1. **INTRODUZIONE**

Il tema d’anno svolto consiste nell’apportare modifiche nel progetto open source WEBVOWL (MIT license).

Le modifiche vertono sull’aggiunta delle feature di modifica dell’ontologia visualizzata, rendendo possibile quindi l’inserimento, la cancellazione o la modifica di un elemento.

La struttura del progetto WEBOWL è, nel suo insieme, ovviamente orientata alla visualizzazione dei dati e non alla loro modifica e, pertanto, è stato necessario sviluppare moduli per la modifica ed integrarli in quello che è il progetto originale. È stata trattata una restrizione degli aspetti di un’ontologia che è possibile modificare.

1. **WEBVOWL**

WEBVOWL è una applicazione web atta alla sola visualizzazione di ontologie. Implementa la visualizzazione delle notazioni per le ontologie OWL attraverso un grafo orientato che rappresenta l’ontologia stessa.

E’ bene specificare che l’applicativo lavora convertendo i file owl in formato vowl (formato accettato). La conversione avviene automaticamente in caso di IRI o URI mentre la visualizzazione di una ontologia in locale è possibile convertendo dapprima il file attraverso il software owl2vowl, applicativo java, scaricabile dal sito web di webvowl.

Webvolw, ad oggi, non permette la visualizzazione di tutti i costrutti di OWL2 o la combinazione di essi.

**2.1 Struttura del progetto**

L’applicativo è del tipo client side (utilizzo del solo JAVASCRIPT) ed è modularizzato mediante node.js.

Nei sorgenti, oltre ai file di stile css, sono presenti diversi script javascript suddivisi nelle cartelle webvowl e app. Nella cartella app sono presenti gli script per la gestione della UI (sidebar, menu, etc) mentre nella cartella webvowl sono presenti gli script relativi all’interpretazione e alla visualizzazione dell’ontologia.

Importanti tra questi sono:

* app/js/app.js: si occupa di inizializzare la pagina e di caricare l’ontologia.
* webvowl/js/graph.js: si occupa della rappresentazione dell’ontologia attraverso un grafo.
* Webvowl/js/parser.js: si occupa del parsing dell’ontologia e della relativa rappresentazione interna.
* Gruntfile.js: si occupa di importare nella cartella deploy tutti i file esterni e di supporto necessari al funzionamento dell’applicativo.
* Index.html: corpo della pagina web di webvowl

Nella cartella deploy sono contenuti i sorgenti compilati.

1. **MODIFICHE APPORTATE**

Essendo il progetto originale orientato alla sola visualizzazione di ontologie si è reso necessario apportare modifiche ai sorgenti originali e sviluppare ulteriori script esterni che abilitassero anche la modifica.

Per assicurare una facilità di integrazione delle modifiche sviluppate in future versioni, si è cercato di limitare i cambiamenti ai file originali, preferendo lo sviluppo di moduli esterni.

In particolare, dei file originali sono stati modificati:

* Graph.js: si è inserito uno script che abilitasse la cattura del click del tasto destro sul nodo e visualizzazione del relativo menu contestuale (righe 810-857). È stato inoltre aggiunta la referenza al nodo cliccato dall’utente per la sua relativa elaborazione.
* Index.html: si è modificata la struttura dell’idex.html per la disabilitazione del menu contestuale di default del browser, integrato jQuery e la cattura della posizione del click del mouse (194-206).
* App.js: esternalizzazione della funzione “loadOntologyFromText” in modo tale sia accessibile dai moduli da noi creati (143-199).
* Gruntfile.js: sono state aggiunti tra gli “import” i file .css, .js e .html da noi creati (69,75,76).
* Parser.js: data la natura differente della elaborazione dei dati che è necessario effettuare nel caso della sola visualizzazione di una ontologia piuttosto che anche della sua modifica, sì è scelto di sviluppare un modulo esterno (myparser.js) che trattasse i dati in modo tale da rendere possibile anche la modifica degli stessi.

Sono stati sviluppati i seguenti moduli:

* Myparser.js: script che si occupa di effettuare un re-parsing dell’ontologia e implementa i metodi per modificarla ed il reperimento dei dati dalla stessa.
* Page.js: script atto alla creazione della pagina popup, mostrata all’utente, per la modifica (insert, edit, delete di un nodo).
* pop.html: pagina popup da mostrare all’utente.
* Css vari per lo styling della pagina.

