



SAPIENZA  
UNIVERSITÀ DI ROMA

## Rappresentazione JavaScript di entità OWL e loro proprietà in linguaggio Graphol

Dipartimento di Ingegneria informatica, automatica e gestionale "Antonio Ruberti"

Corso di Laurea in Ingegneria dei Sistemi Informatici

Candidato

Valerio Longo

Matricola 1655653

Relatore

Prof. Maurizio Lenzerini

Anno Accademico 2018/2019

Tesi non ancora discussa

---

**Rappresentazione JavaScript di entità OWL e loro proprietà in linguaggio Graphol**

Tesi di Laurea. Sapienza – Università di Roma

© 2018 Valerio Longo. Tutti i diritti riservati

Questa tesi è stata composta con  $\text{\LaTeX}$  e la classe Sapthesis.

Versione: 27 novembre 2018

Email dell'autore: [longo.1655653@studenti.uniroma1.it](mailto:longo.1655653@studenti.uniroma1.it)

# Indice

<b>Introduzione</b>	<b>i</b>
<b>1 Parte 1 - Linguaggi e strumenti utilizzati</b>	<b>1</b>
1.1 OWL . . . . .	1
1.2 OWLAPI . . . . .	2
1.3 Graphol . . . . .	2
<b>2 Parte 2 - Da .owl a .json</b>	<b>5</b>
2.1 Main . . . . .	6
2.2 Il metodo LavoraConcetti . . . . .	7
2.3 Il metodo LavoraRuoli . . . . .	8
2.4 Il metodo LavoraAttributi . . . . .	9
<b>3 Parte 3 - Visualizzare i dati con Cytoscape.js</b>	<b>11</b>
3.1 index.html . . . . .	12
3.2 cytotools.js . . . . .	13



# Introduzione

È sempre stata prerogativa dell'uomo la voglia di concettualizzare la realtà che lo circonda. Il desiderio di trovare i corretti criteri di esistenza di determinate entità partendo da un linguaggio formale è espresso nel tema dell'**ontologia**. In ambito informatico, per ontologia si intende una struttura dati gerarchica che contiene tutte le entità rilevanti, le relazioni esistenti fra di esse, le regole, gli assiomi ed i vincoli specifici di un dato dominio. Oggi lo standard W3C per il linguaggio delle ontologie è **OWL** (Ontology Web Language). A causa della sua non facile sintassi però, è molto difficile da analizzare se non da utenti molto esperti. Per questo motivo nasce l'esigenza di creare linguaggi visuali come **Graphol**, con l'obiettivo di eliminare le ostilità sintattiche attraverso l'utilizzo di simbologie facilitandone così l'uso ai neofiti.

L'obiettivo di questo progetto è quello di dare all'utente la possibilità di scegliere una tra le varie entità di un'ontologia e vederla disegnata con tutti i suoi ruoli, attributi, superclassi, sottoclassi e loro relative funzionalità. Lo stile con il quale viene visualizzato il tutto è quello del linguaggio visuale Graphol. Per prima cosa viene fatto partire un programma Java che esamina un file Owl attraverso le OWLAPI e ne genera uno nuovo di estensione json. Dopodiché, su browser, un file HTML tramite JavaScript e JQuery genera un'interfaccia di upload per il file json e la selezione di una tra le entità presenti al suo interno. Il layout è generato attraverso Cytoscape.js, una libreria JavaScript per la visualizzazione di grafi.

Questa relazione è divisa in tre parti. La prima discute in maniera introduttiva OWL, Graphol e gli strumenti utilizzati, la seconda è incentrata sulla parte Java del progetto e l'ultima invece analizza dettagliatamente il codice HTML e Javascript. Prima di iniziare però c'è da chiarire che molti dei tools utilizzati per questo progetto non erano mai stati studiati durante il percorso di studi, pertanto se ci fossero imprecisioni nelle modalità di realizzazione o in qualche parte del codice queste sarebbero nient'altro che ingenuità dettate dalla poca esperienza.



## Capitolo 1

# Parte 1 - Linguaggi e strumenti utilizzati

### 1.1 OWL

OWL (da Web Ontology Language) è una famiglia di linguaggi utilizzati per rappresentare esplicitamente il significato e la semantica di termini e relazioni tra gli stessi. Esistono quindi varie versioni del linguaggio, che differiscono molto tra di loro. In questo progetto sono stati utilizzati file con sintassi funzionale OWL 2. L'utilizzo di OWL ha per obiettivo quello di rappresentare, ragionare, dedurre e collegare informazioni appartenenti allo stesso contesto. Segue quindi che analizzare informazioni provenienti da contesti differenti porterebbe a conclusioni totalmente errate ed è quindi necessario delimitare chiaramente il loro dominio di interesse. La rappresentazione di tale dominio è detta **ontologia**. Al suo interno vi gravitano **entità, espressioni ed assiomi**.

