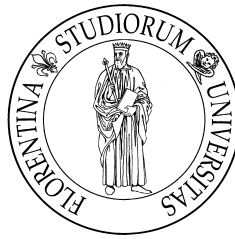


UNIVERSITÀ DEGLI STUDI DI FIRENZE
Facoltà di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea in Informatica



Tesi di Laurea

ANALISI E SVILUPPO DI UN CRAWLER PER LA
CREAZIONE DI UNA BASE DI CONOSCENZA
SEMANTICA DEL PERSONALE UNIVERSITARIO

STEFANO MARTINA

Relatore: *Elena Barcucci*

Corelatore: *Paolo Nesi*

Anno Accademico 2011-2012

SOMMARIO

Il presente testo è il risultato del lavoro di *stage* svolto presso il laboratorio *DISIT* del Dipartimento di Sistemi e Informatica dell'Università degli Studi di Firenze.

Il lavoro effettuato si inserisce nel progetto *OSIM* che ha l'obiettivo di creare una base di conoscenza semantica riguardante il personale universitario. In particolare è stata sviluppata la parte dell'applicazione che riguarda il *data mining* delle informazioni riguardanti le persone, i corsi associati a queste, e le competenze di ogni persona.

Il progetto fa uso delle tecnologie *RDF*, *RDFS* e *OWL* per l'organizzazione della base di conoscenza semantica. Per quanto riguarda l'applicazione, questa è sviluppata in *Java* facendo uso delle *Servlet* e delle *JSP* per fornire l'interfaccia *web* con l'utente.

Il testo è diviso in due parti: una parte consiste in una sostanziosa lettura sullo stato dell'arte delle tecnologie usate, dalle basi dell'*XML* e delle tecnologie correlate sino ad una visione eterogenea sulle tecnologie del web semantico; l'altra parte concerne la descrizione del lavoro effettuato e un'analisi dettagliata dell'applicazione nelle sue parti.

*We can only see a short distance ahead,
but we can see plenty there that needs to be done.*

— Alan Turing

RINGRAZIAMENTI

Desidero ringraziare calorosamente la Prof. Elena Barcucci e il Prof. Paolo Nesi per l'opportunità offertami da questo lavoro e per l'aiuto datomi allo sviluppo dell'applicazione e del presente testo. Ringrazio particolarmente Ing. Alice Fuzier Cayla per il grande aiuto sulla parte introduttiva a *GATE* e Ph.D. Andrea Bellandi e Ph.D. Ing. Gianni Pantaleo per il supporto dato allo sviluppo di alcune parti del programma e per il lavoro di revisione.

Ringrazio inoltre Ph.D. Ing. Pierfrancesco Bellini e Ing. Nadia Rauch per l'aiuto datomi all'inizio dello stage e Ph.D. Ing. Michela Paolucci per il materiale fornitomi. Vorrei ringraziare anche tutto il personale del laboratorio *DISIT* per l'ambiente dinamico, familiare e stimolante nel quale ho avuto il piacere di lavorare in questi mesi.

Desidero infine ringraziare con grande affetto i miei genitori per l'insostituibile supporto ricevuto in questi anni, e Federica per aver reso questo testo più comprensibile e per la sua pazienza nel sopportare la concorrenza del computer alla nostra relazione...

INDICE

I	STATO DELL'ARTE	1
1	XML e tecnologie correlate	3
1.1	XML	4
1.1.1	XML Information Set (Infoset)	6
1.2	Uri e namespaces	8
1.3	Linguaggi di schema	9
1.3.1	DTD	10
1.3.2	XML Schema	16
1.4	Famiglia XSL	26
1.4.1	XPath	28
1.4.2	XSLT	34
2	Web semantico	39
2.1	Visione d'insieme	39
2.2	Resource Description Framework (RDF)	42
2.2.1	Triple	42
2.2.2	Blank nodes (bnodes)	44
2.2.3	Serializzazioni	46
2.2.4	Reificazione	50
2.2.5	Tipi di dato	51
2.3	RDF Schema (RDFS)	52
2.3.1	Classi	52
2.3.2	Proprietà	53
2.3.3	Altri vocabolari	56
2.4	Web Ontology Language (OWL)	61
2.5	Simple Knowledge Organization System (SKOS)	61
2.5.1	Relazioni semantiche	62
2.5.2	Note	62
2.5.3	<i>Concept</i> schema	63
2.5.4	Mappare <i>concept</i> schema	63
2.5.5	Collezioni di <i>concept</i>	64
2.6	<i>Friend Of A Friend</i> (FOAF)	64
3	Framework GATE	67
3.1	Concetti generali	67
3.1.1	Collection of REusable Objects for Language Engineering (CREOLE)	67
3.1.2	Features e Annotazioni	68
3.2	Funzioni e Plugins	69

3.2.1	Gazetteer	69
3.2.2	Java Annotation Patterns Engine	69
3.2.3	A Nearly New Information Extraction System	70
II	ANALISI E LAVORO EFFETTUATO	73
4	Analisi del crawler	75
4.1	Il progetto OSIM	75
4.2	Introduzione	76
4.2.1	Estrazione delle keyword	76
4.2.2	Creazione del gazetteer	77
4.2.3	Estrazione delle competenze	77
4.2.4	Organizzazione delle competenze	77
4.3	Analisi dettagliata	78
4.3.1	Crawling delle <i>keyword</i>	81
4.3.2	Fase di selezione del gazetteer di <i>keyword</i>	87
4.3.3	Crawling delle competenze	88
4.3.4	Organizzazione dello SKOS	91
5	Analisi dell'applicazione	93
5.1	Ontologia	93
5.1.1	<i>Crawling</i> di competenze	93
5.1.2	Organizzazione dello SKOS	98
5.2	Applicazioni GATE	100
5.2.1	Applicazione langDetector.xgapp	101
5.2.2	Applicazione en.xgapp	102
5.2.3	Applicazione it.xgapp	103
5.2.4	Applicazione competenceExtraction.xgapp	103
5.3	Analisi delle <i>JSP</i> e delle chiamate alle <i>servlet</i>	106
5.4	Dettagli delle chiamate nelle due fasi di <i>crawling</i>	107
5.4.1	Dettagli della coda dei processi	110
5.5	Pagine <i>JSP</i>	111
5.5.1	Pagina userPage.jsp	112
5.5.2	Pagina index.jsp	112
5.6	Database	112
5.6.1	Tabella departments	113
5.6.2	Tabella keyword_*	113
5.6.3	Tabella user	114
5.6.4	Tabella authorization	114
5.6.5	Tabella recovery_queue	114
5.6.6	Tabella departments_status	115
5.6.7	Tabella carriera	115
5.7	Classi principali	115
5.7.1	Pacchetto SkosServlet	116
5.7.2	Pacchetto ProcessQueuing	120

5.7.3	Pacchetto AteneoCrawler	122
5.7.4	Pacchetto KeyExtraction	127
5.7.5	Pacchetto Translation	130
5.7.6	Pacchetto Messaging	131
6	Considerazioni finali e sviluppi futuri	133
	Bibliografia	135

Parte I

STATO DELL'ARTE

XML E TECNOLOGIE CORRELATE

L'*Extensible Markup Language* abbreviato con *XML* è, come dice il nome, un linguaggio di *markup*, di *marcatatura*. Esso deriva dallo *Standard Generalized Markup Language* (SGML) e nasce nel 1996 da un gruppo del W3C sotto la guida di Jon Bosak della *Sun Microsystems*.

Secondo le specifiche del W3C¹ gli obiettivi fondamentali che hanno guidato lo sviluppo dell'XML sono:

1. XML deve essere semplicemente usabile in internet;
2. XML deve supportare un'ampia varietà di applicazioni;
3. XML deve essere compatibile con SGML;
4. deve essere facile scrivere programmi in grado di processare documenti XML;
5. il numero delle funzionalità opzionali di XML deve essere mantenuto al minimo, idealmente a zero;
6. i documenti XML devono essere leggibili dall'uomo e ragionevolmente chiari;
7. il design dell'XML deve essere preparato velocemente;
8. il design dell'XML deve essere formale e conciso;
9. i documenti XML devono essere facili da creare;
10. la sinteticità nel markup XML è di minima importanza.

L'uso principale dell'XML è quello di creare documenti e scambiarli tra diverse applicazioni usando una tecnologia che permette di creare documenti strutturati con un livello scalabile di complessità e con un alto livello di flessibilità. Infatti XML ha una sintassi estremamente semplice e permette la rappresentazione strutturata pressoché di qualsiasi tipo di dato in quanto tale struttura viene definita a piacere.

La codifica dei caratteri dei documenti XML è l'*unicode*.

Oltre alle specifiche dei documenti XML esistono anche un certo numero di tecnologie ausiliarie, anche se spesso con *XML* ci si riferisce a tutte quante in realtà esse non fanno parte delle specifiche del nucleo propriamente detto *XML*. Di tali tecnologie noi analizzeremo solo quelle più importanti, ossia:

¹ <http://www.w3.org/TR/xml/> e nello specifico <http://www.w3.org/TR/xml/#sec-origin-goals>.

Namespaces l'uso dei namespace permette di definire dei *vocabolari* che permettono di identificare e differenziare i tag tra di loro;

DTD è un linguaggio di *schema* che definisce praticamente una grammatica con cui validare un documento XML;

XML Schema è il successore di *DTD*, ed è più potente di questo, è in formato XML e fornisce un ricco sistema di tipi di dato;

XSLT sta per *Extensible Stylesheet Language Transformation* e permette di creare dei documenti di trasformazione che, insieme ad un *parser*, permettono di elaborare un documento XML e trasformarlo in un altro documento (che può anche non essere XML);

XPath è un linguaggio che permette di indirizzare parti di un documento XML. Grazie ad XPath è possibile fare operazioni come *pescare* il contenuto di un certo tag o prendere tutti i figli di un certo tag;

XSL-FO Sta per *Extensible Stylesheet Language Formatting Objects* ed è un linguaggio di presentazione di documenti. Nell'idea originale un documento XML doveva essere trasformato tramite uno schema XSLT (con l'uso anche di XPath) in un documento XSL-FO che poteva essere renderizzato da un *processore FO* in un output leggibile.

Di seguito analizziamo nel dettaglio l'xml e tali tecnologie.

1.1 XML

L'*XML* è un *metalinguaggio* di *markup* che definisce una sintassi per esprimere una struttura ad albero dove ogni nodo si definisce *tag*. La caratteristica principale è che è possibile definire una struttura personalizzata a seconda dell'utilizzo che se ne vuole fare, cosa che ne permette un uso molto eterogeneo: dalle pagine web (con l'*XHTML* che è una formalizzazione in XML dell'*HTML*) allo scambio di dati o alla definizione di linguaggi.

La codifica dei caratteri è *unicode* valida in certi range, inoltre è possibile inserire ogni carattere con la sua corrispondente codifica decimale `&#...;` o esadecimale `&#x...;`. Ci sono dei caratteri speciali che non possono essere usati nel contenuto, ma solo nella struttura, per inserirli nel contenuto è necessario usare le entità XML illustrate nella tabella 1.

Un codice XML è costituito da un albero di tags con un singolo tag come radice. Ogni singolo tag è costituito come nel codice 1, dove **contenuto** può essere vuoto, un testo oppure una serie di altri *tags*, **nome** e **attributoX** non devono contenere i caratteri:

`!#$%&'()*+,-./:;<=>?@[\\]^_`{|}~` ,

spazi, o cominciare per un numero o per - o .. Ovviamente vi possono essere

Carattere	Codifica
<	<
>	>
&	&
'	'
"	"

Tabella 1: Entità XML

```
<nome attributo1="valore1" attributo2="valore2" attributoX="valoreX">
  contenuto
</nome>
```

Codice 1: Tag generico

un numero qualsiasi di attributi, anche nessuno. Nel caso in cui **contenuto** sia vuoto, è possibile usare una forma abbreviata come nel codice 2.

```
<nome attributo1="valore1" attributo2="valore2" attributoX="valoreX" />
```

Codice 2: Tag vuoto abbreviato

Oltre alla struttura dei *tags*, nello standard può essere presente anche una riga preliminare di dichiarazione, definita come nel codice 3, che deve essere

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

Codice 3: Dichiarazione

necessariamente la prima e dove vi sono le informazioni sulla versione usata dello standard e sulla codifica dei caratteri.

Se un documento XML rispetta le semplici regole appena definite si dice che esso è *ben formato*, oltre a questo, è possibile definire uno schema che indica la struttura e i nomi possibili dei tags come avviene, per esempio, nelle pagine web in *xhtml* dove vi è una riga preliminare (comunque sempre dopo l'eventuale *dichiarazione XML*) chiamata *DTD* o *Document Type Definition* tipo quella nel codice 4. Se vi è la definizione dello schema allora il documento, oltre ad essere *ben formato*, deve essere anche *valido*, ovvero deve rispettare lo schema definito. I concetti di *schema* e di XML *valido* verranno approfonditi meglio nella sezione 1.3 dove verrà introdotto anche il successore del *DTD*: *XML Schema*.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://
www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

Codice 4: Esempio di DTD

1.1.1 XML Information Set (Infoset)

Con il termine *Infoset* ci si riferisce ad un *data set* astratto che serve per avere un set di definizioni da usare nelle specifiche, dove si ha bisogno di accedere all'informazione in un documento XML ben formato. Più semplicemente serve per poterci riferire in seguito, in questo stesso testo, a certi concetti di un documento XML come i figli di un elemento o il nome di un attributo.

Il *set di informazioni* di un documento XML consiste in un certo numero di *information items*, uno per ogni componente del documento. Deve esistere almeno l'*information item* del documento.

Esistono *information items* per diversi concetti, ma qui ne saranno enunciati solo alcuni con le proprietà più importanti, quelli che saranno necessari allo sviluppo del testo. Per una trattazione esaustiva, con la lista completa degli *information items*, si rimanda a <http://www.w3.org/TR/xml-infoset>.

Information item del documento

C'è un solo *information item* di documento, tutti gli altri sono accessibili *scendendo* nelle proprietà di questo *information item*. Le proprietà sono:

figli contiene una lista ordinata di *information item* di elementi, quelli dei figli della radice del documento;

elemento documento, l'*information item* di tipo elemento della radice del documento;

URI di base l'uri di base del documento²;

schema di codifica caratteri ;

versione, la versione del XML del documento.

Information items degli elementi

Ci sono *information items* di tipo elemento per ogni elemento del documento. Le sue proprietà sono:

nome del namespace, se esiste, dell'elemento in questione. Se l'elemento non appartiene a nessun namespace allora la proprietà non ha valore³;

² Vedi sezione 1.2 per una descrizione degli URI e dei namespaces.

³ Vedi nota 2.

nome locale , il nome dell'elemento escluso l'eventuale prefisso del namespace⁴;

prefisso , il prefisso del namespace, se esistente, dell'elemento⁵;

figli contiene una lista ordinata di *information item* degli elementi figli di questo elemento;

attributi contiene una lista non ordinata di *information item* degli attributi di questo elemento, esclusi quelli di dichiarazione di namespace (**xmlns=...** e **xmlns:nome=...**)⁶;

attributi namespace contiene una lista non ordinata di *information item* degli attributi di dichiarazione di namespace di questo elemento⁷;

URI di base , l'uri di base dell'elemento⁸;

parente , l'*information item* dell'elemento o del documento che contiene l'*information item* di questo elemento nella sua lista di *figli*.

Information items degli attributi

Sono gli *information item* degli attributi che si trovano dentro gli elementi. Ogni attributo ne ha uno. Le proprietà sono:

nome del namespace , se esiste, dell'attributo in questione. Se l'attributo non appartiene a nessun namespace allora la proprietà non ha valore⁹;

nome locale il nome dell'attributo escluso l'eventuale prefisso del namespace¹⁰;

prefisso il prefisso del namespace, se esistente, dell'elemento¹¹;

valore normalizzato il valore dell'attributo normalizzato¹²;

specifico un flag che indica se l'attributo è definito nello *start tag* oppure è un elemento di default del *DTD*¹³;

tipo di attributo indica il tipo di attributo indicato nel *DTD*¹⁴;

⁴ Vedi nota 2.

⁵ Vedi nota 2.

⁶ Vedi nota 2.

⁷ Vedi nota 2.

⁸ Vedi nota 2.

⁹ Vedi nota 2.

¹⁰ Vedi nota 2.

¹¹ Vedi nota 2.

¹² Vedere <http://www.w3.org/TR/REC-xml/#AVNormalize> per la normalizzazione.

¹³ Vedi sezione 1.3.1 per una descrizione di *DTD*.

¹⁴ Vedi nota 13.

riferimento se l'attributo è di un tipo riferimento, allora questa proprietà contiene la lista degli *information items* degli elementi ai quali si riferisce;

elemento proprietario l'*information item* dell'elemento che contiene questo *information item* di tipo attributo nella sua lista di *attributi*.

1.2 URI E NAMESPACES

Un *URI* è un identificatore che permette di indicare in modo univoco una certa risorsa: come gli *URL* nel web permettono di identificare univocamente una pagina (o una sezione di una pagina), così gli *URI*, che sono una generalizzazione degli *URL*, permettono di identificare un concetto. Se un certo *URI* compare in posti diversi allora tutte le sue occorrenze rappresentano la stessa risorsa. La sintassi degli *URI*¹⁵ deve essere nella seguente forma¹⁶:

```
URI          = scheme ":" hier-part [ "?" query ] [ "#" fragment ]

hier-part    = "://" authority path-abempty
              / path-absolute
              / path-rootless
              / path-empty

authority    = [ userinfo "@" ] host [ ":" port ]
```

Dove **scheme** e **path** sono obbligatori, ma il **path** può essere anche vuoto. Quando **authority** è presente allora il **path** deve essere vuoto o cominciare per il carattere /; quando **authority** non è presente allora **path** non può cominciare con i caratteri //. Brevemente accenniamo che nella rappresentazione descritta precedentemente tutti i **path** sono composti da segmenti delimitati dal carattere /, inoltre **path-abempty** può essere vuoto o cominciare con /, **path-absolute** comincia con / ma non con //, **path-rootless** comincia con un segmento senza / iniziale, **path-empty** è una sequenza vuota di caratteri. Per una formalizzazione completa degli *URI* si rimanda a [8].

Gli *URI*, oltre che nella suddetta forma, possono presentarsi anche in una forma *relativa*, a patto che esista un *URI di base* con il quale poter *risolvere* l'*URI* relativo in un *URI target* nella forma vista precedentemente. Questi *URI* relativi sono nella forma:

```
relative-ref = relative-part [ "?" query ] [ "#" fragment ]

relative-part = "://" authority path-abempty
               / path-absolute
               / path-noscheme
               / path-empty
```

¹⁵ Secondo l'*internet standard STD 66*, vedere [8].

¹⁶ Espressa nella forma *ABNF* descritta nello standard *STD 68*, vedere [9].

dove **path-noscheme** è un segmento che non contiene nessun **:** seguito da altri segmenti *normali*, sempre separati da **/**.

Esiste un modo pratico per indicare gli *URI* in una forma maggiormente leggibile e facilmente modificabile: la sintassi in questione prende il nome di *qnames* (*Qualified Names*) e in questa sintassi un *URI* viene rappresentato nella forma

prefix:identificatore

dove **prefix** è usato per creare un mapping con un certo *namespace*, quindi da qualche parte è necessario indicare questa corrispondenza *prefix - namespace*.

I *namespaces* non fanno parte del *core* di XML, ma sono entrati così tanto nell'uso comune da essere di fatto una funzionalità di base. Essi servono per creare una sorta di vocabolario per poter distinguere diverse entità con nome uguale, ad esempio se in un documento XML voglio creare due tag `<num>...</num>` con significato semantico diverso (il primo è l'identificativo di un cliente e il secondo il numero di dischi venduti di un cantante) una soluzione può essere quella di creare due *namespace* diversi ed assegnare ad ogni tag *num* uno dei due. Per fare ciò in XML si usa lo speciale pseudo attributo **xmlns** nel modo seguente:

```
<ancestor xmlns:cliente="http://www.example.org/ns/cliente"
xmlns:disco="http://www.example.org/ns/disco">...
```

adesso per ogni discendente di **ancestor** esistono i due *namespaces* indicati ed è possibile riferirsi ad essi usando la sintassi *qname*

```
<cliente:num>...</cliente:num>
...
<disco:num>...</disco:num>
```

Oltre che per gli elementi, è possibile applicare i *namespaces* anche agli argomenti. È possibile inoltre indicare un *namespace* di default usando la forma **xmlns=http://www.example.org/ns/default** (senza specificare il prefisso), in questo modo tutti gli elementi (ed attributi) senza un prefisso esplicito appartengono a tale *namespace*.

Vi è stato un lungo dibattito sulla questione se trattare i *namespace* come *URI* e quindi *risolverli* nel caso in cui siano nella forma di *URI relativi*, oppure se trattarli puramente come delle stringhe e non risolverli. Il dibattito si è concluso deprecando l'uso degli *URI relativi* (oltre che di quelli *vuoti*) nei *namespaces*¹⁷.

1.3 LINGUAGGI DI SCHEMA

Esistono due linguaggi di schema associati ad XML: il *DTD* e l'*XML Schema*. Entrambi svolgono la stessa funzione di base, ossia creare uno schema che fornisce

¹⁷ Vedere a tal proposito <http://www.w3.org/TR/xml-names/#iri-use>.

dei limiti alla struttura logica che un documento XML esprime. La differenza tra i due consiste in questo: il primo è nato con XML e perciò è supportato in tutte le versioni ed usa una sintassi basata sulle espressioni regolari; *XML Schema*, invece, usa una sintassi XML, inoltre è molto più potente di *DTD* ed usa un ricco set di dati¹⁸.

Se un documento XML prevede anche uno schema, allora oltre ad essere *ben formato* deve essere anche *valido*. Un documento è *valido* se rispetta le regole e le limitazioni imposte da uno schema al quale appartiene.

1.3.1 DTD

Il *Document Type Definition*, *DTD*, consiste in una grammatica per una certa classe di documenti. In un singolo documento è possibile definire una *document type declaration* che può puntare ad un *DTD* esterno, o contenere direttamente le dichiarazioni che costituiscono un *DTD*, nel codice 4 abbiamo già visto un esempio di dichiarazione *DTD* che punta ad un *DTD* esterno al documento.

È possibile vedere la definizione formale completa del *DTD* con la grammatica in notazione *Extended Backus Naur Form* all'indirizzo: <http://www.w3.org/TR/xml/#sec-prolog-dtd>, comunque qui accenniamo all'uso generale della dichiarazione del *DTD* in un documento *XML* e delle dichiarazioni che compongono il *DTD* stesso. Queste ultime dichiarazioni possono essere di quattro tipi:

- dichiarazione tipo di elemento;
- dichiarazione lista attributi;
- dichiarazione entità;
- dichiarazione notazione.

Definiamo preliminarmente delle entità che useremo nella formalizzazione di tali dichiarazioni¹⁹ nel codice 5 dove **Name** è un possibile tipo di nome costituito da uno dei possibili caratteri di start **NameStartChar** seguito da un numero a piacere (anche zero) di possibili caratteri **NameChar**. **Names** è una lista di uno o più nomi separati da spazi. **Nmtokens** è uguale a **Name** senza la limitazione del carattere iniziale (deve comunque essere lungo almeno un carattere). **Nmtokens** similmente a **Names** è una lista di uno o più **Nmtoken** separati da spazi. **S** è un tipo possibile di spazio separatore, comprese tabulazioni e ritorni a capo. **ExternalID** rappresenta un *URI* e può essere di due tipi: usando l'identificatore **SYSTEM** si indica una stringa tra virgolette (che non contiene virgolette) o tra apici (che non contiene apici) che va convertita in un *URI* (la posizione del

¹⁸ Che sarà utile anche in *RDF*, vedere sezione 2.2.5.

¹⁹ In notazione Extended Backus Naur Form.

```

NameStartChar ::= ":" | [A-Z] | "_" | [a-z] | [#xC0-#xD6] |
                  [#xD8-#xF6] | [#xF8-#x2FF] | [#x370-#x37D] |
                  [#x37F-#x1FFF] | [#x200C-#x200D] |
                  [#x2070-#x218F] | [#x2C00-#x2FEF] |
                  [#x3001-#xD7FF] | [#xF900-#xFDCF] |
                  [#xFDF0-#xFFFD] | [#x10000-#xEFFFF]

NameChar       ::= NameStartChar | "-" | "." | [0-9] | #xB7
                  | [#x0300-#x036F] | [#x203F-#x2040]

Name           ::= NameStartChar (NameChar)*
Names          ::= Name (#x20 Name)*
Nmtoken        ::= (NameChar)+
Nmtokens       ::= Nmtoken (#x20 Nmtoken)*

S              ::= (#x20 | #x9 | #xD | #xA)+

ExternalID     ::= 'SYSTEM' S SystemLiteral
                  | 'PUBLIC' S PubidLiteral S SystemLiteral
SystemLiteral  ::= ('"' [^"]* '"') | ('"' [^']* '"')
PubidLiteral   ::= "'" PubidChar* "'" | '"' (PubidChar - '"')* '"'
PubidChar      ::= #x20 | #xD | #xA | [a-zA-Z0-9] | [-'()+,./:=?;!*#@$_%]
```

Codice 5: Costrutti sintattici comuni

DTD esterno), usando invece **PUBLIC** si indicano due stringhe tra virgolette o apici, in quest'ultimo caso il processore dell'*XML* oltre a cercare lo schema nell'*URI* rappresentato dalla seconda stringa come nel caso di **SYSTEM**, ha ulteriori informazioni dalla prima stringa.

La dichiarazione del tipo di documento, all'interno del documento *XML* nel caso di un *DTD* esterno ad esso, assume la forma²⁰ del codice 6 dove **Name**

```
doctypedecl ::= '<!DOCTYPE' S Name (S ExternalID)? S? ('[' intSubset ']' S?)? '>'
intSubset   ::= (markupdecl | DeclSep)*
markupdecl  ::= elementdecl | AttlistDecl | EntityDecl | NotationDecl |
               PI | Comment
DeclSep     ::= PReference | S
```

Codice 6: Dichiarazione del *doctype*

deve essere nella forma vista nel codice 5 e deve essere uguale al nodo radice del documento *XML*, **ExternalID** è un *URI* tra virgolette nella forma vista nel codice 5 e **intSubset** indica il subset del *DTD* in questione da prendere in considerazione. Un esempio di dichiarazione può essere quella dell'*XHTML 1.1*:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
```

che indica che il nodo radice è **html** e il tipo di dichiarazione è **PUBLIC** quindi fornisce due stringhe, una con l'URL del file *DTD* con lo schema e una con ulteriori informazioni per chi processa questo file *XHTML*.

Vengono analizzate e formalizzate nel dettaglio le quattro possibili dichiarazioni *DTD*:

Dichiarazioni di tipo elemento

Servono per poter indicare la struttura di un certo elemento, è possibile indicare per ogni singolo elemento quali figli può contenere e in che numero e ordine usando una notazione in stile *espressioni regolari*, oppure se un certo elemento può contenere del testo opzionalmente inframezzato da figli. Ogni dichiarazione di questo tipo assume la forma²¹:

```
elementdecl ::= '<!ELEMENT' S Name S contentspec S? '>'
contentspec ::= 'EMPTY' | 'ANY' | Mixed | children
```

dove:

²⁰ Vedi nota 19.

²¹ Vedi nota 19.

S è una possibile rappresentazione del carattere spazio come visto sopra nel codice 5;

Name è una sequenza di caratteri come visto sopra nel codice 5;

'EMPTY' indica che il tag deve essere vuoto;

'ANY' che può contenere qualsiasi contenuto;

children è una lista di tags che possono essere figli del tag in questione. Tale lista è espressa secondo la notazione delle *espressioni regolari*, ossia può essere una scelta tra un set di valori separati da |, una sequenza di valori separati da , o una combinazione di queste racchiuse tra (...) annotate con uno tra i caratteri + per possibili occorrenze di una o più volte, * per zero o più volte, ? per zero o una volta, se tale carattere non è specificato implica che deve esserci esattamente una occorrenza.

Ad esempio la dichiarazione:

```
<!ELEMENT div1 (head, (p | list | note)*, div2*)>
```

indica che dentro ogni **div1** ci deve essere un tag **head** seguito da una serie lunga a piacere (anche vuota) di tags che possono essere scelti tra **p**, **list**, **note** in qualsiasi ordine; seguita da una lista sempre lunga a piacere (anche vuota) di tags di tipo **div2**.

Mixed indica che il tag può contenere solo del testo indicato con **#PCDATA**, oppure del testo inframezzato da possibili tags. Nel caso di contenuti misti non è possibile specificare l'ordine della sequenza dei figli, come nel caso precedente, ma solo la possibile scelta, inoltre si può solo usare l'operatore * o nessun operatore.

Ad esempio la dichiarazione:

```
<!ELEMENT p (#PCDATA|a|ul|b|i|em)*>
```

Indica che i tags di tipo **p** possono contenere del testo inframezzato da uno dei possibili tags indicati, in qualsiasi ordine e numero, mentre la dichiarazione:

```
<!ELEMENT b (#PCDATA)>
```

indica che i tag di tipo **b** possono contenere solo testo e non possono essere vuoti.

Dichiarazioni di tipo lista di attributi

Queste dichiarazioni servono per poter definire il set di attributi pertinenti ad un certo tipo di tag, porre dei limiti a tali attributi e fornire dei valori di default per tali attributi. Queste dichiarazioni sono nella forma²² del codice 7 dove sono

```

AttlistDecl ::= '<!ATTLIST' S Name AttDef* S? '>'
AttDef      ::= S Name S AttType S DefaultDecl

AttType     ::= StringType | TokenizedType | EnumeratedType
StringType  ::= 'CDATA'
TokenizedType ::= 'ID'
              | 'IDREF'
              | 'IDREFS'
              | 'ENTITY'
              | 'ENTITIES'
              | 'NMTOKEN'
              | 'NMTOKENS'
EnumeratedType ::= NotationType | Enumeration
NotationType  ::= 'NOTATION' S '(' S? Name (S? '|' S? Name)* S? ')'
Enumeration   ::= '(' S? Nmtoken (S? '|' S? Nmtoken)* S? ')'

DefaultDecl  ::= '#REQUIRED' | '#IMPLIED'
              | (('FIXED' S)? AttValue)

```

Codice 7: Dichiarazioni di tipo lista di attributi

costituite da un nome del tag in questione su cui si vogliono definire i limiti per gli attributi, e una serie di triple chiamate nella definizione sopra **AttType** divise da spazi che indicano tali limiti. Ogni tripla è costituita da un nome dell'attributo per il quale stiamo definendo i limiti, un tipo di attributo e un valore di default. Il tipo di attributo, chiamato sopra **AttType**, può essere di tipo **EnumeratedType** che consiste in una lista di scelte possibili per il contenuto dell'attributo racchiuse tra parentesi (...) e separate da |. Oppure **AttType** può essere anche di tipo stringa indicato con **CDATA** che implica che può essere scelta qualsiasi stringa come contenuto di questo attributo oppure uno dei seguenti tipo con i relativi limiti:

ID deve apparire un solo attributo di questo tipo per tag, il valore di tale attributo deve essere una produzione di **Name** visto sopra nel codice 5; questo attributo identifica univocamente il tag, deve avere indicato come valore di default **#REQUIRED** o **#IMPLIED**;

IDREF il valore dell'attributo deve essere una produzione di **Name** visto nel codice 5;

²² Vedi nota 19.

IDREFS il valore dell'attributo può essere costituito da una lista di valori di tipo **Name**, cioè deve essere produzione di **Names** del codice 5;

ENTITY deve essere produzione di **Name** del codice 5, inoltre deve essere il nome di una entità dichiarata in una dichiarazione di tipo entità;

ENTITIES deve essere produzione di **Names** del codice 5, inoltre deve essere una lista di nomi di entità dichiarate;

NMTOKEN il contenuto di questi attributi deve essere produzione di **Nmtoken** del codice 5;

NMTOKENS deve essere una lista di **Nmtoken** separati da spazi, ossia deve essere una produzione di **Nmtokens** del codice 5.

Il valore di default, chiamato **DefaultDecl** nel codice 7, può essere **#REQUIRED** che indica che l'attributo deve essere inserito, **#IMPLIED** che non viene fornito nessun valore di default, un valore che rappresenta il valore di default o un valore preceduto da **#FIXED** che indica che il valore di default fornito è l'unica scelta possibile per tale attributo.

Dichiarazione di entità

Le *entità* XML sono come quelle di default viste in tabella 1 di pagina 5. Con questo tipo di dichiarazione DTD è possibile definire nuove *entità*, in pratica trattasi di tipi di *placeholder* che vengono sostituiti all'interno del documento con il testo corrispondente indicato da queste dichiarazioni. Le dichiarazioni sono nella forma²³ del codice 8 dove **Name** è il nome dell'entità e il valore dell'en-

```
EntityDecl ::= GEDecl | PEDecl
GEDecl    ::= '<!ENTITY' S Name S EntityDef S? '>'
PEDecl    ::= '<!ENTITY' S '%' S Name S PEDef S? '>'
EntityDef  ::= EntityValue | (ExternalID NDataDecl?)
PEDef     ::= EntityValue | ExternalID
NDataDecl ::= S 'NDATA' S Name
```

Codice 8: Dichiarazioni di entità

tà può essere direttamente un valore racchiuso tra virgolette (**EntityValue**) oppure un identificatore esterno (**ExternalID** vedi codice 5) con una dichiarazione opzionale del tipo di dato dell'identificatore esterno(**NDataDecl**), in pratica un uri esterno che può essere risolto.

²³ Vedi nota 19.

Dichiarazione di notazione

Le dichiarazioni di notazione sono nella forma:

```
NotationDecl ::= '<!NOTATION' S Name S (ExternalID | PublicID) S? '>'
PublicID      ::= 'PUBLIC' S PubidLiteral
```

e servono per definire un formato ad una risorsa che non è né testo né XML. Le notazioni indicate possono essere usate in dichiarazioni di entità e liste di attributi.

1.3.2 XML Schema

XML Schema svolge la stessa funzione di *DTD*, è un linguaggio che fornisce la possibilità di creare uno schema per definire la struttura di un documento XML. È possibile definire inoltre le relazioni tra gli *elementi* e i loro contenuti e gli *attributi* e i loro valori, fornisce inoltre un ricco set di *tipi di dato*. *XML Schema*, inoltre, usa una sintassi basata su XML e fornisce un *namespace* <http://www.w3.org/2001/XMLSchema> per la definizione degli elementi che compongono lo schema. È convenzione chiamare tale namespace con il prefisso **xsd** così come l'estensione del file dello schema, oppure anche con il più conciso **xs**. Quindi nel primo tag è bene definire:

```
xmlns:xs="http://www.w3.org/2001/XMLSchema"
```

Nel file XML a cui deve essere assegnato lo *schema* bisogna usare il namespace <http://www.w3.org/2001/XMLSchema-instance> a cui convenzionalmente si associa il prefisso **xsi**. Il compito di assegnare lo schema ad un certo documento è del *processore* del documento XML, come per il *DTD*, proprio come quest'ultimo viene fornito un set di convenzioni per fornire le informazioni al *processore* per effettuare l'associazione *documento-schema*. Per questo esistono gli attributi **xsi:schemaLocation** e **noNamespaceSchemaLocation**, nel primo è possibile indicare come valore di tale attributo una serie di coppie *namespace - URI* dello schema, tali valori sono tutti separati da spazi e ogni coppia rappresenta un namespace e un percorso dove trovare uno schema da usare per tale namespace. Se in un documento XML viene definito il primo elemento come nel codice 9 per gli elementi associati al namespace di default(<http://www.w3.org/1999/XSL/Transform>) viene applicato lo schema definito dal documento che si trova all'indirizzo <http://www.w3.org/1999/XSL/Transform.xsd>; per quelli associati al namespace **html** (<http://www.w3.org/1999/xhtml>) si applica lo schema in <http://www.w3.org/1999/xhtml.xsd>. Esiste inoltre l'attributo **xsi:noNamespaceSchemaLocation** che, invece di avere una lista di coppie di valori, ha una lista di valori singoli, senza che venga specificato il *namespace* degli elementi a cui viene applicato tale schema.

