Ontologie OWL: Teoria e Pratica

Terza puntata

Questo terzo articolo conclude la breve trattazione sulle ontologie e sulle loro applicazioni informatiche.
Utilizzando un taglio più pragmatico rispetto ai precedenti, si vedrà come realizzare applicazioni ontology-based utilizzando il framework open source Jena di HP Labs.

di Nicola Capuano

Nelle prime due puntate di questo mini-corso abbiamo imparato a creare una semplice ontologia con l'editor open source *Protégé* [1] introducendo, nel contempo, i principali elementi del linguaggio OWL [2]: le classi, le proprietà e gli individui. Utilizzando Protégé abbiamo imparato a definite tassonomie, a dichiarare disgiunzioni tra classi, a definire gerarchie di proprietà, a creare restrizioni sulle classi, ad aggiungere classi definite ed utilizzare un reasoner sia per il controllo di consistenza che per la classificazione dell'ontologia.

In questa terza ed ultima puntata impareremo ad utilizzare *Jena* per realizzare applicazioni Java ontology-based. Jena è un framework open source sviluppato dal Semantic Web Research Group di HP Labs [3]. Nato inizialmente per realizzare applicazioni che supportassero RDF e RDFS, Jena è stato successivamente esteso ai linguaggi per le ontologie, tra cui OWL e DAML+OIL. Jena è reperibile su Sourceforge [4]. Per utilizzarlo è necessario scaricare l'ultima versione, decomprimerla (avendo cura di mantenere intatta la struttura delle directory) e posizionarla in un punto del file system raggiungibile dal classpath del nostro compilatore Java preferito (in particolare va aggiunto al classpath la sottodirectory *lib* della directory di Jena).

Descriveremo nel seguito come creare con Jena, analogamente a quanto abbiamo fatto nelle scorse puntate con l'editor grafico Protégé, una versione semplificata dell'ontologia della birra. Passeremo poi alla manipolazione dell'ontologia creata attraverso l'utilizzo di reasoner esterni per introdurre, infine, alcuni aspetti relativi alla gestione di ontologie multiple e persistenti.

Nicola Capuano ncapuano@infomedia.it

Si occupa di gestione di progetti di Ricerca e Sviluppo in diversi settori delle ICT e, in particolar modo, su tematiche relative alle Tecnologie della Conoscenza ed all'e-Learning. Collabora stabilmente con l'Università di Salerno su tematiche di Intelligenza Artificiale e con il Centro di Ricerca in Matematica Pura ed Applicata nel coordinamento di progetti co-finanziati a livello nazionale ed europeo. È autore di circa 30 lavori scientifici. La sua home page è: www.capuano.biz.

Creiamo un'ontologia con Jena

Jena, pur supportando diversi linguaggi per l'espressione di ontologie, fornisce un'unica API neutrale rispetto al linguaggio. Ciò consente di operare sulle ontologie senza preoccuparsi del linguaggio attraverso il quale l'ontologia stessa verrà serializzata. Tutte le ontologie vengono gestite in Jena attraverso l'interfaccia *OntModel* (implementata dalla classe *OntModelImpl*). Per creare una nuova ontologia è necessario invocare il metodo *createOntologyModel* della factory *ModelFactory*. Per specificare il linguaggio ontologico da utilizzare è possibile passare al metodo l'URI che definisce il linguaggio stesso o, in alternativa, è possibile utilizzare delle costanti all'uopo definite. Alcuni esempi di URI e relative costanti sono riportati in **Tabella 1**. Per creare un'ontologia OWL-DL, ad esempio, è necessario utilizzare il seguente codice:

```
OntModel m = ModelFactory.createOntologyModel
                        (ProfileRegistry.OWL_DL_LANG);
```

