POLITECNICO DI MILANO
Facoltà di Ingegneria
Corso di Laurea Specialistica in Ingegneria Informatica

PROGETTO E IMPLEMENTAZIONE DI UN'APPLICAZIONE BASATA SUL WEB PER LA COSTRUZIONE E LA MODIFICA DI ONTOLOGIE

Relatore: Prof. Marco COLOMBETTI
Correlatori: Ing. David LANIADO
Ing. Mario ARRIGONI NERI

Tesi di Laurea di:
Fabio GRASSANI
Matr. n. 682565

Anno Accademico 2009–2010

Prefazione

Questa tesi si prefigge lo scopo di studiare il software esistente nel campo degli editor di ontologie in OWL, con particolare attenzione alle applicazioni basate sul web, alla ricerca di un software che implementi un sistema collaborativo e si avvalga di funzionalità grafiche in ausilio all'utente.

Una volta appurato che allo stato odierno non è ancora stato rilasciato un software adatto, che risponda in modo adeguato alle esigenze sollevate, si passa alla fase di progetto e alla successiva fase di implementazione di un'applicazione basata sul web che faccia uso di interfacce grafiche user-friendly per produrre e manipolare ontologie.

Indice

Elenco delle figure

Elenco delle tabelle

Elenco dei frammenti di codice

Sigle

Segue una lista delle sigle (formati, protocolli, standard ecc...) citati nel testo.

AJAX ..... *Asynchronous JavaScript And XML*
API ..... *Application Programming Interface*
ChAO ..... *Changes and Annotation Ontology*
CPL ..... *Common Public License*
GUI ..... *Graphic User Interface*
GWT ..... *Google Web Toolkit*
HTML ..... *Hyper-Text Markup Language*
HTTP ..... *Hyper-Text Transfer Protocol*
IDE ..... *Integrated Development Environment*
IRI ..... *Internationalized Resource Identifier*
JDK ..... *Java Development Kit*
JRE ..... *Java Resource Environment*
LGPL ..... *Lesser General Public License*
MPL ..... *Mozilla Public License*
OKBC ..... *Open Knowledge-Base Connectivity (protocol)*
OWL ..... *Ontology Web Language*
OWLIE ..... *Ontology Web-based Lightweight Interactive Editor*
PNG ..... *Portable Network Graphics*
POJO ..... *Plain Old Java Object*
RDF ..... *Resource Description Framework*
RDFS ..... *RDF-Schema*
RIA ..... *Rich Internet Application*
RMI ..... *Remote Method Invocation*
SPARQL ..... *SPARQL Protocol And Query Language*
TCP ..... *Transmission Control Protocol*
UML ..... *Unified Modeling Language*

URI ..... *Uniform Resource Identifier*
URL ..... *Uniform Resource Locator*
W3C ..... *World Wide Web Consortium*
XML ..... *eXtensible Markup Language*
XSD ..... *XML Schema Definition*
XUL ..... *XML User-interface Language*
ZUML ..... *ZK User-interface Markup Language*

Capitolo 1

Introduzione

> *Ho fatto un sogno riguardante il Web...ed è un sogno diviso in due parti. Nella prima parte, il Web diventa un mezzo di gran lunga più potente per favorire la collaborazione tra i popoli. Ho sempre immaginato lo spazio dell'informazione come una cosa a cui tutti abbiano accesso immediato e intuitivo, non solo per navigare ma anche per creare. [...] Inoltre, il sogno della comunicazione diretta attraverso il sapere condiviso dev'essere possibile per gruppi di qualsiasi dimensione, gruppi che potranno interagire elettronicamente con la medesima facilità che facendolo di persona. Nella seconda parte del sogno, la collaborazione si allarga ai computer. Le macchine diventano capaci di analizzare tutti i dati sul Web, il contenuto, i link e le transazioni tra persone e computer. La "Rete Semantica" che dovrebbe renderlo possibile deve ancora nascere, ma quando l'avremo i meccanismi quotidiani di commercio, burocrazia e vita saranno gestiti da macchine che parleranno a macchine, lasciando che gli uomini pensino soltanto a fornire l'ispirazione e l'intuito. Finalmente, si materializzeranno quegli "agenti" intelligenti sognati per decenni. Questo Web comprensibile alle macchine si concretizzerà introducendo una serie di progressi tecnici e di adeguamenti sociali attualmente in fase di sviluppo*
>
> Sir Tim Berners-Lee[1]

Così Tim Berners-Lee, uno tra i primi ad avere l'intuizione della rete globale come archivio e fonte di informazioni disponibili a tutti, descrive il web semantico, di cui rappresenta, in un certo senso, il "padre fondatore".

Il *World Wide Web*, ovvero l'insieme che raggruppa ogni tipo di contenuto pubblicato sulla rete globale, contiene una mole di dati impressionante e in continuo aumento che comprende pagine HTML statiche, contenuti dinamici, documenti e supporti multimediali.

Le pagine web e gli altri contenuti sono organizzati in siti, e collegati link ipertestuali ad altre pagine o contenuti, anche di altri siti. Ciò rende il web simile a un gigantesco archivio.

Qualora si desideri reperire un'informazione ricercandola all'interno del World Wide Web, si fa uso di un motore di ricerca che, a partire da alcune parole restituisce un elenco di pagine (oppure immagini, suoni, filmati) che hanno a che fare con i termini di ricerca, vale a dire che contengono nel loro testo tali parole chiave.

Tra i risultati ottenuti dalla ricerca vi possono essere centinaia, migliaia, talvolta milioni di pagine. Gran parte di esse, con elevata probabilità, pur contenendo le parole chiave specificate è di poco o nessun interesse per l'autore della ricerca. Un essere umano, dotato di capacità di discernimento è in grado di effettuare un filtraggio dei risultati della ricerca selezionando all'interno dei risultati quell'elemento o quegli elementi che rispondono alla sua domanda: questo poiché il cervello umano è in grado di distinguere un termine o un concetto non solo in base alla sintassi (ossia la sequenza di caratteri che formano una parola o una frase), ma anche in base all'interpretazione del suo significato, ossia la sua *semantica*.

Un essere umano che abbia appreso le proprietà del linguaggio è pertanto in grado di rilevare che, per esempio, la parola "Bianchi" ha una semantica nelle due asserzioni "Luca Bianchi è studente del Politecnico di Milano" e "Luca portava pantaloni bianchi". Allo stesso modo un essere umano è in grado di capire se una pagina web tratta dell'argomento effettivamente ricercato, oppure di tutt'altro.

Un agente automatico, un software per essere precisi, non ha le stesse capacità di discernimento a posteriori. Molti motori di ricerca moderni implementano euristiche sofisticate per poter stabilire, per esempio, che la parola "Bianchi" posta dopo un nome proprio di persona con ogni probabilità è un , così come altre tecniche probabilistiche per produrre risultati di ricerca che siano di interesse per l'utente, spesso sfruttando tecniche di *data mining* come quelle associative (ad esempio, produrre i risultati di ricerca tenendo presente che l'utente che cerca pagine inerenti con "vino" subito dopo ha ricercato pagine inerenti a "cantina").

Per sofisticate che siano, le tecniche di ricerca di informazioni non tengono in considerazione la semantica dei criteri di ricerca, per un motivo semplice: la grandissima maggioranza dei contenuti disponibili in rete non porta con sé informazioni che ne determinino la semantica. HTML, il linguaggio di markup con cui sono scritte le pagine web statiche e con cui sono presentate quelle dinamiche, è un linguaggio che non stabilisce la semantica dei contenuti ma solamente la loro forma. E questo, a un agente artificiale serve a ben poco.

Lo scopo del **web semantico** è quello di superare i limiti posti dai contenuti web tradizionali alla ricerca, catalogazione e scambio di informazioni da parte di agenti automatici. La costruzione del web semantico passa attraver-

so la costruzione di modelli che, partendo dalle risorse presenti sul web, ne definiscano la semantica attraverso la formulazione di relazioni tra di esse[1].

Le **ontologie** sono esattamente dei modelli formali, ottenuti definendo opportune relazioni (predicati) tra risorse basandosi sulle regole logiche derivate dalla logica predicativa del I ordine.

Lo scopo di questo progetto è studiare la disponibilità di strumenti idonei alla costruzione di ontologie, determinare quali debbano essere le sue caratteristiche ottimali ed eventualmente produrne un'.

1.1 Motivazioni ed obiettivi

Il consueto software di produttività individuale (o *stand-alone*) non sembra essere la soluzione migliore. Pur completo e di semplice utilizzo (caratteristiche che, ad ogni modo, è improbabile vedere implementate contemporaneamente), presenta numerosi limiti dettati dalla sua stessa natura.

Un software individuale è infatti strettamente legato all'utilizzatore: i prodotti derivati dall'uso dell'applicazione (documenti, file, progetti, ecc...) devono essere memorizzati in una postazione fissa, statica, seppur con tutte le possibilità di backup e di scambio messe a disposizione dalla rete. Si consideri un professionista che deve realizzare un progetto, di qualsiasi tipo, per esempio lo schema di un impianto. Fintantoché egli continua a lavorare da solo e da un'unica postazione, il software di produttività individuale resta la soluzione più adatta: i documenti restano memorizzati sulla stessa macchina dove risiede il software, accessibili e modificabili solo dall'autore. I primi problemi appaiono quando il professionista è chiamato a condividere il suo lavoro con altri soggetti, al fine di chiedere un parere oppure una vidimazione. L'avvento di Internet e, più in generale, delle tecnologie di rete ha reso questo passaggio sistematico, tramite l'utilizzo della posta elettronica diffuso ormai da molti anni.

Il sistema di scambio di dati dettato dalla posta elettronica è tuttavia assai rigido e diviene progressivamente insostenibile quando gli attori della comunicazione sono più di due, e il loro intervento non si limita alla sola lettura del documento ma debbono apportarvi delle modifiche. Dato che la comunicazione via posta elettronica (o altro sistema analogo di condivisione, si pensi ad esempio al peer-to-peer) è di per sé asincrona, l'onere di mantenere la consistenza tra le varie versioni del documento in circolazione è in capo al proprietario del documento, al coordinatore del progetto.

Quando da uno schema a progettista unico e diversi "revisori" si passa a uno schema di progettazione completamente distribuito (il progetto è realizzato in modo condiviso da un *team* o gruppo di lavoro), i canali di scambio di informazioni tradizionali divengono completamente inadeguati dal momento che ogni membro del gruppo non si limita a dare un parere ma contribuisce in modo attivo e sostanziale all'avanzata del progetto.

Nasce pertanto, in questo frangente, l'esigenza di un nuovo tipo di piattaforme software, che possa supportare l'azione congiunta, in tempo reale e/o asincrona, di più operatori sullo stesso progetto. Posto che le tecnologie esistenti non pongono limiti infrastrutturali allo scambio di dati tra due applicazioni installate su macchine diverse, nascono così le *applicazioni collaborative*, diffuse da tempo in ambito aziendale, che superano i limiti dettati dai software di produttività individuale. Tali applicazioni, pur mantenendo un'interfaccia e una logica applicativa separata per ciascun utente, leggono e scrivono i dati collocati su un'unica postazione condivisa (centralizzata), un server applicativo[1]. Un esempio è dato dal sistema di *subversioning* adottato da molti ambienti di sviluppo software.

Applicazioni di questo tipo, che implementano l'architettura client-server mantenendo però gran parte della logica lato client, prendono il nome di *fat-client* (ovvero client "pesante", che richiede installazione e configurazione su ogni macchina su cui deve operare).

Se sino a pochi anni or sono la connettività di rete era limitata a pochi luoghi, ben circoscritti (la cui estensione era limitata dalla presenza o meno del cablaggio di rete), l'evoluzione delle tecnologie e i conseguenti cambiamenti infrastrutturali hanno portato ad un'aumento progressivo dell'estensione dei luoghi in cui era disponibile una connessione ad internet, soprattutto grazie all'avvento delle WLAN, o reti wireless, molte delle quali disponibili in luoghi pubblici ed accessibili quindi tramite un semplice dispositivo portatile. La tendenza ad avere disponibilità di accesso alla rete "sempre e dovunque" non si è fermata, bensì ha avuto un'ulteriore accelerazione con l'introduzione delle reti cellulari a banda larga (UMTS e simili) che rendono di fatto possibile accedere alla rete globale da qualsiasi punto del territorio.

Il progresso infrastruttuale ha reso obsoleti i software collaborativi che erano la naturale evoluzione delle applicazioni di produttività individuale. I "fat-client" non rappresentano più un vantaggio, in quanto il loro uso prevede l'installazione del software su una particolare macchina, quand'anche portatile come un computer notebook. Cambiando postazione e non disponendo del software installato, risulta comunque impossibile accedere al progetto condiviso e lavorarvi. Inoltre se gli autori dell'applicazione vi apportano aggiornamenti, è necessario installarli su ogni .

Parallelamente all'evoluzione infrastrutturale, la rete globale ha attraversato e sta attraversando una fase di rinnovamento anche dal punto di vista del modo di interazione tra utente e software. Le informazioni reperibili sul web non sono più solamente statiche, sotto forma di pagine HTML, bensì consentono un livello di interazione maggiore con l'utente, che può inserirvi informazioni e ricevere risposte personalizzate. È la filosofia **Web 2.0**, che è alla base di un profondo

[1] un'alternativa è il paradigma peer-to-peer, in cui non esistono server centralizzati e ogni nodo della rete, quindi ogni istanza dell'applicazione contiene una propria copia dei dati condivisi e una logica appropriata per assicurare coerenza e consistenza con le altre copie del medesimo dato detenute dagli altri utenti

mutamento non solo nella forma dei contenuti web, ma anche e soprattutto nei termini del concetto stesso di utilizzo dello strumento informatico.

Grazie a tecnologie quali *Asynchronous JavaScript And XML* (AJAX), si rende possibile l'implementazione di interfacce grafiche di notevole applicate a pagine web dinamiche, accessibili da qualsiasi browser (i quali nel frattempo sono stati arricchiti di funzionalità al fine di consentire la visualizzazione e l'interazione con tali pagine dinamiche). La logica applicativa (elaborazione dati) è totalmente concentrata lato server, lasciando così al client (rappresentato dal solo browser web) soltanto l'onere della presentazione: tale paradigma, opposto a quello obsoleto del "fat-client", prende il nome di *thin-client*, client leggero.

I vantaggi di questo paradigma sono evidenti, e tra tutti quello più importante è forse la possibilità di accedere a dati condivisi da qualsiasi postazione che abbia accesso alla rete, dalle postazioni fisse in ufficio al PC di casa, oppure da uno *smartphone* di ultima generazione; inoltre, particolare non secondario, l'accesso può avvenire in qualsiasi momento, previa operazione di login.

Sono parecchie ormai le applicazioni web che permettono di agire in modo collaborativo previa registrazione e login. Si passa dai giochi on-line, alle cosiddette *social network*, a piattaforme come *Google Docs*[2] che rappresentano l'evoluzione, trasposta sul web, dei pacchetti per l'ufficio, e così via.

La domanda a questo punto del discorso suona perentoria: *quale è possibile dare alla costruzione del web semantico, alla luce dell'evoluzione delle tecnologie di rete e dell'espansione del Web 2.0?*

Ciò che occorre in realtà è avere degli strumenti, ovviamente software, in grado di soddisfare le esigenze di chiunque voglia sviluppare un modello oppure aggiornare, modificare, ampliare un modello che altre persone hanno già iniziato.

Gli strumenti a disposizione dei progettisti del web semantico esistono ormai da tempo: un tool come Protégé ha già raggiunto un livello di maturità notevole. Tuttavia esiste una particolarità che ha condizionato lo sviluppo di tali strumenti sin dal loro concepimento: molti di essi sono stati ideati e progettati per fornire supporto a modelli appartenenti a un dato ambito di ricerca. Protégé stesso è nato per supportare l'attività di ricerca in campo biomedico (di cui l'università statunitense di Stanford è uno dei centri più blasonati a livello internazionale), anche se il suo utilizzo, come recitano le note introduttive che si leggono sul suo sito web[3] è stato esteso ad aree tecnico-scientifiche molto diverse da quella per cui inizialmente era stato designato.

Un **primo obiettivo** è, pertanto, la possibilità di realizzare un tool non legato ad un particolare ambito di ricerca ma utilizzabile da più soggetti operanti in aree diverse.

[2] http://docs.google.com

[3] http://protege.stanford.edu

Un **secondo obiettivo**, legato al primo e non meno importante è fare in modo che tale strumento sia utilizzabile da un numero sempre più vasto di utenti, anche senza una formazione specifica riguardo agli standard emessi dal W3C per la formulazione di ontologie: un software, quindi, che permetta di costruire un modello, pienamente valido a livello di sintassi e semantica, senza che colui che lo sviluppa debba preoccuparsi di concetti come *namespaces*, *entities*, *URI* e via discorrendo (ferma restando una conoscenza almeno basilare di che cos'è un'ontologia e da quali elementi è composta).

Normalmente la prontezza all'uso, o per meglio dire l'immediatezza nell'apprendimento dell'uso di un software, è tanto più facile da raggiungere tanto più semplice ed immediata è l'interfaccia utente dell'applicazione: sotto questo punto di vista, è quindi un buon requisito per il software prevedere una modalità di editing che sia il meno testuale possibile, e ricca, pertanto, di elementi grafici.

Oltre ai due obiettivi individuati occorre non dimenticare l'esigenza inderogabile di assicurare la possibilità a un gruppo di lavoro di operare sulla stessa ontologia, in pieno approccio collaborativo. In questo caso le riflessioni da compiere sono diverse: esistono infatti vari modi di consentire una cooperazione, dalla filosofia *wiki* all'interazione in tempo reale (in stile chat), oppure soluzioni di compromesso tra questi due estremi. Tali riflessioni saranno trattate in seguito.

In sintesi ciò che questo progetto si propone di ottenere è un editor di ontologie, leggero (e quindi possibilmente thin-client), di facile apprendimento ed utilizzo *general-purpose*, orientato all'utilizzo collaborativo.

Inizia così il progetto di *Ontology Web-based Lightweight Interactive Editor* (OWLIE).

1.2 Metodologia di progetto

La realizzazione di OWLIE, come per qualsiasi software, deve necessariamente seguire le varie fasi del ciclo di vita dell'applicazione, così come indicato dai canoni dell'ingegneria del software.

Il processo di sviluppo di OWLIE passa attraverso varie fasi, per concludersi con il rilascio di una versione *usabile* del software, il che non significa nel modo più assoluto che il processo non possa essere ripreso e proseguito. La prima fase del processo, come è naturale, è la scelta del metodo con cui procedere, o in altri termini, scegliere un *modello* di sviluppo che stabilisca tempistiche e modalità con cui ogni passo, dallo studio del problema sino al rilascio dell'applicazione, deve essere svolto (quindi una sorta di *modus operandi*).

Per la scelta del modello sono stati considerati diversi fattori. In primo luogo, OWLIE è un progetto individuale, inquadrato in un lavoro di tesi di laurea specialistica e finalizzato alla realizzazione di un'applicazione di piccole dimensioni,

nella previsione di un possibile, ed auspicabile ampliamento futuro. In virtù di queste prime considerazioni, è stato scelto un metodo di sviluppo *a cascata*, composto cioè di alcune macro-fasi eseguite in (rigorosa) sequenza (l'output di una fase è l'input della successiva).

Il modello a cascata è in effetti il metodo di sviluppo software più semplice in assoluto e quello che più si adatta alla situazione in esame. Le fasi di sviluppo individuate sono le seguenti:

- *fasi preliminari*
    1. *determinazione degli obiettivi*
    2. *stato dell'arte* (studio di fattibilità)
- *analisi dei requisiti – determinazione degli scenari d'uso*
- *design dell'applicazione*
- *implementazione*
- *testing*
- *documentazione*
- *rilascio*

Una fase preliminare effettuata con metodo è un lasciapassare indispensabile per poter effettuare le fasi successive, che portano alla realizzazione dell'applicazione. Dopo aver stabilito quali sono gli obiettivi del progetto è necessario intraprendere uno studio dello stato dell'arte, per determinare se esistono allo stato attuale applicazioni che già soddisfano, in tutto o in parte, le alla base del progetto. Questa fase presenta le caratteristiche di studio di fattibilità: l'esistenza di alternative complete o parziali, così come la possibilità di sfruttare altri software come base per la realizzazione del nuovo, consente di stabilire se è conveniente insistere sul progetto, e in caso positivo decidere in che modo.

L'importanza dello studio dello stato dell'arte è tale da caratterizzare un capitolo di questa trattazione (il capitolo 2).

Le fasi successive sono simili a quelle di un processo software relativo a un progetto di maggiori dimensioni. Si passa da un'indispensabile fase di design dell'applicazione (scelta del design-pattern e del modello architetturale) alla fase implementativa vera e propria, che prevede la codifica mediante un linguaggio di programmazione (linguaggio scelto già in una fase precedente).

Durante l'implementazione possono presentarsi occasioni in cui il modello architetturale scelto in precedenza diviene troppo restrittivo oppure inadatto ad implementare alcune caratteristiche dell'applicazione. Il modello a cascata come già accennato impone la rigorosa sequenzialità delle fasi progettuali, tuttavia è possibile rilassare questo vincolo prevedendo una sorta di "feedback"

quando in una certa fase si rivelano problematiche che mettono in discussione le scelte fatte in precedenza. Il feedback consente di effettuare piccole all'output della fase precedente: ad esempio, se in fase implementativa si rivela necessario aggiungere un modulo, ciò è consentito, a patto che questo cambiamento appaia nella documentazione.

1.3 Durata e tempistiche del progetto

Il diagramma temporale rappresentato in figura 1.1 schematizza le del processo di sviluppo di OWLIE. Da notare il fatto che, nonostante gran parte delle attività sia stata svolta in modo sequenziale, alcune di esse sono state svolte in parallelo con attività iniziate in precedenza.

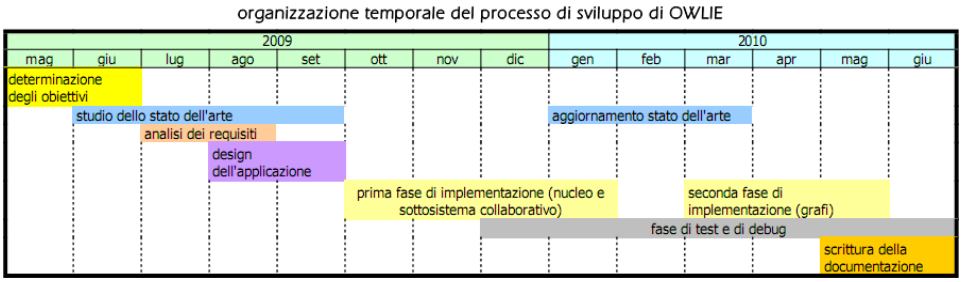


Figura 1.1: *Diagramma temporale del processo di sviluppo di OWLIE*

In totale, la realizzazione di OWLIE ha richiesto circa un anno, e una parte considerevole di questo lasso di tempo è stata assorbita dalla fase implementativa vera e propria, che è stata spezzata in due sottofasi per scelte organizzative.

1.4 L'origine dell'acronimo OWLIE

L'acronimo OWLIE è stato designato come identificativo sia del progetto, sia dell'applicazione che esso ha prodotto. Tuttavia la scelta di tale sigla (che sta per *OWL Web-based Lightweight Interactive Editor*) è avvenuta in una fase ormai avanzata del processo di sviluppo.

La sigla è stata costruita in modo che il suo significato ribadisse gli obiettivi alla base del progetto: un editor per ontologie (scritte in OWL), che sia costituito da un'applicazione web (*web-based*), che supporti la collaborazione di più utenti (*interactive*, inteso nel senso più ampio di questo aggettivo, ovvero non di interazione solo tra utente e software ma anche e soprattutto tra utente e utente) e che sia leggero (*lightweight*) sia nel senso letterale del termine (contenuto consumo di risorse) sia nei termini di ridotta complessità dell'utilizzo e di facilità di apprendimento.

1.5 Struttura del testo

Questa trattazione è organizzata in tre capitoli, oltre all'introduzione.

Il capitolo 2, che segue questo paragrafo, tratta in linee generali gli standard proposti dal *World Wide Web Consortium* (W3C) per la codifica e lo scambio di informazioni nell'ambito del progetto del web semantico, focalizzando l'attenzione su due linguaggi in particolare: *Resource Description Framework* (RDF) e *Ontology Web Language* (OWL) che ne rappresenta un'applicazione. Nei paragrafi successivi del capitolo saranno invece esaminati i principali strumenti software utilizzati dagli operatori del semantic web per il progetto e la realizzazione di ontologie, distinguendo tra programmi per l'utilizzo individuale, applicazioni collaborative ed editor esclusivamente grafici.

L'ultima parte del capitolo è dedicata all'esame di Protégé, uno tra i software più completi e diffusi nell'ambito del web semantico, con approfondimenti circa l'applicazione principale e i suoi prodotti derivati. L'insistenza su Protégé è dovuta al legame che tale software ha con il progetto di OWLIE, oggetto del capitolo successivo.

Il capitolo 3 è dedicato alle fasi di progetto che precedono l'implementazione vera e propria, pertanto al suo interno sono trattate, ognuna in una apposita sezione, le fasi di analisi dei requisiti, determinazione dei casi d'uso e design architetturale e funzionale dell'applicazione.

Il capitolo 4 esamina le scelte effettuate in fase di codifica, descrivendo l'applicazione nei suoi componenti con particolare attenzione alle modalità di interazione tra di essi.

Dal capitolo è stato escluso il manuale d'installazione ed uso, in quanto ritenuto troppo pesante per essere inserito in un capitolo. Considerato tra l'altro che, dal punto di vista dell'utilizzatore finale nonché di un programmatore intenzionato ad ampliare e migliorare l'applicazione, un manuale d'uso è molto più utile se costituisce un capitolo a sé stante all'interno della trattazione, se non addirittura esterno, è stata fatta la scelta di includerlo all'interno delle appendici.

Il capitolo 5 intende esaminare, con l'intento di autocritica, l'applicazione sviluppata considerando il grado di raggiungimento degli obiettivi di cui al presente capitolo introduttivo. Per gli obiettivi raggiunti solo parzialmente si indicano anche le possibili tracce per un futuro ampliamento o miglioramento del software, che si ritiene sia auspicabile oltre che normale.

Le appendici, come già accennato, trattano di argomenti squisitamente tecnici che si è ritenuto non inserire direttamente nel discorso. La prima appendice vuole costituire un breve manuale per l'installazione e l'uso dell'applicazione ed è destinata sia ad un utente finale, sia a uno sviluppatore che intenda proseguirne lo sviluppo. A queste figure è però dedicata in modo particolare la seconda ap-

pendice, che descrive, nei limiti dettati da opportune esigenze di sintesi, come compilare i sorgenti all'interno di un ambiente di sviluppo.

Capitolo 2

Lo stato dell'arte

Il web semantico è un'area di interesse piuttosto recente all'interno dell'ingegneria dell'informazione, ed è in rapida evoluzione. Il progresso dell' della conoscenza, vale a dire la disciplina accademico-scientifica che si occupa della ricerca nell'ambito del web semantico, è decisamente rapido e il numero di applicazioni software sviluppate per risolvere problemi in quest'ambito è destinato a crescere.

Questo capitolo approfondisce una delle fasi iniziali del processo di sviluppo di OWLIE, ovvero lo studio dello *stato dell'arte*: uno sguardo all'evoluzione del web semantico a livello di standard internazionali, seguito da una ricerca sulle principali realtà software che operano a livello di ontologie.

Questa ricerca ha innanzitutto lo scopo di osservare se esistono già, nel parco software globale, applicazioni che rispondono ai medesimi requisiti del progetto che è argomento di questa tesi, e in esse valutare se e in che modo gli obiettivi vengono raggiunti. In alternativa lo scopo della ricerca è rilevare eventuali applicazioni esistenti, che non rispondono ai requisiti richiesti ma che sono strutturate in modo tale da renderne possibile un'estensione.

Al termine del capitolo verranno tratte le dovute conclusioni.

2.1 Web semantico e standard W3C

Figura 2.1: *Il logo del Semantic Web secondo il W3C*

L'obiettivo del W3C è di trasformare l'attuale rete globale in una "rete di dati" (*web of data*[2]) ovvero associare ad ogni documento, file o altra risorsa presente in rete una descrizione che ne specifichi il contesto semantico, in un formato adatto all'elaborazione (interrogazione, modifica, scambio, ecc...) tramite agenti automatici.

Il W3C ha definito una serie di standard, cioè formati e regole raccomandati, che stabiliscono il modo in cui devono essere indicate le risorse (URI o

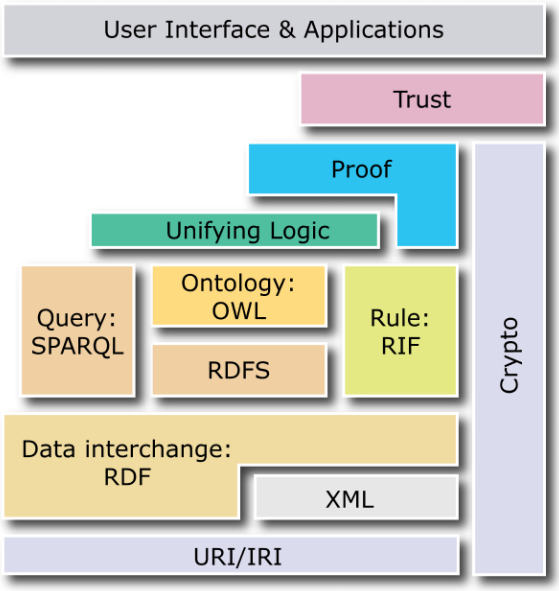


Figura 2.2: *Struttura a livelli del Semantic Web (da* `http://www.w3.org`*)*

più in generale *Internationalized Resource Identifier* (IRI)), la forma in cui i dati devono poter essere scambiati (XML), il modo in cui le risorse devono essere descritte (RDF), la semantica che coinvolge le relazioni tra risorse (RDF Schema, OWL, OWL 2), i linguaggi di interrogazione (SPARQL). La combinazione di questi standard dà luogo al *web semantico*, di cui uno schema a livelli è riportato in figura 2.2.

Il web semantico è un progetto in continua e rapida evoluzione. Nuovi standard rimpiazzano le normative esistenti allo scopo di aumentare la espressiva dei linguaggi o di supportare algoritmi più potenti. Gli strumenti software esistenti devono perciò continuamente essere aggiornati per adeguarsi all'evoluzione dell'architettura.

2.1.1 RDF

RDF è lo strumento formalizzato dal W3C per la codifica e lo scambio di informazioni strutturate e si basa sul concetto di *descrizione* di risorse. Per risorsa si intende qualunque cosa che è possibile descrivere o, in altri termini, qualunque cosa di cui è possibile definire ed elencare le proprietà. In RDF, quindi, possono essere descritti sia risorse virtuali (file, documenti, ecc...), sia oggetti reali e tangibili.

Ogni risorsa in RDF è identificata da una stringa di testo univoca, ossia un *Uniform Resource Identifier* (URI). L'assegnazione dell'URI a una data risorsa è arbitraria, specie nel caso di oggetti reali.

In sostanza, un modello RDF è composto da asserzioni, o *statement*, costituite da una tripla <*soggetto* − *proprietà* − *valore*>. Soggetto e sono sempre risorse, quindi identificate da un URI, mentre il valore è una stringa. Se anche il valore è espresso sotto forma di URI, rappresenta anch'esso una risorsa.

La rappresentazione fisica di un modello costruito in RDF passa attraverso l'operazione di *serializzazione*, che consiste nell'elencare, secondo una data sintassi, le risorse e le loro proprietà in modo che possano essere scritte in un file di testo. Esistono varie modalità di serializzazione: alcune, come *Notation-3* (N3)[1] o *N-TRIPLE*[2] forniscono una rappresentazione del modello piuttosto compatta e facilmente interpretabile da occhio umano, a fronte di uno scarso numero di applicazioni informatiche. Il W3C raccomanda invece di utilizzare *eXtensible Markup Language* (XML) come modalità di serializzazione.

La serializzazione di RDF in un file XML produce un documento dal tipo MIME[3] `application/rdf+xml`. Il modello RDF può essere serializzato in XML seguendo due diverse sintassi: la sintassi estesa e quella abbreviata. La scelta della sintassi da utilizzare è arbitraria e non preclude né il *parsing* da parte di applicazioni informatiche, né lo scambio delle informazioni: la sintassi abbreviata è in genere più facilmente interpretabile dall'occhio umano.

Il codice sottostante riporta un esempio di serializzazione in XML, con sintassi abbreviata, che riporta lo statement di seguito espresso in lingua italiana: «Fabio Grassani è l'autore di owlie».

