UNIVERSITÀ DEGLI STUDI DI FIRENZE
Dipartimento di Sistemi e Informatica

Dottorato di Ricerca in
Ingegneria Informatica, Multimedialità e Telecomunicazioni
ING-INF/05

Architetture Software Ontologiche Principi ed Applicazioni

Valeriano Sandrucci

Dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

in Informatics, Multimedia and Telecommunication Engineering

Ph.D. Coordinator
Prof. Giacomo Bucci

Advisors
Prof. Enrico Vicario
Prof. Giacomo Bucci

XXIII Ciclo – 2009-2011

A Clarice, a mio fratello e ai miei genitori

Sommario

L'insieme di linguaggi, protocolli, componenti e standards introdotti nel contesto delle attività legate al Semantic Web delineano un vero e proprio paradigma ontologico per la rappresentazione ed elaborazione della conoscenza. In questo lavoro proponiamo un'architettura ontologica in grado di valorizzare le potenzialità delle ontologie nello sviluppo di applicazioni e di limitarne gli aspetti critici. In particolare mostreremo l'analisi ed il design relativi al componente di object-ontology mapping, centrale nell'architettura, che abbiamo realizzato per risolvere l'impedance mismatch tra modelli object-oriented e modelli ontology-oriented. Verranno inoltre mostrate alcune sperimentazioni dell'architettura ontologica in ambiti applicativi differenti.

Indice

Introduzione

In Informatica le ontologie sono una tecnologia che abilita la costruzione di concettualizzazioni condivise attraverso processi distribuiti, incrementali e collaborativi [64], permettendo la descrizione formale, esplicita e non ambigua di conoscenza secondo un approccio dichiarativo riconducibile alle logiche descrittive [41][42]. Le ontologie nascono, nella forma in cui verranno discusse in questo lavoro, nel contesto degli sforzi legati alla realizzazione del Semantic Web [5] cioè di quell'evoluzione del web attuale in cui l'informazione avrà una precisa caratterizzazione semantica tale da permettere la creazione di agenti intelligenti ed elaborazioni automatiche. Le ontologie rappresentano uno strumento largamente studiato e sempre più diffuso. Ad oggi è infatti disponibile un insieme di linguaggi, protocolli, componenti e standard [109][106] che delinea un vero e proprio paradigma ontologico per la rappresentazione ed elaborazione della conoscenza.

Abbiamo condotto numerose sperimentazioni, in ambiti applicativi differenti, che si sono concretizzate nella realizzazione di prototipi funzionanti e sono servite ad evidenziare tangibilmente i benefici che le tecnologie per la modellazione e per il ragionamento ontologico portano alla costruzione di architetture software evolute. L'esperienza maturata, insieme allo studio della letteratura, ci ha permesso di verificare l'adeguatezza delle ontologie per la costruzione di sistemi software con alto grado di interoperabilità ed adattabilità ma ha anche evidenziato la scarsità di spiegazioni disponibili su come ingegne-

rizzare l'architettura dei software ontologici e quindi la carenza di informazione su come strutturare le applicazioni.

In questo lavoro approfondiamo le riflessioni riguardanti l'impiego di ontologie nello sviluppo di sistemi software individuando le caratteristiche architetturali che le applicazioni ontologiche dovrebbero possedere ed arrivando a definire un'architettura ontologica di cui descriviamo la realizzazione.

In particolare abbiamo progettato, implementato e testato Loom (Library for Object-Ontology Mapping), un componente software di object-ontology mapping che serve a risolvere il problema di impedance mismatch [89][40] esistente tra i modelli object-oriented e le concettualizzazioni di tipo ontologico.

Proponiamo l'uso di tale componente per l'organizzazione di architetture software che siano in grado di valorizzare i tratti positivi delle ontologie nelle applicazioni limitandone invece problemi e complessità.

Il lavoro è organizzato in quattro capitoli:

- Capitolo 1: fornisce l'insieme di definizioni e riferimenti che servono ad introdurre il problema dell'analisi, della progettazione e della valutazione di architetture software.

- Capitolo 2: descrive la nostra proposta di architettura ontologica, illustrandone la struttura e le motivazioni.

- Capitolo 3: mostra i dettagli dell'analisi e del design relativi componente di object-ontology mapping (Loom) implementato.