**3.1 GRAPH.JS**

Lo script utilizzato (presente in src/js), è un plugin per la creazione di menu contestuali ampiamente configurabile e per scopi multipli, chiamato contextMenu.js. Per il suo funzionamento si affida alla libreria jQuery che è stata importata anch’essa nel progetto e viene esclusivamente utilizzata per il corretto funzionamento dello script menu contestuale.

Per quanto riguarda l’integrazione di tale menù all’interno del progetto, all’interno del file graph.js, che si occupa della gestione del grafo visualizzato, abbiamo individuato la sezione di codice inerente i “click listenerer” dei nodi del grafo ed integrato il codice summenzionato.

In questo modo è stato possibile abilitare il click del tasto destro sui singoli nodi e far apparire il relativo menu contestuale. In particolare, la modifica consiste nell’inserimento di una funzione (myContextMenu()) nel cui interno vi è dichiarato un array di oggetti(menu), uno per ogni voce che si vuole visualizzare.

Ogni oggetto si compone dei campi name, title e fun. Il campo fun è stato modellato di modo tale che al click su di esso, venga visualizzata la finestra popup relativa alla modalità add, edit e delete di un nodo.

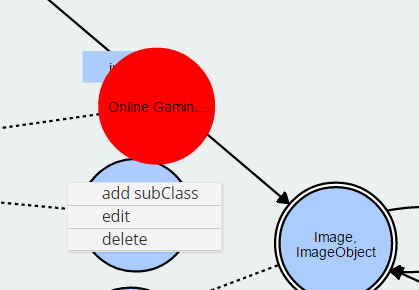


Figura - click destro sul nodo Online Gaming

**3.2 APP.JS**

Questo script, come detto precedentemente, si occupa dell’inizializzazione della pagina.

Cosa più importante è il compito di caricare il file JSON, parsarlo e validarlo, quindi passarlo alle strutture che si occuperanno della visualizzazione dello stesso.

Si è reso necessario creare una copia dell’istanza del modulo app e l’esternalizzazione della funzione loadOntologyFromText da richiamarsi all’interno dei nostri script. Questa necessità è dovuta al fatto che un import del modulo app all’interno degli script causava degli import ciclici che causavano degli errori a runtime. La funzione loadOntologyFromText è stata essenziale al fine di caricare, all’interno del sistema, l’ontologia a cui sono state apportate le modifiche effettuate dall’utente. Sfruttando questa funzione è stato possibile ridisegnare il grafo dell’ontologia ad ogni modifica effettuata, evitando che il file venisse salvato e ricaricato manualmente. Questa funzione è stata richiamata passando come parametro il json aggiornato in formato di testo, in modo tale che il sistema lo validasse e convertisse secondo la propria rappresentazione interna.

**3.3 PAGE.JS**

Script da noi creato che si occupa della creazione della finestra popup in base al click sul menu contestuale.

Le voci del menu, come raffigurato in fig.1, sono 3:

* Add subClass;
* Edit;
* Delete.

In base alla voce selezionata la visualizzazione è differente. Selezionando add subClass si abilita la modalità insert di un nuovo nodo (di default figlio del nodo su cui si è cliccato). Viene visualizzato l’id del nuovo nodo che si sta creando mostrando all’utente il form da popolare con i dati relativi al nodo.

Selezionando, invece, la voce edit, innanzitutto vengono reperite tutte le informazioni sul nodo selezionato (myparser) e quindi visualizzate in un form identico a quello dell’inserimento, mentre selezionando delete, viene caricato una finestra popup che ci permette di eliminare il nodo selezionato.

Ebbene specificare che, data la natura di webvowl orientata alla sola visualizzazione, in caso di nodi equivalenti, questi ultimi vengono rappresentati come un unico nodo e viene meno la possibilità di cliccare sul singolo nodo equivalente.