Frammento di codice 2.1: *Esempio di serializzazione di RDF in XML*

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:au="http://www.elet.polimi.it/owlie/rdf/schema">
<rdf:Description
  about="http://www.elet.polimi.it/owlie/owlie.war">
  <au:author>Fabio Grassani</au:author>
</rdf:Description>
</rdf:RDF>
```

Nell'esempio mostrato, mentre la risorsa "owlie" è espressa mediante un URI, la stringa "Fabio Grassani" non costituisce una risorsa, poiché non è espressa da un URI, ma solamente il valore che assume la proprietà "author".

Appare evidente come, basandosi sulla semplice sintassi di RDF, sia possibile descrivere ogni risorsa, e i modi per farlo sono praticamente infiniti. Ciò significa che, pur avendo stabilito una sintassi per rappresentare un modello, RDF da solo non è in grado di definirne una *semantica* univoca.

[1] http://www.w3.org/DesignIssues/Notation3.html

[2] http://www.w3.org/TR/rdf-testcases/#ntriples

[3] *Multipurpose Internet Mail Extensions*, standard di internet che identifica il formato di documento, applicato in origine ai soli messaggi di posta elettronica, da cui l'acronimo.

2.1.2 RDF-Schema

RDF-Schema[4] (RDFS, RDF-s, RDF(S) oppure RDF/S) è un'estensione di RDF realizzata allo scopo di introdurre una serie di predicati standard per definire una semantica univoca a un modello RDF.

RDF-Schema formalizza le definizioni di classe, sottoclasse, proprietà e sottoproprietà, indispensabili per determinare la semantica di un modello realizzato in RDF.

Per *classe* si intende una risorsa che costituisce un tipo a cui altre risorse possono appartenere. Per ogni classe è possibile specificare se essa è *sottoclasse* di un'altra, creando in questo modo una gerarchia di classi.

Una *proprietà* è una risorsa che costituisce il legame tra altre due risorse. Può essere *sottoproprietà* di un'altra qualora ne costituisca una specializzazione (ovvero l'esistenza del legame tra due risorse è subordinato alla presenza di un legame più forte). Una proprietà può avere un *dominio* (insieme di una o più classi a cui una risorsa deve appartenere per poter instaurare il legame) e un *codominio* (insieme di una o più classi a cui una risorsa deve appartenere per poter essere raggiunta da un legame).

Con la definizione di RDF-Schema è dunque possibile realizzare delle *ontologie*, cioè rappresentazioni formali della conoscenza tramite la descrizione di oggetti e concetti in un certo ambito, o dominio, e la descrizione delle proprietà che intercorrono tra di essi.

Molti dei costrutti definiti in RDFS vengono utilizzati da OWL.

2.1.3 OWL

OWL è una famiglia di linguaggi formali, approvati dal W3C, creati allo scopo di formulare *ontologie*. L'acronimo non si riferisce, quindi, a un linguaggio ben definito, ma piuttosto ad una serie di linguaggi, che presentano sintassi ed espressività differenti.

La prima versione di OWL riconosciuta dal W3C fu formalizzata nel 2004[3] e restò una raccomandazione a livello globale fino al riconoscimento della versione successiva, inizialmente nata come revisione di OWL nel 2006[4] e denominata OWL 1.1, benché in seguito ribattezzata *OWL 2* quando i cambiamenti apportati alla sintassi e all'espressività andarono ben oltre i limiti di una semplice revisione. OWL 2 è da ottobre 2009 il nuovo standard raccomandato dal W3C, sebbene OWL sia tuttora largamente utilizzato.

In base al livello di espressività, OWL si divide in tre profili principali, di seguito elencati in ordine crescente di potenza espressiva, ciascuna delle quali è un'estensione della variante precedente.

- **OWL Lite** – supporta le gerarchie di classi e vincoli di cardinalità semplici (cioè in cui la cardinalità assume valore 0 oppure valore 1). Questa

[4] le cui specifiche W3C si trovano all'indirizzo `http://www.w3.org/TR/rdf-schema`

variante del linguaggio fu formalizzata precipuamente per consentire lo sviluppo di strumenti per la creazione di vocabolari e *tesauri*, ma non ha mai avuto un utilizzo particolarmente esteso.

- **OWL DL** – è un linguaggio creato allo scopo di fornire la massima espressività possibile mantenendo però sia la *completezza* del modello logico, sia la *decidibilità*. DL è un acronimo che sta per *descriptive logic*, logica descrittiva.

- **OWL Full** – è un linguaggio creato allo scopo di mantenere compatibilità con *RDF-Schema* (RDFS), di cui è un'estensione semantica. In breve, cadono molti dei vincoli espressi da OWL DL. Nonostante sia possibile in OWL Full descrivere modelli decidibili, in generale la decidibilità non è garantita, e in pratica non esistono algoritmi di inferenza (*reasoner*) che supportino OWL Full.

La semantica di OWL DL (e quindi, anche di OWL Lite) è basata su una classe di *logiche descrittive*, che a loro volta rappresentano un sottoinsieme delle logiche predicative del I ordine. In particolare, nel caso di OWL DL, la semantica corrisponde a quella della logica *SHOIN (D)*, che è completa (per ogni formula, o la formula stessa o la sua negazione appartengono alla logica) e decidibile (esiste un algoritmo capace, in un numero finito di passi, di dire se una formula è o non è un teorema della logica). Per esigenze di sintesi, le caratteristiche della logica SHOIN (D) non sono riportate in questo paragrafo.

Per ogni linguaggio parte della famiglia di OWL esistono varie sintassi. Il W3C raccomanda la serializzazione su RDF+XML utilizzando un sottoinsieme della semantica di RDFS, ma esistono altre possibili sintassi, come la *Sintassi Manchester*.

2.1.4 OWL 2

Come già accennato nel paragrafo precedente, OWL 2 ha soppiantato da pochi mesi (ufficialmente, dal 26 ottobre 2009) OWL come standard raccomandato dal W3C[5].

Rispetto a OWL, di cui è un'evoluzione, presenta poche ma importanti differenze. Innanzitutto la semantica di OWL 2 non è più basata sulla logica SHOIN (D), bensì sulla logica *SROIQ (D)*, che a fronte di una maggiore espressività non cede in completezza né in decidibilità (essendo stati sviluppati nuovi algoritmi di inferenza, differenti rispetto a quelli operanti su OWL).

L'evoluzione in ambito semantico coinvolge anche la sintassi. Essendo migliorata la potenza espressiva del linguaggio, è stato necessario aggiornare le varie sintassi di OWL aggiungendovi i costrutti mancanti.

[5]Le specifiche di OWL 2 si trovano sul web all'indirizzo `http://www.w3.org/TR/owl2-overview`

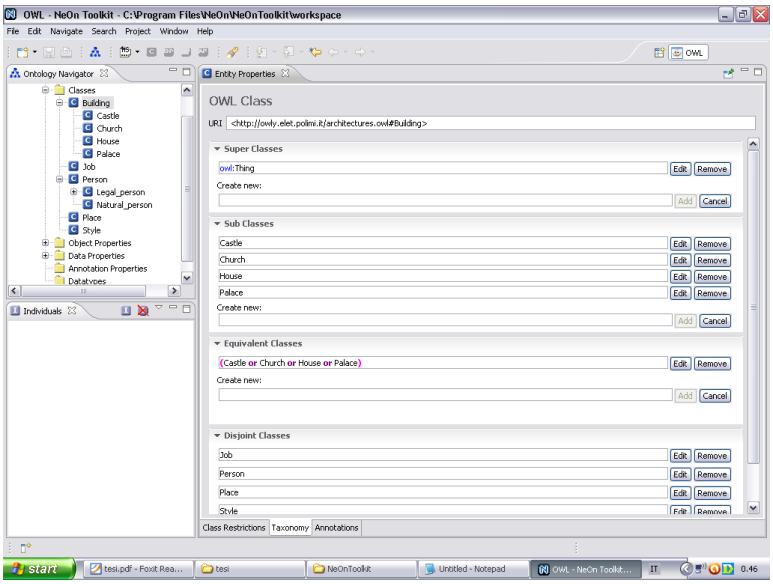


Figura 2.3: *NeOn Toolkit 1.2.3 in ambiente Windows*

2.2 Obiettivi e filoni di ricerca

Il semantic web è un campo in continua espansione, e nonostante la giovinezza di quest'ambito di ricerca, dispone di un parco software piuttosto variegato, sebbene non ancora maturo e quindi decisamente aperto all'innovazione.

L'esigenza di realizzare applicazioni per progettare e costruire ontologie nasce dal fatto che, pur essendo OWL e RDF strumenti formali molto potenti per rappresentare descrizioni della realtà, non è certamente comodo agire direttamente su un documento XML, specialmente nel caso di ontologie di dimensioni medie o grandi. Inoltre un'interfaccia utente, più o meno complessa, permette di ontologie anche a persone con poca o nulla dimestichezza verso XML e i linguaggi di markup in generale, sebbene sia richiesta, da parte dell'utente, una conoscenza almeno basilare dei costrutti di OWL e della sua espressività.

Allo stato attuale, la ricerca segue due filoni principali: la ricerca di un metodo di rappresentazione delle informazioni che sia al tempo stesso completo e compatto, e suò fronte opposto lo sviluppo di sistemi collaborativi, in piena sintonia con i principi del Web 2.0.

La rappresentazione delle ontologie rappresenta il primo nodo cruciale della ricerca. Come già osservato in precedenza, OWL e in generale tutti i linguaggi che permettono di realizzare modelli semantici hanno un elevato grado di espressività, che si traduce in una sintassi particolarmente complessa. Presumendo che colui che progetta un'ontologia sia un esperto del dominio da rappresentare ma non un esperto informatico, il software di editing deve essere in grado di fornirgli una vista dell'ontologia che "mascheri" la sintassi sottostante e mostri soltanto i tratti salienti dell'ontologia, ovvero le risorse e i legami tra di

esse[5].

Tuttavia risulta assai difficile che in una sola vista sia possibile rappresentare tutti gli aspetti salienti della base di conoscenza, prima di tutto date le dimensioni dell'ontologia, che possono risultare particolarmente estese. Diversi tentativi di realizzare dei visualizzatori esclusivamente *grafici* (ove le risorse sono nodi di un grafo, bidimensionale o tridimensionale a seconda dei casi) raggiungono solo in parte gli obiettivi di completezza e compattezza della rappresentazione, spesso perché il numero di risorse rappresentabili è inferiore alle dimensioni dell'ontologia stessa.

Applicazioni che si prefiggono lo scopo di fornire un'interfaccia completa, in grado di far interagire l'utente con ogni aspetto dell'ontologia, acquisiscono la forma di ambiente di sviluppo integrato, molto simile in quanto a ricchezza funzionale e complessità a un *Integrated Development Environment* (IDE) per la progettazione e lo sviluppo di software. Non a caso, software come Protégé[6], TopBraid e NeOn Toolkit[8] si basano su Eclipse, un ambiente di sviluppo integrato inizialmente destinato all'implementazione di applicazioni Java.

Questi software sono organizzati in *viste* e *prospettive*, ognuna delle quali presenta all'utente una parte dell'ontologia (concetti piuttosto che ruoli o individui), superando così le limitazioni dettate dalle dimensioni della base di conoscenza. Normalmente tali applicazioni possono essere estese da una serie di moduli aggiuntivi, o plugin, i quali adducono le funzionalità di validazione, *reasoning* e soprattutto visualizzazione grafica.[6]

2.2.1 Modalità di rappresentazione grafica di un'ontologia

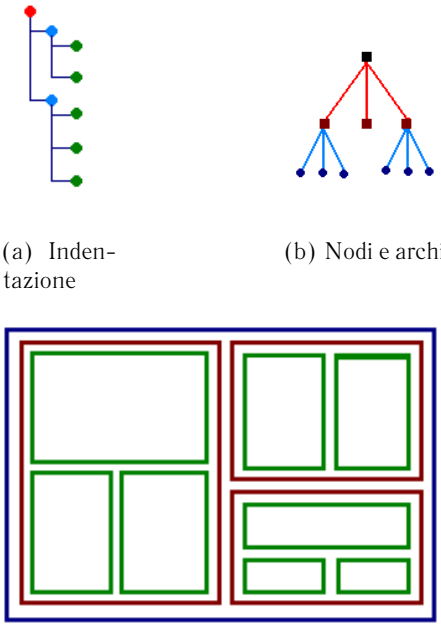
Le tecniche esistenti per la visualizzazione di ontologie possono essere raggruppate in quattro modalità principali, ovvero rete (*network*), gerarchica (*tree*), prossimità (*neighborhood*) e vista iperbolica (*hyperbolic view*).[6].

La vista a *rete* è probabilmente il sistema di organizzazione grafica delle informazioni più semplice e applicabile a ogni ontologia, in quanto consiste in un grafo in cui i nodi identificano gli elementi dell'ontologia stessa e gli archi le connessioni tra di essi. Un esempio di vista a rete è osservabile in GrOWL, un editor per OWL esclusivamente grafico, descritto nel paragrafo 2.3.3 e rappresentato in figura 2.7.

Nei casi in cui esistano relazioni gerarchiche tra membri di un'ontologia, ovvero tassonomie (si pensi per esempio alle gerarchie di concetti, determinate dall'operatore di sussunzione), può risultare significativa una vista *gerarchica* o ad *albero*. Ne sono state studiate ed implementate almeno tre varianti.

La modalità più diffusa è indubbiamente l'indentazione (figura 2.4(a)), utilizzata in gran parte degli editor delle ontologie per mostrare la gerarchia dei concetti (e talvolta anche dei ruoli). L'albero classico costituito da nodi e da archi

[6] la sua più recente versione, Protégé 4, è basata sul framework OSGi di Eclipse



(a) Indentazione

(b) Nodi e archi

(c) Mappa

Figura 2.4: *Tre diverse modalità di visualizzazione gerarchica*

(figura 2.4(b)) è altresì molto utilizzato, specialmente nei visualizzatori grafici associati agli editor (come ad esempio i plugin di visualizzazione di Protégé).

Di notevole interesse anche la terza variante gerarchica, la *treemap* (altrimenti definita "scatola cinese") utile per mostrare graficamente relazioni gerarchiche in uno spazio limitato facendo uso di strutture geometriche rappresentate una all'interno dell'altra.[7].

Le viste ad albero sono adatte a rappresentare qualunque tipo di relazione gerarchica. Tuttavia, dal momento che relazioni di tipo gerarchico ("is-a") coinvolgono solamente una parte dell'ontologia, tutti gli elementi che non sono parte di legami rappresentabili in una struttura ad albero, sono solitamente esclusi e devono essere mostrati in una vista separata.

Altri metodi di visualizzazione si concentrano su un elemento attivo e mostrano gli elementi in diretta connessione con esso (e che risultano quindi in sua prossimità). Possono costruire viste bidimensionali (grafi di *prossimità*) o tridimensionali (viste *iperboliche*).

2.2.2 Da visualizzatore a editor grafico

Sebbene la ricerca nel campo della visualizzazione delle ontologie, come visto nel paragrafo precedente, abbia raggiunto degli obiettivi interessanti, la creazione di editor che facciano uso di costrutti esclusivamente grafici (o, in al-

tri termini, che consentano la modifica dei nodi e degli archi del grafo) è tuttora un filone di ricerca attivo.

Le poche applicazioni sviluppate in questo senso sono per lo più programmi desktop (orientati quindi all'uso individuale, essendo installati sul medesimo elaboratore su cui agisce l'utente) scritti principalmente in Java. Un esempio è GrOWL, descritto nel paragrafo 2.3.3, ottimizzato per ontologie di piccole o medie dimensioni e che consente di rappresentare tutti gli elementi della logica descrittiva definiti da OWL.

Esistono editor grafici ideati e implementati per costituire estensioni (plug-in) di ambienti di sviluppo di ontologie di utilizzo più generale, come ad Protégé. *EzOWL* è un software sviluppato da un gruppo di lavoro sud-coreano[9] rilasciato sotto forma di plugin per Protégé, che dà modo all'utente di costruire un'ontologia (o modificarne una preesistente) agendo sui nodi e sugli archi di un grafo. La possibilità di azione dell'utente è tuttavia parziale (non sono rappresentati tutti i possibili costrutti di OWL) e non sostituisce completamente l'utilizzo dell'editor classico.

Allo stato attuale, non esiste un software basato sul web, che sfrutti un'interfaccia di tipo AJAX o un suo equivalente, in grado di offrire la possibilità di costruire graficamente un'ontologia. Esistono dei visualizzatori, anche piuttosto complessi, come le applet di visualizzazione di Knoodl (vedasi paragrafo 2.3.2), che tuttavia non implementano alcuna interattività.

In questo ambito, il progetto di OWLIE, può pertanto costituire un elemento di novità e di avanzamento della ricerca.

2.3 Situazione attuale del parco software

Senza considerare le applicazioni specializzate per creare solo certi tipi di modelli, riservate quindi a esperti in un ambito definito, si possono individuare due grandi categorie di software per la costruzione di ontologie.

Vi sono le applicazioni orientate ad un utilizzo individuale, che prevedono un'installazione dedicata (come i classici software di produttività personale), definite *applicazioni stand-alone*.

Nell'altra categoria rientrano le applicazioni orientate ad un utilizzo , tipicamente basate sul web e accessibili tramite browser (con alcune eccezioni) in sintonia con l'approccio del Web 2.0. Queste applicazioni possono essere definite come *ambienti collaborativi*.

Il confine tra applicazioni stand-alone e collaborative non sempre è ben delineato. Esistono software che, pur offrendo un ambiente orientato all'utilizzo individuale, sono dotati della possibilità di accedere a risorse condivise.

2.3.1 Le applicazioni stand-alone

Un'applicazione stand-alone è tipicamente un software non basato sul web, pertanto installabile e configurabile su una postazione singola, destinato ad essere utilizzato a livello individuale.

Non essendo soggette alle restrizioni che limitano l'estendersi delle funzionalità e la complessità dell'interfaccia delle applicazioni nel caso di software basato sul web, molte delle applicazioni stand-alone sono particolarmente complesse dal punto di vista strutturale o funzionale, e presentano una *Graphic User Interface* (GUI) solitamente ricca e articolata. Il parco software è ab-

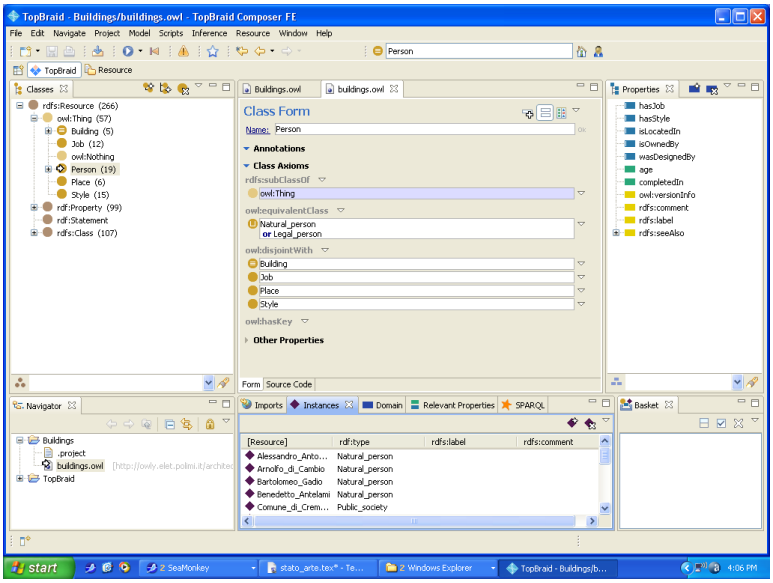


Figura 2.5: *Screenshot di TopBraid Composer Free Edition*

bastanza ricco, considerando sia le applicazioni commerciali (destinate a un utilizzo professionale) sia quelle gratuite od open-source. Se si escludono le applicazioni sviluppate per uno scopo preciso (ad esempio, la realizzazione di ontologie in un ambito ben definito, o la creazione di vocabolari), gran parte del software per lo sviluppo di ontologie in OWL o RDF presenta una serie di caratteristiche comuni o simili:

- *modularità* — un software di medio-grandi dimensioni è solitamente strutturato in moduli, che possono essere aggiunti o rimossi a seconda delle esigenze dell'utente. Tali moduli prendono il nome di *plug-in* (o, più raramente, *add-on* o *add-in*) e tipicamente implementano una funzionalità ben precisa (ad esempio, una particolare visualizzazione grafica oppure il supporto per un certo formato di file).

- *funzionalità* — un editor di ontologie generico (vale a dire, senza un preciso ambito di utilizzo) implementa, tramite il modulo principale o tramite

uno o più plug-in, tutte le operazioni legate allo sviluppo di un'ontologia. Oltre alle operazioni fondamentali (aggiunta di una sottoclasse, formulazione di un'asserzione per un'istanza di una classe, ecc...) sono pertanto di norma disponibili uno o più *reasoner* che permettono di ricavare nuove informazioni, sulla base delle informazioni asserite, ed effettuare controlli di consistenza sulla base di conoscenza.

- *interfaccia grafica* – per quanto differenti, le varie GUI tendono a seguire uno stile uniforme nella rappresentazione dell'ontologia, separando le risorse in base al tipo (un albero per le classi, un altro albero per le proprietà, una lista ordinata per gli individui). Di norma le risorse sono rappresentate da simboli standard, rispettivamente un tondo di colore arancio o marrone per le classi, un rettangolo (di diverso colore a seconda del tipo di) per le proprietà, un rombo di colore viola o nero per gli individui.
  Le classi anonime (ossia i concetti arbitrari) sono solitamente espressi in forma testuale (sintassi Manchester[10]).

Due software che presentano le suddette caratteristiche sono **Protégé** e **TopBraid Composer**[7]. Il primo è un software open-source e gratuito mentre il secondo è un software commerciale sviluppato dalla software-house statunitense *TopQuadrant*. Di Protégé e delle sue caratteristiche si parlerà approfonditamente nel paragrafo 2.4.

TopBraid Composer rappresenta, allo stato attuale, uno dei prodotti più completi per il progetto e lo sviluppo di ontologie. Come in altri software commerciali, ne esistono varie distribuzioni che si differenziano in base al costo e alle funzionalità offerte. La versione *free* è già di per sé particolarmente completa e, oltre alle funzionalità di base permette di utilizzare un *reasoner* per effettuare controlli di consistenza e operazioni di inferenza. Per impostazione predefinita, è disponibile un solo motore inferenziale, TopSPIN, specifico per TopBraid Composer.

In figura 2.5 è rappresentato un esempio di utilizzo di TopBraid Composer. Sono evidenti, nell'interfaccia utente di TopBraid, analogie rispetto alla GUI di Protégé, di cui vi sono esempi nelle figure 2.9 e 2.14.

Dal punto di vista architetturale, TopBraid si appoggia sulle librerie di Jena, esattamente come Protégé-OWL (cfr. paragrafo 2.4.1), e come quest'ultimo supporta lo standard OWL 1.0 (ma non OWL 2.0).

2.3.2 Gli ambienti collaborativi

Per ambiente collaborativo si intendono quelle applicazioni (tipicamente ma non necessariamente basate sul web) che consentono l'interazione di più utenti con un'ontologia (con opportune misure per il controllo degli accessi). L'interazione può essere simultanea (pertanto due o più utenti possono modificare

[7] http://www.topquadrant.com/composer

contemporaneamente l'ontologia, visualizzando le modifiche apportate in tempo reale) oppure sequenziale (come in una *wiki*). Si tratta di due approcci duali, ma che possono essere combinati.

Il parco software collaborativo non è particolarmente ampio. Esclusi Collaborative Protégé e WebProtege (di cui si occupano in modo approfondito i paragrafi 2.4.3 e 2.4.4), vi sono due applicazioni degne di nota, segnalate dal *VoCamp*[8].

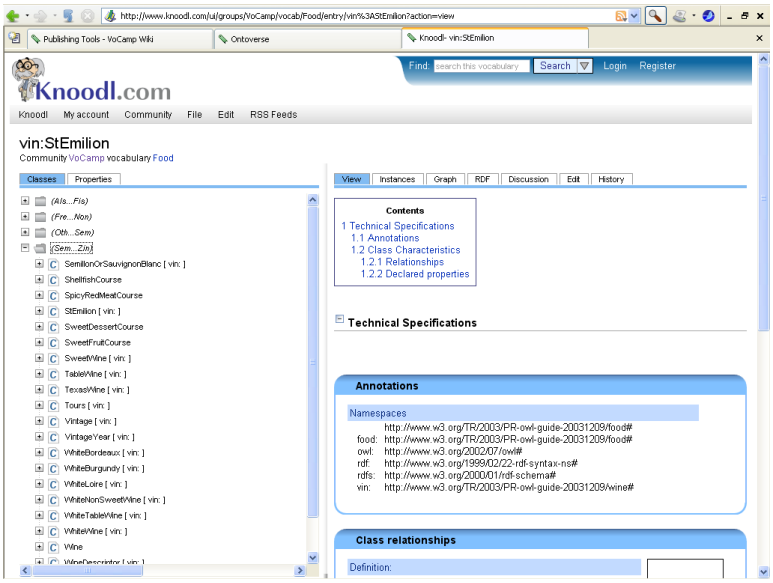


Figura 2.6: *Screenshot di Knoodl*

- **Ontoverse** (`http://www.ontoverse.org`)
  Ontoverse è un'applicazione web (accessibile cioè via browser) sviluppata e mantenuta dall'Università di Düsseldorf con il patrocinio del Ministero per l'educazione e la ricerca del Governo Federale tedesco.
  La piattaforma – liberamente accessibile previa registrazione e login – permette la costruzione cooperativa di ontologie in ambito scientifico, con lo scopo di agevolare la collaborazione tra esperti operanti in diversi settori.
  Allo stato attuale il sito consente la registrazione e il login, ma per qualche ragione che non è dato conoscere ogni tentativo di accedere alle ontologie, anche solamente per una semplice visualizzazione, sfocia in un messaggio di errore, segno eloquente di uno stallo nel mantenimento della piattaforma.

- **Knoodl** (`http://www.knoodl.com`)
  Knoodl è una piattaforma aperta, basata sul web e organizzata in *com-*

[8] `http://vocamp.org`

munities, che combina la creazione di ontologie con la filosofia *wiki*, permettendo così di costruire wiki semantiche e *vocabolari*. Le comunità di utenti possono essere aperte (e i contenuti, siano essi wiki tradizionali o semantiche, modificabili da tutti) oppure chiuse, accettando solo i membri registrati.
In figura 2.6 si può osservare un esempio di utilizzo dell'applicazione: un esempio di vocabolario su cibi e vini sviluppato dalla comunità *VoCamp*. Un'interessante strumento permette di visualizzare una rapprsentazione grafica delle classi e delle relazioni tra di esse, mediante l'utilizzo di applet Java.

Alle applicazioni sopraccitate va aggiunto **NeOn Toolkit** (figura 2.3), un'applicazione che condivide con Protégé e TopBraid Composer molte caratteristiche, soprattutto funzionali, che lo qualificano in primo luogo come un editor stand-alone, che nasce tuttavia con lo scopo di supportare la cooperazione tra più soggetti nella creazione e nella modifica di ontologie condivise[8]. NeOn rappresenta un progetto vasto e complesso e un software in veloce evoluzione.

2.3.3 Gli editor grafici

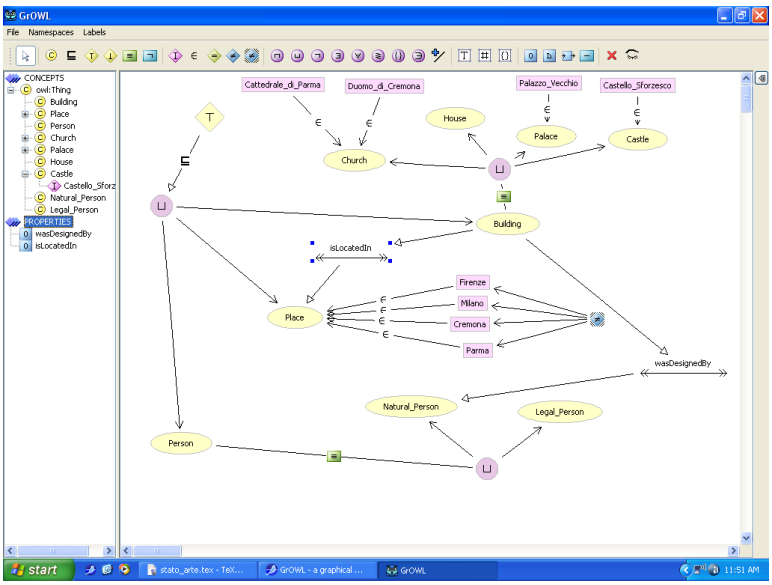


Figura 2.7: *Schermata di GrOWL*

Una menzione particolare va riservata alle applicazioni (tipicamente stand-alone) che permettono di realizzare ontologie utilizzando esclusivamente un approccio grafico. Sebbene esistano convenzioni riguardo la notazione da adottare per editor di questo tipo, non esistono ad oggi notazioni standard, pertanto ogni

applicazione che consenta di creare graficamente un'ontologia ne definisce una propria.

GrOWL (http://growl.novasemantics.it) è un tool open-source sviluppato dall'ing. Mario Arrigoni Neri del Politecnico di Milano. Consente di creare e modificare ontologie scritte in OWL.

Scritto in JAVA, si basa sulle librerie di Jena, e su un motore proprio per la disposizione e il rendering dei nodi e degli archi del grafo. In figura 2.7 è rappresentato uno scenario tipico dell'utilizzo di GrOWL. L'ontologia raffigurata è di piccole dimensioni, pertanto il grafo mantiene un aspetto chiaro e leggibile. Per ontologie di dimensioni molto maggiori, tuttavia, il grafo può assumere un aspetto meno ordinato e al limite diventare incomprensibile. Per questa ragione è presente una funzione per nascondere, senza eliminarli dall'ontologia, i nodi e gli archi ritenuti poco interessanti, anche se agendo in questo modo si perde la vista d'insieme.

GrOWL si dimostra quindi inadatto a rappresentare modelli di grandi dimensioni, per i quali è necessario utilizzare altri tipi di editor: ciò nonostante, si presta ad utilizzi meno onerosi dal punto di vista della complessità del modello da visualizzare, come ad esempio in ambito didattico.

2.3.4 Jena e OWL API

Jena (http://jena.sourceforge.net) è un framework open-source scritto in Java per applicazioni che fanno uso di modelli in OWL (e RDF). Consiste in una serie di *Application Programming Interface* (API) che implementano le operazioni fondamentali di input e output su file (curando in modo particolare la serializzazione), a cui si aggiungono altre librerie che forniscono l'interfaccia per il linguaggio di query semantico, *SPARQL Protocol And Query Language* (SPARQL).

Buona parte delle applicazioni esaminate nei paragrafi precedenti, come accennato, include e utilizza le librerie di Jena. Questa scelta permette ai programmatori, in fase di sviluppo dell'applicazione, di disinteressarsi del della persistenza del modello e della serializzazione, focalizzando invece l'attenzione su altri aspetti del software, come l'interfaccia utente.

OWL API (http://owlapi.sourceforge.net), noto in precedenza come *WonderWeb API*, è un altro insieme di librerie, anch'esso scritto in Java. Più recente rispetto a Jena, è stato sviluppato per rispondere alle stesse esigenze, ma per varie ragioni non ne costituisce in senso stretto un'alternativa. Allo stato attuale, per esempio, OWL API supporta OWL 2, al contrario di Jena, che però è in grado di gestire una maggior varietà di formati.
OWL API è rilasciato nei termini della *Lesser General Public License* (LGPL), che è una licenza open-source.

2.4 Protégé

Protégé (http://protege.stanford.edu) è un'applicazione sviluppata dall'Università di Stanford (Palo Alto, California, USA) in collaborazione con l'Università di Manchester (Regno Unito).

Figura 2.8: *Il logo di Protégé*

Gli sviluppatori descrivono Protégé come 'una piattaforma gratuita e open-source, che supporta una crescente comunità di utenti con una suite di strumenti per realizzare domain model e applicazioni basate sulla conoscenza tramite ontologie'[12]. Protégé nasce quindi come un supporto per l'ingegneria della conoscenza, fornendo strumenti per tutte le attività connesse alla realizzazione di un'ontologia (creazione di modelli, inserimento di dati, modifica e visualizzazione)[11].

Protégé è scritto interamente in Java e strutturato a *plug-in*, in modo tale da consentire lo sviluppo di estensioni, con l'aggiunta di nuove funzionalità, senza alterare il codice sorgente originario. Di per sé, gran parte delle funzionalità predefinite di Protégé sono implementate da plug-in, che vanno a innestarsi sul modulo centrale dell'applicazione, che i progettisti chiamano *Protégé core*. Lo

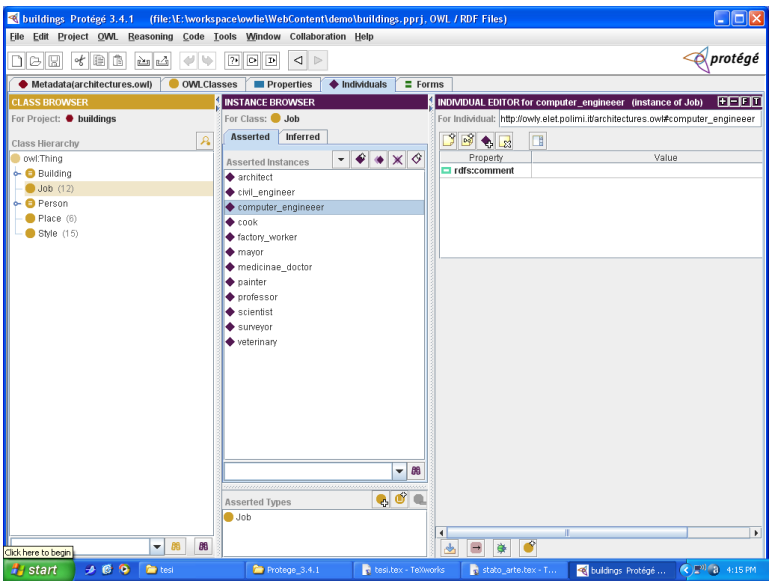


Figura 2.9: *Schermata di Protégé 3.4.1*

sviluppo di Protégé è tuttora in corso e procede su due fronti distinti. La *release* più matura, Protégé 3, subisce di tanto in tanto alcuni ritocchi (in gran parte miglioramenti, ottimizzazioni e correzioni di bachi, più raramente aggiunte di nuove funzionalità), che portano al rilascio di nuove versioni (la più recente al momento della stesura di questo testo è la 3.4.4 rilasciata nel mese di marzo 2010).

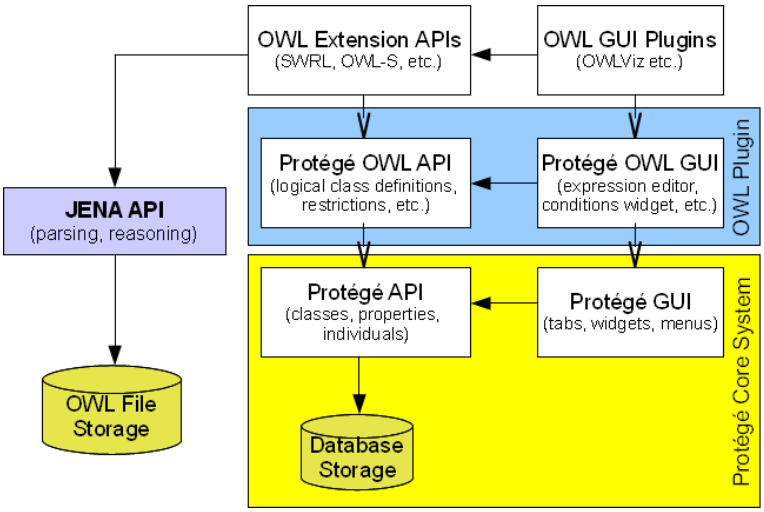


Figura 2.10: *Architettura a plug-in di Protégé 3*

L'altro ramo del progetto ha preso il nome di Protégé 4. Ben lungi dall'essere un semplice aggiornamento della versione precedente, si presenta invece come un software completamente nuovo, sviluppato per rispondere alle stesse esigenze che hanno motivato lo sviluppo del predecessore, con l'obiettivo di supportare nuovi standard. Tuttavia, come verrà discusso nel paragrafo 2.4.5, allo stato attuale Protégé 4 - pur avendo raggiunto uno stadio di maturità sufficiente dal momento del rilascio della prima versione *definita* stabile dai suoi programmatori, la 4.0.2 - presenta ancora diverse lacune e il suo utilizzo come supporto deve essere frutto di approfondite valutazioni che lo facciano preferire rispetto al solido Protégé 3.

Il team di Protégé incoraggia chiunque abbia sviluppato o voglia estendere le funzionalità dell'applicazione, tanto per la versione 3, quanto per la versione 4, consentendo al progettista di accedere al database che raccoglie tutti i plug-in sinora rilasciati e, al termine dello sviluppo del proprio plug-in, consentendogli di pubblicare il proprio lavoro inserendolo nel database. Sul web, all'indirizzo `http://protegewiki.stanford.edu/wiki/Protege_Plugin_Library`, è possibile visionare l'elenco dei plug-in attualmente pubblicati, organizzato per categorie, e inserirne di nuovi.

Come già accennato, Protégé è rilasciato sotto licenza *Mozilla Public License* (MPL), che è una licenza libera. Tale licenza è propagata anche ai plug-in sviluppati direttamente dal team di Protégé, tra cui Protégé-OWL, ma ciò non toglie che possano esistere anche plug-in rilasciati sotto licenze diverse, non necessariamente open-source, così come i prodotti che pur non costituendo plug-in di Protégé, ne utilizzano le API.

2.4.1 Protégé-OWL

Protégé 3, si è detto, è strutturato in maniera modulare.

Il modulo base, chiamato appunto *Protégé core*, fornisce le funzionalità principali del software, e implementa direttamente le funzionalità di creazione, visualizzazione e manipolazione delle ontologie basate su *frames*, da cui la denominazione di *Protégé-Frames*.

Le ontologie basate su *frames* non utilizzano OWL, sebbene possano servirsi di RDF come formato di persistenza dei dati[9].

Il supporto per OWL è stato incluso in un plug-in, installabile separatamente sebbene normalmente incluso nel *bundle* disponibile sul web. Il plug-in di *Protégé-OWL* interagisce con il modulo base di Protégé, estendendone l'interfaccia grafica e aggiungendo al modello basato sui *frames* i costrutti specifici di OWL (ad esempio, i concetti arbitrari, ottenute combinando concetti atomici mediante connettori logici, come già trattato nel paragrafo 2.1.3).

In figura 2.10 è schematizzata la struttura di Protégé-OWL. Il modulo base di Protégé fornisce l'interfaccia grafica (GUI) principale e le API di modellizzazione di Protégé-Frames. Il plug-in di Protégé-OWL contiene un set di API che sfrutta le interfacce riutilizzabili di Protégé-Frames (ad esempio classi, istanze o individui, proprietà) implementando gli elementi mancanti. Allo stesso modo, come mostrato in figura 2.9, Protégé-OWL non definisce una propria interfaccia grafica, utilizzando la stessa GUI del modulo principale alla quale aggiunge alcuni elementi, sotto forma di *tab* o di *widget* (ad esempio l'editor di classi anonime).

L'architettura di Protégé è assai flessibile e permette di realizzare plug-in che si basano su altri plug-in. In questo modo è possibile estendere ulteriormente Protégé-OWL, costruendo plug-in che implementino funzionalità aggiuntive. Un esempio è dato dal plug-in *OWLViz*, presente per impostazione predefinita dopo l'installazione di Protégé-OWL, che consente la visualizzazione della gerarchia di classi sfruttando il motore grafico *Graphviz* (vedasi paragrafo A.1 delle appendici).

Protégé-OWL sfrutta le API di Jena, sia come parsificatore di file `.owl`, sia come strumento di reasoning e di supporto per le query in SPARQL. In altre parole, tramite il plug-in di Protégé-OWL, Protégé diviene un front-end per le librerie di Jena.

2.4.2 Protégé client-server

Un'interessante funzionalità offerta da Protégé 3 è la possibilità di interagire collaborativamente con un'ontologia condivisa, tramite il server di Protégé[14].

[9] Un'ontologia *frame-based*, in accordo con l'*Open Knowledge-Base Connectivity (protocol)* (OKBC), è in sostanza una gerarchia di classi, che rappresenta i concetti rilevanti di un certo dominio. Alle classi è assegnato un insieme di *slot*, che rappresentano le relazioni tra gli elementi di ciascuna classe. Ogni classe può contenere, come nelle ontologie basate su OWL, un certo numero di istanze.

Tale funzionalità è messa a disposizione dal modulo principale di Protégé, senza la necessità di installare plug-in aggiuntivi (salvo ovviamente utilizzare Protégé-OWL se richiesto).

L'interazione tra client e server si basa sulla *Remote Method Invocation* (RMI) di Java, ed è gestita in modo centralizzato. Una speciale ontologia chiamata *metaproject* (metaprogetto) formalizza la politica di controllo degli accessi, registrando gli utenti che possono accedere al sistema, gli oggetti - tipicamente ontologie - da rendere accessibili (ossia condividere), e infine le politiche di accesso associate a ciascun oggetto condiviso (tipicamente formulate a livello di gruppo di utenti). Un esempio di impostazione del metaprogetto è visibile nelle figure A.2 e A.3.

Lato server sono gestite tutte le operazioni per l'accesso concorrente alle risorse condivise. Viene data possibilità a più utenti di caricare il medesimo progetto[10] e di modificarlo simultaneamente.

Le applicazioni esistenti del server di Protégé sono essenzialmente due: *Collaborative Protégé* e *Webprotege*, ma è anche possibile sviluppare applicazioni indipendenti che dialoghino con il server di Protégé incorporandone una parte della logica lato client, tramite il suo ricco insieme di API.

2.4.3 Collaborative Protégé

Collaborative Protégé è un'estensione di Protégé per la modifica collaborativa di ontologie condivise, permettendo agli utenti di *annotare* i cambiamenti fatti (ovvero, scrivere commenti in linguaggio naturale al fine di aiutare gli altri membri del gruppo a comprendere le modifiche effettuate).

L'interazione dell'utente con un'ontologia condivisa può avvenire in due modi:

- in modalità multiutente (*multi-user mode* o *concurrent mode*) l'ontologia condivisa risiede in una postazione remota in cui è attivo il server di Protégé (come descritto nel paragrafo 2.4.2). Gli utenti lato client possono effettuare modifiche simultanee e visibili in tempo reale agli altri utenti collegati. In figura 2.11 è riportato un esempio di utilizzo di Collaborative Protégé in modalità multiutente.

- in modalità *stand-alone* (o *consecutive mode*) l'ontologia risiede, ad , in una directory condivisa in rete e più utenti vi accedono in successione (ma non simultaneamente), annotandovi le modifiche fatte. In questa modalità non è necessario configurare il server.

Una versione dimostrativa di Collaborative Protégé è disponibile sul web all'indirizzo `http://smi-protege.stanford.edu/collab-protege`.

[10] in Protégé 3, un'ontologia costituisce un progetto

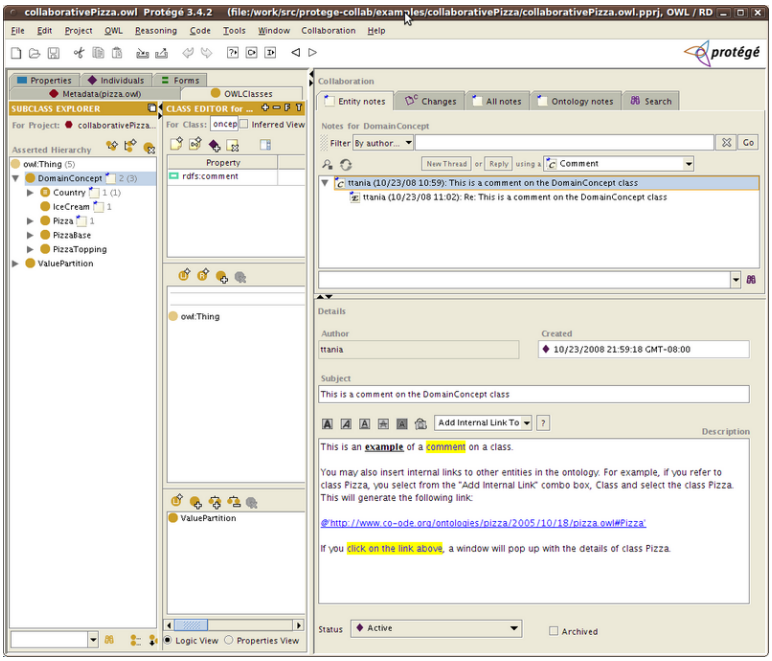


Figura 2.11: *Esempio di utilizzo di Collaborative Protégé (tratto da* `http://protegewiki.stanford.edu/wiki/Collaborative_Protege`*)*

Tanto in modalità multiutente, quanto in modalità stand-alone, al progetto condiviso è necessario associare un'ontologia in formato *Changes and Annotation Ontology* (ChAO), memorizzata in un file RDF oppure in un database, che consente di tenere traccia delle modifiche e delle annotazioni degli utenti. In mancanza di un'*annotation project* collegato all'ontologia condivisa, Collaborative Protégé perde le sue funzionalità.

2.4.4 WebProtege

WebProtege (il cui nome, curiosamente, non è scritto con l'accento acuto sulla e, al contrario dell'applicazione madre) rappresenta il tentativo da parte del team di Protégé di fornire un'interfaccia web nativa a Collaborative Protégé, utilizzando un approccio leggero e mantenendo tutta la logica applicativa a lato server (ovvero senza utilizzare *applet* o *Java web start*).

WebProtege implementa un'architettura client-server a tre livelli, come mostrato in figura 2.13. Il livello più interno è costituito dal server di Protégé (descritto al paragrafo 2.4.2), che implementa la maggior parte della logica di controllo degli accessi e concorrenza, mentre il livello più esterno è costituito semplicemente dal browser web dell'utente, che non implementa logica applicativa ma si occupa unicamente della presentazione.

Gran parte della logica di *business* fa parte del livello intermedio. WebProtege utilizza le API di Protégé 3 per accedere all'ontologia ed interagire con

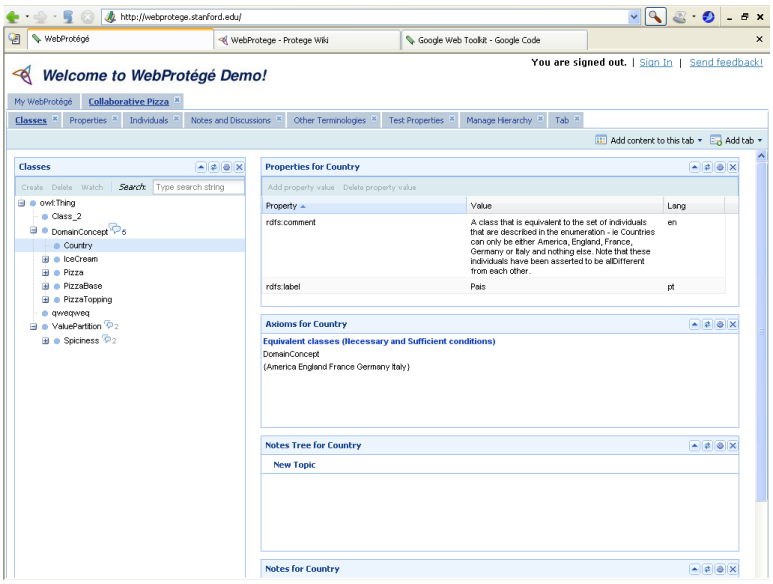


Figura 2.12: *Interfaccia di WebProtege*

essa, dialogando tramite RMI con il server, mentre impiega le librerie grafiche del *Google Web Toolkit* (GWT) per realizzare l'interfaccia grafica.

I componenti GWT[11] consentono di ottenere – utilizzando un semplice web – un'interfaccia notevolmente articolata, che si avvicina molto in termini di qualità, funzionalità e aspetto a quella ottenibile utilizzando le librerie grafiche di Java (SWING o AWT) in un'applicazione standard o un'applet, ma con l'indubbio vantaggio di non occupare risorse eccessive lato client e, soprattutto, senza la necessità da parte dell'utente di installare add-on sul proprio browser[12].

WebProtege si presenta all'utente con un'interfaccia paragonabile a quella di Protégé, anche se molto più scarna, come si vede in figura 2.12. Effettuato il login (anche se esiste la possibilità di effettuare connessioni dimostrative, senza immettere nome utente e password) viene mostrata all'utente la lista delle ontologie condivise sulle quali ha l'autorizzazione di agire (tali autorizzazioni sono contenute nel metaprogetto lato server). È possibile interagire con più ontologie contemporaneamente, a differenza di Collaborative Protégé dove è consentito mantenere il controllo di un unico progetto per volta.

Sebbene WebProtege rappresenti un'innovazione nel panorama del software di supporto al semantic web (la prima versione messa a disposizione del pubblico, la 0.5 *alpha*, risale ad ottobre 2008), pare non sia una priorità nel team di sviluppo di Protégé, visto anche il ritmo assai lento di rilascio di nuove versioni. Al momento dell'inizio dello studio di fattibilità di OWLIE, cioè diversi

[11] http://code.google.com/intl/it-IT/webtoolkit

[12] a patto che il browser supporti i *javascript*, i quali ad ogni modo richiedono elaborazioni lato client che utilizzano risorse di sistema, in misura però trascurabile negli elaboratori di ultima generazione

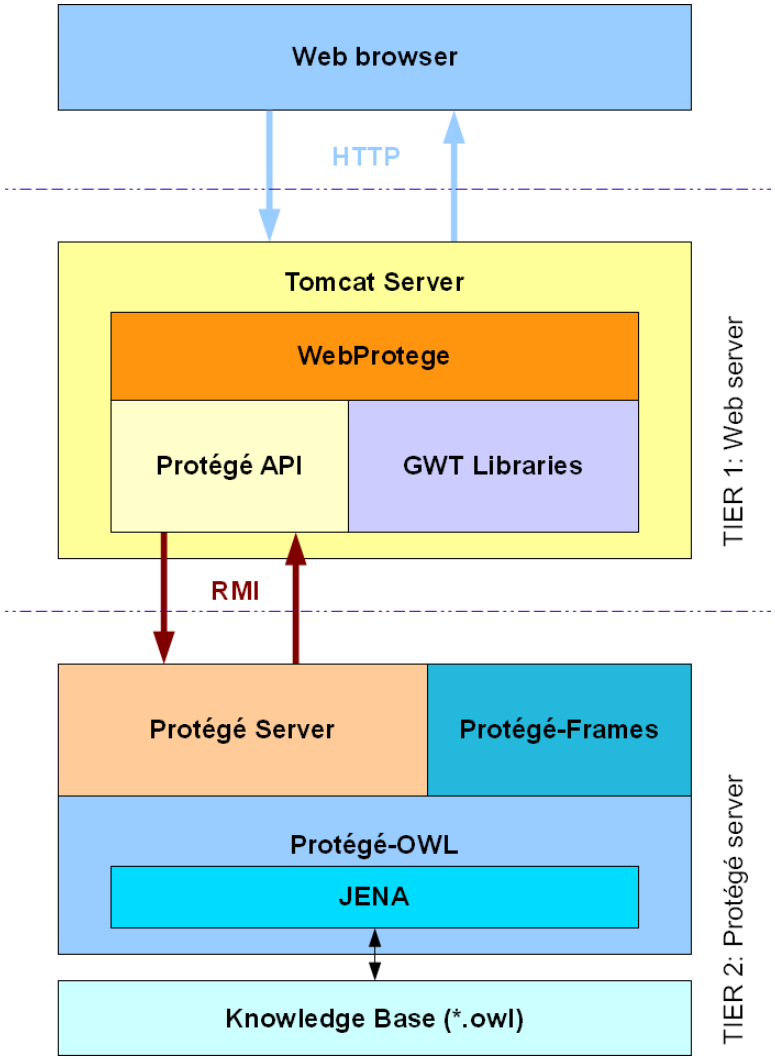


Figura 2.13: *Schema a tre livelli di WebProtege*

mesi dopo il rilascio della prima versione di WebProtege, risultavano effettuate soltanto un centinaio di compilazioni dei sorgenti[13], con pochi cambiamenti tra un rilascio e il successivo[14], mentre un software di tali dimensioni, in piena fase di sviluppo dovrebbe essere manipolato in un numero molto maggiore di volte. Inoltre, dal mese di agosto 2009 non ne vengono rilasciate nuove versioni.

Una possibile spiegazione di questo rallentamento (ma si potrebbe quasi affermare che lo sviluppo sia stato sospeso) si può ritrovare nella complementare, veloce evoluzione di Protégé 4, che può da un lato aver assorbito le attenzioni del team di sviluppo, ma non solo: è lecito pensare che i progettisti siano in attesa che le nuove librerie di Protégé 4 siano sufficientemente mature per poter sostituire quelle mature di Protégé 3, non compatibili con il nuovo standard OWL 2, su cui WebProtege si basa. Inoltre Protégé 4 non implementa le funzionalità di server, e dalla *roadmap* di WebProtege [15] si evince chiaramente che tale migrazione è in programma.

Allo stato attuale (versione 0.5alpha build 200, datata 14 agosto 2009) l'applicazione supporta le operazioni fondamentali di modifica di un'ontologia e le funzionalità collaborative (annotazioni e discussioni). A differenza dell'applicazione standard di Protégé, non è stato implementato il supporto per un *reasoner* ragion per cui l'utente deve fare a meno dell'inferenza.

2.4.5 Protégé 4

Protégé 4 rappresenta una svolta, sotto molti aspetti, nel mondo del software per l'ingegneria della conoscenza. Di recente sviluppo, ne è stata rilasciata una versione stabile (ossia pronta per un utilizzo professionale) da pochi mesi. La versione 4.0.2 infatti ha superato lo stadio *beta* il 3 dicembre 2009, mentre è ad oggi in corso lo sviluppo della versione successiva, la 4.1, che si trova allo stadio *alpha*.

Le differenze rispetto a Protégé 3 risultano molto più evidenti e numerose delle analogie. In prima battuta si può constatare una notevole semplificazione dell'interfaccia grafica, pur mantenendo un layout a linguette, eredità della versione precedente. Uno *screenshot* di Protégé 4.1 *alpha* è mostrato in figura 2.14.

Dal punto di vista dell'architettura permane il paradigma a plug-in come nella versione precedente, il che rende Protégé 4 un'applicazione estendibile arbitrariamente tramite moduli caricati all'occorrenza, esattamente come il predecessore. Vi è stato tuttavia implementato un sistema di aggiornamento automatico dei componenti, che viene effettuato all'avvio o su richiesta dell'utente: il sistema interroga – se è disponibile una connessione ad internet – il server

[13] dato che si evince dalla *wiki* di WebProtege, di cui si riporta l'indirizzo web statico aggiornato a maggio 2010: http://protegewiki.stanford.edu/index.php?title=WebProtege&oldid=7174

[14] http://protegewiki.stanford.edu/index.php?title=WebProtegeReleaseNotes&oldid=6451

[15] http://protegewiki.stanford.edu/index.php?title=WebProtegeRoadMap&oldid=6419

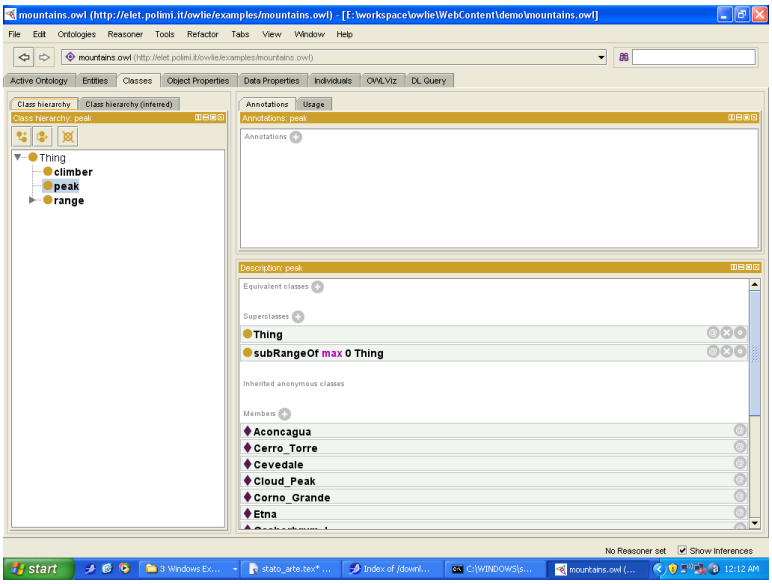


Figura 2.14: *Schermata di Protégé 4.1 alpha*

degli aggiornamenti, e propone all'utente la lista dei componenti che è possibile aggiornare. L'utente può scegliere se aggiornarli tutti o solamente alcuni.

Le differenze più pesanti tra Protégé 3 e Protégé 4 appaiono a livello più basso.[13] Con riferimento all'architettura di Protégé 3 mostrata in figura 2.10, nella nuova versione scompare la distinzione tra Protégé-core e Protégé-OWL, in quanto è direttamente il modulo principale a fornire il supporto per OWL, tramite un set di API completamente diverso, le *OWL API* sviluppate dall'Università di Manchester[16].

Questi cambiamenti così radicali nella struttura del software, che si presume quindi essere stato completamente riscritto, coinvolgono anche le funzionalità offerte dall'applicazione, ora più che mai orientata al nuovo standard OWL 2, completamente supportato a partire dalla versione 4.1 (la 4.0.2 lo supporta solo parzialmente). Protégé 4 non accetta altri formati se non OWL stesso, essendo stato interrotto il supporto per RDF puro: le ontologie basate sui frame, per questa ragione, permangono una prerogativa di Protégé 3.

Evidenti ragioni di sintesi non permettono di addentrarsi in ulteriori comparazioni tra le due *major releases* di Protégé, sebbene vi siano molti altri elementi di discontinuità tra Protégé 3 e Protégé 4[13], tanto dal punto di vista strutturale, quanto da quello funzionale. Sebbene i progettisti di Protégé incoraggino la migrazione verso Protégé 4, se non vi sono esigenze di utilizzare i frame, Protégé 4 permane un software ancora troppo giovane per poterlo utilizzare come base per una nuova applicazione (fatte salve estensioni sotto forma di plug-in): alcune funzioni implementate da Protégé 3, come la possibilità di in-

[16] http://owlapi.sourceforge.net

teragire con un'ontologia da remoto tramite un'interfaccia client/server, devono ancora essere implementate in Protégé 4, e attualmente non è dato sapere se e quando ciò avverrà.

2.5 Conclusioni

Dall'analisi del parco software in essere, emerge un panorama variegato. Tra tutte le applicazioni studiate, in base alle valutazioni fatte, si può senz'altro affermare che nessuna di esse risponde in modo adeguato a tutti i requisiti alla base del progetto in esame.

Protégé è senz'altro un caso particolare su cui va fatta una riflessione. Rappresenta infatti un software particolarmente interessante sotto molte prospettive, in primo luogo la vasta gamma di funzionalità offerte, e secondariamente la sua adattabilità a scenari d'uso differenti.

L'eccezionale semplicità di estensione, tramite la sua struttura a plug-in, sarebbe un punto a favore dello sfruttamento di Protégé (in questo caso, di Protégé 4) per la realizzazione di una nuova plug-in, il cui ciclo di sviluppo sarebbe assai più breve e snello rispetto a un'applicazione da costruire ex-novo. L'accessibilità mediante browser web, uno degli obiettivi cardine del progetto, sarebbe possibile se si ammettesse che Protégé con l'aggiunta del nuovo plug-in possa essere avviato direttamente dal web, utilizzando tecnologie ben note come JNLP o, addirittura, sotto forma di *applet*.

In questo modo, tuttavia, si perderebbe un altro requisito base, la leggerezza del software lato utente (JNLP e Applet prevedono il download del codice dell'applicazione, che verrebbe completamente eseguita lato client come qualsiasi applicazione installata sulla macchina dell'utente, sebbene per le applet ciò avvenga in modo "mascherato").

A seguito delle suddette considerazioni, l'unico candidato a rispondere in modo accettabile ai requisiti di base avrebbe potuto essere WebProtege, se fosse giunto a un livello di sviluppo sufficiente.

L'ipotesi di creare un derivato di WebProtege, in pratica proseguendone lo sviluppo in modo indipendente (quello che in linguaggio informatico si definisce *fork*) ed estendendone le funzionalità, è stata presa in considerazione ma alcuni fattori ne hanno comportato l'abbandono già all'inizio dello studio di fattibilità. Tali fattori comprendono l'insufficiente documentazione fornita assieme ai sorgenti di WebProtege e l'inadeguatezza di alcune scelte tecniche (in particolare, la tecnologia utilizzata per l'interfaccia utente).

Per i motivi evidenziati, è stato ritenuto opportuno sviluppare una nuova applicazione, eventualmente utilizzando componenti e librerie già in uso altrove, ma senza estendere alcun software preesistente.

Capitolo 3

Progetto dell'applicazione

Figura 3.1: *Icona di OWLIE*

Questo capitolo analizza le fasi centrali del processo di sviluppo dell'applicazione così come definito nel capitolo 1.

In primo luogo saranno esaminate le fasi di analisi dei requisiti e di studio dei casi d'uso, da cui emergono i tratti essenziali dell'applicazione, il cui paradigma architetturale e le scelte strutturali che la riguardano sono invece oggetto di una successiva fase di *design* operata sulla base delle precedenti conclusioni.

3.1 Analisi dei requisiti

Riprendendo quanto emerso nel paragrafo 1.1, l'idea di OWLIE nasce dalla necessità di realizzare un editor per ontologie, sotto forma di applicazione web, leggero, di facile apprendimento e orientato all'utilizzo collaborativo. Tali sono, in estrema sintesi, gli obiettivi che questo progetto si propone di raggiungere.

Nella pratica, soprattutto dal punto di vista del progettista, è necessaria una riflessione accurata su chi siano gli utenti che si prevede possano utilizzare l'applicazione (i cosiddetti *attori*, nel gergo dei processi software), su come debbano interagirvi e quali siano le operazioni che conviene supportare (i cosiddetti *scenari*).

3.1.1 Identificazione degli attori

Posto che realizzare un'applicazione che sia basata sul web rappresenta una linea guida non più negoziabile, l'individuazione delle varie tipologie di utenza è un compito piuttosto semplice.

Nei paragrafi introduttivi, come pure nel capitolo 2, è stato osservato come diverse applicazioni esistenti, almeno in origine, erano state concepite per un'utenza che si potrebbe definire "di nicchia", costituita da esperti già in possesso di

conoscenze riguardo a OWL e alle ontologie in generale, a un livello ben al di sopra delle basi.

Se si pensasse a OWLIE come un'estensione di un'applicazione , o un software realizzato ex-novo ma che comunque che condivida con un'altro programma la totalità degli scenari d'uso, per raggiungere l'obiettivo della facilità di apprendimento sarebbe necessario aggiungere, alla figura dell'utente "esperto", anche una figura di utente "principiante"[1]. Tale figura rappresenterebbe un utente che non ha le dirette competenze nell'ambito del web semantico necessarie per interagire con un'applicazione che abbia caratteristiche di complessità elevate, riscontrabili ad esempio in Protégé.

Il dualismo tra utente esperto e utente principiante porterebbe, necessariamente, a una diversificazione dei casi d'uso: l'utente esperto potrebbe voler eseguire operazioni complesse, utilizzando un'interfaccia ricca di elementi e dotata di funzioni a volte "rischiose", riservate appunto a persone che sappiano ciò che fanno. Per contro, l'utente meno esperto, ben consapevole di questo, si accontenterebbe di eseguire le operazioni di base, in perfetta sicurezza (nei limiti del buonsenso) e, quando serve, assistito dal software nelle sue azioni.

I software che prevedono, per almeno una parte delle funzionalità, interfacce separate per grado di competenza dell'utente sono molteplici. Si pensi, ad esempio, ai programmi di installazione di altri programmi, specialmente in Windows: solitamente prevedono un'opzione (che viene consigliata ai soli esperti) per la selezione e l'installazione separata dei vari componenti dell'applicazione.

Una differenziazione dell'interfaccia utente sulla base dell'esperienza sarebbe stata possibile anche in OWLIE, prevedendo, ad esempio, che l'utente potesse scegliere se visualizzare o meno certi elementi dell'ontologia, oppure un'interfaccia molto più articolata e complessa destinata a un utente che si dichiara "esperto" in fase di login.

Ciò non è stato ritenuto opportuno preferendo di considerare unicamente gli utenti non esperti, e modellando gli scenari d'uso dell'applicazione sulla base di questa assunzione, scenari che coinvolgono due soli attori, ossia l'**utente** (generico) e l'**amministratore**.

L'utente di OWLIE può:

1. accedere al sistema mediante l'inserimento di un *username* e di una *password*;
2. una volta ottenuto l'accesso al sistema, visualizzare l'elenco delle ontologie disponibili (ed eventualmente, crearne una nuova);
3. tra le ontologie disponibili (condivise), può sceglierne una e aprirla;

[1] altrimenti definito con l'aggettivo inglese *dummy*, vale a dire una persona che si cimenta con applicazioni informatiche per la prima volta. Tale termine, per nulla dispregiativo nell'accezione attuale, può essere applicato sia ai novizi totali in campo informatico, sia a persone già esperte in un dato ambito che utilizzano del software che non conoscono

4. visualizzare il contenuto dell'ontologia e modificarne gli elementi;
5. se è implementato un sistema di cooperazione con altri utenti, interagire con loro.

L'amministratore può essere anch'egli un utente, ed essere così fornito di credenziali di accesso; tuttavia il suo ruolo gli conferisce altre prerogative:

1. a sistema installato, configura il server in modo tale per cui gli utenti possano accedervi;
2. può essere contattato da nuovi utenti al fine di consegnare loro le credenziali di accesso al sistema (qualora non venga implementato un sistema di sign-up, ossia di registrazione, allorché sia permessa l'auto-registrazione);
3. controlla e assicura il corretto funzionamento del sistema.

L'amministratore agisce quindi sul sistema, sia accedendovi, sia intervenendovi dall'esterno (all'atto della configurazione iniziale, per esempio).

3.1.2 Identificazione degli scenari

Sulla base delle ipotesi fatte sino a questo punto, e con particolare riguardo alle osservazioni emerse al momento dell'identificazione degli *attori* del sistema, il passo successivo nel processo di analisi dei requisiti punta a descrivere i possibili scenari di utilizzo dell'applicazione.

Dato che l'individuazione degli scenari ha lo scopo di dettare al progettista quali caratteristiche dovrà avere il software, una prospezione accurata evita, per quanto possibile, di dover intervenire a implementazione già iniziata per aggiungere altre caratteristiche o eliminarne alcune ritenute più importanti di quanto in realtà non siano.

Nel caso di un editor di ontologie con capacità collaborative si delineano quattro scenari principali:

- creazione di un'ontologia (o modifica di un'ontologia preesistente) secondo un approccio *stand-alone* (non collaborativo: gli interventi sono prettamente individuali anche se l'ontologia risiede in una locazione remota). L'ambiente di lavoro deve essere orientato, in questi casi, alla completezza: deve essere pertanto possibile, da parte dell'utente, poter utilizzare tutti i costrutti che OWL-DL mette a disposizione per realizzare un modello.
- esame di un'ontologia. L'utente esamina il modello, cercando di in prima battuta una visione d'insieme anziché dettagliata. Devono essere quindi presenti funzionalità di visualizzazione, che mettano in risalto, preferibilmente in forma grafica piuttosto che testuale, le relazioni presenti tra le risorse che compongono il modello.
  Un'estensione di questo scenario prevede la possibilità di effettuare modifiche a partire da "viste globali" dell'ontologia.

- costruzione o modifica cooperativa di un'ontologia. Più utenti si connettono nel medesimo istante di tempo e visionano contemporaneamente la stessa ontologia. Tali utenti possono essere membri dello stesso gruppo di lavoro, oppure persone estranee. Lavorando in collaborazione, essi devono per forza di cose agire su un'ontologia condivisa: siccome ogni utente deve agire con una vista *coerente* e *consistente* del modello, le modifiche apportate da un utente debbono essere propagate, notificate e visualizzate da ogni altro utente connesso nel medesimo istante di tempo.

- supervisione del sistema. La figura dell'amministratore è deputata all'installazione del sistema e alla gestione delle eccezioni, nei modi e nei tempi stabiliti in una successiva fase di design architetturale.

Le situazioni identificate sono all'apparenza molto distanti tra loro, e a un primo esame si mostrano quasi in antitesi, specialmente il primo e il secondo tra gli scenari descritti.

Il primo scenario si adatta bene ad un'applicazione *stand-alone*, di cui gli esempi descritti nel paragrafo 2.3.1 rappresentano applicazioni desktop, non basate sul web. Occorre disporre di un'interfaccia che consenta di mantenere la completezza del linguaggio OWL, e tale caratteristica presuppone un livello di dettaglio piuttosto elevato.

Il secondo scenario beneficerebbe di un'interfaccia dotata di un numero maggiore di elementi e costrutti grafici al posto di elementi testuali. Per garantire una visione d'insieme dell'ontologia, si può fare uso di grafi dinamici, che consentano di espandere o collassare elementi a seconda del livello di dettaglio desiderato. La difficoltà maggiore in questo caso riguarda la capacità che un'interfaccia web abbia di poter rappresentare grafi di una certa complessità, per quanto articolata e ricca sia.

Il terzo scenario è applicabile a un'applicazione web che nasca sulla base dei cosiddetti "social network" o comunque secondo un approccio *web 2.0*, in cui sia garantita la possibilità di accesso simultaneo e vengano implementati meccanismi, oltre che di controllo della concorrenza, anche di supporto allo scambio di messaggi.

3.2 Linee guida per lo sviluppo di OWLIE

L'elicitazione degli scenari evidenzia la necessità di un'applicazione che risponda sia ad esigenze di completezza e dettaglio, sia alla facilità di analisi di un modello, con la possibilità di interagire con altri utenti.

Ciò che si intende ottenere mediante lo sviluppo di OWLIE è pertanto un editor che non preveda un'unica "vista" dell'ontologia, bensì risulti diviso in varie "parti" ognuna delle quali mostra un solo segmento di interesse della base di conoscenza (ad esempio, la gerarchia dei concetti piuttosto che l'elenco degli

individui). Ogni "vista" potrà essere associata a una serie di strumenti di modifica delle risorse, strutturati in modo da poter rappresentare la totalità (o quasi) dei costrutti di OWL-DL.

In questo modo si può mantenere facilmente la compatibilità con ontologie scritte mediante un altro software.

Alcune "viste" possono essere costituite da grafici che coprano la parte terminologica dell'ontologia (dunque la T-BOX e la R-BOX) e da altri grafici che invece mostrino le relazioni (ovvero le asserzioni) che intercorrono tra gli individui, facenti parte della A-BOX.

La sintassi e le notazioni devono essere semplici e immediate, al fine di non aggiungere elementi che potrebbero confondere l'utente (il quale può non avere, come già ampiamente discusso, alcuna esperienza riguardo al web semantico).

Infine deve essere implementato un sistema per la gestione degli accessi condivisi che abbia come finalità anche quella di strutturare e regolare lo scambio di messaggi tra gli utenti, il controllo della coerenza sui dati condivisi e la propagazione delle notifiche, vale a dire un *sottosistema collaborativo*.

3.3 Design dell'applicazione

La fase di design dell'applicazione prevede tre fasi susseguenti.

La prima fase è la scelta di un modello architetturale a cui fare riferimento per la realizzazione del software, ovvero, trattandosi di un'applicazione basata sul web, la scelta della tipologia di architettura client-server da adottare.

La seconda fase prevede la scelta del *framework* applicativo per realizzare l'interfaccia utente, aspetto imprescindibile dello sviluppo dell'applicazione (e che condiziona le scelte progettuali successive e lo stile di codifica in fase di implementazione).

Una volta fatte le scelte di base, la terza fase di design si occupa di definire la struttura dell'applicazione, ossia la sua suddivisione in moduli e il modo in cui tali moduli interagiscono tra loro.

3.3.1 Architettura client-server

Gli indirizzi progettuali impongono un'applicazione leggera, ossia accessibile via browser senza la necessità di installare ed eseguire logica applicativa da parte dell'utente, e che, al tempo stesso, implementi un efficace sistema di controllo degli accessi e di regolamentazione della concorrenza, al fine di consentire a più utenti di collaborare a un medesimo lavoro.

Appare evidente come l'architettura più consona per un'applicazione che risponda alle caratteristiche sopraccitate segua un approccio distribuito, secondo un *design-pattern* architetturale di tipo client-server a più livelli.

OWLIE implementa un modello client-server a tre livelli, come schematizzato in figura 3.2.

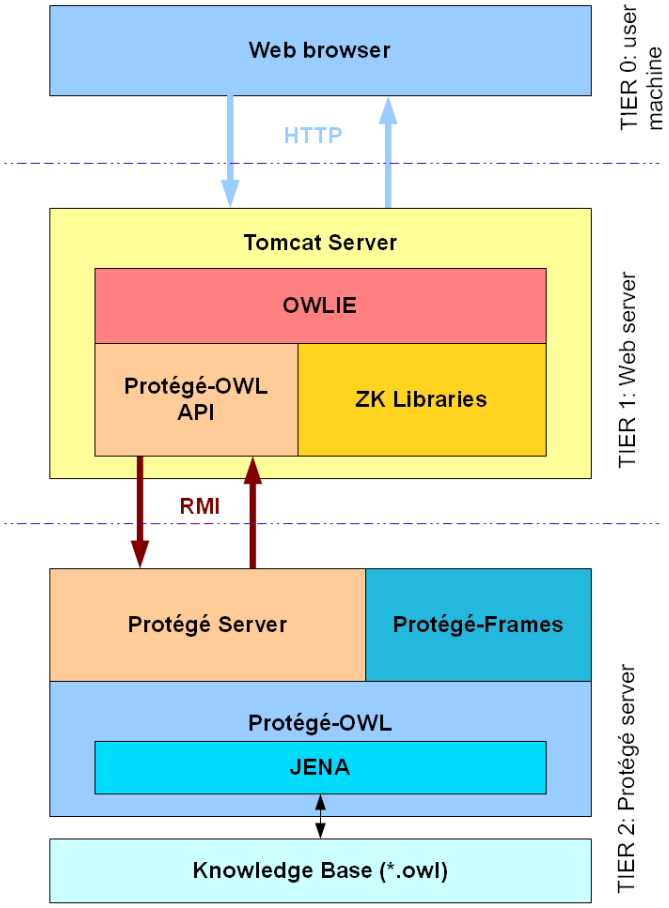


Figura 3.2: *Schema a tre livelli di OWLIE*

Il livello più esterno (livello 0) trova posto l'interfaccia utente, costituita da una *pagina web dinamica* visualizzabile all'interno di un comune browser web, pertanto senza la necessità che l'utente debba installare, configurare od eseguire alcun tipo particolare di software.

Il contenuto dell'interfaccia utente è generato lato server, al livello intermedio dell'architettura (livello 1), dove trova posto la cosiddetta *business logic* o logica di business, vale a dire il segmento di logica applicativa che si occupa di interpretare i comandi impartiti dall'utente, prelevare e rielaborare dati, e infine produrre i risultati da presentare all'utente.

L'interazione tra il livello 0 e il livello 1 avviene tramite *Hyper-Text Transfer Protocol* (HTTP), ragion per cui la logica applicativa a livello 1 deve essere contenuta in un web server particolare, chiamato *web application server*. Uno

di essi, che si adatta bene ad applicazioni sviluppate in Java, è Apache Tomcat[2].

L'applicazione web a livello 1 non si occupa della persistenza dei dati, demandando al livello più interno dell'architettura, il livello 2, la gestione delle politiche di accesso ai dati, condivisione delle risorse, consistenza e persistenza.

OWLIE sfrutta appieno l'architettura client-server a tre livelli, basandosi sul medesimo paradigma architetturale di WebProtege (esaminato in dettaglio al paragrafo 2.4.4 e schematizzato in figura 2.13), ritenuto un modello efficace su cui costruire l'applicazione.

OWLIE e WebProtege condividono in pratica i livelli 0 e 2 dell'architettura. In entrambi i casi, infatti, un browser web costituisce il client mentre, al livello più interno, il server di Protégé (vedasi paragrafo 2.4.2) si occupa di gestire l'accesso ai dati.

Una soluzione di questo tipo presenta numerosi vantaggi. OWLIE, così come WebProtege, è interamente contenuta a livello 1 dell'architettura. Ciò consente, in fase di progettazione ma anche in fase di implementazione, un notevole risparmio di tempo poiché Protégé rappresenta una piattaforma matura, sufficientemente stabile e ben documentata e pertanto non vi è necessità di progettare, implementare e testare un nuovo meccanismo di persistenza dei dati.

In secondo luogo, una caratteristica importante del server di Protégé è di implementare, al suo interno, un efficace sistema di controllo della concorrenza, rendendo di fatto possibile sviluppare all'interno dell'applicazione le funzionalità collaborative, che richiedono accesso cooperativo a risorse condivise e che sono fondamentali per raggiungere gli obiettivi minimi posti dal progetto, il mancato raggiungimento dei quali declassificherebbe OWLIE a un più semplice tool per uso individuale per il quale non vi sarebbe alcuna necessità di seguire un approccio distribuito, a maggior ragione a più livelli.

Un altro punto a favore della soluzione adottata è la presenza delle API di Protégé, un ricco set di classi ed interfacce Java che fornisce una mappa ad alto livello dell'ontologia (classi, proprietà, individui, ecc... sono presentati sotto forma di oggetti Java e il programmatore può utilizzarli senza doversi preoccupare di estrarre informazioni dal file XML) e implementa le operazioni fondamentali (ad esempio, l'aggiunta di una classe al dominio di una proprietà).

L'approccio considerato comporta anche alcuni svantaggi, legati in prima battuta alle caratteristiche rigide (a meno di modifiche dei sorgenti, ipotesi mai presa in considerazione durante la progettazione di OWLIE) del server di Protégé, che va configurato in fase di installazione dell'applicazione (come affrontato nelle appendici, paragrafo A.2). Si può ovviare a questo inconveniente, riducendo al minimo il ricorso a metodi implementati dal server, implementandoli direttamente a livello di applicazione web, come per esempio è stato fatto nel caso del tracciamento delle modifiche concorrenti).

[2] http://tomcat.apache.org

Un altro svantaggio è rappresentato dalla necessità di rilasciare un aggiornamento di OWLIE ogniqualvolta venga rilasciata una nuova versione di Protégé (e quindi delle sue API), in quanto si possono verosimilmente sperimentare problemi di compatibilità qualora la versione delle Protégé API incluse in OWLIE differisca dalla versione di Protégé implementata dal server. Questo problema è facilmente superabile, se si pone come vincolo (a cura dell'amministratore di sistema che installa e configura OWLIE) che le API incluse in OWLIE abbiano lo stesso numero di versione rispetto al server di Protégé con cui l'applicazione va a dialogare.

3.3.2 L'interfaccia grafica

Uno degli obiettivi principali di OWLIE è fornire un'interfaccia grafica completa ma al tempo stesso di facile interazione con l'utente, e soprattutto basata sul web e che quindi può essere visualizzata direttamente tramite un qualsiasi browser.

L'utilizzo del solo codice *Hyper-Text Markup Language* (HTML), pur dinamico e dotato di elementi interattivi (pulsanti, caselle di testo, mappe, ecc...) è di gran lunga insufficiente per realizzare un editor di ontologie. L'interfaccia deve essere arricchita con componenti e controlli tipici di un'applicazione tradizionale, vale a dire liste, alberi, menu a cascata, menu contestuali, e via discorrendo. Un'applicazione web che presenta tali caratteristiche viene definita *Rich Internet Application* (RIA).

Le applicazioni descritte nel paragrafo 2.3.2 utilizzano un'interfaccia RIA, come pure WebProtege, che presenta una GUI decisamente accattivante, anche se, per le motivazioni esaminate nei paragrafi precedenti, non ancora basata su un sistema maturo.

Una tecnica che consente di ottenere interfacce utente particolarmente complesse prende il nome di AJAX, che si basa sullo scambio di dati tra browser web e application server in modo asincrono, cioè senza necessità che l'utente aggiorni manualmente la pagina.[15]

Un'applicazione web tradizionale interagisce con l'utente inviando al server una richiesta di aggiornamento della pagina, sotto forma di messaggio *HTTPRequest*, quando si verifica un evento lato client, che può essere il click sul pulsante "invia" di un modulo oppure la semplice richiesta di aggiornamento della pagina. Il server risponde inviando al client il codice HTML che compone la pagina, unitamente alle parti non testuali della pagina stessa, se presenti (ad esempio le immagini). È evidente che se la pagina contiene molto codice o elementi non testuali pesanti, il caricamento delle informazioni può essere lento, comportando un'attesa da parte dell'utente, durante la quale non è possibile interagire nuovamente con la pagina. Ciò comporta, specialmente se la mole di dati è grande, un notevole spreco di banda.

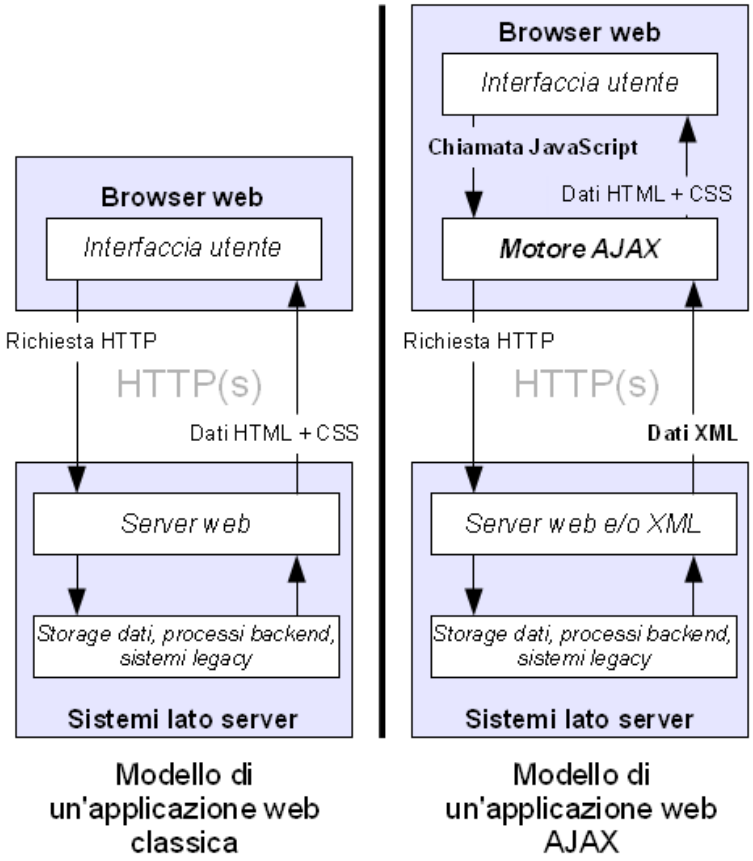


Figura 3.3: *Confronto tra come un'applicazione web tradizionale elabora l'input dell'utente e come lo fa un'applicazione web AJAX*

Un'applicazione AJAX, al contrario, viene aggiornata solo quando necessario e non sempre l'intervento dell'utente produce una richiesta al server. La pagina mostrata all'utente è composta da codice che include tag HTML classici collegati a fogli di stile CSS come in una pagina tradizionale, e una certa mole di script (scritti in JavaScript o più raramente in altri linguaggi) allo scopo di manipolare il *Document Object Model* (DOM) della pagina. L'interazione tra browser web e client non è effettuata direttamente dall'utente partendo dalla pagina, bensì da un *motore* che interpreta i comandi JavaScript, e produce variazioni all'interfaccia utente, cioè alla pagina. Se è necessario interrogare il server per poter produrre i risultati richiesti, il motore AJAX invia una richiesta *XMLHTTPRequest* al server, che risponde inviando dati strutturati in XML.

Questo approccio consente innanzitutto un notevole risparmio di banda, ma soprattutto limita la quantità di elaborazioni che il server deve effettuare per soddisfare le richieste del client. Per contro, il browser web deve poter processare un numero maggiore di dati rispetto ad una pagina HTML tradizionale, e ciò può peggiorarne le prestazioni. Inoltre i browser di vecchia generazione,

sebbene ormai poco diffusi, non supportano AJAX. In figura 3.3 è schematizzato il confronto tra applicazioni web tradizionali e applicazioni web che fanno uso di AJAX.

L'acronimo AJAX denota un modello teorico e non una tecnologia in particolare. Ne esistono infatti varie interpretazioni che si traducono in altrettanti *framework* per lo sviluppo di applicazioni web interattive. WebProtégé per esempio utilizza *Google Web Toolkit* (GWT), uno strumento potente per realizzare applicazioni web dinamiche, che produce buoni risultati combinando l'utilizzo di Java e JavaScript.

Per quanto riguarda l'interfaccia grafica di OWLIE è stato ritenuto più opportuno utilizzare un altro framework, **ZK**[3].

ZK è un framework AJAX open-source, che permette di realizzare applicazioni web dotate di interfaccia utente paragonabile ad applicazioni desktop, e che, rispetto alle piattaforme concorrenti (tra cui GWT) offre vantaggi non trascurabili dal punto di vista del programmatore.

Se per GWT è necessaria una conoscenza, almeno basilare, di JavaScript, ZK consente di progettare ed implementare l'interfaccia dell'applicazione anche senza conoscere JavaScript o altri linguaggi di scripting. La progettazione della GUI può avvenire in modo diretto, specificando i componenti della pagina tramite un linguaggio di markup chiamato *ZK User-interface Markup Language* (ZUML), un linguaggio di markup basato su XML (sebbene non propriamente un suo derivato).

La descrizione in ZUML di una pagina (o di una parte di essa) è contenuta in un file `.zul` che viene posizionato lato server all'interno della directory principale dell'applicazione web pubblicata sul server, o una delle sue sotto-directory.

L'esempio sottostante riproduce un frammento di codice ZUML che, interpretato dal server mediante le librerie di ZK (che devono essere incluse nell'applicazione web), produce una finestra di login con l'immissione di nome utente e password, come ritratto in figura 3.4:

Frammento di codice 3.1: *Codice ZUML di finestra di login*