TABELLA 1 I linguaggi supportati da Jena e le relative URI e Costanti

Linguaggio	URI	Costante
RDFS	http://www.w3.org/2000/01/rdf-schema#	ProfileRegistry.RDFS_LANG
DAML+OIL	http://www.daml.org/2001/03/daml+oil#	ProfileRegistry.DAML_LANG
OWL Full	http://www.w3.org/2002/07/owl#	ProfileRegistry.OWL_LANG
OWL DL	http://www.w3.org/TR/owl-features/#term_OWLDL	ProfileRegistry.OWL_DL_LANG
OWL Lite	http://www.w3.org/TR/owl-features/#term_OWLLite	ProfileRegistry.OWL_LITE_LANG

Dopo aver creato l'ontologia, è necessario definirne il *default namespace*: una stringa di caratteri che verrà utilizzata come prefisso standard per tutte le classi, le proprietà e gli individui definiti all'interno di un'ontologia. L'utilizzo dei namespace è fondamentale quando si utilizzano contemporaneamente diverse ontologie: mantenere namespace diversi per ontologie diverse consente di riferirsi a classi, proprietà ed individui appartenenti ad un'altra ontologia in maniera non ambigua. I namespace prendono la forma di URI che terminano con il carattere #. Quando, nel corso delle puntate precedenti, abbiamo definito l'ontologia *birra.owl*, non è stato necessario definire esplicitamente il default namespace. Ciò non significa che il namespace non sia stato definito affatto. Protégé stabilisce infatti che, se non altrimenti specificato, il default namespace è *http://www.owl-ontologies.com/unnamed.owl#*. Ciò implica, ad esempio, che il nome completo della classe *Birra* definita nella prima puntata è, in realtà, *http://www.owl-ontologies.com/unnamed.owl#Birra*. La stringa di default può essere facilmente modificata accedendo al box *Default Namespace* del pannello *Metadata* di Protégé. Purtroppo Jena non offre lo stesso automatismo di Protégé e ci costringe a definire esplicitamente il default namespace della nostra ontologia. Per fare ciò è necessario utilizzare il metodo *setNsPrefix* di *OntModel*. Il primo parametro del metodo è, nel nostro caso, una stringa vuota e indica che quello che stiamo definendo è il namespace di default e non un qualsiasi altro prefisso dell'ontologia. Il secondo parametro è l'URI. Il frammento di codice seguente, ad esempio, definisce il default namespace dell'ontologia *m* come *http://www.owl-ontologies.com/birra.owl#*.

```
String ns = "http://www.owl-ontologies.com/birra.owl#";
m.setNsPrefix("", ns);
```

Creata l'ontologia e stabilitone il default namespace, è possibile cominciarne a definirne gli elementi componenti: classi, proprietà ed individui.

Popoliamo l'ontologia con classi, proprietà ed individui

Il **Listato 1** è un semplice esempio che mostra i passi principali necessari alla definizione di un'ontologia direttamente da codice Java sfruttando le potenzialità di Jena. Come mostrano le prime righe del listato, occorre innanzitutto importare i package *com.hp.hpl.jena.rdf.model* e *com.hp.hpl.jena.ontology* di Jena. I passi successivi (passi 1 e 2) sono relativi alla creazione dell'ontologia ed alla definizione del default namespace che abbiamo già chiarito nella sezione precedente. Vanno poi definite (passo 3) alcune classi primitive (ci limitiamo in questa sede alle classi *Birra*, *Pilsner*, *Lievito* e *LievitoBassaFermentazione*) attraverso l'invocazione del metodo *createClass* di *OntModel*. È importante notare che nella definizione di nuove classi, così come di ogni altro elementi dell'ontologia, il nome fornito deve essere completo di namespace.

Il passo seguente (passo 4) consiste nell'aggiungere alcune relazioni tassonomiche tra le classi (specificando superclassi e sottoclassi) con il metodo *addSubClass* di *OntClass*. In particolare dichiariamo che *Pilsner* è sottoclasse di *Birra* mentre *LievitoBassaFermentazione* è sottoclasse di *Lievito*. Poi (passo 5) si procede alla definizione delle disgiunzioni tra classi invocando il metodo *addDisjointWith* di *OntClass*. A differenza di Protégé, per dichiarare che le classi *Birra* e *Lievito* sono disgiunte, è necessario dichiarare sia che *Birra* è disgiunta da *Lievito* sia che *Lievito* è disgiunta da *Birra*.