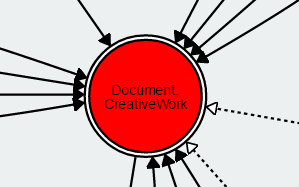
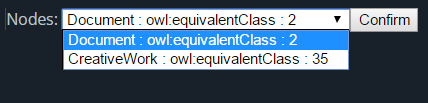


Figura - rappresentazione grafica di due nodi equivalenti

Come è possibile vedere in figura due, il nodo Document è equivalente al nodo CreativeWork. Cliccando su tale nodo vengono reperite le sole informazioni del nodo Document rendendo impossibile, di fatto, editare e cancellare direttamente CreativeWork.

Per poter operare su CreativeWork sarebbe stato necessario dapprima scindere la relazione di equivalenza (in modo tale che fossero rappresentati come singoli nodi) e quindi operare sul nodo di interesse.

Per ovviare a questa inconvenienza, all’interno dello script page, viene dapprima analizzato il nodo e, se in modalità di editing o cancellazione, si è in presenza di un nodo che abbia equivalenze verrà mostrata una select che permetta all’utente di specificare il nodo su cui operare(fig.3-4).



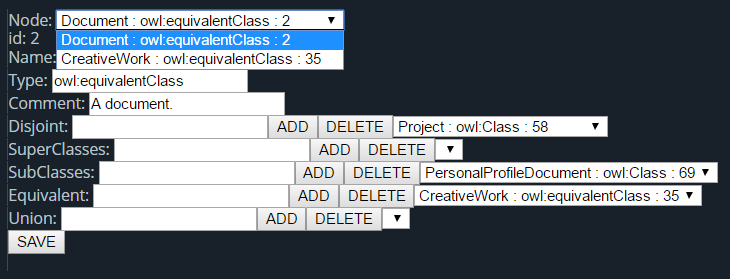


Figura - Selezione del nodo da eliminare e modificare

**3.3.1 RAPPRESENTAZIONE DEI DATI ALL’UTENTE**

Come è possibile vedere in figura 3 sì è scelto di rappresentare all’utente i nodi dell’ontologia nel formato *Name : Type : ID* dove:

* Name: è il nome nel linguaggio corrente per il nodo selezionato;
* Type: è il tipo owl del nodo selezionato;
* Id: numero identificativo univoco del nodo selezionato all’interno del JSON.

La scelta di questa rappresentazione è stata fatta per distinguere nodi con lo stesso nome ma con iri differenti (avranno id differenti, vedi nodo Person in foaf). Come ulteriore supporto all’utente nell’identificazione del nodo corretto si è riutilizzata la features di highlight node già presente in graph.

In questo modo, durante la selezione di un nodo esso verrà cerchiato in rosso nel grafo.

**4 MY PARSER**

E’ il modulo più importante fra quelli realizzati. Si occupa principalmente del reperimento dei dati dall’ontologia e della gestione della stessa (inserimento, modifica, cancellazione di nodi).

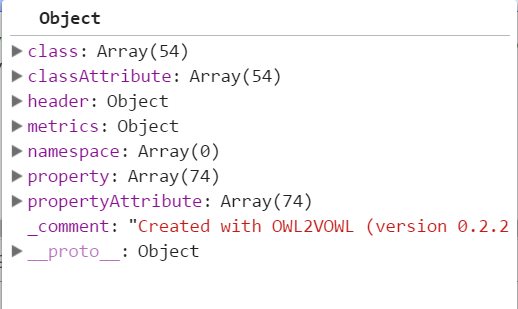
Poiché Webvowl ha un forte imprinting alla visualizzazione dei dati, le strutture dati già esistenti nel progetto, in particolare quella fornita da parser.js e gestita in app.js in seguito alla loadOntologyFromText, sono poco adatte per gestire la modifica dei nodi caricati dall’ontologia. Da ciò si è scelto di sviluppare questo modulo separato per la gestione dell’ontologia e di tutte le modifiche da apportarvici.

**4.1 STRUTTURE DATI**

Come detto nel paragrafo precedente sono state sviluppate delle strutture dati ad hoc per la gestione dell’ontologia. Innanzitutto è stato effettuato il parsing dell’ontologia (in formato .json) e tutte le modifiche sono state apportate alla struttura risultante. In questo modo, sfruttando i metodi nativi di javascript per la gestione dei file json, l’intero processo si sviluppa in tre fasi:

* Parsing dell’ontologia caricata;
* Modifica della stessa;
* Serializzazione della struttura dati modificata e reload.