```
<?page title="ZK Page Demo"
       contentType="text/html;charset=UTF-8"?>
<zk>
   <window title="Login" closable="true"
           width="300px" border="normal">
      <vbox>
         <label>
            Type username and password then click Login
         </label>
         <hbox>
            <label>Username</label>
            <textbox id="usr" width="200px" />
```

[3] http://www.zkoss.org

```
        </hbox>
        <hbox>
           <label>Password</label>
           <textbox id="pwd" type="password"
                    width="200px" />
        </hbox>
        <button id="login" label="Login" />
      </vbox>
   </window>
</zk>
```

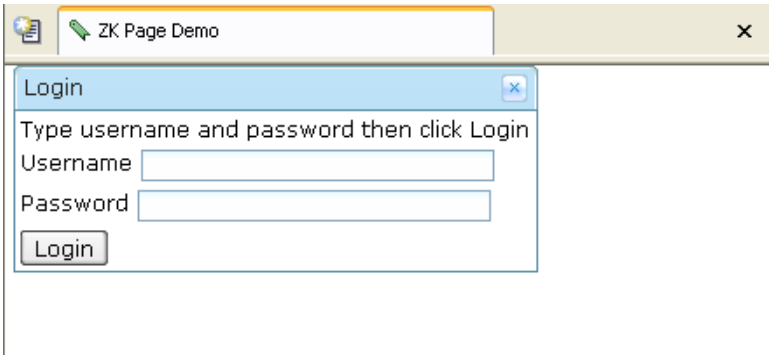


Figura 3.4: *Esempio di interfaccia utente ottenuta con l'uso di ZK*

Come si può osservare, un file ZUML contiene la descrizione degli elementi che compongono la pagina, disposti in sequenza e/o in cascata. Ciascuno di tali elementi (ad eccezione del tag `<zk>` che costituisce sempre e solo il primo elemento e all'interno del quale devono essere contenuti tutti gli altri) è definito *component*: ve ne sono parecchie decine, ognuno dei quali rappresenta un oggetto grafico o una parte di esso (per esempio, l'elemento `<treeitem>` costituisce un nodo in un albero, e non ha senso che appaia al di fuori di un elemento `<tree>`).

Gran parte dei *component* di ZUML è costituita da componenti *XML User-interface Language* (XUL)[4] a cui si aggiungono componenti realizzati da ZK (talvolta aggregando componenti XUL già esistenti). 'E tuttavia possibile da parte del programmatore inserire anche codice HTML, oppure definire nuovi componenti mediante l'aggregazione di elementi standard: le possibilità di estensione del linguaggio, per poter realizzare interfacce grafiche di notevole impatto e complessità che si avvicinano molto ad applicazioni desktop.

L'uso di ZUML pertanto semplifica notevolmente il progetto, l'implementazione e l'aggiornamento dell'interfaccia grafica, essendo in sé privo di codice

[4] XUL (`http://www.mozilla.org/projects/xul`) è un linguaggio di markup basato su XML e sviluppato dalla Mozilla Foundation per definire interfacce grafiche di applicazioni web o desktop. Software desktop ben noti, come Mozilla Firefox o Mozilla Thunderbird, possiedono un'interfaccia utente scritta in buona parte in XUL e interpretata da un motore grafico come *XULRunner*.

procedurale. L'interpretazione di ZUML avviene sempre lato server: il client riceve solamente codice XHTML arricchito con una certa mole di script (JavaScript) generati dall'interprete ZK in modo completamente trasparente al programmatore, che non deve inserire, di norma, nemmeno una riga di codice JavaScript.

La progettazione di un'applicazione interattiva, che sia quindi in grado di interpretare i comandi impartiti dall'utente lato client, è supportata da ZK per mezzo di un sistema di gestione degli eventi, analogo alla gestione degli eventi tipica di un'applicazione desktop sviluppata in Java utilizzando le librerie *swing*.

Per ciascun *evento* (click su un pulsante, selezione di un elemento in un albero, chiusura di una finestra, ecc...) è necessario definire un *EventListener*, uno speciale oggetto che, applicato ad un certo componente, si mette in attesa che si verifichi l'evento per cui è stato designato, e quando ciò avviene esegue un certo insieme di istruzioni.

Gli EventListener possono essere definiti "inline", cioè inserendo uno script all'interno del file ZUML (gli script possono essere scritti in Java, che è l'opzione predefinita, oppure in un altro linguaggio di scripting, come Phyton, JavaScript o Ruby). In alternativa è possibile associare ad ogni componente una classe Java (che deve essere compilata prima di pubblicare l'applicazione sul server) per gestirne gli eventi. Quest'ultimo approccio è indicato soprattutto nel caso di metodi di gestione degli eventi particolarmente pesanti, e qualora si desideri adottare il design-pattern *Model-View-Controller* (MVC) in cui l'interfaccia (view) deve essere separata dalla gestione degli eventi (controller). Una classe Java creata appositamente per gestire gli eventi è definita *composer*.

Talvolta come già accennato può risultare comodo definire componenti ad hoc, che possano essere utilizzati più volte all'interno della stessa applicazione senza necessità di ridefinirli ogni volta. Per fare un esempio, un editor di ontologie può prevedere un componente che consenta di modificare un individuo, una proprietà o un concetto. Definire un "editor" diverso per ciascuna risorsa dell'ontologia è contrario al principio di riutilizzo del codice, e oneroso dal punto di vista dell'implementazione nonché in fase di debug. In questo caso è opportuno definire e implementare un componente "parametrico" che possa essere richiamato ogni volta senza doverne variare l'implementazione, in piena sintonia con i principi della programmazione orientata agli oggetti.

ZK mette a disposizione del programmatore un set di API Java che permette di realizzare nuovi componenti come estensioni (sia in senso logico, sia in senso sintattico) di elementi esistenti. Ogni componente ZUML è infatti una classe Java che implementa l'interfaccia `Component`, la quale definisce una serie di metodi comuni ad ogni componente. Creando una nuova classe che è estensione di un componente già esistente, si definisce un nuovo elemento di cui è possibile impostare l'aspetto grafico, e modificare il comportamento aggiungendovi le caratteristiche desiderate.

Il frammento di codice 3.2 sotto riportato definisce un componente `LoginWindow` analoga a quella mostrata in figura 3.4.

Frammento di codice 3.2: *Componente personalizzato basato su elemento ZK*

```
import org.zkoss.zul.*

public class LoginWindow extends Window{

   private Button login;
   private Textbox pwd;
   private Textbox usr;

   public LoginWindow(){
          // instanzia l'oggetto padre
      super();
      this.setWidth("300px");
      this.setClosable(true);
      this.setBorder("normal");

      // crea la struttura del componente
      Vbox vbox1=new Vbox();
      Label lbl1=new Label("Type username and password
                           then click Login");
      vbox1.appendChild(lbl1);

      Hbox hbox1=new Hbox();
      Label lbl2=new Label("Username");
      usr=new Textbox();
      usr.setWidth("200px");
      hbox1.appendChild(lbl2);
      hbox1.appendChild(usr);
      vbox1.appendChild(hbox1);

      Hbox hbox2=new Hbox();
      Label lbl3=new Label("Password");
      pwd=new Textbox();
      pwd.setWidth("200px");
      hbox2.appendChild(lbl3);
      hbox2.appendChild(pwd);
      vbox1.appendChild(hbox2);

      login=new Button("Login");
      vbox1.appendChild(login);

      this.appendChild(vbox1);
   }

}
```

L'approccio Java puro, sebbene a parità di interfaccia da comporre richieda un maggior numero di istruzioni rispetto all'utilizzo di ZUML, presenta alcuni vantaggi, oltre a quello, già osservato in precedenza, di agevolare il riutilizzo del codice.

Il vantaggio più evidente è la possibilità di definire in un'unico file `.java` sia la descrizione del componente da creare (ovvero, da quali sotto-componenti è composto), sia le routine di gestione degli eventi. Ovviamente questo approccio è opposto al design pattern MVC, che prescrive una separazione tra la presentazione delle informazioni e la gestione degli eventi.

Un ulteriore vantaggio è dato dalle migliori performances esecutive. Una routine di gestione degli eventi contenuta in una classe compilata *prima* del deploy sul server ha tempi di accesso ed esecuzione inferiori a uno script inserito in un file ZUML. Tali script, infatti, anche se scritti in Java (che normalmente è un linguaggio compilato) passano attraverso un interprete al posto di un compilatore, ed è noto che, tranne poche eccezioni, un compilatore ottimizza il codice meglio di quanto non riesca a fare un interprete[5].

Il vantaggio delle classi pre-compilate rispetto a ZUML è rilevante quando tutte le fasi di sviluppo dell'applicazione sono state completate. In fase di programmazione e di debug, al contrario, si evidenzia come sia molto più comodo per il programmatore servirsi di ZUML. Infatti, una modifica a un file `.zul` non richiede l'arresto e il riavvio del server Tomcat, ma solamente l'aggiornamento della pagina visualizzata dal browser lato client. Una modifica a una classe comporta la sua ricompilazione e la pubblicazione della versione modificata della classe sul server, che perciò deve essere arrestato e poi riavviato.

Ritenuti i vantaggi dello sviluppo secondo l'approccio Java puro superiori ai relativi svantaggi, lo sviluppo di OWLIE è avvenuto in tal senso. Con alcune eccezioni, che saranno esaminate, assieme alla struttura del software, a partire dal paragrafo seguente.

3.3.3 Progetto strutturale

Un utilizzo intelligente di ZK permette di realizzare software modulari ed estendibili, a patto che l'applicazione sia progettata in tal senso e possa prevedere estensioni future.

Per applicazione modulare si intende un'applicazione che ammetta un qualche genere di modulo o componente principale al quale si connettono diversi moduli periferici, ognuno dei quali esegue una certa operazione o mette a disposizione all'utente una data funzionalità. A loro volta tali moduli possono istanziare al loro interno altri sub-moduli, creando in questo modo uno schema "a stella".

OWLIE è frutto di un'interpretazione di questo *pattern* che prevede:

[5] per la precisione, gli script interni ai documenti ZUML sono scritti in un linguaggio interpretato che utilizza la stessa sintassi e la stessa semantica di Java, *BeanShell* (`http://www.beanshell.org`). Questa differenza è trasparente per il programmatore, che può realizzare gli script "fingendo" di scrivere in vero codice Java.

1. una serie di classi dotate di metodi pubblici, che implementano funzioni di *backend* accessibili a tutti i moduli di livello superiore (ad esempio un *bean*[6] che memorizzi i dati della sessione e che fornisca i metodi per estrarli e manipolarli). Queste classi non processano direttamente eventi generati dall'utente e non contengono codice atto a produrre elementi grafici.

2. un modulo centrale, caricato automaticamente ad ogni nuovo accesso tramite browser. Questo modulo si occupa di inizializzare eventuali variabili o bean di sessione, e di fornire l'interfaccia grafica iniziale. Il modulo centrale istanzia e richiama i moduli periferici.

3. una serie di moduli periferici, definiti altresì moduli *funzionali* poiché ognuno di essi ha una funzione ben definita e aggiunge i propri elementi grafici all'interfaccia utente (con la possibilità di gestire i relativi eventi). Ogni modulo periferico può istanziare al suo interno componenti condivisi e richiamare metodi delle classi di backend tra cui i bean di sessione, ma non può istanziare altri moduli al medesimo livello. Inoltre, ogni modulo periferico non è accessibile dall'utente se non attraverso l'interfaccia del modulo principale.

4. un insieme di componenti condivisi, ognuno dei quali implementa una funzionalità richiesta da uno o più moduli periferici (o da altri componenti condivisi). Possono produrre o costituire essi stessi elementi dell'interfaccia utente, implementando i relativi sistemi di gestione degli eventi. I componenti condivisi non comunicano direttamente con l'oggetto chiamante, ovvero non possono effettuare retroazione (se non utilizzando un sistema passivo descritto in seguito).

5. una serie di classi accessorie, ognuna delle quali costituisce uno strumento ad uso di uno o più moduli dell'applicazione. Tra queste classi, che possono essere definite *utilities*, ve ne sono alcune che annoverano esclusivamente metodi statici e/o costanti.

L'applicazione è quindi costituita da più livelli funzionali (da non confondersi, ovviamente, con i livelli dell'architettura client-server descritta al paragrafo 3.3.1).

Il modello funzionale di OWLIE è schematizzato nella figura 3.5.

Il *backend* è costituito da un bean di sessione (`SessionManager`) il cui scopo è quello di memorizzare lo stato dell'utente durante la sessione di lavoro, e fornire

[6] Un *bean* è per definizione una classe con costruttore che non riceve parametri e una serie di metodi per estrarre e cambiare lo stato dell'oggetto; un tipo particolare di bean può essere utilizzato in un'applicazione web per memorizzare i dati di un utente durante la sua sessione di lavoro e per gestire la comunicazione con i livelli inferiori dell'architettura client-server

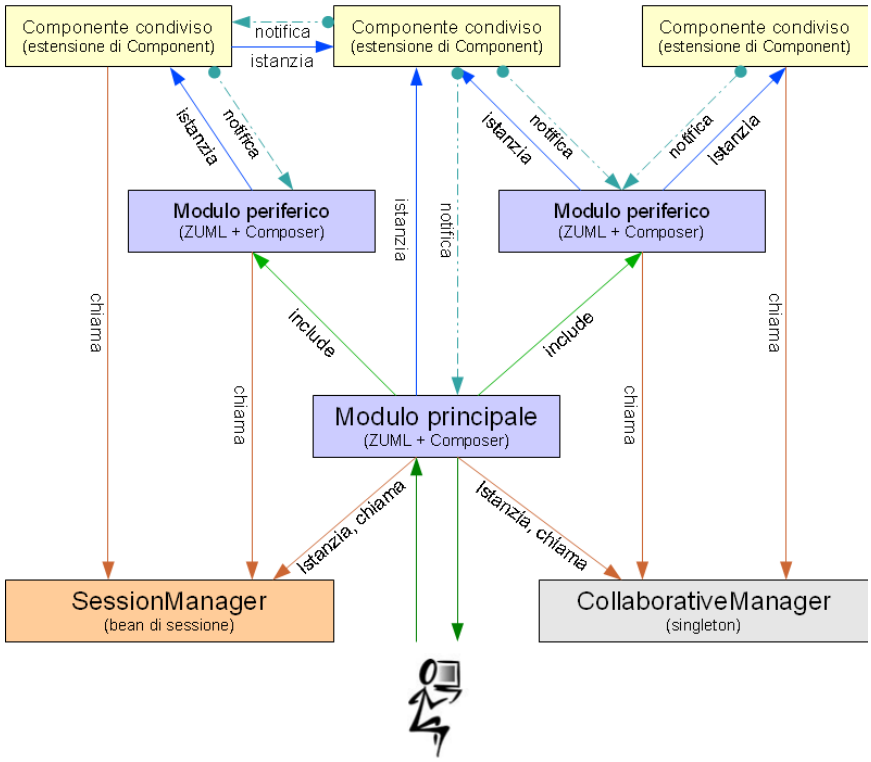


Figura 3.5: *Rappresentazione schematica della struttura di OWLIE*

il supporto alla comunicazione con il livello inferiore dell'architettura client-server (ossia il server di Protégé). Esiste un'istanza di SessionManager per ogni sessione lato client. Il SessionManager fornisce un set di metodi pubblici in lettura e in scrittura: può essere pertanto utilizzato da qualsiasi oggetto facente parte dell'applicazione.

Nel momento in cui l'utente, dal proprio browser web, avvia l'applicazione, il server tramite le librerie di ZK interpreta il contenuto di un documento ZUML, denominato `index.zul`. Tale file costituisce il modulo principale di OWLIE, e definisce pertanto l'interfaccia grafica di base, comune a tutti i moduli periferici.

Al documento ZUML è associata una classe che implementa l'interfaccia `Composer` di ZK, che come già spiegato nel paragrafo precedente permette di scrivere in Java puro il codice per i gestori degli eventi.

Il modulo principale, tramite il suo Composer provvede, ad ogni nuovo accesso (pertanto, appena dopo la sua stessa creazione) ad istanziare ed inizializzare un nuovo `SessionManager` e ad impostarlo come variabile di sessione (funzionalità offerta da ZK).

All'interno del codice del Composer è previsto che, immediatamente dopo l'accesso oppure a seguito di un intervento da parte dell'utente (ad esempio, un login avvenuto con successo), siano caricati uno o più moduli funzionali tramite

un metodo di *inclusione* (vale a dire, la grafica del modulo compare all'interno della stessa pagina relativa al modulo principale e non c'è cambio di contesto). Ciascuno di essi è costituito, come il modulo principale, da un documento ZUML al quale può essere associato un Composer (qualora il modulo sia interattivo, e cioè preveda che l'utente possa produrre eventi da processare).

I moduli funzionali previsti sono inizialmente quattro, fatta salva la possibilità di estensioni future qualora si riveli conveniente implementare nuove funzionalità. I primi tre moduli rappresentano le diverse *viste* di un'ontologia, secondo uno schema di facile comprensione ed utilizzo implementato a sua volta da altri software esaminati nel capitolo precedente, uno tra tutti Protégé (vedasi paragrafo 2.4).

- ***Class View*** – vista delle classi (o concetti atomici) con possibilità di selezione, aggiunta, cancellazione, modifica;
- ***Property View*** – vista delle proprietà (o ruoli) con la possibilità di compiere tutte le operazioni su di essi;
- ***Individuals View*** – vista delle istanze (o individui) con la possibilità di selezione, clonazione, aggiunta, eliminazione e modifica;

Il quarto modulo prende il nome di ***Graph View*** e fornisce supporto per i previsti visualizzatori grafici dell'ontologia.

Ogni modulo è indipendente dagli altri e non può interagire con gli altri, così come non interagisce con il modulo principale (l'interazione è unidirezionale e limitata alla sola inclusione). Tuttavia ogni modulo, tramite il relativo Composer, può richiamare il SessionManager dalla sessione in corso e dialogare con esso.

Il principio di riutilizzo del codice suggerisce di implementare una certa funzione una sola volta. Questo principio vale anche per i componenti ZK. Dal momento che moduli diversi talvolta debbono utilizzare la stessa combinazione di componenti (ad esempio, nel caso di OWLIE, l'albero dei concetti atomici o *named classes*), è opportuno raggruppare tale combinazione in un "macrocomponente" che svolga la funzione desiderata. Questi componenti condivisi sono scritti in Java puro per poter trarre massimo vantaggio dal riutilizzo del codice. Ogni componente è estensione di un componente ZK esistente (tipicamente, `Window`) e comunque implementa sempre l'interfaccia `Component` definita all'interno dei package di ZK.

Ogni componente implementa al proprio interno sia l'interfaccia grafica, sia la gestione degli eventi generati dai sotto-componenti interni. Inoltre, all'interno del codice è possibile utilizzare non solo elementi standard ZUML, ma anche altri componenti condivisi. Essendo stati instanziati all'interno di un'applicazione web, le variabili di sessione sono disponibili, quindi anche il SessionManager (anche se è possibile strutturare il codice in modo tale da non averne bisogno).

Non è possibile da parte di un componente interagire attivamente con l'oggetto da cui è stato istanziato, ovvero chiamarne un metodo. Ciò si rende necessario per garantire il riutilizzo del componente a prescindere da chi sia il chiamante. Esiste un modo per *notificare* al chiamante che qualcosa all'interno del componente è avvenuto (di norma un cambiamento di stato), e consiste nella generazione di un evento normalmente non associato ad alcuna azione utente, l'evento `onNotify` definito nell'oggetto `Events` di ZK. Si tratta di un evento creato appositamente per gestire situazioni di questo tipo. La condizione necessaria e sufficiente affinché questo meccanismo funzioni è che il chiamante abbia implementato al suo interno un gestore di eventi specifico per l'evento `onNotify` relativo al componente.

Nei frammenti di codice di seguito riportati è esemplificato il meccanismo di interazione descritto poc'anzi. Il frammento 3.4 si riferisce ad un componente costituito da una finestra (`Window`) contentente un pulsante (`Button`). Il frammento 3.3 si riferisce ad un oggetto (un componente esso stesso oppure un Composer) che chiama il componente definito in precedenza. Nell'esempio è illustrata la tecnica utilizzata in OWLIE per consentire ai componenti chiamati di informare il chiamante che qualcosa è avvenuto (in questo caso, la pressione del pulsante da parte dell'utente).

Frammento di codice 3.3: *Codice di esempio un generico componente condiviso*