- Capitolo 4: riporta alcune significative sperimentazioni che abbiamo svolto, con lo scopo di evidenziare l'effettivo potenziale delle ontologie nello sviluppo di applicazioni.

Capitolo 1

Architetture Software

Al crescere delle dimensioni dei sistemi software aumentano in maniera corrispondente le difficoltà che analisti e sviluppatori devono affrontare per rispondere alle aspettative dei vari soggetti interessati al sistema durante tutto il suo ciclo di vita.

Quando i sistemi software sono costituiti da molti componenti interconnessi ed interagenti tra di loro, la definizione degli algoritmi e delle strutture dati finiscono per non essere più il problema progettuale maggiore.

L'organizzazione complessiva del sistema software ovvero la sua architettura rappresenta un nuovo insieme di problemi progettuali di cui tenere conto e particolarmente importati diventano quei metodi e quelle tecniche che permettono di svolgere analisi e progettazione a tale livello di granularità.

Con riferimento alla definizione di Architettura Software di Eoin Woods [32] si può infatti dire che l'architettura di un software è quell'insieme di decisioni che, se prese male, portano alla cancellazione del progetto.

1.1 Definizioni

Le seguenti definizioni, diverse tra loro, riassumono le principali posizioni sul significato di architettura.

- L'architettura software è l'insieme di quelle decisioni progettuali che, se prese male, possono causare la cancellazione del progetto [32].
- La progettazione e la descrizione della struttura complessiva del sistema risultano essere un nuovo tipo di problema. Questi aspetti strutturali includono l'organizzazione di massima, la struttura del controllo, i protocolli di comunicazione, di sincronizzazione e di accesso ai dati [24].
- Un'architettura software è una descrizione dei sottosistemi e componenti di un sistema software e delle relazioni tra loro. Sottosistemi e componenti sono tipicamente specificati in differenti viste che mostrano le caratteristiche funzionali e non funzionali rilevanti. L'architettura software di un sistema è un artefatto ed è il risultato di un'attività di progettazione [84].
- L'architettura software è l'organizzazione di base di un sistema, espressa dai suoi componenti, dalle relazioni tra di loro e con l'ambiente, e dai principi che ne guidano il progetto e l'evoluzione [66].
- L'architettura software è l'insieme delle strutture del sistema, costituite dai componenti software, dalle loro proprietà visibili e dalle relazioni tra di loro [60].

In questo lavoro verrà utilizzata la definizione che segue e che ha tratto spunto dalle precedenti:

L'architettura di un sistema software (in breve architettura software) è la struttura del sistema; è costituita dalle parti del sistema, dalle relazioni tra le parti e dalle loro proprietà visibili. La struttura definisce principalmente

la scomposizione del sistema in sottosistemi dotati di un'interfaccia e le interazioni tra essi, che avvengono attraverso le interfacce. Le proprietà visibili di un sottosistema definiscono le assunzioni che gli altri sottosistemi possono fare sul sistema (come servizi forniti, prestazioni, uso di risorse condivise, trattamento di malfunzionamenti, etc.)

Considerare solo le proprietà visibili aiuta a chiarire la differenza tra progettazione architettonica e progettazione di dettaglio: solo in quest'ultima, infatti, ci si occupa degli aspetti "non visibili" dei sottosistemi, quali ad esempio strutture dati o algoritmi utilizzati per la loro realizzazione.

1.2 Stili e Pattern Architetturali

L'architettura costituisce un modello relativamente compatto del modo in cui il sistema è strutturato e di come di suoi componenti collaborano tra loro. Questo modello è trasferibile, nel senso che gli schemi e gli stili architetturali utilizzati per un particolare progetto possono essere ripresi ed applicati anche nella progettazione di altri sistemi. Essi infatti rappresentano un insieme di astrazioni, o pattern, che consentono agli architetti di individuare e rappresentare l'architettura secondo modalità ripetibili e prevedibili. Alcune definizioni interessanti di pattern sono quella di [84] che lo definisce come una regola composta da tre parti che esprime la relazione tra uno specifico contesto, un problema ed una soluzione oppure ancora quella citata da Fowler in [36] in cui si afferma che ciascun pattern descrive: sia un problema che ricorre ripetutamente in un ambiente, sia il nocciolo della soluzione di tale problema; la soluzione individuata può inoltre essere impiegata più di milione di volte senza che si ripeta mai nella stessa maniera. Un architectural pattern [84] (pattern architetturale) descrive soluzioni progettuali nell'ambito delle architetture software ed opera quindi ad un livello di astrazione diverso rispetto ai design pattern [39]. Esistono diversi pattern architetturali, ciascuno caratterizzato da: un insieme particolare di componenti pensati per assolvere a determinati compiti; una serie di "connettori" che permettono la comunicazione tra i componenti;