Per definire le proprietà (passo 6), il metodo da invocare varia a seconda della tipologia scelta. La **Tabella 2** mostra quali metodi invocare in corrispondenza di specifiche tipologie. Ciascun metodo accetta due parametri: il primo è il nome della proprietà, il secondo (opzionale) è il flag di funzionalità: se è vero, la proprietà è anche funzionale. Nel nostro caso, per definire le proprietà transitive *haIngrediente* e *haLievito* useremo il metodo *craeteTransitiveProperty* di *OntModel*. La gerarchia delle proprietà (passo 7) andrà inoltre definita con il metodo *addSubProperty* di *OntProperty*. Infine, per definire dominio e range (passo 8), si utilizzeranno i metodi *addDomain* e *addRange* di *OntProperty*.

Vogliamo ora dichiarare che le birre pilsner sono fatte solo con lieviti a bassa fermentazione aggiungendo la coppia di restrizioni $\exists$ *haLievito LievitoBassaFermentazione* e $\forall$ *haLievito LievitoBassaFermentazione* (vedi prima puntata) alla classe *Pilsner*. Per far ciò (passo 9) occorre in primo luogo definire due restrizioni anonime utilizzando il metodo *createSomeValuesFromRestriction* per la restri-

LISTATO 1 Realizzare una semplice ontologia con Jena

```
package birra;

import com.hp.hpl.jena.rdf.model.*;
import com.hp.hpl.jena.ontology.*;
import java.io.*;
import java.util.*;

public class birra {
  public static void main(String[] args) {

    // passo 1: creaiamo l'ontologia
    OntModel m = ModelFactory.createOntologyModel (ProfileRegistry.OWL_DL_LANG);

    // passo 2: definiamo il default namespace
    String ns = "http://www.owl-ontologies.com/birra.owl#";
    m.setNsPrefix("", ns);

    // passo 3: definiamo alcune classi primitive
    OntClass cBirra = m.createClass(ns + "Birra");
    OntClass cPilsner = m.createClass(ns + "Pilsner");
    OntClass cLievito = m.createClass(ns + "Lievito");
    OntClass cLievitoBassaFermentazione = m.createClass(ns + "LievitoBassaFermentazione");

    // passo 4: creiamo la tassonomia
    cBirra.addSubClass(cPilsner);
    cLievito.addSubClass(cLievitoBassaFermentazione);

    // passo 5: definiamo le disgiunzioni
    cBirra.addDisjointWith(cLievito);
    cLievito.addDisjointWith(cBirra);

    // passo 6: definiamo le proprietà
    OntProperty pHaIngrediente = m.createTransitiveProperty(ns + "haIngrediente");
    OntProperty pHaLievito = m.createTransitiveProperty(ns + "haLievito");

    // passo 7: definiamo la gerarchia delle proprietà
    pHaIngrediente.addSubProperty(pHaLievito);

    // passo 8: definiamo dominio e range delle proprietà
    pHaIngrediente.addDomain(cBirra);
    pHaLievito.addRange(cLievito);

    // passo 9: definiamo alcune restrizioni sulla classe Pilsner
    Restriction cSomeBassaFermentazione = m.createSomeValuesFromRestriction(null, pHaLievito,
      cLievitoBassaFermentazione);
    Restriction cAllBassaFermentazione = m.createAllValuesFromRestriction(null, pHaLievito,
      cLievitoBassaFermentazione);
    cPilsner.addSuperClass(cSomeBassaFermentazione);
    cPilsner.addSuperClass(cAllBassaFermentazione);

    // passo 10: aggiungiamo la classe definita Lager
    OntClass cLager = m.createIntersectionClass(ns + "Lager", m.createList(new OntClass[]
      {cBirra, cSomeBassaFermentazione, cAllBassaFermentazione}));

    // passo 11: definiamo alcuni individui
    Individual iPilsnerUrquell = m.createIndividual(ns + "pilsnerUrquell", cPilsner);
    Individual iLager = m.createIndividual(ns + "lager", cLievitoBassaFermentazione);

    // passo 12: applichiamo le proprietà agli individui
    iPilsnerUrquell.addProperty(pHaLievito, iLager);

    // passo 13: salviamo l'ontologia
    try {
      FileOutputStream fout = new FileOutputStream("jBirra.owl");
      m.write(fout);
    }
    catch (Exception ex) {
      System.out.println(ex);
    }
  }
}
```