```
//codice del componente condiviso
import org.zkoss.zul.*
public class ComponenteDiEsempio extends Window{

   public ComponenteDiEsempio{
      super();
      Button premiqui=new Button("Premi qui");
      this.appendChild(premiqui);

      //dichiarazione del gestore di eventi
      //associato alla pressione del pulsante
      premiqui.addEventListener(Events.ON_CLICK,
         new EventListener(){
            public void onEvent(Event argument)
                        throws Exception{
               //quando l'utente preme il pulsante,
               //l'istruzione seguente viene
               //eseguita
               createNotifyEvent();
            }
         }
      );
   }

   //metodo per generare un evento di notifica
   //"onNotify" associato a questo stesso oggetto
```

```
   private void createNotifyEvent(){
      Events.postEvent(Events.ON_NOTIFY, this, null);
   }
}
```

Frammento di codice 3.4: *Utilizzo e controllo di un componente condiviso*

```
//codice del chiamante
//(limitato al solo segmento di interesse)
...
   //il componente viene istanziato
   ComponenteDiEsempio pippo=new ComponenteDiEsempio();

   //il componente puo' essere modificato nel suo
   //aspetto grafico, essendo estensione di un
   //componente ZK predefinito ed ereditandone
   //proprieta' e metodi
   pippo.setClosable(true);
   pippo.setBorder("normal");
   pippo.setTitle("Premi il pulsante!");

   //il componente pippo genera un evento quando al suo
   //interno viene premuto il pulsante. Tale evento va
   //intercettato.
   pippo.addEventListener(Events.ON_NOTIFY,
      new EventListener(){
         public void onEvent(Event argument)
                        throws Exception{
                System.out.println("il componente
                ha cambiato il suo stato");
         }
      }

