UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA DELL'INFORMAZIONE

eLaw: gestione automatica modulistica notarile

Alessandro Secco

Relatore:

Professor Enoch Peserico Stecchini Negri De Salvi

Correlatore:

Ing. Federica Bogo

Anno Accademico 2011/2012

21 Febbraio 2012

Alessandro Secco: *eLaw: gestione automatica modulistica notarile,* Laurea in Ingegneria dell'informazione, Febbraio 2012

A Nicola Sammarco
padre nel karate... maestro di vita...

RINGRAZIAMENTI:

Desidero ringraziare tutti coloro che mi hanno sostenuto ed appoggiato nella realizzazione di questa tesi.

Il Prof. Peserico per aver ideato il progetto ed avermi dato la possibilità di parteciparvi.

L' ing. Federica Bogo per la costante presenza e gentilezza e per tutto l'aiuto che mi ha dato nello sviluppare il progetto.

Il mio collega e amico Matteo Ceccarello, sulle cui conoscenze, abilità ed impegno ho sempre potuto contare senza mai riceverne delusioni.

Tutte le persone che hanno creduto in me: i miei genitori ed i miei amici.

SOMMARIO

Questa tesi vuole descrivere il processo produttivo ed il funzionamento di un software di gestione automatica di documenti notarili. Con questo documento si fa riferimento in particolar modo all'utilizzo di informazioni rinvenute in uno specifico atto ed alla rielaborazione di queste informazioni secondo degli standard ministeriali. Il software ivi descritto si presenta come un plugin per OpenOffice e si vuole proporre come un'utile alternativa agli attuali software notarili, semplificando e, ove possibile, automatizzando il lavoro che un notaio dovrebbe altrimenti svolgere. Punti di forza e prerogative di questo lavoro sono infatti la semplicità di utilizzo e l'automatizzazione. Si capirà inoltre con la lettura dell'elaborato come sarà possibile utilizzare questa estensione in ambiti ben più generici, che trascendono quello esclusivamente notarile.

INDICE

ELENCO DELLE FIGURE

ELENCO DELLE TABELLE

1 INTRODUZIONE

Al giorno d'oggi un notaio, negli studi attuali, per redigere un atto, deve seguire una serie di passaggi fondamentali, ma molto spesso ripetitivi. La scrittura dell'atto e la compilazione di diversi moduli tramite molteplici software distinti (che non comunicano sufficientemente tra loro), obbligano l'utente a ripetere più e più volte le stesse digitazioni, rendono il procedimento pedante, noioso e lungo. Il presente lavoro nasce dall'interrogazione sulla possibilità di agevolare queste fasi, garantendo all'utente finale di poter automatizzare gran parte di questo lavoro minimizzandone i tempi ed incrementando la produttività del notaio. A tale proposito ci si è chiesti se fosse stato possibile limitare le digitazioni delle più ripetitive informazioni creando un software che gestisse tutto in maniera semi automatizzata.

1.1 IL SOFTWARE ELAW

eLaw è un progetto il cui scopo primario consiste nella creazione di un ufficio completamente informatizzato per studi notarili. eLaw si propone di essere un pacchetto che comprende uno o più computer completi dal punto di vista hardware e software (Hardware e Sistema operativo selezionati), una serie di periferiche immancabili in un ufficio notarile (quali un lettore di firme digitali, una stampante multifunzione, fax e apparato telefonico/voip) ed un'ulteriore serie di utili features, quali un database condiviso ed un'efficiente gestione della rete tra i diversi uffici.

Al fine di ottenere la struttura dell'ufficio modulare e scalabile, si è scelto di fornire attrezzature di un unico modello così che, in caso si necessiti una sostituzione di un qualunque componente dell'ufficio, questa possa avvenire in tempi rapidi senza che si debbano impiegare tempi eccessivi per la riconfigurazione. Lo stesso ragionamento vale anche nel caso in cui si voglia aggiungere una macchina al sistema. Questa sarà semplicemente integrata senza perdita di tempi per la scelta e la riconfigurazione.

In merito al sistema operativo, si è scelto un ambiente Linux: Debian (noto per la sua affidabilità, stabilità e robustezza) modificato appositamente per l'impiego. Il software preinstallato consisterà, essenzialmente, di:

- Un editor di testo (processore di testo) atto a svolgere agilmente i principali compiti richiesti in uno studio notarile;
- Un software di telefonia analogica/voip con possibilità di fungere da segreteria;
- Un browser specifico in grado di interagire con l'editor di testo al fine di garantire semplicità nello svolgimento delle più frequenti operazioni richieste in uno studio notarile;
- Un programma per l'utilizzo della strumentazione per le firme digitali;

- Una semplice integrazione con le periferiche di stampa, in grado di stampare con inchiostro indelebile (necessario a norma di legge per la redazione di atti notarili).

1.2 IL PLUGIN PER OPENOFFICE

Nell'ambito dell'apparato software, in particolare, il progetto prevede di creare un estensione di un processore di testi che automatizzi (o comunque semplifichi) la creazione di documenti da parte dell'utente. Questi, in genere, deve scrivere un atto notarile (in forma di documento di testo con formattazione rigida) e compilare diversi forms a seconda dell'operazione che deve essere svolta; spesso, però, i dati richiesti dai forms sono presenti nell'atto redatto ed ivi individuabili. Il progetto eLaw si propone di automatizzare questo processo, il che consisterebbe quindi nell'identificare questi dati all'interno del documento di testo (durante la stesura dello stesso) per poi inserirli automaticamente all'interno dei forms adeguati e creare infine gli output necessari che verranno successivamente inviati agli uffici competenti.

Come programma di base si è dovuto scegliere un processore di testi open source dotato di una semplice interfaccia utente. I principali candidati erano OpenOffice e Lyx; è stato scelto il primo perchè si riteneva potesse avere un'interfaccia utente grafica più familiare.

Ma a cosa serve questa estensione ad OpenOffice? Qual è il suo scopo?

Ora un notaio, per una qualsiasi operazione, deve scrivere un atto tramite un processore di testi e successivamente, a seconda della tipologia di atto redatto, deve compilare alcuni forms e generare determinati output a struttura fissa che devono poi essere inviati agli uffici competenti. Per svolgere questa operazione si rende necessario l'utilizzo di tre ulteriori programmi (a seconda dell'operazione notarile da eseguire):

1. *UniMod Client*: produce file *XML* da inoltrare all'Agenzia del Territorio;
2. *Fedra Plus*: produce file *XML* e binari da inoltrare al Registro Delle Imprese;
3. *ComUnica*: unisce l'output dei due programmi precedenti in un pacchetto da inviare a Camere di Commercio, INPS, INAIL e Agenzia delle Entrate.

Ciò che propone il pacchetto eLaw è di eliminare l'interazione con questi programmi e, inoltre, di minimizzare la fase del lavoro relativa alla compilazione. Il plugin da creare dovrà quindi soddisfare i seguenti requisiti (coincidenti con le fasi principali dell'utilizzo del software):

1. La maggior parte dei campi da riempire nei tre programmi sono presenti nell'atto ed ivi semplicemente individuabili. Si vorrebbe quindi garantire all'utente di dover compilare manualmente solo una piccola parte dei campi altrimenti richiesti dai programmi sopra citati. Ciò che si propone di fare l'estensione, una volta ritrovate le parole chiave, è quindi di:
   a) individuarle automaticamente (o comunque individuarne la maggior parte);
   b) renderle visibili semplicemente all'utente;

c) permettere a questi di modificarli in caso di errato riconoscimento o di riconoscimento erroneo e aggiungerne alcune che non siano state riconosciute automaticamente[1].

2. Si è resa necessaria la creazione di un archivio contenente sia le parole chiave individuate automaticamente, sia quelle modificate o aggiunte dall'utente. Questo archivio doveva essere considerato come un'istanza del processore di testi e non esterno, così da poter memorizzare correttamente tutti gli elementi alla chiusura del programma.

3. A questo punto, una volta riconosciute le parole o le frasi necessarie, il plugin si propone di generare i files di output (a seconda del software notarile che si deve emulare). Questi files hanno una struttura che deve essere rigidamente rispettata per l'approvazione dell'atto da parte degli uffici cui questi files sono indirizzati. La struttura di un file dipende dal tipo di atto da redigere e dalla destinazione (ognuno dei programmi sopra citati crea un suo output con una sua struttura).

In questo documento si vuole attenersi agli ultimi due punti, corrispondenti con le ultime fasi del lavoro, ovvero la memorizzazione dei campi a runtime e la gestione/creazione dei file di output.

1.3 *unimod client*

Nell'ambito del progetto sopra descritto si farà riferimento esclusivamente al programma *Unimod Client* e quindi si spiegherà come si è riusciti ad emulare il funzionamento di questo software sincronizzandolo con l'atto redatto dall'utente (e quindi integrando il processore di testi delle funzionalità tipiche del software in questione).

UniMod Client è un pacchetto software che consente di compilare il Modello Unico Informatico e produrre come output il relativo plico contenente il file *XML*. Questo software crea un' interfaccia che garantisce all'utente di compilare determinati campi (obbligatori o facoltativi a seconda della tipologia e della modalità in cui si dovrebbe svolgere l'operazione notarile). Successivamente crea degli output: file generati in formato *XML* che contengono i dati fondamentali a descrivere l'atto e l'operazione eseguita.

UniMod Client non gestisce una base dati, quindi è necessario inserire ogni volta tutti i dati del soggetto o dell'immobile, anche se tale soggetto o immobile, compare in più adempimenti. Il pacchetto *UniMod Client* permette l'aggiornamento di alcune tabelle (liste) utilizzate dal programma che potrebbero dover essere modificate a seguito di variazioni normative. E' possibile infatti scaricare attraverso Internet le nuove tabelle in una cartella a scelta e richiamare da programma la cartella dove sono stati scaricati i file di aggiornamento. Automaticamente il programma sostituirà le vecchie tabelle con le nuove e i nuovi dati saranno automaticamente messi a disposizione per una corretta lavorazione. Il pacchetto recepisce le specifiche tecniche Unico16042008.dtd che descrivono il Modello Unico.

Il Modello Unico riunisce tutte le informazioni necessarie alla registrazione, alla trascrizione nonché alle volturazione degli atti immobiliari.

1 Di tutta questa fase del lavoro si è occupato il collega Matteo Ceccarello.

Il professionista esterno compila il Modello Unico utilizzando le funzionalità del pacchetto, controlla che esso sia formalmente corretto e lo invia al Centro attraverso il canale telematico dell'Ufficio del Territorio (Sister). Il file inviato è controllato e memorizzato sull'application server del Centro. In caso di controlli positivi, si effettua la registrazione dell'atto, si attiva il sistema di riscossione dell'autoliquidazione, mentre i dati dell'Ufficio del Territorio (nota/e ricostruita/e con richiesta di voltura automatica catastale) sono inviati alle Conservatorie dei RR.II. di competenza.

Come si è potuto evincere dalla descrizione del funzionamento del software, il lavoro di compilazione risulta essere noioso e ripetitivo, specialmente nel caso in cui si dovessero reinserire dati che già erano stati inseriti precedentemente.

Il pacchetto eLaw si propone di minimizzare questa parte del lavoro con una procedura semiautomatizzata che garantisca l'inserimento e la compilazione dei campi già presenti nell'atto (che comunque dovrebbe essere redatto ed allegato ai documenti *XML* in output ad *UniMod Client*) mettendo inoltre a disposizione un database che memorizzi i dati relativi ad atti già redatti in precedenza o a soggetti/immobili già registrati, minimizzando ulteriormente il lavoro di compilazione.

Il software *UniMod Client* utilizza un file direttivo denominato Unico16042008.dtd che descrive la struttura dei file generati in uscita dal programma. I file *XML* prodotti in output devono essere quindi validati da questo *DTD*; in caso di erronea validazione, l'atto non potrà essere registrato e verrà automaticamente scartato dall'agenzia del territorio, rendendo necessario l'intero processo di stesura dell'atto e della compilazione. Parte fondamentale del progetto eLaw consiste dunque in una corretta ed efficiente generazione dei file di output seguendo le direttive descritte dal file *DTD* direttamente e liberamente scaricabile dal sito dell'agenzia del territorio.

Prima di procedere con l'illustrazione del progetto si ritiene importante dare una descrizione di cosa siano file *XML* e *DTD*.

2 *XML* E *DTD*: LO STATO DELL'ARTE

Come precedente illustrato, parte fondamentale nello sviluppo del progetto risiede nelle specifiche dettate da un documento *DTD* e la sua tramutazione in *XML* da esso validato. Prima di illustrare le metodologie utilizzate per sviluppare il programma, si ritiene di fondamentale importanza illustrare brevemente il funzionamento di *XML* e *DTD* soffermandoci brevemente anche sulle rispettive sintassi e sullo stato dell'arte.

2.1 IL LINGUAGGIO XML

XML (*eXtensible Markup Language*) è un linguaggio di markup che descrive una struttura di dati. La sintassi di questo linguaggio risulta particolarmente simile alla sintassi HTML, tuttavia garantisce in più:

- La possibilità di definire nuovi tag;
- La possibilità di una nidificazione dei tag garantendo una visualizzazione gerarchica del documento;
- La possibilità di contenere, al suo interno, una descrizione e spiegazione della sintassi utilizzata, in modo che possa essere utilizzata da applicazione che richiedono una validificazione della struttura del documento.