La struttura dati parsata (contenuta nella variabile parsed) si compone di 7 campi:



* **class**
* **classAttribute**
* **header**
* **metrics**
* **property**
* **propertyAttribute**
* **namespace**

**Class:**  è un array di oggetti contenente tutti i nodi dell’ontologia. Ogni oggetto si compone dei campi id e type dove id è un identificativo progressivo univoco (stringa) e type identifica il tipo di nodo.

**ClassAttribute:** è un array di oggetti. Mantiene una corrispondenza uno ad uno col vettore class. Ogni oggetto si compone di diversi campi (potrebbero essere omessi):

* id: identificativo univoco del nodo;
* label: è un oggetto contenente il nome del nodo nelle lingue previste nell’ontologia;
* baseIri: è una stringa che contiene la parte iniziale dell’IRI (IRI= baseIRI + “#” + label);
* comment: è un oggetto che contiene il commento nelle lingue previste nell’ontologia;
* attributes: è un vettore di stringhe identificanti gli attributi del nodo (“equivalent”, “union”);
* equivalents: è un vettore contenente gli id dei nodi equivalenti al nodo in analisi
* union: stessa cosa del vettore equivalents in riferimento, però ai nodi che fanno parte dell’unione col nodo stesso;
* subClasses: vettore indicante tutti gli id che sono sottoclassi del nodo in esame;
* superClasses: vettore indicante tutti gli id che sono nodi padre del nodo in esame.

**Property:** è un array di oggetti contenente le proprietà di tutti i nodi. Ogni oggetto si compone dei campi id e type dove id è un identificativo progressivo univoco (stringa) e type identifica il tipo di proprietà;

**PropertyAttribute:** è un array di oggetti. Mantiene una corrispondenza uno ad uno col vettore property. Ogni oggetto si compone di diversi campi (potrebbero essere omessi):

* id: identificativo univoco della proprietà;
* range: id del nodo di arrivo della proprietà;
* domain: id del nodo di partenza della proprietà;
* attributes: è un vettore di stringhe identificanti gli attributi della proprietà.

**Header:** è un oggetto contenente tutti i baseIRIs usati nell’ontologia, l’IRI di riferimento e i linguaggi presenti nell’ontologia.

**Metrics:** è un oggetto contenente le metriche dell’ontologie (Classcount= numero di classi presenti, etc.).

Sono state definite, inoltre, altre strutture dati di lavoro (spiegate successivamente).

**4.2 FUNZIONI DI SUPPORTO**

Sono state sviluppate varie funzioni di supporto. Le più importanti sono:

* getClasses (id,insert): funzione che si occupa di popolare un vettore contenente tutte le classi presenti nell’ontologia. Questo vettore viene usato per il supporto all’utente attraverso l’autocompletamento. Viene passato l’id del nodo selezionato e un booleano che indica se si è in modalità inserimento o meno.
* existNode(name, iri): funzione che controlla se un nodo inserito/modificato dall’utente non sia omonimo di uno già esistente, basandosi sull’IRI.
* typeReparsing(): questa funzione garantisce la consistenza dell’ontologia, ovvero che ad ogni nodo sia assegnato il tipo di classe corretto.
* getDifference (editData,baseData, op): dati in input il nodo prima e dopo la modifica ed il tipo di operazione, la funzione ritorna due vettori, added e deleted.

Added contiene gli id dei nodi da aggiungere al nodo in esame ( in riferimento all’operazione indicata da op: disjoint, subclassOf, superclass, union, equivalent) che prima non erano presenti mentre deleted contiene gli oggetti che non sono più presenti rispetto all’ontologia precedente alla modifica.

* addProperty(id,add, type): in base al tipo di proprietà/relazione indicata da type, questa funzione opera in modo da aggiungere tale proprietà all’ontologia. Vengono sfruttate due strutture dati (property e propertyAttribute), strutturate come le rispettive strutture del file JSON, e popolate in modo tale da descrivere correttamente la proprietà che coinvolge il nodo selezionato (id) e tutti i nodi presenti in add. Ad esempio se venisse aggiunta una relazione di sottoclasse fra il nodo con id=1 e i nodi add= [2,3], verrebbero create due proprietà di subClassOf, aventi come domain=1 e range=2, per la prima proprietà, e range=3 per la seconda.
* removeProperty(id,del, type): opera in modo duale ad addProperty.