zione esistenziale e il metodo *createAllValuesFromRestriction* per la restrizione universale (entrambi sono metodi di *OntModel*). Tali metodi accettano tre parametri: il primo è l'eventuale nome della classe definita dalla restrizione (nel nostro caso posto a *null* trattandosi di restrizioni anonime), il secondo è la proprietà coinvolta nella restrizione mentre il terzo è la classe argomento (filler).

Dopo aver definito le restrizioni occorre applicarla alla classe *Pilsner* attraverso il metodo *addSuperClass* di *OntClass*. In pratica, applicare una restrizione ad una classe si traduce nel dichiararla sotto-classe della restrizione, ovvero della classe che include tutti gli individui che rispettano la restrizione. Ciò è reso possibile dal fatto che OWL supporta l'ereditarietà multipla e quindi una stessa classe può essere derivata da più di una super-classe. Vediamo ora come è possibile aggiungere la classe "definita" *Lager* (vedi seconda puntata) composta da tutte le birre che hanno tra gli ingredienti solo lieviti a bassa fermentazione. Per far ciò (passo 10) è necessario definire la classe *Lager* come intersezione tra la classe *Birra* e le restrizioni anonime definite in precedenza utilizzando il metodo *createIntersectionClass* di *OntClass* che accetta come argomento la lista delle classi di cui calcolare l'intersezione. Come mostra la **Figura 1** la classe definita *Lager* è diversa dalla classe primitiva *Pilsner* dichiarata come sotto-classe comune alla classe *Birra* ed alle succitate restrizioni. Non ci resta che popolare l'ontologia con individui (passo 11) attraverso il metodo *createIndividual* della classe *OntModel*. Per esercitarci ci limitiamo a creare l'individuo *pilsnerUrquell* della classe *Pilsner* e l'individuo *lager* (da non confondere con la classe *Lager*) della classe *LievitoBassaFermentazione*. Per definire infine che l'individuo *pilsnerUrquell* ha tra gli ingredienti il lievito *lager* (passo 12) occorre utilizzare il metodo *addProperty* di *Individual*. Per serializzare (salvare) l'ontologia realizzata (passo 13) è sufficiente utilizzare il metodo *write* di *OntModel* che accetta come argomento un output stream. Scegliamo di salvarla nel file "jBirra.owl". L'ontologia serializzata può essere agevolmente importata in Protégé avendo cura di selezionare l'opzione *Build New Project* dal menu *File*. Di contro è possibile importare in un *OntModel* di Jena un'ontologia esistente (eventualmente creata con Protégé) semplicemente invocando il metodo *read* su un qualsiasi input stream. Ad esempio per leggere l'ontologia della birra creata nelle puntate precedenti è sufficiente utilizzare il seguente frammento di codice

TABELLA 2 I metodi corrispondenti alle diverse tipologie di proprietà

Metodo di OntModel	Tipologia di Proprietà
createOntProperty	Proprietà generica
createDatatypeProperty	Proprietà di tipo datatype
createObjectProperty	Proprietà di tipo object
createSymmetricProperty	Proprietà object simmetrica
createTransitiveProperty	Proprietà object transitiva
createInverseFunctionalProperty	Proprietà object funzionale inversa

```
try {
  FileInputStream fin = new FileInputStream("birra.owl");
  m.read(fin, "http://www.owl-ontologies.com/
                                    unnamed.owl#");
}
catch (Exception ex) {
  System.out.println(ex);
}
```

A differenza del metodo *write*, al metodo *read* di *OntModel* va passato anche il namespace (eventualmente nullo) da utilizzare per convertire i nomi completi dell'ontologia letta in nomi abbreviati locali. Come abbiamo visto, il default namespace dell'ontologia della birra era *http://www.owl-ontologies.com/unnamed.owl#*: nel nostro esempio utilizziamo dunque questo URI. Una variante interessante del metodo *read* (che non approfondiamo) è quella che accetta come parametro, piuttosto che un input stream, un URL che punta ad un'ontologia da scaricare dalla rete.