   //pippo e' una finestra: la si puo' mostrare all'utente
   pippo.doOverlapped();
...
```

Come si vede dall'esempio riportato, questa tecnica di comunicazione tra oggetto chiamato e oggetto chiamante è parecchio laboriosa dal punto di vista del programmatore e non sempre l'analisi del software realizzato in questo modo è immediata: tuttavia un design-pattern così formulato rende i componenti condivisi indipendenti dal chiamante.

La struttura dell'applicazione si completa con l'aggiunta delle funzionalità collaborative, di cui si parlerà più estesamente in seguito. La classe `CollaborativeManager` ammette una sola istanza (singleton) a livello di intera applicazione web. Ciò rende adatto l'oggetto a memorizzare dati comuni a tutte le sessioni utente.

3.3.4 Scelte notazionali e semantiche

Tra gli obiettivi di OWLIE vi è la facilità di apprendimento all'uso. Il raggiungimento di questo obiettivo è vincolato all'uso di notazioni e semantica facilmente interpretabili e il più possibile conformi agli standard nonché, ove possibile, alle consuetudini espresse da applicazioni preesistenti, tra cui naturalmente Protégé.

Per quanto riguarda le notazioni, in OWLIE è stato scelto di utilizzare un codice grafico ben preciso. Ogni risorsa è associata a un'icona che ne stabilisce il *tipo*, soprattutto nei casi in cui non è esplicitamente dichiarato.

Le icone utilizzate sono le seguenti:

-  per la classe universale (*owl:Thing*)
-  per i concetti atomici (*named classes*)
- diverse icone per i concetti anonimi (*anonymous classes*):
  -  per i concetti anonimi costituiti da unione di altri concetti (anonimi o non)
  -  per le intersezioni di concetti
  -  per i concetti definiti mediante enumerazione di individui
  -  ,  ,  per i concetti definiti mediante restrizione di cardinalità su ruoli (rispettivamente cardinalità minima, esatta, massima)
  -  per i concetti definiti mediante qualificatore esistenziale di un ruolo
  -  per i concetti definiti mediante qualificatore universale di un ruolo
  -  per i concetti definiti mediante vincolo "ha valore"
-  per i concetti che sono definiti da equivalenze
-  per i ruoli (*object properties*)
-  per le proprietà sui dati (*datatype properties*)
-  per gli individui (istanze di concetti)

A tali icone si aggiungono simboli grafici definiti "ad hoc" per rappresentare sia le azioni (e costituiscono quindi l'interfaccia di pulsanti e barre degli strumenti) sia elementi dell'ontologia per i quali non è definita una rappresentazione standard (ad esempio, i letterali).

Per quanto concerne le scelte semantiche, vengono rispettate ed utilizzate ove possibile le regole standard di OWL, che prevedono una semantica *congiuntiva*.

La semantica di OWL viene applicata alle relazioni di *tipo*. Ciò significa che se una classe A ha indicate come superclassi le classi B, C, D, ciò deve essere interpretato come "ogni elemento di A deve appartenere sia a B, sia a C, sia a D". Allo stesso modo, se un individuo è istanza delle classi L, M, N, ciò significa che è istanza di tutte e tre le classi contemporaneamente.

Al contrario, le relazioni di *dominio* e *codominio* si applica una semantica *disgiuntiva*. Questa scelta è legata esclusivamente a un fattore tecnico-pratico: le API di Protégé adottano la medesima semantica e dal punto di vista del programmatore è più semplice sfruttarne le caratteristiche in modo diretto piuttosto che implementare logica di conversione.

La principale conseguenza di questa scelta è che, per esempio, la lista delle classi che compongono il dominio di un ruolo non deve essere letta allo stesso modo di una lista di classi che costituiscono il tipo di un'altra classe. In OWLIE, come in Protégé, i membri del dominio o del codominio di un ruolo sono legati da una relazione logica di unione (OR) anziché intersezione (AND).

Nella semantica di OWLIE, dato un generico ruolo R, dire che le classi A, B, C sono nel dominio di R significa che un individuo x per poter avere un ruolo R uscente deve appartenere alla classe A, o alla classe B, o alla classe C (in modo non esclusivo, se le classi non sono disgiunte).

Capitolo 4

Implementazione

Questo capitolo costituisce un approfondimento rispetto alle scelte implementative che seguono dalle più ampie scelte strutturali e di architettura, che sono state oggetto del capitolo precedente.

In questo capitolo, pertanto, si andranno ad analizzare le componenti di OWLIE, seguendo un approccio top-down.

Saranno esaminati nella prima parte del capitolo gli elementi che costituiscono il nucleo dell'applicazione e che pertanto implementano le operazioni fondamentali descritte nell'analisi degli scenari d'uso.

Nella seconda parte del capitolo sarà invece preso in esame il *sottosistema collaborativo* ovvero un insieme di aggiunte al sistema principale che consente di ottenere le funzionalità di accesso condiviso da parte di più utenti, uno degli obiettivi ineludibili del progetto.

4.1 Il nucleo di OWLIE

Sulla base delle linee guida descritte nel paragrafo precedente, alla fase di design è seguita la vera e propria codifica delle classi e delle interfacce che compongono l'applicazione.

Ovvie esigenze di sintesi non permettono di riportare in questa sezione l'intero codice sorgente: l'obiettivo dei paragrafi che seguiranno è fornire una descrizione schematica delle finalità di ciascun modulo o componente implementato e delle relazioni con gli altri elementi dell'applicazione.

In figura 4.1 è rappresentata sinteticamente la struttura di OWLIE sotto forma di grafo orientato. I nodi del grafo rappresentano gli elementi che costituiscono il software, contraddistinti da un colore diverso a seconda della tipologia (e del ruolo che essi assumono all'interno del programma). Gli archi, ove indicati, assumono il significato di relazione tra un elemento e un altro.

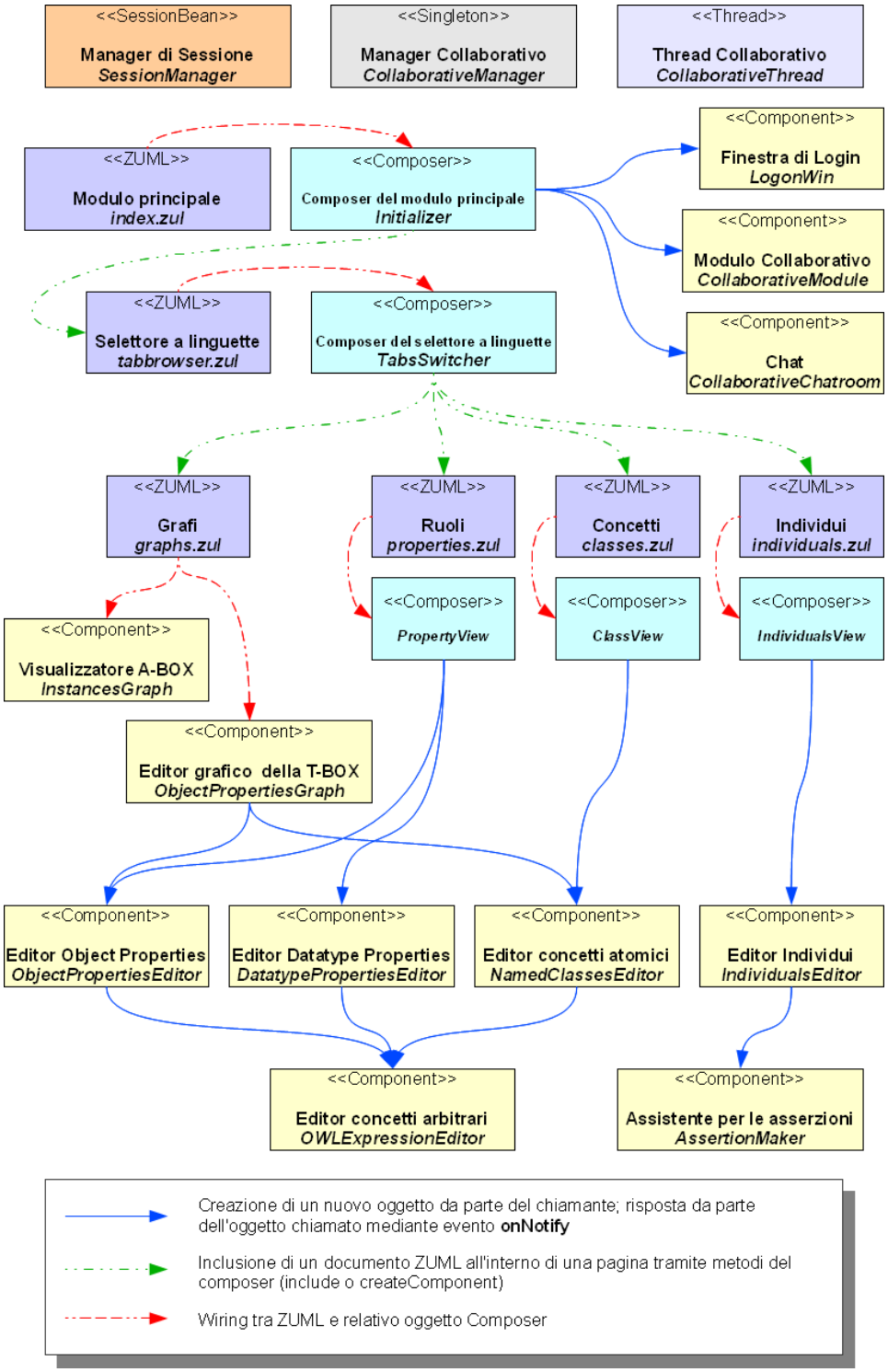


Figura 4.1: *Diagramma strutturale di OWLIE*

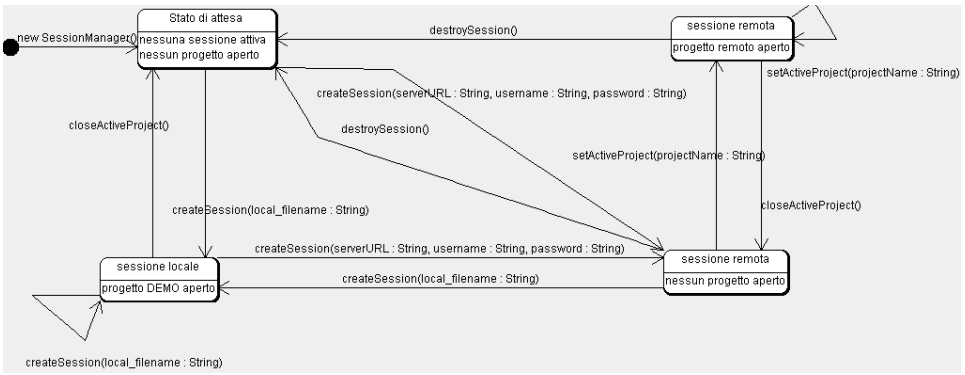


Figura 4.2: *Diagramma a stati del SessionManager*

4.1.1 Il SessionManager

Il `SessionManager` è un *bean* che implementa tutte le operazioni che non prevedono la generazione di elementi dell'interfaccia utente, bensì l'interazione tra i moduli del sistema e i livelli sottostanti.

Compito principale del SessionManager è fornire ai moduli e ai componenti dell'applicazione un insieme di metodi in lettura e scrittura per accedere all'ontologia ed estrarne risorse, in modo completamente trasparente all'utilizzatore.

Il SessionManager ha l'aspetto di un *Plain Old Java Object* (POJO), vale a dire un oggetto Java semplice con costruttore e metodi pubblici. Non è pertanto un *Enterprise Javabean* (EJB) e viene utilizzato come variabile (attributo) di sessione.

L'inizializzazione dell'oggetto è effettuata dal composer del modulo principale al momento dell'accesso da parte dell'utente, anche se quest'ultimo non ha ancora effettuato operazioni.

Il SessionManager fornisce due tipologie di metodi pubblici. Esistono metodi che producono un cambiamento di *stato* dell'oggetto, e metodi per estrarre informazioni o dati in relazione allo stato corrente.

Come rappresentato dallo *statechart* in figura 4.2, l'oggetto (e quindi l'intero sistema, dal momento che è possibile utilizzare un modulo solo se il SessionManager si trova in uno stato preciso) può trovarsi in quattro diversi stati:

1. *stato di attesa* – l'oggetto `''SessionManager''` è stato inizializzato ed è presente in memoria come variabile di sessione. Non essendo stata alcuna operazione di login da parte dell'utente, non esiste alcuna connessione con il server di Protégé, e nessuna ontologia è attiva (i funzionali e i componenti dell'interfaccia utente non possono chiamare i metodi di estrazione dei dati in quanto viene sollevata un'eccezione).

2. *sessione locale* – tramite il metodo `createSession(String filename)` è stato selezionato un progetto "de-

mo" (tra le ontologie di esempio fornite all'interno dell'archivio web di OWLIE) che non risiede sul server di Protégé. I metodi per l'estrapolazione dei dati sono parzialmente disponibili, i moduli funzionali e i componenti a più alto livello possono accedere all'ontologia come se fosse un progetto remoto (senza la possibilità di salvare le modifiche).

3. *sessione remota* – tramite il metodo `createSession(...)`[1] è stato aperto un canale di comunicazione con il server di Protégé sul quale è stata avviata una sessione remota. Restano non disponibili i metodi per l'estrapolazione della base di conoscenza.

4. *sessione remota con progetto attivo* – tramite il metodo `setActiveProject(String projectName)` una delle ontologie presenti sul server è stata caricata nel SessionManager: tutti i metodi di estrazione di dati sono disponibili.

Il SessionManager cessa il proprio ciclo di vita quando l'utente termina la sessione di lavoro (ovvero quando chiude la finestra del browser). In questa occasione, se il sistema si trova nello stato di attesa oppure è instaurata una sessione locale, la chiusura abrupta della connessione non causa inconsistenze a livello di server di Protégé.

Nel caso invece sia stata creata una sessione remota (dopo l'operazione di login eseguita con successo) con o senza apertura di un'ontologia, l'interruzione improvvisa della sessione utente non è seguita dalla chiusura della relativa sessione sul server di Protégé. Per questa ragione sono sconsigliate le interruzioni abrupte della connessione senza prima aver effettuato il logout.

4.1.2 Il modulo principale

Il modulo principale è implementato nel documento ZUML `index.zul`. Tale denominazione non è frutto di una scelta casuale in quanto esso è il primo ad essere caricato in una qualsiasi applicazione ZK che conservi le impostazioni di default.

Al documento ZUML è associato un gestore di eventi (composer) costituito dalla classe Java `Initializer`.

In sé, il file ZUML implementa solamente la cornice esterna dell'interfaccia grafica (che comprende il titolo, la barra dei menu, la barra di stato, la barra di fondo) e imposta le dimensioni predefinite dei vari *box* dove caricare i moduli. La figura 4.3 fornisce un'illustrazione di come si presenta OWLIE appena l'utente apre la pagina iniziale.

Il composer associato si occupa di intercettare gli eventi dell'utente: siccome il caricamento dei moduli funzionali avviene solo dopo aver effettuato di login, il composer controlla quali elementi del menu devono essere abilitati, quali devono

[1] `createSession(String serverURL, String username, String password)`

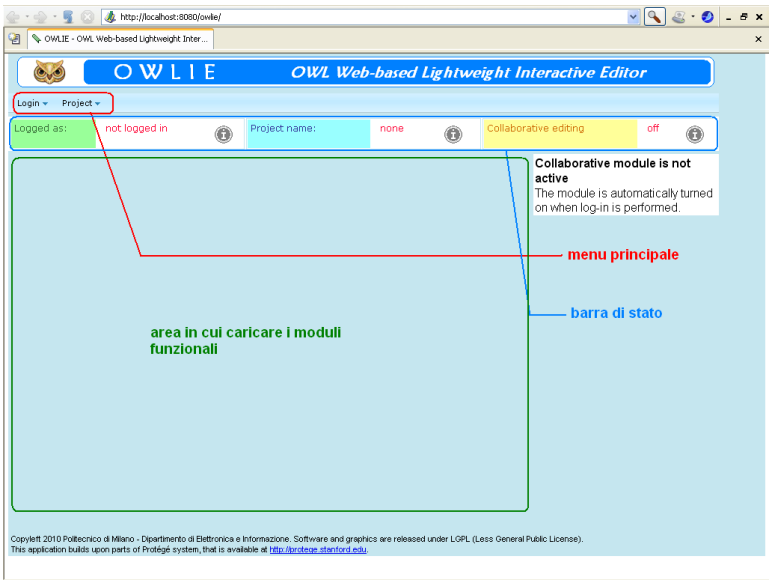


Figura 4.3: *Interfaccia del modulo principale di OWLIE*

essere disabilitati, che aspetto deve avere la barra di stato e cosa deve essere visualizzato nel box centrale.

A seguito dell'operazione di login, il composer inserisce nell'area centrale della pagina un elenco dei *progetti* (ontologie) disponibili sul server di Protégé, o in alternativa, se l'utente ha scelto la modalità dimostrativa, l'elenco delle ontologie di esempio presenti nella directory `demo` dell'applicazione web. Gli elenchi di progetti sono creati tramite una routine interna al composer e non utilizzando un componente separato, cosa che invece avviene con la finestra di login.

Quando l'utente sceglie di aprire un progetto ovvero un'ontologia, il composer richiama un modulo ausiliario, una sorta di connettore tra il modulo principale e i moduli funzionali, implementato nel file ZUML `tabbrowser.zul` a cui è associato il composer `TabSwitcher`. Tale modulo altro non è che il selettore a linguette (*tabs*) che permette all'utente di selezionare su che modulo lavorare. Il compito del composer in questo caso è quello di gestire gli eventi generati dall'apertura di una linguetta: quando l'utente clicca con il mouse su un tab, il composer istanzia il modulo corrispondente, e stacca tutti gli altri, consentendo di risparmiare risorse del server (in quanto di norma in ZK tutti i tab restano in memoria anche se non attivi, vale a dire anche se "coperti" dal modulo su cui l'utente sta lavorando in un dato momento).

Gli eventi generati dagli elementi del menu principale, tuttavia, restano in capo al composer del modulo principale (in quanto essi sono definiti in quest'ultimo), che può intercettare il comando di "chiusura" del progetto impartito dal-

l'utente. Nella versione attuale non sono previsti sistemi di salvataggio , in quanto è il server di Protégé a compiere il salvataggio automatico dell'ontologia (se correttamente configurato a tal fine). Per tale ragione all'utente che chiude il progetto agendo sul corrispondente comando nel menu principale, non viene visualizzato alcun messaggio o richiesta di conferma dell'azione. In tal caso il composer stacca dall'interfaccia il modulo ausiliario tornando alla condizione precedente, quella post-login, con l'elenco dei progetti disponibili.

L'operazione di logout invece riporta l'applicazione allo stato iniziale.

4.1.3 I moduli funzionali

I moduli funzionali di OWLIE, come già accennato in precedenza, sono tre (a cui si aggiunge il quarto modulo che non svolge altra funzione se non quella di supporto per il visualizzatore di A-BOX e l'editor grafico della T-BOX i quali saranno oggetto di uno dei prossimi paragrafi). Ogni modulo implementa una vista dell'ontologia, concentrandosi su un sottoinsieme dei suoi elementi costitutivi.

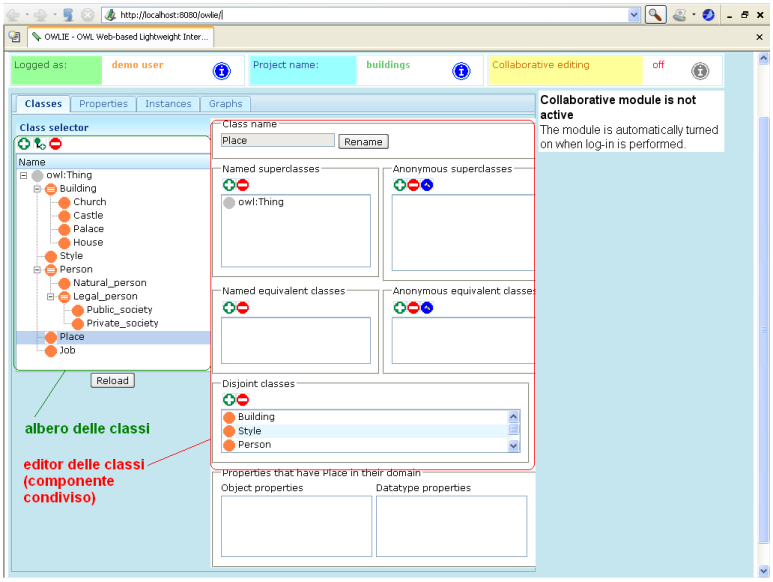


Figura 4.4: *Modulo della Class View*

Sebbene ogni modulo non sia vincolato ad un particolare *layout* (vale a dire uno schema rigido che indichi quali spazi del box occupare e con che cosa), tutti i moduli sono stati sviluppati in modo speculare, seguendo uno schema concettuale comune (facilitando così l'apprendimento dell'uso del software da parte dell'utente).

Ogni modulo è dotato di un selettore che trova posto nella parte sinistra del box. Tale selettore è tipicamente un componente condiviso, dotato quindi di una logica propria (che può prevedere menu contestuali, gestione di eventi, metodi

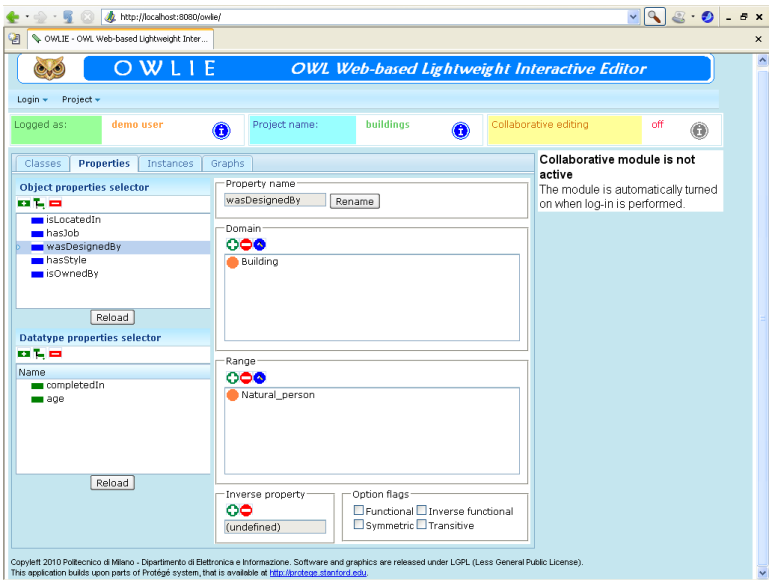


Figura 4.5: *Modulo della Property View*

di aggiornamento definiti al suo interno) e che è possibile controllare servendosi di routine di gestione degli eventi come descritto nel paragrafo 3.3.3.

Al di sopra del selettore vi è una barra degli strumenti in cui trovano posto dei pulsanti ciascuno dei quali è contraddistinto da una funzione particolare (aggiunta di un elemento, rimozione dell'elemento selezionato, ecc...).

La selezione di un elemento (il selettore di norma non permette la selezione multipla) produce un evento che viene intercettato dal composer, il quale istanzia un componente chiamato *editor* e lo inserisce nell'interfaccia all'interno dell'area ad esso dedicata nella parte destra del box. Esiste un editor specifico per ogni tipo di elemento dell'ontologia, e si tratta di componenti condivisi, dotati quindi di una logica propria. Una volta istanziato l'oggetto editor, il composer si disinteressa di ciò che avviene al suo interno, ovvero di ciò che l'utente compie. Come si vedrà in seguito, infatti, ogni modifica effettuata all'interno dell'oggetto editor viene propagata all'ontologia in modo diretto senza passare per il composer che l'ha invocato. Tuttavia il composer intercetta l'evento `onNotify` elevato dall'editor in modo tale da gestire eventuali modifiche che comportino il dover aggiornare il selettore, quali ad esempio operazioni di ridenominazione o di stravolgimento della gerarchia.

La **Class View** (implementata dal documento ZUML `classes.zul` collegato al composer `ClassView`) fornisce una vista dei *concetti atomici* dell'ontologia, ossia le classi OWL dotate di nome ("named classes").

La gerarchia delle classi definite nell'ontologia è rappresentato da un albero posto sul lato sinistro del box, come rappresentato in figura 4.4.

Alla selezione di una classe da parte dell'utente segue l'apertura di una fines-

tra nella parte destra del box, inizialmente vuota, in cui appaiono l'editor della classe selezionata (in alto) e l'elenco dei ruoli di cui la classe selezionata è membro del dominio (in basso).

Ogni modifica apportata alla classe tramite l'editor (ad esempio, aggiunta o rimozione di una sopraclasse) viene propagata, se necessario, all'albero di selezione, senza che l'utente debba confermarla.

La **Property View** (implementata dal documento ZUML `properties.zul` collegato al composer `PropertyView`) fornisce una vista dei *ruoli* dell'ontologia, ossia le proprietà sugli oggetti ("object properties") e le proprietà sui dati ("datatype properties").

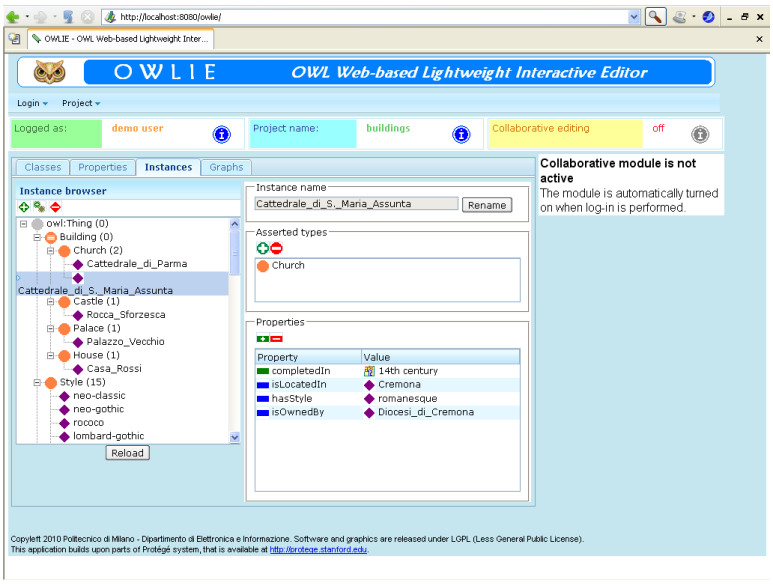


Figura 4.6: *Modulo della Individuals View*

Nella stessa view, sono accessibili sia le proprietà sugli oggetti, sia le sui dati, le cui gerarchie sono rappresentate da altrettanti alberi nella parte sinistra del box, come rappresentato in figura 4.5.

Analogamente a quanto avviene nella Class View, la selezione di una Datatype property o di una Object property produce l'apertura di una finestra di modifica che occupa l'apposito settore a destra del box.

La **Individuals View** (implementata dal documento ZUML `individuals.zul` collegato al composer `IndividualsView`) fornisce la vista degli *individui* dell'ontologia, le istanze delle classi.

Nella parte sinistra del box trova posto il selettore degli individui, sotto forma di albero i cui nodi intermedi rappresentano la gerarchia delle classi (tenendo presente che un individuo può essere istanza di più classi contemporaneamente) come rappresentato in figura 4.6.

Anche in questo caso, come nei moduli visti in precedenza, la selezione di un elemento dell'albero produce l'apertura del corrispondente editor. Siccome nell'albero di selezione sono presenti nodi misti (classi e loro istanze) selezionando una classe appare l'editor di classe, mentre selezionando un individuo appare l'editor degli individui. Questa funzionalità rende possibile effettuare operazioni sulle classi senza dover uscire dalla view degli individui.

4.1.4 I componenti condivisi

I selettori ad albero e gli editor di classi, proprietà e individui sono tutti componenti condivisi: non sono legati, come già visto, a un modulo in particolare ma possono essere potenzialmente istanziati e utilizzati da qualsiasi modulo funzionale. Esistono inoltre componenti richiamati e utilizzati all'interno di altri componenti.

I componenti condivisi si possono dividere in due categorie. Tenendo sempre presente che si tratta di componenti ZK a tutti gli effetti e quindi soggetti a un ciclo di vita ben delineato (creazione, inclusione in una pagina, assegnazione a un componente padre, orfanizzazione, dismissione), vi sono alcuni componenti il cui ciclo di vita non è legato a una particolare risorsa dell'ontologia bensì all'intera base di conoscenza. In questo gruppo trovano posto i selettori ad albero e a lista. Vi sono al contrario altri componenti che sono invece legati ad una specifica risorsa facente parte della *knowledge base*: in questa categoria trovano posto i vari editor (classi con nome, classi anonime, proprietà e individui).

Dal punto di vista della programmazione orientata agli oggetti, solo una piccola parte dei componenti è in sé rappresentata da oggetti *mutabili*, cioè per i quali sono definiti metodi che consentano di modificare lo stato interno dell'oggetto a partire da uno stato interno. Per tutti gli altri, la scelta della risorsa associata è definitiva e non può essere cambiata a posteriori.

I componenti condivisi comunicano con il chiamante per mezzo di eventi da essi generati (lasciando al chiamante l'onere di gestire tali eventi). Essendo ogni componente l'estensione di un componente ZK già esistente, va da sé che gli eventi generati da quest'ultimo sono comunque generati anche dal componente figlio. Lo stesso discorso si applica ai metodi e alle proprietà dell'oggetto.

Si consideri il componente `NamedClassesEditor` ossia l'editor dei concetti atomici. Esso è definito da una classe Java che estende un'altra classe contenuta nelle API di ZK, `Window`. A meno dei metodi sovrascritti (*overrides*) i metodi della classe `Window` sono ereditati da `NamedClassesEditor`, sicché è possibile, ad esempio, impostare il titolo della finestra tramite il metodo `.setTitle()`, ma anche gestire l'evento `onClose` generato quando l'utente preme il pulsante di chiusura della finestra, se presente.

L'evento `onNotify` al contrario non è generato dal componente `Window`, e lo si può utilizzare, come esaminato nel paragrafo 3.3.3, per produrre una al chiamante nel caso l'utente abbia compiuto qualche modifica alla risorsa associata, come un'operazione di ridenominazione.

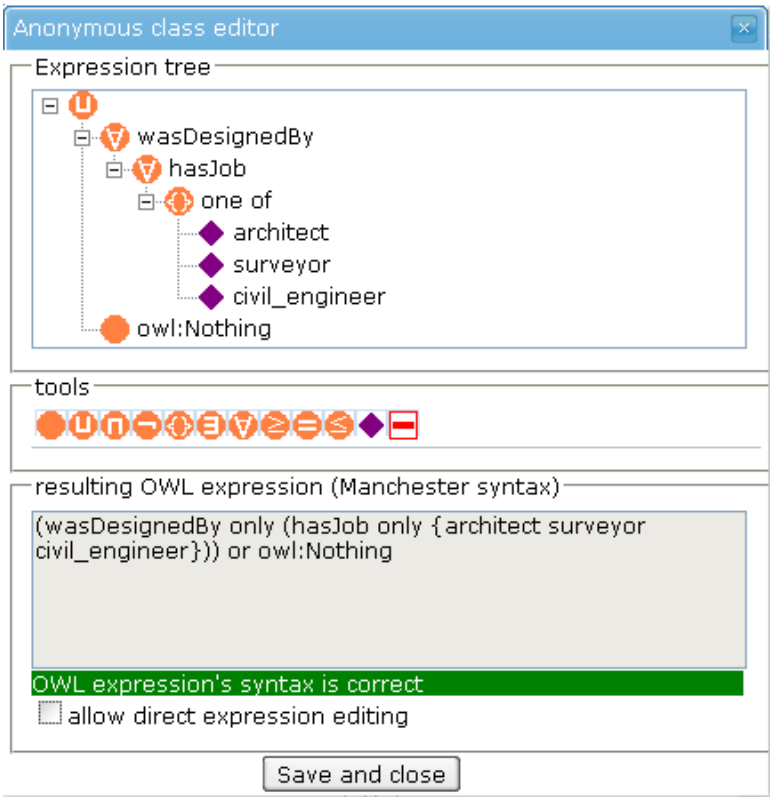


Figura 4.7: *Editor dei concetti arbitrari o espressioni OWL*

I componenti condivisi *non associati alla risorsa* sono sei:

- `LogonWin` – estensione di `Window`, è una finestra richiamata esclusivamente dal modulo principale (più precisamente dal suo composer) che consente l'immissione di nome utente e password; per dialogare con il server utilizza i metodi del `SessionManager`.

- `NamedClassesTree` – estensione del componente ZK `Tree`, è un albero i cui nodi rappresentano i concetti atomici ("named Classes") dell'ontologia, organizzati gerarchicamente. A differenza del componente grezzo `Tree`, che è vuoto al momento della creazione, il suo contenuto viene generato automaticamente al momento della chiamata del costruttore.
  Il costruttore ha diversi parametri che permettono di ottenere un albero adatto ad esigenze diverse. Esistono opzioni per visualizzare o meno il menu contestuale, per troncare i nomi troppo lunghi, o se visualizzare accanto al nome della classe il numero di individui che ne fanno parte. Un'ulteriore opzione permette di ottenere un albero contenente solo le classi che potrebbero essere impostate come superclasse di una data classe (in questo caso, l'oggetto diviene associato a una risorsa).

`NamedClassesTree` si riferisce alla variabile di sessione `SessionManager` per reperire informazioni circa le classi presenti nell'ontologia.

- `DataTypePropertiesTree` e `ObjectPropertiesTree` — estensioni di `Tree`, sono due componenti omologhi tra loro che rappresentano la gerarchia dei ruoli dell'ontologia, rispettivamente le Data Properties e le Object Properties. Esiste oltre al costruttore di default (che non ha parametri e produce un albero popolato con tutte e sole le proprietà visibili) un ulteriore costruttore che accetta come parametro un oggetto `OWLClass`[2] che produce un albero popolato con tutte e sole le proprietà che hanno la classe specificata nel proprio dominio.

- `IndividualsTree` — estensione di `Tree`, è un componente che rappresenta un albero delle classi in cui sono visualizzate anche le loro istanze (individui). Esiste un costruttore di default, che non accetta parametri, e un costruttore che accetta un parametro booleano, il quale serve ad impostare se visualizzare il menu contestuale o no.

- `IndividualsList` — estensione di `Listbox`, è un componente che rappresenta una lista di individui. Il costruttore di default produce una lista i cui elementi sono tutti e soli gli individui visibili dell'ontologia (cioè gli individui che non sono istanze di classi invisibili); esiste tuttavia un costruttore che produce un elenco delle istanze delle sole classi specificate.

I componenti condivisi *collegati a una risorsa* sono invece cinque, a cui se ne aggiungono due i quali svolgono funzioni speciali all'interno di un altro componente.

L'**editor delle classi** (`ClassEditor`) è una estensione del componente ZK `Window` che consente di modificare la *definizione* e le caratteristiche di un concetto atomico (stabilito in fase di creazione dell'oggetto come parametro del costruttore). Graficamente l'editor si può presentare come un semplice box inserito senza soluzione di continuità nel tessuto dell'interfaccia del modulo funzionale (come ritratto in figura 4.4) oppure come finestra pop-up modale[3]. In quest'ultimo caso in fase di creazione viene dotata di pulsante di chiusura.

L'editor delle classi è dal punto di vista dell'utente una semplificazione del corrispondente modulo di Protégé 4, al quale si ispira. La finestra è divisa in vari riquadri, corrispondenti ai diversi attributi della classe collegata. Il primo riquadro contiene il nome della classe (senza riportare l'URI completo ma

[2] la classe `OWLClass` è definita dalle API di Protégé e rappresenta una classe OWL, con nome o anonima

[3] una finestra *pop-up* in ZK è una finestra che non viene visualizzata assieme al resto dell'interfaccia ma in modo sovrapposto ad essa, come un oggetto fluttuante. Esistono varie modalità in cui una finestra può apparire sovrapposta al resto dell'interfaccia. OWLIE utilizza quasi esclusivamente pop-up di tipo *modale* in cui non è possibile interagire con il resto della GUI fintantoché la finestra pop-up rimane aperta

soltanto il nome abbreviato) e un pulsante di ridenominazione. Il secondo e terzo riquadro contengono le liste, rispettivamente, delle super-classi e dei concetti equivalenti. Per maggiore chiarezza rispetto a Protégé, le liste sono divise in due parti: la prima parte contiene soltanto i concetti atomici, la seconda parte i concetti arbitrari ossia le classi anonime. Il quarto e ultimo riquadro contiene la lista delle classi disgiunte.

Per ogni lista è disponibile una piccola barra degli strumenti per l'aggiunta e la rimozione di elementi dalla lista medesima; nel caso di classi anonime è presente un terzo pulsante che ne permette la modifica tramite l'editor delle classi anonime illustrato in seguito.

L'**editor delle classi anonime** (`OWLExpressionEditor`) è un componente speciale che consente di creare e modificare le espressioni che definiscono concetti arbitrari (unione o intersezione di classi, restrizioni di cardinalità su un ruolo, enumerazioni, ecc...).

OWLIE utilizzando le API di Protégé ne eredita lo stile con cui vengono rappresentate le classi anonime, ossia mediante un'espressione in *sintassi Manchester*[4]. Tale sintassi è comoda e rapida ma soltanto in presenza di utenti che già la conoscono bene. Per questa ragione l'editor delle classi anonime implementato in OWLIE pur utilizzando la sintassi Manchester come metodo di rappresentazione dei concetti arbitrari, assiste l'utente nella formulazione delle definizioni per mezzo di un editor grafico, rappresentato in figura 4.7.

Per gli utenti esperti è data la possibilità di modificare direttamente l'espressione in sintassi Manchester, la cui validità è controllata dal medesimo parser utilizzato da Protégé.

I due **editor delle proprietà** (`DatatypePropertyEditor` e `ObjectPropertyEditor`) sono due estensioni del componente `Window` di ZK. La loro struttura è assai simile a quella dell'editor delle classi, ossia a riquadri: nel riquadro superiore è contenuta la lista delle classi (anonime e non) la cui unione costituisce il dominio della proprietà, mentre nel riquadro intermedio è contenuta la definizione del codominio (o *range*) del ruolo, che è una lista di classi nel caso delle *Object property*, e un elenco di letterali ammissibili nel caso di *Datatype property*.

Nella finestra sono indicati e possono essere modificati, inoltre, gli attributi del ruolo (transitività, simmetricità, funzionalità) e nel caso delle *Object property* può essere impostata o rimossa la proprietà inversa, ammessa in OWL. Un'istanza dell'editor delle Object property è raffigurata all'interno dell'interfaccia del modulo funzionale della Property View in figura 4.5.

Infine, l'ultimo componente di interesse è l'**editor degli individui** (`IndividualsEditor`) anch'esso estensione della classe `Window` e che condivide

[4] `http://www.co-ode.org/resources/reference/manchester_syntax/`

con gli altri componenti esaminati sinora l'impianto concettuale e il paradigma grafico, come ritratto in figura 4.6.

4.1.5 L'editor grafico della T-BOX

L'editor grafico della T-BOX fornisce all'utente un sistema di ausilio visivo per la modifica della parte terminologica dell'ontologia, vale a dire le definizioni della gerarchia dei concetti e del dominio e codominio dei ruoli.

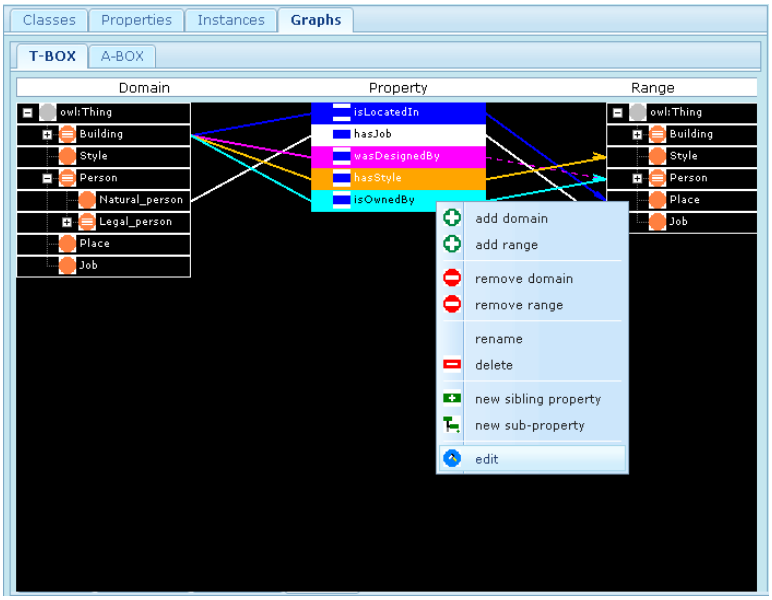


Figura 4.8: *Editor grafico della T-BOX*

La realizzazione di un editor grafico si rende utile dal momento in cui non esiste nell'applicazione una visione d'insieme della T-BOX che comprenda, nella medesima vista, sia le classi sia le proprietà.

L'editor è implementato da un componente riutilizzabile, `TBOXGraph` che è estensione del componente standard ZK `Window`.

Al suo interno, il componente implementa sia l'interfaccia utente, sia le routine di gestione degli eventi.

L'interfaccia è costituita da un *layout* a tre alberi, come si può vedere in figura 4.8. L'albero centrale rappresenta la gerarchia dei ruoli, mentre i due alberi laterali rappresentano entrambi la gerarchia dei concetti. Gli elementi degli alberi delle classi sono messi in relazione molti-molti con i nodi dell'albero delle proprietà. Gli archi orientati hanno il significato, rispettivamente, di *fa parte del dominio di...* e *ha come codominio...*.

