la sua reperibilità.
Tornando al metodo `_checkElement` sarà più chiaro. Dopo aver estratto il nome dell'elemento e aver fatto un controllo sullo stato della restrizione *choice*, il metodo verifica prima di tutto se il nome dell'elemento esiste effettivamente. Se non dovesse esserci invoca direttamente `_digger`, in modo comunque da scavare nella struttura per segnare che un altro nodo è comunque presente, se verificato invece procede con la costruzione dell'array del nodo.
Le informazioni associate ai nodi da estrapolare e inserire nell'array dell'elemento sono: le occorrenze, sapere se è un attributo o meno, l'xpath, il tipo e il nodo padre. `Xsd2GuiStatics::$nodeData` è la proprietà array usata per queste costruzioni.
Ecco come si presenta un array che descrive un elemento:

```
[video_metrics] => Array
  (
     [e4c63e11847e23c505782551f9c9b54c] => Array
         (
            [is_attribute] =>
            [min]   => 1
            [max]   => 1
            [xpath] => >/video_group/video_metrics
            [type]  => default#1#5f6f9400fa7bffcfd33d8834f3eb59ef#
                       video_spatialmetrics#eff7e3039ee6ce3889...
            [father]=> default#1#5f6f9400fa7bffcfd33d8834f3eb59ef#
                       video_group#9a5b6563fd1f7207ff3dc07fb5d...
```

Per mettere in piedi queste informazioni vengono comunque utilizzati altri metodi: `_nodeSimpleData` per aggiungere le occorrenze, ovvero per sapere se un elemento è da usare obbligatoriamente o meno e quante volte lo si può ripetere.

```
protected function _nodeSimpleData($node){
   $valueMin = $node->getAttribute('minOccurs');
   switch($valueMin){
      case   ''        : $min = 1;
                         break;
      case   $valueMin : $min = $valueMin;
                         break;
   }
    $valueMax = $node->getAttribute('maxOccurs');
    switch($valueMax){
      case   ''          : $max =  1;
                           break;
      case   'unbounded' : $max = -1;
                           break;
      case   $valueMax   : $max = $valueMax;
                           break;
   }
   return $elementsArrayNode = Array('min'    => $min,
                                     'max'    => $max);
}
```

Qui vengono analizzati i valori degli attributi XML *minOccurs* e *maxOccurs*. A seconda del caso cambia il valore riportato nel risultato finale, ossia l'array che viene ritornato e attaccato quindi a `Xsd2GuiStatics::$nodeData`. Sarà quest'ultimo, come si può vedere tra poco, che descrivendo il nodo in questione verrà attaccatto a sua volta nella struttura dati.

Per il *type* dell'elemento viene invocato il metodo `_type`. Per i motivi spiegati nel capitolo 3.1.3 le informazioni da collocare in questo campo richiedono un lavoro più oneroso, perché bisogna raccogliere più informazioni nel caso in cui il tipo di dato non appartenga alla categoria dei tipi semplici o nativi. Se ad esempio un elemento fosse di tipo stringa o di tipo intero basterebbe indicarlo come tale. A tale scopo infatti si vedrà come il tipo dell'elemento viene confrontato con gli elementi del seguente array che raccoglie tutti i tipi semplici:

```
static public $simpleTypes = Array('ID', 'string', 'decimal', 'integer',
                                   'positiveInteger', 'boolean', 'date',
                                   'time', 'dateTime', 'anyURI',
                                   'unsignedLong');
```

Se c'è corrispondenza con uno di questi allora viene indicato solamente qual'è, se però il tipo fa riferimento ad un altro elemento dichiarato in un altro punto del documento allora diventa necessario avere la traccia per arrivare a quest'ultimo.
In questo caso le informazioni per indicare il tipo dell'elemento sono: il Namespace URI, lo schema attuale, il nome dell'elemento a cui fa riferimento e l'MD5 calcolato per l'xpath sempre di quest'ultimo.

```
protected function _type($node, $name){
...
 if(!empty($attributeType)){
  if(strpos($attributeType, ':') !== FALSE){
   $nsType   = explode(':', $attributeType);
   $dfSchema = Xsd2GuiStatics::$actualSchema;
  }else{
   $nsType = Array(Xsd2GuiStatics::$actualNameSpace,
                   $attributeType);
  }
  if(in_array($nsType[1], Xsd2GuiStatics::$simpleTypes)){
   $output = $nsType[1];
   }elseif(isset(Xsd2GuiStatics::$nameSpacesURI
                [Xsd2GuiStatics::$actualSchema]
                ['array_path'][$nsType[0]])){
    $typeXpath = $this->_getTypeXpath($mode, $name,
                         $node->localName, $nsType[0], $nsType[1]);
   if(empty($typeXpath)){
    $typeXpath = '';
   }else{
    $typeXpath = md5($typeXpath);
   }
   $output =  Xsd2GuiStatics::$nameSpacesURI
              [Xsd2GuiStatics::$actualSchema]['array_path']
              [$nsType[0]].'#'.$nsType[1].'#'.$typeXpath;
   }else{
```

```
    $output = null;
   }
 }else{
  $output = $attributeType;
 }
 return Array('type' => $output);
}
```