```
<stylesheet xmlns="http://www.w3.org/1999/XSL/Transform"
             xmlns:html="http://www.w3.org/1999/xhtml"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://www.w3.org/1999/XSL/Transform
                                http://www.w3.org/1999/XSL/Transform.xsd
                                http://www.w3.org/1999/xhtml
                                http://www.w3.org/1999/xhtml.xsd">
```

Codice 9: Dichiarazione XML SCHEMA dentro un documento

Uno schema è costituito da componenti che possono essere di diverso tipo:

- componenti definizione di tipo;
- componenti dichiarazione;
- componenti definizione di gruppi attributo;
- componenti gruppo modello e definizione di gruppi modello;
- componenti definizione *identity-constraints*;
- componenti annotazione.

Di seguito vengono analizzati nel dettaglio tali componenti e, inoltre, viene dedicata una sezione anche alla descrizione del ricco *set di dati* predefiniti che fornisce XML SCHEMA.

I componenti sono definiti in termini delle sue proprietà, e per ogni proprietà definendo la sua immagine. Si può immaginare uno schema come un grafo orientato con radice dove questa radice è lo schema stesso, ogni nodo è un componente o un letterale e ogni arco è una proprietà.

Tipo di dato

In XML SCHEMA un tipo di dato è costituito da una tripla consistente di:

spazio valori , un set di valori distinti che sono l'insieme dei valori per un certo tipo di dato. Ogni valore corrisponde ad uno o più letterali dello *spazio lessicale*.

Questo spazio può essere definito:

- in modo assiomatico;
- enumerandolo;
- restringendo lo *spazio valori* di un tipo di dato già esistente, partendo da dei tipi di dato primitivi o da altri già *derivati*;
- come combinazione di valori da uno o più *spazi valori* già esistenti;

spazio lessicale , un set di *letterali validi* per un certo tipo di dato. Spesso c'è una corrispondenza uno ad uno tra gli elementi dello *spazio lessicale* e gli elementi dello *spazio valori*, ma non sempre è così, a volte un singolo valore può essere rappresentato con diversi modi lessicali: ad esempio **100** e **1.0E2** sono due letterali diversi dello *spazio lessicale* del tipo di dato *float* che denotano entrambi lo stesso valore dello *spazio valori*;

facets , letteralmente sfaccettature, sono singoli aspetti che definiscono uno spazio, ogni *facet* caratterizza uno *spazio valori* su un'asse o dimensione indipendente. Esistono *facets fondamentali* e *facets non fondamentali*, i primi sono delle proprietà astratte che servono per caratterizzare semanticamente i valori di uno *spazio valori* e queste sono tutte le seguenti autoesplicative:

- equal;
- ordered;
- bounded;
- cardinality;
- numeric.

I *facets non fondamentali* sono proprietà opzionali che servono per restringere lo *spazio valori*, questi sono:

- length;
- minLength;
- maxLength;
- pattern;
- enumeration;
- whiteSpace;
- maxInclusive;
- maxExclusive;
- minExclusive;
- minInclusive;
- totalDigits;
- fractionDigits.

Quest'ultimi *facets non fondamentali* non si applicano indiscriminatamente a tutti i tipi di dato, ma ogni tipo primitivo e i suoi derivati ha una possibile scelta di *facets* che possono essere usati.

Per una trattazione esaustiva dei *facets primitivi* e *non fondamentali* si rimanda a <http://www.w3.org/TR/xmlschema-2/#rf-fund-facets> e a <http://www.w3.org/TR/xmlschema-2/#rf-facets>.

XML SCHEMA fornisce un *namespace* di default per tutti i tipi di dato built in e tutti i facets visti:

```
http://www.w3.org/2001/XMLSchema
```

in modo che possano essere indirizzati aggiungendo il nome del tipo di dato o del facet come *frammento* dell'URI nel modo seguente:

```
http://www.w3.org/2001/XMLSchema#int
http://www.w3.org/2001/XMLSchema#maxInclusive
```

Inoltre per i tipi di dato viene fornito anche il namespace:

```
http://www.w3.org/2001/XMLSchema-datatypes
```

Componenti definizione di tipo

I *componenti definizione di tipo* servono per definire i tipi di dato come quelli visti nella sezione 1.3.2. Esistono due tipi di definizioni di tipo: tipo semplice e tipo complesso.

I componenti per la definizione dei tipi semplici sono una serie di restrizioni sulle stringhe e informazioni sui valori che rappresentano applicabili ai valori di un attributo XML o ai contenuti testuali di un elemento. Ogni definizione di tipo semplice, sia di quelle predefinite del *dataset* dell'XML SCHEMA²⁴, sia creata dall'utente, sono delle restrizioni su delle particolari definizioni di tipo di base tramite i facets. Ad esempio nel codice 10 viene definito un nuovo tipo

```
<xs:simpleType name="civico">
  <xs:restriction base="xs:string">
    <xs:pattern value="[0-9]+(N|R)?([a-z]+)?" />
  </xs:restriction>
</xs:simpleType>
```

Codice 10: Esempio di componente definizione di tipo semplice

civico restringendo lo *spazio lessicale* (e quindi lo *spazio valori*) di **string** con un pattern definito con una *espressione regolare*.

I *tipi complessi* permettono elementi e attributi nel suo contenuto. Sono definiti usando l'elemento

```
xsi:complexType
```

e di solito contengono un set di definizioni di elementi, nomi di elementi e dichiarazioni di attributi, ad esempio come nel codice 11. Si noti che in questo

²⁴ Vedere sezione 1.3.2.

```
<xs:complexType name="Indirizzo" >
  <xs:sequence>
    <xs:element name="via" type="xs:string"/>
    <xs:element name="numero" type="xs:civic"/>
    <xs:element name="citta" type="xs:string"/>
    <xs:element name="cap" type="xs:decimal"/>
  </xs:sequence>
  <xs:attribute name="stato" type="xs:NMTOKEN" fixed="IT"/>
</xs:complexType>
```

Codice 11: Esempio di componente definizione di tipo complesso

codice è stato usato come tipo del numero civico quello definito prima nel codice 10.

Per una trattazione completa dei componenti di definizione di tipo semplice si rimanda a

http://www.w3.org/TR/xmlschema-1/#Simple_Type_Definitions e a

<http://www.w3.org/TR/xmlschema-2/#rf-defn>,

per i tipi complessi si rimanda invece a

http://www.w3.org/TR/xmlschema-1/#Complex_Type_Definitions.

Componenti dichiarazione

Le dichiarazioni possono essere di *attributi* o *elementi*, le *dichiarazioni di attributi* servono per fornire la validazione degli attributi usando una definizione di tipo semplice, inoltre permettono anche di specificare dei valori di default o fissi per il valore dell'attributo. Ad esempio

```
<xs:attribute name="eta" type="xs:positiveInteger" use="required"/>
```

indica che l'attributo **eta** è obbligatorio e deve essere un intero positivo.

Le dichiarazioni di *elementi* servono per fornire la validazione degli elementi usando una definizione di tipo, definire valori di default o fissi per l'*information item* di un elemento²⁵), stabilire criteri di univocità e relazioni tra i valori degli elementi e attributi correlati. Ad esempio il codice 12 definisce un elemento chiamato **PurchaseOrder** di tipo **PurchaseOrderType** e un altro elemento di nome **gift** che deve avere come figli un elemento **birthday** di tipo **date** e un elemento di tipo **PurchaseOrder** definito sopra.

Per una trattazione esaustiva sui componenti dichiarazione si rimanda a

http://www.w3.org/TR/xmlschema-1/#Attribute_Declarations e a

http://www.w3.org/TR/xmlschema-1/#cElement_Declarations.

²⁵ Vedere sezione 1.1.1.

```

<xs:element name="PurchaseOrder" type="PurchaseOrderType"/>

<xs:element name="gift">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="birthday" type="xs:date"/>
      <xs:element ref="PurchaseOrder"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

```

Codice 12: Esempio di dichiarazione elemento

Componenti definizione di gruppi attributo

È possibile definire dei gruppi di dichiarazioni di *attributi* per poi poterli riutilizzare all'interno di dichiarazioni di tipo complesse, ad esempio il codice 13 raggruppa delle definizioni di attributo dentro il gruppo **myAttrGroup** e poi lo

```

<xs:attributeGroup name="myAttrGroup">
  <xs:attribute . . ./>
  . . .
</xs:attributeGroup>

<xs:complexType name="myelement">
  . . .
  <xs:attributeGroup ref="myAttrGroup"/>
</xs:complexType>

```

Codice 13: Esempio di definizione di gruppo di attributi

usa per definire il tipo *myelement*.

Si rimanda a:

http://www.w3.org/TR/xmlschema-1/#cAttribute_Group_Definitions

per una trattazione completa.

Componenti gruppo modello e definizione di gruppi modello

I *gruppi modello* servono per poter definire nel dettaglio la sequenza degli elementi figlio. Possono essere di tre tipi e possono essere anche concatenati, per questo ai contenuti dei *gruppi modello* ci si riferisce con il termine *particelle* che possono essere dichiarazioni di elemento o anche altri *gruppi modello*. Questi tre tipi sono:

sequence corrisponde alle *particelle* indicate nell'ordine indicato;

choice corrisponde ad una delle particelle indicate;

all corrisponde a tutte le *particelle* indicate in qualsiasi ordine.

Abbiamo già visto un *gruppo modello sequence* nel codice 12.

Inoltre è possibile definire dei *gruppi modello* con **xs:group** in modo da poterli inserire per riferimento in altri posti. Nel codice 14 è possibile vedere

```
<xs:group name="myModelGroup">
  <xs:sequence>
    <xs:element ref="someThing"/>
    . . .
  </xs:sequence>
</xs:group>

<xs:complexType name="trivial">
  <xs:group ref="myModelGroup"/>
  <xs:attribute .../>
</xs:complexType>
```

Codice 14: Esempio di definizione di gruppo modello

un esempio di creazione di un gruppo modello con una *sequence* chiamato **myModelGroup**, e poi usato con riferimento dentro la dichiarazione di un tipo di dato complesso chiamato **trivial**.

Per chiarezza, le *particelle* che abbiamo citato prima (il contenuto dei gruppi modello) oltre a poter essere delle dichiarazioni di elemento o dei gruppi modello a sua volta, esse possono essere anche delle *wildcard*. Le *wildcard* possono *matchare information items* di elementi e attributi dipendentemente dai loro nomi di namespace e indipendentemente dai loro nomi locali²⁶.

Per una trattazione esaustiva dei *gruppi modello* si rimanda a http://www.w3.org/TR/xmlschema-1/#Model_Groups, mentre per la dichiarazione di *gruppi modello* http://www.w3.org/TR/xmlschema-1/#cModel_Group_Definitions.

Componenti definizione identity-constraints

Questi componenti forniscono la possibilità di definire un sistema di riferimenti e univocità degli elementi o attributi di dove sono definiti. Esistono tre tipi di questi componenti e tutti richiedono che sia specificato un **selector** e almeno un **field**, oltre che un **name** con cui identificare il componente:

²⁶ Vedere sezione 1.1.1.

unique garantisce l'univocità di tutte le tuple risultanti dalla valutazione dei campi **field** rispetto al campo **selector**. Nei campi **field** vi è indicata una espressione *XPath*²⁷ con cui effettuare la valutazione in questione;

key oltre a garantire l'univocità come per *unique*, assicura che le tuple risultanti dalla valutazione effettivamente siano presenti;

keyref assicura una corrispondenza tra le tuple risultanti dalla valutazione e quelle identificate dal componente indicato nel campo **refer**.

Per meglio comprendere prendiamo ad esempio il codice XML 15 e il relativo codice dello schema 16, oltre alla definizione dei componenti di definizione

```
<dipartimento xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="schema.xsd">
  <docenti>
    <docente codiceCorso="386">
      <nome>Mario Rossi</nome>
      <ruolo valore="Professore"/>
    </docente>
    <docente codiceCorso="103">
      <nome>Paolo Verdi</nome>
    </docente>
    <assistente codiceCorso="386">
      <nome>Franco Neri</nome>
      <ruolo valore="Ricercatore"/>
    </assistente>
  </docenti>
  <corsi>
    <corso>
      <codice>386</codice>
      <nome>Informatica</nome>
      <cfu>12</cfu>
    </corso>
    <corso>
      <codice>103</codice>
      <nome>Analisi matematica</nome>
      <cfu>12</cfu>
    </corso>
  </corsi>
</dipartimento>
```

Codice 15: Esempio documento XML

identity-constraints è possibile vedere anche i componenti di dichiarazione degli elementi e degli attributi.

²⁷ Vedere sezione 1.4.1 per le espressioni *XPath*.

```

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="dipartimento" type="DipartimentoType">
    <xs:keyref name="corsoKeyRef" refer="corsoKey">
      <xs:selector xpath="dipartimento/*"/>
      <xs:field xpath="@codiceCorso"/>
    </xs:keyref>
    <xs:key name="corsoKey">
      <xs:selector xpath="./corso"/>
      <xs:field xpath="codice"/>
    </xs:key>
  </xs:element>
  <xs:complexType name="DipartimentoType">
    <xs:sequence>
      <xs:element name="docenti" type="DocentiType"/>
      <xs:element name="corsi" type="CorsiType"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="DocentiType">
    <xs:choice maxOccurs="unbounded">
      <xs:element name="docente" type="DocenteType"/>
      <xs:element name="assistente" type="DocenteType"/>
    </xs:choice>
  </xs:complexType>
  <xs:complexType name="DocenteType">
    <xs:sequence>
      <xs:element name="nome" type="xs:string"/>
      <xs:element name="ruolo" type="RuoloType" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="codiceCorso" type="xs:integer"/>
  </xs:complexType>
  <xs:complexType name="CorsiType">
    <xs:sequence>
      <xs:element name="corso" type="CorsoType"
        maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="CorsoType">
    <xs:sequence>
      <xs:element name="codice" type="xs:integer"/>
      <xs:element name="nome" type="xs:string"/>
      <xs:element name="cfu" type="xs:integer"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="RuoloType">
    <xs:attribute name="valore" type="xs:string"/>
  </xs:complexType>
</xs:schema>

```

Codice 16: Esempio di schema per il codice 15

Come si può vedere dal codice, oltre alla definizione dello schema degli elementi e attributi, viene usato un componente **xs:key** chiamato **corsoKey** che definisce l'esistenza e l'univocità di elementi **codice** figli di **corso**. Viene inoltre dichiarata una **xs:keyref** di nome **corsoKeyRef** con riferimento a **corsoKey** che garantisce che i valori degli attributi **codiceCorso** dei discendenti di **dipartimento** siano corrispondenti a quelli indicati nel componente **corsoKey**.

Si ricorda che per comprendere le espressioni *XPath* si rimanda alla sezione 1.4.1.

Componenti annotazione

Questi componenti servono per creare delle annotazioni adatte al consumo umano o anche da parte di un programma. Queste annotazioni non sono necessarie al funzionamento dello schema, ma sono utili per tenere traccia delle revisioni o commentare delle parti del codice di uno schema o anche per fornire ad un programma delle informazioni su un componente, ad esempio come trattare un tipo di dato.

Queste annotazioni possono essere di tre tipi e sono tutti e tre visibili nel codice di esempio 17 tutte all'interno di una dichiarazione di un tipo di dato

```
<xs:simpleType fn:note="special">
  <xs:annotation>
    <xs:documentation>A type for experts only</xs:documentation>
    <xs:appinfo>
      <fn:specialHandling>checkForPrimes</fn:specialHandling>
    </xs:appinfo>
  </xs:annotation>
</xs:simpleType>
```

Codice 17: Esempio di annotazioni

semplice. **xs:annotation** è il *contenitore* delle annotazioni e al suo interno può contenere gli altri due componenti: **xs:documentation** inteso per un consumo umano e **xs:appinfo** inteso per un consumo automatico.

È possibile anche fornire delle note usando un namespace diverso da quello dello schema, sia all'interno di un componente annotazione, sia per l'elemento che racchiude l'annotazione (come per **fn:note** nell'esempio).

1.4 FAMIGLIA XSL

XSL sta per *Extensible Stylesheet Language* e, come indica il nome, lo scopo della sua esistenza è quello di fornire un metodo per creare dei fogli di stile per i documenti XML. Con XSL è possibile dividere il contenuto dei documenti XML dalla sua presentazione, il modo in cui vengono visualizzati.

Per funzionare, *XSL* ha bisogno di un *processore XSL* che porti a compimento il procedimento di creare una presentazione per un documento *XML* (o anche più documenti). Tale *processore* esegue due passaggi (tralasciando i passaggi intermedi non utili alla comprensione), partendo dal documento *XML* iniziale effettua una *trasformazione* in un altro documento *XML* formattato secondo un linguaggio chiamato *XSL-FO*, *XSL Formatting Objects*, poi effettua una *formattazione* di tale documento *XSL-FO* interpretandolo e visualizzandolo, o stampandolo, o qualunque cosa a seconda del dispositivo che effettua tale passaggio. Quest'ultimo passaggio in realtà è composto da diversi sottopassaggi concatenati che non sono utili per lo scopo di questo testo²⁸.

Per poter fare la prima *trasformazione* è necessario dover fornire uno schema da seguire, per creare tale schema esiste un linguaggio apposta: *XSLT*, *XSL Transformation*. I documenti in formato *XSLT* prendono il nome di *stylesheet*.

Come spesso accade nel mondo dell'*XML*, cose che sono create con un certo scopo finiscono con l'essere utili anche in altri ambiti, così l'*XSLT* è talmente flessibile da poter essere usato per trasformare un *XML* in qualsiasi tipo di documento, non solo in documenti *XML-FO*, anche in documenti non *XML*. Per questo esistono dei *processori XSLT* che, di fatto, implementano solo la prima fase del *processore XSL* visto prima, prendendo come input il documento *XML* da trasformare e lo stylesheet *XSLT* e fornendo in output il risultato della trasformazione.

In figura 2 è possibile vedere uno schema del funzionamento di un *processore*

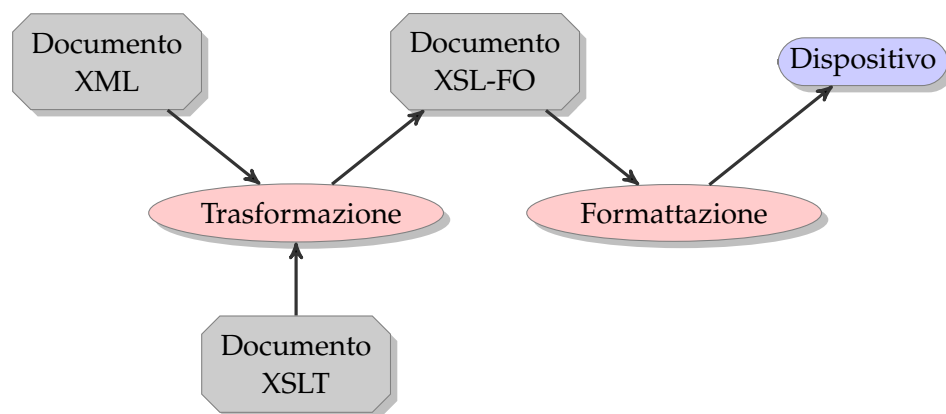


Figura 2: Schema del processore XSL

XSL.

Per il suo funzionamento *XSLT* usa un linguaggio per identificare parti del documento *XML* da trasformare. Questo linguaggio prende il nome di *XPath*. Nello specifico viene usato per *pescare* gli elementi o gli attributi, anche a blocchi

²⁸ Si veda <http://www.w3.org/TR/xsl/#d0e209> per una introduzione a tali sottopassaggi.

dal documento *XML*, per poi poter decidere cosa farne e come devono essere visualizzati nella trasformazione.

Di seguito vengono analizzati nel dettaglio *XPath* e *XSLT*. *XSL-FO* viene tralasciato in questo testo in quanto si tratta di un vocabolario non molto usato e non è necessario per gli scopi di questo scritto.

1.4.1 *XPath*

Lo scopo principale di *XPath* è quello di indirizzare parti di documenti *XML*. Pur essendo nato all'interno della famiglia *XSL*, e più precisamente all'interno di *XSLT*, esso è abbastanza versatile da poter essere usato anche in altri ambiti. Ad esempio esistono librerie *Java* che permettono di accedere a documenti *XML* tramite *XPath*.

XPath è formulato in modo da basarsi sulla *struttura logica* di un *XML* più che sulla struttura sintattica. Un documento *XML* viene interpretato come un *albero* di *nodi* dove ogni nodo può essere di tipo *elemento*, *attributo*, *testo*. Ognuno di questi nodi viene valutato come un valore testuale, qualche tipo di nodo ha anche un nome.

Il modo in cui *XPath* rappresenta il documento può essere derivato dagli *information items* forniti da *Infoset* (vedere sezione 1.1.1). Per dettagli su questa derivazione si rimanda all'indirizzo <http://www.w3.org/TR/xpath/#infoset>.

XPath, a differenza degli altri due linguaggi di *XSL*, non usa una sintassi *XML*, bensì consiste in una *espressione* la cui valutazione ritorna degli indirizzi di punti precisi del documento. Cosa fare con questi indirizzi dipende da chi usa *XPath*, ad esempio usato nelle *XSLT* è possibile prendere dei contenuti dagli attributi o elementi da poter portare nel documento risultante della trasformazione.

XPath supporta anche i *namespaces*, quindi i nomi dei nodi consistono in una *parte locale* e un *URI* del *namespace* eventualmente vuoto.

Più precisamente la valutazione dell'*espressione* ritorna uno dei seguenti possibili *oggetti*:

node-set una collezione non ordinata di nodi senza duplicati;

booleano vero o falso;

numero un numero floating point;

stringa una sequenza di caratteri.

La valutazione dell'*espressione* avviene relativamente ad un *contesto* che deve essere deciso da chi usa *XPath* e richiama l'*espressione*. Tale *contesto* consiste in:

context node un nodo che identifica il punto dal quale valutare una certa espressione;

context position e *Context size* due interi positivi tali che la *context position* è sempre minore o uguale della *context size*;

variable bindings un set che consiste in un *mapping* da nomi di variabile a valori di variabile. I valori possono essere del tipo dei risultati dell'espressione visti prima, o anche di altri tipi;

function library un set che consiste in un *mapping* da nomi di funzione a funzioni. Esiste una libreria di *core* che tutte le implementazioni di *XPath* implementano, inoltre gli utilizzatori di *XPath* possono estendere tale set con altre funzioni personalizzate;

namespace declaration consiste in un mapping dai prefissi dei *namespace* e i relativi *URI*.

I *variable bindings*, la *function library* e i *namespace declaration* rimangono costanti per tutta l'espressione. Il *context node*, la *context position* e la *context size* possono variare nella valutazione di *sottoespressioni* rispetto all'espressione contenitrice.

Una espressione deve essere nella forma²⁹ di **Expr** del seguente codice:

```
Expr ::= OrExpr

OrExpr ::= AndExpr
        | OrExpr 'or' AndExpr
AndExpr ::= EqualityExpr
        | AndExpr 'and' EqualityExpr
EqualityExpr ::= RelationalExpr
        | EqualityExpr '=' RelationalExpr
        | EqualityExpr '!=' RelationalExpr
RelationalExpr ::= AdditiveExpr
        | RelationalExpr '<' AdditiveExpr
        | RelationalExpr '>' AdditiveExpr
        | RelationalExpr '<=' AdditiveExpr
        | RelationalExpr '>=' AdditiveExpr

AdditiveExpr ::= MultiplicativeExpr
        | AdditiveExpr '+' MultiplicativeExpr
        | AdditiveExpr '-' MultiplicativeExpr
MultiplicativeExpr ::= UnaryExpr
        | MultiplicativeExpr MultiplyOperator UnaryExpr
        | MultiplicativeExpr 'div' UnaryExpr
        | MultiplicativeExpr 'mod' UnaryExpr
UnaryExpr ::= UnionExpr
        | '-' UnaryExpr
```

²⁹ In forma *Extended Backus Naur Form*.

```

UnionExpr      ::= PathExpr
                  | UnionExpr '|' PathExpr
PathExpr       ::= LocationPath
                  | FilterExpr
                  | FilterExpr '/' RelativeLocationPath
                  | FilterExpr '//' RelativeLocationPath
FilterExpr     ::= PrimaryExpr
                  | FilterExpr Predicate

LocationPath   ::= RelativeLocationPath
                  | AbsoluteLocationPath
AbsoluteLocationPath ::= '/'RelativeLocationPath?
                  | AbbreviatedAbsoluteLocationPath
RelativeLocationPath ::= Step
                  | RelativeLocationPath '/' Step
                  | AbbreviatedRelativeLocationPath
AbbreviatedAbsoluteLocationPath ::= '//'RelativeLocationPath
AbbreviatedRelativeLocationPath ::= RelativeLocationPath '//' Step

Step           ::= AxisSpecifier NodeTest Predicate*
                  | AbbreviatedStep
AxisSpecifier  ::= AxisName ':'
                  | AbbreviatedAxisSpecifier
AbbreviatedStep      ::= '.'
                  | '..'
AbbreviatedAxisSpecifier ::= '@'?

NodeTest      ::= NameTest
                  | NodeType '(' ')'
                  | 'processing-instruction' '(' Literal ')'
NameTest      ::= '*'
                  | NCName ':' '*'
                  | QName

Predicate     ::= '[' PredicateExpr ']'
PredicateExpr ::= Expr

PrimaryExpr   ::= VariableReference
                  | '(' Expr ')'
                  | Literal
                  | Number
                  | FunctionCall

VariableReference ::= '$' QName

Literal       ::= "'" [^']* "'"
                  | '"' [^"]* '"'
Number       ::= Digits ('.' Digits)?

```



```

| '.' Digits
Digits ::= [0-9]+

FunctionCall ::= FunctionName '(' ( Argument ( ',' Argument )* )? ')'
Argument ::= Expr
FunctionName ::= QName - NodeType

NodeType ::= 'comment'
           | 'text'
           | 'processing-instruction'
           | 'node'

AxisName ::= 'ancestor'
           | 'ancestor-or-self'
           | 'attribute'
           | 'child'
           | 'descendant'
           | 'descendant-or-self'
           | 'following'
           | 'following-sibling'
           | 'namespace'
           | 'parent'
           | 'preceding'
           | 'preceding-sibling'
           | 'self'

QName ::= PrefixedName
        | UnprefixedName
PrefixedName ::= Prefix ':' LocalPart
UnprefixedName ::= LocalPart
Prefix ::= NCName
LocalPart ::= NCName

NCName ::= Name - (Char* ':' Char*)

```

dove **Name** è nella stessa forma dei nomi XML visti nel codice 5 a pagina 11.

Location paths

Le espressioni più importanti sono quelle che usano **LocationPath** in quanto permettono di selezionare un set di nodi relativo al *context node* e di contenere ricorsivamente altre espressioni. Ci sono due tipi di *location path*: relativi e assoluti.

Quelli relativi consistono in una sequenza di uno o più *steps* separati da *"/*". Ogni *step* seleziona un set di nodi relativo ad un *context node*, poi ogni nodo di tale set è usato come *context node* per lo *step* successivo.

Un *path* assoluto consiste in un carattere “/” seguito da un *path* relativo. L’effetto di rendere un *path* assoluto è che il *context node* viene posto uguale alla radice del documento che conteneva il *context node* e poi viene valutato il *path* relativo indicato.

Un singolo *step* ha tre parti:

un *axis* che specifica la relazione nell’albero tra il nodo selezionato e il *context tree*;

un *note test* che specifica il tipo o il nome del nodo selezionato;

zero o più *predicati* che usano espressioni arbitrarie per raffinare il set di nodi selezionato.

Gli *axes* possono essere i seguenti:

child i figli del *context node*;

descendant i figli del *context node* e ricorsivamente tutti i discendenti;

parent il genitore del *context node*, se esiste;

ancestor il genitore e ricorsivamente tutti gli antenati;

following-sibling il fratello in ordine successivo del *context node*. I nodi di tipo attributo o namespace hanno *following-sibling* vuoto;

preceding-sibling come il precedente ma per il fratello precedente;

following tutti i nodi che sono successivi al *context node* nell’ordine del documento, escludendo i discendenti e i nodi di tipo attributo e namespace;

preceding come il precedente ma per i nodi precedenti;

attribute contiene tutti gli attributi del *context node*. Se il *context node* non è un elemento, allora è vuoto. Esiste una abbreviazione per questo *axis* che corrisponde a @;

namespace contiene tutti i namespace del *context node*. Se il *context node* non è un elemento, allora è vuoto;

self contiene il *context node* stesso;

descendant-or-self contiene il *context node* e i discendenti di questo;

ancestor-or-self come il precedente per gli antenati.

Si può notare che **ancestor**, **descendant**, **following**, **preceding** e **self** costituiscono una partizione dell'insieme di tutti i nodi di tipo elemento del documento di appartenenza del *context node*.

Passando ai *node test* è bene premettere che ogni *axis* ha il suo *tipo principale di nodo*, per **attribute** è *attributo*, per **namespace** è *namespace*, e per tutti gli altri è *elemento*. Detto ciò, i *node test* servono per limitare il set dei nodi dell'*axis*. Ogni *node test* torna un valore di verità per ogni nodo, se vero il nodo in questione viene selezionato, altrimenti no.

Osservando la grammatica si nota che un *node test* può essere nelle forme:

- **QName** è vero se il tipo del nodo è il tipo principale di nodo per l'*axis* in questione e se ha un nome esteso uguale al nome esteso specificato da **QName**. Ad esempio l'espressione **child::div** seleziona gli elementi *div* figli del *context node*, **attribute::href** seleziona gli attributi **href** del *context node*;
- **NCName:*** è vero per ogni nodo del tipo principale che ha l'*URI* del *namespace* uguale all'espansione del prefisso **NCName**;
- ***** è vero per ogni nodo di tipo uguale al tipo di nodo principale, ad esempio **child::*** seleziona tutti i figli del *context node*;
- **text()** è vero per ogni nodo *testo*. **child::text()** seleziona i nodi *testo* figli del *context node*;
- **comment()** è vero per ogni nodo *commento*;
- **processing-instruction()** è vero per ogni nodo *processing instruction*;
- **processing-instruction(LITERAL)** è vero per ogni *processing instruction* che ha nome uguale al valore **LITERAL**;
- **node()** è vero per ogni nodo di ogni tipo.

Per quanto riguarda i *predicati*, un *axis* può essere *discendente* che seleziona nodi nell'ordine del documento, o *ascendente* se li seleziona nell'ordine inverso. *ancestor*, *ancestor-or-self*, *preceding* e *preceding-sibling* sono gli *axis* ascendenti, gli altri sono discendenti.

La *proximity position* di un nodo membro di un *node-set* risultante di un *axis* indica la posizione di tale nodo all'interno del *node-set*. Se l'*axis* è discendente il *node-set* è ordinato nell'ordine del documento, viceversa, se è ascendente, è ordinato in senso contrario.

Un *predicato* filtra un *node-set* risultante da un *axis* nel seguente modo: per ogni nodo del *node-set* originale valuta l'espressione **PredicateExpr** usando come *context node* il nodo che sta valutando, come *context position* la *proximity position* del nodo che sta valutando e come *context size* il numero di nodi nel *node-set*.

L'espressione **PredicateExpr** è valutata determinando **Expr** e convertendo il risultato in un booleano. Se il risultato è un numero viene convertito in *vero* se è uguale alla *context position*, in *falso* altrimenti. Se è un *node-set* è *vero* se non è vuoto, *falso* altrimenti. Se è una stringa è *vero* se non è vuota, *falso* altrimenti.

Esistono delle utili abbreviazioni per rendere l'espressione più concisa e leggibile, queste abbreviazioni sono:

- **child::** può essere omesso, cosicché l'espressione **div/a** è equivalente a **child::div/child::a**;
- **attribute::** può essere abbreviato in **@**, ad esempio **child::div/child::a[attribute::href=abc]** diventa **div/a[@href=abc]**;
- **//** è l'abbreviazione di **/descendant-or-self::node()/**. Ad esempio **//a** è equivalente a **/descendant-or-self::node()/child::a** (è stata usata anche l'abbreviazione di **child::**);
- **.** è l'abbreviazione per **self::node()**;
- **..** è l'abbreviazione per **parent::node()**.

Funzioni

Come già detto *XPath* fornisce un set di funzioni di *core* che possono essere usate. Nella grammatica indicata prima è possibile vedere come usare le funzioni chiamate **FunctionCall**, comunque sia queste sono costituite da un nome, un valore di ritorno e un insieme di parametri.

Per vedere l'elenco completo di queste funzioni di *core* si rimanda a <http://www.w3.org/TR/xpath/#section-Node-Set-Functions>.

1.4.2 XSLT

Le trasformazioni *XSLT* sono espresse in *XML* e viene usato un *namespace* specifico per gli elementi che ha URI <http://www.w3.org/1999/XSL/Transform>. Solitamente si usa il prefisso **xsl** per tale *namespace*.

Come già detto in sezione 1.4, queste trasformazioni servono per poter trasformare tramite un *processore* un documento *XML* in un documento *XSL-FO*. O perlomeno questo era lo scopo iniziale. In realtà *XSLT* permette di trasformare un generico *albero sorgente* in un generico *albero destinazione*, anche non in formato *XML*.

Le trasformazioni prendono il nome di *stylesheet* perché, quando applicate per ottenere un documento *XSL-FO*, funzionano come un foglio di stile. Gli *stylesheet* sono composti da una serie di *regole template*. Ogni *regola template* è composta da due parti:

- un *pattern* da matchare con l'*albero sorgente*;
- un *template* da istanziare come parte dell'*albero destinazione*.

Un *template* può contenere elementi che specificano parti letterali della struttura del risultante, ma può anche contenere elementi del namespace *XSLT* che specificano istruzioni per creare frammenti dell'albero risultante. Durante l'istanziamento ogni istruzione è sostituita con il frammento corrispondente.

Quando un *template* è istanziato, avviene rispetto ad un *nodo corrente* e ad una *lista di nodi correnti*. Qualche istruzione cambia questi due valori, durante l'istanziamento di queste regole la *lista dei nodi correnti* cambia in una nuova lista di nodi e ogni nodo a turno diventa il *nodo corrente*. Dopo l'istanziamento il *nodo corrente* e la *lista dei nodi correnti* tornano ai valori precedenti.

Come già accennato, *XSLT* fa uso di *XPath* per selezionare elementi durante il processamento.

È possibile definire uno *stylesheet XSLT* per un certo documento XML direttamente all'interno di quest'ultimo, usando l'elemento **xsl:stylesheet**. Oppure è possibile creare un documento indipendente XML contenente esclusivamente lo *stylesheet*, questo si ottiene usando il solito elemento **xsl:stylesheet** come radice del documento.

I nodi di *input* della trasformazione vengono processati nel seguente modo:

1. viene processata inizialmente una lista contenente solo la radice dell'albero sorgente;
2. una lista di nodi viene processata concatenando, nell'ordine della lista stessa, i nodi risultanti dal processamento di ogni nodo della lista;
3. un nodo è processato cercando i *matching* con i *pattern* definiti nelle regole, scegliere il migliore se non univoco e istanziare il relativo *template* della regola usando il nodo come *nodo corrente* e la lista dei nodi sorgenti come *lista dei nodi correnti*;
4. un *template* può contenere istruzioni per selezionare una lista ulteriore di nodi da processare.

Patterns

I *patterns* usano la grammatica indicata nel codice 18 che si appoggia alla grammatica e ai tipi di dato di *XPath*³⁰ per completarsi.

Un *pattern* ha la funzione di trovare un *match* con un nodo del documento sorgente. Poiché il suo funzionamento è di fatto quello delle espressioni *XPath*, non viene approfondito ulteriormente l'argomento.

³⁰ Vedere sezione 1.4.1.