> Il default namespace è il prefisso standard per tutti gli elementi definiti all'interno di un'ontologia

Colleghiamo ed utilizziamo un reasoner esterno

È interessante notare che Jena non trasforma la struttura dell'ontologia in una struttura di classi. Tutte le informazioni sono mantenute da *OntModel* come statement RDF. Ogni volta che un'informazione viene ricercata nell'ontologia, si cerca di desumerla analizzando gli statement presenti. Di contro, ogni volta che nuove informazioni vengono inserite, si aggiungono statement RDF. Se all'ontologia è agganciato un *reasoner*, nuovi statement vengono automaticamente creati ed aggiunti al modello a partire da deduzioni effettuate sugli statement dichiarati. Gli statement dedotti vengono poi utilizzati nella fase di ricerca delle informazioni alla stregua di statement dichiarati. Jena dispone di tre reasoner interni di complessità crescente. Purtroppo il più complesso dei tre riesce a malapena a trattare un sottoinsieme di OWL-DL molto vicino ad OWL-Lite. Per ovviare a

questa mancanza Jena rende possibile la connessione con reasoner esterni conformi allo standard DIG tra cui *Racer* [5] (vedi seconda puntata).

Il **Listato 2** mostra come effettuare questa connessione e sfruttare il reasoner per classificare l'ontologia realizzata.

I reasoner interni forniti da Jena non sono in grado di trattare ontologie OWL-DL

Il passo 14 del **Listato 2** introduce un frammento di codice che, per ogni classe dell'ontologia *m*, elenca tutte le sotto-classi.

Il frammento sfrutta il metodo *listClasses* di *OntModel* che restituisce un iteratore sulle classi dell'ontologia; il metodo *listSubClasses* di *OntClass* che restituisce un iteratore sulle sotto-classi di una classe data ed il metodo *getLocalName* di *OntClass* che restituisce il nome di una classe data. Aggiungendo il solo codice del passo 14 al **Listato 1** ed eseguendolo, si ottiene un risultato simile al seguente (la similitudine sta nel fatto che le righe potrebbero essere visualizzate in un ordine diverso).

```
La Classe Birra ha le sotto-classi: Pilsner
La Classe Pilsner ha le sotto-classi:
La Classe Lievito ha le sotto-classi: LievitoBassaFer-
                                                mentazione
La Classe LievitoBassaFermentazione ha le sotto-classi:
La Classe Lager ha le sotto-classi:
```

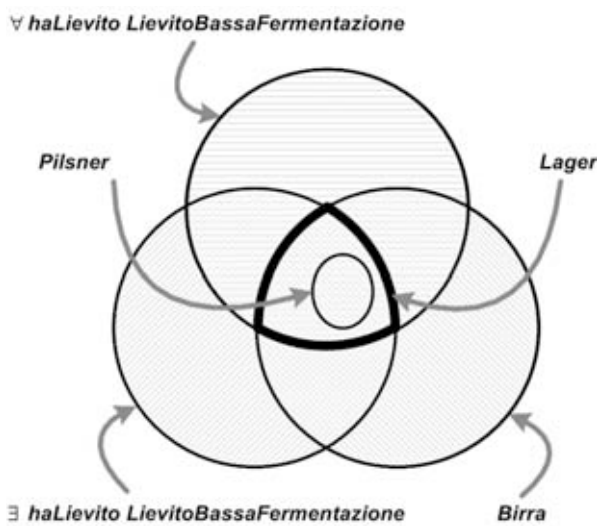
Non essendo in funzione alcun reasoner, Jena non è in grado di dedurre, ad esempio, che *Pilsner* è sottoclasse di *Lager* né, tanto meno, che *Lager* è sottoclasse di *Birra*. Tali informazioni non sono state infatti esplicitamente dichiarate.