Le **entità** sono gli elementi atomici di un ontologia. Sono identificati da un IRI e possono essere classi, individui o proprietà. Una classe è per esempio p:Uomo, la quale può essere usata per rappresentare l'insieme di tutti gli uomini a livello estensionale. Un individuo invece è p:Lucio, usato per rappresentare a livello intensionale un uomo chiamato "Lucio". Una proprietà (o relazione) può essere p:padreDi, usata per rappresentare il ruolo di padre.

Le **espressioni** invece rappresentano complesse nozioni interne al dominio di interesse. Per esempio, un'espressione descrive un insieme di uomini accomunati dalla caratteristica di essere padri.

Infine, gli **assiomi** sono affermazioni prese indiscutibilmente come vere ed utilizzate come premesse per ragionamenti più complessi. Un esempio di assioma può essere: la classe p:Uomo è una sottoclasse della classe p:Persona.



















Queste tre categorie sintattiche sono usate per esprimere la parte logica delle ontologie OWL. Queste sono interpretate attraverso una precisa semantica che permette di estrarre deduzioni molto utili. Per esempio, se un individuo p:Lucio è istanza della classe p:Uomo, e p:Uomo è sottoclasse di p:Persona, allora da ciò si può dedurre che p:Lucio è istanza anche di p:Persona. È quindi facilmente intuibile che serva una metodologia per estrarre informazioni da un'ontologia e con Java, tramite le OWL API, questo compito riesce piuttosto bene.

## 1.2 OWLAPI



OWL API è un API Java open source per la manipolazione e l'analisi di ontologie OWL. Esso è compatibile con moltissimi ragionatori, fondamentali per questo progetto in quanto servono per **dedurre** assiomi. In questo progetto è stato utilizzato HermiT come ragionatore nella sua versione 1.3.8.500. OWL API è fortemente legato da dipendenze, perciò è stato praticamente d'obbligo l'utilizzo di Maven come software di gestione per la parte Java del progetto.

## 1.3 Graphol

Graphol è un linguaggio visuale per le ontologie. Un'ontologia viene considerata come un grafo con **nod**i (ruoli, concetti, attributi) e **archi** (collegamenti). Questi vengono disegnati con uno stile molto semplice e intuitivo che permette di evitare l'analisi di complesse strutture sintattiche.

Symbol	Name	Symbol	Name	Symbol	Name
	Concept node		Attribute node		OR operator
	Role node		Functional attribute		NOT operator
	Functional role		Value-domain node		Inverse operator
	Inverse functional role		Domain restriction node		NOR operator
	Functional and inverse functional role		Range restriction node		Chain operator
	Individual		AND operator		OneOf operator