```

Pattern ::= LocationPathPattern
           | Pattern '|' LocationPathPattern
LocationPathPattern ::= '/' RelativePathPattern?
                       | IdKeyPattern (('/' | '//') RelativePathPattern)?
                       | '///'? RelativePathPattern
IdKeyPattern ::= 'id' '(' Literal ')'
                | 'key' '(' Literal ',' Literal ')'
RelativePathPattern ::= StepPattern
                       | RelativePathPattern '/' StepPattern
                       | RelativePathPattern '//' StepPattern
StepPattern ::= ChildOrAttributeAxisSpecifier NodeTest Predicate*
ChildOrAttributeAxisSpecifier ::= AbbreviatedAxisSpecifier
                                   | ('child' | 'attribute') '::'

```

Codice 18: Grammatica di un pattern XSLT

Template

Un *template* è una regola associata ad un *pattern* e viene definita con un elemento di tipo **xsl:template** che possiede un attributo **match** che corrisponde al *pattern*. Il contenuto dell'elemento **xsl:template** rappresenta l'albero di output della trasformazione applicata da questa specifica regola.

Un esempio di definizione di regola è quello del codice 19 che, quando esiste

```

<xsl:template match="in">
  <out/>
</xsl:template>

```

Codice 19: Esempio di definizione di regola XSLT

un *match* con un elemento **in**, produce un elemento **out**.

Come detto prima un *template* può avere delle istruzioni per processare una ulteriore lista di nodi, tale istruzione è **xsl:apply-template** con l'attributo opzionale **select** che deve essere una **Expr** di XPath³¹. Se **select** non è presente vengono aggiunti tutti i figli del *nodo corrente*. Ad esempio nel codice 20 quando avviene un *match* con un nodo di *input* **div**, vengono aggiunti alla lista di processamento tutti i **p** suoi discendenti.

La definizione dei **xsl:template** prevede di indicare un attributo **priority** con un valore numerico, che permette di identificare quale regola scegliere in caso di conflitti durante il *matching* di un nodo di *input*.

Le definizioni dei *template* danno la possibilità di definire un attributo **mode** che permette di creare una sorta di classe assegnandoli un certo nome. Quando un'applicazione **xsl:apply-templates** indica anch'essa un attributo **mode**,

³¹ Vedi nota 30.

```
<xsl:template match="div">
  <blocco>
    <xsl:apply-template select="p"/>
  </blocco>
</xsl:template>
```

Codice 20: Esempio di definizione di regola *XSLT* con applicazione di regola

allora i *matching* dei nodi verranno scelti solo tra i template che presentano un attributo **name** uguale.

Una definizione di *template* permette anche di assegnare un nome alla regola tramite l'attributo opzionale **name**. Se un *template* ha un nome definito, è possibile applicarlo direttamente senza *matching* con l'istruzione **xsl:call-template** e il suo attributo **name** che, ovviamente, deve corrispondere al nome del template. Nel caso di chiamata diretta gli attributi **match**, **mode** e **priority** vengono ignorati.

All'interno dei **xsl:template** è possibile indicare quale deve essere l'albero risultante del *template*, non solo scrivendolo direttamente nel contenuto della definizione come visto negli esempi prima, ma è possibile anche usare a tal scopo due istruzioni particolari: **xsl:element** e **xsl:attribute**.

Inoltre l'*XSLT* prevede anche delle istruzioni condizionali come **xsl:if** e **xsl:choose** e altre funzionalità che esulano dagli scopi di questo testo. Per approfondire si rimanda a <http://www.w3.org/TR/xslt>.

WEB SEMANTICO

Il web semantico nasce dalla necessità di rendere i contenuti in rete disponibili non solo agli esseri umani, ma di cercare di renderli accessibili anche alle macchine.

Attualmente la maggior parte dell'informazione contenuta nel web è concepita principalmente per essere fruita dagli esseri umani, se io voglio sapere quali sono i corsi di un certo professore devo andare sulla sua pagina web, cercare la sezione insegnamenti ed elaborare il testo. Se volessi far fare lo stesso lavoro ad una macchina dovrei prevedere tecniche di intelligenza artificiale e di *NLP* (Natural Language Processing). Il web semantico cerca di fornire un'alternativa all'uso di questi mezzi agendo direttamente all'origine, ossia organizzando il contenuto informativo di una certa pagina in modo da rendere un programma in grado di comprenderne il contenuto semantico senza effettuare alcun processo sul testo. Un programma quindi potrebbe direttamente venire a conoscenza della relazione che intercorre tra un certo professore e i corsi da lui insegnati.

Il *web semantico* è sostenuto dal W_3C^1 ed è stato promosso inizialmente da *Tim Berners Lee*.

2.1 VISIONE D'INSIEME

Nascendo come evoluzione del web come attualmente lo conosciamo, il web semantico riprende alcuni concetti ad esso associati, nel dettaglio:

- ognuno può inserire qualsiasi tipo di contenuto ed informazione e può contribuire alla creazione della conoscenza per un certo argomento;
- vige l'ipotesi del mondo aperto, ossia il fatto di non trovare alcuna sentenza su un dato argomento non implica la negazione di questa sentenza;
- la stessa entità può essere chiamata con nomi diversi, ad esempio a seconda di chi ne parla.

Nella progettazione del web semantico è stato deciso di tenere in considerazione questi punti, cosicché potesse essere *facilmente* integrato all'interno della struttura esistente.

¹ L'indirizzo del gruppo specifico che si occupa di web semantico è: <http://www.w3.org/2001/sw/>.

Gli standard del web semantico sono organizzati a *strati* con un livello crescente di *espressività*, nel senso che ogni strato fornisce un linguaggio in grado di modellare concetti in domini diversi e di complessità crescente, inoltre ogni linguaggio basa la sua esistenza su quello dello strato precedente. Questi strati sono:

RDF *Resource Description Framework*, è il *layer* di base, questo permette di esprimere qualsiasi sentenza di base e fornisce un modello comune per farlo;

RDFS *RDF Schema Language*, è un linguaggio che permette l'espressività tipica dei linguaggi ad oggetti, permettendo di organizzare *classi*, *sottoclassi* e *proprietà*;

OWL *Web Ontology Language*, fornisce la potenza della logica (descrittiva) al web semantico aggiungendo capacità di *reasoning*.

Di seguito verranno analizzati dettagliatamente tutti questi livelli. È bene sapere anche che, oltre a questi, esistono altri livelli, per esempio esiste *RDFS-Plus* che è un sottoinsieme di *OWL* e quindi un livello intermedio tra *RDFS* e *OWL*. Inoltre, essendo una tecnologia in fase di sviluppo, esistono anche livelli previsti ma ancora non esistenti. In figura 3 si può vedere lo stack così come progettato da *Tim Berners Lee*, la linea tratteggiata divide la parte realizzata (inferiore) da quella non implementata (superiore). Questi strati non sono da considerarsi livelli nettamente separati gli uni dagli altri, è possibile immaginarli come strutture che aumentano le funzionalità di un'unica entità. I tre strati inferiori appartengono alle tecnologie del web preesistenti, non strettamente collegate con il web semantico ma su cui esse si appoggiano:

URI *Uniform Resource Identifier* come dice il nome permettono di identificare univocamente una risorsa nel web, successivamente verrà analizzato dettagliatamente il loro uso nel web semantico;

Unicode è la codifica dei caratteri;

XML *eXtensible Markup Language* è un linguaggio di *markup* che permette la creazione di documenti strutturati.

In realtà *RDF* non si basa necessariamente su *XML* e neppure su *Unicode* in quanto esistono anche rappresentazioni alternative che verranno descritte in seguito nel dettaglio.

Brevemente, le parti non ancora implementate sono:

Rules permetteranno di aggiungere il supporto per le regole, in pratica permetteranno di ampliare la *logica descrittiva* di *OWL* rendendola (circa) una *logica a clausole di Horn*;

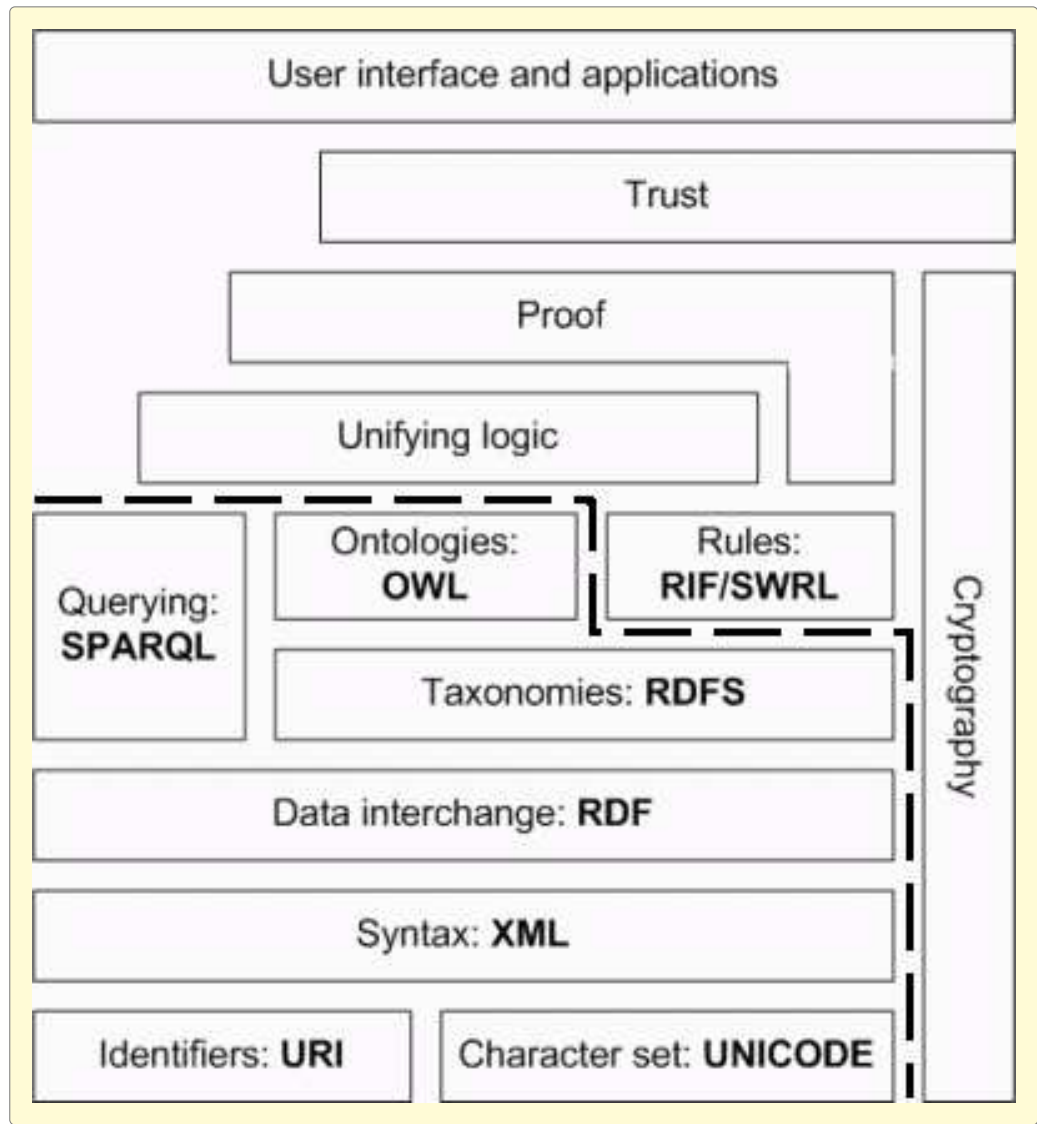


Figura 3: Stack web semantico

Cryptography aggiungerà il supporto alla crittografia per verificare le fonti;

Trust verificherà l'affidabilità delle sentenze derivate verificando le fonti e basandosi sulla *logica formale*;

User interface sarà l'interfaccia utente che permetterà agli umani l'uso delle applicazioni web semantiche.

Per quanto riguarda *SPARQL*, esso è un linguaggio di quering e serve per poter interrogare i dati basati su *RDF*.

2.2 RESOURCE DESCRIPTION FRAMEWORK (RDF)

L'*RDF* indica il modo in cui i dati o meglio le *risorse* e le relazioni tra di esse vengono rappresentate. Con risorsa si può intendere qualsiasi cosa in quanto *RDF* è stato pensato per avere un alto potere descrittivo. I concetti base dell'*RDF* sono gli *URI* e le *triple*. Degli uri e dei namespaces abbiamo già parlato nella sezione 1.2, e abbiamo anche parlato della sintassi *qname*. Nell'*RDF* (perlomeno nelle sue serializzazioni *XML/RDF* e *N3* che saranno analizzate successivamente nella sezione 2.2.3) si possono usare i *namespaces* e la sintassi *qname* in modo assolutamente analogo a quello dell'*XML*.

Sia l'*RDF* che l'*RDFS* che *OWL* prevedono dei *namespaces*, questi sono rispettivamente:

- <http://www.w3.org/1999/02/22-rdf-syntax-ns#>;
- <http://www.w3.org/2000/01/rdf-schema#>;
- <http://www.w3.org/2002/07/owl#>.

Quando in seguito verranno citati i prefissi **rdf:**, **rdfs:** e **owl:**, se non diversamente specificato, dovranno essere considerati prefissi mappati ai sudetti *namespaces*.

2.2.1 Triple

Così come gli *URI* rappresentano le risorse, le *triple* implementano le relazioni che intercorrono tra di esse e le proprietà che esse hanno. Queste triple in pratica rappresentano delle sentenze, concettualmente assumono la forma di tre valori ordinati che prendono in prestito la terminologia dalla grammatica: *soggetto*, *predicato* e *oggetto*. Il *soggetto* si riferisce ad una specifica risorsa quindi deve essere indicato con un *URI*, il *predicato* indica la proprietà che la tripla descrive e anche questa deve essere indicata univocamente con un *URI*, l'*oggetto*

è il valore della proprietà, questo può essere sia un'altra risorsa (quindi un *URI*) oppure un letterale.

Un modo per rappresentare e comprendere meglio un insieme di triple è quello di disegnare un grafo orientato facendo in modo che i soggetti e gli oggetti siano i nodi del grafo (distinguendo gli oggetti che sono *URI* da quelli letterali) e tanti archi quante sono le triple, orientati da soggetto ad oggetto. Ad esempio: volendo indicare delle sentenze riguardo ad un professore, innanzitutto si deve indicare il soggetto ed i predicati con degli uri, poi si può usare **<http://www.unifi.it/rdf/namespace#>** o qualcosa di simile come namespace per gli *URI* e si indica in seguito con il prefisso **unifi**. Si crea così la prima tripla dove vengono usati i *qnames* per chiarezza e compattezza, anche se questi non potranno essere usati in tutte le serializzazioni di *RDF*; i letterali invece vanno racchiusi tra virgolette. Ne risulta che la tripla indicata nella tabella 2

Soggetto	Predicato	Oggetto
unifi:mario-rossi	unifi:nome	"Mario Rossi"

Tabella 2: Esempio di tripla

può essere rappresentata semplicemente con il grafo in figura 4.

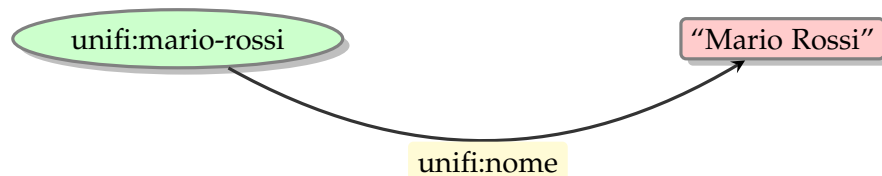


Figura 4: Esempio grafo tripla

Viene elaborato di seguito un set più sostanzioso di quello precedente. Se si vuole rappresentare delle informazioni su del personale di un certo dipartimento, per esempio, si indica per ognuno il nome, il ruolo e il dipartimento di appartenenza. Per quanto riguarda invece il dipartimento, si indica il nome. Vengono definiti i seguenti namespace per poter usare la forma compatta: **unifi**=**<http://www.unifi.it/rdf/unifi#>**, **prof**=**<http://www.unifi.it/rdf/prof#>**, **dip**=**<http://www.unifi.it/rdf/dip#>**. Le proprietà della persona sono rispettivamente: **unifi:ha-nome**, **unifi:ha-ruolo** e **unifi:afferisce-a**. La proprietà del dipartimento, usando lo stesso namespace di quelle della persona, è **unifi:ha-nome**. Come *URI* delle persone e dei dipartimenti vengono usati **prof:nome-cognome** e **dip:nome**. Viene così costruita la tabella 3, e si può notare che vi sono rappresentate le triple riguardanti tre

Soggetto	Predicato	Oggetto
prof:mario-rossi	unifi:ha-nome	"Mario Rossi"
prof:mario-rossi	unifi:ha-ruolo	"Professore"
prof:mario-rossi	unifi:afferisce-a	dip:sisinfo
prof:paolo-verdi	unifi:ha-nome	"Paolo Verdi"
prof:paolo-verdi	unifi:ha-ruolo	"Ricercatore"
prof:paolo-verdi	unifi:afferisce-a	dip:sisinfo
prof:franco-neri	unifi:ha-nome	"Franco Neri"
prof:franco-neri	unifi:ha-ruolo	"Professore"
prof:franco-neri	unifi:afferisce-a	dip:chimica
dip:sisinfo	unifi:ha-nome	"sistemi e informatica"
dip:chimica	unifi:ha-nome	"Chimica"

Tabella 3: Triple dei professori e dipartimenti

persone e due dipartimenti, per maggiore chiarezza viene mostrato il grafico corrispondente a queste triple in figura 5.

Una delle caratteristiche più utili dell'*RDF* è la possibilità di *combinare* dati da fonti diverse. Poiché gli *URI* identificano univocamente una risorsa, si possono creare delle triple facendo riferimento a degli *URI* anche esterni al sistema. Ad esempio, se da qualche parte è presente un *RDF* che rappresenta l'elenco delle città italiane, è possibile usare gli uri presenti per aggiungere delle informazioni alle triple della figura 5 sulla residenza dei professori o sulla sede dei dipartimenti. Tutto ciò è conforme al principio enunciato nella sezione 2.1 secondo cui ognuno può aggiungere informazione riguardante un concetto. Gli *URI* garantiscono che ci si riferisca effettivamente alle stesse cose in punti diversi e l'estrema flessibilità dell'*RDF* permette di creare ed estendere nuovi concetti quando necessario.

2.2.2 Blank nodes (*bnodes*)

I *blank nodes*, anche detti *bnodes*, servono ad indicare un soggetto od un oggetto quando non si hanno informazioni su di esso, o quando si desidera che tale entità non abbia un *URI*. Alcune serializzazioni permettono anche di usare i *bnodes* per il predicato, ma non verrà trattato il caso in quanto non è di nessuna utilità pratica.

Nella pratica i *bnode* offrono la possibilità di creare una o più triple che hanno come *soggetto* o *oggetto* nessun uri, nella sintassi si indica un *bnode* con il

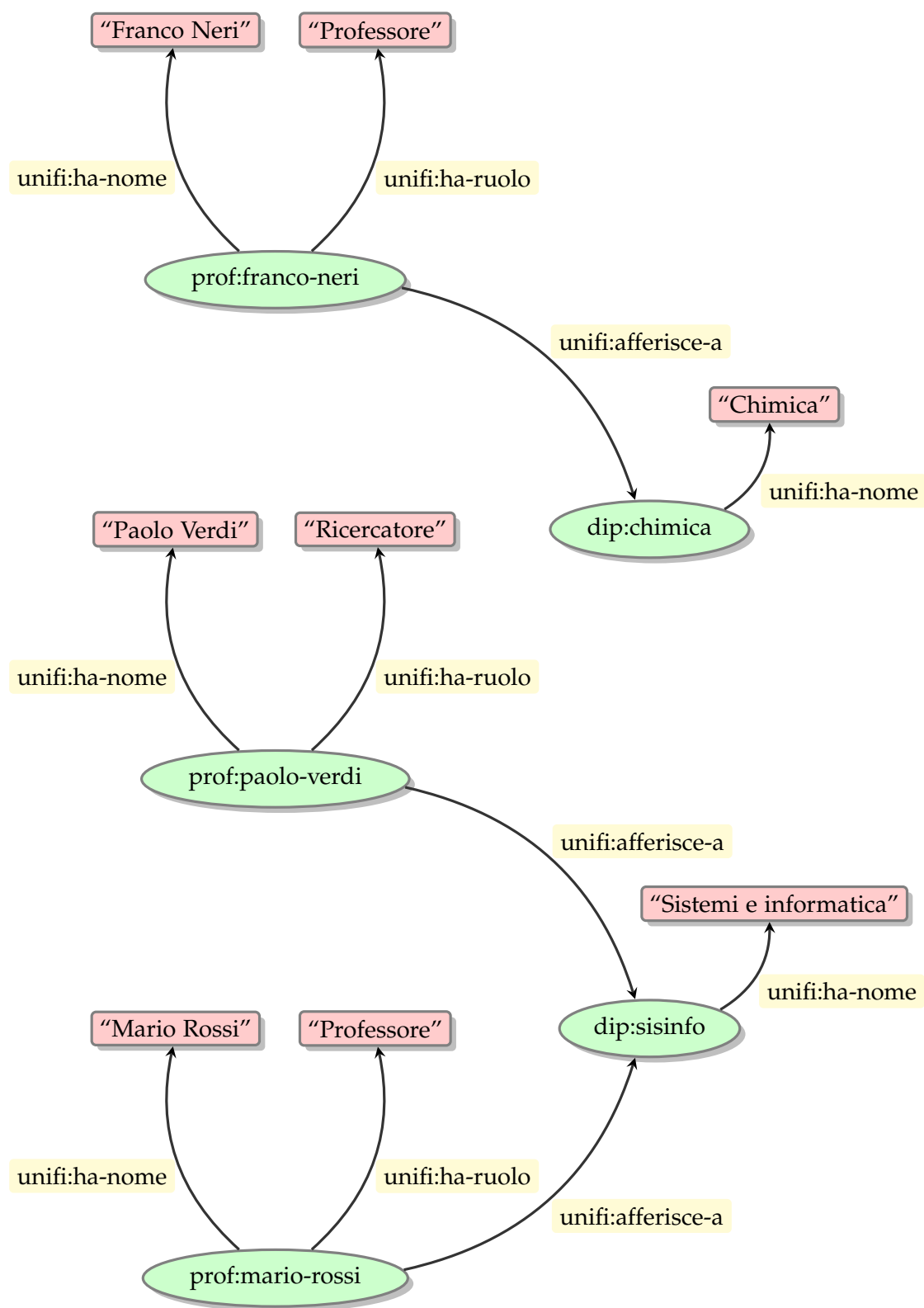


Figura 5: Grafo delle triple dei professori e dipartimenti

prefisso `_` seguito da un eventuale identificatore che indicherà univocamente un certo *bnode*. Concettualmente i *bnodes* possono essere interpretati come delle variabili quantificate esistenzialmente nella logica dei predicati del primo ordine, le triple della tabella 4 sono estratte da quelle viste prima nella tabella

Soggetto	Predicato	Oggetto
<code>_:X</code>	<code>unifi:ha-nome</code>	"Mario Rossi"
<code>_:X</code>	<code>unifi:ha-ruolo</code>	"Professore"
<code>_:X</code>	<code>unifi:afferisce-a</code>	<code>_:Y</code>
<code>_:Y</code>	<code>unifi:ha-nome</code>	"sistemi e informatica"

Tabella 4: Esempio di *bnode*

3 a cui sono stati sostituiti gli uri delle *entità* che rappresentavano il professore Mario Rossi e il dipartimento di sistemi e informatica con dei *bnodes*. Adesso il significato che assumono è che "esiste un *qualcosa* che ha nome Mario Rossi e ruolo Professore che afferisce a *qualcos'altro* che ha nome sistemi e informatica".

I *bnodes* sono molto usati in OWL per la loro affinità con le variabili quantificate esistenzialmente.

2.2.3 Serializzazioni

In questa sezione viene trattata la rappresentazione pratica dei dati denominati nelle sezioni precedenti come triple, per poterli scambiare e memorizzare. Esistono tre tipi di serializzazioni: *N-Triples*, *Notation 3 RDF* e *RDF/XML*.

N-Triples

È la forma più semplice e consiste in un susseguirsi di triple delimitate da un carattere `..`. Gli *URI* si indicano racchiudendoli tra `<...>`, i *bnodes* esattamente come nella sezione 2.2.2, ossia `_:X`, i letterali invece si racchiudono tra virgolette `"..."`. Ad esempio, le triple della tabella 4 sono serializzate in *ntriples* come nel blocco seguente:

```
_:X <http://www.unifi.it/rdf/unifi#ha-nome> "Mario Rossi" .
_:X <http://www.unifi.it/rdf/unifi#ha-ruolo> "Professore" .
_:X <http://www.unifi.it/rdf/unifi#afferisce-a> _:Y .
_:Y <http://www.unifi.it/rdf/unifi#ha-nome> "sistemi e informatica" .
```

È da notare che la notazione *n-triples* non permette l'uso dei *qnames*.

Notation 3 RDF

Anche abbreviata con N_3 , è una serializzazione più compatta di *n-triples*, ideata da Tim Berners Lee. Estende la sintassi *n-triples* aggiungendovi scorciatoie. Innanzitutto permette l'uso dei *qnames* e per dichiararli si ricorre alla seguente sintassi: @prefix unifi:<http://www.unifi.it/rdf/unifi#> poi, una volta dichiarati, possono essere usati nella scrittura delle triple con la solita sintassi:

```
_:X unifi:ha-nome "Mario Rossi" .
_:X unifi:ha-ruolo "Professore" .
_:X unifi:afferisce-a _:Y .
_:Y unifi:ha-nome "sistemi e informatica" .
```

Un'altra particolarità è che N_3 permette di raggruppare le triple che hanno lo stesso soggetto, separando le coppie di predicato e di oggetto con il carattere ; nel seguente modo:

```
_:X
    unifi:ha-nome "Mario Rossi" ;
    unifi:ha-ruolo "Professore" ;
    unifi:afferisce-a _:Y ;
_:Y unifi:ha-nome "sistemi e informatica" .
```

Inoltre N_3 permette, allo stesso modo, di raggruppare triple che hanno il solito soggetto e il solito predicato, separando i soggetti con delle , nel seguente modo:

```
_:X unifi:ha-figlio
    "Mario Rossi" ,
    "Paolo Bianchi" .
```

N_3 mette a disposizione anche delle scorciatoie, per esempio **rdf:type** può semplicemente essere indicata con **a**.

RDF/XML

È la notazione più usata in quanto sfrutta il ben noto e supportato XML per la rappresentazione delle triple. Per meglio comprendere la sintassi è utile partire dalla rappresentazione a grafo dell'*RDF* come già visto prima, usiamo a tal proposito il grafo della figura 6. Dove il nodo vuoto è un *bnode*², e i prefissi

corrispondono a:

```
prof = "http://www.unifi.it/rdf/prof#"
unifi = "http://www.unifi.it/rdf/namespace#"
```

Seguendo le *syntax specification* dell'*RDF/XML*³ vengono trascritti nell'*XML* i nodi *non letterali* del grafo come elementi **rdf:Description** e gli archi come elementi con il loro nome; convenzionalmente si indicano i primi come *elementi*

² Senza nessuna label in quanto non ci interessa.

³ Visibili all'indirizzo <http://www.w3.org/TR/rdf-syntax-grammar/>.

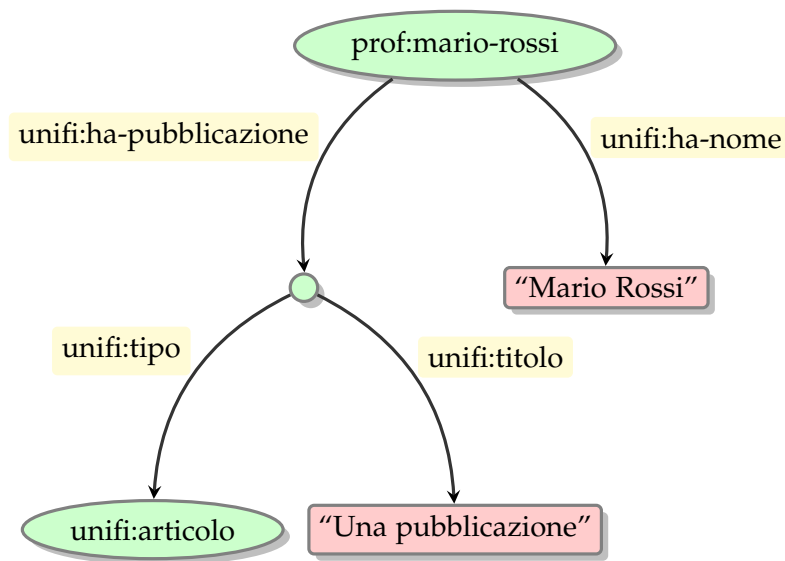


Figura 6: Grafo di esempio per *RDF/XML*

nodo e i secondi come *elementi proprietà*. È necessario trascriverli facendo in modo che ogni *elemento proprietà* sia figlio dell'*elemento nodo* che rappresenta l'origine dell'arco e padre dell'*elemento nodo* che rappresenta la destinazione dell'arco.

Se l'arco finisce in un *letterale* allora l'*elemento proprietà* che rappresenta l'arco non è padre di un elemento, ma contiene il testo del *letterale*. Gli *uri* dei nodi, quando sono presenti⁴, sono indicati con un attributo di tipo **rdf:about** dentro l'*elemento nodo* che li rappresenta.

Gli elementi possono essere sia rappresentati sequenzialmente, svincolati gli uni dagli altri, che messi in cascata, raggruppandoli gli uni dentro agli altri. Poiché l'*XML* necessita di una radice è previsto un elemento **rdf:RDF** da usare a tal proposito. Tale elemento è opzionale nel caso in cui il documento contenga già una sola radice, comunque un uso di tale elemento può essere anche quello di indicare i *namespaces* che servono per i *prefissi* dei *qnames* (ed anche il *namespace rdf*).

Il grafo della figura 6 può essere serializzato nel seguente modo:

```
<?xml version="1.0"?>
<rdf:RDF
  xmlns:rdf=
    "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:unifi=
    "http://www.unifi.it/rdf/namespace#">
  <rdf:Description
```

⁴ Ossia quando non sono bnodes.