vincoli che stabiliscono le modalità di interazione tra le parti, modelli semantici che consentono al progettista di comprendere le proprietà globali di un sistema analizzando le proprietà note delle parti costituenti. Ad esempio, alcuni pattern architetturali, operanti a livelli di astrazione differente, cui faremo successivamente riferimento, sono:

- Layers che ripartisce le funzionalità dell'applicazione in una pila ordinata di gruppi (layer) in cui ciascun livello può accedere alle funzionalità di quello sottostante ed offrire funzionalità a quello immediatamente superiore
- N-Tier /3-Tier che colloca le funzionalità in segmenti separati in maniera simile al Layers con la differenza che ogni segmento è un tier situato su un computer fisicamente diverso.
- Model-View-Controller che divide una applicazione interattiva in tre componenti: il modello che contiene le funzioni ed i dati; le viste ed i controller che insieme definiscono l'interfaccia utente; un meccanismo di propagazione delle modifiche che garantisce la consistenza tra l'interfaccia utente ed il modello.
- Pipe and Filters che fornisce una struttura per sistemi che processano flussi di dati. Ciascun passo di elaborazione è incapsulato in un componente filtro. I dati sono passati attraverso pipe tra filtri adiacenti. Ricombinare filtri permette di costruire famiglie di sistemi correlati.
- Publisher-Subscriber che aiuta a tenere sincronizzato lo stato di componenti cooperanti abilitando la propagazione one-way delle modifiche: un publisher notifica ad un qualunque numero di subscribers le modifiche del proprio stato.
- Message Bus che prescrive l'uso di sistemi software che ricevono o spediscono messaggi usando uno o più canali di comunicazione cosicché le applicazioni possano interagire senza conoscere i dettagli l'una dell'altra

- Object-Oriented che suddivide le responsabilità di un'applicazione o sistema in oggetti autosufficienti ed individualmente riusabili, ciascuno contenente i dati ed il comportamento rilevante per l'oggetto.
- Service-Oriented Architecture che prescrive l'utilizzo di messaggi e contratti per descrivere i servizi su cui basare le applicazioni

L'architettura di un sistema software non è quasi mai limitata all'utilizzo di un solo pattern architetturale ma è spesso la combinazione di stili architetturali diversi che formano il sistema complessivo. Per esempio si può progettare un sistema che ha architettura SOA composto da servizi sviluppati adoperando un'architettura a Layers e l'approccio Object-Oriented.

1.3 Disegno Architetturale

Molti autori identificano l'architettura di un software con la sua descrizione. In questo lavoro i due concetti saranno tenuti distinti. E' vero che un'architettura è un'entità astratta documentata solo da una descrizione ma, data un'architettura, possono essere fornite diverse descrizioni, che differiscono, per esempio, per il livello di dettaglio. Identificando un'architettura con la sua descrizione si dovrebbe parlare di architetture diverse. Nel processo di sviluppo software, il progettista (o un gruppo di progettisti), dati i requisiti del sistema ed i requisiti del software, definisce un'architettura e la descrive attraverso documenti di progetto. Essendo l'insieme di questi documenti l'unica descrizione dell'architettura immaginata dal progettista, si tende a far coincidere queste descrizioni con l'architettura stessa. Tale insieme sarà invece qui chiamato disegno di progetto architettonico. L'importanza di distinguere tra un'architettura e la sua descrizione si coglie ragionando a posteriori: dato un sistema completamente realizzato, la sua architettura è la sua struttura, e non la descrizione della struttura. Infatti potrebbero esserci più descrizioni, a livelli di dettaglio diversi, o focalizzate su aspetti diversi. La struttura di un software rispetta l'architettura se il disegno di progetto architettonico ne è una

descrizione. Impiegheremo UML [6] per descrivere i vari aspetti (funzionale, logico, strutturale, deploy, etc.) delle soluzioni architetturali proposte.