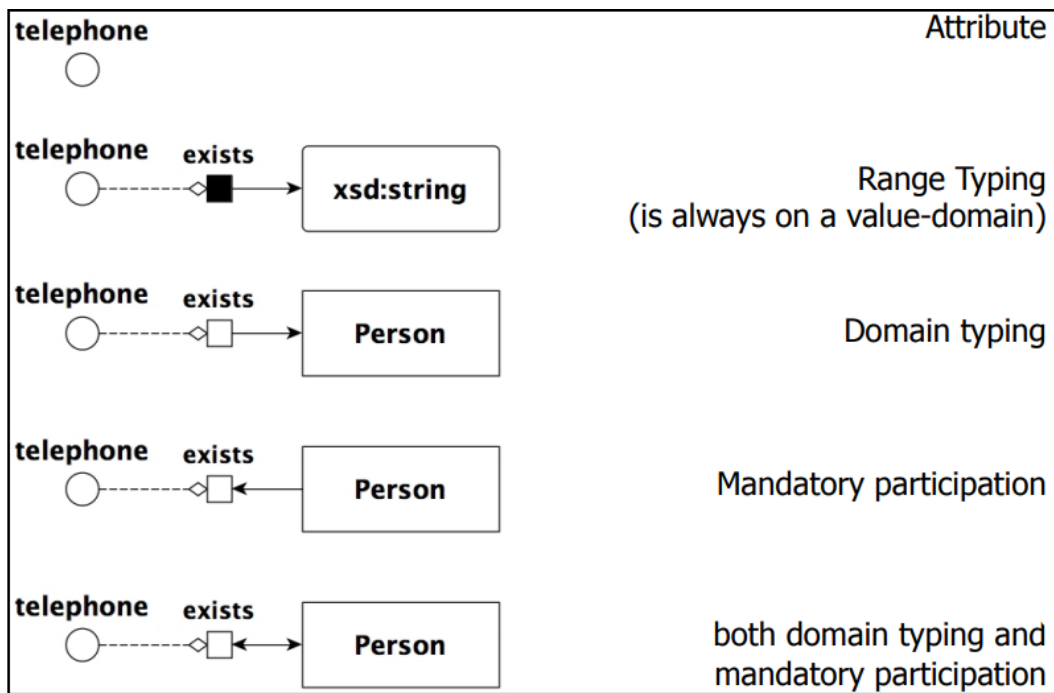
  

Symbol	Name	Symbol	Name
	Inclusion edge		Input edge

**Figura 1.1.** Alcuni simboli di Graphol, ispirati a quelli dei diagrammi ER

I versi degli archi possono avere diverse configurazioni a seconda del tipo di partecipazione di ogni nodo. Una partecipazione obbligatoria (Mandatory participation) impone che ogni oggetto nelle istanze di un concetto (atomico o complesso) è anche nelle istanze del dominio (oppure del range) di un ruolo. Una partecipazione di tipizzazione (che può essere di dominio o di range) specifica il "tipo" degli oggetti che vengono istanziati nel dominio (o nel range) di un ruolo.





**Figura 1.2.** diverse configurazioni di partecipazione tra un concetto ed un attributo, il discorso è simile anche tra concetti e ruoli

Ora che sono state chiarite le basi e gli intenti di questo progetto, andiamo ad analizzarlo più approfonditamente. Per evitare di rendere questa relazione troppo prolissa, verrà riportato il codice solo per le varie inizializzazioni, mentre per quanto riguarda i singoli algoritmi di estrapolazione, analisi e visualizzazione verranno solamente descritti e/o saranno riportati solo i procedimenti più interessanti.



## Capitolo 2

# Parte 2 - Da .owl a .json

L'IDE sul quale è stata sviluppata la parte Java del progetto è NetBeans. Come già accennato, le OWL API sono tanto utili quanto legate a dipendenze, a causa di ciò è stato necessario l'utilizzo di Maven. Inoltre è stata di aiuto *simple-JSON*, una libreria per la creazione e gestione del formato JSON. L'input di questo programma è un file owl scelto dall'utente e l'output è un file JSON che ha la seguente struttura:

```
{
  "Concepts": [
    {"CONCETTO1": {"Super_Concepts": ["SUPERCONCETTI"]}},
    {"CONCETTO2": {
      "Mandatory_Attributes": ["ATTRIBUTI_MANDATORI"],
      "Sub_Concepts": [["GRUPPO_DI_SOTTOCONCETTI1"], ["GRUPPO_DI_SOTTOCONCETTI2"]],
      "Optional_Roles": ["RUOLI_OPZIONALI"]}
    },
    {"CONCETTO3": {"Mandatory_Roles": ["RUOLI_MANDATORI"]}}
  ],
  "Roles": [
    {"RUOLO1": {
      "Domain": ["CONCETTO_DOMINIO"],
      "Range": ["CONCETTO_RANGE"],
      "ObjectProperty": ["Functional"]
    }
  ],
  "Attributes": [
    {"ATTRIBUTO1": {
      "Domain": ["CONCETTODOMINIO"],
      "ObjectProperty": "Functional",
      "Mandatory_in": ["CONCETTO"]
    }
  ]
}
```

La classe `GenerateJson.java` è la sola e unica di questa parte del progetto. È composta dal `main` e da tre metodi *LavoraConcetti*, *LavoraRuoli*, *LavoraAttributi*. Come si può intuire dal nome, ognuno di questi tre metodi "lavora" sul singolo gruppo di entità dell'ontologia, estraendo informazioni dall'input.