```

    rdf:about=
      "http://www.unifi.it/rdf/prof#mario-rossi">
  <unifi:ha-pubblicazione>
    <rdf:Description>
      <unifi:tipo>
        <rdf:Description
          rdf:about=
            "http://www.unifi.it/rdf/namespace#articolo"
          />
        </unifi:tipo>
        <unifi:titolo>Una pubblicazione</unifi:titolo>
      </rdf:Description>
    </unifi:ha-pubblicazione>
    <unifi:ha-nome>Mario Rossi</unifi:ha-nome>
  </rdf:Description>
</rdf:RDF>

```

Si nota che non può essere usato il *namespace* per risolvere gli uri degli *elementi nodo*. Nell'esempio il *bnode* non ha nessuna etichetta; nel caso in cui fosse necessario attribuirne una, si usa l'attributo **rdf:nodeID** del *nodo elemento* che rappresenta il *bnode*.

La sintassi *RDF/XML* prevede numerose funzionalità e scorciatoie, ne enunciamo alcune:

- un *elemento nodo* che non ha nessun figlio può essere inglobato nell'*elemento proprietà* padre tramite l'attributo **rdf:resource**;
- se un *elemento proprietà* ha solo un figlio che è un *elemento nodo letterale* allora entrambi possono essere inglobati nell'*elemento nodo* padre, usando il nome dell'*elemento proprietà* come nome di attributo e il *letterale* come valore di tale attributo;
- i *bnodes* senza id possono essere omessi usando l'attributo **rdf:parseType=Resource** che rende l'*elemento proprietà* che era padre del *bnode* anche un *elemento nodo* che quindi prende i figli che erano del *bnode*.

Usando tali scorciatoie è possibile scrivere il codice in modo più compatto:

```

<?xml version="1.0"?>
<rdf:RDF
  xmlns:rdf=
    "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:unifi=
    "http://www.unifi.it/rdf/namespace#">
  <rdf:Description
    rdf:about=
      "http://www.unifi.it/rdf/prof#mario-rossi"
    unifi:titolo="Una pubblicazione">

```

```

<unifi:ha-pubblicazione rdf:parseType="Resource">
  <unifi:tipo
    rdf:resource=
      "http://www.unifi.it/rdf/namespace#articolo"
  />
</unifi:ha-pubblicazione>
<unifi:ha-nome>Mario Rossi</unifi:ha-nome>
</rdf:Description>
</rdf:RDF>

```

Per ulteriori informazioni sulla sintassi si rimanda al link:
<http://www.w3.org/TR/rdf-syntax-grammar/>.

2.2.4 Reificazione

La *reificazione* nasce dalla impossibilità di esprimere certi concetti con il sistema di triple visto precedentemente, ad esempio, è semplice modellizzare il concetto: «Mario Rossi ha pubblicato “Una pubblicazione”», ma asserire che «Paolo Verdi ha fatto da relatore a tale pubblicazione» è più complesso. Il problema è che sarebbe necessario poter fare delle *asserzioni* sopra un'altra *asserzione*, la *reificazione* implementa proprio questa possibilità.

L'*RDF* mette a disposizione un vocabolario per implementare la *reificazione*.

È possibile vedere nella tabella 5 un esempio di una tripla da *reificare*, in 6 la

Soggetto	Predicato	Oggetto
prof:mario-rossi	unifi:pubblica	“Una Pubblicazione”

Tabella 5: Tripla di esempio

Soggetto	Predicato	Oggetto
pubb:pubb123	rdf:type	rdf:Statement
pubb:pubb123	rdf:subject	prof:mario-rossi
pubb:pubb123	rdf:predicate	unifi:pubblica
pubb:pubb123	rdf:object	“Una Pubblicazione”

Tabella 6: Esempio di reificazione

stessa tripla *reificata* con l'*uri* **pubb:pubb123**, e in 7 un esempio di come un'altra tripla si può riferire a questa.

Soggetto	Predicato	Oggetto
prof:paolo-verdi	unifi:relatore	pubb:pubb123

Tabella 7: Esempio di asserzione su una asserzione

Gli elementi del vocabolario che servono per il procedimento sono:

rdf:type e **rdf:Statement** indicano che il soggetto **pubb:pubb123** è di tipo *Statement* quindi che rappresenta una tripla.

rdf:subject indica il *soggetto* della tripla reificata.

rdf:predicate indica il *predicato* della tripla reificata.

rdf:object indica l'*oggetto* della tripla reificata.

2.2.5 Tipi di dato

Formalmente un *tipo di dato* è composto da tre parti:

- uno spazio lessicale, un set non vuoto di stringhe di caratteri.
- uno spazio di valori, un set non vuoto.
- un mapping dallo *spazio lessicale* allo *spazio di valori*.

RDF supporta un *subset* dei tipi di dato definiti in *XML Schema*⁵, inoltre fornisce un singolo tipo di dato:

rdf:XMLElement

definito come⁶:

spazio lessicale: l'insieme delle stringhe che sono XML ben bilanciato, in forma canonica esclusiva⁷, per il quale l'*embedding* tra uno *start tag* ed un *end tag* porta alla formazione di un documento conforme con i *namespace*;

spazio di valori: un set di entità che è disgiunto dallo spazio lessicale, disgiunto da ogni spazio di valori di ogni tipo di dato dell'*XML Schema*, disgiunto dal set delle stringhe delle stringhe dei caratteri *unicode* e con una corrispondenza uno ad uno con lo spazio lessicale;

⁵ Vedere sezione 1.3.2.

⁶ Vedere <http://www.w3.org/TR/rdf-concepts/#section-XMLLiteral>.

⁷ Vedere <http://www.w3.org/TR/xml-exc-c14n/>.

mapping: un mapping uno ad uno tra i due spazi, iniettivo e suriettivo.

Per una definizione formale dei tipi di dato in *RDF* e per sapere qual è il subset preciso dei tipi di dato di *XML Schema* si rimanda al link <http://www.w3.org/TR/rdf-mt/#DTYPEINTERP>.

2.3 RDF SCHEMA (RDFS)

L'*RDF Schema* è una estensione semantica dell'*RDF*, esso fornisce un vocabolario che permette di definire e descrivere classi, proprietà e altre risorse, in modo simile a come fanno linguaggi ad oggetti come il *Java*. Viene usata la notazione **prefix:suffix** dove **prefix** è uno dei prefissi già visti nella sezione 2.2:

- **rdf** = <http://www.w3.org/1999/02/22-rdf-syntax-ns#>;
- **rdfs** = <http://www.w3.org/2000/01/rdf-schema#>.

Viene usato anche il namespace **rdf** nonostante si tratti di *RDFS* perché, come già detto precedentemente, non esiste una divisione netta tra i livelli che compongono il *web semantico*, queste sono tecnologie nate per gradi e sviluppate l'una sull'altra nel tempo.

Ad esempio *RDF* fornisce già il supporto per i contenitori come **rdf:bag**, nel sistema di classi di *RDFS* quest'ultimo viene formalizzato rendendolo sottoclasse di **rdfs:Container**. I concetti di classi e sottoclassi in *RDF* verranno spiegati di seguito. La *reificazione* vista prima nella sezione 2.2.4 viene in questa sezione formalizzata.

Buona parte delle informazioni contenute in questa sezione possono essere reperite nella raccomandazione del W3C dove è descritto il vocabolario *RDFS*: <http://www.w3.org/TR/rdf-schema/>.

Il vocabolario principale dell'*RDFS* può essere diviso tra:

- classi;
- proprietà.

Le prime identificano dei gruppi di risorse, le seconde specificano relazioni tra il soggetto e l'oggetto. Oltre a questi due vocabolari esistono anche altri vocabolari, alcuni estendono e formalizzano concetti appartenenti a *RDF*.

2.3.1 Classi

Le risorse possono essere divise in *classi*, i membri di tali classi sono chiamate *istanze*, le classi sono a sua volta delle *risorse*.

La proprietà **rdf:type** è usata per indicare che una risorsa è una istanza di una classe. L'insieme delle istanze di una certa classe è chiamato *estensione della*

classe. una classe ha ovviamente una sola estensione, però due classi diverse possono avere la stessa estensione, ossia, a differenza di altri linguaggi ad oggetti come il Java, una risorsa può essere istanza di più classi. Una classe può essere una istanza di se stessa. Il gruppo delle risorse che sono classi è a sua volta una classe, precisamente la classe **rdfs:Class**.

Si possono definire delle *sottoclassi* tramite la proprietà **rdfs:subClassOf**: se una classe "B" è sottoclasse di "A" allora ogni istanza di "B" è anche una istanza di "A". Il termine *superclasse* è usato come inverso di *sottoclasse*: se "B" è una sottoclasse di "A" allora "A" è una superclasse di "B".

Le classi principali sono:

rdfs:Resource è la classe di qualsiasi risorsa, tutte le altre classi sono sottoclassi di questa classe.

rdfs:Resource è una istanza di **rdfs:Class**;

rdfs:Class indica le risorse che sono classi, ossia tutte le istanze di **rdfs:Class** rappresentano delle classi.

rdfs:Class è una istanza di **rdfs:Class** e sottoclasse di **rdfs:Resource**;

rdfs:Literal è la classe dei valori letterali come stringhe e interi, possono essere *tipati* o *plain*. Quelli tipati sono istanze di una classe *datatype*, non esiste la classe di quelli plain.

rdfs:Literal è una istanza di **rdfs:Class** e sottoclasse di **rdfs:Resource**;

rdfs:Datatype è la classe dei tipi di dato, ogni istanza di **rdfs:Datatype** è una sottoclasse di **rdfs:Literal**,

rdfs:Datatype è una istanza e una sottoclasse di **rdfs:Class**;

rdf:XMLLiteral è la classe dei letterali nella serializzazione RDF/XML.

rdf:XMLLiteral è una istanza di **rdfs:Datatype** e una sottoclasse di **rdfs:Literal**;

rdf:Property è la classe delle proprietà, tutte le proprietà sono istanze di questa.

rdf:Property è una istanza di **rdfs:Class**.

2.3.2 Proprietà

Le *proprietà* sono relazioni che legano una *risorsa oggetto* ad una *risorsa soggetto*, di fatto le proprietà sono i *predicati* delle triple viste nella sezione 2.2. Ciò che implementa l'*RDFS* è la possibilità di definire *Sottoproprietà* tramite la proprietà **rdfs:subPropertyOf**, e di definire un *dominio* e un'*immagine* alla proprietà tramite le proprietà **rdfs:domain** ed **rdfs:range**.

Se una proprietà “B” è *sottoproprietà* di una proprietà “A”, allora ogni coppia di risorse che viene posta in relazione da “B” è anche in relazione tramite “A”. Esiste anche il termine *superproprietà* per riferirsi all’opposto di sottoproprietà: se “B” è sottoproprietà di “A”, allora “A” è superproprietà di “B”.

Per quanto riguarda il *dominio* e l'*immagine* di una proprietà, essi servono ad indicare tra quali istanze di classi è possibile applicare tale proprietà. Il *dominio* definisce la classe del soggetto della proprietà, l'*immagine* quella dell’oggetto. La terminologia è uguale a quella matematica perché il loro significato è simile a quello delle funzioni: così come una funzione mette in relazione l’insieme dei punti del dominio con quelli del codominio, una proprietà mette in relazione l’insieme delle risorse del dominio con quelle dell’immagine.

Le proprietà principali sono:

rdfs:domain definisce il dominio di una proprietà, la tripla

P rdfs:domain C

indica che **P** è una istanza di **rdf:Property**, **C** è una istanza di **rdfs:Class** e le risorse denotate dai *soggetti* delle triple il cui *predicato* è **P** sono istanze della classe **C**. Se una proprietà ha più domini definiti, allora i *soggetti* delle triple il cui *predicato* è **P** sono istanze di tutte le classi **C** definite da tali domini.

rdfs:domain è una istanza di **rdf:Property**, il dominio di **rdfs:domain** è **rdf:Property**, l’immagine di **rdfs:domain** è **rdfs:Class**;

rdfs:range definisce l’immagine di una proprietà, la tripla

P rdfs:range C

indica che **P** è una istanza di **rdf:Property**, **C** è una istanza di **rdfs:Class** e le risorse denotate dagli *oggetti* delle triple il cui *predicato* è **P** sono istanze della classe **C**. Se una proprietà ha più immagini definite, allora gli *oggetti* delle triple il cui *predicato* è **P** sono istanze di tutte le classi **C** definite da tali immagini.

rdfs:range è una istanza di **rdf:Property**, il dominio di **rdfs:range** è **rdf:Property**, l’immagine di **rdfs:range** è **rdfs:Class**;

rdf:type indica che una risorsa è una istanza di una certa classe. La tripla

R rdf:type C

indica che **C** è una istanza di **rdfs:Class** e che **R** è una istanza di **C**.

Il dominio di **rdf:type** è **rdfs:Resource**, l'immagine è **rdfs:Class**;

rdfs:subPropertyOf indica che una proprietà è sottoproprietà di un'altra.

La tripla

P1 rdfs:subPropertyOf P2

indica che **P1** e **P2** sono istanze di **rdf:Property** e che se

S P1 O

allora vale anche

S P2 O

rdfs:subPropertyOf è una istanza di *rdf:Property*, il dominio e l'immagine di **rdfs:subPropertyOf** sono **rdf:Property**;

rdfs:subClassOf è usata per indicare che una classe è sottoclasse di un'altra. La tripla

C1 rdfs:subClassOf C2

indica che **C1** e **C2** sono istanze di **rdfs:Class** e che se

I rdf:type C1

allora

I rdf:type C2

rdfs:subClassOf è una istanza di **rdf:Property**, il dominio e l'immagine di **rdfs:subClassOf** sono **rdfs:Class**;

rdfs:label può essere usato per fornire una forma leggibile di un nome di una certa risorsa. La tripla

R rdfs:label L

indica che **R** è una istanza di **rdfs:Resource** e che **L** è una istanza di **rdfs:Literal** che rappresenta una etichetta leggibile per **R**.

rdfs:label è una istanza di **rdf:Property**, il dominio è **rdfs:Resource**, l'immagine è **rdfs:Literal**;

rdfs:comment può essere usato per fornire una descrizione e documentazione in forma leggibile di una certa risorsa. La tripla

R rdfs:comment L

indica che **R** è una istanza di **rdfs:Resource** e che **L** è una istanza di **rdfs:Literal** che rappresenta una descrizione leggibile per **R**.

rdfs:comment è una istanza di **rdf:Property**, il dominio è **rdfs:Resource**, l'immagine è **rdfs:Literal**.

2.3.3 Altri vocabolari

In questa sezione vengono analizzate altre classi e proprietà come i contenitori e formalizzeremo nella gerarchia dell'*RDFS* concetti già visti in precedenza come la reificazione.

Contenitori

I contenitori sono risorse che rappresentano collezioni, esistono tre classi di contenitori: **rdf:Bag**, **rdf:Seq**, **rdf:Alt**. Sintatticamente e nel modo d'uso non ci sono differenze tra le tre classi, la differenza sta nel significato semantico che gli utenti danno a tali strutture. Per convenzione **rdf:Bag** è usata come lista non ordinata, **rdf:Seq** come lista ordinata nella quale l'ordine è significativo e, infine, **rdf:Alt** indica che il processo tipico presuppone una scelta di un elemento tra i suoi membri.

Le risorse del vocabolario sono:

rdfs:Container è la super classe di tutti i contenitori, **rdf:Bag**, **rdf:Seq** e **rdf:Alt**;

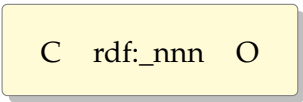
rdfs:member è la super proprietà delle proprietà di appartenenza di un contenitore.

rdfs:member è una istanza di **rdf:Property**. Il dominio e l'immagine di **rdfs:member** sono **rdfs:Resource**;

rdfs:ContainerMembershipProperty è la classe delle proprietà di appartenenza di un contenitore.

rdfs:ContainerMembershipProperty è una istanza di **rdfs:Class** e una sottoclasse di **rdf:Property**;

rdf:_nnn, dove **nnn** è un intero *maggiore di zero* senza zeri in testa, è la proprietà che indica l'appartenenza di una risorsa ad un contenitore. La tripla



C rdf:_nnn O

indica che la risorsa **O** è membro del contenitore **C**.

rdf:_nnn è una istanza di **rdfs:ContainerMembershipProperty** e una sottoproprietà di **rdfs:member**. Notare che, poiché il dominio di **rdfs:member** è **rdfs:Resource** e non **rdfs:Container**, allora **rdf:_nnn** può essere applicata anche a risorse che non sono contenitori;

rdf:Bag è la classe dei contenitori non ordinati. L'ordinamento numerico, dato dal numero **nnn** delle proprietà di appartenenza **rdf:_nnn** applicate ad esso, è da considerarsi privo di significato.

rdf:Bag è una sottoclasse di **rdf:Container**;

rdf:Seq è la classe dei contenitori ordinati, l'ordinamento numerico, dato dal numero **nnn** delle proprietà di appartenenza **rdf:_nnn** applicate ad esso, è da considerarsi significativo.

rdf:Seq è una sottoclasse di **rdf:Container**;

rdf:Alt è la classe dei contenitori che prevedono una selezione tra i suoi membri. Il primo membro, con valore numerico **nnn** minore delle proprietà di appartenenza **rdf:_nnn**, è da considerarsi come il valore selezionato di default.

rdf:Alt è una sottoclasse di **rdf:Container**.

Collezioni

Le collezioni sono liste di elementi, anche vuote, rappresentate in modo simile a come vengono implementate le liste nei linguaggi funzionali. Nella figura 7 è mostrato un uso tipico di queste liste.

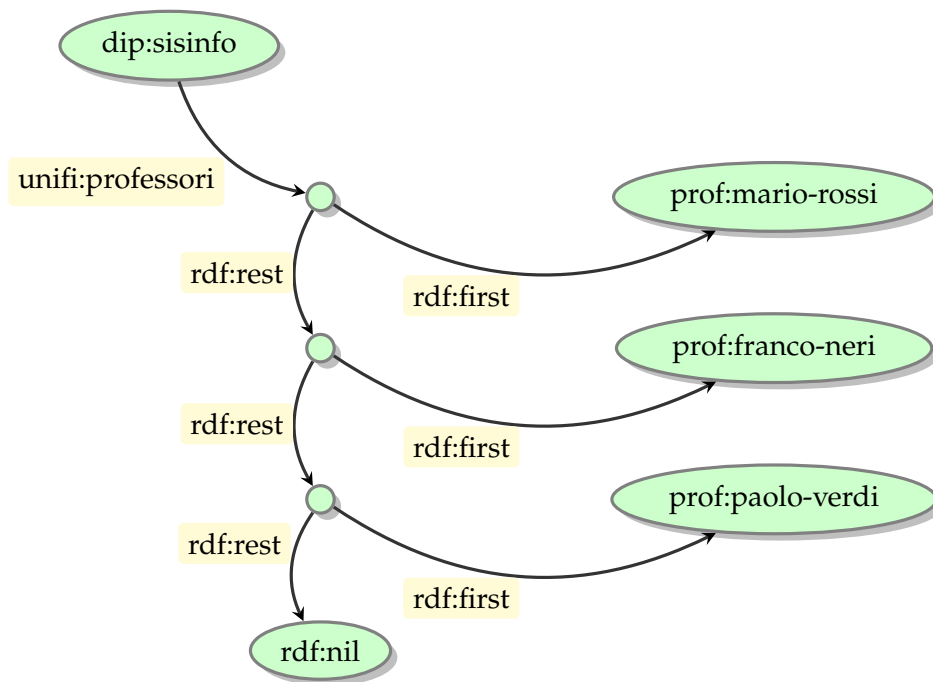


Figura 7: Esempio di utilizzo delle Collection

Ogni *bnode* è istanza di **rdf:List** così come **rdf:nil** che rappresenta una lista vuota. **rdf:first** indica il primo elemento della lista in questione, **rdf:rest** il resto della lista senza il primo elemento.

Nel dettaglio il vocabolario è composto da:

rdf:List è la classe delle liste, le sue istanze rappresentano liste.

rdf:List è una istanza di **rdfs:Class**;

rdf:first indica il primo elemento di una lista. La tripla:

L rdf:first O

indica che **L** è una istanza di **rdf:List**, **O** una istanza di **rdfs:Resource** e che **O** è il primo elemento della lista **L**.

rdf:first è una istanza di **rdf:Property**, il suo dominio è **rdf:List** e la sua immagine **rdfs:Resource**;

rdf:rest indica gli elementi di una lista escluso il primo. La tripla:

L rdf:rest O

indica che **L** è una istanza di **rdf:List**, **O** una istanza di **rdfs:List** e che **O** è una lista uguale ad **L** togliendo il primo elemento.

rdf:rest è una istanza di **rdf:Property**, il suo dominio è **rdf:List** e la sua immagine **rdfs:List**;

rdf:nil indica una lista vuota.

rdf:nil è una istanza di **rdf:List**.

Reificazione


Nella sezione 2.2.4 è già stato introdotto il concetto di reificazione e la sua implementazione. È già state descritte la classe **rdf:Statement** e le proprietà **rdf:subject**, **rdf:predicate** e **rdf:object**; ma non erano ancora noti i concetti di *classe* e *proprietà*.

Viene descritto come formalizzare tali classe e proprietà in *RDFS*:

rdf:Statement è la classe delle sentenze, le sue istanze rappresentano delle triple, ognuna con un soggetto indicato da **rdf:subject**, un predicato indicato da **rdf:predicate** e un oggetto indicato da **rdf:object**.

rdf:Statement è una istanza di **rdfs:Class**;

rdf:subject è la proprietà che indica il soggetto di una tripla (una istanza di **rdf:Statement**). La tripla:

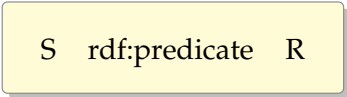


S rdf:subject R

indica che **S** è una istanza di **rdf:Statement**, **R** di **rdfs:Resource** e che **R** è il soggetto di **S**.

rdf:subject è una istanza di **xml:Property**, il suo dominio è **rdf:Statement**, l'immagine **rdf:Resource**;

rdf:predicate è la proprietà che indica il predicato di una tripla (una istanza di **rdf:Statement**). La tripla:



S rdf:predicate R

indica che **S** è una istanza di **rdf:Statement**, **R** di **rdfs:Resource** e che **R** è il predicato di **S**.

rdf:predicate è una istanza di **xml:Property**, il suo dominio è **rdf:Statement**, l'immagine **rdf:Resource**;

rdf:object è la proprietà che indica l'oggetto di una tripla (una istanza di **rdf:Statement**). La tripla:

S rdf:object R

indica che **S** è una istanza di **rdf:Statement**, **R** di **rdfs:Resource** e che **R** è l'oggetto di **S**.

rdf:object è una istanza di **xml:Property**, il suo dominio è **rdf:Statement**, l'immagine **rdf:Resource**.

Proprietà di utility

Le seguenti tre proprietà sono definite per fornire più informazioni riguardo una risorsa:

rdfs:seeAlso serve per indicare una risorsa che fornisce ulteriori informazioni su un'altra risorsa. La tripla:

S rdfs:seeAlso O

indica che **S** e **O** sono istanze di **rdfs:Resource** e che **O** può fornire informazioni addizionali su **S**.

rdfs:seeAlso è una istanza di **rdf:Property**, il suo dominio è **rdfs:Resource**, la sua immagine **rdfs:Resource**;

rdfs:isDefinedBy serve per indicare una risorsa che definisce un'altra risorsa. La tripla:

S rdfs:isDefinedBy O

indica che **S** e **O** sono istanze di **rdfs:Resource** e che **O** definisce **S**.

rdfs:isDefinedBy è una istanza di **rdf:Property**, il suo dominio è **rdfs:Resource**, la sua immagine **rdfs:Resource**;

rdf:value non ha un significato formale, ma di solito viene usata per fornire il valore principale di un dato strutturato. La tripla:

S rdf:value O

indica che **S** e **0** sono istanze di **rdfs:Resource** e che **0** è il valore principale di **S**.

rdf:value è una istanza di **rdf:Property**, il suo dominio è **rdfs:Resource**, la sua immagine **rdfs:Resource**.

2.4 WEB ONTOLOGY LANGUAGE (OWL)

Web Ontology Language estende *RDFS* fornendo un vocabolario addizionale per descrivere altre proprietà e classi. Le strutture descritte con *OWL* prendono il nome di *ontologie*. *OWL* fornisce un suo *namespace*: <http://www.w3.org/2002/07/owl#>, e solitamente viene usato il prefisso **owl**.

Le ontologie includono descrizioni di classi, di proprietà e le loro istanze. La principale potenzialità nel descrivere ontologie rispetto a schemi *RDFS* sta nel fatto che le ontologie sono strutture logiche dalle quali è possibile estrarre conoscenza. Il mezzo con cui viene estratta questa conoscenza da un'ontologia è il *reasoner* che è un programma che fornisce supporto generico applicabile ad ogni ontologia *OWL*.

OWL è composto da tre sottolinguaggi con livelli di espressività logica crescenti:

1. *OWL Lite* fornisce supporto alle necessità di una classificazione gerarchica con vincoli semplici;
2. *OWL DL* fornisce la massima espressività senza perdita di computabilità e decidibilità dei *reasoning*;
3. *OWL Full* fornisce la massima espressività senza garanzie computazionali.

OWL Full implementa una logica del primo ordine, mentre gli altri due implementano delle logiche descrittive più restrittive.

Una analisi avanzata di *OWL* va oltre gli scopi di questo testo, per uno studio approfondito si rimanda a [5].

2.5 SIMPLE KNOWLEDGE ORGANIZATION SYSTEM (SKOS)

SKOS è un vocabolario *RDF* che permette la descrizione di semplici sistemi di organizzazione della conoscenza come tesauroi, tassonomie e schemi di classificazione. Rappresenta un anello tra i linguaggi formali, come *OWL*, e le attuali implementazioni informali delle applicazioni web.

L'elemento fondamentale del vocabolario è il *concept*, *SKOS* mette a disposizione la classe **skos:Concept** per indicare che una certa risorsa è un *concept*. Con la tripla

```
ex:esempioConcept rdf:type skos:Concept
```

si indica che **ex:esempioConcept** è un *concept*.

skos:prefLabel, **skos:altLabel**, **skos:hiddenLabel** servono per indicare l'espressione in linguaggio naturale con cui ci si riferisce ad un *concept*, sono tutte sottoproprietà di **rdfs:label**. **skos:prefLabel** indica l'etichetta principale, mentre con **skos:altLabel** è possibile definire sinonimi. Con **skos:hiddenLabel** è possibile indicare etichette che non devono essere visibili.

2.5.1 Relazioni semantiche

Con SKOS è possibile definire le seguenti relazioni semantiche tra *concept*:

- **skos:broader** e **skos:narrower** definiscono relazioni gerarchiche nelle quali un *concept* è una specificazione di un altro, oppure una sua parte. **skos:broader** e **skos:narrower** sono una l'inversa dell'altra;
- **skos:relate** indica che vi è una associazione non gerarchica tra due *concept*.

La figura 8 indica il modo di utilizzo di **skos:narrower** e **skos:broader**, nell'esempio **animali:gatto** è una specializzazione di **animali:mammifero**.

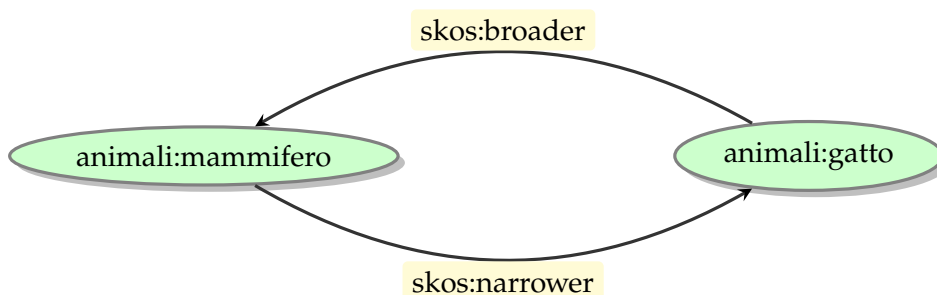


Figura 8: Esempio di utilizzo di **skos:narrower** e **skos:broader**.

2.5.2 Note

SKOS fornisce la proprietà **skos:note** e le sue specializzazioni per indicare descrizioni e documentazioni testuali di un *concept*. **skos:note** è per una documentazione generica, le sue specializzazioni sono:

- **skos:scopeNote** per indicazioni su l'uso di un concept e sui suoi limiti;

- **skos:definition** per una spiegazione completa del significato di un *concept*;
- **skos:example** per fornire esempi sull'uso di un *concept*;
- **skos:historyNote** per descrivere cambi sostanziali nel significato e nella forma di un *concept*;
- **skos:editorialNote** per fornire informazioni indirizzate all'amministrazione, come indicazioni sul lavoro ancora da fare;
- **skos:changeNote** per documentare i cambiamenti di un *concept* a scopo amministrativo.

2.5.3 Concept *schema*

I *concept* possono essere usati come entità a se stanti, oppure possono essere organizzati in vocabolari chiamati schemi, SKOS fornisce la possibilità di rappresentare questi schemi con la classe **skos:ConceptScheme**. Per indicare che un *concept* appartiene ad un certo schema si usa la proprietà **skos:inScheme**. Per indicare i punti di accesso ad una gerarchia di uno schema definita con **skos:narrower** e **skos:broader**, si usa la proprietà di un **skos:ConceptScheme:skos:hasTopConcept**.

In pratica con **skos:narrower** e **skos:broader** si definiscono alberi di *concept*, le radici di questi alberi possono essere associate ad uno schema con **skos:hasTopConcept**.

2.5.4 Mappare concept *schema*

È possibile creare relazioni tra schemi diversi usando le proprietà dei *concept*:

- **skos:exactMatch**;
- **skos:closeMatch**.

Entrambe possono essere usate per indicare che due *concept* appartenenti a schemi diversi hanno lo stesso significato. La differenza tra i due è che **skos:exactMatch** è transitivo, **skos:closeMatch** no.

Due *concept* di due schemi diversi possono essere anche mappati con le proprietà parallele a quelle per le relazioni semantiche:

- **skos:broadMatch**;
- **skos:narrowMatch**;
- **skos:relatedMatch**.

2.5.5 Collezioni di concept

I *concept* possono essere organizzati in collezioni che possono essere ordinate o no. Queste collezioni vengono indicate con le classi:

- **skos:Collection** indica una collezione non ordinata, i *concept* membri vengono indicati con la proprietà **skos:member**;
- **skos:OrderedCollection** indica una collezione non ordinata, i *concept* membri vengono indicati con la proprietà **skos:memberList**;

2.6 friend of a friend (foaf)

FOAF consiste in un vocabolario per definire una rete sociale tramite *RDF*, *RDFS* e *OWL*. Con FOAF è possibile descrivere persone e relazioni tra di esse. FOAF usa il *namespace* <http://xmlns.com/foaf/0.1/> a cui solitamente viene associato il prefisso **foaf:**.

In questo testo vengono analizzate solo le classi e le proprietà stabili delle specifiche, per un elenco completo riferirsi a [7].

FOAF fornisce le seguenti classi:

- **foaf:Agent** è la superclasse di tutti gli agenti che possono essere gruppi, persone o organizzazioni;
- **foaf:Group** rappresenta una collezione di agenti, ed è a sua volta un agente, i suoi membri sono definiti con la proprietà **foaf:member**, è una sottoclasse di **foaf:Agent**;
- **foaf:Organization** rappresenta un agente corrispondente ad una organizzazione, impresa, etc..., è una sottoclasse di **foaf:Agent**;
- **foaf:Person** rappresenta una persona, è una sottoclasse di **foaf:Agent**;
- **foaf:Document** rappresenta un documento che può essere sia elettronico che fisico.

FOAF fornisce le seguenti proprietà:

- **foaf:homepage** indica la pagina web di qualcosa, una certa pagina può essere associata a una risorsa, il dominio è **owl:Thing**, l'immagine **foaf:Document**;
- **foaf:isPrimaryTopicOf** mette in relazione qualcosa con un documento che tratta principalmente di questa, è l'inversa di **foaf:primaryTopic**, il dominio è **owl:Thing**, l'immagine **foaf:Document**;

- **foaf:primaryTopic** mette in relazione un documento con qualcosa che è il suo argomento principale, è l'inversa di **foaf:isPrimaryTopicOf**, il dominio è **foaf:Document**, l'immagine **owl:Thing**;
- **foaf:knows** mette in relazione una persona con un'altra che egli conosce, il dominio è **foaf:Person**, l'immagine **foaf:Person**;
- **foaf:made** mette in relazione un agente con qualcosa fatta da questa, è l'inversa di **foaf:maker**, il dominio è **foaf:Agent**, l'immagine **owl:Thing**;
- **foaf:maker** mette in relazione una cosa con l'agente che l'ha fatta, è l'inversa di **foaf:made**, il dominio è **owl:Thing**, l'immagine **foaf:Agent**;
- **foaf:mbox** mette in relazione un agente con la sua casella di posta, solitamente identificata con lo schema *URI mailto:*, il dominio è **foaf:Agent**, l'immagine **owl:Thing**;
- **foaf:member** indica il membro di un gruppo, il dominio è **foaf:Group**, l'immagine **foaf:Agent**;

La General Architecture for Text Engineering (GATE) [11] è un framework JAVA Open Source per lo sviluppo di software per l'analisi del linguaggio naturale. È stato creato dall'Università di Sheffield a partire dal 1995 ed è attualmente aggiornato anche da altre organizzazioni e istituti, quali, ad esempio Ontotext, un'azienda bulgara che sviluppa software principalmente nell'ambito Semantic Web. Oggi è uno strumento molto utilizzato dalla comunità scientifica per la soluzione di vari problemi nell'ambito del Natural Language Processing (NLP) quali, ad esempio, l'Information Extraction, Information Retrieval, Taxonomy Learning, e così via.

GATE fornisce una serie di plugin che eseguono alcuni tipi di analisi sul testo, nonché una API per la creazione di plugin ad hoc. Tali plugin possono essere organizzati e utilizzati secondo una pipeline specifica per il task da svolgere, permettendo così la massima flessibilità.

GATE fornisce anche un'interfaccia grafica, GATE Developer, che permette di creare e cambiare agevolmente i componenti di una pipeline, garantendo un'ottima configurabilità e possibilità di customizzazione. GATE fornisce anche una API, GATE Embedded, che permette di creare applicazioni stand alone che non necessitano di usare l'interfaccia grafica.

3.1 CONCETTI GENERALI

Questo paragrafo descrive quali sono i concetti alla base del funzionamento del framework GATE. Ogni applicazione GATE è costituita da plugin, chiamati plugin CREOLE, che sono eseguiti in cascata. Ciascun plugin aggiunge, rimuove o modifica annotazioni ai documenti. I documenti, insieme a queste annotazioni, possono essere salvati in un datastore che ne garantisce la persistenza.

3.1.1 *Collection of REusable Objects for Language Engineering (CREOLE)*

L'architettura di GATE è basata su componenti, cioè elementi di software riutilizzabile con interfacce ben definite, che possono essere installati e utilizzati in contesti molto diversi fra loro. Sono implementati tramite una classe Java, che al suo interno può fornire accesso a un database, chiamare un eseguibile esterno o semplicemente implementare l'intero componente in modo autonomo.

I componenti di GATE possono essere di tre tipi:

- Language Resource (LR): rappresentano le entità quali i corpora, i documenti, le ontologie e così via.
- Processing Resource (PR): rappresentano entità principalmente algoritmiche che eseguono operazioni sul testo. Alcuni esempi sono i parser, i tagger e così via.
- Visual Resource (VR): sono i componenti che servono per la visualizzazione e l'editing di altre entità nell'interfaccia grafica.

L'insieme composto da tutti i componenti integrati in GATE è chiamato Collection of REusable Objects for Language Engineering (CREOLE).

I componenti possono essere organizzati in una pipeline di GATE. GATE fornisce vari tipi di pipeline: alcune lavorano su un singolo documento, altre a livello di corpus, cioè a gruppi di documenti, prendendoli uno alla volta. Si può condizionare l'esecuzione delle pipeline secondo determinate condizioni sulle features del documento o interromperla dopo un determinato tempo di timeout.

3.1.2 *Features e Annotazioni*

Le features e le annotazioni sono i due elementi utilizzati da GATE per associare informazioni aggiuntive ai documenti.

Le features sono coppie nella forma chiave=valore, dove la chiave è una stringa, mentre il valore può essere un qualunque oggetto JAVA.

Una *feature map* è una lista di features e può essere associata a un documento, oppure a un'annotazione. Ad esempio la *feature map* di un documento contiene informazioni come il nome del file, il percorso, il mimetype, e così via.

Le annotazioni sono l'elemento che, nella maggior parte dei casi, viene utilizzato dai componenti CREOLE per eseguire le loro operazioni. Ciascuna PR, infatti, analizza il testo di un documento insieme alle sue annotazioni e, eventualmente, ne aggiunge alcune nuove o modifica quelle esistenti.

Le annotazioni hanno una *feature map*, un tipo e un ID univoco. Vanno da uno *start node* a un *end node*. Lo *start node* e l'*end node* sono punti nel testo e sono identificati dal loro offset dall'inizio del documento.

Le annotazioni sono organizzate in *annotation set*, cioè insiemi di annotazioni. Un *annotation set* può anche essere visto come un grafo direzionato e aciclico di cui ogni annotazione rappresenta un arco.

Ogni documento può avere uno o più *annotation set*. Ad esempio ogni documento ha sempre un *annotation set* di default, inoltre alcuni documenti possono avere un altro *annotation set*, chiamato **Original Markup** che contiene alcune informazioni estratte del documento in fase di caricamento.

3.2 FUNZIONI E PLUGINS

In questo paragrafo sono descritte alcune delle funzionalità e dei plugin distribuiti insieme a GATE. Nel prossimo paragrafo sono descritti i *gazetteer* che permettono di individuare nel testo parole o termini composti da più parole appartenenti a liste predefinite. Nel paragrafo 3.2.2 sono descritte le grammatiche JAPE che permettono di individuare dei pattern nelle annotazioni dei documenti utilizzando le espressioni regolari. Sulle annotazioni individuate tramite il matching basato su espressioni regolari è poi possibile effettuare modifiche o aggiunte. Il paragrafo 3.2.3 descrive la pipeline ANNIE e i suoi componenti. ANNIE è una pipeline completa per l'Information Extraction su testi in lingua inglese, ma i suoi componenti possono essere riutilizzati anche in altre pipeline.

3.2.1 *Gazetteer*

Un *gazetteer* è una insieme di liste, ciascuna contenente un insieme di nomi. Tali insiemi di nomi possono essere ad esempio nomi di città, nomi di organizzazioni, giorni della settimana, ecc. I *gazetteer* plugin sono usati per individuare nel testo occorrenze delle parole appartenenti a queste liste.

In GATE il termine *gazetteer* viene impiegato per indicare sia l'insieme di liste, sia i plugin CREOLE che gestiscono le liste e individuano le occorrenze nel testo.

GATE mette a disposizione molti plugin di tipo *gazetteer*, ciascuno dei quali crea, per ogni occorrenza individuata, un'annotazione di tipo Lookup.

3.2.2 *Java Annotation Patterns Engine*

Java Annotation Patterns Engine (JAPE) è una versione di CPSL (Common Pattern Specification Language)[12] e permette di individuare pattern nelle annotazioni dei documenti utilizzando espressioni regolari.

Le annotazioni dei documenti rappresentano un grafo, come già spiegato nel paragrafo 3.1.2. Le espressioni regolari, però, hanno una espressività più limitata e non possono rappresentare i grafi. Questo fa sì che in alcuni casi il matching effettuato con una grammatica JAPE potrebbe essere non prevedibile, dato che, nei casi in cui è necessario un potere espressivo superiore a quello offerto dalla grammatica, JAPE sceglie una delle alternative arbitrariamente.

Questo non è però problematico come potrebbe sembrare, dato che la maggior parte dei dati in GATE è molto meno complessa di un grafo e può essere modellata come una semplice sequenza di informazioni.

Per l'esecuzione di grammatiche JAPE, GATE mette a disposizione il plugin JAPE Transducer. Una grammatica JAPE consiste in un insieme di fasi, ciascuna

delle quali è composta da una coppia pattern-azione. Le fasi vengono eseguite in cascata una di seguito all'altra.

Ogni fase è costituita da una Left Hand Side (LHS), che è la descrizione del pattern da individuare e da una Right Hand Side (RHS) che contiene istruzioni per manipolare le annotazioni.

Le annotazioni che corrispondono al pattern definito dalla LHS sono manipolate secondo le istruzioni della RHS.

Il codice 21 mostra una regola di esempio. Il simbolo `-->` separa la LHS