In particolare la proprietà della nidificazione conferisce al documento proprietà simili a quelle di una struttura di dati ad albero. Per questo motivo, quando si parla di *XML* si fa spesso implicitamente riferimento ad un albero descritto dal documento. La descrizione gerarchica dei tag definiti nel documento, descrivono la struttura logica dello stesso; i dati effettivi contenuti nel documento descrivono invece la struttura fisica del documento. Un esempio di struttura di un file *XML* è il seguente:

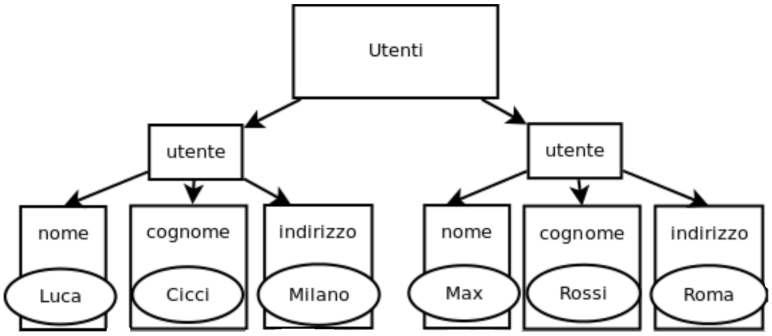
Algorithm 2.1 Esempio di *XML*

```
<?xml version="1.0" encoding="UTF-8"?>
<utenti>
    <utente>
        <nome>Luca</nome>
        <cognome>Cicci</cognome>
        <indirizzo>Milano</indirizzo>
    </utente>
    <utente>
        <nome>Max</nome>
        <cognome>Rossi</cognome>
        <indirizzo>Roma</indirizzo>
    </utente>
</utenti>
```

La visualizzazione ad albero del documento posto come esempio è la seguente, dove i quadrati definiscono la struttura logica dell'*XML*, mentre i tondi portano la descrizione della struttura fisica:

Queste proprietà sopra elencate garantiscono una massima flessibilità nella stesura di un *XML*. È tuttavia possibile descrivere un modello che indica la struttura che può assumere un *XML*. Questo modello

Figura 1: Rappresentazione ad albero dell'*XML* dell'algoritmo 2.1



è rappresentato da un *DTD* (*Document Type Definition*). Un file *DTD* definisce quindi una grammatica a cui deve obbedire un certo file *XML* per poter essere validato e può essere descritto internamente o esternamente all'*XML* stesso.

2.2 VALIDAZIONE DI *xml* TRAMITE *dtd*

2.2.1 *Cosa è un DTD?*

Come precedentemente accennato, un *DTD* (*Document Type Definition*) è un linguaggio atto a validare un documento *XML*. Questo viene utilizzato per definire gli elementi che possono essere inseriti all'interno di un documento *XML*. Sostanzialmente un documento *DTD* definisce la grammatica; un documento *XML* definisce una parola. Quando si parla di "Validazione di un documento *XML* tramite *DTD*" si intende la verifica che il documento *XML* in questione rispetti le regole descritte e definite nel *DTD*.

Un documento *DTD* si occupa di:

- definire gli elementi che possono (o devono) presentarsi in un file *XML* da esso validato;
- definisce la struttura di ogni elemento: ovvero ne definisce la cardinalità, cosa può questo contenere, l'ordine, la quantità degli elementi che possono in esso comparire e se questi sono opzionali, obbligatori;
- per ogni elemento definisce gli attributi dello stesso ed un dominio per i valori di questi attributi;

2.2.2 *La sintassi DTD*

La sintassi di un *DTD* prevede la suddivisione in elementi, ciascuno dei quali è descritto da:

- una lista di elementi figli (eventualmente vuoto): modello di contenuto;
- una lista di attributi dell'elemento in questione (eventualmente vuoto);

la formulazione di una lista di elementi o di attributi e diretta da alcune semplici regole che indicano il numero di inserimenti possibili/richiesti per ciascuno dei valori indicati. Per semplicità di trattazione, si farà riferimento ad alcuni semplici esempi che descrivono come deve essere interpretata la sintassi del documento.

Rifacendosi all'esempio precedentemente utilizzato per la descrizione di un file *XML*, l'elemento "utente" possiede tre elementi figli: gli elementi "nome", "cognome" e "indirizzo", ciascuno dei quali è presente in quantità unica. La dichiarazione di tale elemento è descritta come segue:

<!ELEMENT utente (nome, cognome, indirizzo)>

Tuttavia è possibile definire altre regole che identificano la molteplicità di ogni singolo elemento figlio. Questa sintassi prevede:

- EMPTY viene utilizzato per definire un elemento privo di figli: nell'esempio di cui sopra, la dichiarazione del nodo "nome" sarà: <!ELEMENT nome EMPTY>;
- ANY indica l'esatto opposto: l'elemento potrà possedere qualunque tipo di elemento come figlio. Il codice risulta: <!ELEMENT Object ANY>. In questo caso l'elemento Object può avere come figlio qualunque elemento (che comunque dovrà essere descritto all'interno dello stesso documento *DTD*).
- * indica che l'elemento precedente può essere ripetuto un numero arbitrario di volte. Nell'esempio precedente, è il caso dell'elemento "utenti", che può contenere o uno o più elementi di tipo "utente": <!ELEMENT utenti (utente*)>;
- + indica che l'elemento deve essere presente almeno una volta. Si consideri il solito esempio e si voglia rendere possibile l'accesso ad utenti che posseggono più di un nome (ma almeno uno), il codice sarebbe: <!ELEMENT utente (nome+, cognome, indirizzo)> (in questo caso, si avrebbe anche uno ed un solo cognome ed uno ed un solo indirizzo);
- ? indica che l'elemento può essere presente o meno (zero o una occorrenze);
- #PCDATA (Parsed Character Data) indica che l'elemento contiene dati di testo leggibili da un analizzatore *XML* ed elaborati opportunamente.;
- #CDATA: indica una sequenza di caratteri di cui non verrà fatto il parsing;
- (element1 | element2): indica un blocco either/or ove solo uno dei due elementi può essere presente nella struttura dell'*XML* (ma almeno uno dei due). Può presentarsi con 2 o più alternative ed anche esso può avere degli operatori;
- contenuto misto: ciò che accade più di frequente è che ci sia un insieme di vario tipo di elementi che possono essere richiamati.

Per quanto riguarda invece la struttura sintattica per la definizione degli attributi, questa è inserita all'interno della dichiarazione di un elemento immediatamente sotto la dichiarazione degli elementi figli. La sintassi è la seguente:

<!ATTLIST ElementName AttributeName Type Default>

ElementName è il nome dell'elemento e AttributeName il nome dell'attributo.

Type identifica la tipologia di attributo identificato e può essere:

- CDATA: in questo caso i dati inseribili sono solo sotto forma di caratteri;
- ENTITY: il valore fa riferimento ad una entità binaria esterna dichiarata dal *DTD*;
- ENTITIES: equivalente ad ENTITY ma consente l'inserimento di più entità separate da spazi;
- ID: è un identificatore univoco (identificatori uguali per elementi diversi generano un errore in compilazione);
- IDREF: il valore è un riferimento ad un ID identificato in un qualunque punto del *DTD*;
- IDREFS: equivalente, ma con la possibilità di inserirne diversi separati da spazi;
- NMTOKEN: il valore consiste in una qualsiasi combinazione di token del nome, rappresentati da lettere, numeri, punti trattini, due punti o caratteri di sottolineatura;
- NMTOKENS: E' equivalente all'attributo NMTOKEN, ma consente l'utilizzo di più valori separati da spazi;
- NOTATION: Il valore dell'attributo deve fare un riferimento a un'annotazione dichiarata in un altro punto della *DTD*. La dichiarazione può anche essere costituita da un elenco di annotazioni. Il valore deve corrispondere a una delle annotazioni dell'elenco. Ogni annotazione deve avere la relativa dichiarazione nella *DTD*;
- Enumerated: Il valore dell'attributo deve corrispondere a uno dei valori inclusi. Ad esempio: <!ATTLIST MyAttribute (content1 | content2)>.

Default indica invece l'impostazione predefinita dell'attributo. I possibili valori sono:

- #REQUIRED: ogni elemento deve avere l'attributo marcato con questa parola chiave;
- #IMPLIED: attributo opzionale;
- #FIXED fixedvalue: questo valore deve avere il valore fixedvalue specificato;
- Default: indica un valore predefinito per un attributo. Se l'elemento non include l'attributo, viene stabilito il valore default.

Come si può intuire dalla sua sintassi, ad un documento *DTD* possono essere associati infiniti *XML*.

2.2.3 *Parser DTD*

Per realizzare gli *XML* da fornire come output si rendeva necessaria una conoscenza della struttura che questi potevano assumere. Tuttavia il problema non aveva ancora una formulazione così semplice, in quanto si supponeva che non tutti i dati potessero essere riconosciuti ed individuati nell'atto. Il software doveva quindi garantire intrinsecamente un meccanismo di rigido controllo nella creazione di un archivio che memorizzasse passo passo tutti i dati necessari, garantendo però all'utente elasticità nel poterli verificare ed eventualmente modificarli o aggiungerne. Per poter far questo si rendeva necessario l'utilizzo di una classe che facesse il parsing di un documento *DTD*, generando una struttura di dati che semplificasse la lettura e in modo da rendere lo schema consultabile velocemente ad ogni inserimento/rimozione dall'archivio di output. Si sono a tal proposito cercate delle librerie in *JAVA* che permettessero una lettura ed una traduzione di un file *DTD* in una struttura su cui fosse più semplice lavorare: una rappresentazione ad albero. Si evidenzia infatti che la caratteristica fondamentale intrinseca alla struttura di un *DTD* è quella di poter definire elementi annidati; ciascun elemento è infatti descritto da una lista di attributi e da una lista di elementi "figli". Per questo motivo la struttura dati più semplice da ipotizzare quando si lavora con questo genere di file è una struttura ad albero.

Le ricerche effettuate hanno portato ad evidenziare un fondamentale disequilibrio tra l'abbondante numero di librerie utilizzabili per la validazione di un *XML* tramite *DTD* e lo scarso numero di librerie che garantissero semplicemente ed esplicitamente un parsing del file *DTD*.

I due problemi sono fondamentalmente diversi, in quanto:

- Validare un *XML* significa, dato un *DTD* o un *XML* Schema, esaminare la struttura dell'*XML* in questione ed asserire se questo sia formulato correttamente, ovvero se la sua truttura grammaticale è corretta e rispetta le specifiche definite dal file direttivo.

Il problema che ci si poneva invece in questo caso era di poter effettuare una visualizzazione differente del formato *DTD* per eventualmente costruire un documento *XML* che sarebbe stato sicuramente validato dallo stesso file direttivo.

Purtroppo la letteratura in questione, seppur molto vasta nell'ambito della validazione *XML*, risulta piuttosto scarna per quanto riguarda questo secondo problema. Si sono studiate, in merito al linguaggio di programmazione *JAVA*, alcune librerie che realizzassero quanto auspicato.

2.2.4 *Librerie disponibili e necessità di una nuova libreria*

Le fondamentali librerie nelle quali ci si è imbattuti sono:

- *JDOM*. Questa libreria, già utilizzata per la realizzazione del nostro pacchetto software, garantisce in maniera semplice ed intuitiva la lettura e la creazione di un file . Tuttavia non dà accesso al file *DTD*. Questa libreria, seppur vasta (comprende entrambe le librerie *DOM* e *SAX*), non garantisce il parsing esplicito di un file direttivo e, in particolare, non rende possibile allo sviluppatore una lettura del *DTD* direttivo.

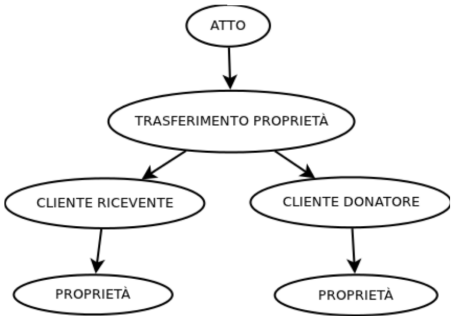
- *Jaxp*. Come *JDOM*, questa libreria permette una lettura di un file *DTD* interna relazionata alla validazione di un *XML*. Come *JDOM* non da quindi accesso ad una struttura dati con cui interagire per valutare la struttura del Document Type Definition.

- *oracle.xml.parser.v2*. Questo pacchetto, seppur indirizzato ad un parsing di file *XML*, offre la possibilità di leggere il documento *DTD* relativo all'*XML*. Esso dispone infatti di una utile classe denominata appunto *DTD* che, stando a quanto riporta la documentazione, contiene le informazioni del Document Type Definition relativo all'*XML* in questione. Tuttavia, questa classe non permetteva in alcun modo una lettura di tutto il documento *DTD*, ma solo delle informazioni relative ad un singolo nodo del file *XML*. Inoltre, partendo dalle conoscenze di quel nodo, non si rendeva possibile la visita di nodi vicini a quello di partenza, seppur esistessero metodi accessori dedicati (come *hasChildNodes()*, *getChildNodes()* etc.). Si era pensato infatti di utilizzare un file *XML* base da cui leggere le direttive successivamente. Tuttavia il modo in cui era stata implementata la classe non rendeva possibile questa soluzione: dato un nodo iniziale, non è infatti possibile scorrere l'albero definito dalla struttura del *DTD* in quanto ne viene negato l'accesso. Come si legge dalla documentazione al metodo *hasChildNodes()*: *"This is a convenience method to allow easy determination of whether a node has any children. Return false always, as DTD cannot have any overrides method in XMLNode"*. I problemi riscontrati con questa classe si riassumono quindi in una scelta degli sviluppatori di non rendere "troppo" leggibile il *DTD* associato ad un *XML* onde evitare una modifica allo stesso, che potrebbe compromettere la validazione dell'*XML* in questione. Questa realizzazione, per quanto condivisibile per un problema di validazione di un documento *XML*, rendeva però questo pacchetto non utilizzabile per la risoluzione del nostro problema.