## 2.1 Main

Prima di tutto bisogna inizializzare il programma e fornirgli il file owl.

```
JFileChooser chooser = new JFileChooser();
FileNameExtensionFilter filter = new FileNameExtensionFilter("OWL FILES", "owl");
chooser.setFileFilter(filter);
chooser.setMultiSelectionEnabled(false);
chooser.setFileSelectionMode(JFileChooser.FILES_AND_DIRECTORIES);
int retval = chooser.showOpenDialog(new JFrame("GenerateJson"));
```

Utilizzando il componente `JFileChooser` viene creato un `JFrame` che permette all'utente di scegliere l'input. Viene permesso di selezionare solamente file con estensione owl. Il ritorno di `chooser.showOpenDialog()` viene catturato.

```
File file;
if (retval == JFileChooser.APPROVE_OPTION) {
    try {
        file = chooser.getSelectedFile();
        OWLOntologyManager manager = OWLManager.createOWLOntologyManager();
        OWLOntology o = manager.loadOntologyFromOntologyDocument(file);

        JSONObject obj = new JSONObject();

        //INIZIO ELABORAZIONE DATI PER I CONCETTI
        JSONArray arrayconcetti = new JSONArray();
        LavoraConcetti(arrayconcetti,o);
        obj.put("Concepts", arrayconcetti);
        //FINE ELABORAZIONE DATI PER I CONCETTI

        //INIZIO ELABORAZIONE DATI PER I RUOLI
        JSONArray arrayruoli = new JSONArray();
        LavoraRuoli(arrayruoli,o);
        obj.put("Roles", arrayruoli);
        //FINE ELABORAZIONE DATI PER I RUOLI

        //INIZIO ELABORAZIONE DATI PER GLI ATTRIBUTI
        JSONArray arrayattributi = new JSONArray();
        LavoraAttributi(arrayattributi,o);
        obj.put("Attributes", arrayattributi);
        //FINE ELABORAZIONE DATI PER GLI ATTRIBUTI
```

```

        //SCRITTURA DEL RISULTATO SU FILE SU DEKSTOP.
        FileWriter localfile = new FileWriter("/Users/theta/Desktop/prova.json")
        localfile.write(obj.toJSONString());
        System.out.println("Successfully Copied JSON Object to File...");
        System.out.println("\nJSON Object: " + obj);

    }
    catch (Exception ex) {
        JOptionPane.showMessageDialog(
            new JFrame("Exception"), "Error opening file!", "Error!", retval
        );
        System.exit(1);
    }
}
if(retval == JFileChooser.CANCEL_OPTION){
    JOptionPane.showMessageDialog(
        new JFrame("operazione annullata"), "operazione annullata", "Error!", retval
    );
    System.exit(1);
}
}

```

In sostanza, nel caso di esito positivo nella scelta del file di input, viene creato *JSONObject obj* e *OWLOntology o*. Quest'ultimo non è altro che il file owl convertito in ontologia grazie alle OWLAPI ed è l'elemento sul quale verranno eseguite le operazioni di analisi, infatti è argomento di tutti e tre i metodi statici. L'oggetto *obj* invece, non è altro che il contenuto del file (all'inizio vuoto) di output. Esso verrà riempito in tre step, grazie ai tre metodi statici. Per ogni gruppo di entità (concetti, ruoli e attributi) viene creato un array che verrà poi aggiunto ad *obj* al termine della chiamata di ogni metodo. Infine *obj* viene scritto su un nuovo file .json. Nel caso ci fosse un problema nell'accettazione dell'input oppure un annullamento durante la scelta del file, il programma termina.

## 2.2 Il metodo LavoraConcetti

```
private static void LavoraConcetti(JSONArray arrayconcetti, OWLOntology o)
```

Questo metodo statico ha come obiettivo quello di riempire *arrayconcetti* con tutti i concetti presenti in *o* ed i loro relativi collegamenti. Per fare ciò è necessario un ragionatore:

```

OWLReasonerFactory rf = new org.semanticweb.HermiT.ReasonerFactory();
OWLReasoner r = rf.createReasoner(o);
// RIEMPIO LISTA CON STRINGHE DI TUTTI I CONCETTI
List<String> listanomiConcetti = new LinkedList<String>();
List<OWLClass> listanomiOWLConcetti = new LinkedList<OWLClass>();

```

```

for (OWLClass cls : o.getClassesInSignature()){
    listanomiConcetti.add(cls.getIRI().getFragment());
    listanomiOWLConcetti.add(cls);
}