Sono esclusi, nella versione attuale del componente, i concetti non atomici (classi anonime) che tuttavia possono essere agevolmente inclusi. La scelta di rappresentare soltanto i concetti atomici ovvero le classi con nome deriva

dall'esigenza di realizzare un editor semplice e graficamente ordinato. Se fosse stata fatta la scelta di rappresentare tutti i tipi di dominio, anche quelli non composti semplicemente da un'unione di concetti atomici, sarebbe stato necessario introdurre costrutti grafici molto più complessi, che avrebbero appesantito l'interfaccia e reso molto più difficile all'operatore l'utilizzo dello strumento.

L'interfaccia del componente è realizzata sfruttando per quanto possibile le librerie ZK, quindi componenti web, senza ricorrere a software esterno come invece è stato fatto nel caso del visualizzatore di A-BOX. Ciò ha consentito di realizzare un'interfaccia compatta e integrata con il sistema. Gli elementi grafici non rappresentabili mediante componenti di ZK sono stati ottenuti utilizzando la classe `Graphics2D` e i metodi di disegno contenuti nel package `java.awt`.

Oltre alla visualizzazione delle triple dominio-proprietà-codominio, l'utente ha la possibilità di alterare la gerarchia delle classi (mediante un menu contestuale accessibile con il tasto destro del mouse) e la gerarchia delle proprietà, nonché modificarne i membri sfruttando la specifica opzione del menu contestuale, come esemplificato in figura 4.8. Inoltre è possibile modificare le relazioni di dominio e codominio. In sintonia con i principi di riutilizzo del software, per la modifica delle risorse sono utilizzati gli stessi componenti condivisi esaminati nel paragrafo 4.1.4.

4.1.6 Il visualizzatore della A-BOX

Il visualizzatore della A-BOX è un componente speciale, implementato dalla classe `ABOXGraph` che è estensione di `Window`.

Il suo comportamento e le sue funzionalità sono notevolmente differenti rispetto all'editor della T-BOX. Premesso che, nella maggior parte dei casi, la parte terminologica delle ontologie è di dimensioni molto minori rispetto alla sua applicazione (ovvero la A-BOX). Inoltre per sua natura la A-BOX può contenere un numero di relazioni tale da non poter essere rappresentato in un layout semplice, come quello utilizzato nel grafo della T-BOX.

Il componente `ABOXGraph` implementa al suo interno l'interfaccia grafica che consiste in un'area di selezione e un'area di rappresentazione. L'area di selezione contiene un selettore delle classi, un selettore per le proprietà, e un selettore per gli individui.

L'area di rappresentazione è costituita da un *iFrame* inizialmente vuoto e nel quale viene visualizzato il grafico a seguito del comando dell'utente. In figura 4.9 è rappresentato un esempio di utilizzo. L'uso di un iFrame (che è un elemento HTML e non di ZUML) è necessario per non stravolgere l'interfaccia dell'intera applicazione: i grafici delle A-BOX possono diventare estremamente voluminosi se l'utente sceglie di visualizzare tutti o comunque gran parte degli individui dell'ontologia, e siccome il grafico è in sostanza un'immagine *Portable Network Graphics* (PNG) prodotta da un software esterno, come spiegato in seguito, OWLIE non ha il controllo sulle dimensioni che questa possa avere.

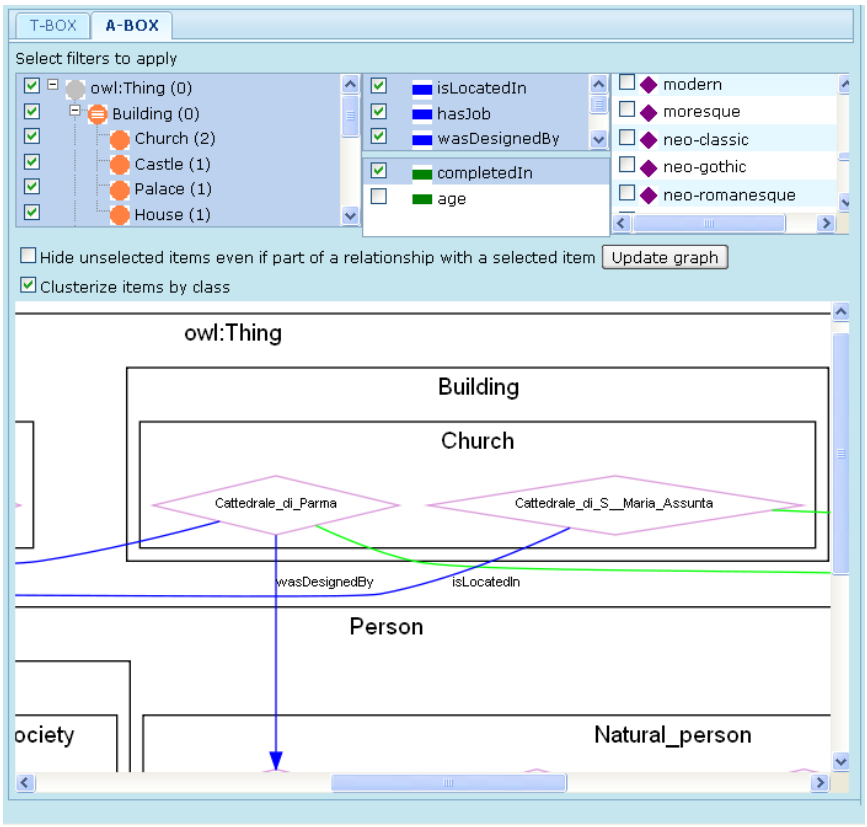


Figura 4.9: *Visualizzatore grafico della A-BOX*

Il grafo vero e proprio è ottenuto in quattro fasi:

1. in base alla selezione dell'utente vengono estratte dall'ontologia le risorse da rappresentare, costruendo un insieme (set) di asserzioni, rappresentate dalla classe `Assertion` (una delle classi ausiliarie di OWLIE implementate ad hoc per risolvere un particolare problema, come già accennato nel paragrafo 3.3.3).
   Un'asserzione è una tripla individuo-proprietà-valore.

2. il set di asserzioni viene passato come parametro a un metodo statico della classe `DOTUtilities` (altra classe ausiliaria) che produce un sorgente in linguaggio DOT e lo restituisce sottoforma di array di istruzioni.

3. un altro metodo di `DOTUtilities` riceve come parametro l'array di istruzioni in linguaggio DOT e le salva in un file temporaneo. Tale file è passato come argomento a un'applicazione esterna ad OWLIE (che pertanto deve essere correttamente installata sulla stessa macchina in cui è configurato il web server di OWLIE) che fa parte del pacchetto *GraphViz*.

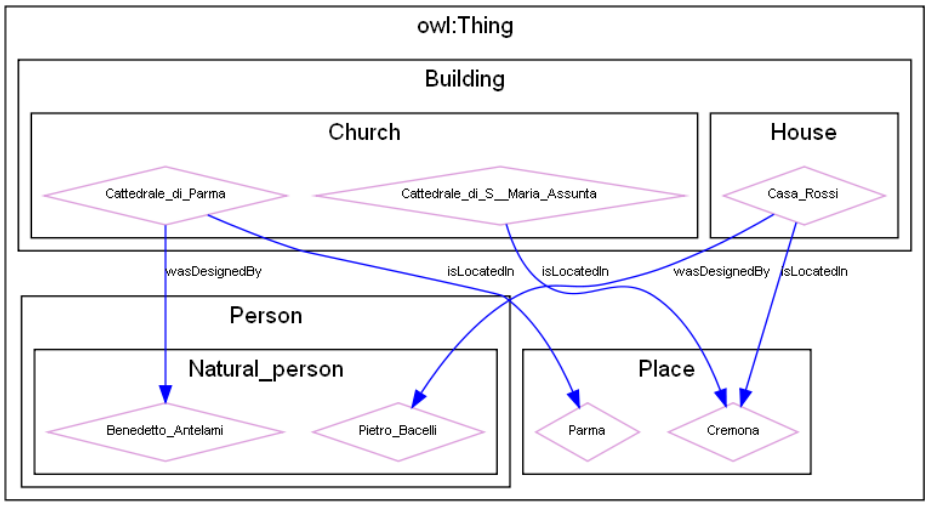


Figura 4.10: *Un grafico della A-BOX*

4. il programma esterno compila il codice fornitogli e produce un output costituito da un'immagine PNG. Tale immagine, memorizzata in una directory temporanea all'interno del web server, è importata all'interno del componente e posizionata all'interno dell'*iFrame* creato a tale scopo. Il file contenente il codice viene cancellato, mentre l'immagine generata permane nella directory temporanea (che viene periodicamente ripulita secondo una strategia FIFO).

Il grafo risultante è costituito da parallelogrammi dal contorno viola pallido che rappresentano gli individui, racchiusi all'interno di rettangoli neri che rappresentano le classi di appartenenza (se l'utente ha selezionato tale opzione). Ogni rettangolo è a sua volta contenuto in un rettangolo più grande, coerentemente con la gerarchia delle classi (ottenendo in questo modo una vista gerarchica a *treemap*, descritta nel paragrafo 2.2.1).

I ruoli sono costituiti da archi orientati di colore blu (per le proprietà sugli oggetti) o verde (proprietà sui dati).

A differenza dell'editor della T-BOX, il grafo della A-BOX non implementa funzionalità interattive. Tuttavia è auspicabile un suo potenziamento in una fase di futuro aggiornamento dell'applicazione.

In figura 4.10 è rappresentato un grafo della A-BOX nella sua interezza.

4.2 Il sottosistema collaborativo

Il sottosistema collaborativo di OWLIE consente agli utenti di conoscere quali altre persone sono connesse nel medesimo istante e scambiare informazioni

con loro in tempo reale, nonché permette a un utente che sta operando su una particolare ontologia di conoscere le operazioni eseguite da altri utenti sulla stessa ontologia.

Il sistema è stato progettato per essere indipendente rispetto al resto dell'applicazione: a differenza di quanto avviene in altri software preesistenti, l'interfaccia dei moduli e dei componenti che implementano funzionalità collaborative è separata rispetto all'interfaccia dei moduli e dei componenti che consentono di operare sull'ontologia.

In OWLIE sparisce pertanto il supporto per le annotazioni sulle risorse dell'ontologia, e il supporto alla cooperatività degli utenti è maggiormente orientato al real-time e allo scambio diretto di informazioni mediante chat, anche se nulla toglie che, nel corso di futuri ampliamenti del software, possano essere introdotte anche funzionalità di annotazione.

4.2.1 Schema generale

In generale il principio di funzionamento del sottosistema collaborativo è piuttosto semplice ed è incardinato su pochi elementi dalla funzione ben precisa.

Quello che il sistema deve fare è in sostanza mostrare all'utente, tramite un componente dedicato, quali altri utenti sono attivi nel momento considerato, e qualora stiano lavorando alla medesima ontologia condivisa, permettere di comunicare direttamente con loro e visualizzare le modifiche da essi effettuate in tempo reale.

Il sottosistema si compone di elementi puramente visivi (i pannelli della chat e del monitor degli utenti attivi) e di elementi di backend, il cui scopo è di ricevere segnali di notifica da parte di OWLIE ogni qualvolta si verifica un evento da pubblicare (login di un utente, apertura o chiusura di un progetto, evento di modifica di un'ontologia), e propagarlo a tutti gli altri utenti (previo opportuno filtraggio).

Fulcro del sottosistema è il ***Collaborative Manager***, un oggetto che tiene traccia di tutti gli utenti connessi (ossia tutti gli utenti che abbiano effettuato con successo il login ad OWLIE) e che accetta notifiche da parte dei moduli e dai componenti del sistema centrale di OWLIE. Tale oggetto deve essere l'unica istanza della propria classe presente in memoria, in modo tale per cui risulta essere accessibile staticamente da qualsiasi altro oggetto all'interno dell'applicazione web, per tutte le sessioni utente senza distinzioni, e senza perdere il proprio valore fintantoché il server web resta attivo. La classe che definisce il Collaborative Manager deve pertanto essere implementata secondo il design-pattern *singleton*.[5]

Il Collaborative Manager riceve da OWLIE (dal modulo principale, piuttosto che da uno qualsiasi dei suoi moduli funzionali oppure da uno dei componen-

[5] il design-pattern creazionale *singleton* garantisce che una data classe possa ammettere una ed una sola istanza nell'ambito della medesima macchina virtuale Java.

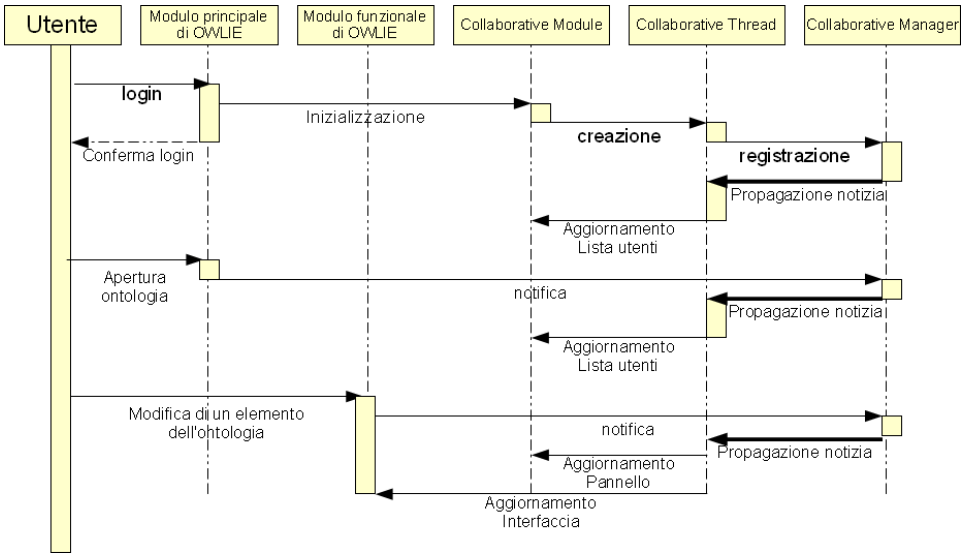


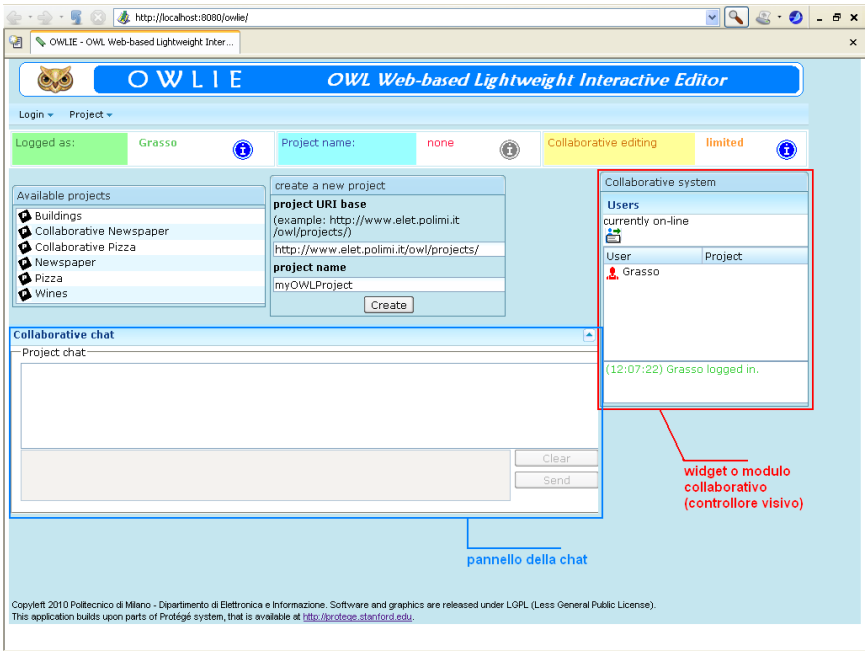
Figura 4.11: *Diagramma sequenziale del sottosistema collaborativo di OWLIE*

ti condivisi) delle *notifiche* in occasione di eventi che potenzialmente possono interessare più utenti. Questi eventi sono:

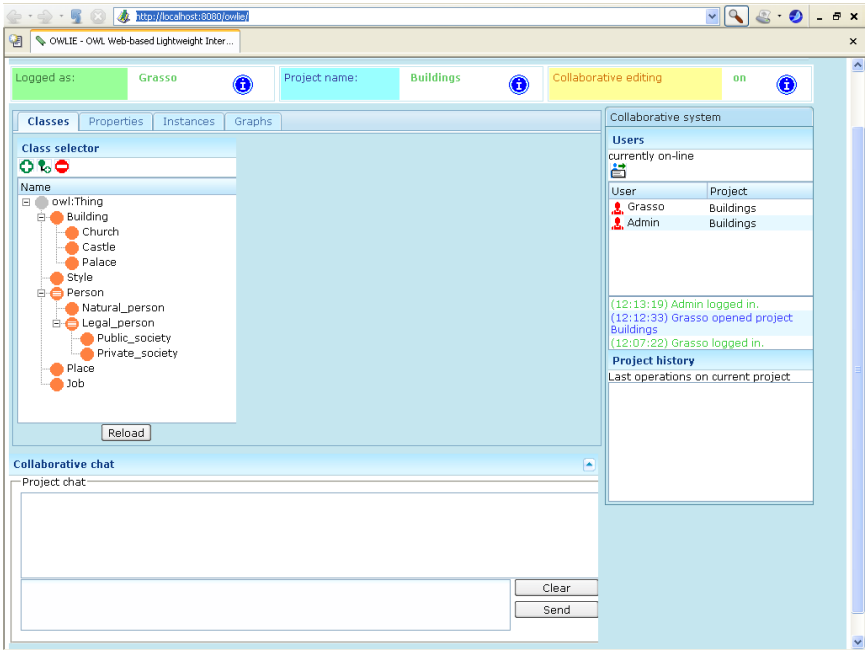
- login effettuato con successo (un nuovo utente deve essere registrato tra gli utenti attivi)
- logout (l'utente deve essere rimosso dall'elenco degli utenti attivi)
- apertura / chiusura di un'ontologia
- modifica di un elemento dell'ontologia (la modifica deve essere propagata a tutti gli altri utenti che stanno operando sulla stessa ontologia, poiché occorre aggiornarne l'interfaccia grafica)
- invio di un messaggio in chat

Il Collaborative Manager tuttavia non può effettuare la propagazione delle informazioni in modo diretto, agendo sulle interfacce grafiche relative alle varie sessioni attive, poiché l'architettura AJAX di norma non consente che una classe lato server possa inviare dati al browser web del client se non in risposta a una richiesta HTTP[16].

Questo ostacolo architetturale è superato mediante una tecnica ben descritta all'interno della documentazione di ZK, chiamata *Server push*[16] la quale prescrive l'utilizzo di un *thread* per modificare in modo asincrono (vale a dire anche non in risposta a un evento-utente) elementi dell'interfaccia grafica lato client.



(a) widget collaborativo e pannello della chat



(b) esempio di concorrenza degli accessi

Figura 4.12: *Attivazione delle funzionalità collaborative di OWLIE*

In virtù di questa possibilità, viene istanziato un thread (chiamato ***Collaborative Thread***) che costituisce un tramite tra i moduli e i componenti dell'applicazione e il Collaborative Manager. Esiste un thread per ogni utente connesso, che viene istanziato in seguito all'atto di Login, e disattivato in seguito all'atto di logout. Il thread provvede a "registrarsi" sul Collaborative Manager, che tiene quindi in memoria la lista completa di tutti i thread attivi.

Nel momento in cui il Collaborative Manager riceve una notifica da parte di un modulo, la propaga a tutti i Collaborative Thread relativi ad utenti ai quali l'argomento della notifica risulta di interesse (la propagazione non è in broadcast, per evidenti ragioni di risparmio di risorse). Ognuno dei thread contattati provvederà ad aggiornare, in piena autonomia, le parti di interfaccia utente interessate dalla notifica. Ciò non provoca alcun conflitto con l'utente poiché in tale (breve) lasso di tempo l'interfaccia diviene controllata dal thread e non sono attive le routine di gestione degli eventi definite nei composer e nei componenti.

Il meccanismo descritto trova una sua rappresentazione grafica nel diagramma sequenziale riportato in figura 4.11.

Caratteristica importante che emerge dal discorso sinora fatto è che il sottosistema collaborativo di OWLIE è completamente indipendente dall'ontologia, a differenza dei sistemi basati su ChAO. Tra i vantaggi di questa impostazione vi è la possibilità di concentrare tutto il sottosistema entro il livello 1 dell'architettura client-server, ossia senza dover interpellare il server di Protégé che quindi si occupa solo ed esclusivamente di gestire la concorrenza. D'altro canto, lo svantaggio più evidente è il rischio che si possano creare inconsistenze tra quanto memorizzato nel CollaborativeManager e le sessioni remote gestite dal server di Protégé: tale rischio si concretizza nei casi in cui l'utente non esegua correttamente il logout. Appare quindi di notevole interesse un perfezionamento del sistema anche a livello architetturale.

4.2.2 Componenti visuali: il widget

Il *widget* collaborativo è un componente visuale (ma non solo) che consente a ogni utente lato client di conoscere chi sono gli altri utenti e cosa stanno facendo. Il widget è definito dalla classe `CollaborativeModule` (estensione del componente predefinito `Window`).

Il widget è inizializzato e aggiunto all'interfaccia utente a seguito di un'operazione di login conseguita con successo (in caso contrario viene visualizzato un avviso all'utente che lo informa dell'impossibilità di avvio del sistema collaborativo), e come descritto nel diagramma sequenziale in figura 4.11, tale operazione è svolta dall'`Initializer` ossia il composer correlato al modulo principale.

All'atto della creazione dell'oggetto, viene effettuata la creazione del thread collaborativo relativo all'utente. Allo stesso modo, il thread viene disattivato (non distrutto) nel momento in cui, a seguito per esempio dell'operazione di logout, il widget viene staccato dalla pagina e orfanizzato. Il ciclo di vita del thread e quello del widget sono pertanto legati tra loro.

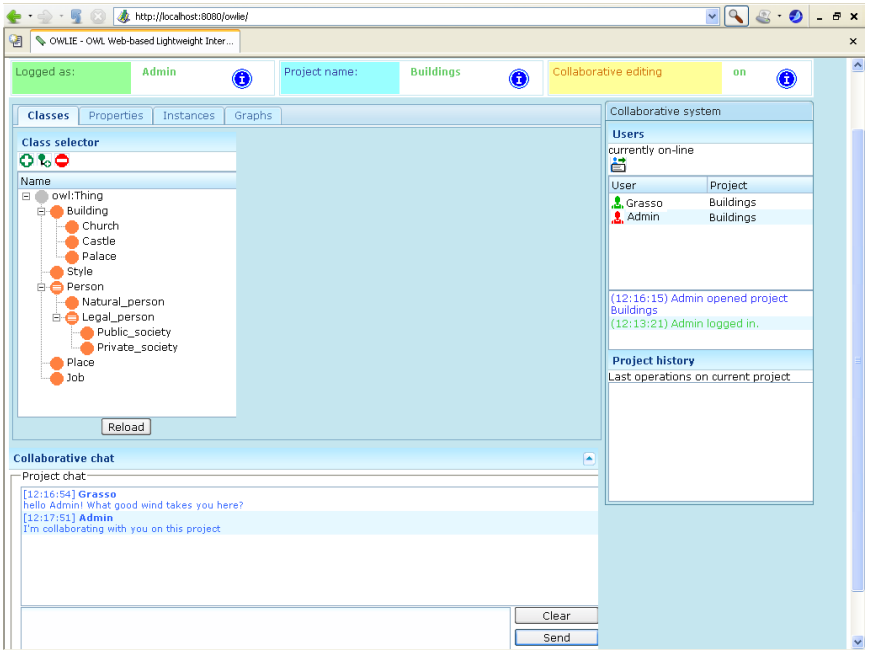


Figura 4.13: *Esempio di utilizzo della chat per scambiare messaggi*

Uno dei compiti del thread è di aggiornare, ma solo quando necessario, alcune parti del widget ovvero:

- la lista degli utenti connessi (a seguito di operazioni di login, logout)
- lo stato degli elementi della lista utenti connessi (a seguito di operazioni di apertura, chiusura di ontologie)
- il box elencante gli ultimi "avvenimenti" da segnalare agli utenti (gli stessi eventi citati poc'anzi)
- il notificatore delle modifiche avvenute all'ontologia, in ordine cronologico

In figura 4.12 è riportato l'aspetto del widget in un caso di accesso concorrente di due utenti.

4.2.3 La chat

La possibilità di interagire in tempo reale con gli altri utenti che stanno operando sulla medesima ontologia è uno dei punti cardine di ogni sistema collaborativo. Alcuni, come WebProtege (vedasi paragrafo 2.4.4) utilizzano il sistema delle annotazioni combinato con un sistema di scambio di messaggi (in perfetto stile forum). In OWLIE la metodologia adottata è diversa: si dà la possibilità agli utenti connessi contemporaneamente di comunicare in modo immediato tramite una chat integrata, gestita dall'apposito componente caricato con-

seguentemente all'apertura di un'ontologia condivisa sul server e che utilizza, per il recapito dei messaggi, lo stesso thread utilizzato per le notifiche.

La versione attuale della chat è molto rudimentale, non permettendo altre funzioni che non siano il recapito di messaggi in broadcast verso tutti e soli gli utenti che, nel momento dell'invio del messaggio, sono al lavoro sulla medesima ontologia, come esemplificato in modo sintetico in figura 4.13.

4.3 Distribuzione del software

4.3.1 I package

Classi e Interfacce di OWLIE costituiscono la parte compilata dell'applicazione. Come ogni applicazione Java, ogni classe o interfaccia di OWLIE deve essere contenuta in un *package* per facilitarne la localizzazione e migliorarne l'organizzazione interna.

In ottemperanza alle convenzioni dell'ingegneria del software[17], i nomi dei package di OWLIE hanno una radice comune ossia

`it.polimi.elet.owlie`

.

La struttura dei package di OWLIE è organizzata nel modo seguente:
`it.polimi.elet.owlie`
Package radice, non contiene classi né interfacce

- `it.polimi.elet.owlie.backend`
  contiene le classi che costituiscono il *backend* dell'applicazione (manager di sessione, manager delle ontologie)

- `it.polimi.elet.owlie.collaborative`
  contiene le classi che implementano le funzionalità collaborative a livello di backend (non a livello di interfaccia)

- `it.polimi.elet.owlie.owl.utils`
  contiene classi ed interfacce accessorie all'applicazione, utilizzate da altre classi

- `it.polimi.elet.owlie.ui`
  comprende i sotto-package a loro volta contenenti classi ed interfacce costituenti l'interfaccia utente dell'applicazione.

  - `it.polimi.elet.owlie.ui.components`
    Contiene i *componenti condivisi* dell'interfaccia grafica dell'applicazione (e relative interfacce).

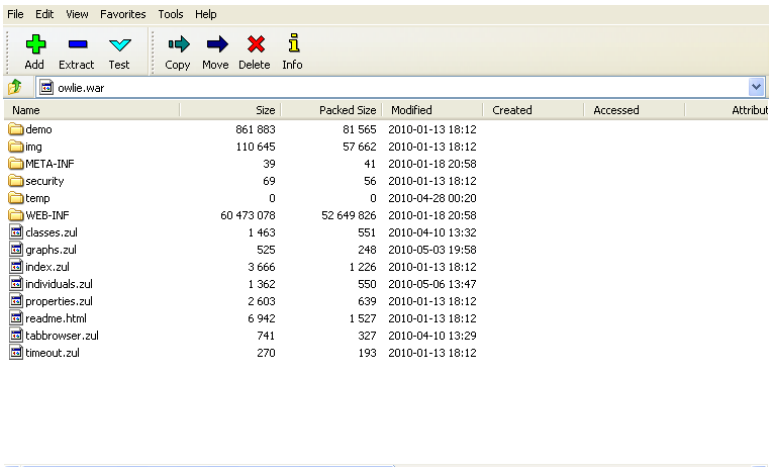


Figura 4.14: *Contenuto di owlie.war*

    * `it.polimi.elet.owlie.ui.components.collaborative`
      contiene i componenti dell'interfaccia grafica che implementano funzionalità collaborative (chat e pannello di controllo collaborativo)
    * `it.polimi.elet.owlie.ui.components.extensions`
      contiene alcuni componenti derivati da analoghi elementi ZUML a cui sono aggiunte funzionalità particolari.
    * `it.polimi.elet.owlie.ui.components.graph`
      contiene le classi che definiscono il visualizzatore della A-BOX e l'editor grafico della T-BOX.

- `it.polimi.elet.owlie.ui.composers`
  Contiene i *composer* associati ai moduli ZUML (principale e periferici).
- `it.polimi.elet.owlie.ui.utils`
  Contiene classi accessorie ad uso dei componenti condivisi.

4.3.2 Struttura dell'archivio web

OWLIE è distribuita sotto forma di archivio web (*web archive*, WAR) che in sostanza è una directory compressa, analoga in quanto a formato a un archivio JAR, con la differenza che normalmente quest'ultimo tipo di archivio contiene un'applicazione Java destinata ad utilizzo desktop, oppure una raccolta di classi (organizzate o meno in packages) ad uso di altre applicazioni (ossia una libreria).

All'interno dell'archivio (denominato `owlie.war`) trova posto l'intera applicazione web e la sua struttura interna di directory, come ritratto in figura 4.14.

Nella directory principale (detta anche directory radice, o equivalentemente *root directory* in lingua inglese) trovano posto i file `.zul` corrispondenti ai moduli ZUML dell'applicazione e a pagine accessorie, come in tabella 4.1.

La directory `WEB-INF` è il cuore dell'applicazione. Contiene infatti i *descrittori* dell'applicazione (`web.xml` e `zk.xml`), automaticamente generati dall'ambiente di sviluppo.

La sotto-directory `lib` contiene le librerie necessarie per l'esecuzione di OWLIE, sotto forma di archivi JAR. Tali librerie costituiscono una parte considerevole della dimensione complessiva dell'archivio, e ciò è dovuto al fatto che i JAR che le contengono sono stati inclusi in toto, senza averne estratto solamente le classi che effettivamente vengono richiamate durante l'esecuzione di OWLIE.

L'altra sotto-directory, `classes`, contiene le classi e le interfacce, organizzate in *packages* come descritto nel paragrafo 4.3.1, che costituiscono la parte compilata di OWLIE (mentre i file `.zul` contenuti nella directory radice ne costituiscono la parte interpretata).

file	descrizione
`classes.zul`	modulo periferico che implementa la *Class View*
`graphs.zul`	modulo periferico che implementa l'area dedicata ai visualizzatori grafici
`index.zul`	modulo principale, caricato ad ogni accesso da parte dell'utente; definisce la cornice e include i moduli periferici
`individuals.zul`	modulo periferico che implementa l'*Individuals View*
`properties.zul`	modulo periferico che implementa la *Property View*
`readme.html`	istruzioni rapide per gli sviluppatori che desiderano compilare i sorgenti di OWLIE in Eclipse
`tabbrowser.zul`	estensione del modulo principale: implementa il selettore a linguette presente nell'interfaccia dell'applicazione a login effettuato
`timeout.zul`	pagina accessoria che viene mostrata all'utente nel caso di inattività protrattasi oltre il termine massimo consentito dal server (timeout) comportando l'invalidità della sessione di lavoro
directory	**contenuto**
`demo`	contiene le ontologie di esempio da utilizzare in modalità *demo*
`img`	contiene icone, loghi e immagini in uso in OWLIE
`META-INF`	directory creata dall'ambiente di sviluppo e prevista dall'architettura del file `.war`. In questo caso non contiene informazioni.
`security`	contiene il file `security.policy` che descrive le politiche di sicurezza di OWLIE (indispensabile per l'utilizzo di RMI)
`temp`	directory temporanea ad uso dell'applicazione. Inizialmente è vuota.
`WEB-INF`	contiene le classi di OWLIE e le librerie esterne incluse nell'applicazione

Tabella 4.1: *Contenuto della root directory di owlie.war*

Capitolo 5

Conclusioni e sviluppi futuri

> *Non trovare difetti, trova rimedi: a lamentarsi sono capaci tutti.*
> (Henry Ford)

Con questa citazione di Henry Ford[1] si apre l'ultimo capitolo della trattazione, dedicato a raccogliere i giudizi del progettista a implementazione ultimata e gli auspici per futuri ampliamenti, o aggiornamenti del software realizzato.

Il processo di sviluppo e mantenimento di OWLIE non si esaurisce, infatti, con l'implementazione e la consegna della prima versione funzionante dell'applicazione e della relativa documentazione: ciò costituisce, al contrario, un punto di partenza più che un punto di arrivo. Il rilascio e i primi utilizzi sul campo possono far nascere l'esigenza di ampliamenti, adattamenti, introduzione di nuove funzionalità al software in modo che risponda in modo più esteso agli obiettivi prefissati, o, nel caso, ad altri obiettivi inizialmente non considerati.

Le osservazioni e le conclusioni tratte in questo capitolo, assieme agli spunti per future versioni del software, pur nello sforzo di essere il più possibile obiettive, rappresentano un giudizio *soggettivo* proveniente dalla stessa persona che ha progettato l'applicazione e ne ha redatto la documentazione.

5.1 Analisi del raggiungimento degli obiettivi

Nei capitoli precedenti sono state esaminate le molteplici scelte tecniche, procedurali e concettuali effettuate durante le varie fasi del processo di sviluppo. Lo scopo di questo passaggio è verificare se e in che modo tali scelte hanno influenzato la capacità dell'applicazione di risolvere i problemi per i quali è stata concepita, ovvero in altri termini determinare il livello di raggiungimento dei vari obiettivi alla base del progetto.

[1] http://it.wikiquote.org/wiki/Henry_Ford

Gli obiettivi principali che si intende raggiungere, come riportato nel paragrafo 1.4, sono espressi all'interno dell'acronimo che denota l'applicazione stessa: *Ontology (od OWL) Web-based Lightweight Interactive Editor*, vale a dire:

- un'applicazione basata sul web e pertanto accessibile via browser;
- un editor di ontologie in OWL (che abbia pertanto una capacità espressiva non inferiore a quella di OWL-DL);
- un'applicazione leggera, intendendo con tale aggettivo sia un ridotto utilizzo di risorse, sia un approccio leggero, da parte dell'utente, in termini di sforzo per l'apprendimento all'uso e di complessità di utilizzo;
- un editor interattivo, anche in questo caso con il duplice significato di capacità di interazione tra uomo e software e di scambio di informazioni tra più utenti.

L'architettura client-server a più livelli scelta per supportare l'applicazione, sfruttando in tal modo il paradigma "thin-client", consente di accedere al software tramite un comune browser, a condizione ovviamente che l'amministratore abbia provveduto ad una corretta installazione e ad una corretta configurazione. Il primo obiettivo risulta quindi raggiunto e non è necessaria un'ulteriore analisi in merito.

Le valutazioni sul raggiungimento degli altri obiettivi richiedono tuttavia un maggior dettaglio.

5.1.1 La potenza espressiva di OWLIE

OWL, o per meglio dire la sua variante OWL-DL la quale garantisce la completezza del modello logico senza sacrificare la decidibilità[2], è un linguaggio dalla medesima potenza espressiva delle logiche `SHOIN(D)`.

OWLIE è in grado di rappresentare una parte assai consistente dei costrutti di OWL-DL, con alcune eccezioni.

Per quanto riguarda i *concetti* (o *classi* secondo la terminologia ereditata da RDF-S), OWLIE è in grado di rappresentare la gerarchia dei concetti atomici, cioè quei concetti dotati di un nome, di cui mostra e consente di modificare la lista delle sussunzioni e delle equivalenze, nonché la lista dei concetti con i quali si trova in disgiunzione.

I concetti anonimi sono mostrati evidenziando gli elementi che li ed è possibile modificarli. Tutti i costrutti in grado di produrre concetti anonimi sono presenti ed utilizzati.

[2] capacità di dedurre, tramite un algoritmo che fornisca una risposta in un intervallo di tempo finito, se una formula è o non è un teorema della logica, ovvero se la sua aggiunta all'insieme dei teoremi crea o meno una contraddizione

costrutto SHOIN(D)	**rappresentato in OWLIE**
concetti atomici	SI
concetti anonimi	SI
disgiunzione di concetti	SI (solo concetti atomici)
sussunzione di concetti	SI (gerarchia di classi)
doppia sussunzione (equivalenza) di concetti	SI (definizione di classi)
ruoli	SI (*object properties*)
sussunzione tra ruoli	SI (gerarchia di proprietà)
combinazione di ruoli	NO
vincoli di simmetria, funzionalità, transitività e funzionalità inversa	SI
ruolo inverso	SI
combinazione di ruoli	NO
restrizioni di cardinalità sui ruoli	SI
attributi degli individui	SI (*datatype properties*)
individui	SI
vincoli di disgiunzione o uguaglianza tra individui	NO

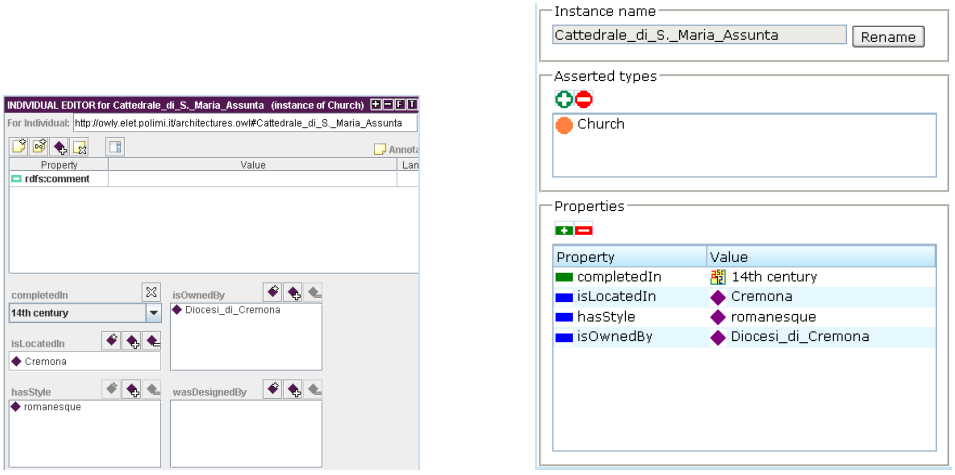
Tabella 5.1: *Confronto tra l'espressività di OWLIE e le logiche SHOIN(D)*

In merito ai *ruoli* (o *proprietà*) si evidenzia una prima riduzione di espressività, frutto di una scelta tecnica più che di una semplificazione concettuale (come descritto nel paragrafo 3.3.4). Dominio e codominio sono infatti espressi con semantica disgiuntiva, mentre OWL utilizza anche in questo caso la semantica congiuntiva. Ciò significa che i domini e i codomini in OWLIE sono *sempre* intesi come un'unico concetto anonimo che rappresenta l'unione delle classi mostrate e inserite nel dominio o codominio, esattamente come in Protégé.

Questa scelta espressiva non preclude la possibilità che un ruolo possa avere, a dominio, più concetti in semantica congiuntiva (quindi in intersezione tra di essi), ma è necessario esplicitarlo mediante la costruzione di un concetto anonimo.

I problemi si presentano quando l'ontologia viene creata in un altro editor e poi condivisa tramite il server di Protégé il quale a sua volta supporta OWLIE. Se in tale ontologia esiste un ruolo in cui il dominio – o il codominio – contiene più di un concetto, in semantica congiuntiva quindi in intersezione, OWLIE (che utilizza le API di Protégé-OWL per accedere alle risorse della base di conoscenza) ne mostra soltanto una, la prima in ordine di serializzazione.

Gli *individui* sono rappresentati in ogni caratteristica, ad esclusione dell'operatore `AllDifferent` che, applicato ad una lista di individui, specifica il vincolo della non coincidenza. Questo operatore potrebbe essere facilmente



(a) in Protégé 3.4.2 (b) in OWLIE

Figura 5.1: *Confronto degli editor degli individui in Protégé e in OWLIE*

inserito in una prossima fase di aggiornamento dell'applicazione.

In generale si può quindi affermare che OWLIE si avvicini molto in termini di potenza espressiva a quella offerta dal linguaggio OWL, pur non raggiungendola.

5.1.2 La semplificazione dell'interfaccia: esempi pratici

L'interfaccia utente di OWLIE si presenta molto più semplice, essenziale, rispetto ad altri editor di ontologie (specialmente quelli non basati sul web, come Protégé).

Se da un lato, infatti, il maggior numero di funzionalità disponibili rende l'applicazione più completa, il prezzo da pagare è la maggior difficoltà da parte dell'utente a prendere confidenza con l'ambiente operativo, in modo particolare se, come già sottolineato nel paragrafo 3.1.1, egli ha poca dimestichezza o competenze poco approfondite riguardo OWL e in generale il web semantico.

Premesso che l'interfaccia grafica di OWLIE prende spunto, a grandi linee, da quella di Protégé 3.4 (un'applicazione desktop, non basata sul web), alcuni esempi pratici aiutano a comprendere in che modo la semplificazione dell'interfaccia utente è stata operata.

In figura 5.1 sono riportati due immagini riportanti l'editor degli individui, in Protégé 3.4.2 e in OWLIE, applicati al medesimo individuo all'interno della stessa ontologia[3].

Rispetto all'omologo componente visuale di Protégé, OWLIE nasconde i nomi "reali" delle risorse descritte (gli URI) al fine di non confondere l'utente.

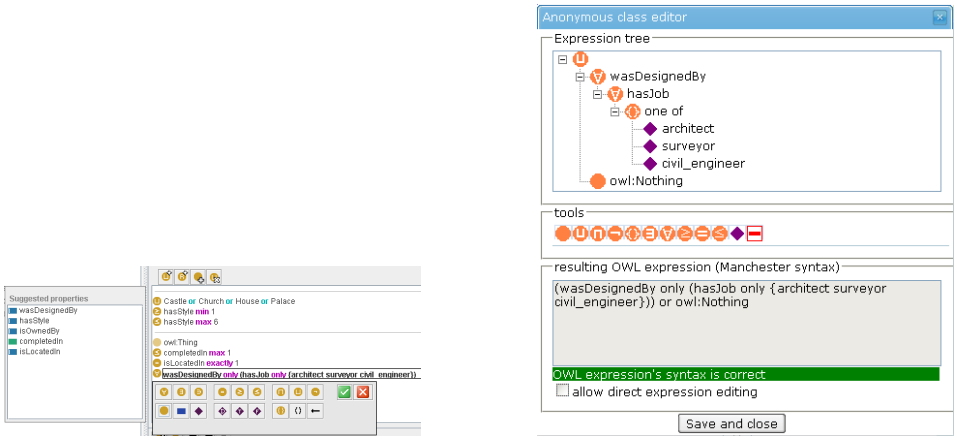
[3] `Buildings.owl`, ontologia che descrive alcuni edifici e le loro proprietà

Ogni risorsa è infatti specificata, a livello visuale, dal proprio identificatore che non è altro che l'ultima parte dell'URI. Ogni identificatore è univoco all'interno dello stesso spazio dei nomi, pertanto non vi sono possibilità di confusione considerato che OWLIE non permette di definire namespaces secondari all'interno della stessa ontologia (operazione che Protégé al contrario consente).

La scelta di nascondere gli URI e mostrare solo l'identificatore rappresenta un vantaggio in termini di facilità d'uso per gli utenti meno esperti, ma comporta un prezzo da pagare, quantificabile in perdita di completezza dell'applicazione poiché non è possibile, in OWLIE, cambiare l'URI dello spazio dei nomi principale.