- *Castor*. Si tratta di un progetto il quale si offre, oltre che di eseguire parsing e validazione di *XML*, di creare una rappresentazione di documenti *DTD* in modo permettendo di ricavare in maniera semplice tutte le informazioni relative ad un elemento descritto nel file direttivo. Tuttavia, questa descrizione, non viene fornita tramite una struttura dati in grado, dato un elemento, di individuare il ramo dell'albero nel quale esso è contenuto. Questa "mancanza", che non pregiudica l'utilità di queste classi per un generico utilizzo di file *DTD*, rende questa libreria una non esauriente soluzione del nostro problema: infatti il problema che ci si ritrova a fronteggiare non è quello di una semplice lettura di tutti gli elementi di un file *DTD*, ma quello di determinare, dati gli elementi riconosciuti, quali siano ancora inseribili o dei quali si richiede l'inserzione. Il problema indotto da questa classe può essere chiarito per mezzo di un esempio concreto. Sia data la struttura ad albero seguente:

Si supponga che questa struttura descriva un passaggio di proprietà tra i due clienti (tutti i nodi sono necessari e con unica ricorrenza, senza ulteriori nodi facoltativi nè contenuti misti); le due proprietà del cliente ricevente e del cliente donatore sono diverse, determinano lo scambio e sono descritte entrambe dall'elemento "proprietà". Si supponga inoltre che tutti i nodi siano necessari e che siano stati individuati nel testo

Figura 2: Rappresentazione ambigua nell'utilizzazione del pacchetto Castor



dell'atto tutti ad eccezione del nodo "proprietà", figlio del nodo "cliente ricevente". Il problema cui si incorre è di identificare, data la struttura, l'elemento mancante di cui richiedere le specifiche all'utilizzatore finale. La libreria in esame non da la possibilità, non memorizzando o comunque non rendendo accessibile la struttura dati, di determinare il padre del nodo mancante per determinarne la corretta posizione di inserimento (in questo modo abbiamo tutte le informazioni riguardo al nodo "proprietà", ma non abbiamo conoscenza riguardo quale proprietà mancano informazioni); in parole semplici non ci viene fornita l'informazione "qual è la proprietà che manca?". Certamente si sarebbe potuto risolvere questo problema partendo dalla radice e verificare la presenza del nodo proprietà utilizzando solo attraversamenti dall'alto verso il basso. È chiaro che la soluzione risulta non ottimale nel caso di alberi con un numero di nodi maggiore e nel caso in cui questa operazione dovesse essere eseguità un elevato numero di volte (come nel nostro caso);

La presa coscienza di questa limitatezza ci ha portato alla fruizione di questa libreria, appoggiandola ad una struttura nuova, di cui si farà riferimento in seguito.

- *matra*. Questa libreria si proponeva di realizzare, dato un documento *DTD*, una struttura ad albero associata. Sembrava quindi essere adatta alla risoluzione del problema. Tuttavia, quando si è provato a realizzare una classe di prova, si sono verificate spiacevoli situazioni dove il programma non riusciva ad identificare la corretta locazione del file di cui fare il parsing (in particolare, utilizzando un sistema UNIX nel progetto, il percorso veniva modificato dal programma così da modificare "*/path/from/root/to/file.dtd*" in "*/path/from/root/to\file.dtd*"). Questo lasciava presupporre che vi fossero state delle modifiche interne al codice che non modificavano il funzionamento, mantenendolo compatibile con un sistema operativo Windows (il sistema operativo Windows, a differenza di un SO Linux utilizza "\" al posto di "/"). Questo rendeva il programma non utilizzabile nel pacchetto eLaw, in quanto prerogativa fondamentale era che il sistema operativo fosse basato su UNIX ed Open Source. Inoltre, non mettendoci l'autore a disposizione i file sorgenti, non si è potuto mettere mano al codice per rendere la libreria utile ai nostri scopi.

Non avendo quindi trovato librerie che risolvessero interamente i nostri problemi, si è pensato di creare una libreria base che, appoggiandosi

per quanto possibile alle risorse software a nostra disposizione, ci permettesse di leggere un file direttivo e crearne una struttura ad albero utilizzabile semplicemente e facilmente integrabile con il resto del pacchetto software. Tuttavia, il tempo e le conoscenze che il progetto richiedeva erano troppo al di fuori delle risorse a disposizione, perciò si è optato per una creazione basilare che si proponesse di esaminare esclusivamente file che sarebbero stati utilizzati nell'ambito del progetto eLaw (limitatamente al task riguardante il plugin per OpenOffice), tutto questo con l'auspicio di poter continuare il progetto per rendere la libreria più completa ed utilizzabile per qualunque tipo di utilizzo.

3

MEMORIZZAZIONE DEI DATI

Seguendo lo sviluppo naturale di un processo notarile, il primo incontro con i dati si ha al livello di scrittura dell'atto. Alcuni dei campi necessari per la compilazione dei file di output sono quindi ivi rintracciabili tramite un parser che scorra e cerchi espressioni regolari che li determinino. Una volta individuate queste keywords (tramite il parser o manualmente dall'utente tramite interfaccia grafica), se ne rende necessaria una rapida memorizzazione in modo da poter svolgere ulteriori funzioni sempre a livello di scrittura dell'atto. Questo è il ruolo del FieldArchive.

3.1 REQUISITI

I requisiti chiave che questo archivio temporaneo si propone sono di:

1. memorizzare più valori per una stessa chiave;
2. collegare semplicemente più elementi con caratteristiche in comune (ad esempio, si rende necessaria la correlazione di "nome", "cognome" e "codice fiscale" di una stessa persona);
3. estire rapidamente le interconnessioni tra tutti gli elementi con una stessa chiave (tutti i codici fiscali, tutti i nomi e tutti i cognomi delle persone cui si fa riferimento nell'atto);
4. gestire situazioni particolari (che altrimenti genererebbero sgradevoli errori o malfunzionamenti) come la rimozione di un campo taggato dal testo.
5. tenere a memoria gli eventuali errori nel riconoscimento automatico, di modo da non incorrere nei medesimi ad un nuova scansione del parser nel testo.

3.2 IL FUNZIONAMENTO DI *fieldarchive*

3.2.1 *Formulazione delle classi*

Il pacchetto contiente le seguenti classi:

1. *FieldValue*: questa è la classe che descrive l'oggetto principale nella forma nella quale viene memorizzato nell'archivio. Consiste di:
   a) una coppia di *XTextRange* (elementi che rappresentano una posizione nel documento di testo), che puntano all'inizio e alla fine del campo;
   b) un *XText* contenente il riferimento all'istanza chiamata dal documento di testo;
   c) una variabile booleana che asserisce se l'elemento non era stato riconosciuto in precedenza.

2. *RecognizedFieldList*: questa consiste in una lista che tiene memoria degli elementi erroneamente riconosciuti sotto forma di *Item* in modo che questi non vengano nuovamente riconosciuti o associati ad una stessa chiave;

3. *Item*: si tratta di una coppia *(String, FieldValue)*. La stringa fa riferimento alla chiave cui era stato associato erroneamente quel campo;

4. *FieldDictionary*: questa classe descrive una *Hashtable* contenente, per ogni stringa (chiave) una *LinkedList<FieldValue>*. Questo oggetto è utilizzato per soddisfare i requisiti 1 e 2. Ciascun elemento riconosciuto nel testo verrà inserito in questo dizionario. La chiave corrisponde al tag cui viene associato quel cursore, e corrisponde alla descrizione dell'attributo nel *DTD*. Prima di performare l'inserimento, controlla che il *FieldValue* non sia presente nella *RecognizedFieldList*;

5. *InterconnectedFieldValue*: questa classe contiene la descrizione di una *Hashtable*. La chiave viene passata sotto forma di stringa e memorizza elementi di tipo *FieldValue*. Rappresenta un elemento del file *DTD*: le chiavi equivalgono agli attributi descritti nell'elemento, di cui la stringa contenuta nel *FieldValue* associato (che è lo stesso presente nel dizionario) ne è valore;

6. *FieldArchive*: è la classe principale tramite la quale si gestiscono tutte le azioni ed i metodi delle altre. Rappresenta un contenitore di *InterconnectedFieldValue* sotto forma di *Hashtable*. La chiave è una stringa e rappresenta il nome dell'elemento descritto nel *DTD*. Questo oggetto contiene come variabile di istanza privata un *FieldDictionary*.

La struttura appena descritta suggerisce la seguente procedura di inserimento:

1. La ricerca nel testo individua alcuni campi;

2. i campi vengono inseriti nel *FieldDictionary* come riconosciuti;

3. l'utente corregge eventuali errori che potrebbero verificarsi nel riconoscimento ed aggiunge i campi non riconosciuti;

4. vengono creati gli *InterconnectedFieldValue* necessari;

5. l'utente associa i determinati campi trovati ai vari elementi (movimento di un *FieldValue* da *FieldDictionary* a rispettivo *InterconnectedFieldValue*);

6. i *FieldValue* vengono "copiati" dal dizionario negli *InterconnectedFieldValue*.

Il linguaggio di programmazione utilizzato è *JAVA*. Questo consente di non memorizzare più copie dello stesso elemento, ma, al contrario, garantisce un passaggio dell'informazione per riferimento. In questo modo l'intervento di "copiatura" si riferisce al solo puntatore. Questa caratteristica presenta due rilevanti conseguenze:

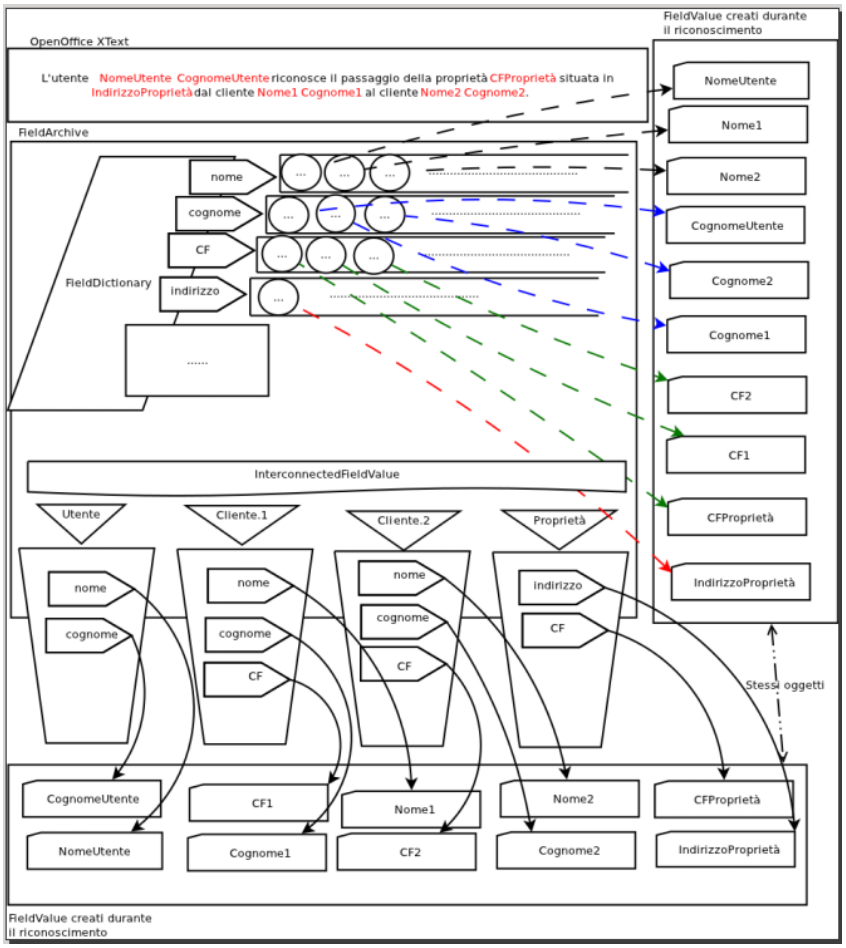
1. garantisce un aggiornamento automatico dei dati memorizzati nell'archivio ad una qualunque modifica nel testo;

2. in caso di cancellazione di un elemento nel testo, i puntatori salvati nell'archivio si riferirebbero ad elementi non più presenti, dando origine a situazioni di errore[1].

3.2.2 *Rappresentazione grafica*

Si arriverà ad avere infine una struttura rappresentabile nel modo seguente:

Figura 3: Rappresentazione del completo funzionamento di *FieldArchive*



Gli elementi riconosciuti (in rosso) si presentano nel testo (marcato con etichetta OpenOffice *xText*), quindi vengono memorizzati nel *FieldArchive*, dapprima a livello di *FieldDictionary*, dove vengono riunite tutte le chiavi uguali, poi copiati negli *InterconnectedFieldValue* (rappresentati come dei cestini), che racchiudono tutte le chiavi relative ad uno stesso elemento.