```

Viene creato il ragionatore, vengono create le liste con all'interno tutti i concetti. Adesso si opera su ogni elemento di una delle due liste.

```

for (int i = 0; i < listanomiConcetti.size(); i++) {
    JSONObject oggettoConcetto = new JSONObject();
    JSONObject infoOggettoConcetto = new JSONObject();
    String nomeConcetto = listanomiConcetti.get(i);
    //GESTIONE ATTRIBUTI
    List<String> MandAttrlist= new LinkedList<String>();
    //...ALGORITMO DI ESTRAPOLAZIONE DEGLI ATTRIBUTI ATTRAVERSO IL RAGIONATORE
    if(!MandAttrlist.isEmpty()){
        infoOggettoConcetto.put("Mandatory_Attributes", MandAttrlist);
    }

    //GESTIONE RUOLI MANDATORI
    List<String> MandRolelist = new LinkedList<String>();
    //...ALGORITMO DI ESTRAPOLAZIONE DEGLI RUOLI MANDATORI..
    if(!MandRolelist.isEmpty()){
        infoOggettoConcetto.put("Mandatory_Roles", MandRolelist);
    }
    //continua....
}

```

In sostanza, per ogni concetto vengono create delle liste per ogni tipo di dato (superclassi, sottoclassi, attributi, ruoli mandatori, ruoli opzionali ecc..). Successivamente, attraverso il ragionatore, si estraggono le informazioni per ogni tipo di dato e vengono inserite all'interno della relativa lista. Se la lista ha almeno un elemento, deve essere inserita in *infoOggettoConcetto* insieme alla descrizione del tipo di dato.

```

oggettoConcetto.put(nomeConcetto, infoOggettoConcetto);
arrayconcetti.add(oggettoConcetto);

```

Così facendo viene fatto side-effect e viene riempito l'array che è stato dato come input nel main.

Appena terminata l'esecuzione del metodo, arrayconcetti viene inserito in obj.

## 2.3 Il metodo LavoraRuoli

```

private static void LavoraRuoli(JSONArray arrayruoli, OWLOntology o)

```

Ciò che viene fatto qui è molto simile a quello svolto da *LavoraConcetti()*, cambia solamente il contenuto delle liste per rispettare la formattazione del file json vista ad inizio capitolo.

```
List<String> listanomiRuoli = new LinkedList<String>();
List<OWLObjectProperty> listaowlruoli = new LinkedList<OWLObjectProperty>();
for (OWLObjectProperty cls : o.getObjectPropertiesInSignature()){
    listaowlruoli.add(cls);
    listanomiRuoli.add(cls.getIRI().getFragment());
}
```

Ciò che effettivamente cambia è il soggetto. Infatti qui vengono analizzati tutti gli elementi dell'ontologia con proprietà *getObjectPropertiesInSignature()*.

La struttura è sempre la stessa, per ogni ruolo corrente, viene creata una serie di liste, ognuna con al suo interno le informazioni utili alla sua rappresentazione Graphol. Una volta raggruppate le informazioni in liste, vengono inserite nell'*JSONObject infoOggettoRuolo*. Successivamente

```
oggettoRuolo.put(nomeRuolo, infoOggettoRuolo);
arrayruoli.add(oggettoRuolo);
```

A fine chiamata, arrayruoli (contente tutti i ruoli dell'ontologia) viene inserito in obj.

## 2.4 Il metodo LavoraAttributi

```
private static void LavoraAttributi(JSONArray arrayattributi, OWLOntology o)

for (OWLDataProperty cls : o.getDataPropertiesInSignature()){
    listaowlattributi.add(cls);
    listanomiAttributi.add(cls.getIRI().getFragment());
}
```

Questa volta, la lista di attributi viene fornita tramite *o.getDataPropertiesInSignature()*, ed i suoi elementi verranno analizzati prima per vedere se sono funzionali (se sì, in che modo), poi per vedere quali sono i domini, i range e per ultimo se sono mandatori. Poi, una volta inserite tutte le liste in *infoOggettoAttributo*, come al solito:

```
oggettoAttributo.put(nomeAttributo, infoOggettoAttributo);
arrayattributi.add(oggettoAttributo);
```

Alla fine dell'esecuzione di questo metodo, verranno inseriti tramite arrayattributi tutti gli attributi in obj, il quale sarà finalmente completo e andrà ad essere il contenuto del file json di output.