In OWLIE l'operazione di ridenominazione di una risorsa (in figura 5.1(b) è riportato l'editor degli individui ma ciò si applica anche a classi e proprietà) è assistita dal sistema tramite l'apposita funzione (nell'esempio raffigurato, il pulsante "rename"), al contrario di Protégé ove per ridenominare una risorsa è modificare manualmente l'URI della stessa oppure utilizzare la funzione di *refactoring* tramite menu contestuale.

Una differenza importante tra l'interfaccia di Protégé e quella di OWLIE è anche il diverso metodo di inserimento e modifica dei concetti anonimi, le cosiddette *anonymous classes*, rappresentato in figura 5.2.



(a) in Protégé 3.4.2 (b) in OWLIE

Figura 5.2: *Confronto degli editor dei concetti anonimi in Protégé e in OWLIE*

In Protégé la modalità di definizione e modifica di classi anonime è puramente testuale, seppur assistita da una serie di strumenti, un evidenziatore di sintassi e da un validatore in tempo reale (che accetta solo ed esclusivamente espressioni ben formate). La sintassi utilizzata è la *Manchester syntax*, uno dei possibili modi di scrivere le formule di OWL. Tale sintassi per semplice che sia potrebbe essere completamente sconosciuta a un utente di poca esperienza, rendendo l'editor dei concetti anonimi di Protégé piuttosto grezzo.

OWLIE eredita da Protégé la sintassi Manchester come modo per conferire ai concetti anonimi una rappresentazione *human-readable* che altrimenti non avrebbero dal momento che non sono identificabili da un nome come qualsiasi altra risorsa. A differenza di Protégé, tuttavia, OWLIE implementa un grafico in cui l'utente meno esperto risulta facilitato nella formulazione del concetto.

5.1.3 L'interazione multi-utente

Il sottosistema collaborativo implementato in OWLIE consente a più utenti di accedere alla medesima ontologia e di apportarvi modifiche in modo simultaneo, venendo notificati in tempo reale (o quasi) di ciò che gli altri utenti fanno e di quali modifiche hanno apportato al progetto sul quale si sta lavorando. La chat permette di scambiare messaggi in broadcast a tutti gli altri utenti che stanno lavorando sulla stessa ontologia.

Tuttavia si può ragionevolmente ritenere che le possibilità di interazione multi-utente non si debbano esaurire alle funzionalità sopraccitate: esistono numerosi aspetti del sistema collaborativo di OWLIE che debbono necessariamente essere potenziati.

Innanzitutto, manca nella versione attuale un sistema di tracciamento che consenta a un utente che effettua l'accesso di sapere quali modifiche sono state apportate all'ontologia dal momento in cui egli ha chiuso l'applicazione la volta precedente (sulla falsariga del sistema ChAO in uso in Protégé e Webprotege).

Gli utenti potranno essere dotati, inoltre, di un proprio profilo (ispirandosi quindi ai social-network ove ogni utente può gestire liberamente le informazioni che lo riguardano) in cui decidere quali dati mostrare e quali tenere privati, cambiare le proprie credenziali di accesso, creare "gruppi di lavoro", ecc...

Il concetto di "gruppo di lavoro" è ignorato, per ora, da OWLIE. Tutti gli utenti hanno la medesima dignità e le stesse possibilità di accesso a tutti i progetti condivisi sul server, compatibilmente con quanto stabilito nel meta-progetto che configura le regole di accesso al server di Protégé. Appare impensabile che un'applicazione possa essere pronta per un uso abituale, al di là delle necessità accademiche, senza un opportuno adeguamento delle modalità di accesso e condivisione.

Da queste riflessioni emergono pertanto due spunti per una versione futura di OWLIE: il potenziamento del sottosistema collaborativo da una parte, e l'introduzione di criteri di accesso di gruppo dall'altra.

5.2 Elementi innovativi

OWLIE è un'applicazione web che implementa funzionalità già piuttosto diffuse in applicazioni desktop.

Il layout a linguette (tab) ognuna delle quali è collegata a una particolare vista dell'ontologia (tipicamente classi, proprietà e individui) è molto comune.

Anche Webprotege, pur con tutti i suoi limiti, adotta un'interfaccia strutturata in modo analogo. La vera innovazione, dunque, non riguarda né il layout dell'interfaccia utente né il livello di espressività.

Il primo elemento che realmente costituisce una novità nel panorama universale degli editor delle ontologie è l'editor grafico della T-BOX. Sebbene infatti esistano applicazioni non basate sul web (ad esempio, GrOWL) che consentono l'editing grafico dell'intera ontologia (non solo la T-BOX), OWLIE è, per ora, l'unico editor di ontologie basato sul web che compie un passo in tal senso.

L'editor della T-BOX implementato in OWLIE è piuttosto essenziale, anche se aperto a prossimi incrementi di espressività: non è in altri termini possibile compiere tramite di esso la stessa gamma di operazioni disponibili utilizzando le due viste "classiche" della T-BOX, ossia gli editor delle classi e delle proprietà. Ciò è dovuto in primo luogo alle limitazioni che l'uso di un'interfaccia utente costituita da elementi AJAX pone nella realizzazione di costruzioni grafiche complesse. Non occorre dimenticare, infatti, che sino a pochi anni or sono le pagine web erano costituite solamente da testo ed immagini statiche.

Le limitazioni dell'interfaccia web ad ospitare strutture grafiche dinamiche di particolare complessità, costituiscono il motivo per il quale non è stato possibile realizzare per la A-BOX un editor grafico simile a quello implementato per la T-BOX. Ciò nonostante, in OWLIE è realizzato un visualizzatore in grado di rappresentare il segmento di A-BOX di interesse (o al limite la sua totalità).

Come l'editor della T-BOX rappresenta un'innovazione, anche il visualizzatore della A-BOX costituisce un elemento di novità tra le applicazioni basate sul web. L'unico potenziale concorrente infatti potrebbe essere Ontoverse (vedasi paragrafo 2.3.2), che implementa, secondo i suoi autori, potenti funzionalità di visualizzazione ma che, allo stato attuale e come già osservato, pare non essere più funzionante.

Un'interessante prospettiva, in vista dello sviluppo di versioni future di OWLIE, è rendere il visualizzatore della A-BOX capace di interpretare e rispondere a comandi da parte dell'utente in grado di modificare le asserzioni, trasformandosi in tal modo in un editor grafico. Questo passaggio non è affatto semplice e comporta lo studio di tecniche di *mapping* e di gestione degli eventi diverse da quelle supportate nativamente da ZK e utilizzate in OWLIE.

5.2.1 Confronto con software preesistenti

In tabella 5.2 è riportato un confronto sintetico tra OWLIE e gli altri software considerati nel capitolo 2, che riassume quanto argomentato poc'anzi.

In questa tabella sono raffrontate varie caratteristiche dell'applicazione, quali la presenza di un editor classico (ossia che presenti le varie viste dell'ontologia, divise per tipo di elemento, e che consenta di modificare la definizione e le proprietà di ogni singolo elemento, sia esso un concetto, un ruolo o un individuo), la possibilità di effettuare editing grafico (sfruttando cioè costrutti puramente grafici), il tipo di visualizzazione grafica implementata, sulla base dello

Applicazione	Basata sul web	Editor classico	Editor grafico	Tipo di visualizzazione	Collaborazione	Supporto OWL 2
Protégé 3	NO	SI	SI (plugin ezOWL)	gerarchica, A rete (OWLViz, OWLPropViz, Jambalaya)	SI	NO
Protégé 4	NO	SI	NO	gerarchica, A rete (OWLViz)	NO	SI
WebProtege	SI	SI	NO	nessuno	SI	NO
NeOn Toolkit	NO	SI	NO	gerarchica	SI	SI
TopBraid Composer	NO	SI	NO	gerarchica	NO	NO
Knoodl	SI	SI (wiki)	NO	gerarchica	SI	NO
Ontoverse	SI	NO	NO	gerarchica, A rete	SI	NO
GrOWL	NO	NO	SI	a rete	NO	NO
OWLIE	**SI**	**SI**	**SI (T-BOX)**	**gerarchica, Mista gerarchica-rete (TBOXGraph, ABOXGraph)**	**SI**	**NO**

Tabella 5.2: *Confronto tra OWLIE ed altri software considerati*

studio di cui al paragrafo 2.2.1, la presenza di un sistema collaborativo e, infine, il supporto per il nuovo standard OWL 2.

5.3 I limiti dell'architettura utilizzata

L'architettura client-server su cui OWLIE si basa, distribuita su tre livelli, è di per sé la soluzione migliore nel caso di applicazioni che offrano al client un servizio, ne elaborino le richieste e vi forniscano risposte, lasciando però a un livello più interno l'onere dell'elaborazione e della persistenza dei dati.

Protégé server è molto potente, e il livello di astrazione fornito tramite le sue API permette un'implementazione veloce e leggera. Ciò nonostante, il suo utilizzo come server di dati presenta alcune criticità, per rispondere alle quali sono necessari accorgimenti tecnici la cui implementazione e gestione è, nel migliore dei casi, contorta e complessa.

La prima criticità è rappresentata dalle politiche di accesso e condivisione. Come già analizzato nel paragrafo 2.4.2, il *meta-progetto* (*metaproject*) contiene sia i dati di login degli utenti registrati (nome utente, password e qualifica, ovvero gruppo di appartenenza), sia le politiche di controllo degli accessi, ossia le specifiche di quali ontologie sono condivise, quali utenti possono accedere a ciascuna di esse e in che modo (sola lettura, lettura e scrittura).

La presenza del metaprogetto, che non è un'ontologia in OWL bensì un'applicazione di Protégé-Frames, vincola il sistema ad accettare solamente le di accesso definite in esso, e non poterle eludere se non tramite sotterfugi. In altri termini, OWLIE non può definire politiche proprie di accesso, ovvero non può implementare un sistema di controllo dell'identità dell'utente che vada in contrasto con le informazioni contenute nel metaprogetto del server di Protégé.

Ovviamente, il metaprogetto può essere manipolato sia direttamente dall'amministratore, agendo sulla stessa macchina ove il server è installato tramite un'istanza di Protégé, sia in modo automatico dall'applicazione a livello 1 dell'architettura, mediante metodi messi a disposizione dalle API di Protégé.

Il server di Protégé oltre alla rigidità delle politiche di accesso e condivisione presenta un altro svantaggio, dall'impatto non trascurabile: il consumo di risorse.

In fase di avvio, infatti, il server di Protégé procede caricando in memoria tutte le ontologie condivise tramite il metaprogetto (ovvero creando gli oggetti java che rappresentano le ontologie stesse e le risorse che le costituiscono). Supposto che, verosimilmente, server di Protégé e server web di OWLIE sulla stessa macchina e che tale elaboratore non abbia altra finalità se non quella di costituire l'host su cui far girare entrambi i server, per ontologie di piccole o medie dimensioni il consumo di risorse può non essere troppo invalidante sulle prestazioni del sistema nel suo complesso.

Per ontologie di grandi dimensioni, nelle quali il numero delle triple supera le diverse centinaia, l'uso eccessivo di risorse da parte del server di Protégé può degradare le prestazioni di OWLIE in fase di interrogazione e di aggiornamento dei propri componenti interni, e tale rallentamento è naturalmente avvertito, per non dire sofferto, anche dall'utente che agisce lato client.

In questo caso il problema è tutto interno al server di Protégé, e non vi sono, anche dopo uno studio accurato della relativa documentazione, soluzioni accettabili.

L'inadeguatezza del server di Protégé a supportare ontologie realizzate secondo i nuovi standard W3C è alla base di una riflessione più approfondita su come superare i limiti derivati dal suo utilizzo, oggetto del prossimo paragrafo.

5.4 Verso OWLIE 2.0?

Sebbene lo studio di fattibilità fosse stato avviato tempo prima, ossia nei primi mesi del 2009, lo sviluppo vero e proprio di OWLIE ha avuto inizio durante il mese di luglio dello stesso anno.

Durante le prime fasi che hanno seguito lo studio dello stato dell'arte, ovvero la valutazione di tutte le tecnologie candidate a fornire supporto allo sviluppo del software e la successiva scelta di un sottoinsieme di esse, ritenute migliori delle altre, anche lo stato dell'arte andava evolvendosi.

Il rischio che qualcuna delle scelte effettuate in fase di design dell'applicazione si rivelasse obsoleta ancor prima di terminarne lo sviluppo si presentava tutt'altro che infondato. OWL 2, in effetti, non era ancora stato elevato a standard raccomandato dal W3C, cosa che è avvenuta mesi più tardi, pertanto si è ritenuto di concentrare tutti gli sforzi su un'applicazione che almeno lo standard precedente, il più datato (ma non per questo deprecato) OWL.

D'altro canto, le applicazioni che si sono evolute verso OWL 2 si contano sulle dita di una mano. Protégé 4 tuttora manca di molte funzionalità presenti nella vecchia versione, che comunque non supporta OWL 2. La mancanza di alcune di tali capacità, nello specifico l'assenza del supporto client-server, lo

rende per ora totalmente inadatto a costituire la base per una futura versione di OWLIE al posto di Protégé 3.

Occorre inoltre non trascurare un importante dettaglio implementativo. OWLIE fa un uso intenso delle API di Protégé-OWL che, è bene ribadire, supportano i costrutti di OWL ma non di OWL 2. Il team di Protégé decise di non supportare più tali librerie, che fanno uso di Jena (vedasi paragrafo 2.3.4), nelle versioni future del programma a partire da Protégé 4.0, sostituendole con un altro set di librerie, le OWL API, non più basate su Jena, al prezzo di una totale riscrittura del software.

Le domande, come si usa dire, sorgono spontanee: quale futuro per OWLIE? Quale strada intraprendere per allineare il software agli standard?

Applicare ad OWLIE una scelta simile a quella fatta dagli sviluppatori di Protégé riguardo alle API di Protégé-OWL, ossia sostituirle con le OWL API, costituirebbe una strada senza dubbio molto onerosa e rischiosa.

Le occorrenze di classi e interfacce appartenenti a package di Protégé-OWL andrebbero sostituite da oggetti che siano istanze di classi analoghe appartenenti ad un nuovo set di API, ad esempio OWL API. Siccome non è garantito che per ogni classe di Protégé-OWL esista un omologo in OWL API, la migrazione potrebbe rivelarsi più complicata del previsto, ma pur sempre meno onerosa di una totale riscrittura del codice.

Una volta completato il passaggio ad OWL API l'applicazione supporterebbe OWL 2, anche se in tal caso occorrerebbe aggiungere all'interfaccia grafica i costrutti specifici della logica SROIQ(D) che non erano presenti nella logica SHOIN(D); ciò è tuttavia compreso nel prezzo da pagare per l'aggiornamento.

La sostituzione aprirebbe una seconda questione, che riguarda la presenza o meno del server di secondo livello (che nella versione attuale è rappresentato dal server di Protégé).

In questo caso le possibilità potrebbero essere due. Una prima soluzione, probabilmente la più semplice, comporta di attendere fintantoché gli sviluppatori di Protégé non completino l'implementazione delle funzionalità client-server anche sul nuovo framework di Protégé 4. L'attesa, che oggettivamente si può prevedere nell'ordine di mesi, potrebbe essere riempita svolgendo con la dovuta cautela il passaggio di sostituzione delle librerie descritto poc'anzi.

Un'alternativa, interessante ma al tempo stesso più impegnativa, prevede di implementare la gestione degli accessi e della concorrenza in un nuovo server di secondo livello, il cui progetto deve iniziare dalle fondamenta, al posto del server di Protégé. Con ogni probabilità, le tempistiche di realizzazioni sarebbero più contenute rispetto all'attesa del rilascio del Protégé4-server, di cui del resto nulla è dato sapere, al momento.

Riassumendo, raffrontando le funzionalità attuali di OWLIE con le racco-

mandazioni del W3C e con lo stato degli altri software operanti nel medesimo ambito, non si può certo affermare che OWLIE nasca "vecchio", in quanto risponde alle esigenze per cui è stato progettato. Se nel frattempo gli standard si sono innalzati e con essi i requisiti minimi, non vi è altra soluzione che un aggiornamento che tenga conto del progresso nello stato dell'arte.

La soluzione più logica individuata è la sostituzione delle librerie di Protégé-OWL con un analogo set di API che supporti OWL 2, a cui far seguire l'appoggio sul server di Protégé 4 se nel frattempo è stato implementato, o altrimenti la realizzazione ex-novo di un server di secondo livello (ossia che gestisca la concorrenza e l'accesso ai dati).

In questo modo è auspicabile che forma **OWLIE 2.0**.

Appendice A

Manuale d'installazione

Questa appendice vuole fornire una guida sintetica, ma al tempo stesso completa, riguardo all'installazione di OWLIE e al suo utilizzo.

Nelle sezioni che seguiranno, sarà fatto riferimento alle due piattaforme più diffuse, ossia GNU/LINUX e Microsoft Windows.

A.1 Prerequisiti

Come già ampiamente trattato nei paragrafi 3.3.1, 3.3.3, OWLIE si basa su un'architettura client-server a tre livelli. È possibile tuttavia utilizzare l'applicazione, a scopo di test o di valutazione, anche utilizzando un unico elaboratore.

Indipendentemente dal sistema operativo in uso, devono essere installati e correttamente configurati i seguenti componenti software (reperibili facilmente sul web e liberamente utilizzabili in quanto tutti distribuiti sotto licenze non commerciali).

- **Piattaforma JAVA**
  Necessaria in quanto *Protégé*, OWLIE e il Server Tomcat sono scritti in Java. È possibile reperire liberamente la versione più recente del *Java Resource Environment* (JRE) sul web all'indirizzo `http://www.java.sun.com`.
  In alternativa, è possibile installare ed utilizzare un *Java Development Kit* (JDK) (che contiene al suo interno anche un JRE), reperibile anch'esso liberamente sul web all'indirizzo `http://java.sun.com/javase/downloads/index.jsp`. Ne esistono versioni per i principali sistemi operativi.

- **Server web Apache Tomcat**
  Necessario per eseguire OWLIE. Reperibile gratuitamente sul web, all'indirizzo `http://tomcat.apache.org`. OWLIE è ottimizzata per la versione 6 del server web Apache Tomcat.

L'installazione e la configurazione di Tomcat non sono argomento di questa trattazione, pur essendo necessarie affinché l'applicazione funzioni correttamente. Sono tuttavia disponibili diverse guide on-line, tra cui una, abbastanza completa, è fornita dai suoi medesimi sviluppatori e si trova all'indirizzo `http://tomcat.apache.org/tomcat-6.0-doc/index.html` (riferita alla versione 6).
In alternativa a Tomcat possono essere utilizzati altri server web supportanti le *servlet* JAVA. Uno di questi, per esempio, è *JBOSS* (`http://www.jboss.org`), tuttavia, essendo lo sviluppo di OWLIE effettuato sulla base di Tomcat, ed essendo l'applicazione ottimizzata per l'esecuzione su tale piattaforma, la migrazione su altri server, seppur possibile, richiede ulteriori operazioni di configurazione e messa a punto, non descritti da questo manuale.

- **Protégé server**
  Seppur non strettamente necessario per effettuare test dimostrativi su OWLIE, consente l'utilizzo multi-utente, fornisce i metodi per il controllo degli accessi e gestisce la persistenza dei dati.
  Sull'installazione e la configurazione di Protégé Server è dedicata una sezione apposita (A.2.1), mentre riguardo ad informazioni di carattere generale su Protégé si faccia riferimento al paragrafo 2.4.

- **Graphviz**
  Il software Graphviz, distribuito sotto licenza *Common Public License* (CPL), è una suite di applicazioni utili per rappresentare grafi e diagrammi. OWLIE richiede che sul medesimo elaboratore in cui è installato il server Tomcat, sia installata e correttamente configurata l'applicazione *dot* di Graphviz.

A.2 Installazione

A.2.1 Installazione di Protégé

Come esaminato nel paragrafo 2.4.2, Protégé Server si occupa della persistenza dei dati, delle autorizzazioni e della concorrenza degli accessi. OWLIE utilizza le API di Protégé 3, la *major release* stabile più recente. Nonostante lo sviluppo di Protégé 4 sia in continuo avanzamento (rimanendo tuttavia, come già discusso nel paragrafo 2.4.5, inadatto a fornire supporto per un'applicazione completa), i progettisti di Protégé proseguono nello sviluppo di Protégé 3, rilasciando di tanto in tanto nuove versioni del software.

La versione incorporata da OWLIE, quindi lato client (rispetto alla connessione RMI), è la 3.4.2. In data 8 marzo 2010 è stata rilasciata la versione 3.4.4. Per cause interne alle librerie di Protégé, è altamente probabile, come osservato

Figura A.1: *Programma di installazione di Protégé 3.4.1*

in fase di test dell'applicazione, che versioni differenti delle librerie lato client e lato server risultino in problemi di incompatibilità.

Per questa ragione è opportuno che la versione di Protégé installata sia la stessa che accompagna OWLIE (non solamente in termini di numero di versione).

La versione di Protégé più recente è ad ogni modo liberamente reperibile sul web, nel sito dedicato al progetto (`http://protege.stanford.edu`), come pure la documentazione.

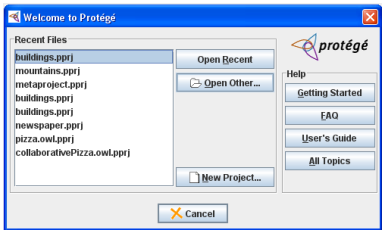
L'installazione è semplice ed è assistita dal software (un'immagine del programma di installazione è ritratta in figura A.1). Sono disponibili installer guidati per la maggior parte delle piattaforme.

Il passaggio successivo consiste nell'avvio del server, ma ciò non è possibile senza aver correttamente configurato il *metaprogetto*.

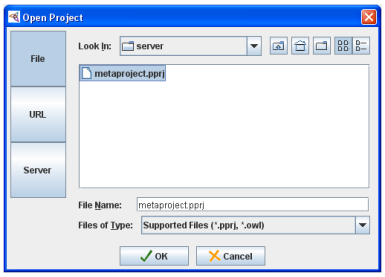
A.2.2 Impostazione del metaprogetto

Il metaprogetto (*metaproject*) è lo strumento utilizzato da Protégé Server per effettuare il controllo degli accessi. In questo speciale progetto (che non è un'ontologia, ma piuttosto una base di dati) sono registrati gli utenti che hanno accesso al server (e pertanto ai progetti OWL), e tutti gli oggetti per i quali definire una politica di accesso (ad esempio, i progetti).

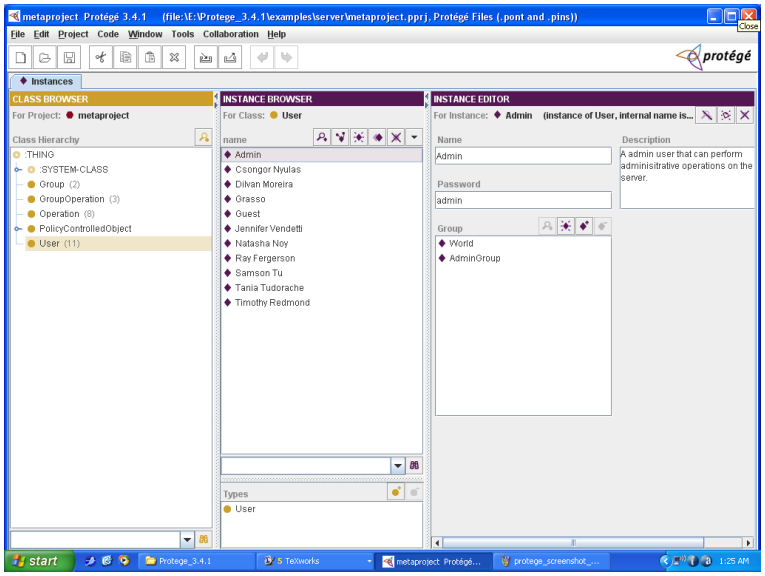
La prima operazione da svolgere è assicurarsi che un metaprogetto esista, quindi configurarlo inserendovi i nomi degli utenti e le relative password, infine collocarvi i riferimenti ai progetti che si vuole siano raggiungibili dal client, quindi da OWLIE.



(a) apertura progetti



(b) apertura metaprogetto



(c) configurazione degli utenti

Figura A.2: *Configurazione del metaprogetto in Protégé (1)*

Allo stato attuale, OWLIE non implementa le funzionalità di creazione di un nuovo progetto e la registrazione di nuovi utenti (sebbene questo sia uno dei task che è auspicabile implementare in un prossimo futuro), pertanto è necessario che tali operazioni siano effettuate a basso livello, interagendo direttamente con Protégé.

Per impostazione predefinita, il metaprogetto è collocato nella directory `%PROTEGE_DIR%/examples/server` (dove con `%PROTEGE_DIR%` si intende la directory del *file system* dove è installato Protégé). Consiste in tre file distinti: `metaproject.pins`, `metaproject.pont`,`metaproject.pprj`. Il modo più diretto di intervenire sul metaprogetto è aprirlo direttamente con Protégé, come mostrato in figura A.2.

Ora è necessario configurare il metaprogetto. Tutto ciò che serve, al solo scopo di testare le funzionalità di OWLIE, è impostare almeno due utenti (al-

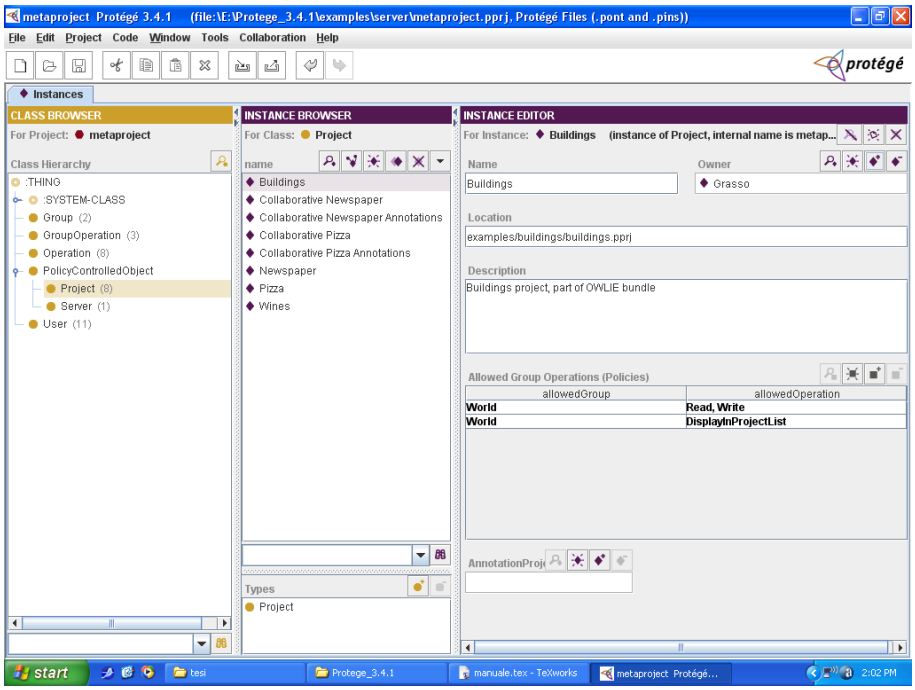


Figura A.3: *Configurazione del metaprogetto in Protégé (2)*

trimenti non è possibile usufruire delle funzionalità collaborative) e almeno un progetto condiviso.

Una volta caricato il metaprogetto, Protégé mostra un layout a tre colonne. In quella più a sinistra, vi è un albero che organizza in classi tutti gli elementi configurabili sul server.

- **Aggiunta di un utente**
  Nella classe `User` sono già preimpostati alcuni utenti (i loro nomi sono quelli dei programmatori che fanno parte del team di Protégé); è possibile utilizzare uno di essi, oppure aggiungerne un altro. Per aggiungere un utente utilizzare il comando create instance presente nella barra degli strumenti della colonna centrale. Nella colonna a destra, infine, è possibile impostare nome utente e password.

- **Aggiunta di un progetto**
  Nella classe `PolicyControlledObject` selezionare la sottoclasse `Project`. Aggiungere un'istanza tramite il comando "create instance" nella barra degli strumenti della colonna centrale, quindi configurare il progetto nella colonna a destra. Nel form che appare, inizialmente vuoto, occorre impostare un nome pubblico per il progetto (che apparirà nella lista dei progetti remoti mostrata da OWLIE), definire il percorso assoluto del file, e infine definirne il proprietario (ogni progetto deve "appartenere" a un

utente, definito in precedenza).
Dato che ogni progetto è composto da più file, il percorso da inserire nella casella "Location" deve puntare al file descrittore del progetto, con estensione `.pprj`.
Prima di passare ad eventuali operazioni successive, assicurarsi che nella casella "Allowed Group Operation (Policies)" compaiano le voci corrette (operazioni che l'utente può svolgere collegandosi al progetto da remoto) come in figura A.3.

- **Salvataggio del metaprogetto**
  Il metaprogetto deve essere salvato su disco per poter garantire la persistenza delle modifiche fatte.

A.2.3 Avvio di Protégé Server

Protégé Server non è un'applicazione a sé stante, bensì è parte di ogni installazione standard di Protégé 3. Una guida dettagliata alla configurazione e all'amministrazione del server è presente, in lingua inglese, all'interno della documentazione *wiki* di Protégé ([14]): per esigenze di sintesi, verranno di seguito riportati i passi fondamentali per installare e configurare Protégé Server al fine di eseguire correttamente OWLIE.

A prescindere dal sistema operativo, il requisito fondamentale per l'utilizzo del server è un'installazione standard di Protégé 3 (è sufficiente l'installazione del modulo principale, senza il plug-in di Protégé-OWL).

Una volta installato il modulo base di Protégé sull'hard disk, nella directory principale di Protégé saranno presenti gli script di shell (in GNU-Linux) o i file batch (in Windows, come riportato in figura A.4) per avviare e arrestare il server.

Tali script sono generati dal programma di installazione di Protégé e contengono riferimenti alla *Java Virtual Machine* (JVM) scelta durante il processo di installazione.

Per poter utilizzare Protégé server come base per l'esecuzione di OWLIE, occorre impostare altri due parametri nel file batch (o nello script di shell) di avvio del server stesso, come esemplificato, nel caso di piattaforma Windows, in figura A.5.

- *intervallo di salvataggio automatico* – OWLIE non prevede il salvataggio automatico delle ontologie condivise. Ciò è effettuato dal server di Protégé tramite un'opzione che va impostata prima dell'avvio dello stesso. Nel file batch o nello script di shell, occorre attivare (togliere il commento) dal comando che imposta la variabile `SAVE_INTERVAL`. Il valore predefinito è di 120 secondi: si consiglia un intervallo più ridotto, non superiore ai 30 secondi. Ciò protegge da eventuali arresti imprevisti del server (ad esempio, per un black out che comporti lo spegnimento improvviso dell'elaboratore che lo ospita).

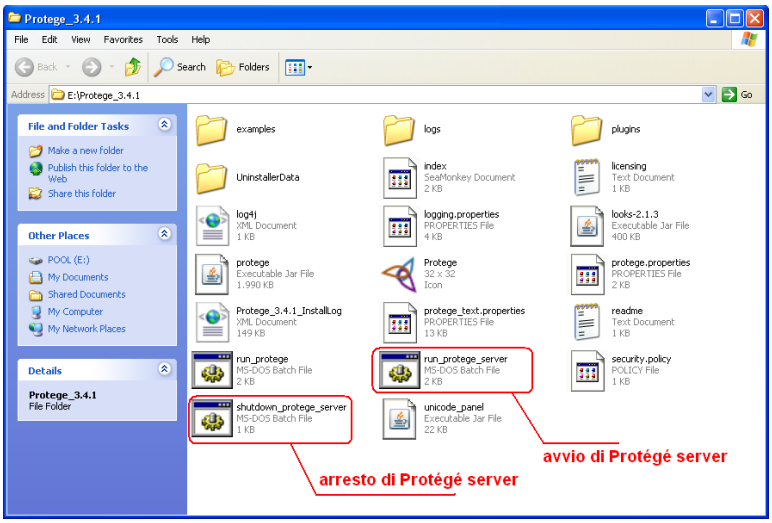


Figura A.4: *Comandi batch per l'avvio e l'arresto del server di Protégé in Windows*

- *definizione delle politiche di sicurezza* – in fase di progetto, sono stati rilevati problemi di sicurezza durante l'utilizzo di Protégé server da parte di OWLIE. Per ovviare a questo inconveniente occorre impostare il file batch come in figura in modo che all'avvio del server vengano utilizzate le opzioni di sicurezza contenute nel file `security.policy` anziché quelle di default, più restrittive. Il file `security.policy` è contenuto nella directory principale di OWLIE. Per comodità, è consigliabile copiarlo e importarlo nella directory di Protégé.

A.2.4 Deployment di OWLIE su server web

L'operazione di *deployment* è la fase di installazione di OWLIE in senso stretto. Le versioni più recenti di Tomcat, in particolare la versione 6, consentono per impostazione predefinita l'*auto-deploy*, che consiste nella creazione automatica della struttura di directory dell'applicazione web (nonché la sua configurazione) senza l'intervento dell'utente.

Tipicamente, non occorre fare altro che disattivare il server (se già attivo) tramite il comando `shutdown`, copiare il file `owlie.war` nella directory `webapps`, quindi riavviare il server con il comando `startup`[1].

Conseguentemente all'azione di riavvio del server, sarà creata automaticamente una directory denominata `owlie` all'interno della directory `owlie`; il

[1] utilizzare il comando `startup.sh -security` in GNU/Linux e `startup.bat -security` in Windows. L'opzione `security` è necessaria al fine di caricare con successo il security-manager a sua volta indispensabile per la corretta esecuzione di RMI

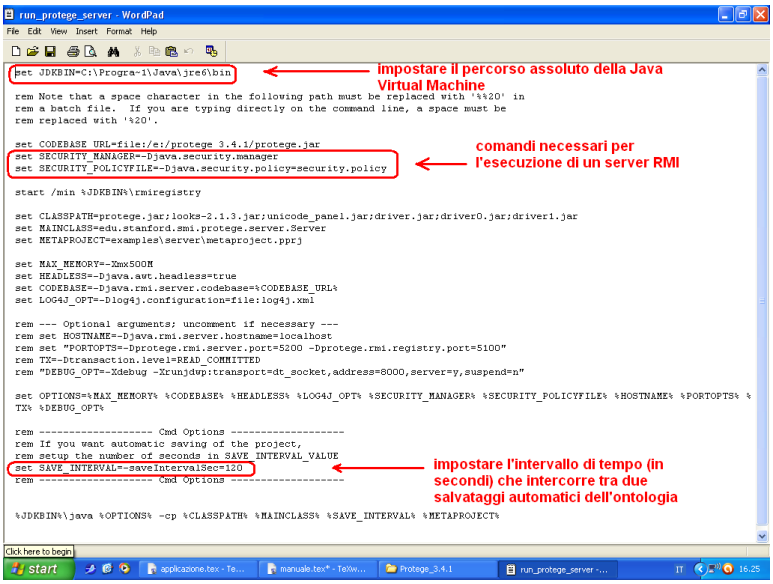


Figura A.5: *Parametri critici nel file batch per l'avvio del server di Protégé in Windows*

contenuto del file `owlie.war` verrà estratto e copiato nella nuova directory. Da questo momento in poi, OWLIE è attivo sul server e utilizzabile via browser web.

Per effettuare un primo test, si può avviare il browser web digitando l'indirizzo