3.2.3 *Come utilizzare FieldArchive*

Le classi *FieldDictionary*, *InterconnectedFieldValue* e *FieldArchive* estendono la classe *Hashtable* presente nelle librerie *JAVA*. Per questa ragione ereditano tutti i suoi metodi. Le principali azioni sono eseguite come segue:

1 In seguito si spiegherà come vengono gestite queste situazioni di errore

- È sufficiente creare solo *FieldArchive*. Tutti gli altri oggetti relativi verranno automaticamente generati (a meno che non si voglia specificare il *FieldDictionary* di partenza per il *FieldArchive*): per modificare il *FieldDictionary* sono stati creati i metodi *getter* e *setter*, e comunque esiste un costruttore apposito della classe in oggetto che riceve come parametro il *FieldDictionary* da memorizzare.
- L'inserimento nel dizionario è semplicemente definito dal metodo *put(String chiave, FieldValue valore)*, il quale ritorna il *FieldValue* appena inserito (*null* se era già presente).
- L'inserimento di un nuovo *InterconnectedFieldValue* avviene tramite il metodo *put(String chiave, InterconnectedFieldValue valore)*. Questo richiama il metodo della superclasse.
- La rimozione dal dizionario avviene tramite il metodo *remove(String chiave, FieldValue valore)*. La rimozione performa anche l'inserimento dell'entry in *RecognizedFieldList*; per questa ragione non verrà più inserita se riconosciuta in fase di parsing.
- L'inserimento di un valore dal *FieldDictionary* in un *InterconnectedFieldValue* avviene per mezzo del metodo *putFromDictionaryIntoMap*. Questo metodo, presente in duplice copia (nel caso si abbia a disposizione la chiave o l'*InterconnectedFieldValue* stesso), consente di avere una collezione di Mappe, in cui ogni elemento è associato ad una sola chiave.

Conseguenze di questa struttura

Il metodo *get()* richiamato da *FieldArchive* ritorna una *InterconnectedFieldValue*. Per ottenere un qualunque *FieldValue* si rende necessario un ulteriore passaggio. Richiamando il metodo *get()* dell'*InterconnectedFieldValue* ottenuto, si otterrà il *FieldValue* desiderato. Questo procedimento evidenzia la necessità di disporre di due livelli gerarchici di conoscenza (cui si farà riferimento nel seguito della trattazione): uno relativo all'*InterconnectedFieldValue* ed uno relativo al tag associato alla stringa riconosciuta. L'idea di questa metodologia di sviluppo è di rendere l'archivio il più simile possibile alla struttura fisica del file di output descritto nel documento *DTD*: in questo modo infatti un elemento è definito dalla coppia *(InterconnectedFieldValue, chiave1)*, e un attributo dalla chiave *(FieldValue, chiave2)*, dove la numerazione indica il livello gerarchico di conoscenza: dapprima si richiede l'elemento, successivamente, dell'elemento, si richiede l'attributo.

Cosa succede se un cursore memorizzato nell'archivio viene eliminato?

Questa situazione, già precedentemente accennata, manifesta un errore e genera un'eccezione generica di OpenOffice. La questione è stata gestita tramite un metodo interno a *FieldValue*, *getValue()*, che deve viene richiamato ad ogni azione nelle classi di *FieldArchive*. Questo metodo verifica il valore dei cursori e, se ci si imbatte in questa situazione di errore, il cursore viene collassato all'inizio del testo ed assume uno stato particolare (valore sentinella), riconosciuto in tutti i metodi delle classi descritte che quindi eliminano ogni riferimento all'oggetto dall'archivio. In questo modo, la gestione del problema è stato trasferito ad un livello superiore di modo che, ad ogni invocazione di un metodo delle classe *FieldDictionary*, *FieldArchive* o *InterconnectedFieldValue*, tutti i

riferimenti per i quali si deve passare vengono esaminati ed eliminati se presentano la caratteristica definita sopra. Allo scopo di eliminare questa condizione di errore è stato creato anche un il metodo *refresh()* che scansiona tutti gli elementi salvati nell'archivio verificandone il valore.

3.3 ANALISI PRESTAZIONALE

L'analisi prestazionale prevede la suddivisione in due parti:

- Analisi dell'inserimento nel *FieldDictionary*;
- Analisi dell'inserimento da *FieldDictionary* a *InterconnectedFieldValue* (tramite il metodo *putFromDictionaryIntoMap*).

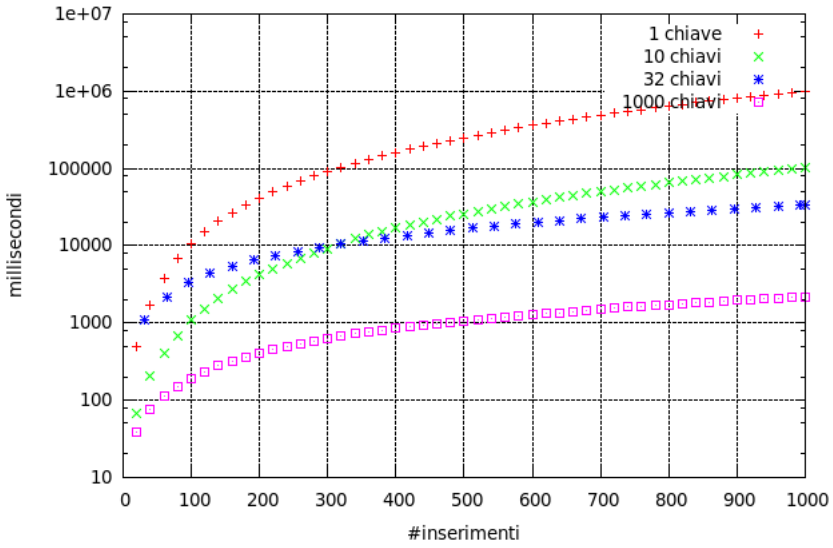
3.3.1 *Inserimento nel FieldDictionary*

L'analisi dell'inserimento tiene conto di due fattori che influenzano la resa del software:

- Il numero di elementi inseriti;
- Il numero di chiavi nelle quali si suddividono le *Entries*.

Questi sono i risultati delle misurazioni ottenute:

Figura 4: Rappresentazione del tempo di inserimento di valori nel *FieldDictionary* (scala logaritmica)



Questi grafici indicano il tempo sperimentale di inserimendo di 1000 elementi nella parte dizionario dell'archivio.

Nota: l'inserimento è stato effettuato in modo da riempire omogeneamente le chiavi, in modo che lo scarto del numero di elementi tra due liste di uno stesso archivio fosse al più uno.

Come si può notare, il caso peggiore si ha nel caso in cui tutte le 1000 inserzioni fanno riferimento alla medesima chiave. Il trend generale invece è quadratico. Questo trend è spiegabile con l'analisi del collo di bottiglia. L'archivio controlla prima di ogni inserzione che l'elemento (un FieldValue, ovvero una sorta di cursore) non sia già presente nel dizionario con quella chiave assegnata, quindi effettua due controlli:

1. Controlla che la chiave sia stata inserita (scorre tutte le chiavi);

2. Se la chiave è stata già inserita, prende la lista ad essa associata, e controlla che l'elemento che si va ad inserire non sia già presente nella struttura.

Questa analisi da una spiegazione chiara anche all'incrocio presente nel secondo grafico tra linea verde e linea blu. Infatti sotto i 320 inserimeti, l'archivio con 10 chiavi ha prestazioni migliori di quello a 32 (come quello a singola chiave), questo perchè il collo di bottiglia è dato dalla "dimensione" maggiore tra numero di chiavi e lunghezza della lista: ciò che impiega più tempo è il controllo che coinvolge più elementi tra quello relativo alle chiavi o quello relativo ai cursori inseriti. La tabella di seguito esemplifica la situazione chiarendo, nei due casi ad analisi, il numero di chiavi utilizzate e il numero di elementi della lista più lunga.

Tabella 1: Rappresentazione dell'andamento degli inserimenti nell'archivio a 32 e 10 chiavi fino al momento dell'incrocio rappresentata in figura 3

#inserimenti	10 chiavi	32 chiavi	max
20	10 chiavi | 2 elementi	20 chiavi | 1 elemento	20 chiavi
40	10 chiavi | 4 elementi	32 chiavi | 2 elementi	32 chiavi
60	10 chiavi | 6 elementi	32 chiavi | 2 elementi	32 chiavi
80	10 chiavi | 8 elementi	32 chiavi | 3 elementi	32 chiavi
100	10 chiavi | 10 elementi	32 chiavi | 4 elementi	32 chiavi
120	10 chiavi | 12 elementi	32 chiavi | 4 elementi	32 chiavi
140	10 chiavi | 14 elementi	32 chiavi | 5 elementi	32 chiavi
160	10 chiavi | 16 elementi	32 chiavi | 5 elementi	32 chiavi
180	10 chiavi | 18 elementi	32 chiavi | 6 elementi	32 chiavi
200	10 chiavi | 20 elementi	32 chiavi | 7 elementi	32 chiavi
220	10 chiavi | 22 elementi	32 chiavi | 7 elementi	32 chiavi
240	10 chiavi | 24 elementi	32 chiavi | 8 elementi	32 chiavi
260	10 chiavi | 26 elementi	32 chiavi | 9 elementi	32 chiavi
280	10 chiavi | 28 elementi	32 chiavi | 9 elementi	32 chiavi
300	10 chiavi | 30 elementi	32 chiavi | 10 elementi	32 chiavi
320	10 chiavi | 32 elementi	32 chiavi | 10 elementi	uguali
340	10 chiavi | 34 elementi	32 chiavi | 11 elementi	34 elementi

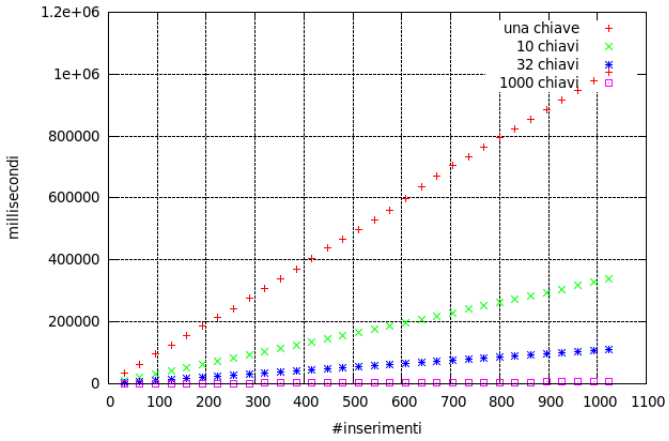
Si stima che un atto notarile su cui potrebbe essere utilizzato il software non possiede in genere più di 6/7 chiavi e non si identificano più di 300/400 campi da memorizzare.

3.3.2 *Passaggio di riferimento da FieldDictionary a InterconnectedFieldValue*

L'analisi di questa parte si incentra sui seguenti parametri: numero di chiavi, numero di passaggi di riferimenti, numero di elementi presenti nel *FieldDictionary*. Risulta logico aspettarsi che, al diminuire della dimensione delle liste e del numero di elementi memorizzati nel dizionario, il tempo di esecuzione cali, infatti:

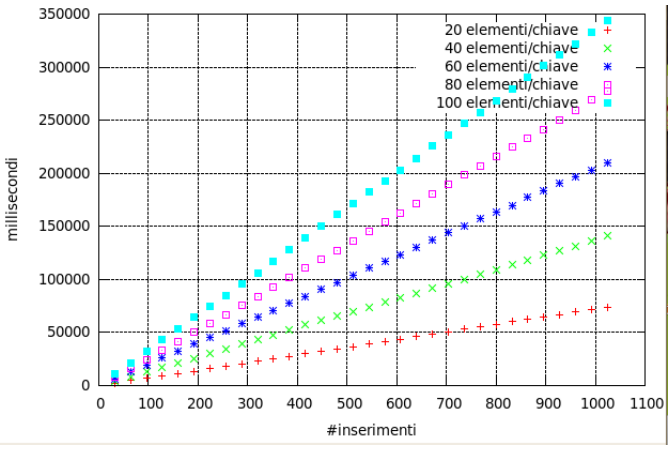
Il grafico di figura 6 rappresenta il tempo impiegato per passare i riferimenti da *FieldDictionary* a *InterconnectedFieldValue* per archivi di 1000 elementi omogeneamente distribuiti tra le chiavi.

Figura 5: Rappresentazione nel passaggio di riferimenti da *FieldDictionary* a *InterconnectedFieldValue*



Si nota immediatamente l'andamento lineare. Tenendo invece in considerazione il solo archivio con 10 chiavi (caso più simile a quello che si verifica) si ha un andamento simile a quello di figura 6.

Figura 6: Rappresentazione nel passaggio di riferimenti da *FieldDictionary* a *InterconnectedFieldValue* con dizionario da 10 chiavi



Questo grafico indica il tempo impiegato dall'archivio per passare riferimenti da *FieldDictionary* a *InterconnectedFieldValue* al variare del numero di elementi per chiave.