1.4 Metodologie per la valutazione delle Architetture Software

Lo studio di problemi architetturali, qualunque sia l'insieme di pattern architetturali usati od il formalismo di rappresentazione impiegato ha un unico scopo: quello di produrre software di qualità. Numerosi sono gli studi che affrontano il problema di definire quali sono gli attributi che rendono buono un software e quali sono le caratteristiche che rendendo buona un'architettura, determinando un'alta qualità del software.

Lo standard ISO/IEC 9126 [49] individua una serie di normative e linee guida, sviluppate dall'ISO (Organizzazione mondiale per la normazione) in collaborazione con l'IEC (Commissione Elettrotecnica Internazionale), preposte a descrivere un modello di qualità del software. Il modello propone un approccio alla qualità in modo tale che le società di software possano migliorare l'organizzazione ed i processi e quindi, come conseguenza concreta, la qualità del prodotto sviluppato. Tale normativa è suddivisa in quattro parti: modello di qualità, metriche per qualità esterne, metriche per qualità interne, metriche per qualità in uso. Il modello di qualità individua le caratteristiche fondamentali che determinano la qualità di un software e le organizza in sei gruppi fondamentali di cui il primo, la funzionalità, si riferisce ai requisiti funzionali mentre le altre cinque riguardano i requisiti non funzionali:

- Funzionalità: la capacità di un prodotto software di fornire funzioni che soddisfano esigenze stabilite, necessarie per operare sotto condizioni specifiche. Fanno parte di questo gruppo: appropriatezza, accuratezza, interoperabilità, conformità e sicurezza.

- Affidabilità: la capacità del prodotto software di mantenere un livello specificato di prestazioni quando usato in date condizioni per un dato

periodo. Fanno parte di questo gruppo: maturità, tolleranza agli errori, recuperabilità, aderenza.

- Efficienza: la capacità di fornire appropriate prestazioni relativamente alla quantità di risorse usate. Fanno parte di questo gruppo: comportamento rispetto al tempo, utilizzo delle risorse, conformità.

- Usabilità: la capacità del prodotto software di essere capito, appreso, usato e gradito all'utente, quando usato sotto condizioni specificate. Fanno parte di questo gruppo: comprensibilità, apprendibilità, operabilità, attrattiva, conformità.

- Manutenibilità: la capacità del software di essere modificato, includendo correzioni, miglioramenti o adattamenti. Fanno parte di questo gruppo: analizzabilità, modificabilità, stabilità, testabilità.

- Portabilità: la capacità del software di essere trasportato da un ambiente di lavoro ad un altro. Fanno parte di questo gruppo: adattabilità, installabilità, conformità, sostituibilità.

Le metriche esterne misurano i comportamenti del software sulla base di test, operatività ed osservazione durante la sua esecuzione, in funzione degli obiettivi stabiliti in un contesto tecnico rilevante o di business.

Le metriche interne si applicano al software non eseguibile (ad esempio il codice sorgente) durante le fasi di progettazione e codifica. Le misure effettuate permettono di prevedere il livello di qualità esterna ed in uso del prodotto finale, poiché gli attributi interni influiscono su quelli esterni e su quelli in uso.

Le metriche per le qualità in uso servono a misurare il punto di vista dell'utente sul software. La qualità in uso permette di abilitare specifici utenti ad ottenere specifici obiettivi con efficacia, produttività, sicurezza e soddisfazione.

Questo lavoro farà riferimento al modello di qualità definito dallo standard ISO 9126, seppur in modo non quantitativo e senza impiegare le metriche, al fine di rendere maggiormente esplicito quali siano gli attributi di qualità che beneficiano dell'adozione di un'architettura ontologica.

Capitolo 2

Architettura Ontologica

In Informatica, un'ontologia è una esplicita specificazione di una concettualizzazione [41], un modello che caratterizza la semantica di un dominio definendone entità e relazioni [42].

Le ontologie sono una tecnologia per la rappresentazione della conoscenza e sono state introdotte nel contesto degli sforzi volti a realizzare il Semantic Web [5], quell'evoluzione del web attuale in cui l'informazione avrà una precisa connotazione semantica al fine di rendere possibili elaborazioni automatiche e la realizzazione di agenti intelligenti.