```
Phase: Jobtitle
Input: Lookup
Options: control = appelt debug = true
Rule: Jobtitle1
(
    {Lookup.majorType == jobtitle}
    (
        {Lookup.majorType == jobtitle}
    )?
)
:jobtitle
-->
:jobtitle.JobTitle = {rule = 'JobTitle1'}
```

Codice 21: Esempio di regola JAPE

dalla RHS.

La regola di esempio individua tutte le annotazioni di tipo *Lookup* con *major type* "jobtitle", seguite opzionalmente da un'altra annotazione *Lookup* con *major type* "jobtitle". La RHS annota il testo individuato con un'annotazione di tipo *JobTitle* che contiene una *feature* "rule=JobTitle1".

Il matching nella LHS può essere fatto usando vari tipi di operatori che permettono la massima flessibilità. Fra gli altri è possibile usare anche gli operatori di Kleene, operatori di espressioni regolari su stringhe, operatori di confronto, operatori di contesto, che permettono di individuare annotazioni contenute in altre annotazioni, ecc.

Nella RHS le istruzioni possono essere scritte secondo la sintassi JAPE, come nell'esempio precedente, oppure utilizzando codice Java.

3.2.3 *A Nearly New Information Extraction System*

GATE viene distribuito insieme a un sistema di Information Extraction, chiamato *A Nearly New Information Extraction System (ANNIE)* [13] sviluppato principalmente per l'inglese e che si basa su algoritmi a stati finiti e sul linguaggio JAPE (vedi paragrafo 3.2.2).

I componenti di ANNIE formano una pipeline, i cui componenti vengono descritti di seguito

Document Reset

La Document Reset PR è un plugin che permette di rimuovere alcuni *annotation set* dal documento sul quale viene eseguito, lasciando all'utente la possibilità di scegliere quali mantenere. Di default si utilizza all'inizio della pipeline per togliere tutti gli *annotation set* tranne gli Original Markups, prima di ricrearli e ripopolarli con l'uso degli altri plugin CREOLE.

Tokeniser

Il Tokeniser divide il testo in piccole unità, chiamate Token, che rappresentano le parole, i numeri o la punteggiatura. Le annotazioni di tipo Token create contengono alcune features che le descrivono.

Tutti i Token contengono una feature *kind* che indica il tipo di token e che può avere i seguenti valori:

- **word**: il token è composto da un insieme contiguo di lettere maiuscole o minuscole senza segni di punteggiatura se non un trattino. I token di tipo "word" contengono anche una feature *orth* che contiene informazioni ortografiche che indicano la distribuzione delle lettere maiuscole e minuscole nel token.
- **number**: il token è una sequenza di cifre.
- **symbol**: il token contiene una sequenza di simboli.
- **punctuation**: il token è un simbolo di punteggiatura. Può essere uno *start punctuation* (ad es. '('), un *end punctuation* (ad es. ')'), oppure un tipo di punteggiatura generico.

Il Tokeniser crea anche un altro tipo di annotazione chiamato SpaceToken, che rappresenta tutti gli spazi e tutti i caratteri di controllo come i line feed o i carriage return.

ANNIE include, oltre al Tokeniser generico appena descritto, un English Tokeniser costituito dal Tokeniser generico e da un insieme di regole per adattare l'output generico al POS Tagger per l'inglese

Gazetteer

Il Gazetteer, come spiegato nel paragrafo 3.2.1, ha lo scopo di individuare nel testo parole specificate in una lista. Tale lista è un file di testo in cui a ogni riga corrisponde una parola.

I file di testo che contengono le liste sono individuati da un file di indice (*lists.def*) che indica anche per ogni lista un *minor type* e un *major type*: ogni match individuato sarà annotato con un'annotazione di tipo Lookup che conterrà anche in corrispondenti *minor type* e *major type* come features.

RegEX Sentence Splitter

Il RegEx Sentence Splitter è un plugin basato sulle espressioni regolari che individua le frasi nel testo e le annota con un'annotazione di tipo Sentence. La punteggiatura di fine frase (come un punto) viene annotata con una annotazione di tipo Split.

Part Of Speech Tagger

Il POS Tagger è una versione modificata del Brill Tagger e aggiunge alle annotazioni di tipo Token una feature *category* che indica la parte del discorso del Token (ad esempio se è un verbo, un nome, un aggettivo, e così via).

Il POS Tagger fa uso di un insieme di regole e di un dizionario costruiti a partire da un sottoinsieme di articoli del Wall Street Journal.

Tagger Framework

Il Tagger Framework è un plugin inizialmente sviluppato per integrare con GATE il TreeTagger, ma che poi è stato trasformato in un generico wrapper che supporta molti tagger diversi. Fra i tagger supportati ci sono GENIA (un tagger biomedico), Hunpos (che supporta l'inglese e l'ungherese), TreeTagger (che supporta il tedesco, il francese, lo spagnolo, l'italiano e l'inglese), e lo Stanford Tagger (che supporta l'inglese, il tedesco e l'arabo).

Il plugin si basa su alcune semplici ipotesi sul funzionamento del tagger esterno. La prima ipotesi è che il tagger deve leggere un file esterno in cui ogni riga corrisponde a un'annotazione (ad esempio una riga per ogni frase o una riga per ogni token). La seconda ipotesi è che il tagger deve stampare il suo output sullo standard output, anch'esso con un'annotazione per riga.

Parte II

ANALISI E LAVORO EFFETTUATO

ANALISI DEL CRAWLER

In questo capitolo viene analizzato nel dettaglio il lavoro svolto all'interno del progetto *OSIM* dove è stato svolto il lavoro, ossia lo studio e la realizzazione di un servizio di crawling per l'individuazione e l'estrazione delle competenze del personale universitario dal sito web dell'ateneo.

4.1 IL PROGETTO *osim*

Il progetto *OSIM*, *Open Space Innovative Mind*, ha l'obiettivo di creare una base di conoscenza dell'Università di Firenze che contenga tutte le informazioni necessarie per identificare le competenze in termini di ricercatori, docenti e gruppi.

In figura 9 è possibile vedere uno schema della struttura globale del progetto. La parte di *crawling* visibile a sinistra è quella che si occupa di fare *data ingestion*

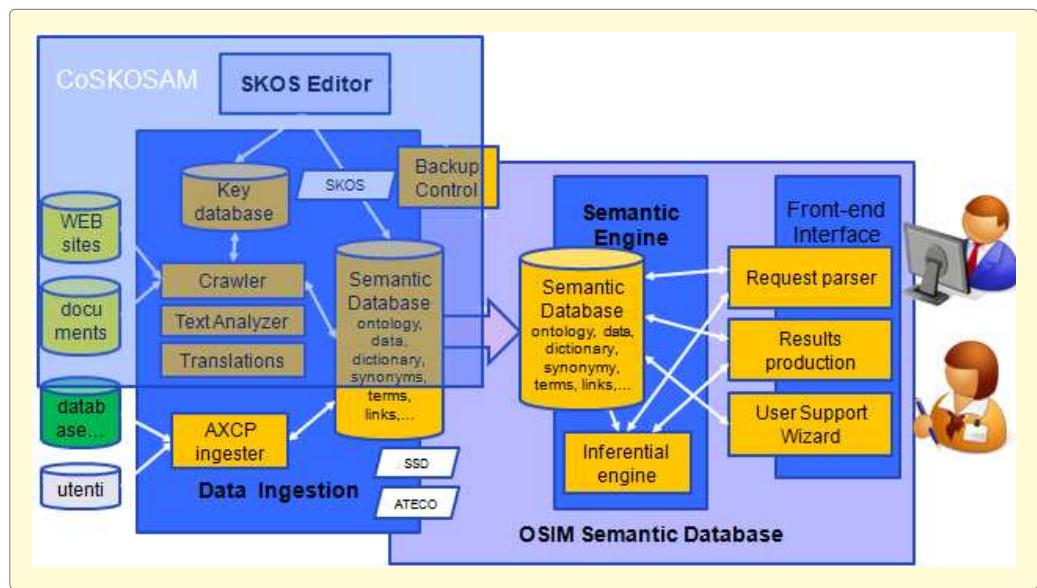


Figura 9: Schema generale del progetto *OSIM*.

della struttura dell'ateneo in termini di corsi, dipartimenti, facoltà, personale strutturato e delle competenze attribuibili ad ognuno di questi. Il processo di *crawling* estrae informazioni dalle pagine web dell'università ed immagazzina questa conoscenza in un *database* semantico.

Questa base di dati semantica deve essere strutturata per poter essere interrogata dagli utenti che desiderano cercare personale con competenze specifiche.

Il lavoro svolto e presentato nel seguente testo è quello di analizzare e sviluppare un *crawler* delle competenze del personale realizzato in *Java* e che si interfaccia con l'ontologia della base di dati semantica di *OSIM*. Il *crawling* viene gestito da un pannello di controllo realizzato con una pagina *JSP* che si interfaccia con una *Servlet Java*.

4.2 INTRODUZIONE

Il crawler è progettato in due passaggi. Durante il primo vengono estratte le parole chiave; successivamente un esperto di dominio deve effettuare una selezione di tali parole, scegliendo se inserirle in un gazetteer o in una black list. Il gazetteer scelto viene usato nel secondo passaggio per individuare le competenze di ciascuna persona, al termine di questa fase l'esperto di dominio può organizzare tali competenze in una ontologia definendo relazioni di tipo gerarchico. L'ontologia ottenuta può essere scaricata in formato *3-store* o *rdf/xml*. Analizziamo nel dettaglio le azioni descritte precedentemente:

1. estrazione delle keyword;
2. creazione del gazetteer;
3. estrazione delle competenze;
4. organizzazione delle competenze.

I punti 1 e 3 sono automatici, i punti 2 e 4 necessitano dell'intervento dell'esperto

4.2.1 Estrazione delle keyword

Il processo parte dalla pagina della lista del personale afferente ad un certo dipartimento (l'interfaccia utente, e quindi anche il crawler, sono divise per dipartimenti) e viene lanciato tramite il pulsante *launch keyword extraction task*. Appoggiandosi al framework *GATE* si effettua un primo passaggio di segnatura della pagina usando un *gazetteer* con la lista precompilata del personale. In seguito tramite delle regole *JAPE* vengono ricavati per ciascun nominativo il link alla rispettiva pagina. Ogni pagina viene poi processata

Per ogni persona in modo analogo si ricava il link alla pagina di *Penelope* che è quella che effettivamente ha le informazioni utili. Quest'ultima pagina viene aggiunta ad un elenco di pagine da processare, si ricavano i link alle pagine degli eventuali corsi, e si aggiungono questi ultimi allo stesso elenco. Una volta

ricavato l'elenco di tutte le pagine utili del personale e dei corsi si usa *GATE* per estrarre da queste le frasi che contengono. Non è presente l'associazione keyword-persona, visto che interessa solo avere un elenco di keyword. Queste frasi vengono salvate in *file* temporanei.

GATE permette di riconoscere la lingua, quindi ogni frase viene concatenata ad un *file* di frasi in lingua italiana o ad uno in lingua inglese. Da questi due *file* vengono estratti i sostantivi sempre usando *GATE* e vengono tradotti usando le *api* di *Google translate*. Viene inserito un record contenente una tripletta <keyword in italiano, keyword in inglese, stato> in una tabella di un database *Mysql*. Il significato di *stato* è quello di indicare nel passaggio successivo se la keyword è da aggiungere al *gazetteer*, alla *black list* o a nessun dei due.

4.2.2 Creazione del *gazetteer*

In questa fase l'esperto di dominio deve intervenire scegliendo le *keyword* che ritiene più rilevanti, inserendole in un *gazetteer*, e scartando quelle meno significative inserendole eventualmente in una *black list*. Tutte le operazioni sono effettuate attraverso una interfaccia *user-friendly* che permette di creare facilmente il *gazetteer*. Il procedimento è semplice e intuitivo e non necessita di ulteriori delucidazioni.

4.2.3 Estrazione delle competenze

Questa fase viene avviata cliccando sul pulsante *launch competence crawler*. Ripartendo dall'elenco del personale del dipartimento vengono ricavati i *link* alla pagina della persona, quella alla pagina di *penelope*, e quelli dei corsi, in modo simile a quello visto nella sezione 4.2.1. In seguito viene popolata l'ontologia di dominio dell'ateneo indicando per ogni persona l'afferenza al dipartimento e i corsi svolti. Dopodiché usando *GATE* vengono segnate le keyword scelte nel passaggio precedente, e con delle regole *JAPE* si individuano le competenze composte da più *keyword*. Il procedimento viene effettuato sulla pagina della persona e su quelle dei corsi. Queste competenze (anche solo le keyword non composte), individuate nelle pagine della persona, vengono aggiunte all'ontologia delle competenze, ovviamente con la relazione che le collega alla persona. Per lo *storage* dell'ontologia viene usato il *framework sesame*.

4.2.4 Organizzazione delle competenze

L'ontologia delle competenze può essere organizzata da un esperto come una tassonomia arricchita con la relazione semantica tra competenze *related*. Vengono mostrate due colonne; sulla sinistra c'è l'elenco delle competenze estratte nel passo precedente, queste possono essere trascinate una ad una nella colon-

na di destra e organizzate in modo gerarchico ad albero, ogni volta che viene spostata una competenza l'ontologia viene aggiornata aggiungendo la relazione scelta.

4.3 ANALISI DETTAGLIATA

Adesso verranno descritti nel dettaglio i diversi passaggi del crawler. Come già detto ci sono due funzionalità principali: Il *crawling delle keyword* e il *crawling delle competenze*.

In figura 10 è possibile vedere la schermata iniziale che appare subito dopo

The screenshot displays the 'Open Space Innovative Mind' web application. At the top, there is a header with the logo 'OSIM' and 'Open Space Innovative Mind' text, along with a 'BETA' badge and the 'Università degli Studi di Firenze' logo. Below the header is a navigation bar with links: Home, Documentation, Search, Managing Knowledge, Browsing People & Publications, and Contact DISIT. The main content area shows a login status 'Benvenuto root' and a 'Logout' link. The title 'Open Space Innovative Mind' is prominently displayed. Below the title, a summary line indicates 'Total keywords: 92329 - Total keywords in gazetteers: 5233'. A table lists five departments, each with its name, last extraction details for keywords and competencies, and a summary of gazetteer statistics (Persons and Documents). To the right of each department row, there are statistics for 'sessioni in corso (R)' and 'sessioni in corso (W)', both showing 0, and two action links: 'leggi' (read) and 'amministra' (administrate), each accompanied by a magnifying glass icon.

Dipartimenti	Utenti	Azione
Dipartimento di Area Critica Medico Chirurgica	Ultima estrazione keywords: 7:0:37 pm, 15/6/2012 - Esito: OK Ultima estrazione competenze: 6:0:7 pm, 17/6/2012 - Esito: OK Gazetteer: 15311531 - Persone: 90 di 127 - Documenti: 722	sessioni in corso (R): 0 sessioni in corso (W): 0 leggi amministra
Dipartimento di Anatomia, Istologia e Medicina Legale	Ultima estrazione keywords: 8:10:39 pm, 15/6/2012 - Esito: OK Ultima estrazione competenze: unknown - Esito: unknown Gazetteer: 10011001 - Persone: 26 di 36 - Documenti: 400	sessioni in corso (R): 0 sessioni in corso (W): 0 leggi amministra
Dipartimento di Architettura - Disegno, Storia, Progetto	Ultima estrazione keywords: 9:46:19 pm, 15/6/2012 - Esito: OK Ultima estrazione competenze: unknown - Esito: unknown Gazetteer: 161837 - Persone: 48 di 56 - Documenti: 245	sessioni in corso (R): 0 sessioni in corso (W): 0 leggi amministra
Dipartimento di Biotecnologie Agrarie	Ultima estrazione keywords: 11:31:1 pm, 15/6/2012 - Esito: OK Ultima estrazione competenze: unknown - Esito: unknown Gazetteer: 645645 - Persone: 52 di 70 - Documenti: 293	sessioni in corso (R): 0 sessioni in corso (W): 0 leggi amministra
Dipartimento di Biologia Evoluzionistica Leo Pardi	Ultima estrazione keywords: 0:48:53 am, 16/6/2012 - Esito: OK Ultima estrazione competenze: unknown - Esito: unknown Gazetteer: 111 - Persone: 44 di 75 - Documenti: 238	sessioni in corso (R): 0 sessioni in corso (W): 0 leggi amministra

Figura 10: Screenshot della schermata iniziale.

aver fatto il login. Questa pagina è costruita dalla *JSP index.jsp* e presenta una lista di dipartimenti, con delle statistiche sulle varie fasi di *crawling* effettuate. Per ogni dipartimento sono presenti anche due *link*, uno chiamato "leggi" e, se l'utente corrente ha i permessi per il dipartimento in questione, un altro chiamato "amministra". Entrambi i *link* portano alla stessa pagina di amministrazione del dipartimento. Il primo permette solo di compiere operazioni di lettura delle *keyword* e delle competenze. Il secondo fornisce i diritti per amministrare tutto il dipartimento.

La pagina di amministrazione del dipartimento è creata dalla *JSP userPage.jsp*, e per il layout viene usato il plugin *jQueryUI Layout*¹. Tale pagina presenta due *tabs* denominate:

1. "Ontology manager";
2. "Keys selection".

La prima *tab* visibile in figura 11 presenta una serie di pulsanti in alto che

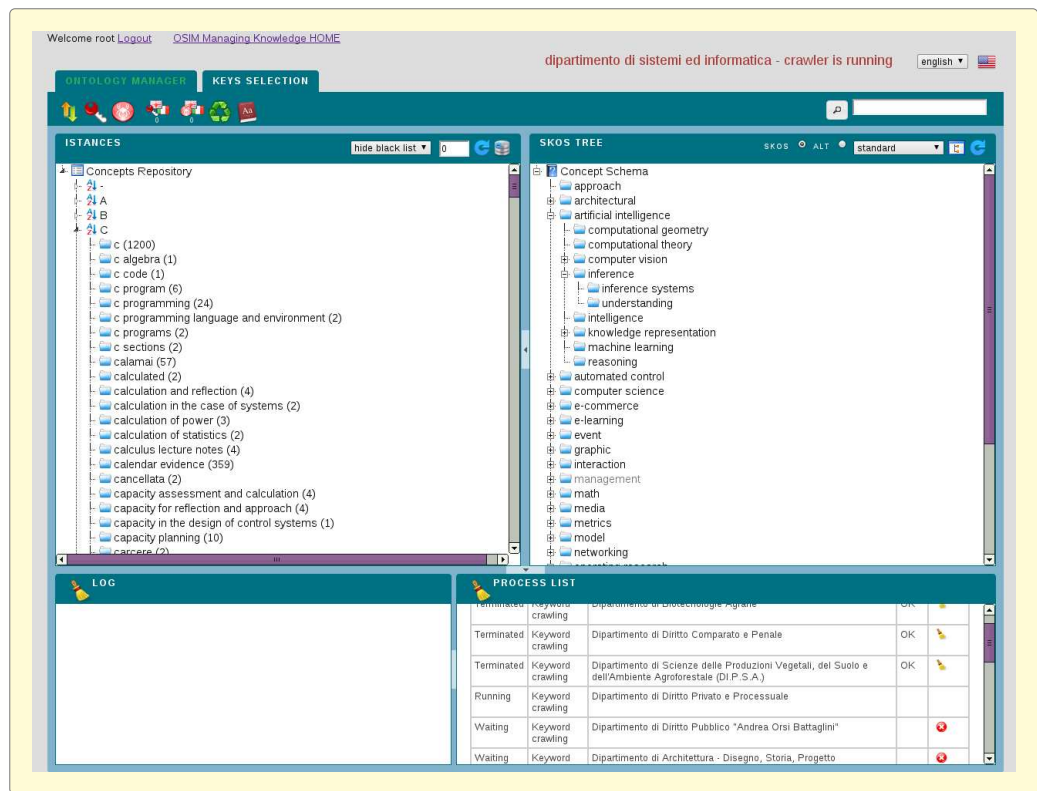


Figura 11: Screenshot del pannello di amministrazione, prima *tab*.

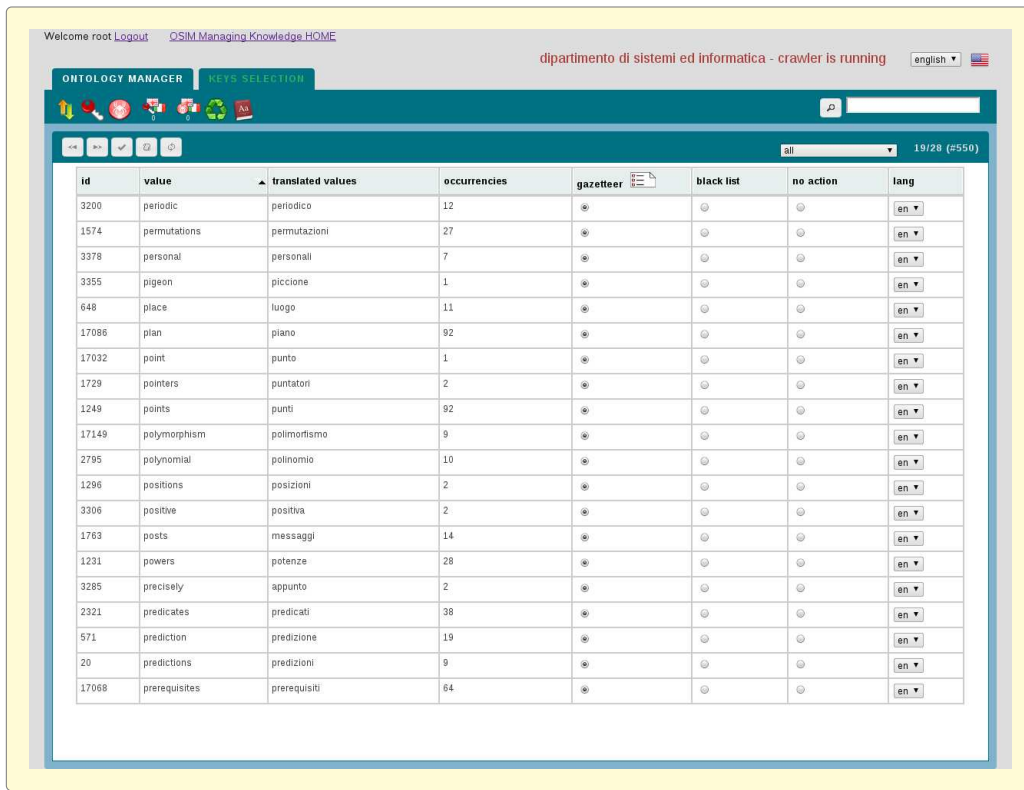
servono per poter lanciare le varie fasi e funzioni del *crawler*, inoltre è divisa in quattro parti. In alto a sinistra è presente una lista di competenze organizzate in ordine alfabetico, queste sono il risultato della seconda fase di *crawling* di competenze. Tali competenze possono essere trascinate nel *quadrante* in alto a destra dove vi è la possibilità di organizzarle in una ontologia delle competenze organizzata tramite *SKOS*² in modo gerarchico. Allo *SKOS* è possibile anche aggiungere elementi nuovi, non risultanti dal *crawling*.

¹ Vedere [10].

² Vedere [6] e sezione 5.1.

Il *quadrante* in basso a sinistra costituisce un'area dove vengono visualizzati tutti i messaggi lanciati dai processi del dipartimento in questione. Il *quadrante* in basso a destra è costituito dalla lista di processi in coda, già eseguiti, e quello correntemente in esecuzione. Dalla coda è possibile eliminare i processi ancora da eseguire.

La seconda *tab* del pannello di amministrazione, visibile in figura 12 è



The screenshot shows a web interface for an ontology manager. At the top, there's a navigation bar with 'Welcome root Logout' and 'OSIM Managing Knowledge HOME'. Below this, a status bar indicates 'dipartimento di sistemi ed informatica - crawler is running' and a language selector set to 'english'. The main content area is titled 'ONTOLOGY MANAGER' and 'KEYS SELECTION'. It features a table with 8 columns: 'id', 'value', 'translated values', 'occurencies', 'gazetteer', 'black list', 'no action', and 'lang'. The table contains 20 rows of keyword data. Each row has a unique 'id', a 'value' in English, its 'translated values' in Italian, the number of 'occurencies', and checkboxes for 'gazetteer', 'black list', and 'no action'. A 'lang' dropdown menu is at the end of each row. The interface also includes a search bar and a filter dropdown set to 'all'.

id	value	translated values	occurencies	gazetteer	black list	no action	lang
3200	periodic	periodico	12	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	en ▼
1574	permutations	permutazioni	27	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	en ▼
3378	personal	personali	7	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	en ▼
3355	pigeon	piccione	1	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	en ▼
648	place	luogo	11	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	en ▼
17086	plan	piano	92	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	en ▼
17032	point	punto	1	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	en ▼
1729	pointers	puntatori	2	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	en ▼
1249	points	punti	92	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	en ▼
17149	polymorphism	polimorfismo	9	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	en ▼
2795	polynomial	polinomio	10	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	en ▼
1296	positions	posizioni	2	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	en ▼
3306	positive	positiva	2	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	en ▼
1763	posts	messaggi	14	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	en ▼
1231	powers	potenze	28	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	en ▼
3285	precisely	appunto	2	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	en ▼
2321	predicates	predicati	38	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	en ▼
571	prediction	predizione	19	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	en ▼
20	predictions	predizioni	9	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	en ▼
17068	prerequisites	prerequisiti	64	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	en ▼

Figura 12: Screenshot del pannello di amministrazione, seconda *tab*.

costituita da una tabella che rappresenta tutte le *keyword* estratte alla fine dell'esecuzione della prima fase di *crawling* per quel dipartimento.

Anche in questa *tab* vi sono replicati i pulsanti per lanciare le due fasi di *crawling*, e la tabella è costituita da una serie di righe dove per ognuna è presente il valore della *keyword* nel linguaggio corrente, il valore della *keyword* negli altri linguaggi (attualmente esistono due linguaggi), il numero di occorrenze di tale *keyword*, e lo stato attuale della *keyword*: se è stata scelta per far parte del *gazetteer*, se è in *black list*, o se non è ancora stato deciso niente su questa. Il campo *id* rappresenta un identificatore univoco della *keyword*, mentre il campo *lang* serve per poter cambiare lingua ad una *keyword* che è stata classificata erroneamente. È possibile ordinare la visualizzazione delle *keyword* per occorrenza sia in ordine crescente che decrescente.

Il blocco dei bottoni, di cui è possibile vederne un ingrandimento in figura 13 è costituito da una serie di pulsanti che, in ordine da sinistra a destra, svolgono



Figura 13: Dettaglio del blocco dei pulsanti

le seguenti funzioni:

download dell'*RDF Store* permette di scaricare un file in uno dei due formati *N-Triples* o *RDF/XML*³ contenente tutto lo *store* semantico. Viene scaricato sempre lo *store* di tutti i dipartimenti, indipendentemente da dove viene richiesto;

lancia il *crawling* delle *keyword* per il dipartimento corrente, vedere la sezione 4.3.1;

lancia il *crawling* delle competenze per il dipartimento corrente, vedere la sezione 4.3.3;

lancia il processo di traduzione delle *keyword* che non sono state tradotte, per qualche eventuale problema, nel corso della fase di *crawling* delle *keyword*. Sotto al pulsante è presente un contatore che indica in tempo reale il numero di *keyword* non tradotte;

lancia il processo di traduzione delle competenze che non sono state tradotte, per qualche eventuale problema, nel corso della fase di *crawling* delle competenze. Sotto al pulsante è presente un contatore che indica in tempo reale il numero di competenze non tradotte;

cancella la tabella delle *keyword* che non fanno parte del *gazetteer*;

cancella le *keyword* che compaiono nel *gazetteer* delle *stop words*. Questa funzione è utile quando si vuole aggiornare la lista di *stop words* evitando di rilanciare il processo di estrazione delle *keyword*;

lancia il processo di pulizia dell'*RDF* che si occupa di eliminare eventuali triple ridondanti o altri casi particolari che si possono verificare nel corso dei processi di *crawling* e traduzione.

4.3.1 *Crawling* delle *keyword*

Il *crawling* delle *keyword* viene lanciato dal pannello di amministrazione di un dipartimento cliccando sul pulsante corrispondente visibile in figura 14 che

³ Vedere la sezione 2.2.3 per i dettagli sulle serializzazioni.

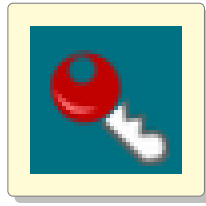


Figura 14: Pulsante per lanciare il *crawling* delle keyword.

tramite una chiamata *Javascript* alla *servolet* esegue il metodo **execute** del gestore di questa operazione **OSIM_RequestKBCommand**⁴. Tale metodo, che è il solito che gestisce la richiesta di *crawling* di competenze, aggiunge un nuovo processo alla coda dei processi⁵ e, quando arriva il turno di tale processo, viene eseguito un thread dell'oggetto *runnable* descritto da **CompetenceKeyCrawler**⁶ che è la stessa classe del thread lanciato nella fase di *crawling* delle competenze.

Tale thread verifica che l'operazione richiesta sia di *crawling* di *keyword* e quindi istanzia la classe **KeywordExtractionEngine**⁷, e lancia il suo metodo **extractKey**.

Questa fase comincia da una pagina del portale "*cercachi*" con la lista del personale del dipartimento⁸ di cui vogliamo prendere le *keyword*. Tale pagina è possibile reperirla all'interno della tabella **departments** del database⁹, dove ci sono memorizzate tutte le pagine di partenza.

In figura 15 è possibile vedere lo schema del funzionamento della prima fase. Lo schema indica il flusso logico delle operazioni che vengono svolte dal processo. Il crawler usa quattro *pipe* di GATE diverse, che saranno spiegate in dettaglio nella sezione 5.2.

In questa prima fase il crawler si preoccupa preventivamente di reperire tutti i *link* alle pagine utili salvandoli in un vettore, in seguito usando GATE individua tutte le frasi presenti su tutte le pagine puntate dai *link* e salva ogni frase in un file temporaneo. Su questi *file* viene usato nuovamente GATE per individuare la lingua di ogni *file* e in seguito le frasi vengono accodate in un singolo *file* di frasi in italiano, uno in inglese, e uno di lingua sconosciuta. Quello di lingua sconosciuta viene in seguito ignorato.

Su questi due *file* viene usato GATE per individuare tutti i *sostantivi* delle frasi che non compaiano nella lista delle *stop words*, e per ognuno di essi viene calcolato il numero di occorrenze. Questi sostantivi, che rappresentano il possibile elenco di *keyword*, vengono salvati in una tabella temporanea che ha le tre

⁴ Vedere la sezione 5.3 per i dettagli sul funzionamento di tali chiamate.

⁵ Vedere la sezione 5.4.1.

⁶ Vedere sezione 5.7.

⁷ Vedi nota 6.

⁸ Ad esempio <http://www.unifi.it/cercachi/show.php?f=s&codice=051400&fonte=informatica> per il dipartimento di informatica e sistemi.

⁹ Vedere sezione 5.6.

Figura 15: Schema della fase di *crawling* delle *keyword*

colonne:

- keyword;
- lingua;
- occorrenze.

Per ogni keyword di questa tabella viene calcolata la traduzione nell'altra lingua, avvalendosi del servizio di *Google Translate*. Prima di accedere al servizio esterno, viene interrogata una cache interna composta da righe nella forma:

- valore italiano;
- valore inglese.

che viene aggiornata ogni volta che è necessario tradurre un valore non presente.

Tutti i valori delle *keyword*, le relative traduzioni, e le occorrenze nelle pagine, vengono memorizzati nella tabella del database relativa alle *keyword* del dipartimento per il quale è stata lanciata questa fase. Oltre ai valori detti viene memorizzato per ogni *keyword* un valore *status* inizializzato a zero. Questo status è il valore che l'esperto modifica quando va a scegliere le *keyword* da candidare per la seconda fase, o quelle da mettere in *black list*.

Vengono adesso analizzate nel dettaglio le varie parti di questa fase.

Estrazione link

In questo punto si sfrutta *GATE*, e in particolare le annotazioni in base ai *gazetteer*, per poter definire un sottoinsieme di persone su cui può essere effettuato il *crawling*. Vengono ricavati tutti i *link* alle pagine delle persone che sono nella *lista del personale di un dipartimento* e che sono anche nel *gazetteer*. Questi link di primo livello non sono utili allo scopo del crawler in quanto le pagine a cui puntano non contengono informazioni utili¹⁰, quindi per ognuna di queste pagine vengono riutilizzate delle regole *JAPE*, per ricavare il link che punta alla pagina della persona sul portale *Penelope*.

Queste pagine del portale *Penelope*¹¹ sono utili in quanto sono presenti frasi che contengono sostantivi dai quali è possibile estrarre informazioni sulle competenze della persona, ad esempio il *curriculum* della persona o gli interessi personali. Il loro *url* viene memorizzato in un vettore da passare al punto successivo di estrazione delle frasi.

¹⁰ Sono pagine nella forma:

<http://www.unifi.it/cercachi/scheda.php?f=p&codice=3718&bol=AND&cognome=nesi&nome=paolo>.

¹¹ Ad esempio:

<http://www.unifi.it/index.php?module=ofform&mode=2&cmd=1&AA=2011&fac=200006&ord=N&doc=3f2a3d2e36302b>.

Nelle pagine delle persone di *Penelope* vi è una serie di *link* alle pagine dei corsi¹² (sempre sul portale *Penelope*), che a loro volta contengono informazioni utili per il *crawling* delle competenze. Quindi anche l'*url* di queste pagine dei corsi viene aggiunto al solito vettore.

L'output di questa parte è un vettore di *url*.

Estrazione frasi

In questa parte viene effettuato *natural language processing* tramite *GATE*, vengono analizzate le pagine indicate nel vettore di *url* precedente e individuate le frasi presenti. Ogni frase estrapolata viene memorizzata in un singolo *file* temporaneo.

Alla fine si ha una serie di *file* contenenti una singola frase, che vengono letti nella parte successiva.

Riconoscimento lingua

Anche in questa parte viene usato *GATE* per fare *natural language processing*. Per ogni frase (quindi per ogni *file* di input) viene individuata la lingua nel quale è scritta. Quindi a seconda di questa, viene accodata la frase in un *file* di frasi in italiano, oppure in uno in inglese. Se la lingua di una frase non viene riconosciuta oppure se viene riconosciuta come una lingua diversa dall'italiano o dall'inglese, questa viene accodata in un *file* di lingua sconosciuta. Quest'ultimo viene ignorato ed ha funzione di *debug*. Anche questi *file* come quelli delle frasi sono temporanei e vengono cancellati dopo l'esecuzione del *crawling* di *keyword*.

Estrazione sostantivi

In questa parte vengono usate due *pipe* distinte di *GATE* per individuare i sostantivi nelle frasi. Per le frasi in inglese viene usata la *pipe* standard di *GATE*, per quelle in italiano viene usata *TreeTagger*.

Queste *pipe* si occupano anche di non annotare i sostantivi che compaiono in un certo *gazetteer* di *stop words*.

Individuati ed estratti i sostantivi, questi vengono memorizzati in una tabella temporanea con colonne:

- keyword;
- lingua;
- occorrenze.

¹² Ad esempio:

<http://www.unifi.it/index.php?module=ofform&mode=1&cmd=3&AA=2011&fac=200006&cids=B047&pds=GEN&afId=292585&lan=0&ord=N&doc=3f2a3d2e36302b>.

Dove *keyword* è un singolo sostantivo estratto da una frase, la *lingua* è nota sapendo da quale dei due *file* (quello delle frasi italiane e quello delle inglesi) è stato estratto il sostantivo, e *occorrenze* è il numero di occorrenze di un certo sostantivo all'interno di uno dei due *file*.

Per la precisione la tabella temporanea è un **Hashtab Java** di due campi: un campo **key** che contiene una stringa nel formato **valore@lang** con **valore** che indica il valore del sostantivo e **lang** che indica la lingua (attualmente **it** o **en**), e un campo **freq** che contiene il numero di occorrenze.

Traduzione

Questa ultima parte si occupa di tradurre i sostantivi trovati nel passaggio precedente e di unire le occorrenze di eventuali doppioni in modo da avere una *keyword* unica per le stesse occorrenze in italiano e in inglese. Inoltre in questo modo è possibile rendere tutto il sistema multilingua, e un utente può effettuare *query* sia in italiano che in inglese.

La traduzione viene effettuata usando le *api* del servizio *Google Translate* integrato con una cache interna implementata con una tabella nel *database* con due colonne:

- valore in italiano;
- valore in inglese.

La cache funziona nel seguente modo: quando viene richiesta una traduzione di una *keyword* della tabella di *input*, viene prima interrogata la tabella di *cache*, se esiste già una riga con il valore da tradurre cercato allora viene usato il corrispondente valore tradotto di tale riga della *cache*; se non esiste una riga corrispondente viene invocato il servizio di *Google* e viene aggiunta una riga alla tabella di *cache* con i valori corrispondenti.

Essendo necessario l'uso di un servizio esterno è previsto un meccanismo di posticipazione delle traduzioni. Nel caso in cui, per esempio, *Google Translate* fosse momentaneamente non disponibile, la *keyword* viene comunque presa in considerazione, ma la sua traduzione viene impostata al valore **TO BE TRANSLATED**. Nel pannello di amministrazione è presente un pulsante con il relativo contatore di *keyword* da tradurre che lancia un nuovo tentativo di traduzione di tutte le *keyword* rimaste non tradotte. Ovviamente prima di lanciare la seconda fase di *crawling* delle competenze è bene che non ci siano *keyword* non tradotte.

Le *keyword* con il relativo valore tradotto vengono salvate in una tabella di *keyword*, specifica per ogni dipartimento¹³.

Questa tabella presenta le colonne:

- valore in italiano;

¹³ La tabella è descritta nella sezione 5.6.2.

- valore in inglese;
- occorrenze,
- *status*.

I primi due valori sono il valore originale della *keyword* e il suo valore tradotto nell'altra lingua. Le occorrenze indicano quante volte la *keyword* è stata trovata. *Status* è il valore che indica l'azione scelta dall'esperto di dominio sulla *keyword* (in *gazetteer* o in *black list*).

La tabella delle *keyword* non viene azzerata prima dell'esecuzione di questa fase, quindi i valori preesistenti rimangono memorizzati. Nel caso in cui vi è una richiesta di inserire una riga di una *keyword* già esistente nella tabella, viene semplicemente aggiornato il valore delle occorrenze della *keyword* e lasciato inalterato lo *status*.

Un'ultima considerazione è che esistono certe *keyword* che pur avendo valori diversi in una lingua, hanno la stessa traduzione nell'altra, ad esempio i sostantivi inglesi "program" e "programme" hanno la stessa traduzione italiana "programma". Questo problema viene risolto rendendo le *keyword* in questa forma effettivamente indipendenti, ad esempio, i valori precedenti vengono memorizzati con le due righe distinte della tabella 8. Queste *keyword* vengono

en_value	it_value	...
"program"	"programma"	...
"programme"	"programma"	...

Tabella 8: Esempio di *keyword* diverse con lo stesso valore

trattate come *keyword* differenti nonostante abbiano un valore che è lo stesso in entrambe.

4.3.2 Fase di selezione del *gazetteer* di *keyword*

Finita la fase di *crawling* delle *keyword* è necessario l'intervento di un esperto che decida quale deve essere il *gazetteer* delle *keyword* da prendere in considerazione per la seconda fase, ossia quelle che sono considerate significative per il personale di un certo dipartimento. Anche questa fase è divisa dipartimento per dipartimento e l'esperto deve agire nel pannello di configurazione di un dipartimento nella seconda *tab* vista in figura 12 a pagina 80.

Nella figura 16 è possibile vedere uno schema di questa fase, dove l'esperto può cambiare lingua ad una *keyword*, segnare una *keyword* come appartenente al *gazetteer* di *input* della fase di *crawling* di competenze, oppure segnarla come in *black list*.

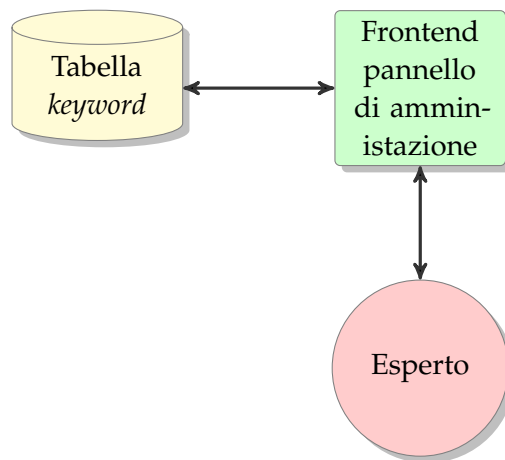


Figura 16: Schema della fase di creazione del *gazetteer*

4.3.3 *Crawling delle competenze*

Quando è stato scelto un *gazetteer* di *keyword* consono, è possibile lanciare la fase di *crawling* di competenze cliccando sul pulsante in figura 17.



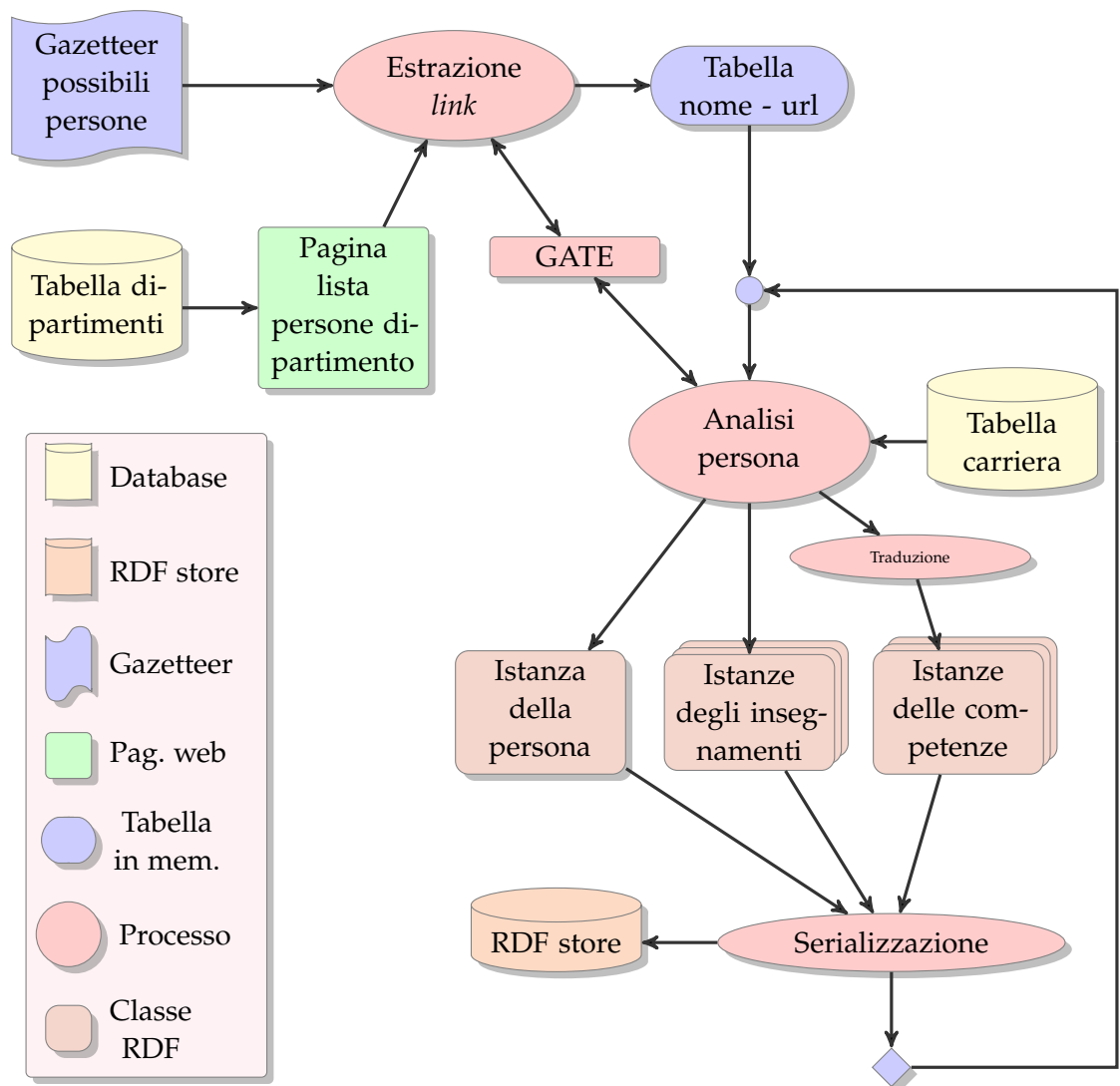
Figura 17: Pulsante per lanciare il *crawling* delle competenze.

Il funzionamento iniziale della chiamata è identico a quello del *crawling* delle *keyword*. Viene fatta una chiamata *Javascript* come spiegato in sezione 5.3, viene aggiunto un processo in coda, e quando è il suo turno viene eseguito il *thread* **CompetenceKeyCrawler**. A differenza della fase precedente il *thread* riconosce che l'operazione è di *crawling* di competenze, perciò istanzia la classe **CompetenceExtractionEngine** ed esegue il suo metodo **run**.

In figura 18 è possibile vedere uno schema del funzionamento di questa fase. I dettagli della traduzione sono stati omessi per motivi di spazio, comunque sono completamente analoghi a quelli in fase di *crawling* di *keyword*.

In questa fase vengono create le istanze delle classi *RDFS* che riguardano le persone, i corsi insegnati e le competenze. In particolare vengono create, o modificate, le classi di tipo:

- **foaf:Person** per la persona;

Figura 18: Schema della fase di *crawling* delle competenze

- **uni:Course** per gli insegnamenti;
- **uni:temporaryXXXStore** per le competenze, dove **XXX** sta per il codice associato ad un certo dipartimento.

Nella sezione 5.1 è possibile vedere nel dettaglio lo schema *RDFS* e le relazioni tra queste istanze.

Inizialmente è simile alla fase di estrazione delle *keyword*, vengono estratti i *link* relativi alle persone di un dipartimento che compaiono nel *gazetteer*. A differenza di questa, viene esplorato solo il primo *link* di ogni persona e memorizzato in una tabella assieme al nome di quest'ultima.

Le fasi di analisi della persona e di serializzazione delle classi vengono iterate per ogni riga di tale tabella, quindi per ogni persona.

Analisi della persona

La fase di analisi della persona si occupa di creare le istanze delle classi *RDFS*, anche se non vengono memorizzate nell'*RDF Store*. Nello specifico vengono creati degli oggetti *Java* di classe **UnifiOntologyInstance**; la memorizzazione nell'*RDF Store*, assieme alle proprietà che mettono in relazione le diverse istanze, avviene nella fase successiva di serializzazione.

Partendo dall'*url* della pagina e dal nome della persona, viene richiamato il metodo **analyze** della classe **PersonAnalyzer** che esegue le seguenti operazioni:

1. vengono recuperate le informazioni riguardanti la persona:
 - l'hash del codice fiscale tramite la tabella **carriera**;
 - l'indirizzo delle pagine della persona, della piattaforma *cercachi*, e di *Penelope*;
 - i link alle pagine degli eventuali corsi insegnati dalla persona;
2. viene creata una istanza della persona;
3. Per ogni link ad un eventuale corso vengono recuperate le sue informazioni dalle pagine di questi:
 - il nome del corso;
 - il codice del corso;
4. vengono create le istanze di tali corsi;
5. vengono cercate le competenze su ogni pagina del corso e create le loro istanze;
6. vengono cercate le competenze sulla pagina della persona e create le loro istanze.

Il recupero dell'hash del codice fiscale serve per creare l'*URI* della persona, ed avviene facendo una *query* al *database* usando il nome della stessa. Se la *query* non torna il valore allora la persona viene saltata.

Il recupero degli indirizzi delle pagine avviene sfruttando *GATE*, così come gli eventuali indirizzi delle pagine dei corsi.

L'istanza della persona viene creata usando l'*hash* del codice fiscale come parte dell'*URI*¹⁴, alla persona vengono assegnate le proprietà riguardanti l'indirizzo delle sue pagine, e il dipartimento di appartenenza.

Viene usato *GATE* sulla pagina di *Penelope* per trovare eventuali *link* alle pagine dei corsi insegnati dalla persona. Ognuna di queste pagine di corsi viene analizzata singolarmente, viene usato *GATE* per estrarre il nome del corso e il suo codice da usare come parte dell'*URI*. Viene creata una istanza per tale corso assegnando le proprietà del suo nome e della sua pagina.

Sempre sulla pagina del corso viene usato *GATE* per trovare le competenze, e per ognuna di esse viene tradotto il valore con un sistema analogo a quello usato durante il *crawling* delle *keyword*, e viene creata l'istanza della competenza. La traduzione viene rieffettuata perché possono essere estratte competenze composte da più *keyword*. Viene memorizzato nell'oggetto che rappresenta l'istanza della competenza, anche il riferimento al corso dove questa è stata trovata.

Il procedimento si ripete per tutti i corsi presenti, e infine viene analizzata anche la pagina di *Penelope* della persona.

Serializzazione

Questa è la fase in cui le istanze create vengono effettivamente memorizzate nell'*RDF Store* e in cui vengono anche create le relazioni tra queste.

Le operazioni effettuate sono:

1. controlla se le competenze trovate esistono già o no; questo serve per sapere se assegnare alla risorsa il tipo **uni:temporaryXXXStore**¹⁵;
2. scrittura nell'*RDF Store* dell'istanza della persona e di quelle delle competenze e dei corsi;
3. scrittura dell'associazione tra persona e competenza usando un *bnode*.

I dettagli della fase di serializzazione sono descritti meglio nella sezione 5.1.

4.3.4 *Organizzazione dello SKOS*

Questa fase, come quella di creazione del *gazetteer*, prevede l'intervento di un esperto. In questa fase vengono organizzate le competenze, estratte nella fase

¹⁴ Vedere la sezione 5.1 per i dettagli sugli *URI* e sulle proprietà.

¹⁵ Vedi sezione 5.1.

precedente, in un albero gerarchico, per descrivere la specificità o la generalità di una competenza rispetto ad un'altra.

Durante la creazione dell'albero, vengono create, in maniera trasparente all'esperto, le opportune relazioni semantiche nello *SKOS*.

ANALISI DELL'APPLICAZIONE

5.1 ONTOLOGIA

L'applicazione si interfaccia ad una ontologia gestita da un *RDF store Sesame*. In questa ontologia vengono inserite le istanze delle classi riguardanti le persone, i loro insegnamenti, e le loro competenze.

In figura 19 è possibile vedere uno schema dell'ontologia di dominio, con le varie classi e le relazioni. Il prefisso **uni:** corrisponde al *namespace* **<http://www.dsi.unifi.it/CMSAteneoCompetence#>**.

Il *crawler* popola l'ontologia con le informazioni che estrae dal sito web dell'ateneo. L'applicazione scrive sull'ontologia in due punti diversi:

- quando viene fatto il *crawling* delle competenze;
- quando l'esperto organizza lo SKOS delle competenze;

5.1.1 Crawling di competenze

In figura 20 vengono mostrate le istanze che vengono create, le loro proprietà e relazioni durante la fase di *crawling* di competenze. Tali istanze se già esistenti, ad esempio nel caso di un lancio di *crawling* ripetuto, vengono aggiornate con eventuali modifiche.

Durante la fase di *crawling* delle competenze vengono analizzate le persone singolarmente¹, e per ogni persona vengono create in ordine:

1. l'istanza della persona;
2. gli eventuali insegnamenti di quella persona;
3. le competenze associate ai corsi e alla persona.

Dopo la creazione di tali risorse vengono creati per ogni competenza i *blank node* che indicano l'associazione della competenza alla persona, e le relazioni tra le varie istanze.

Per quanto riguarda l'istanza della persona, in ordine vengono effettuate le seguenti operazioni:

1. viene creata la risorsa usando un *URI* composto dal prefisso **urn:u-gov:unifi:AC_AB0:** seguito dall'*hash* del codice fiscale della persona prelevato dalla tabella **carriera**²;

¹ Vedere sezione 4.3.3 per dettagli sul *crawler* di competenze.

² Per i dettagli vedere sezione 5.6.7.

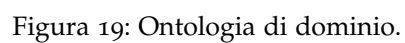
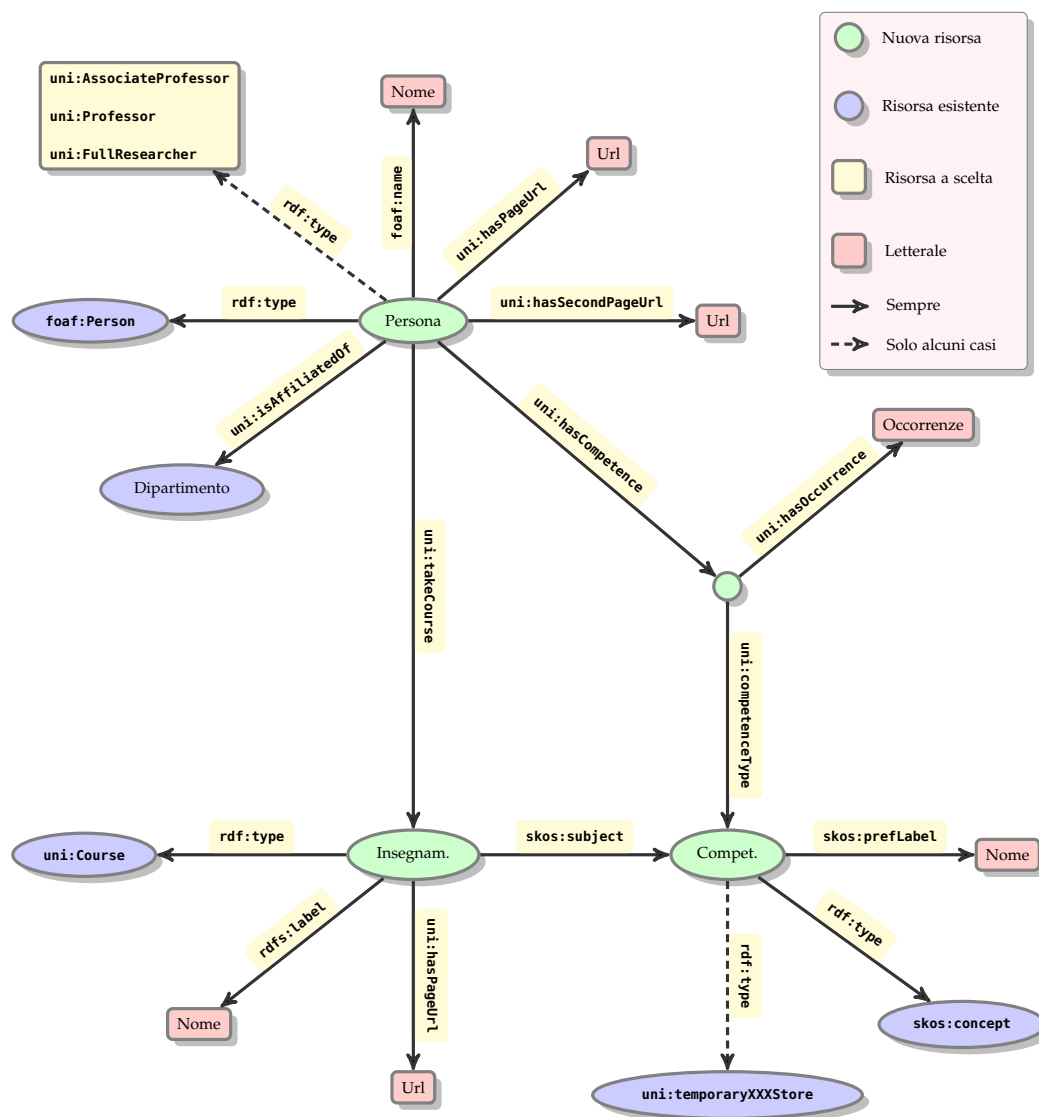


Figura 19: Ontologia di dominio.

Figura 20: Istanze create in fase di *crawling* di competenze

2. viene impostato il tipo della risorsa con **rdf:type** a **foaf:Person**;
3. viene impostato il nome della persona con la proprietà **foaf:name**;
4. vengono impostate le pagine della persona tramite le proprietà **uni:hasPageUrl** e **uni:hasSecondPageUrl**, la prima indica la pagina della piattaforma *Cercachi*, la seconda di *Penelope*;
5. vengono create le risorse riguardanti gli insegnamenti;
6. vengono create le risorse riguardanti le competenze trovate nella pagina della piattaforma *Penelope*.

Viene valutato se la persona analizzata è un professore associato, un professore, o un ricercatore, e, nel caso, viene impostato il tipo della persona anche³ a uno dei seguenti:

- **uni:AssociateProfessor** per un professore associato;
- **uni:Professor** per un professore;
- **uni:FullResearcher** per un ricercatore associato.

Le istanze degli insegnamenti vengono create durante l'analisi della persona, e vengono effettuate le seguenti operazioni:

1. viene creata la risorsa usando sempre il prefisso **urn:u-gov:unifi:AC_AB0:** seguito dal codice del corso prelevato dalla sua pagina;
2. viene impostato il tipo della risorsa con **rdf:type** a **uni:Course**;
3. viene impostato il nome della risorsa con la proprietà **rdfs:label**;
4. viene impostata la pagina del corso sulla piattaforma *Penelope* con la proprietà **uni:hasPageUrl**;
5. vengono create le risorse riguardanti le competenze trovate nella pagina del corso.

Le istanze delle competenze possono venire create a partire dalla pagina di una persona o dalle pagine degli insegnamenti di tale persona; le risorse create nei due casi sono identiche, cambia una proprietà che associa l'insegnamento alla competenza che viene scritta nella fase successiva. Le operazioni effettuate sono le seguenti:

³ In *RDF* è possibile assegnare più di un tipo alle risorse.

1. viene creata la risorsa con l'*URI* composto da **http://www.dsi.unifi.it/** seguito dal codice del dipartimento per il quale è stato lanciato il *crawling*, seguito dal carattere **#**, seguito dal valore testuale della competenza (codificata come un uri);
2. viene impostato il tipo della risorsa con **rdf:type** a **skos:Concept**;
3. vengono impostati i valori nelle diverse lingue (attualmente due) con la proprietà **skos:prefLabel** differenziando le lingue appendendo al valore **@** seguito dal codice della lingua;

Le fasi appena analizzate corrispondono nel codice ai blocchi in output della fase di analisi della persona dello schema di figura 18 a pagina 89. La fase successiva di serializzazione, oltre che a scrivere fisicamente nell'*RDF store* le istanze viste sopra effettua le seguenti operazioni:

1. se la competenza inserita è nuova viene aggiunto alla competenza il tipo **uni:temporary** seguito dal codice del dipartimento che si sta analizzando (indicato con **XXXStore** nello schema);
2. se la competenza inserita è stata trovata su una pagina di un corso viene creata l'associazione tra il corso e la competenza con la proprietà **skos:subject**;
3. crea i *blank node* che si vedono nello schema;
4. crea anche le proprietà che indicano le relazioni tra queste istanze.

L'uso del vocabolario *SKOS* per rappresentare le competenze e per associare un corso con queste rende possibile identificare i soggetti trattati da un certo corso oltre che organizzare le competenze di una persona.

La fase di creazione dei *blank node* si articola come segue, per ogni competenza estratta:

1. viene creato un *blank node* con identificativo **_:** seguito dal nome della persona senza spazi, seguito dalla stringa **Competence**, seguito dall'*URI* della competenza;
2. viene creata la relazione tra la persona e il *blank node* tramite la proprietà **uni:hasCompetence**;
3. viene creata la relazione tra il *blank node* e la competenza tramite la proprietà **uni:competenceType**;
4. viene aggiunta la proprietà del *blank node* **uni:hasOccurrence** con il numero di occorrenze della competenza per quella persona (comprese le pagine dei suoi corsi).

Le altre relazioni rimaste sono quelle tra la persona e i suoi insegnamenti; per ogni corso viene associata la persona a questi tramite la proprietà **uni:takeCourse**.

Se le istanze create esistono già in queste fasi vengono solamente aggiornati i valori che eventualmente differiscono, come il numero di occorrenze di una competenza su una persona.

5.1.2 Organizzazione dello SKOS

Durante la creazione e la modifica dello SKOS l'esperto organizza le competenze in alberi gerarchici. Dal lato dell'ontologia questo significa assegnare alle competenze, che già sono istanze di **skos:Concept**, delle relazioni semantiche tramite la proprietà **skos:narrower**. Inoltre vengono organizzati dei vocabolari tramite delle istanze di **skos:ConceptScheme**, una per ogni dipartimento, e le proprietà di questi **skos:hasTopConcept**.

Gli uri dei *concept schema* sono formati da **http://www.dsi.unifi.it/** seguito dal codice del dipartimento, seguito da **#conceptSchema**, e questi *concept schema* vengono sempre inizializzati per tutti i dipartimenti.

Nel pannello di amministrazione di figura 11 a pagina 79 vengono mostrati a sinistra le competenze ancora non organizzate nello SKOS, a destra quelle organizzate. Le competenze di sinistra vengono prelevate interrogando l'*RDF Store* su tutti gli *skos:Concept* di tipo **uni:temporaryXXXStore** con **XXXStore** che indica il codice del dipartimento che viene amministrato. Quando una competenza viene inserita nello SKOS di destra, viene eliminato da questa il tipo **uni:temporaryXXXStore**, con **XXXStore** codice del dipartimento.

L'esperto può anche creare nuove istanze di **skos:Concept** fornendo una *label*, e inserirle nella relazione semantica di un certo albero di competenze allo scopo di creare degli aggregatori di competenze effettivamente prelevate dalle pagine.

Nell'operazione di spostamento delle competenze, il suo *URI* rimane inalterato, quindi le proprietà **uni:competenceType** e **skos:subject** rimangono valide.

In figura 21 è possibile vedere uno schema delle possibili relazioni tra una competenza appena inserita, o spostata, nello SKOS, e le altre risorse. Notare l'assenza del tipo **uni:temporaryXXXStore**, che come detto viene eliminato. Inoltre vengono omesse le proprietà **uni:competenceType** e **skos:subject** che hanno come oggetto le competenze (a meno che non siano competenze create dall'esperto).

Le due proprietà:

- **skos:hasTopConcept**,
- **skos:narrower**,

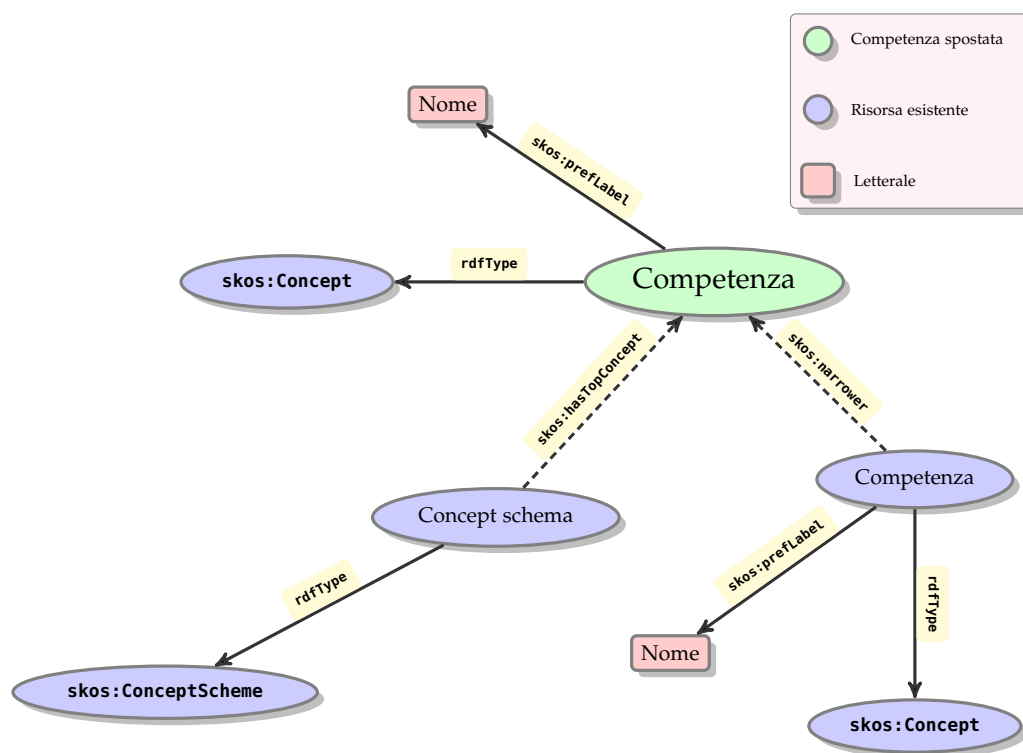


Figura 21: Possibili relazioni di una competenza nello SKOS

sono mutuamente esclusive rispetto alla nuova competenza inserita. La proprietà **skos:hasTopConcept** da un *concept* schema, viene creata se la competenza viene spostata a livello zero nello SKOS. **skos:narrower** se viene spostata sotto ad un'altra competenza.

Vedere la sezione 2.5 per informazioni sul significato semantico delle proprietà usate.

5.2 APPLICAZIONI GATE

Durante l'esecuzione delle operazioni di *crawling* vengono usate quattro *pipe* di GATE identificate dal nome del file che viene caricato tramite *GATE embedded* all'interno del codice nei diversi punti del programma:

- **langDetector.xgapp** viene caricata in fase di *crawling* di *keyword*, durante la fase di riconoscimento della lingua dello schema di figura 15 di pagina 83 e svolge la funzione di identificare il linguaggio del documento di *input*;
- **en.xgapp** viene caricata nella stessa fase di *crawling* di *keyword*, durante la fase di estrazione dei sostantivi del solito schema del precedente; la sua funzione è quella di annotare i sostantivi del documento di *input* che deve essere in lingua inglese;
- **it.xgapp** viene caricata nello stesso punto del precedente e svolge la stessa funzione, però il documento di *input* deve essere in lingua italiana;
- **competenceExtraction.xgapp** è l'applicazione più importante, viene richiamata in ogni punto in cui vengono usate le funzioni di GATE che non siano i precedenti tre punti. Svolge funzioni diverse a seconda del contesto in cui viene richiamata e del documento che viene fornito come *input*.

Queste applicazioni vengono preventivamente sviluppate tramite *GATE developer* e salvate nei file sopra indicati. Java fa uso di tali applicazioni interfacciandosi nel codice tramite *GATE embedded*.

Il metodo di interfacciamento tra l'applicazione e GATE è simile in tutti i punti in cui viene usato, e per le quattro diverse *pipe*:

1. viene istanziata la classe **StandAloneAnnie** che contiene i metodi necessari per potersi interfacciare con *GATE embedded* con il codice:

```
StandAloneAnnie annie = new StandAloneAnnie();
```

il costruttore di **StandAloneAnnie** fornisce la possibilità di stabilire un codice numerico per indicare quale delle quattro applicazioni caricare;

- viene creato un corpus contenente i documenti da analizzare, un corpus consiste semplicemente in un **ArrayList** di stringhe di indirizzi di documenti. Spesso il corpus contiene un solo documento, e si istanzia nel seguente modo:

```
ArrayList<String> corpusDocs = new ArrayList<String>();  
corpusDocs.add(pageLink);
```

- vengono scelte quali annotazioni reperire che vanno indicate in un **HashSet** di stringhe nel seguente modo:

```
HashSet<String> annotationTypes = new HashSet<String>();  
annotationTypes.add("UnifiCompetence");
```

- viene lanciato gate richiamando il codice:

```
Set<Annotation> annotations = annie.computeAnnotationsOnCorpus(  
    corpusDocs, annotationTypes);
```

che ritorna un **Set** iterabile di **Annotation**;

- viene scorso il **Set** ritornato nel seguente modo:

```
Iterator<Annotation> annotIter = annotations.iterator();  
while(annotIter.hasNext()){  
    Annotation currentAnnotation = (Annotation)annotIter.next();  
    ...  
}
```

- è possibile ottenere il testo annotato da una certa annotazione nel seguente modo:

```
String currentAnnotationText = annie.getTextByOffset(  
    currentAnnotation.getStartNode().getOffset(),  
    currentAnnotation.getEndNode().getOffset(),  
    pageLink);
```

- è possibile ottenere una certa *feature*, associata ad una certa annotazione, nel seguente modo:

```
String lang = (String) currentAnnotation.getFeatures().get("lang");
```

5.2.1 Applicazione **langDetector.xgapp**

Questa applicazione usa il *plugin* **LingPipe** di GATE, che è un *wrapper* per la suite di librerie *Java LingPipe*.

Tale plugin fornisce diverse risorse tra cui: un *tagger*, un *sentence splitter* e un *tokenizer*. L'unica risorsa che viene usata in questa *pipe* è **LingPipe Language Identifier** che, a differenza delle altre risorse, non produce annotazioni, ma crea una *feature* di documento chiamata *lang* avente come valore il codice della lingua del documento.

Prima di questa risorsa viene eseguita la risorsa **Document Reset** che riporta il documento allo stato iniziale azzerando eventuali annotazioni o *feature* prima di eseguire la *pipe*.

Il risultato dell'esecuzione di questa *pipe* è una *feature* di documento che viene interpretata dal *crawler* delle *keyword* per riconoscere se un *file* contiene una frase in italiano o in inglese.

5.2.2 Applicazione *en.xgapp*

Questa applicazione usa una *pipe* ANNIE composta dalle seguenti risorse in ordine:

1. **Document Reset**;
2. **ANNIE Gazetteer** con i seguenti *gazetteer*:
 - **GazetteerCaseSensitive**;
 - **GazetteerCaseInsensitive**;
 - **GazetteerStopWords**;
3. **GATE Unicode Tokeniser**;
4. **ANNIE Sentence Splitter**;
5. **ANNIE POS Tagger**;
6. **ANNIE NE Transducer** che usa le regole indicate nella grammatica *en.jape*.

Le risorse vengono eseguite in cascata nell'ordine indicato.

Document Reset riporta il documento alle condizioni iniziali. I *gazetteer* permettono di annotare nel documento certe parole, in particolare vengono annotate le *stop words* indicate nel *gazetteer GazetteerStopWords-en* che sono una serie di parole indesiderate. **Unicode Tokeniser** annota nel testo ogni *token* ossia ogni singola parola, o segno di punteggiatura. **Sentence Splitter** annota ogni frase nel testo, per funzionare ha bisogno che il testo sia già annotato con i *token*. **POS Tagger** annota le parti del testo (nomi, verbi, etc. . .), per funzionare ha bisogno che il testo sia già annotato con le frasi. **NE Transducer** applica le regole JAPE indicate nella sua grammatica, tali regole fanno in modo di eliminare le annotazioni di tipo *nome* risultanti dall'esecuzione del *pos tagger* che sono state annotate come *stop word* dall'esecuzione dei *gazetteer*.

Il risultato della *pipe* è una serie di annotazioni denominate *Noun* che indicano tutti i punti del testo dove compare un sostantivo. Questi sostantivi vengono estratti dal *crawler* delle *keyword* e passati al processo di traduzione e memorizzazione nel *database* delle *keyword*.

5.2.3 Applicazione **it.xgapp**

Questa applicazione è diversa dalla precedente nonostante abbia le stesse funzioni. Il motivo è che *ANNIE*, di base, non è in grado di funzionare sulla lingua italiana.

Per risolvere questo problema viene usato il plugin di GATE **TreeTagger** che consiste in un *wrapper* per un programma esterno chiamato *TreeTagger*.

La *pipe* quindi consiste nelle seguenti risorse:

1. **Document Reset**;
2. **ANNIE Gazetteer** con i seguenti *gazetteer*:
 - **GazetteerCaseSensitive**;
 - **GazetteerCaseInsensitive**;
 - **GazetteerStopWords**;
3. **GATE Unicode Tokeniser**;
4. **TreeTagger**;
5. **ANNIE NE Transducer** che usa le regole indicate nella grammatica **it.jape**.

Le funzioni sono simili alla *pipe* per la lingua inglese, il *sentence splitter* e il *pos tagger* vengono sostituiti da *TreeTagger*, e la grammatica *JAPE* usata è quella del file **it.jape** invece di **en.jape**.

Il risultato della *pipe* è identico a quello di **en.xgapp** e viene usato nel solito modo.

5.2.4 Applicazione **competenceExtraction.xgapp**

È la *pipe* principale del programma, essa viene richiamata ogni volta che vengono richiesti i servizi di GATE che non siano uno dei punti visti per le altre *pipe*.

Ogni volta che viene richiamata viene eseguita tutta la *pipe*, e viene specificato quali sono le annotazioni da considerare significative.

La *pipe* completa è composta da:

1. **document reset** per riportare il documento allo stato originale;

2. **ateneo gazetteer** è il *gazetteer* che presenta le seguenti liste:
 - **ateneoPerson.lst** annota con il nome **ateneoPerson** tutti i nomi delle persone presenti nella lista;
 - **unifiAddedCompetence_skos.lst** annota con il nome **AteneoCompetence** tutte le *keyword* di lingua inglese presenti nella lista, tale lista viene inizializzata con le *keyword* del *database* all'inizio della fase di *crawling* di competenze;
 - **unifiAddedCompetence_skos_it.lst** è analoga alla precedente, ma il nome delle annotazioni è **AteneoCompetence_it**;
3. **Unicode tokenizer** annota nel documento ogni parola e simbolo di punteggiatura con **Token**;
4. **Sentence splitter** annota nel documento le frasi con **Sentence**;
5. **Pos tagger** riconosce le parti del testo (sostantivo, verbo, etc...) nel documento e le segna con la *feature category* di **Token**;
6. **Annotation set transfer** serve per identificare i tag *HTML* nel documento come annotazioni dello stesso nome del *tag*;
7. **competence grammar** applica le regole *JAPE* specificate, che sono:
 - **linkToPerson** annota i link alle pagine delle persone nelle pagine della lista persone del dipartimento con il nome **LinkToPerson** e con l'*url* del *link* come *feature href* di tali annotazioni, tali link vengono cercati tra le parti del documento già annotate con **ateneoPerson** dal *gazetteer*;
 - **linkToPenelope** annota il link alla piattaforma *Penelope* nella pagina della persona su *cercachi* con il nome **LinkToPenelope** e con l'*url* del *link* come *feature href* di tali annotazioni;
 - **personType** annota nella pagina della persona su *cercachi* il tipo che può essere **Professor**, **AssociateProfessor**, o **FullResearcher**;
 - **personSection** annota i link agli eventuali corsi insegnati da una persona con il nome **PersonSection**, la *feature* di tale annotazione **section** uguale a **corso**, e l'*url* del *link* come *feature href*;
 - **courseSection** annota il nome e il codice del corso in una pagina di un insegnamento di una persona con il nome **CourseSection** e le sue *feature name* e **code**;
 - **ateneo_competence** sfruttando i risultati del *Pos tagger* annota con il nome di **UnifiCompetence** le composizioni di una o più *keyword* tramite congiunzioni di **AteneoCompetence**.

Come già detto anche se la *pipe* vista viene eseguita nell'ordine indicato tutte le volte che viene usato *GATE*, i suoi effetti saranno diversi a seconda delle annotazioni che vengono prelevate e del documento sul quale viene eseguita.

Riferendoci allo schema della fase di estrazione delle *keyword* di figura 15 di pagina 83, la *pipe* **competenceExtraction.xgapp** viene usata solo nelle fasi:

estrazione dei link dalla pagina con la lista iniziale delle persone;

estrazione delle frasi da tutte le pagine delle persone e dei loro insegnamenti.

Durante l'estrazione dei link vengono cercate dal programma le annotazioni **LinkToPerson** nella pagina iniziale del dipartimento, quindi della *pipe* viene sfruttato il *gazetteer* di persone iniziale, e la regola *JAPE* che permette di recuperare i link di tali persone.

Nella fase di estrazione delle frasi vengono cercate le annotazioni di tipo **Sentence**, quindi si sfrutta il *Sentence splitter* che a sua volta si appoggia al *tokenizer*.

Le altre due volte che viene usato *GATE* nella fase di estrazione delle *keyword* vengono usate le altre *pipe* **langDetector.xgapp**, **it.xgapp** ed **en.xgapp**.

Nella fase di *crawling* delle competenze visibile nello schema di figura 18 di pagina 89, *GATE* viene usato in due punti:

estrazione dei link dalla pagina con la lista iniziale delle persone;

analisi della persona per ottenere le istanze della persona, dei suoi insegnamenti, e delle sue competenze.

Il primo punto è analogo all'estrazione dei link nella fase di *crawling* di *keywors*.

Nel secondo punto viene usato *GATE* più volte al suo interno, e per ogni persona:

1. viene estratto l'indirizzo della pagina alla piattaforma *Penelope* usando l'annotazione **LinkToPenelope** creata dalle regole *JAPE*;
2. viene estratto il tipo di persona valutando quale annotazione crea la regola *JAPE* **personType**;
3. vengono estratti l'indirizzi alle eventuali pagine dei corsi insegnati usando le annotazioni **PersonSection** delle regole *JAPE*;
4. vengono estratte le competenze nella pagina della persona usando le annotazioni **UnifiCompetence** della regola **ateneo_competence** che si appoggia alle *keyword* annotate tramite *gazetteer*.

Per ogni pagina del corso estratta nel terzo punto, vengono eseguite le seguenti operazioni:

1. viene estratto il nome del corso e il suo codice tramite l'annotazione **CourseSection** basandosi sui *tag HTML*;
2. vengono estratte le competenze nella pagina del corso usando le annotazioni **UnifiCompetence** della regola **ateneo_competence** che si appoggia alle *keyword* annotate tramite *gazetteer*.

5.3 ANALISI DELLE *jsp* E DELLE CHIAMATE ALLE *serolet*

Tutte le interazioni tra le pagine *JSP*, che sono quelle con cui interagisce direttamente l'utente, e la *serolet Java* che sta dietro, seguono uno schema ben preciso. Questo schema viene usato sia quando un utente invia una richiesta alla *serolet*, ad esempio di far partire un *crawling*, sia quando la *serolet* deve far visualizzare qualcosa all'utente, ad esempio la tabella delle *keyword*.

Questo schema è visibile in figura 22. A sinistra vi sono le pagine *JSP* vis-

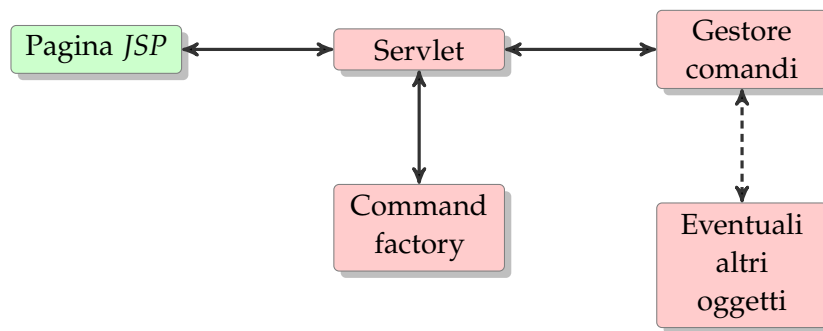


Figura 22: Schema delle chiamate *Javascript*

ibili dall'utente⁴. Da queste vengono effettuate delle chiamate *HTTP* tramite *Javascript* a **/skosServlet/SkosManager?operation=codiceOperazione** con il parametro obbligatorio **operation** che consiste in una stringa che indica l'operazione richiesta, e altri parametri opzionali a seconda dell'operazione richiesta. Queste chiamate possono essere effettuate a partire da un'azione dell'utente, ad esempio un click su un pulsante, oppure possono essere richieste direttamente dalla pagina per poter visualizzare dei dati, ad esempio la tabella delle *keyword*; comunque sia tutte vengono gestite dalla *serolet* rappresentata dalla classe **OSIM_SkosServlet**⁵, precisamente dal metodo **executeRequest** di tale classe.

Il *Javascript* chiamante si aspetta un valore di ritorno che la *serolet* si occupa di fornire scrivendo su un oggetto **HttpServletResponse** dove viene settato un valore e un tipo. Solitamente il tipo è **application/json; charset=utf-8** che indica che il valore è un oggetto *JSON*, infatti questo è il formato usato dalla

⁴ Vedere sezione 5.5.

⁵ Vedere sezione 5.7.

servlet per comunicare con la pagina *JSP*. A sua volta la *servlet* riceve un oggetto **HttpServletRequest**, da dove è possibile estrarre i diversi parametri passati con la chiamata, compreso quello indispensabile **operation**.

La *servlet*, tramite il *command factory* **OSIM_CommandFactory**⁶, crea un oggetto gestore di comandi che può essere descritto da diverse classi che però deve implementare l'interfaccia **OSIM_RequestCommand**⁷. La funzione del *command factory* è proprio quella indicare quale tipo di classe gestore istanziare per una certa operazione.

Qualunque sia il gestore, viene eseguito il metodo **execute** su di esso. Tale metodo ha come parametri un oggetto **HttpServletRequest**, il solito passato alla *servlet* che lo *gira* al gestore, da cui estrae il tipo di operazione e gli eventuali altri parametri necessari allo svolgimento di essa. Il valore di ritorno è un **JSONObject** che poi viene *incapsulato* dalla *servlet* all'interno della risposta *http*, che è il risultato dello svolgimento dell'operazione.

Sovente il gestore dei comandi si avvale dell'uso di altre classi o thread.

5.4 DETTAGLI DELLE CHIAMATE NELLE DUE FASI DI *crawling*

Vengono analizzate nel dettaglio il flusso di chiamate che avviene dopo il gestore dei comandi nel caso delle fasi di *crawling*. Nello schema di figura 23 è possibile vedere tale flusso di chiamate. Ogni blocco è associato ad una classe, la scelta dei nomi in inglese per questi è dettata dalla somiglianza con i nomi delle classi che li implementano. Il colore differenzia i processi principali da quelli di supporto, inoltre sono evidenziati anche i processi che creano l'output per le due fasi di *crawling*.

La corrispondenza dei blocchi con le classi è la seguente⁸:

gestore comandi è implementato da **OSIM_RequestKBCommand**, come già visto nella sezione 5.3;

coda processi è implementata da **ProcessQueue** e le relative classi del pacchetto **ProcessQueuing**, la coda dei processi è spiegata nel dettaglio in sezione 5.4.1;

thread è implementato dalla classe **CompetenceKeyCrawler**;

keyword engine è implementato dalla classe **KeywordExtractionEngine**;

competence engine è implementato dalla classe **CompetenceExtractionEngine**;

chunk splitter è implementato dalla classe **ChunkSplitter**;

⁶ Vedere nota 5.

⁷ Vedere nota 5.

⁸ Vedere la sezione 5.7 per i dettagli sulle classi.

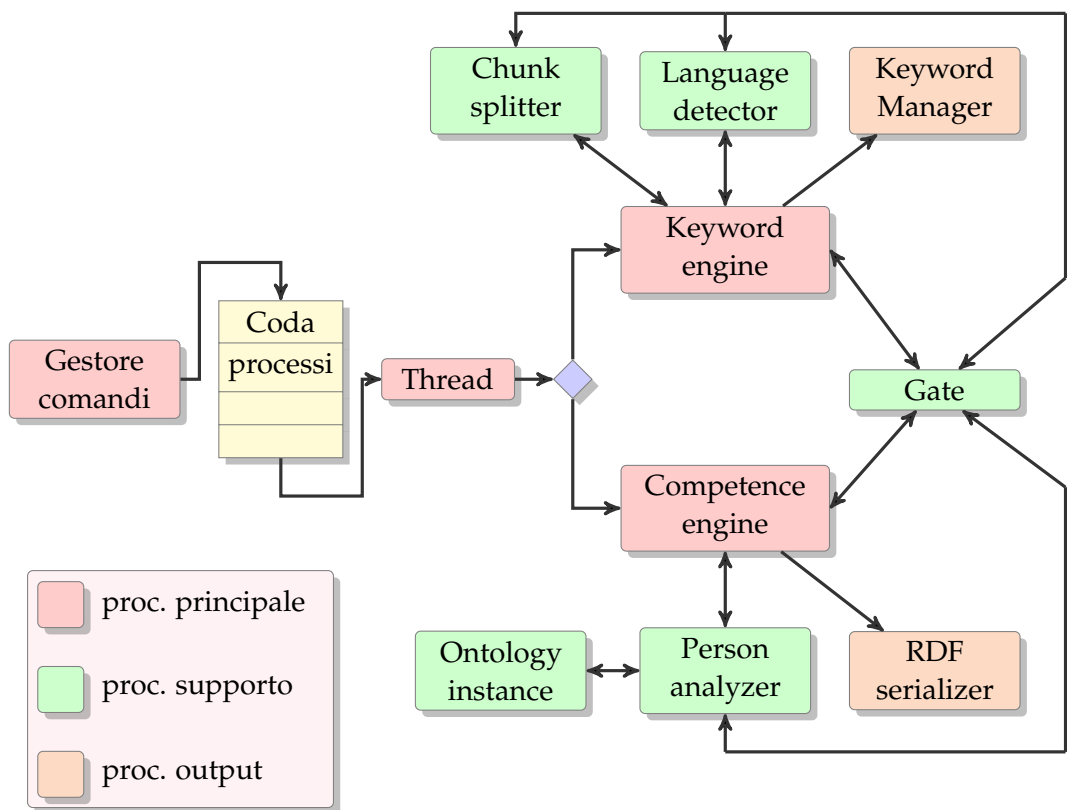


Figura 23: Schema delle chiamate per le fasi di *crawling*

language detector è implementato dalla classe **LanguageDetector**;

keyword manager è implementato dalla classe **OSIM_KeywordManager**;

person analyzer è implementato dalla classe **PersonAnalyzer**;

RDF serializer è implementato dalla classe **OSIM_RdfSerializer**;

gate è implementato dalla classe **StandAloneAnnie**.

La coda si occupa di far sì che solo un thread di *crawling* sia attivo in ogni istante, questo perché tali processi sono molto onerosi.

Il *thread* si occupa di interrogare il tipo di operazione richiesta, se è richiesto il *crawling* di *keyword* allora crea un nuovo **KeywordExtractionEngine** e lancia il suo metodo **extractKey**. Se è richiesto un *crawling* di competenze istanzia **CompetenceExtractionEngine** e lancia il suo metodo **run**⁹.

Completata l'esecuzione di uno dei due processi, il *thread* notifica la coda del completamento così che questa lanci il successivo *thread*.

Il *keyword engine* usa le classi per:

1. **ChunkSplitter** per estrarre le frasi dalle pagine del dipartimento e salvarle nei *file* temporanei;
2. **LanguageDetector** per riconoscere la lingua di questi *file* e accodarli ai tre *file* delle frasi in lingua italiana, inglese o sconosciuta;
3. **StandAloneAnnie** per estrarre i sostantivi dai due *file* di frasi in lingua italiana e inglese;
4. **KeywordManager** per salvare i sostantivi estratti nel database.

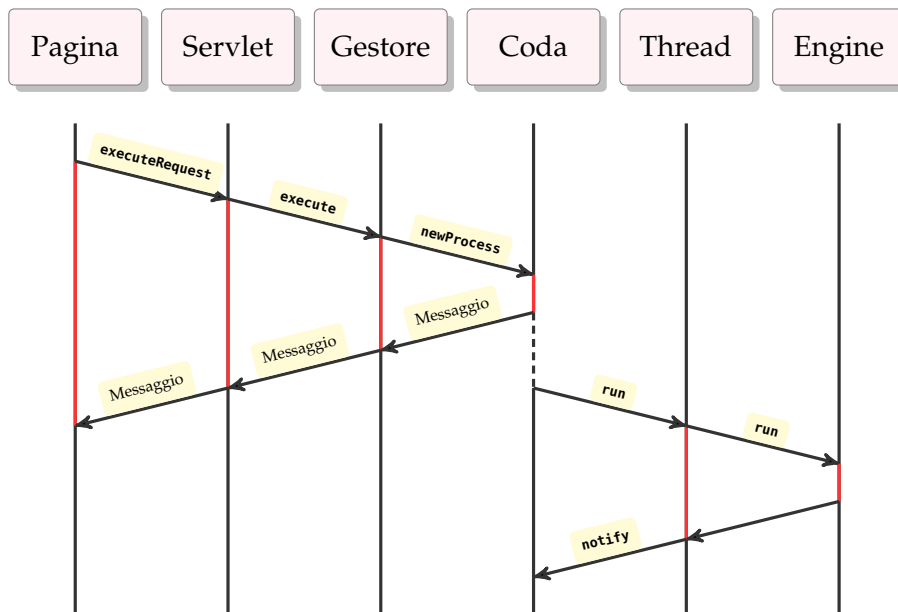
A sua volta **ChunkSplitter** e **LanguageDetector** fanno uso di **StandAloneAnnie**.

Il *competence engine* usa il metodo **analyze** di **PersonAnalyzer** per analizzare ogni persona del dipartimento e ricavare degli oggetti di tipo **UnifiOntologyInstance** da passare alla classe **OSIM_RdfSerializer** che interpreta tali oggetti e scrive le triple corrispondenti nell'*RDF store*.

CompetenceExtractionEngine si affida a **StandAloneAnnie** per estrarre dalle pagine le competenze e la struttura di ogni persona.

In figura 24 è possibile vedere uno schema temporale delle chiamate che avvengono quando l'esperto decide di mettere in coda una fase di *crawling*. Dalla pagina viene chiamata la servlet, che passa la chiamata al gestore del comando di *crawling*, che inserisce un nuovo processo in coda. Viene passato indietro alla pagina un messaggio di avvenuto inserimento nella coda. Quando è il turno del processo accodato, viene lanciato un *thread* che sfrutta il corretto

⁹ Vedere la sezione 5.7 per i dettagli sulle classi.

Figura 24: Schema temporale delle chiamate di *crawling*

engine per svolgere le operazioni di *crawling*. La coda nel frattempo rimane libera di gestire l'accodamento di altri processi. Quando l'operazione di *crawling* conclude, il *thread* notifica la coda e poi termina. La coda può quindi lanciare il successivo processo.

5.4.1 Dettagli della coda dei processi

I processi di *crawling*, sia di *keyword* che di competenze, non vengono eseguiti immediatamente appena l'utente clicca sul relativo pulsante. Le richieste vengono messe in attesa in una apposita coda dei processi, ed ogni volta che un processo termina (o nel caso in cui sia vuota) viene lanciato il successivo processo in attesa.

Questa coda è unica in tutto il sistema, nel senso che è condivisa tra tutti i dipartimenti, e il suo scopo è avviare più processi di dipartimenti diversi, nello stesso momento. La coda gestisce la loro mutua esecuzione in quanto sono processi molto onerosi in termini di risorse.

Questa coda è implementata dalle classi **ProcessQueue**, **ProcessTable**, e **ProcessTableRow**¹⁰. La coda fornisce un metodo **newProcess** da richiamare all'atto di aggiungere una nuova chiamata di *crawling*, questo metodo ha come parametro un oggetto di tipo **HttpServletRequest** da dove preleva tutti i parametri necessari per l'esecuzione del processo, come il dipartimento da dove

¹⁰ Vedere la sezione 5.7 per la descrizione dettagliata di queste classi.

è stato lanciato e il tipo di operazione. Con questi parametri crea un oggetto di tipo **ProcessTableRow** e lo accoda a **ProcessTable**.

ProcessTable implementa una coda *fifo* con l'aggiunta di una coda per i processi già conclusi (che vengono visualizzati per riepilogo nel pannello di amministrazione dei dipartimenti), ossia implementano uno schema come quello di figura 25. I processi vengono inseriti in fondo alla coda dei processi in attesa

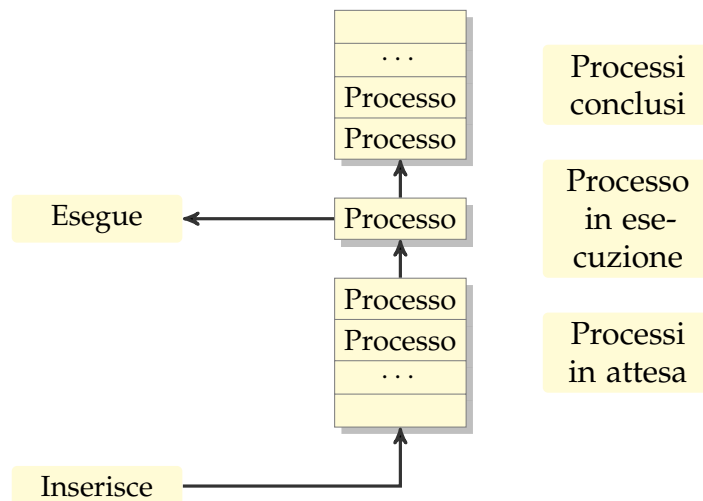


Figura 25: Schema della coda dei processi

sa. Quando un processo termina, viene inserito nei processi conclusi ed il primo processo in attesa passa in esecuzione. La classe **ProcessQueue** implementa **Observer** e viene notificata quando un processo termina, poiché tali processi sono implementati con dei *thread* di classe **CompetenceKeyCrawler**.

5.5 PAGINE *jsp*

L'applicazione presenta tre pagine *JSP*:

- **login.jsp** che presenta all'utente la pagina con il login;
- **userPage.jsp** che presenta all'utente la lista dei dipartimenti con le loro statistiche e i pulsanti per accedere al pannello di amministrazione di ogni dipartimento;
- **index.jsp** che presenta la pagina del pannello di amministrazione di un singolo dipartimento.

Queste pagine sono scritte in *JSP* ossia sono delle pagine in *HTML* integrate da pezzi di codice *Java* che vengono elaborati da *Tomcat* prima di visualizzarli all'utente.

Il sito è dinamico, è presente una iterazione con l'utente. Questa iterazione è realizzata tramite chiamate da *Javascript* con il sistema indicato nella sezione 5.3.

Le pagine ogni secondo lanciano una serie di queste chiamate *HTTP* per controllare se esistono aggiornamenti alla visualizzazione.

5.5.1 Pagina *userPage.jsp*

La pagina, visibile in figura 10 a pagina 78, presenta una lista di dipartimenti creata dinamicamente leggendo la tabella **departments** dal database. Inoltre accanto ad ogni dipartimento viene visualizzata una lista di statistiche reperita dalla tabella **departments_status**.

A seconda dei privilegi che ha l'utente connesso, specificati nella tabella **authorization**, vengono o meno presentati due *link* accanto ai dipartimenti:

leggi per entrare nel pannello di amministrazione di tale dipartimento in modalità di sola visualizzazione;

amministra per entrare sempre nel solito pannello di amministrazione, però con la possibilità di modifica.

Entrambi i *link* portano alla pagina **index.jsp**, usando parametri diversi.

Questa pagina presenta in fondo alla lista dei dipartimenti anche una replica della tabella dei processi completati, in corso, e in coda ricavata interrogando la classe **ProcessQueue**.

5.5.2 Pagina *index.jsp*

La pagina **index.jsp** visibile nelle figure 11 e 12 da pagina 79, è la pagina principale dell'applicazione. Essa mostra il pannello di amministrazione di un dipartimento che a seconda dei parametri può essere solo in visualizzazione o completo. In modalità visualizzazione è possibile solo vedere lo *SKOS* delle competenze e l'elenco delle *keyword*, ma non è possibile apportare modifiche.

La pagina del pannello di amministrazione è aggiornata interamente tramite chiamate *HTTP* da *Javascript*. Uno degli script è richiamato ogni secondo e serve per aggiornare la lista dei processi in basso a destra e per prelevare eventuali messaggi di *log*.

Per il suo layout la pagina usa *jQuery UI Layout*.

5.6 DATABASE

Il database usato si chiama **ieateneo**, e le tabelle principali sono descritte nei seguenti paragrafi.

5.6.1 Tabella **departments**

departments
store:varchar
url:varchar
description:varchar
keywordTable:varchar

È la tabella di indice per tutti i dipartimenti, contiene un campo stringa **store** che è la chiave rappresentativa di un certo dipartimento, la stessa usata in altre tabelle, e anche come parametro all'interno delle chiamate dei metodi nel codice. Il campo **url** rappresenta l'url iniziale da cui far partire il *crawling* delle *keyword* dei differenti dipartimenti. Il campo **description** è la descrizione del dipartimento (ad esempio "dipartimento di sistemi e informatica"). Infine il campo **keywordTable** è il nome della tabella delle *keyword* risultanti dal completamento della fase di *crawling* delle *keyword* di un certo dipartimento.

5.6.2 Tabella **keyword_***

keyword_*
id:int
en_value:varchar
it_value:varchar
frequencies:int
staus:char
type:varchar

È la tabella dove vengono memorizzate le *keyword* che vengono estratte nella prima fase di *crawling* delle *keyword*. Ci sono tante tabelle di questo tipo quanti sono i dipartimenti, e il loro nome è costituito da **keyword_** seguito da un suffisso specifico per ogni dipartimento. L'elenco di tali nomi e il dipartimento a cui queste tabelle sono associate, si trovano nella tabella **departments**.

I campi di questa tabella sono: **id** che è un intero progressivo che funge da chiave primaria per la tabella; **en_value** e **it_value** che sono i valori della *keyword* nelle due lingue attualmente disponibili; **frequencies** che indica il numero di occorrenze della *keyword* in questione all'interno delle pagine del dipartimento; **status** è lo stato della *keyword* che è un carattere che può essere **0** quello di default, **1** se l'esperto ha scelto di inserire tale *keyword* nel *gazetteer* per la seconda fase di *crawling* delle competenze, **2** se l'esperto ha inserito la *keyword* in *black list*; infine **type** indica il tipo di *keyword* estratta, questo valore attualmente è sempre **noun**, ma è prevista in futuro l'implementazione dell'estrazione dei predicati.

5.6.3 Tabella **user**

user
user:varchar
mail:varchar
password:varchar

Si tratta della tabella dei possibili utenti che possono fare *login* nell'applicazione. Nel campo **password** la stringa è codificata con il proprio *hash*.

5.6.4 Tabella **authorization**

authorization
id:int
store:varchar
related_user:varchar
type:char

Assieme alla tabella **user** definisce chi ha i privilegi di amministrazione e per quali dipartimenti. Infatti ogni riga è composta da un **id** che serve da chiave primaria univoca, uno **store** che corrisponde ad uno degli **store** della tabella **departments** dei dipartimenti, **related_user** corrisponde ad uno **user** della tabella **user** degli utenti, e **type** è il tipo di permesso associato al corrispondente utente per il corrispondente dipartimento. Il tipo di permesso **type** è un carattere che può essere uguale ad **a** se ha il permesso di accedere in amministrazione totale, **b** in amministrazione parziale, **c** in sola lettura.

5.6.5 Tabella **recovery_queue**

recovery_queue
timestamp:bigint
store_name:varchar
op_type:varchar

Questa tabella memorizza i processi che sono messi in coda, quelli che possono essere visionati nel riquadro in basso a destra del pannello di amministrazione di un dipartimento. Il campo **timestamp** è il *timestamp*¹¹ di quando è stato lanciato un processo, **store_name** è il nome dello *store*¹², **op_type** indica se il processo è un *crawling* di *keyword* o di competenze. Il suo valore può essere **keyExtraction** o **competenceExtraction**.

Con questa tabella è possibile ripristinare i processi che l'amministratore aveva messo in coda nel caso in cui si verifichi un *crash* dell'applicazione.

¹¹ Il *timestamp* usato da *Java*, ossia il numero di millisecondi trascorsi dalla mezzanotte del primo gennaio del 1970.

¹² lo stesso indicato nella tabella **departments**.

5.6.6 Tabella **departments_status**

departments_status
department_id:varchar
kw_extract_phase:varchar
comp_extract_phase:varchar
n_person:varchar
n_doc:varchar
status_kw_phase:varchar
status_comp_phase:varchar

In questa tabella sono memorizzate tutte le informazioni visibili nelle statistiche di riepilogo nella pagina della lista dei dipartimenti. Il campo **department_id** è il nome del dipartimento¹³. **kw_extract_phase** indica la data e l'ora in formato testuale dell'ultima esecuzione della fase di *crawling* delle *keyword*. **comp_extract_phase** indica la stessa informazione per la fase di *crawling* delle competenze. **n_person** indica in formato testuale il numero di persone per cui è stato fatto il *crawling* e il numero totale di persone (per vedere per quante persone è fallito il *crawling*). **n_doc** indica il numero di pagine del dipartimento che il *crawler* ha processato durante la sua ultima esecuzione. **status_kw_phase** e **status_comp_phase** indica lo stato attuale del processo delle due fasi di *crawling*: può essere **idle** se non è né in esecuzione né in coda, **running** se è in esecuzione, **pending** se è in coda.

5.6.7 Tabella **carriera**

carriera
cognome:varchar
nome:varchar
...
cf_md5:varchar

Questa è una tabella di servizio che viene usata durante la seconda fase di *crawling* delle competenze per ricavare l'*hash md5* del codice fiscale, di ogni persona da usare come *URI* per l'entità della persona nell'*RDF*. Contiene molti campi che per lo scopo del *crawler* non vengono usati. Quelli usati sono: **cognome** che contiene il cognome della persona, **nome** il nome della persona, e **cf_md5** l'*hash md5* del codice fiscale di tale persona.

5.7 CLASSI PRINCIPALI

In questa sezione vengono presentate in breve le principali classi usate dall'applicazione. La lista degli attributi e dei metodi non è esaustiva ma cerca

¹³ come nel campo **store** della tabella *departments*.

di coprire solo ciò che è maggiormente significativo per la comprensione del codice.

Le classi principali del crawler riguardano tutte il progetto **skosServlet**, inoltre la lista di queste viene presentata divisa per *package* (in seguito pacchetto).

5.7.1 Pacchetto **SkosServlet**

È il pacchetto della *servlet*, il punto di accesso a tutto il resto del codice.

Classe **OSIM_SkosServlet**

OSIM_SkosServlet :: HttpServlet : Observer
doGet(request : HttpServletRequest, response : HttpServletResponse)
doPost(request : HttpServletRequest, response : HttpServletResponse)
executeRequest(request : HttpServletRequest, response : HttpServletResponse)
parseResponse(currentState : JSONObject, response : HttpServletResponse)

È la classe principale della *servlet*, i suoi metodi **doGet** e **doPost** rispondono alle chiamate *HTTP* che vengono effettuate dalle pagine *JSP* che devono fare qualche richiesta all'*servlet*. Entrambi questi metodi passano la chiamata al metodo **executeRequest** senza fare altro. Quest'ultimo metodo ha due unici parametri:

- **request** di tipo **HttpServletRequest** che è l'oggetto che racchiude tutte le informazioni della chiamata *http* che è stata fatta dal *Javascript* delle *JSP*. Da questo parametro vengono estratti tutti i campi della *query*, soprattutto **operation** che è il nome dell'operazione richiesta dalla pagina;
- **response** di tipo **HttpServletResponse** serve per poter passare il valore di ritorno della chiamata, formattato come una risposta *HTTP*.

ExecuteRequest ricava **operation** da **request**, usa la classe **OSIM_CommandFactory** per ricavare il giusto gestore di comandi di tipo **OSIM_RequestCommand** e richiama il metodo **run** su tale gestore che ritorna una risposta in formato **JSONObject**.

Per ritornare un valore nella forma richiesta, ossia tramite **response**, viene invocato il metodo **parseResponse** che usa il *JSON* dell'esecuzione del gestore comandi, e il **response** che gli viene passato assieme, per scrivere una risposta in base al valore del **JSONObject**.

Classe **OSIM_CommandFactory**

OSIM_CommandFactory
factory : OSIM_CommandFactory
instanceCommand : ArrayList<String>
keywordCommand : ArrayList<String>
osimCommand : ArrayList<String>
searchCommand : ArrayList<String>
skosCommand : ArrayList<String>
getFactory() : OSIM_CommandFactory
create(operationType : String) : OSIM_RequestCommand

Si tratta di un *singleton* che viene richiamato da **OSIM_SkosServlet** per creare il giusto tipo di gestore delle chiamate *HTTP*. Tali gestori sono tutti ereditati da **OSIM_RequestCommand**.

La classe presenta un attributo **factory** che è l'istanza del *singleton* restituita dal metodo **getFactory**. Oltre a questo ha una serie di vettori di stringhe definiti staticamente contenenti ognuno tipi di operazioni possibili, in base a questi vettori il metodo **create** istanzia il giusto gestore in base al tipo di operazione passato e ritorna tale oggetto.

Classe **OSIM_RequestCommand**

OSIM_RequestCommand
execute(request : HttpServletRequest) : JSONObject

È una *interfaccia* che viene implementata da tutte le classi che descrivono i gestori delle chiamate. L'unico metodo **execute** identifica l'esecuzione di una chiamata, prevede come parametro la richiesta *HTTP* e come valore di ritorno un oggetto *JSON*.

Classe *OSIM_RequestKBCommand*

OSIM_RequestKBCommand :: OSIM_RequestCommand
messenger : Messenger
execute(request : HttpServletRequest) : JSONObject
runTranslateKey(request : HttpServletRequest) : JSONObject
runTranslateCrawl(request : HttpServletRequest) : JSONObject
runGetTranslateKeyCount(request : HttpServletRequest) : JSONObject
runGetTranslateCrawlCount(request : HttpServletRequest) : JSONObject
getNextMessage(request : HttpServletRequest) : JSONObject
getProcessTable(request : HttpServletRequest) : JSONObject
dumpStore(request : HttpServletRequest) : JSONObject
runGetKeysRemover(request : HttpServletRequest) : JSONObject
runStopWordsFilter(request : HttpServletRequest) : JSONObject
runSK0SCleaner(request : HttpServletRequest) : JSONObject

È la classe del gestore delle chiamate dei pulsanti del pannello di controllo di un dipartimento, quelli di figura 13 a pagina 81, e per il prelievo di messaggi per il *log* e per il prelievo della tabella dei processi. Il metodo **execute** controlla quale è il tipo di operazione richiesta e a seconda del quale lancia uno dei corrispondenti metodi:

- **dumpStore** per il *download* dell'*RDF*;
- **runTranslateKey** per il lancio della traduzione delle *keyword* non tradotte;
- **runTranslateCrawl** per il lancio della traduzione delle competenze non tradotte;
- **runGetTranslateKeyCount** per ottenere il numero di *keyword* non tradotte visualizzato sotto il pulsante della traduzione;
- **runGetTranslateKeyCrawl** per ottenere il numero di competenze non tradotte visualizzato sotto il pulsante della traduzione;
- **runKeysRemover** per il lancio della rimozione delle *keyword* non già nel *gazetteer*;
- **runStopWordsFilter** per il lancio della rimozione delle *stop words* dal *gazetteer*;
- **runSK0SCleaner** per il lancio della pulizia dell'*RDF*;
- **getNextMessage** per ottenere il prossimo messaggio del *log*;
- **getProcessTable** per ottenere la tabella dei processi in esecuzione, in attesa, ed eseguiti.

L'attributo **messenger** è un riferimento al *singleton* che si occupa di gestire i messaggi del log.

Ci sono altre operazioni che vengono gestite all'interno del metodo **execute**:

- il lancio del *crawler* di *keyword*, viene accodato un processo al gestore della coda **ProcessQueue**¹⁴;
- il lancio del *crawler* delle competenze, come il precedente viene accodato un processo;
- la cancellazione di un processo in coda, o la pulizia della visualizzazione di un processo completato, viene richiamato il metodo **deleteProcess** della classe **ProcessQueue**.

Classe **OSIM_KeywordRequestCommand**

OSIM_KeywordRequestCommand :: OSIM_RequestCommand
keywordManager : OSIM_KeywordManager
maxRowsForPage : int
execute(request : HttpServletRequest) : JSONObject
getTablePage(request : HttpServletRequest) : JSONObject
submit(request : HttpServletRequest) : JSONObject
countTotalPage(request : HttpServletRequest) : String
getKeyTableName(store : String) : String
getflipLanguageCB(lang : String) : String

È il gestore delle operazioni collegate con la tabella delle *keyword* visibile nella figura 12 a pagina 80. Il suo **execute** e i suoi metodi si occupano di gestire tutte le operazioni di visualizzazione della tabella, cambio pagina, aggiornamento della tabella, cambio lingua di una riga.

¹⁴ Vedere il funzionamento della coda in sezione 5.4.1.

Classe **OSIM_SkosRequestCommand**

OSIM_SkosRequestCommand :: OSIM_RequestCommand
skosManager : OSIM_SkosManager
execute(request : HttpServletRequest) : JSONObject
addAlternativeLabel(request : HttpServletRequest) : JSONObject
createNode(request : HttpServletRequest) : JSONObject
getSkosChildren(request : HttpServletRequest) : JSONObject
loadConcept(request : HttpServletRequest) : String
moveNode(request : HttpServletRequest) : String
removeNode(request : HttpServletRequest) : String
renameNode(request : HttpServletRequest) : String
saveConcept(request : HttpServletRequest) : String
searchNode(request : HttpServletRequest) : String
searchNodeByUri(request : HttpServletRequest) : String
translate(request : HttpServletRequest) : String

È il gestore delle operazioni collegate con lo *Skos* visibile nella figura 11 a pagina 79. Questa classe gestisce tutte le operazioni di gestione dello *Skos*, a partire dalla visualizzazione, alle operazioni di modifica come la creazione di gerarchie tra competenze, oppure la creazione di nuovi concetti.

5.7.2 Pacchetto **ProcessQueuing**

Le classi di questo pacchetto si occupano di gestire la coda dei processi di *crawling*, sia delle *keyword* che delle competenze. Queste due operazioni non vengono gestite immediatamente dal gestore dei comandi, ma questi demanda il compito alle classi di questo pacchetto.

Per la descrizione del funzionamento della coda dei processi, riferirsi alla sezione 5.4.1.

Classe **processQueue**

ProcessQueue :: Observer
crawler : CompetenceKeyCrawler
table : ProcessTable
newProcess(request : HttpServletRequest)
launchNextProcess()
update(o : Observable, arg : Object)
deleteProcess(timestamp : String)
deleteOldProcesses()
getProcessTable(loggedUser : String) : String

È la classe principale del gestore della coda dei processi, quella che si interfaccia all'esterno. È sviluppata come un *singleton*, e prevede gli attributi:

- **crawler** è il thread che si occupa di effettuare il *crawling*, questa classe è in osservazione su tale *thread* in modo che possa venir notificata quando un processo di *crawling* conclude;
- **table** è l'istanza della classe che effettivamente implementa la coda dei processi.

I metodi sono:

- **newProcess** per poter aggiungere un nuovo processo in coda, come parametro ha direttamente la richiesta *HTTP* dal quale estrae i campi utili;
- **update** è il metodo che viene richiamato quando un *thread* completa la sua esecuzione;
- **launchNextProcess** si occupa di far scorrere la coda dei processi e lanciare il *thread* di *crawling* sul successivo processo;
- **deleteProcess** si occupa di eliminare dalla lista il processo che è stato inserito nel dato *timestamp* passato, infatti per ogni nuovo processo inserito viene memorizzato anche il *timestamp* con l'obiettivo di poterlo poi cancellare usando tale *timestamp* come chiave;
- **deleteOldProcesses** elimina dalla lista tutti i processi che sono conclusi.

Questa classe si occupa anche di mantenere aggiornata la tabella **recovery_queue** dove viene memorizzato lo stato corrente della coda per poterlo ripristinare quando necessario.

Classe **ProcessTable**

ProcessTable
past : LinkedList<ProcessTableRow>
present : ProcessTableRow
past : LinkedList<ProcessTableRow>
put(newValues : ProcessTableRow)
get() : ProcessTableRow
getAll() : LinkedList<ProcessTableRow>
step()
delete(timestamp : String) : ProcessTableRow
deletePast()

Questa classe implementa fisicamente la coda dei processi; la coda viene mantenuta divisa in tre parti, corrispondenti ai tre attributi della classe:

- **past** è una lista di elementi di tipo **ProcessTableRow** che sono i processi già completati;

- **present** è un unico elemento di tipo **ProcessTableRow** e rappresenta il processo che attualmente è in esecuzione;
- **future** è una lista di elementi dei processi in attesa di essere eseguiti.

I metodi che la classe fornisce sono quelli che vengono richiamati dalla **ProcessQueue** per svolgere le proprie funzioni. Questi sono:

- **put** inserisce un metodo in coda alla lista **future**;
- **get** ritorna l'elemento attualmente in **present** senza modificare niente;
- **getAll** ritorna una lista costituita dalla concatenazione di **past**, **present** e **future**;
- **step** fa avanzare di un passo le code, sposta **present** in testa a **past**, e sposta l'elemento in testa a **future** in **present**;
- **delete** elimina l'elemento che ha *timestamp* pari a quello passato;
- **deletePast** svuota la lista **past**.

Classe **ProcessTableRow**

ProcessTable
contextDirectory : String
note : String
operationType : String
state : String
storeName : String
timestamp : String
getContextDirectory() : String
setContextDirectory(newContextDirectory : String)
getNote() : String
setNote(newNote : String)
...

È la classe che identifica un elemento della tabella dei processi, essa ha tutti gli attributi che servono per poter lanciare un *thread* di *crawling* e in più ha l'attributo **timestamp** per poterlo identificare nell'eliminazione e l'attributo **state** che indica in che stato è tra: in attesa, in esecuzione, e eseguito.

5.7.3 Pacchetto **AteneoCrawler**

È il pacchetto che contiene la classe che svolge la funzione di *thread* per i processi di *crawling* e tutte le classi necessarie allo svolgimento della fase di *crawling* delle competenze. Contiene inoltre le classi necessarie per interfacciarsi con *GATE embedded*.

Classe **CompetenceKeyCrawler**

CompetenceKeyCrawler :: Observable : Runnable
engine : KeywordExtractionEngine
competenceEngine : CompetenceExtractionEngine
homePage : String
currentDatastore : String
servletTempDir : String
operationCode : String
setDatastore (storename : String)
setContextDirectory (dir : String)
setOperationMode (code : int)
run ()
getStatus () : String

Questa classe implementa il *thread* principale che viene lanciato al momento dell'esecuzione di una fase di *crawling*. Viene usato questo *thread* per entrambe le fasi, quella delle *keyword* e quella delle competenze; l'attributo **operationCode** da settare con il metodo **setOperationMode** serve proprio ad indicare il tipo di *crawling* che deve fare al momento dell'esecuzione.

Gli attributi **homePage**, **currentDatastore**, **servletTempDir**, e i metodi che li modificano **setDatastore**, e **setContextDirectory**, servono per indicare l'ambiente di esecuzione di questo *thread*. Tali attributi hanno la seguente funzione:

- **homePage** indica la pagina iniziale da dove deve partire il *crawling*;
- **currentDatastore** è il codice del dipartimento dove è stato lanciato il *crawling*;
- **servletTempDir** è la directory di esecuzione della servlet, questa serve per poter sapere dove salvare i *file* temporanei necessari durante il *crawling*.

Implementando un *thread*, questa classe dispone del metodo **run** che viene eseguito al momento del lancio. Questo metodo in base al **operationCode** indicato, si avvale dell'uso della classe **KeywordExtractionEngine** per eseguire il *crawling* delle *keyword* o della classe **CompetenceExtractionEngine** per il *crawling* delle competenze. In entrambi i casi vengono passati all'*engine* i valori degli attributi necessari per conoscere l'ambiente in cui deve operare.

Classe **CompetenceExtractionEngine**

CompetenceExtractionEngine
startPage : String
currentDatastore : String
homePage : String
annieEngine : StandAloneAnnie
run() : String
getAteneoPages(annie:StandAloneAnnie,contextPath:String):ArrayList<String>
saveDataStore(urimap : Map<String, String>, contextPath : String)

Questa classe implementa il processo principale durante il *crawling* di competenze. La classe **CompetenceKeyCrawler** se riconosce che è richiesta una fase di *crawling* di competenze lancia il metodo **run** di questa classe, dopo averla istanziata con i corretti attributi.

La classe presenta un attributo **annieEngine** di classe **StandAloneAnnie** che usa per richiamare la piattaforma *GATE* e prelevare le annotazioni trovate. La *pipe* di *gate* usata è **competenceExtraction.xgapp**.

Il metodo *run* svolge i seguenti passaggi:

1. usa **annieEngine** per annotare i links nella pagina della lista del personale **startPage**;
2. usa la funzione **buildPersonMap** con le annotazioni trovate per estrarre le features di tali annotazioni e ricavare una mappa "nome persona"-link alla pagina della persona";
3. per ogni persona della mappa usa il metodo **analyze** della classe **PersonAnalyzer** per ricavare gli oggetti di classe **UnifiOntologyInstance**, che rappresentano le parti di triple da inserire nell'*RDF*, riguardanti:
 - la persona;
 - le materie insegnate dalla persona, se presenti;
 - i corsi di laurea nei quali insegna la persona, se presenti;
4. sempre per la solita persona usa il metodo **serialize** della classe **OSIM_RdfSerializer** per salvare nell'*RDF store* le triple riguardanti la persona.

La classe contiene anche due metodi che non vengono sfruttati in fase di *crawling* di competenze, ma in fase di *crawling* di *keyword*. Tali metodi sono:

- **getAteneoPages** ritorna un *array* contenente i *link* a tutte le pagine di un dipartimento;

- **saveDataStore** richiamata durante l'esecuzione della precedente, si occupa di salvare su disco i documenti corrispondenti a tutte le pagine di un dipartimento nel formato di *GATE* che preserva le annotazioni.

Quest'ultima funzione è necessaria per una funzionalità del *crawler* future.

Classe **StandAloneAnnie**