Nota: queste misurazioni sono state effettuate con i seguenti criteri:

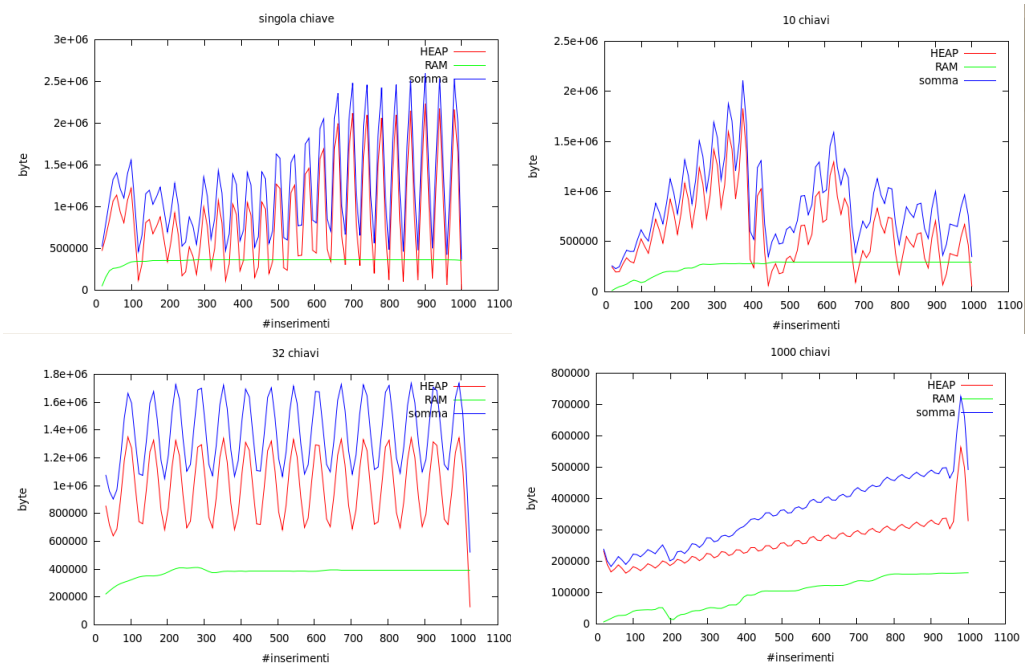
- le chiavi del dizionario erano riempite uniformemente (non esistono due liste di uno stesso dizionario le cui lunghezze differiscano per più di un'unità);
- i riferimenti vengono passati uniformemente in 32 *InterconnectedFieldValue*.

3.3.3 *Spazio occupato*

Nelle pagine seguenti si propongono i grafici relativi il consumo di memoria dell'archivio nelle due fasi. Le misurazioni evidenziano il numero di byte utilizzati dall'archivio nell'inserimento di valori. I grafici tengono traccia della memoria occupata nello *HEAP* della *JVM* e nella *RAM*. Si nota immediatamente che l'ammontare della memoria allocata (tra *HEAP* e *RAM*) non supera il picco di 2.5 MB, mentre l'ammontare della memoria *RAM* utilizzata rimane sempre sotto la soglia di 500KB.

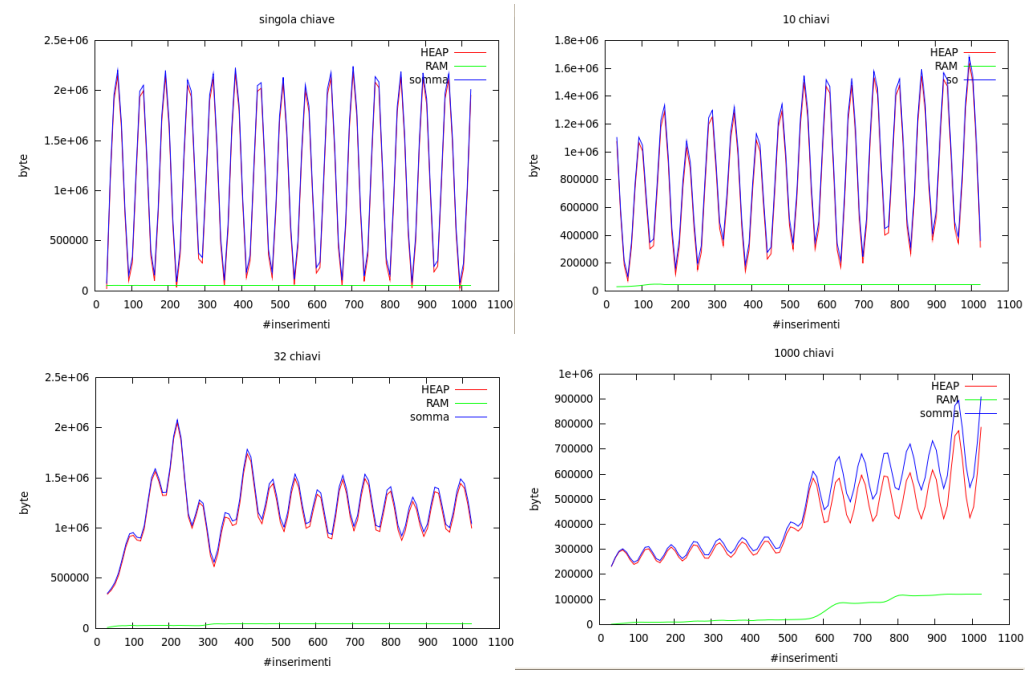
Inserimento nel Dizionario:

Figura 7: Rappresentazione Spazio occupato in *HEAP*, *RAM* e somma dei due nei *FieldDictionary*, rispettivamente con 1, 10, 32, 1000 chiavi



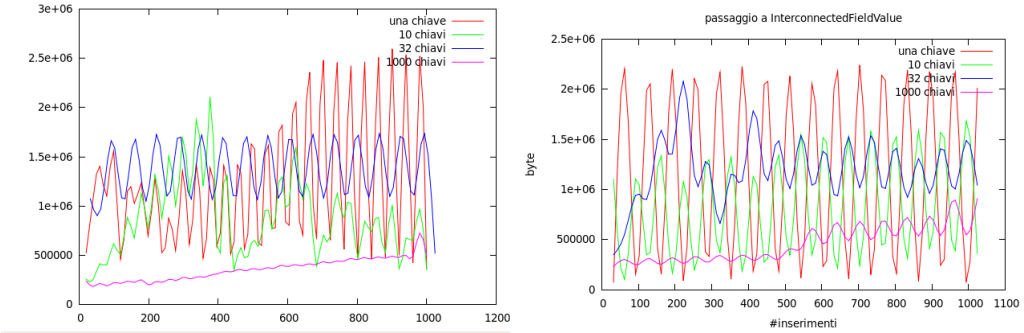
Passaggio di riferimento dal *FieldDictionary* agli *InterconnectedFieldValues*:

Figura 8: Rappresentazione del passaggio di riferimenti da *FieldDictionary* a *InterconnectedFieldValue* (tempi), rispettivamente per *FieldDictionary* con 1, 10, 32, 1000 chiavi



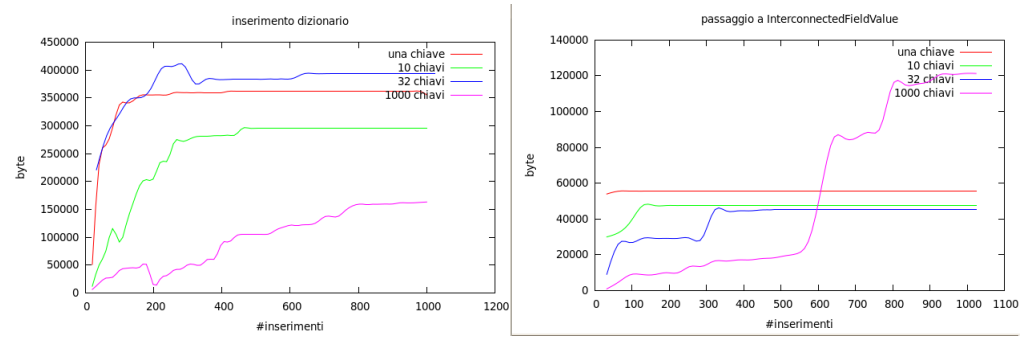
Confronto dell'incremento di memoria della somma tra *HEAP* e *RAM*:

Figura 9: Confronto prestazionale in termini di memoria occupata (somma *HEAP+RAM*): inserimento in *FieldDictionary* a sinistra e passaggio a *InterconnectedFieldValue* a destra



Confronto relativo al solo incremento della memoria *RAM*:

Figura 10: Confronto prestazionale solo per l'incremento di memoria *RAM*: inserimento in *FieldDictionary* a sinistra e passaggio a *InterconnectedFieldValue* a destra



Considerazioni: il quantitativo di memoria utilizzato dall'archivio è praticamente trascurabile (si consideri che solo il processore di testi su cui si sono eseguite le misurazioni occupava 1254520byte: tre volte la quantità di *RAM* occupata dall'archivio pieno nel caso peggiore). Questi risultati sono riconducibili alla caratteristica tipica del linguaggio di programmazione *JAVA*, sfruttata nella creazione dell'archivio: la memorizzazione per riferimento; in questo modo ciò che realmente si memorizza non sono oggetti ma puntatori.

4 PREPARAZIONE DELL'OUTPUT

Una volta che sono stati individuati e memorizzati (nel *FieldArchive*) tutti i campi salienti presenti nell'atto, si passa alla fase successiva: la creazione dei vincoli e delle relazioni tra questi campi.

Come precedentemente esposto, l'output che si deve arrivare a creare consiste di un file *XML*, che descrive l'atto in un formato rigido (descritto dal suo direttivo *DTD*), di modo che questo possa essere accettato dagli uffici competenti. Tuttavia ci si ritrova difronte a diversi ostacoli:

- Non si ha la certezza che tutti i campi siano stati trovati;
- Non si ha la certezza che tutti i campi necessari alla compilazione dei file di output siano effettivamente presenti nell'atto;
- Gli *InterconnectedFieldValue* creati e riempiti rimangono delle entità ben divise tra loro: non se ne conoscono le relazioni.

Queste limitazioni hanno portato alla necessità di definire un nuovo tipo di archivio, che non lavorasse più su cursori ma su stringhe (offrendo la possibilità di inserire elementi non presenti nell'atto) e che gestisse gli inserimenti nell'archivio memorizzando le relazioni tra gli *InterconnectedFieldValue* (già presenti o da creare) appoggiandosi ad una struttura dati che descrivesse le possibili relazioni tra i vari elementi[1]. Si è dovuto pertanto creare un secondo archivio: l'*OutputArchive*.

In questa parte si descriverà la struttura spiegando come si è resa possibile la realizzazione dell'output nella sua formulazione corretta.

4.1 REQUISITI

Le tipologie di atto sono molteplici. Dalla scansione dell'atto risulta difficile capire che tipo di atto si stia redigendo. Per questo motivo non risulta semplice definire neanche una bozza della possibile struttura che l'output debba avere. L'unica informazione di cui si dispone, a questo livello, è quella relativa ai campi identificati nel testo dell'atto (che comunque non sono tutti i campi necessari nè tantomeno sono tutti necessari). Questi campi individuati rappresentano gli attributi necessari alla compilazione dell'*XML*. L'inserimento negli *InterconnectedFieldValues* (in parte applicabile automaticamente nella fase di parsing del testo dell'atto) contribuisce a dare una relazione tra alcuni degli attributi individuati.

Tutte le informazioni di cui si è dunque in possesso si riducono ad una manciata di elementi, utili ed inutili (con alcuni elementi necessari non presenti); ciò che ci si richiede di fare è di ricostruire, sfruttando al massimo questi elementi, una struttura di output valida secondo le normative descritte dal *DTD* direttivo.

La metodologia con cui si è pensato di adempire a quanto richiesto si basa su una costruzione passo passo dell'output, partendo dagli elementi che sicuramente dovranno essere inseriti ed aggiungendo, via via, con l'ausilio dell'utente, le informazioni mancanti. Per utilizzare questa

1 nel seguito della trattazione si farà riferimento agli *InterconnectedFieldValue* come Elementi (di un *DTD*) e ai *FieldValue* come Attributi (di un *DTD*)

strategia si è resa necessaria la creazione di una struttura d'appoggio che consenta una rapida lettura del file *DTD* direttivo.

Alla luce di quanto descritto, si possono riassumere i requisiti fondamentali nei seguenti punti:

- Necessità di avere un semplice parallelismo tra struttura di un *XML* e FieldArchive
- Necessità di avere una struttura che "imiti" sempre un *XML* ***validabile*** secondo il *DTD* di riferimento;
- Deve memorizzare Stringhe (così da poter rappresentare anche gli elementi non presenti nell'atto e quindi non inseribili in *FieldArchive*);
- Deve quindi essere una struttura piuttosto rigida, ma che consenta una interazione con l'utente, il quale deve poter aggiungere e rimuovere campi/attributi non correttamente individuati;

4.2 LA STRUTTURA DI *outputarchive*

4.2.1 *La struttura d'appoggio*

La struttura cui si fa riferimento consiste di una descrizione delle direttive imposte dal *DTD*. Come precedentemente illustrato, un documento *DTD* è un documento atto a descrivere le possibili interrelazioni tra gli elementi al fine di generare un file *XML* Validato. A tal proposito si è ritenuta come struttura dati più adatta ai nostri scopi un albero, di modo da tener traccia, dato un qualunque elemento, del proprio padre, così da garantire una semplice analisi relazionale (garantendo una rapida ricostruzione della linea genealogica di ogni singolo nodo).

Per poter descrivere un documento *DTD*, l'albero deve memorizzare:

- le relazioni di parentela tra nodi (caratteristica data dalla struttura ad albero);
- la tipologia di un elemento (ovvero quante volte è inseribile);
- gli attributi che un elemento possiede;
- tutte le informazioni necessarie a mappare un nodo *Either/Or* (e quindi la conoscenza della gerarchia con cui si presentano nodi fratelli).

Per la creazione di questa struttura si è fatto uso della libreria *Castor* nella fase di parsing del documento *DTD*.