Nella prima parte che è stata omessa viene effettuato un controllo e una selezione sul nome dell'attributo *type*, che viene salvato nella variabile `$attributeType.`

Nel momento in cui questa variabile non è vuota viene analizzato il suo contenuto. Se questo a sua volta è costituito da un valore avente prefisso, ad esempio **`niso:`**`format`, attraverso la funzione *explode* viene separato e messo nell'array `$nsType` ciò che c'è prima e dopo il carattere ":" in due valori separati.

In un array se non sono specificate dal programmatore le chiavi per indicare i valori, esse sono automaticamente create da PHP quando si cerca di creare un nuovo elemento, e questo avviene tramite un valore numerico incrementale. Quindi se nell'array c'è come primo elemento *niso* e come secondo *format*, se si vuole indicare *niso* si scrive "$nsType[0]" e "$nsType[1]" per *format*.

Comunque se in `$attributeType` non c'è un valore di questo tipo, l'array `$nsType` viene formato con altri due valori: quello dell'attuale Namespace e quello dell'attributo stesso presente sempre in `$attributeType.` A questo punto avviene il confronto di `$attributeType` con gli elementi presenti in `$simpleTypes,` cioè l'elenco degli elementi semplici. Come già detto se si trova una corrispondenza si raggiunge il risultato finale ponendo `$output`, che è la variabile dove fissiamo il risultato finale da ritornare, uguale alla corrispondenza stessa. Viceversa viene controllato se è settato e quindi verificato il Namespace URI con anche il Namespace attuale. Viene chiamato il metodo `_getTypeXpath` che ritornerà come risultato ciò che possiamo trovare nella variabile `$typeXpath`, ovvero l'xpath dell'elemento a cui si sta facendo riferimento.

Se questa porta con sé un valore viene applicato su esso l'algoritmo MD5 e quindi ci sono tutte le informazioni necessarie per formare `$output`, e si può restituire il risultato finale da attaccare ancora a `Xsd2GuiStatics::$nodeData.`

Alla fine di queste operazioni viene passato il nodo, così costituito, al metodo `_digger`. Adesso egli scaverà nella struttura posizionando l'array `Xsd2GuiStatics::$nodeData` che rappresenta l'elemento così formato.

Adesso è stato visto il caso per il tag “element”. La stessa logica ma con operazioni ovviamente diverse è stata seguita per gli altri tag. In questo modo le classi istanziate per i tag utili possono lavorare in modo indipendente, pur sfruttando le chiamate a metodi necessari a tutti.
Se si guarda ad esempio la gestione del tag “attribute” si può notare come anch'esso invoca il metodo per formare il type, ma allo stesso tempo chiama anche altri metodi per formare quelle informazioni che servono a descrivere solo gli attributi.
Questa è un'altra occasione per ribadire il concetto di programmare con le classi così organizzate. Si possono utilizzare metodi per formare informazioni comuni e metodi indipendenti per formare informazioni utili al singolo tipo di nodo.

```
class Xsd2GuiAttribute extends Xsd2GuiTag{
 public function __construct(DOMElement $node){
 //##--> Add XPath
 Xsd2GuiStatics::$xpath .= '/[@'.$name.']';
 Xsd2GuiStatics::$nodeData[$name]['xpath']=Xsd2GuiStatics::$xpath;

 if($node->getAttribute('use') != 'prohibited'){
   //##--> Add Default Data
   Xsd2GuiStatics::$nodeData[$name]['is_attribute'] = TRUE;
   Xsd2GuiStatics::$nodeData[$name] += $this->_attributeSimpleData($node);
   Xsd2GuiStatics::$nodeData[$name] += $this->_type($node, $name);
  }
  //##--> Add to Library array
  $this->_digger($node);
  //##--> Xpath recover
  Xsd2GuiStatics::$xpath = dirname(Xsd2GuiStatics::$xpath);
 }
}
```