**4.3 READ**

Funzione che, passato l’id del nodo di cui reperire le informazioni, popola la struttura creata ad hoc per contenere queste informazioni e le ritorna al chiamante.

La struttura dati utilizzata è la seguente:

var data= {name:"", type:"", comment:"", disjointWith:[], subClassOf:[], equivalent:[], superClasses:[], union:[]};

dove:

* name: nome del nodo letto, per la lingua corrente, dal campo label di parsed;
* type: tipo del nodo;
* comment: commento per la lingua corrente;
* disjoinWith, subClassOf, equivalent, union, superClasses: vettori di oggetti contenenti i rispettivi nodi.

L’oggetto contenuto è così definito:

var element= {name:"", internalindex:"", id:"", type:""};

dove internalIndex è la posizione occupata dal nodo nel vettore classAttribute/class.

Questa funzione viene utilizzata durante la creazione della pagina popup, in particolare quando viene richiamata l’htmlCreator, nella fase di editing di un nodo.

**4.4 EDIT**

Funzione che si occupa di riportare, in maniera consistente, le modifiche effettuate dall’utente all’ontologia.

Come argomento vengono passate due strutture dati, baseData e data, contenenti, rispettivamente, tutte le informazioni del nodo da modificare prima e dopo la modifica.

Ove già presenti, i campi vengono solamente aggiornati col valore inserito dall’utente altrimenti vengono creati. Per quanto riguarda la proprietà di disjoint o le relazioni di equivalent, subclassOf, union, superClasses si è operati in questo modo.

* Disjoint: sono stati popolati i vettori added e delete. Per ogni elemento added viene creata una proprietà sfruttando la funzione addProperty.

Questa non fa altro che aggiungere nei vettori property e propertyAttribute di parsed due oggetti ben formattati secondo l’impostazione del json.

Per quanto riguarda la delete viene sfruttata la funzione removeProperty, duale dell’addProperty.

* SubClassOf: viene dapprima assegnato il vettore contenete gli id dei nodi figli attuali del nodo in esame alla struttura parsed. Successivamente, ad ogni nodo figlio viene aggiunto, come nodo padre, il nodo in esame. Infine vengono popolati i vettori added e deleted ed aggiunte/rimosse le proprietà di subClassOf sfruttando i metodi addProperty/removeProperty.
* SuperClasses: lavora in modo duale rispetto SubClassOf.
* Union: vengono assegnati al nodo in esame tutti gli id contenuti in data.union, quindi viene assegnato/rimosso l’attributo “union” al nodo corrente.
* Equivalent: viene dapprima assegnato al nodo il contenuto di data.equivalent; per ogni nodo contenuto in equivalent viene assegnato, come nodo equivalente, il nodo in esame ed infine assegnato l’attributo di “equivalent” al nodo in esame e a tutti i nodi trattati.

Dualmente vengono rimossi i nodi di equivalent che non sono più presenti dopo la modifica dell’utente.

Per quanto concerne l’union e l’equivalent viene anche settato, in maniera consistente, il tipo della classe:

se il nodo avrà dei nodi equivalenti il tipo sarà settato ad “owl:equivalentClass” e, altresì, settato ad “owl:Class” se non ve ne saranno più presenti. Stesso discorso viene applicato parallelamente per l’unione di nodi.

**4.5 DELETE**

Questa funzione si occupa dell’eliminazione di un nodo, di cui è fornito l’id come parametro, e di tutte le proprietà ad esso correlato. Inoltre elimina ogni referenza presente nell’ontologia riferita al nodo. Ad esempio, se il nodo da eliminare avesse una relazione di sottoclasse, si individuano i nodi sottoclasse e su ciascuno di essi viene rimosso l’id del nodo da cancellare nell’attributo superClass. Infine viene controllata la consistenza dell’ontologia.

**4.6 INSERT**

Questa funzione riceve come parametro la struttura dati contenente le informazioni del nodo da inserire nell’ontologia e, processandolo in maniera analoga all’edit, procede ad apportare le modifiche all’ontologia.