```

StandAloneAnnie :: CorpusListener
computeAnnotationOnCorpus(documentUrls:ArrayList<String>,
    annotationsType:HashSet<String>):Set<Annotation>

```

Questa classe è quella che si occupa di interfacciarsi con il *framework GATE*, il suo metodo **computeAnnotationOnCorpus** viene chiamato ogni volta che è necessario reperire delle annotazioni su un documento. **documentUrls** contiene la lista dei documenti da analizzare (nel codice solitamente ha dimensione uno), **annotationsType** la lista delle annotazioni significative da prelevare (anche questo nel codice ha solitamente dimensione uno). Il valore di ritorno è una lista di **Annotation** che è la classe di *GATE* che rappresenta una annotazione.

Il costruttore di questa classe accetta un parametro *intero* che indica quale *pipe* applicare. Tale parametro ha i seguenti significati:

- **0** viene usata **langDetector.xgapp**;
- **1** viene usata **it.xgapp**;
- **2** viene usata **en.xgapp**;
- **3** viene usata **competenceExtraction.xgapp**.

Classe **PersonAnalyzer**

```

PersonAnalyzer
annie : StandAloneAnnie
person : UnifiOntologyInstance
courses : ArrayList<UnifiOntologyInstance>
currentCV : ArrayList<UnifiOntologyInstance>
analyze(fullName : String, personLink : String)
analyzePersonCourses(courseLink : String)
analyzePersonSinglePage(pageLink : String, courseUri : String)
getPerson() : UnifiOntologyInstance
getCourses() : ArrayList<UnifiOntologyInstance>
getCurrentCV() : ArrayList<UnifiOntologyInstance>

```

Questa classe svolge l'analisi di una singola persona in fase di *crawling* delle competenze. Il suo attributo **annie** viene usato per interfacciarsi a *GATE*. Gli

attributi **person**, **courses**, **currentCV** sono usati per memorizzare i risultati dell'analisi. Questi sono di tipo **UnifiOntologyInstance** che rappresenta una istanza di una classe *RDFS*. In questi vengono memorizzate le istanze della persona, degli eventuali corsi insegnati da essa, e delle competenze trovate.

Il metodo **analyze** viene richiamato dall'*engine* implementato dalla classe **CompetenceExtractionEngine** e crea l'istanza **person**. Il metodo **analyzePersonCourse** permette di analizzare le pagine dei corsi della persona e crea le istanze **courses**. Il metodo **analyzePersonSinglePage** viene richiamato da entrambi i precedenti metodi e crea le istanze **currentCV** delle competenze. Se a quest'ultimo metodo viene passato **courseUri**, annota nelle competenze il corso da dove queste sono state estratte, questo permette di istanziare in fase di serializzazione la proprietà **skos:subject**¹⁵.

I metodi **get** servono per poter prelevare le istanze create.

Classe **UnifiOntologyInstance**

UnifiOntologyInstance
uri : String
types : ArrayList<String>
subject : String
literalFeatures : HashMap<String,List<String>
setUri (newUri : String)
setType (t : String)
setSubject (s : String)
getUri () : String
getTypes () : ArrayList<String>
getSubject () : String
addFeatures (key : String, value : String)
getFeatures () : HashMap<String,List<String>

È la classe che rappresenta una istanza di una classe *RDFS*, gli attributi hanno il seguente significato:

- **uri** è l'*URI* della risorsa;
- **types** è un array di classi *RDFS* che questa risorsa istanzia;
- **subject** contiene l'uri di una risorsa di un corso per poter creare la proprietà **skos:subject** in fase di serializzazione¹⁶;
- **literalFeatures** è un **HashMap** dove vengono memorizzate le proprietà della risorsa. Vengono memorizzate in una forma contratta in cui ad ogni chiave, che è un predicato *RDF*, corrisponde una lista di valori che sono i soggetti;

¹⁵ Vedere sezione 5.1.

¹⁶ Vedere sezione 5.1.

Il metodo **addFeatures** permette di aggiungere una singola proprietà alla risorsa, viene controllato se la chiave esiste già ed in tal caso viene aggiunto il valore alla lista di tale chiave; in caso contrario viene creata la nuova chiave con associata una lista unitaria con il valore.

Gli altri metodi sono auto esplicativi.

Classe **OSIM_RdfSerializer**

OSIM_RdfSerializer
currentDatastore : String
serialize (person:UnifiOntologyInstance, competences:ArrayList<UnifiOntologyInstance>, courses:ArrayList<UnifiOntologyInstance>
checkConsistenceOfCompetence (person:UnifiOntologyInstance, competences:ArrayList<UnifiOntologyInstance>)
writeInstance (person:UnifiOntologyInstance)
writeCompetenceRelationship (person:UnifiOntologyInstance, predicateUri:String, competences:ArrayList<UnifiOntologyInstance>)
writeCompetenceRelationship (person:UnifiOntologyInstance, predicateUri:String, complements:ArrayList<UnifiOntologyInstance>)

È la classe che si occupa della serializzazione sull'*RDF Store* delle istanze trovate con la classe **PersonAnalyzer**. Il metodo **serialize** viene chiamato dall'*engine* **CompetenceExtractionEngine** dopo l'analisi della persona. In questo metodo vengono effettuate le operazioni:

1. viene chiamato **checkConsistenceOfCompetence** che controlla se le competenze trovate esistono già per la persona, in caso positivo aggiorna il valore del numero di occorrenze;
2. viene chiamato **writeInstance** che scrive sull'*RDF Store* le istanze della persona, di ogni competenza, e di ogni corso;
3. viene chiamato **writeCompetenceRelationship** che si occupa di scrivere le relazioni tra le istanze delle competenze e la persona; si occupa anche di creare il *bnode*;
4. viene chiamato **writeRelationship** che si occupa di scrivere le relazioni tra la persona e i suoi insegnamenti.

5.7.4 Pacchetto **KeyExtraction**

Contiene le classi necessarie allo svolgimento della fase di *crawling* delle *keyword*.

Classe **KeywordExtractionEngine**

KeywordExtractionEngine
annie : StandAloneAnnie
enGrammar : StandAloneAnnie
itGrammar : StandAloneAnnie
datastore : String
extractKey() : Boolean
preprocessAteneo() : Boolean
split() : Boolean
mergeByLanguage() : Boolean
extraction() : Boolean
extractNoun() : Boolean
updateDatabase(datastore : String) : Boolean

È la classe che implementa l'engine della fase di estrazione di *keyword*. In figura 26 è possibile vedere uno schema delle chiamate che avvengono durante l'uso di questa classe. In questo schema è possibile vedere sia le chiamate interne alla classe, che quelle esterne.

Il metodo **preprocessAteneo** avvia la fase di preparazione delle pagine. Vengono eseguiti in ordine:

1. **split** che appoggiandosi alla classe **ChunkSplitter** prima recupera tutte le pagine del dipartimento, poi estrae le frasi e le mette nei *file* temporanei;
2. **mergeByLanguage** che appoggiandosi alla classe **LanguageDetector** accorpa i *file* delle frasi per lingua.

I metodi **extraction** e **extractionNoun** eseguono l'estrazione dei sostantivi dai *file* tramite *GATE*.

Infine il metodo **updateDatabase** esegue la memorizzazione nel *database* delle *keyword* estratte.

Classe **ChunkSplitter**

ChunkSplitter
annie : StandAloneAnnie
split(documentUrl : String, folderPath : String)

Il suo metodo **split** estrae dalla pagina passata le frasi tramite *GATE* e le memorizza in *file* nella cartella **folderPath**.

Classe **LanguageDetector**

LanguageDetector
annie : StandAloneAnnie
detectAndMerge(documentsDir : String)

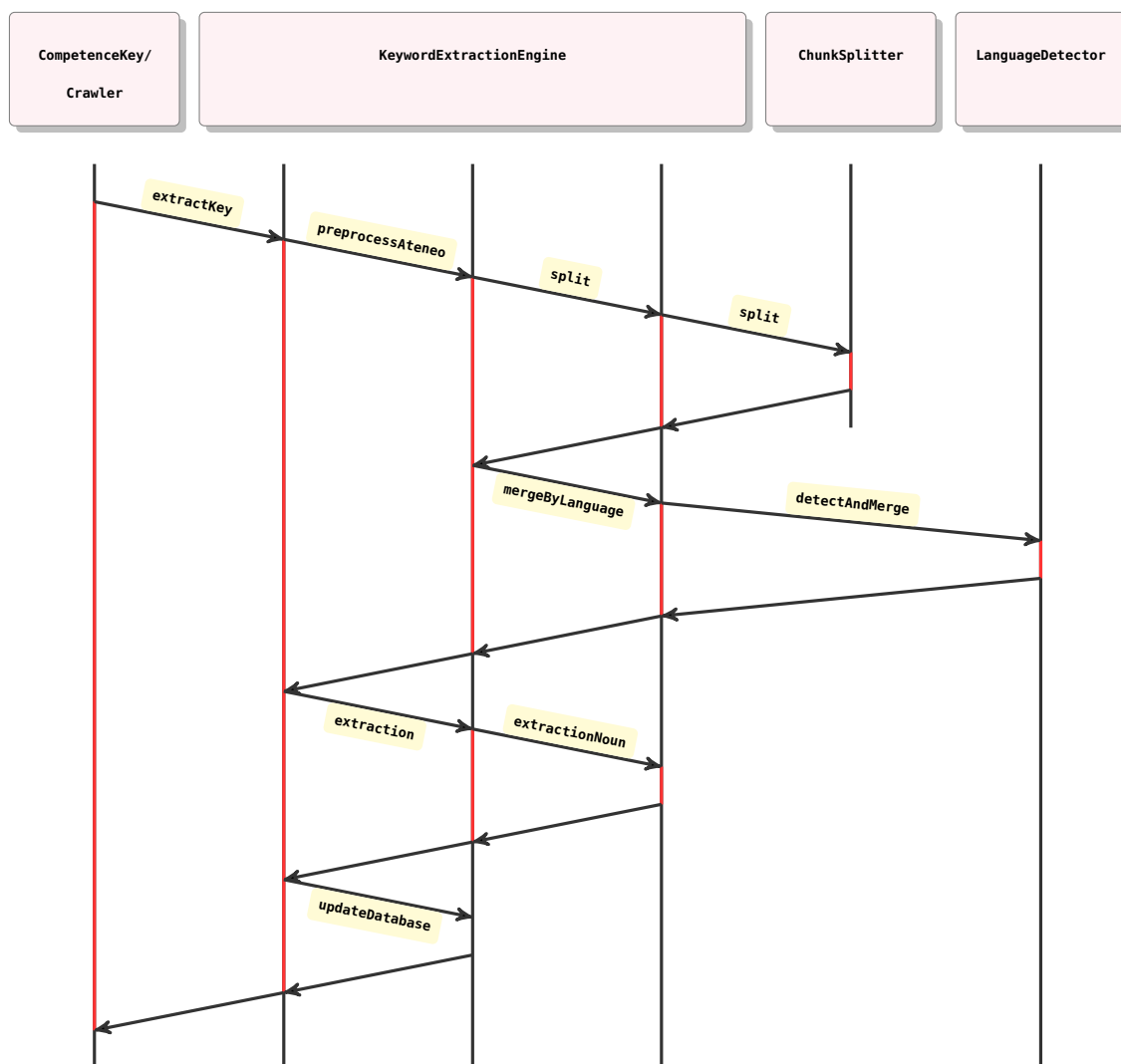


Figura 26: Schema temporale delle chiamate di **KeywordExtractionEngine**

Il suo metodo **detectAndMerge** concatena tutti i *file* presenti nella cartella passata in tre *file* dividendoli per lingua.

5.7.5 Pacchetto **Translation**

Contiene le classi necessarie ad interfacciarsi con il servizio di *Google Translate* per poter fare le traduzioni.

In figura 27 è possibile vedere un esempio di iterazione tra le classi **SingleTranslation**, **Authenticator**, **Translator**, la *cache* delle traduzioni nel db, e il servizio esterno di *Google Translate*. Nell'esempio si ipotizza il caso in

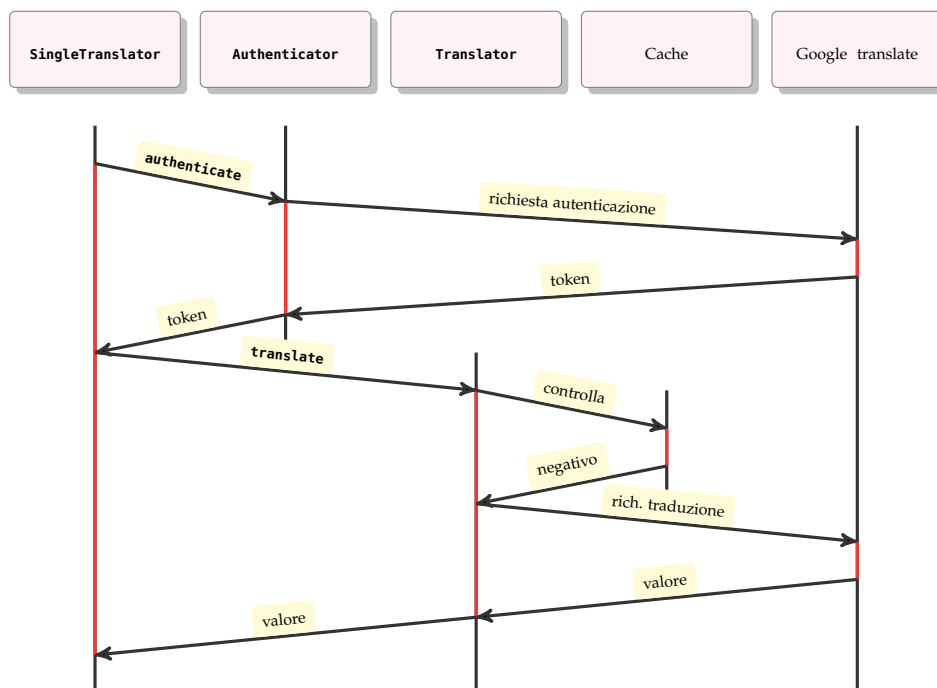


Figura 27: Schema temporale del funzionamento delle traduzioni

cui il controllo nella cache dia esito negativo, inoltre le chiamate al metodo **authenticate** e **translate** non ritornino direttamente il valore del *token* e del valore tradotto come mostrato nello schema. Queste tornano un *booleano* che indica l'esito dell'operazione, e se positivo è possibile prelevare i valori tramite chiamate ai metodi **getToken** e **getLast**. Questo passaggio è stato omesso nella figura per rendere più conciso lo schema.

5.7.6 Pacchetto *Messaging*

Contiene un'unica classe responsabile della conservazione dei messaggi di log da visualizzare nelle pagine di amministrazione dei dipartimenti.

Classe *Messenger*

Messenger
queues : Map<String,Queue<String>
println (mess : String, datastore : String)
getln (user : String, datastore : String)

Implementa un produttore-consumatore; i processi quando necessitano di comunicare un output scrivono i messaggi nella coda, e le pagine *JSP* leggono periodicamente tali messaggi dalla coda.

L'attributo **queues** implementa una lista di coppie:

- codice dipartimento;
- coda di messaggi.

Il metodo **println** scrive il messaggio indicato nella coda associata al dipartimento indicato. Il metodo **getln** restituisce il primo messaggio della coda associata al dipartimento indicato, solo se l'utente indicato ha i permessi.

CONSIDERAZIONI FINALI E SVILUPPI FUTURI

L'applicazione presenta notevoli possibilità di sviluppo. Preliminarmente provvederei ad una pulizia generale del codice che presenta diversi punti dove non è ottimizzato e che risente del fatto che è stata sviluppata da più persone in momenti diversi non sempre in modo coerente. Sono inoltre presenti molti frammenti di codice morto e classi non usate.

Spesso viene usato *GATE* in punti dove non è molto appropriato, e le *pipe* non sono ottimizzate per l'esecuzione differenziata. Inoltre non sembra molto appropriato l'utilizzo di *GATE* per il *natural language processing* in lingua italiana. Ho ricevuto notizie che per questo ultimo punto sono già state valutate delle soluzioni che verranno applicate.

Un'altro punto su cui cercherei di intervenire è la struttura dell'ontologia, questa non sfrutta a pieno le possibilità offerte dal web semantico, in particolar modo quelle offerte da *OWL*.

Per quanto riguarda le possibili estensioni dell'applicazione, cercherei di risolvere uno dei maggiori problemi attualmente presenti: le fasi di *crawling* dipendono fortemente dalla struttura del sito sul quale vengono lanciate. Questo è un problema complesso e non di facile risoluzione implica che, ogni volta che il sito sul quale viene lanciato il *crawling* cambia struttura, è necessario mettere mano in maniera pesante al codice.

Per mitigare questo problema propongo l'adozione di una fase di preparazione delle pagine tramite trasformazioni *XSLT* come in figura 28.

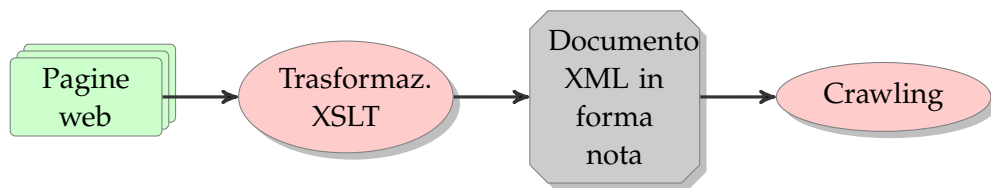


Figura 28: Proposta di pre-fase

L'idea è di usare le potenzialità delle trasformazioni *XSLT* per trasformare l'insieme delle pagine web desiderate in un documento, o insieme di documenti, in formato *XML* con una struttura ben definita a priori, di fatto si tratterebbe di creare uno *stylesheet* per le pagine del sito. Quindi il *crawler* verrebbe lanciato su questo documento *XML* e non più direttamente sulle pagine. I vantaggi sono che, in caso di modifica alla struttura del sito, o anche nel caso si voglia applicare il crawling ad altri siti, le modifiche necessarie per

rendere nuovamente operativo il programma si riducono alla modifica del solo schema di trasformazione *XSLT*, senza toccare minimamente il codice *Java* dell'applicazione.

In tal senso ho anche provveduto a effettuare degli esperimenti per valutarne la fattibilità, e questi hanno avuto risultati positivi.

Un altro punto a favore di questa idea è che la sua attuazione potrebbe aprire altre vie per il futuro di questa applicazione, ad esempio è possibile immaginare la creazione di un *tool* che aiuti nello sviluppo dello *stylesheet XSLT* applicato ad un certo sito, rendendo l'operazione di aggiornamento della trasformazione più semplice che non scrivendo da zero lo schema.

Per i dettagli sulle trasformazioni *XSLT* vedere la sezione 1.4.2.

BIBLIOGRAFIA

- [1] Antoniou Grigorios, Harmelen Frank. *A semantic web primer*. Cambridge MIT press, 2004.
- [2] Dean Allemang, Jim Hendler. *SEMANTIC WEB for the WORKING ONTOLOGIST*. Morgan Kaufmann, 2008.
- [3] W3C. *Raccomandazioni del w3c relative a XML e alle tecnologie relative*.
“<http://www.w3.org/TR/xml/>”,
“<http://www.w3.org/TR/xml-names/>”,
“<http://www.w3.org/TR/xmlschema-0/>”,
“<http://www.w3.org/TR/xmlschema-1/>”,
“<http://www.w3.org/TR/xmlschema-2/>”,
“<http://www.w3.org/TR/xslt/>”,
“<http://www.w3.org/TR/xpath/>”,
“<http://www.w3.org/TR/xsl/>”,
“<http://www.w3.org/TR/xml-infoset/>”.
- [4] W3C. *Raccomandazioni del w3c relative a RDF*.
“<http://www.w3.org/TR/rdf-syntax-grammar/>”,
“<http://www.w3.org/TR/rdf-primer/>”,
“<http://www.w3.org/TR/rdf-concepts/>”,
“<http://www.w3.org/TR/rdf-syntax-grammar/>”,
“<http://www.w3.org/TR/rdf-nt/>”,
“<http://www.w3.org/TR/rdf-schema/>”,
“<http://www.w3.org/TR/rdf-testcases/>”.
- [5] W3C. *Raccomandazioni del w3c relative a OWL*.
“<http://www.w3.org/TR/owl-guide/>”,
“<http://www.w3.org/TR/owl-features/>”.
- [6] W3C. *Raccomandazioni del w3c relative a SKOS*.
“<http://www.w3.org/TR/skos-primer/>”,
“<http://www.w3.org/TR/skos-reference/>”,
“<http://www.w3.org/2004/02/skos/>”.
- [7] D. Brickley, L. Miller. *FOAF Vocabulary Specification*.
“<http://xmlns.com/foaf/spec/>”.
- [8] T. Berners Lee, et al. *STD 66, RFC 3986, Uniform Resource Identifier (URI): Generic Syntax*.
“<http://tools.ietf.org/html/std66>”, January 2005.

- [9] D. Crocker, P. Overell. *STD 68, RFC 5234, Augmented BNF for Syntax Specifications: ABNF*.
“<http://tools.ietf.org/html/std68>”, January 2008.
- [10] *jQuery UI Layout*.
“<http://layout.jquery-dev.net/>”
- [11] Hamish Cunningham and Diana Maynard and Kalina Bontcheva and Valentin Tablan and Niraj Aswani and Ian Roberts and Genevieve Gorrell and Adam Funk and Angus Roberts and Danica Damljanovic and Thomas Heitz and Mark A. Greenwood and Horacio Saggion and Johann Petrak and Yaoyong Li and Wim Peters. *Text Processing with GATE (Version 6)*. 2011.
“<http://tinyurl.com/gatebook>”
- [12] Appelt, Douglas E. and Onyshkevych, Boyan. *The common pattern specification language*, 1998.
- [13] Hamish Cunningham and Diana Maynard and Kalina Bontcheva and Valentin Tablan. *GATE: A Framework and Graphical Development Environment for Robust NLP Tools and Applications*. 2002.