Questo è un esempio di array che rappresenta un attributo:

```
[Type] => Array
   (
      [a56397f52b94e519cd3153a176348b59] => Array
         (
           [xpath] => >/audio_creation/transcriptionchain/
                       device_description/[@Type]
           [is_attribute] => 1
           [optional] => required
           [default] =>
           [fixed] =>
           [type] => string
           [father] => default#1#5f6f9400fa7bffcfd33d8834f3eb59ef#
                       device_description#c60c547fe7a33ad6f679...
         )
```

Ognuna di queste informazioni che viene riportata nell'array che descrive un singolo elemento, che sia un attribuo, un elemento o altro, diventano fondamentali per la descrizione del super-array multidimensionale. Grazie a questo sistema automatizzato si è in grado di raccogliere e ordinare tutto ciò che serve, risolvendo anche le problematiche legate alle strutture referenziate e ai tipi di dato non semplici. A questo punto rimane solo da vedere l'effettiva costruzione della struttura dati, o meglio, vedere il metodo che la realizza.

Adesso che tutte le informazioni utili sono state raccolte, ogni array di ogni nodo porta con sé tutto quello di cui necessita per la rappresentazione corretta dello schema XSD e quindi per la relativa costruzione del documento XML.

3.2.3 - libraryBuilder

```
protected function _libraryBuilder($node, &$returnPath,&$is_attribute){
  if(!empty(Xsd2GuiStatics::$nodeData)){
    $keys = array_keys(Xsd2GuiStatics::$nodeData);
    Xsd2GuiStatics::$nodeData[$keys[0]]['father'] = Xsd2GuiStatics::$father;

    Xsd2GuiStatics::$nodeLibrary[Xsd2GuiStatics::$actualNameSpace][Xsd2GuiSt
atics::$actualSchema][Xsd2GuiStatics::$actualVersion][$keys[0]][md5(Xsd2GuiS
tatics:$nodeData[$keys[0]]['xpath'])] = Xsd2GuiStatics::$nodeData[$keys[0]];

     $returnPath   = Xsd2GuiStatics::$actualNameSpace.'#'.
Xsd2GuiStatics::$actualSchema.'#'.Xsd2GuiStatics::$actualVersion.'#'.$keys[0
].'#'.md5(Xsd2GuiStatics:$nodeData[$keys[0]]['xpath']);

    $is_attribute = (isset(Xsd2GuiStatics::$nodeData
[$keys[0]]['is_attribute']))?Xsd2GuiStatics::$nodeData[$keys[0]]['is_attribu
te']:null;

     $this->_typesLibraryBuilder($node->localName, $keys[0]);
      Xsd2GuiStatics::$nodeData = Array();
  }
}
```

Questo è il metodo che costruisce la struttura dati e non a caso è quindi chiamato da `_digger`. `_libraryBuilder()` si occupa a questo punto di attaccare al super-array il nodo analizzato. Il primo passo è verificare che *Xsd2GuiStatics::$nodeData* non sia nullo, quindi assicurarsi che l'array che descrive il singolo nodo in questione giunga con tutte le informazioni utili. Successivamente bisogna ovviamente reperire il nodo padre per sapere con precisione dove collocare il nuovo nodo. Infine si passa alla concatenazione delle informazioni che formano la struttura dati.

Ecco come viene montato: qui di seguito è riportato la stampa dell'inizio del super-array:

```
Array     // Questo è l'inizio della struttura dati, il super-array
 (
  [default] => Array  //  $actualNameSpace
    (
     [1] => Array     //  $actualSchema
      (
       [5f6f9400fa7bffcfd33d8834f3eb59ef] => Array //$actualVersion
        (
         [audio_mimetype] => Array // $keys[0], nome del nodo
           (
            [0c021911bb31d7dc5a28992ed81c578f] => Array

              // md5(Xsd2GuiStatics:$nodeData[$keys[0]]['xpath'])
                (
                    [is_attribute] =>
                    [min] => 1
                    [max] => 1
                    [xpath] => >/audio_mimetype
                    [type] =>
                    [father] =>
                 )


           )
    ...
```

Xsd2GuiStatics::$nodeData[$keys[0]] Questo è l'array che descrive il singolo nodo, in questo caso è un elemento

Tutte le informazioni contenute nell'array che rappresenta il singolo elemento forniscono tutto quello che c'è da sapere sul suo ruolo nella struttura.