Vengono sfruttate due strutture dati impostate analogamente alla struttura del JSON (classe e classAttribute). Una volta popolate queste strutture con i dati forniti dall’utente, vengono aggiunte con delle push all’oggetto parsed (struttura dati dell’ontologia parsata). Infine vengono generate le proprietà in base alle proprietà inserite dall’utente (subClassOf, disjoint, etc.).

N.B. prima che vengano elaborati i dati, sia per la insert che per l’edit, viene fatto un check sull’esistenza del nodo (funzione existNode()) per evitare omonimie nell’ontologia.

**5. DATI ELABORATI**

Ai fini del tema d’anno è stata elaborata una restrizione dei dati gestibili da Webvowl. In particolare sono state gestite le relazioni di:

* Equivalenza (equivalent);
* Disgiunzione (disjoint);
* Unione (union);
* Sottoclasse (subclasses);
* Superclasse (superclasses);
* Nome (label);
* IRI;
* Tipo di nodo (type);
* Commento (comment).

**6. POPUP WINDOW**

Come detto precedentemente quando si clicca col tasto destro su un nodo, si ha la possibilità di effettuare 3 operazioni. Ciascuna di esse richiama la pagina pop.html che viene formattata in modo consono all’operazione scelta.

**6.1 ADDSUBCLASS**

Selezionando addSubClass si ha la finestra in figura:

Come si può notare viene già indicato l’id del nodo all’interno dell’ontologia, mentre tutti i restanti campi sono vuoti ad eccezione di base IRI e superClass. Infatti è stato previsto che l’aggiunta di un nodo sia effettuata in modo simile a Protegé. Di fatto se si volesse creare un nodo isolato basta semplicemente rimuovere, cliccando sul bottone delete, il nodo contenuto nella selectbox della voce SUPERCLASS.

In ogni inputBox è stata prevista una lista di autocompletamento che suggerisca tra le classi presenti nell’ontologia.

Cliccando sul tasto ADD viene aggiunta nella selectbox relativa il nodo contenuto nella inputbox. Sono stati gestiti alcuni errori semantici quali:

* Aggiunta di un nodo non esistente come range della relazione;
* Aggiunta di un nodo padre come nodo figlio del nodo selezionato;
* Aggiunta di un nodo figlio come nodo padre del nodo selezionato;
* Aggiunta di sé stesso come range di una relazione.

Infine per rendere effettive le modifiche all’ontologia basterà cliccare sul tasto SAVE.

In fase di salvataggio viene dapprima controllato che non si stia inserendo un nodo già esistente (omonimia), restituendo eventualmente un errore all’utente, e successivamente viene processata la modifica dell’ontologia. Fatto ciò, il popup viene automaticamente chiuso ed il grafo, rappresentante l’ontologia, ricaricato con le modifiche apportate.

**6.2 EDIT**

Selezionando edit si ha la finestra in figura:

Tutti i campi presenti vengono popolati con i dati relativi al nodo cliccato.

Sarà possibile, a questo punto, interagire con i campi previsti e modificare tutte le relazioni del nodo.

Data la natura grafica di Webvowl che raggruppa i nodi equivalent è stata prevista, in questa modalità, una selectbox aggiuntiva che permetta di selezionare quale fra i nodi equivanti si voglia editare.

Anche in questa modalità vengono effettuati i check semantici descritti per la funzione addSubClass.

**6.3 DELETE**

Selezionando delete si ha la finestra in figura:

In questo caso è prevista solo una selectbox contenete il/i nodo/i cliccati ed un tasto di conferma.

**7 SVILUPPI FUTURI**

Il codice è stato pensato per essere facilmente espandibile ed adattabile ad eventuali aggiornamenti del sistema Webvowl. Eventuali sviluppi futuri consisterebbero nel trattare ulteriori relazioni/proprietà da noi non trattate.

**8 CONCLUSIONI**

Superate le difficoltà iniziali comportate dallo studio del funzionamento interno del sistema, delle loro strutture dati, del formato utilizzato per l’ontologia (json) e paragoni fatti con Protegè per la modellazione delle operazioni di inserimento, modifica ed eliminazione su una ontologia il restante lavoro è stato abbastanza agevole. Il tutto si è tradotto nel creare dei moduli per facilitare le operazioni summenzionate e sfruttare i metodi già esistenti per la visualizzazione dell’ontologia e relativi aggiornamenti.