Il passo 1 (scomposto in quattro sotto-punti) del **Listato 2** mostra come creare un'ontologia *m* alla quale è agganciato un reasoner DIG accessibile via HTTP alla porta 8080 (la porta di default utilizzata da Racer). Per motivi di spazio non ci addentreremo nella spiegazione del passo 1 che richiederebbe l'introduzione di diversi concetti che vanno al di là degli obiettivi di questo articolo (si rimandano gli interessati alla lettura di [6]). Ci basti sapere che, sostituendo il passo 1 del **Listato 2** al passo 1 del **Listato 1** e aggiungendo i tre import relativi ai package di gestione dei reasoner, all'ontologia *m* viene data facoltà di accedere ai servizi offerti da un reasoner DIG esterno. Proviamo ora ad eseguire il **Listato 1** esteso con i passi 1 e 14 del **Listato 2**. Perché esso funzioni è ovviamente necessario che sulla porta 8080 della stessa macchina su cui si esegue il codice sia in ascolto un reasoner DIG come il già citato Racer (si rimanda alla seconda puntata per maggiori informazioni sull'installazione e la configurazione di Racer). Il risultato dell'elaborazione è il seguente.

```
La Classe Birra ha le sotto-classi: Pilsner Lager Nothing
La Classe Pilsner ha le sotto-classi: Nothing
La Classe Lievito ha le sotto-classi: LievitoBassaFer-
                                                mentazione Nothing
La Classe LievitoBassaFermentazione ha le sotto-
                                                classi: Nothing
La Classe Lager ha le sotto-classi: Pilsner Nothing
```

Come possiamo notare, sfruttando le capacità di inferenza offerte dal reasoner, a partire dalla stessa ontologia è possibile estrarre una quantità maggiore di informazioni. In particolare la classe definita *Lager* è diventata sotto-classe di *Birra* e super-classe di *Pilsner*. Inoltre, ogni classe ha per sotto-classe anche la classe *owl:Nothing*. Tale classe, definita da OWL, è la sotto-classe universale: così come *owl:Thing*, per definizione, è super-classe di ogni classe OWL, così *owl:Nothing*, per definizione, è sotto-classe di ogni classe OWL. Agganciando un reasoner ad un'ontologia è possibile accedere anche ad ulteriori metodi offerti dalla super-interfaccia *InfModel* di *OntModel* tra cui, ad esempio, il metodo *validate* per effettuare il controllo di consistenza dell'ontologia. Maggiori dettagli su queste funzionalità sono forniti in [7].

FIGURA 1 Differenza tra la classe primitiva Pilsner e la classe definita Lager.



Cenni su aspetti avanzati

Dopo aver descritto alcune funzionalità di base per la gestione delle ontologie offerte da

LISTATO 2 Come agganciare un reasoner DIG ad un'ontologia con Jena

```
import com.hp.hpl.jena.reasoner.*;
import com.hp.hpl.jena.reasoner.dig.*;
import com.hp.hpl.jena.vocabulary.*;
…

    // Passo 1.1: Genera una configirazione per il reasoner
    Model cm = ModelFactory.createDefaultModel();
    Resource conf = cm.createResource();
    conf.addProperty(ReasonerVocabulary.EXT_REASONER_URL,
      cm.createResource("http://localhost:8080"));

    // Passo 1.2: Crea il reasoner
    DIGReasonerFactory drf = (DIGReasonerFactory)
      ReasonerRegistry.theRegistry().getFactory(DIGReasonerFactory.U
RI);
    DIGReasoner r = (DIGReasoner) drf.create(conf);

    // Passo 1.3: Crea la specifica per la creazione dell'ontologia
    OntModelSpec spec = new OntModelSpec(OntModelSpec.OWL_DL_MEM);
    spec.setReasoner(r);

    // Passo 1.4: Crea l'ontologia
    OntModel m = ModelFactory.createOntologyModel(spec, null);
…

    // Passo 14: Mostra le sotto-classi di ogni classe non anonima
    for (Iterator ic = m.listClasses(); ic.hasNext(); ) {
       OntClass cl = (OntClass) ic.next();
       if (cl.getLocalName() != null) {
         System.out.print("La Classe " + cl.getLocalName() + " ha le
sotto-classi: ");
         for (Iterator is = cl.listSubClasses(); is.hasNext(); ) {
           OntClass cs = (OntClass) is.next();
           if (cs.getLocalName() != null)
             System.out.print(cs.getLocalName() + " ");
         }
         System.out.println();
       }
    }
…
```