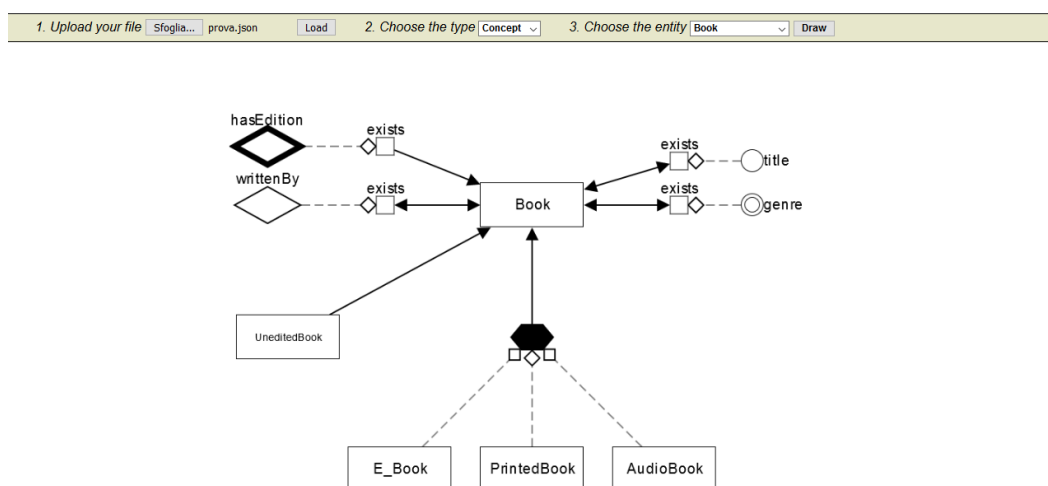




## Capitolo 3

# Parte 3 - Visualizzare i dati con Cytoscape.js

Ora che è stato generato un file che raggruppa in modo abbastanza semplice le informazioni su un'ontologia, verrà utilizzato HTML e Javascript per poter riuscire a visualizzare in Graphol il tutto.



**Figura 3.1.** Screenshot della demo eseguita su Firefox 63.0.1.

Come si vede dalla figura, Per prima cosa si sceglie il file json desiderato, una volta cliccato su *Load*, viene analizzato tramite JavaScript e viene proposto all'utente di scegliere il tipo di dato (concetto, ruolo o attributo) da visualizzare con tutte le sue relazioni. Una volta scelto il tipo di dato, si passa a scegliere l'entità specifica. Una volta finito il setup, se si clicca su *Draw* viene disegnato il grafo. Il layout è molto simile per tutti e tre i tipi di dato ed è così composto:

Se il tipo di dato è un **concetto**, esso avrà nella parte superiore i superconcetti, in quella inferiore i sottoconcetti, a sinistra i ruoli e a destra gli attributi ai quali è collegato.

Se invece il tipo di dato è un **ruolo**, avrà a sinistra tutti i concetti facenti parte del

dominio, a destra quelli del codominio, superiormente e inferiormente super ruoli e sotto ruoli.

Infine, se il dato è un **attributo**, a destra e sinistra andranno domini e range.

### 3.1 index.html

Per la parte di setup è stato sufficiente utilizzare una `<form>` con diversi `<button>` legati a diversi metodi. Se premuto, il `<button>` *Load* fa eseguire il metodo `LoadFile()`. Grazie a `window.FileReader` (non disponibile in alcuni tipi di browser), viene permesso di caricare il file e leggerlo. Come ultimo comando, `LoadFile()` chiama un'altro metodo, `initialize()`, il quale ha il compito di parsare il file json e compilare il form con i tipi di dato.

Più precisamente, vengono create tre liste, una per tipo di dato. All'interno di ognuna vengono aggiunte le entità così da avere una suddivisione per tipo. Dopodiché, attraverso JQuery, quando l'utente sceglie di vedere un tipo di entità, il form viene compilato solo con gli elementi della lista relativa a quel tipo.

```
$( 'select[name = "ConOrRolOrAtt"]' ).live("change",function(){
    if( $( 'select[name = "ConOrRolOrAtt"] option:selected' ).text() == "Role" ) {
        tipo = "starRole";
        $( 'select[name = "lista"]' ).empty();
        $( 'select[name = "lista"]' ).append(new Option("---",0));
        $.each(listaruoli,function(index,elem){
            $( 'select[name = "lista"]' ).append(new Option(elem,index));
        });
    }
    //..
    //...
}
```

Questo snippet di codice fa compilare l'elemento `<select [name = "lista"]>` con tutti gli elementi di *listaruoli*, la lista dentro la quale sono contenuti tutti i ruoli dell'ontologia. Ciò avviene se è stato scelto "Role" all'interno di `<select[name = "ConOrRolOrAtt"]>`, quindi l'utente desidera scegliere uno tra tutti i ruoli dell'ontologia. La stessa cosa viene fatta con attributi e concetti.