La rappresentazione dell'albero

Nella realizzazione dell'archivio, si è cercato di rendere la rappresentazione di questa struttura il più possibile simile a quella leggibile nel documento *DTD*. Questa Struttura presenta due oggetti fondamentali:

- **Nodo:** ogni singolo nodo è rappresentato da:
  - un padre (*null* se il nodo è la radice);
  - una lista di figli (*LinkedList<DTDNode>*);

- l'informazione trasportata è rappresentata da un Element del pacchetto *Castor* (consiste della descrizione di un elemento dichiarato nel *DTD* e comprende gli elementi figli e tutti gli attributi);
- un riferimento al documento *DTD* cui si sta facendo riferimento (*DTDdocument* del pacchetto *Castor*);
- un booleano d'appoggio;
- un *ContentParticle*[2] di appoggio;
- uno *Stack<ContentParticle>* statico (uno per tutti i *DTDNode* creati).

- **Albero:** questo è un insieme di nodi che tiene traccia di:
  - Nodo radice (costante);
  - un nodo temporaneo di appoggio;
  - un riferimento al documento *DTD* (*DTDDocument*);

Rappresentazione grafica della struttura

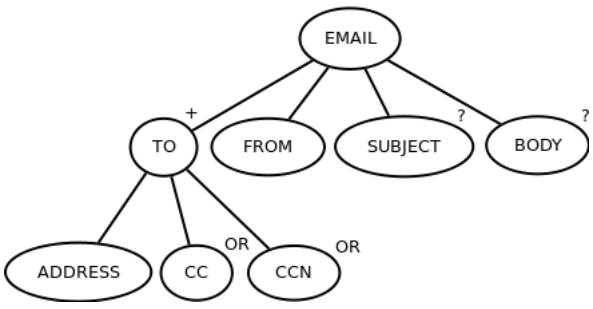
La struttura si può rappresentare nel modo di seguito esposto. Dato un file *DTD* che contiene il seguente codice:

Algorithm 4.1 Esempio di codice *DTD*

```
<!DOCTYPE EMAIL [
  <!ELEMENT EMAIL (TO+, FROM, SUBJECT?, BODY?)>
  <!ATTLIST EMAIL
            LANGUAGE(Western|Greek|Latin|Universal) " Western"
           ENCRYPTED CDATA #IMPLIED
           PRIORITY (NORMAL|LOW|HIGH) "NORMAL">
  <!ELEMENT TO (ADDRESS, (CC|CCN)*)>
  <!ELEMENT ADDRESS (#PCDATA)>
  <!ELEMENT CC (#PCDATA)>
  <!ELEMENT CCN (#PCDATA)>
  <!ELEMENT FROM (#PCDATA)>
  <!ELEMENT SUBJECT (#PCDATA)>
  <!ELEMENT BODY (#PCDATA)> ]>
```

La struttura dati sarà la seguente:

Figura 11: Rappresentazione ad albero del codice 4.1



dove +,? rappresentano la tipologia dell'elemento, OR rappresenta un flag inizializzato all'interno dell'oggetto *Element* del pacchetto *Castor* ed indica la presenza di un *ContentParticle* a scelta multipla, le scritte

2 un *ContentParticle* definisce una cella fondamentale: l'insieme di tutti gli elementi racchiusi tra parentesi; viene utilizzato da appoggio per ricavare ogni volta che risulta necessario la struttura gerarchica tra nodi fratelli.

dentro i nodi sono tutti *Element* del paccheto *Castor*. Si noti che il tipo * di CC e di CCN non è rappresentato, in quanto non è tipo dell'elemento CC o CCN ma del blocco (CC|CCN).

Memorizzazione delle informazioni fondamentali mediante questa struttura

Una struttura così descritta permette di memorizzare tutte le informazioni che ci interessano relative al documento *DTD* garantendo un rapido accesso a ciascuna di esse.

Per quanto concerne le prime tre necessità, queste trovano semplice risposta:

- Le relazioni tra i nodi sono descritti dai legami padre-figlio;
- Il numero di inserimenti possibili di un elemento è un'informazione incapsulata nell'oggetto *Element* del pacchetto *Castor* e quindi semplicemente reperibile;
- Gli attributi di un elemento sono come prima memorizzati nell'oggetto *Element* del pacchetto *Castor*;

Il discorso si complica invece per quanto riguarda l'individuazione della gerarchia tra nodi fratelli (figli di uno stesso nodo) nel caso di blocchi *EITHER/OR*. Il problema trova una formulazione più semplice come segue:

- **Dato un nodo *Either/Or* come fare per ottenere tutte le informazioni sui nodi/gruppi di nodi che sono coinvolti nello stesso blocco?**

O meglio ancora:

- **Una volta inserito un nodo *Either/Or* ad inserimento unico, come faccio a rintracciare tutti i nodi che non saranno più inseribili?**

Si è deciso di risolvere questo problema durante l'esecuzione del programma interrogando, quando richiesto, il padre del nodo interessato ed agendo ricorsivamente utilizzando il metodo descritto dall'algoritmo 4.2.

Alla fine della ricorsione, lo *stack* di appoggio contiene la gerarchia degli elementi iteressati nel blocco *Either/Or*. Un rapido check dello *stack* consente di visualizzare tutte le informazioni interessanti al momento dell'inserimento: il numero di *pop* dallo *stack* indica il livello della gerarchia.

In questo modo, lo *stack* dell'esempio precedente relativo al frammento di codice 4.3.

verrà rappresentato, tramite una ricorsione col metodo precedente partendo dall'elemento "TO" e relativamente all'elemento CC, nel modo raffigurato nella figura 12.

La scelta di rendere lo *stack* statico rende necessario il ricalcolo dello *stack* per ogni elemento *Either/Or* che viene inserito, ma consente di mantenere la struttura più leggera in termini di spazio occupato.

4.2.2 *L'archivio di output*

Nell'archivio di output vengono memorizzate tutte le informazioni necessarie alla compilazione del documento da inviare agli uffici competenti.

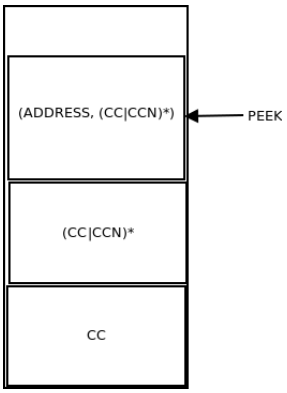
Algorithm 4.2 Metodo per l'impostazione ricorsiva dello *Stack* di appoggio

```
private boolean recursionToSetTMPContentParticleToLastLevel(
     ContentParticle cp){

// La ricorsione parte dal ContentParticle del padre del nodo interessato:
// this.setTMPContentParticleToLastLevel(parent.getElement().getContent())
     ;

   if (cp == null){  // Caso Base
         return tmpUtilBoolean;
   }
      boolean found = tmpUtilBoolean;
      if (found) {   // Caso Base
         return false;
      }

      if (cp.isReferenceType()) {   // Il ContentParticle contiene
                                    // un solo elemento: "(Element)"
         if (cp.getReference().equals(getName())) {

//Il nome del ContentParticle è quello del nodo da inserire nell'archivio

            tmpContentParticle = cp;
            tmpUtilBoolean = true;  // trovato
            OR_Stack.push(cp);   // lo inserisco nello stack di appoggio
            return true;
         } else
            return false;
      } else {
         Enumeration<ContentParticle> child = cp.getChildren();
         ContentParticle tmp = null;

// procedo ricorsivamente su tutti i ContentParticle figli di cp

         while (child.hasMoreElements() && !found) {
            tmp = child.nextElement();
            found = (recursionToSetTMPContentParticleToLastLevel(tmp) ||
                  tmpUtilBoolean);
         }
         if (found) {   // trovato dopo la ricorsione
            OR_Stack.push(cp);   // inserisco nello stack
            tmpContentParticle = tmp;
            tmpUtilBoolean = true;
         }
      }
   return found;
   }
```

Algorithm 4.3 Frammento di *DTD* in cui si rappresenta un blocco *Either/Or*

```
<!ELEMENT TO (ADDRESS, (CC|CCN)*)>
<!ELEMENT ADDRESS (#PCDATA)>
<!ELEMENT CC (#PCDATA)>
<!ELEMENT CCN (#PCDATA)>
```

Figura 12: Rappresentazione dello *Stack* di appoggio chiamato all'elemento "TO" nel frammento di codice 4.1



Dal momento che non tutti i campi possono risiedere nel testo dell'atto, non è possibile gestire una collezione di cursori, ma risulta necessario gestire un tipo di dato più "rigido": una Stringa. Questa considerazione, unitamente alla necessità di creare una struttura sicuramente accettata e all'impossibilità di disporre immediatamente di tutti gli elementi necessari alla compilazione dei documenti, ha condotto necessariamente alle seguenti conseguenze:

- La memorizzazione risulta rigida, e quindi l'archivio di Output deve essere costruito solo una volta ultimato e reso definitivo il testo dell'atto;
- L'archivio riempie tutte le chiavi obbligatorie (senza le quali il documento *XML* non sarebbe accettato), tuttavia, se una chiave non ha un cursore associato al corrispettivo elemento presente nel *FieldArchive*, questa viene associata ad un valore particolare: Stringa vuota.

L'idea fondamentale che sta alla base di suddetto archivio risiede nell'associazione di un di valore significativo alla chiave che memorizza la stringa. La chiave (Stringa) identifica l'attributo di cui si deve memorizzare il valore, l'elemento cui questo attributo è associato e la collocazione di questo elemento all'interno dell'albero gerarchico.

La struttura delle chiavi

Come accennato nell'introduzione della sottosezione, anche le chiavi utilizzate nell'archivio presentano un contenuto informativo di rilievo, memorizzando:

- il nome dell'attributo;
- il nome dell'elemento cui si riferisce l'attributo;
- il percorso dalla radice all'elemento in questione.

La memorizzazione di queste informazioni permette una semplice suddivisione tra gli elementi presenti nella mappa. Infatti si tenga presente che la struttura deve essere in grado di differenziare elementi uguali situati in locazioni diverse nell'albero dei *DTD* (Si faccia riferimento all'esempio riportato in).

Per tenere nota di quanto sopra, si è pensato di utilizzare una chiave della forma:

"*NomeElemento: Percorso/Dalla/Radice/All/Elemento/NomeElemento.attributo*", dove, dopo i ':' è presente l'indicazione del percorso seguito per l'inserimento di quell'elemento tramite un elenco dei nodi attraversati nell'albero per raggiungerlo.

Questa conformazione renderà molto agibile, come si vedrà nel seguito della trattazione, la creazione del file di output.

4.3 Interazione con *fieldarchive*: la creazione ed il metodo di inserimento

In questa sezione si cercherà di illustrare come avviene l'inserimento di un elemento nell'archivio di output, cercando, ove possibile, di estrapolare autonomamente le informazioni già presenti nel FieldArchive.

4.3.1 *Cosa hanno in comune FieldArchive e OutputArchive?*

Si ricorda che la struttura del *FieldArchive* permette una memorizzazione di *InterconnectedFieldValue*, ciascuno dei quali altro non è che una mappa contenente alcuni valori.

Questa struttura è stata pensata appositamente per suddividere in maniera rapida e semplice attributi (le chiavi di un *InterconnectedFieldValue*) ed elementi (gli *InterconnectedFieldValue* presenti nell'archivio); ciascun elemento possiede i suoi attributi: gli identificativi sono le chiavi associate (all'*InterconnectedFieldValue* nel caso degli elementi, a *FieldValue* nel caso di attributi) e i valori sono, per gli attributi, quanto presente nel testo dell'atto. Si è pensato infatti, nella realizzazione di *FieldArchive*, che una struttura di questo genere avrebbe potuto trasformare semplicemente, con l'aggiunta di alcune informazioni presenti nella struttura di appoggio, le chiavi di un *InterconnectedFieldValue* in chiavi di un attributo e la chiave di un *InterconnectedFieldValue* nella chiave di un elemento.

4.3.2 *Prerogative da rispettare nella creazione dell'archivio*

L'operazione di inserimento risulta complicata, dal momento che, prerogativa fondamentale da rispettare in fase di creazione, è di mantenere sempre una struttura valida secondo il documento *DTD* direttivo. A tal proposito si è scelta una strategia di "autocompletamento intelligente" che permettesse, ad ogni inserimento di un elemento, l'inserimento di tutte le chiavi necessarie a mantenere una struttura valida: ogni volta che un elemento viene inserito, l'archivio inserirà automaticamente altri elementi così da garantire che, in qualunque momento, l'output generato dall'archivio sia valido secondo il direttivo *DTD*.

Oltre a questa, si aggiunge la necessità di interrogare agilmente il FieldArchive così da passare automaticamente i valori presenti nel nuovo archivio.

4.3.3 *La struttura dell'archivio*

L'archivio di Output si presenta come una mappa che associa ad una stringa un'altra stringa. Tuttavia, per riuscire a rispettare le prerogative sopra esposte, si è reso necessario l'impiego di altre strutture. Di seguito si darà una rapida descrizione delle strutture che sono utilizzate da *OutputArchive*:

- Un riferimento ad un *FieldArchive* (necessario per l'inserimento automatico di quanto rinvenuto nel testo dell'atto);
- Un *XMLReader*: un *XMLReader* è una classe realizzata separatamente che avesse il ruolo di leggere un documento *XML* e, in fase di letture, di inserire singole *Entry* all'interno di *OutputArchive*. *XMLReader* è anche la classe che performa la sincronizzazione con *FieldArchive* e l'interrogazione della struttura di appoggio (*DTDTree*: si faccia riferimento alla (6.1)). Si è scelto di creare una classe separata al fine di un impacchettamento che suddividesse i compiti. Tale struttura contiene riferimenti a:
  - Il *FieldArchive* memorizzato in *OutputArchive*;
  - L'archivio *OutputArchive* che lo contiene;
  - un *DTDTree*: la struttura di appoggio di (6.1);
  - Un *DTDNode* di appoggio;
  - una mappa contenente come chiavi gli elementi inseribili (i nomi) e come valori una lista di puntatori ai rispettivi nodi nel *DTDTree*.