Jena, passiamo a introdurre brevemente alcuni aspetti avanzati. Un primo aspetto di interesse consiste nella gestione di *ontologie multiple*. Spesso, nella definizione di ontologie OWL è necessario riferirsi a classi, proprietà o istanze definite in altre ontologie. In questo caso risulta spesso necessario importare ontologie esterne o parti di esse. Jena consente di gestire questa necessità attraverso la classe *OntDocumentManager*, una classe che si occupa, appunto, della gestione dei collegamenti tra ontologie.

Modificando opportunamente le proprietà di *OntDocumentManager* è possibile definire la politica di import delle ontologie seguita da Jena. È possibile, ad esempio, attivare l'import automatico di ontologie esterne (ogni qual volta un'ontologia esterna viene richiamata, questa viene reperita in base all'URL ed unita al modello selezionato), attivare il caching delle ontologie esterne (piuttosto che reperire le ontologie sulla rete, ci si riferisce a delle copie locali), attivare l'importazione ricorsiva (vengono importate non solo le ontologie richiamate dall'ontologia selezionata ma anche, ricorsivamente, le ontologie richiamate dalle ontologie importate), e così via. In alternativa è possibile procedere alla fusione esplicita di due ontologie invocando il metodo *addSubModel* di *OntModel*. Un ulteriore strumento molto potente offerto da Jena sono le *ontologie persistenti*. Tali ontologie, pur mantenendo le stesse funzionalità delle ontologie finora descritte, non hanno bisogno di essere serializzate e deserializzate in file XML, essendo esse legate ad un database esterno che mantiene tutti gli statement RDF di cui l'ontologia è composta. Ciò consente di lavorare, mantenendo performance elevate, anche con ontologie di grosse dimensioni. Quando poi si rende necessaria la distribuzione di un'ontologia persistente all'esterno, è sempre possibile esportarne una copia in formato XML. Maggiori dettagli sulle ontologie persistenti in Jena sono forniti in [8] attraverso un esempio di utilizzo.

Conclusioni

Abbiamo completato con questo articolo una trattazione di alto livello sulle ontologie OWL viste sia da un punto di vista teorico (cosa sono, a cosa servono e come sono fatte) che da un punto di vista ingegneristico (come si progettano) che da un punto di vista informatico (come si utilizzano nell'ambito di applicazioni Java). Per chi volesse approfondire la trattazione di Jena consigliamo di consultare il sito ufficiale ospitato da Sourceforge [4]. Un'ulteriore fonte di preziose informazioni per gli sviluppatori Jena è la mailing list Jena-dev [9].

Riferimenti

[1] http://protege.stanford.edu – *il sito di Protégé ospitato dalla Stanford University*

[2] http://www.w3c.org/2004/OWL – *la sezione del sito del World Wide Web Consortium dedicata a OWL*

[3] http://www.hpl.hp.com/semweb/ – *la home page del Semantic Web Research Group di HP Labs*

[4] http://jena.sourceforge.net/ – *il sito di riferimento di Jena ospitato da SourceForge*

[5] HTUhttp://www.sts.tu-harburg.de/~r.f.moeller/racerUTH – *il sito di Ralf Möller dedicato a Racer*

[6] http://jena.sourceforge.net/how-to/dig-reasoner.html – *HOWTO use Jena with an external DIG reasoner*

[7] http://jena.sourceforge.net/ontology/ – *Jena 2 Ontology API*

[8] http://jena.sourceforge.net/ontology/examples/persistent-ont-model/ – *Persistent Ontologies Example*

[9] jena-dev@groups.yahoo.com – *La mailing list Jena-dev*