Infine, se cliccato, `<button>` *Draw* chiama il metodo `go()`, il quale ha il compito di gestire la "Fase Cytoscape.js" del progetto.

```
function go(){
    var name;
    var cy;
    //INIZIALIZZAZIONE DELL'ELEMENTO CY
    cy = cytoscape({
        container: document.getElementById('cyo'),
        style: stile
    });
    //PARSING DEL JSON
    readTextFile(name, jparse);
```

```

//INIZIO GESTIONE DEL LAYOUT
cy.layout({
  name:'preset',
}).run();

if(tipo == 'starConcept') GrapholyConcept();
else if(tipo == 'starRole') GrapholyRole();
else if(tipo == 'starAttribute') GrapholyAttribute();

cy.fit(cy.$id(soggetto),280);

// GESTIONE NODI FAKE E LABEL

FakeAttrGen();
FakeRoleGen();
FakeManager();
fixSizeLabel()
}

```

Per prima cosa viene inizializzata la variabile `cy`, l'elemento rappresentante l'intero grafo. Successivamente si passa a parsare il file json, tramite `jpase()` (passato come argomento alla funzione `readTextFile`). Come si vedrà nella sezione successiva, `jpase()` è il vero e proprio manager del grafo, poiché crea i nodi e va cercare tutte le relazioni tra gli elementi presi in considerazione.

A seconda del contenuto della variabile `tipo`, è chiamato un metodo diverso di Grapholy, che dispone i nodi creati con il layout descritto ad inizio capitolo. In seguito viene centrato bene l'elemento protagonista tramite `cy.fit()`.

C'è da dire che anche se Cytoscape.js è stato davvero fondamentale per questo progetto, ha però delle limitazioni. Infatti l'ultima parte del metodo `go()` crea degli eventuali nodi falsi al fine di disegnare poligoni particolari (come ad esempio ruoli e attributi funzionali) e gestisce il loro movimento. Tutti questi metodi sono raggruppati in `cytotools.js`, un piccolo gruppo di metodi creati con l'intento di allineare Cytoscape.js alle regole di Graphol. Come ultimo passaggio, verranno analizzate le parti interessanti di questi metodi.

## 3.2 cytotools.js

I metodi all'interno di questo file posso essere suddivisi in tipi.

I metodi di tipo **Grapholy**, come anche scritto anche precedentemente, dispongono i nodi creati tramite un layout definito dal tipo di entità. Esistono quindi tre metodi Grapholy: `GrapholyConcept()`, `GrapholyRole()` e `GrapholyAttribute()`. Questi hanno in comune la struttura.

Per esempio in `GrapholyConcept()` si ha:

```
var attributes = cy.$('node[type = "simpleAttribute"],[type = "complexAttribute"]');
```

```

var roles = cy.$(
    'node[type = "simpleRole"],
    [type = "doubleSimpleRole"],
    [type = "complexRole"],
    [type = "doubleComplexRole"]'
);
var subouos = cy.$('node[type = "subOuo"],
    [type = "subAndornot"],
    [type = "simpleSubConcept"]'
);
var superconcepts = cy.$('node[type = "superConcept"]');

var star = cy.$id(soggetto);
var starx = star.position('x');
var stary = star.position('y');
//...

```

Tramite i selettori di cytoscape.js vengono prese e create le variabili raffiguranti tutti i nodi da disporre sullo schermo. Inoltre, vengono prese le coordinate del concetto protagonista.

Successivamente, per ogni gruppo di nodi (in questo caso attributi, ruoli, super e sotto concetti) verrà applicato un algoritmo di posizionamento basato sulle variabili *starx* e *stary*. Questi algoritmi sono stati pensati per rendere tutto il più "pulito" possibile. Quindi nel caso di numerose entità dello stesso tipo, il disegno complessivo apparirà più "largo" ma più chiaro.

Oltre ai *Grapholy*, ci sono le funzioni del tipo **Generator**. Queste, seguendo lo stile del nodo, generano il simbolo graphol con tutto ciò che è annesso (simboli di unione, disjoint ecc..). Di questo tipo sono i metodi: *RoleGenerator()*, *ConceptGenerator()*, *SuperConceptGenerator()*, *SubConceptGenerator()*, *FakeRoleGen()*, *FakeAttrGen()*, *AttrRoleGenerator()*.