Oltre a queste strutture, per il funzionamento dell'archivio, devono essere presenti nell'hard disk dei file *XML* (uno per elemento) che descrivono gli elementi figli ed attributi obbligatori che devono essere necessariamente inseriti una volta che si chiama il metodo di inserimento sull'elemento in questione.

La presenza di questi file è condizionata dalla necessità di rendere l'archivio compatibile anche per gli altri programmi descritti nell'introduzione, i quali non utilizzano un file direttivo *DTD*.

Può essere utile, per capire la struttura dell'archivio ed il funzionamento del metodo di inserimento, fare riferimento alla seguente rappresentazione grafica illustrata in Figura 13.

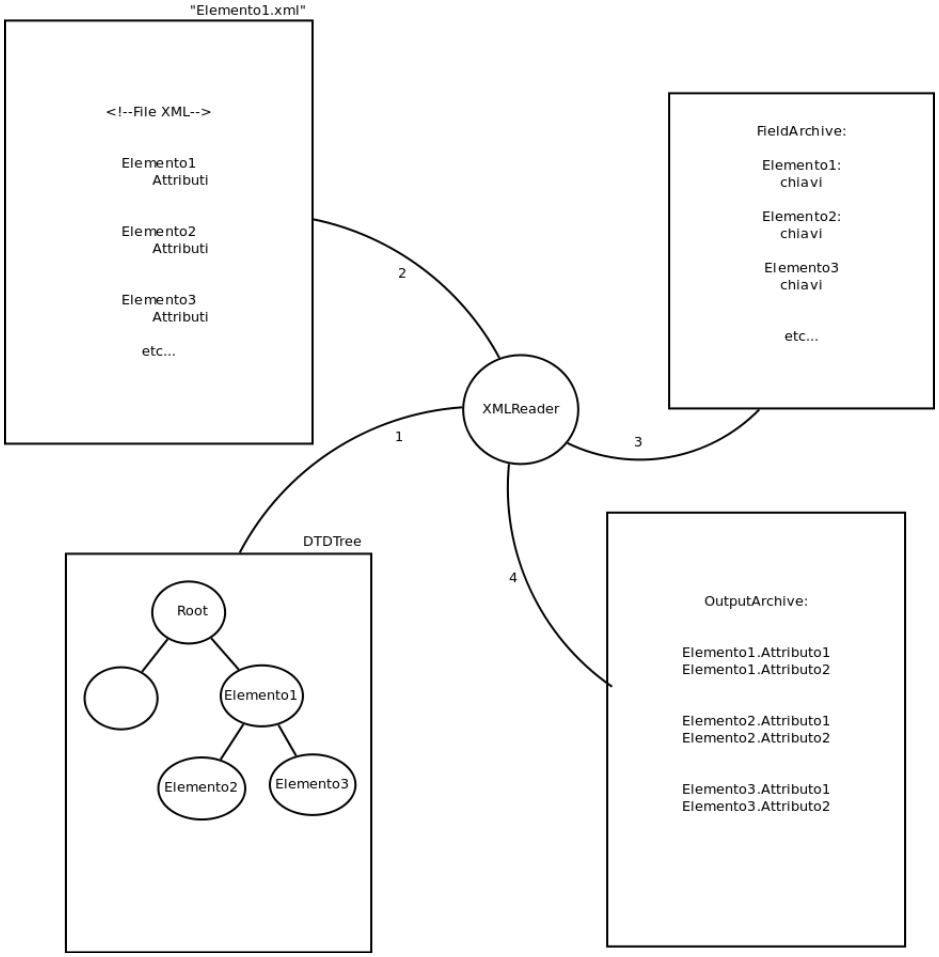
4.3.4 *Il funzionamento di OutputArchive*

In questa sezione si tratterà del funzionamento del suddetto archivio, con particolare riferimento al metodo di inserimento. Gli altri metodi necessari all'utilizzo dell'archivio sono ereditati da *HashTable* e presentano poche modifiche e comunque non rilevanti.

LA CREAZIONE DI *outputarchive*: L'INSERIMENTO In questa parte si cercherà di descrivere come avviene l'inserimento in questo archivio. Per semplificare la trattazione si descriveranno i passi fondamentali del metodo di inserimento suddivisi in fasi del processo che descrive il metodo *put()*.

Si tenga presente che il metodo in questione non accetta come parametro chiave e valore (quindi non è una sovrascrizione del metodo della

Figura 13: Rappresentazione Grafica di *OutputArchive*. La numerazione degli archi indica l'ordine in cui vengono lette le varie strutture di appoggio ad ogni chiamata del metodo di inserimento.



superclasse), ma solo una stringa che rappresenta il nome dell'elemento da inserire: *"put(String fileName)"*.

Ecco come agisce l'archivio alla chiamata del metodo *put("NomeElemento")* (i passi sono suddivisi in base all'arco di riferimento in figura 13);

Arco 1:

1. *XMLReader* cerca in *DTDTree* il nodo denominato *"NomeElemento"* (che rappresenta il nodo radice del file *XML* che si sta leggendo) e ne tiene un riferimento;
2. l'elemento *"NomeElemento"* è tra i nodi inseribili? (all'inizio l'unico nodo inseribile è la radice dell'albero dei *DTD*);
   - No: lancia un'eccezione specifica "*MissingFatherException*" e termina;
   - Sì: prosegue;
3. memorizza nel nodo di appoggio in *XMLReader* il padre del nodo che verrà inserito;
4. viene eliminato il nodo che verrà inserito dai nodi inseribili seguendo il seguente criterio:
   - Se il nodo appartiene ad un blocco *EITHER/OR*:
     - Vengono eliminati dalla mappa dei possibili nodi inseribili i nodi appartenenti al blocco se sono del tipo "" o "?" (ovvero se non può avere più di un'occorrenza);
   - Se il nodo presenta il tipo "" o "?" (ovvero se non può avere più di un'occorrenza) lo elimina dalla mappa dei possibili nodi inseribili;
5. vengono inseriti i nuovi nodi inseribili utilizzando una ricorsione, per tutti i nodi figli del nodo che si sta inserendo, sul seguente metodo:
   - Se questo nodo è del tipo: "+", "*" o "?" viene aggiunto ai nodi inseribili (il nodo può avere zero ricorrenze);
   - Se questo nodo è del tipo: "" o "+" (il nodo deve essere presente almeno una volta), fa ricorsione del punto 5 su tutti i suoi figli.

Arco 2:

1. Apre il file *XML* denominato *"NomeElemento.xml"* (che contiene la dichiarazione essenziale di tale elemento);
2. memorizza un riferimento al *RootElement* del file *XML* aperto;

Arco 3 e 4:

1. inserisce gli attributi e controlla se ci sono in *FieldArchive* degli *InterconnectedFieldValue* con chiave *"NomeElemento"* che abbiano come attributo gli attributi scritti per quell'elemento nell'*XML* aperto: se ci sono ne associa il valore trovato, altrimenti li associa alla Stringa vuota.
2. Ritorna sul punto 1 di questo arco per tutti gli elementi figli di *RootElement* fino a che il file *XML* non è stato completamente letto.

Dopo aver agito con questa procedura si ottengono i seguenti cambiamenti:

- Si sono aggiornati i nodi inseribili;
- Si sono inseriti tutti e soli i nodi obbligatori (ed i rispettivi attributi obbligatori);
- Si è associato, dove possibile, alla chiave di ogni attributo il valore rinvenuto in *FieldArchive* (il valore memorizzato sarà successivamente editabile, quel che importa è che la struttura generata sia corretta per quanto descritto nel direttivo *DTD*).

4.4 ANALISI PRESTAZIONALE

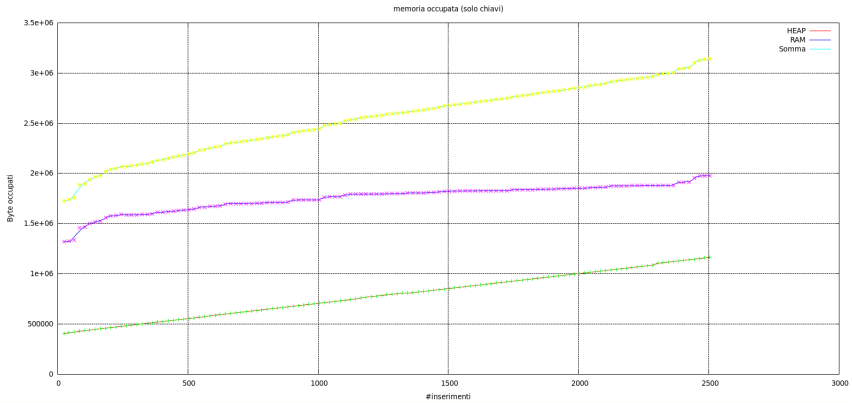
In questa sezione si tratterà, analogamente al *FieldArchive*, delle prestazioni in termine di velocità nella computazione e di memoria occupata dalla struttura. L'analisi verrà distinta in tre parti:

- Inserimento a *FieldArchive* vuoto (la sincronizzazione non avviene): analisi di memoria occupata;
- Inserimento a *FieldArchive* pieno (ad ogni inserimento, ci saranno valori in *FieldArchive* che dovranno essere passati nel nuovo archivio, associando quindi ad ogni chiave un valore diverso da Stringa vuota): analisi di memoria occupata.
- Confronto tra le due modalità descritte nei punti precedenti: memoria e tempo impiegato.

La creazione della struttura (compresa la creazione dell'albero dopo la lettura del *DTD* direttivo) impiega un tempo compreso tra i 55 e i 75 ms ed occupa una quantitativo di circa 300kB in entrambe le modalità sopra esposte.

4.4.1 *Inserimento a FieldArchive vuoto*

Figura 14: Memoria occupata da *OutputArchive* nel caso in cui *FieldArchive* sia vuoto, e di conseguenza i valori inseriti siano stringhe vuote

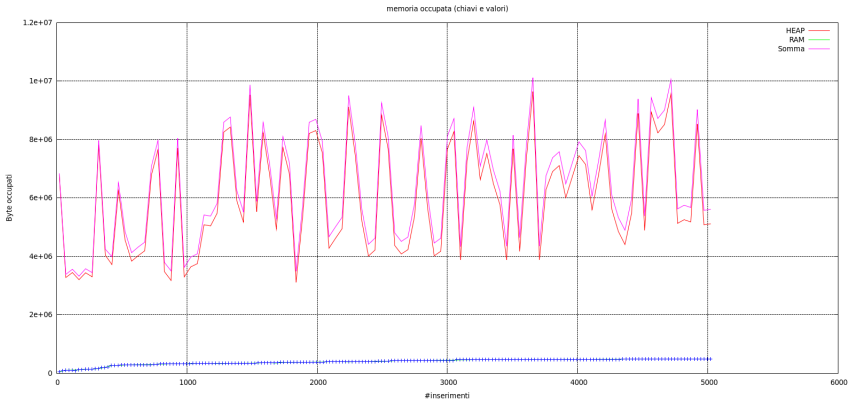


Non ci sono grosse considerazioni da fare nell'analisi del grafico: come facilmente intuibile, la memoria allocata aumenta pressochè linearmente all'aumentare del numero di chiavi (eccetto alcune oscillazioni

causate dall'allocazione di memoria nello *HEAP* della *JVM*). Il picco di memoria occupata raggiunta si assesta a circa 3.5 MB a 2500 elementi inseriti.

4.4.2 *Inserimento a FieldArchive pieno*

Figura 15: Memoria occupata da *OutputArchive* nel caso in cui *FieldArchive* sia pieno: i valori inseriti non sono mai stringhe vuote



Anche qui, come si poteva supporre, l'incremento di memoria è circa lineare (fatta eccezione per lo *HEAP* della *JVM* che tende ad oscillare nonostante le frequenti chiamate del Garbage Collector). Il picco di memoria occupata raggiunto in questa modalità risulta di circa 10MByte per quasi 5000 inserimenti.

4.4.3 *Confronto tra le due modalità di inserimento*

Di seguito l'analisi della memoria occupata:

Si osserva rapidamente che l'inserimento di chiave e valore, come ci si sarebbe potuti aspettare, occupa più memoria. Si nota inoltre, nel primo grafico, che la somma di memoria *RAM* e memoria *HEAP* allocata risulta, in media, circa doppia nel caso di inserimento di chiave e valore rispetto all'inserimento della sola chiave.

Passando ora alle più interessanti analisi dei tempi:

Il tempo impiegato è lineare e cresce all'aumentare del numero di inserimenti. Si osserva che il tempo di inserimento di chiave e valore impiega all'incirca tre volte il tempo impiegato per inserire la sola chiave. Questa differenza è dovuta al fatto che, nel caso più oneroso, automaticamente *OutputArchive* "chiede" a *FieldArchive*, per ogni elemento/attributo inserito, se ci sono Interconnected che contengono le chiavi che si stanno inserendo, oltre ad inserire un valore non vuoto. In caso di risposta affermativa, i valori che vengono inseriti sono tutti quelli rinvenuti in *FieldArchive* per quell'elemento/attributo. In caso di erroneo autoinserimento, sarà possibile modificare il valore per quella chiave.