```
http://localhost:8080/owlie
```

(dove il numero 8080 è la porta *Transmission Control Protocol* (TCP) su cui il server Tomcat accetta connessioni dall'esterno: se la porta TCP di Tomcat è impostata a un valore diverso, è necessario sostituire il numero 8080 con il numero di porta appropriato[2]).

Se non si verificano errori, dovrebbe apparire la schermata principale di OWLIE, come riportato in figura A.6 Eventuali errori possono essere dovuti a una non corretta configurazione del server, oppure ad impostazioni troppo restrittive che non consentono al server di ricevere connessioni in ingresso, anche dalla medesima macchina.

Un errore tipico del primo caso è la mancata impostazione della variabile d'ambiente `JAVA_HOME` (la quale deve puntare alla directory dove è installato il JRE) necessaria affinché Tomcat possa avviarsi. I comandi che consentono di impostare una variabile d'ambiente variano in base al sistema operativo utilizzato.

[2]Di norma, la porta usata per il protocollo HTTP è la numero 80. In tal caso, nella maggior parte dei browser la si può omettere. Tomcat, tuttavia, per impostazione predefinita utilizza la porta 8080, a meno che tale impostazione non venga modificata dall'utente, per ragioni di sicurezza.

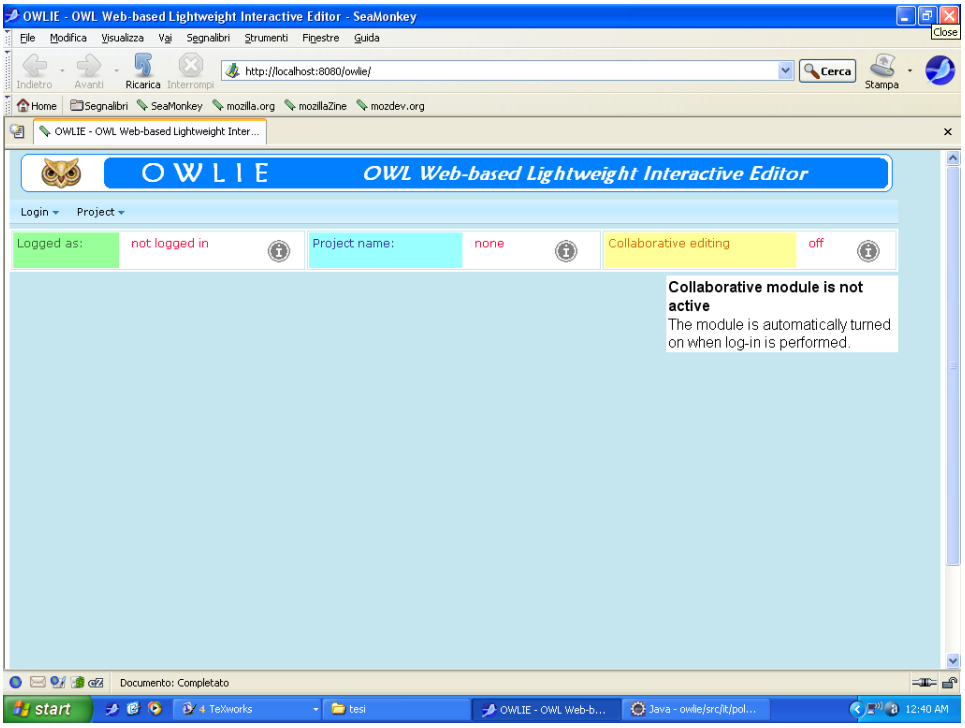


Figura A.6: *Schermata iniziale di OWLIE*

Bibliografia

[1] Tim Berners-Lee. *L'architettura del nuovo Web*, Feltrinelli, Milano 2001.

[2] *W3C Semantic Web Activity* (http://www.w3.org/2001/sw)

[3] Michael K. Smith, Chris Welty, Deborah L. McGuinness. *OWL Web Ontology Language Guide*, W3C 2004. (http://www.w3.org/TR/owl-guide).

[4] Peter Patel-Schneider, Ian Horrocks. *OWL 1.1 Web Ontology Language*, W3C 2006. (http://www.w3.org/Submission/2006/SUBM-owl11-overview-20061219/).

[5] M. Piccinno. *Funzionalità avanzate di visualizzazione tridimensionale di ontologie*, Politecnico di Torino, 2005 (http://elite.polito.it/files/thesis/fulltext/piccinno.pdf)

[6] Alessio Bosca, Dario Bonino, Paolo Pellegrino. *OntoSphere: more than a 3D ontology visualization tool*, Politecnico di Torino, 2005.

[7] Ben Shneiderman. *Treemaps for space-constrained visualization of hierarchies*, ACM Transactions on Graphics (TOG), 1992, volume 11, issue 1, pagg. 92-99

[8] P. Haase, H. Lewen, R. Studer, T. Tran, M. Erdmann, M. d'Aquin, Enrico Motta. *The NeOn Ontology Engineering Toolkit*, WWW2008 (http://www.aifb.uni-karlsruhe.de/WBS/pha/publications/neon-toolkit.pdf)

[9] M. Chung, S. Oh, K. I. Kim, H. S. Cho, H. K. Cho *Visualizing and Authoring OWL in ezOWL* ICACT, 2005

[10] *Manchester OWL Syntax* (http://www.co-ode.org/resources/reference/manchester_syntax/)

[11] Holger Knublauch. *An AI tool for the real world: Knowledge modeling with Protégé* JavaWorld, 20 giugno 2003.

[12] *Protégé Overview* (http://protege.stanford.edu/overview/index.html)

[13] *Protégé Wiki, Protégé 4 migration* (http://protegewiki.stanford.edu/wiki/Protege4Migration)

[14] *Protégé Wiki, client-server tutorial* (http://protegewiki.stanford.edu/index.php?title=Protege_Client-Server_Tutorial&oldid=6542)

[15] Jesse J. Garret. *Ajax: A New Approach to Web Applications*, 2005 (http://www.adaptivepath.com/ideas/essays/archives/000385.php)

[16] Robbie Cheng, *Simple and Intuitive Server Push with a Chat Room Example*, ZK Smalltalks, 2007 (http://docs.zkoss.org/wiki/Simple_and_Intuitive_Server_Push_with_a_Chat_Room_Example)

[17] Harvey M. Deitel, Paul J. Deitel. *Java: fondamenti di programmazione*, pag. 314