Si prenda come esempio la struttura di quest'ultimo:

```

function AttrRoleGenerator( idSrc, idTargt, arrstyle, colorExists,typ){
//idSrc è l'id del nodo sorgente
//idTargt è l'id del nodo d'arrivo
//arrstyle è lo stile dell'arco tra l'exists e il target (mandatorio, opzionale)
//colorExists è bianco se dominio e nero se range
//typ è il tipo del nodo sorgente
  cy.add([
    { group:"nodes",data: { id: idSrc, type: typ }}, //creo nodo giusto
    { group:"nodes",data: { id: 'exists' +idSrc+idTargt, type: colorExists }}, /

    {
      group:"edges",
      data: { // creo edge nodo--->exists
        id: 'edgeExists' +idSrc+idTargt,

```

```

        source: idSrc,
        target: 'exists' +idSrc+idTargt,
        type: 'descendant' }
    },
    {
        group:"edges",
        data: { //creo edge exists--->target
        id: 'edgetarget' +idSrc+idTargt,
        source: 'exists' +idSrc+idTargt,
        target: idTargt,
        type: arrstyle }
    }
  ]);
}

```

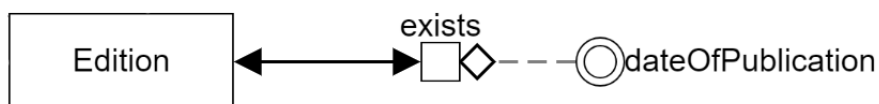
Questi tipi di funzioni utilizzano quasi esclusivamente cytoscape.js. In quest'esempio infatti vengono aggiunti 2 nodi e relativi archi. Come si può notare, questi metodi sono abbastanza slegati dalle regole Graphol a causa appunto del disallineamento tra i linguaggi. Se utilizzati correttamente però, portano ad ottimi risultati. Per esempio, una sua chiamata è

```

AttrRoleGenerator(
  "DateofPublication","Edition",'doublenormal',
  'existsW','complexAttribute'
);

//DateofPublication è l'id dell' attributo
//Edition è il concetto protagonista
//'doublenormal' indica il tipo di arco
//'existsW' dominio
//complexAttribute vuol dire che è un attributo funzionale

```



**Figura 3.2.** Cosa viene generato dalla chiamata.

Tutti gli altri metodi di questo tipo sono collegati dall'uso massiccio di cytoscape.js e quindi nella creazione di nodi particolari.

Infine, come ultimo tipo di funzioni, ci sono quelle **manageriali** come *FakeManager()*,

*readTextFile()* e la più importante *jparsed()*. Come già accennato, e come suggerisce il nome, *jparsed* ha il compito di parsare ed analizzare il file json scegliendo quando chiamare un metodo e con quali parametri. È così strutturato:

```
function jparsed(text){
    var data = JSON.parse(text);

    if(tipo == 'starConcept'){.....}

    else if(tipo == 'starRole'){.....}

    else {.....}
}
```

La variabile *data* contiene l'intero file json. Per ogni tipo di entità scelta dall'utente, viene applicato un solo e determinato algoritmo di analisi del file e relativa creazione e costruzione di nodi. Tutti i tre casi hanno alcuni aspetti in comune, come per esempio la creazione del nodo centrale e il pattern per l'analisi. Il legame che c'è tra questi tre casi è paragonabile a quello che c'è tra i tre metodi Java *LavoraConcetti*, *LavoraRuoli*, *LavoraAttributi* esaminati nei capitoli precedenti. Difatti se prima venivano analizzati i concetti, i ruoli e gli attributi del file owl, ora vengono analizzati con formattazione json.

Si ricorda che nel caso si avesse interesse ad esaminare più approfonditamente i singoli algoritmi, si invita ad esaminare il codice sorgente dotato di commenti utili per capire al meglio il lavoro svolto.

# Bibliografia

- [1] *OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax (Second Edition)* [www.w3.org/TR/owl2-syntax](http://www.w3.org/TR/owl2-syntax)
- [2] [github.com/owlcjs/owlapi](https://github.com/owlcjs/owlapi)
- [3] *Graphol, a visual language for ontologies* [www.obdasystems.com/graphol](http://www.obdasystems.com/graphol)
- [4] *Cytoscape.js, graph theory / network library for analysis and visualisation* [js.cytoscape.org/](http://js.cytoscape.org/)
- [5] [maven.apache.org/](http://maven.apache.org/)
- [6] *Eloquent JavaScript 3rd edition*
- [7] *Repository del progetto* [github.com/1655653/Conceptoscape](https://github.com/1655653/Conceptoscape)