4.5 GENERAZIONE DEL FILE DI OUTPUT

Una volta creato l'archivio di output, sapendo che la struttura delle chiavi è idonea a definire un documento valido, si procede alla traduzio-

Figura 16: Analisi della memoria occupata da *OutputArchive* nel caso *FieldArchive* sia pieno (viene salvato chiave + valore) e vuoto (solo chiave): somma *HEAP+RAM* sopra, solo *RAM* sotto

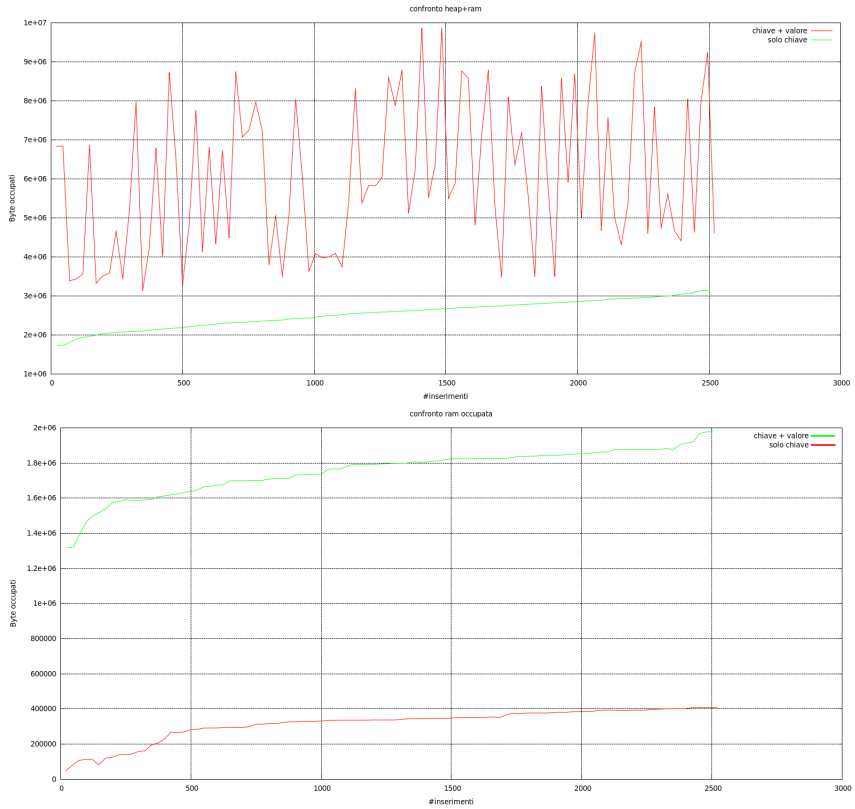
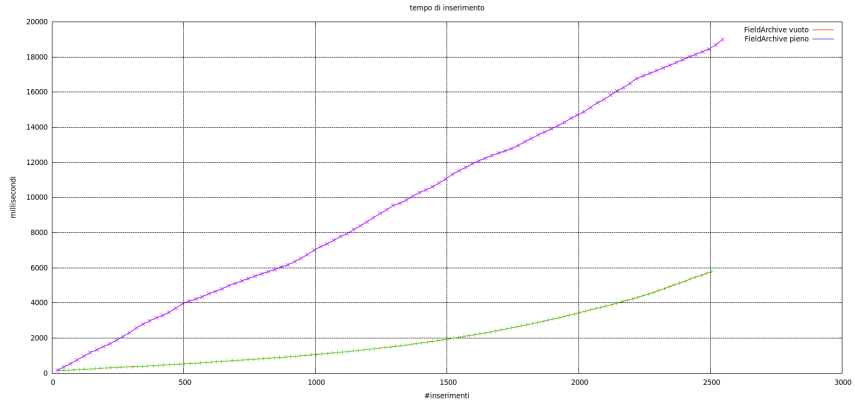


Figura 17: Confronto dei tempi di inserimento in *OutputArchive* nel caso *FieldArchive* sia pieno (chiavi + valori) e vuoto (solo chiavi)



ne dell'archivio in un file *XML*, validato dal *DTD* direttivo, contenente tutti i dati rinvenuti nell'atto o inseriti manualmente.

La generazione del file di output vero e proprio viene semplicemente eseguita grazie alla struttura delle chiavi utilizzate in OutputArchive (si veda 4.2.2).

Per la creazione del file *XML* di output ci si è appoggiati alla libreria JDOM (descritta in 2.2.4).

L'algoritmo si basa sull'idea di, per ogni chiave memorizzata in *OutputArchive*, seguire il percorso dalla radice fino all'attributo, per poi aggiungere il valore relativo a quella chiave.

L'algoritmo utilizzato è presentato nella pagina seguente:

Algorithm 4.4 Metodo per l'inserimento di un attributo nell'albero che rappresenta l'*XML*

```
private void follow(String path, String key){

//estrae dalla chiave il percorso dalla radice
String remainingPath=path.substring(root.getName().length()+1);
    Element tmp=root;
    Element toAppend;
    String tmpElem;
    String attName;
    String value="";
    while(remainingPath.length()>0){
      if(remainingPath.contains("/")){

        //estrae il nome del prossimo elemento
        tmpElem=remainingPath.substring(0, remainingPath.indexOf('/'));

        //se l'elemento non esiste nell'albero che si sta creando
        if(tmp.getChild(tmpElem)==null){
          toAppend=new Element(tmpElem);
          tmp.addContent(toAppend);//l'elemento viene aggiunto
        }
        tmp=tmp.getChild(tmpElem);// si passa all'elemento appena letto

        // si riduce il percorso passando oltre il primo '/'
        remainingPath=remainingPath.substring(remainingPath.indexOf('/')
            +1);
      }
      //Si è giunti all'attributo: sono finiti i nodi (sono finiti i
          caratteri '/')
      else if(remainingPath.contains(".")){

        //il nome dell'attributo è memorizzato dopo l'ultima occorrenza di
            ".",
        //l'elemento prima ("Elemento.Attributo").
        tmpElem=remainingPath.substring(0, remainingPath.indexOf('.'));

        //se l'elemento non esiste nell'albero che si sta creando
        if(tmp.getChild(tmpElem)==null){
          toAppend=new Element(tmpElem);
          tmp.addContent(toAppend);//l'elemento viene aggiunto
        }
        tmp=tmp.getChild(tmpElem);// si passa all'elemento appena letto
        remainingPath=remainingPath.substring(remainingPath.indexOf('.')
            +1);
        attName=remainingPath;// l'attributo è tutto ciò che segue il "."
        value=archive.get(key);
        tmp.setAttribute(attName, value);// viene appeso a quel nodo
        //il valore di quell'attributo.
      }
      else
        remainingPath="";
    }
}
```

Si noti che l'algoritmo sopra esposto, qualora vi fossero nodi inseriti più volte, manterrebbe inalterata la struttura della chiave (che prevede una struttura del tipo *NomeElementoRipetuto.NumeroRipetizione*): il

metodo non confonde il numero della ripetizione con un'attributo, dal momento che l'attributo è denotato da un '.' dopo l'ultimo '/'. L'albero creato mediante questo metodo, mantiene le numerazioni degli elementi ripetuti, e per questo motivo, una volta terminata l'iterazione del metodo di cui sopra per tutte le chiavi, si scorrerà l'albero una volta per eliminare i caratteri '.'.

A questo punto si è pronti a trascrivere il contenuto dell'albero, per ora memorizzato in *RAM*, su un file fisico, aggiungendo la validazione del documento *DTD*.

5 INTERAZIONE CON L'UTENTE

Tutto ciò che è stato trattato sin'ora riguarda il lavoro che la macchina fa automaticamente durante il flusso naturale dell'applicazione. Tuttavia si rende spesso necessario, a fronte di errori nel riconoscimento dei campi di interesse, fornire all'utente una possibilità per la configurazione manuale dei campi. A tal proposito è in fase di sviluppo FieldTable.

5.1 L'IDEA DI *fieldtable*: I REQUISITI

L'idea principale di *FieldTable* è quella di rendere semplice ed agevole l'interazione con l'utente finale in qualunque fase della stesura dell'atto. Le sue prerogative fondamentali constano di:

- Rendere possibile l'inserimento di nuovi Elementi/Attributi (presenti o non presenti nell'atto);
- Garantire l'inseribilità di solo Elementi/Attributi utili alla validificazione del documento di output;
- Gestire la modifica dei valori già presenti;
- Avere una interfaccia grafica semplice ed intuitiva.

Rispettando questi requisiti, *FieldTable* potrà effettuare delle modifiche *lecite*[1] agli elementi memorizzati in *OutputArchive*. Una volta effettuate tutte le operazioni, infine, OutputArchive sarà in grado di creare l'output opportuno ricostruendo la struttura dagli elementi in esso memorizzati.

5.2 IMPLEMENTAZIONE

L'idea di sviluppo è semplice. Si intende creare una tabella (accessibile tramite un apposito pulsante sull'interfaccia grafica di OpenOffice) che mostri all'utente quanto riconosciuto sin'ora nell'atto e quanto memorizzato in *OutputArchive*. La visione viene splittata in due parti. In una parte viene posta all'utente una visualizzazione ad albero, cosicchè abbia sempre sotto controllo che elemento sta aggiungengo e in che elemento sta aggiungendo un determinato attributo; nella seconda parte vengono presentati delle celle. Ciascuna cella rappresenta un Attributo dell'elemento selezionato nell'albero. Ciascun attributo è formato da due celle, una sola delle quali modificabile. In quella non modificabile è scritto il nome dell'attributo, in quella modificabile viene invece scritto il campo rinvenuto (se rinvenuto) o una stringa vuota.

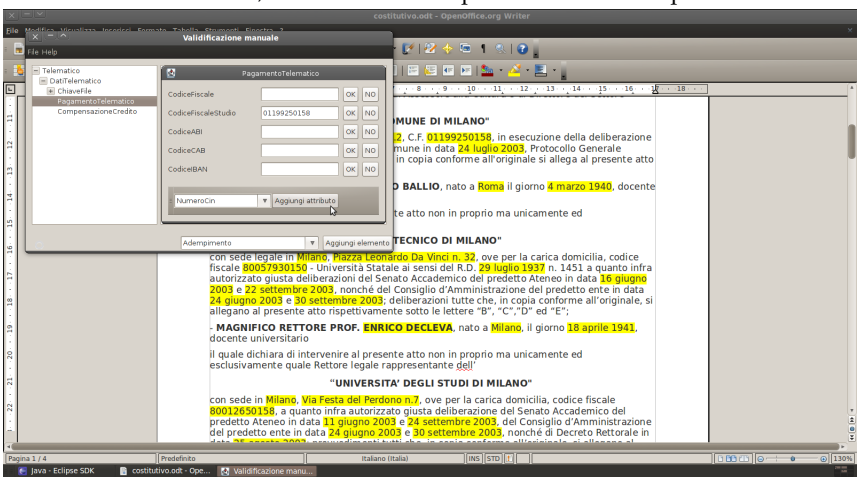
L'inserimento di nuovi Elementi è garantito dalla presenza di una *ComboBox* a completamento automatico di parola in fondo alla pagina. La *ComboBox* mostrerà gli inserimenti possibili auto-sincronizzandosi con *OutputArchive*. Per l'utente sarà sufficiente cercare l'elemento da inserire e premere il pulsante *"inserisci"*. In automatico si aggiungerà un

1 Tutte le modifiche effettuate ad *OutputArchive* dovranno sempre essere tali da mantenere, nell'archivio, una struttura di valori che possa ricostruire un file di output coerente con le normative descritte nel file *DTD* di validazione.

nuovo elemento nell'albero con tutti gli attributi minimi di quell'oggetto atti a validare il file di Output.

L'inserimento di nuovi Attributi avviene in maniera simile, ponendo una *ComboBox* nella visuale a destra; successivamente la *ComboBox* gestirà l'inserimento in maniera affine all'inserimento di nuovi Elementi.

Figura 18: La tabella si presenta come una finestra separata da OpenOffice al fine di garantirne la massima gestibilità: potrà essere aperta, chiusa, massimizzata, ridotta ad icona o passata in un altro spazio di lavoro.



Agenzia del Territorio. URL http://www.agenziaterritorio.it/?id=3303.

Castor. URL http://www.castor.org/.

JDOM. URL http://www.jdom.org/downloads/docs.html.

Matra. URL http://matra.sourceforge.net/.

OpenOffice Uno Documentation. URL OpenOfficeUnoDocumentation.

oracle.parser.v2. URL http://download.oracle.com/docs/cd/B10500_01/appdev.920/a96609/arj_xmlparserv2.htm.

Xerces. URL http://xerces.apache.org/xerces2-j/.

Agostino Avanzini, Luca iberati iberati, and Arturo Lovato. *Formulario degli atti notarili 2006*. Utet giuridica, 2006.

Elliotte Rusty Harold and W. Scott Means. *XML Guida di Riferimento*. Apogeo, 2001.

Brett McLaughlin. *Java e XML*. Apogeo, 2001.

Roger S. Pressman. *Principi di ingegneria del software*. McGraw-Hill Companies, 2000.

Ian Sommerville. *Ingegneria del software*. Pearson Addison Wesley, 2007.