4 - Conclusioni

Grazie alle tecnologie utilizzate è stato possibile unire e combinare una serie di strumenti capaci di elaborare i documenti XML.
In questo modo si è giunti alla costruzione di una serie di metodi capaci di ordinare in modo preciso le informazioni necessarie, arrivando così alla costruzione del core indipendentemente dal tipo di schema XSD che si presenta in ingresso.
In questo progetto è stato cruciale capire come gestire le informazioni di input e la successiva creazione di una struttura dati precisa e ordinata. Per riuscire a raggiungere tale obiettivo si è dovuto necessariamente, più di una volta, ricominciare da capo tutto il lavoro piuttosto che cambiare improvvisamente il percorso previsto. Questo tipo di scenario si verificava quando, man mano che si disegnavano i metodi per costruire questa struttura, anche solo una di tutte le informazioni utili diveniva irraggiungibile. La soluzione di questi casi è stata quella di fare sempre un passo indietro affinché si potesse ridisegnare il percorso e comprendere tutto il necessario.
Capire come gestire le informazioni di input e la successiva creazione di una struttura dati precisa e ordinata era effettivamente l'obiettivo, e questo è stato raggiunto nonostante le difficoltà che si sono presentate su diversi fronti. Un aspetto altrettanto importante, nonché da considerarsi anch'esso come un obiettivo raggiunto, è stato il fatto di essere riusciti a costruire questo sistema complesso in poche righe di codice: il giusto posizionamento ed uso dei metodi e delle proprietà, combinato con la rispettiva organizzazione delle classi, hanno permesso di condensare e quindi di programmare un codice tanto economico quanto efficace.
Grazie a questo super-array, tutti i dati che compongono lo schema XSD e quindi che rappresentano i dati del documento XML da creare, sono descritti da tutte le informazioni necessarie per sapere come un singolo elemento si costituisce e come fa ad organizzarsi nel caso di una struttura referenziata. La struttura diventa quindi navigabile da qualunque punto: è sempre possibile sapere come un singolo nodo è organizzato nell'intera struttura.
É da notare come questa dinamicità è l'aspetto che riguarda maggiormente le prospettive future di questo progetto: permettere ad utenti non specializzati di creare documenti XML. Quando verrà creato il sistema automatizzato che costruisce l'interfaccia grafica a partire dalla costruzione della struttura dati, il prodotto sarà effettivamente utilizzabile a tal fine. Questo programma si collocherà nell'applicativo di *creatore* sulla piattaforma Codex[ml], il modulo di data entry, aggiornamento e gestione degli standard XML.
Per quanto riguarda le competenze acquisite, queste si riflettono su un duplice aspetto: da una parte

l’entrare in contatto con un ambiente professionale e dall’atro il saper applicare le conoscenze puramente tecniche, cioè quelle inerenti alle tecnologie coinvolte.

Ringrazio il prof. Luca Andrea Ludovico e le persone che mi hanno sostenuto, formato ed aiutato nello sviluppo dell'intero progetto presso il CILEA: Matteo Perelli, il dr. Michele Meloni, dr. Claudio Cortese, dr. Giuseppe Digilio e la dr.ssa Emilia Groppo.

Bibliografia

[B1] - PHP architect's ZEND PHP 5 Certification / Study Guide - Toronto : Marco Tabini & Associates, Inc., 2006

[B2] - Programming PHP - O'Reilly, Autori: Rasmus Lerdorf, Kevin Tatroe, March 2002

[B3] - Usare XML : Special Edition - L.A. Phillips. - 1. - Milano : Mondadori Informatica, 2000

[B4] - XML Pocket Reference. - O' Reilly, Autori: R. Eckstein, Released:August 2005

[B5] - XML and PHP by Vikram Vaswani, Paperback: June 16, 2002, New Riders

[B6] - Document Object Model: Processing Structured Documents - McGraw-Hill/OsborneMedia; 1st edition: July 24, 2002

[B7] - Beginning XML with DOM and Ajax: From Novice to Professional (Beginning: From Novice to Professional) - Sas Jacobs – Paperback: June 5, 2006

[B8] - XML Schema - Eric van der Vlist and Eric van der Vlist – Paperback: June 15, 2002

[B9] - Definitive XML Schema - Priscilla Walmsley – Paperback: Dec. 17, 2001

[B10] - PHP Object-Oriented Solutions - David Powers – Paperback: Mar. 9, 2010

Sitografia

[S1] - http://www.php.net/

[S2] - http://www.w3schools.com

[S3] - http://www.html.it/

[S4] - http://it.wikipedia.org