I notevoli sforzi legati alla Semantic Web Initiative del W3C non si sono ancora concretizzati nella completa realizzazione del Semantic Web ma come prodotto intermedio hanno generato un sistema di linguaggi, protocolli, componenti e standards che delineano un vero e proprio paradigma ontologico per la rappresentazione ed elaborazione della conoscenza.

Le ontologie rappresentano oggi uno strumento largamente studiato, piuttosto diffuso e si mostrano adeguate alla realizzazione di sistemi informativi. Tuttavia sono poche le esperienze riportate su come ingegnerizzare un'architet-

tura software e quindi carenti le spiegazioni su come strutturare le applicazioni affinché traggano il massimo beneficio dall'impiego delle ontologie.

2.1 Ontologie

In informatica col termine ontologia si intende una specifica formale ed esplicita di una concettualizzazione condivisa [41][42]. Le ontologie sono cioè una tecnologia per la rappresentazione della conoscenza e costituiscono uno dei componenti fondamentali del Semantic Web [5]. Per meglio comprendere le caratteristiche e le potenzialità delle ontologie è conveniente considerale nel contesto della pila di tecnologie e standard, vedi 2.1 che formano il Semantic Web stesso.

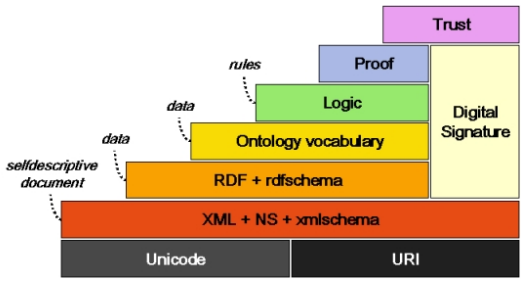

Figura 2.1. Pila delle tecnologie per il Semantic Web

Si osservi come le ontologie, ed in particolare OWL [106], lo standard W3C che le riguarda, dipendano da RDF [109] e RDF Schema [108] che dipendono a loro volta da XML [105] ed XML Schema [112].

Per una descrizione dettagliata dei vari componenti del Semantic Web si rimanda alla documentazione specifica disponibile on-line. In questa sede saranno descritte soltanto le caratteristiche principali di quelle tecnologie utili per comprendere l'architettura ontologica proposta.

2.1.1 RDF ed RDF Schema

RDF (Resource Description Framework) [109] è lo standard proposto dal W3C per la codifica, lo scambio ed il riutilizzo di metadati sul web ed è costituito da due componenti: l'RDF Model and Syntax che descrive la struttura dei modelli ed una loro possibile rappresentazione in XML e l'RDF Schema che descrive la sintassi per definire schemi e vocabolari per i metadati.

In un modello RDF ciascuna risorsa descritta è identificata per mezzo di un URI e l'unità minima di informazione è costituita da un'asserzione, o statement, cioè da una tripletta di risorse, vedi Fig. 2.2, in cui il primo elemento è detto subject, il secondo predicate ed il terzo object. Gli elementi object delle triple possono essere espressioni datatype come stringhe e numeri oltre che risorse.

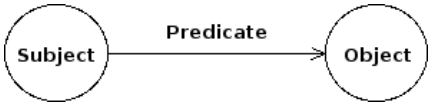

Figura 2.2. Tripla RDF

Ogni modello RDF è costituito da un insieme di statement e, poiché la risorsa che svolge il ruolo di subject in una tripla, può svolgere senza alcuna limitazione, anche quello di predicate o object in altre triple, si ottiene che ciascun modello RDF è rappresentabile per mezzo di un grafo orientato. RDF non attribuisce alcun particolare significato alle risorse descritte nei modelli che risultano quindi estremamente flessibili ed è solo ricorrendo ad RDF Schema che si possono definire gerarchie di classi ed insiemi di oggetti specificando il significato delle risorse. RDF Schema è un vocabolario per la descrizione di proprietà e classi delle risorse RDF, insieme alla semantica per le gerarchie di generalizzazione di tali classi e proprietà.

2.1.2 OWL

OWL (Ontology Web Language) [106] è il linguaggio proposto dal W3C per la codifica dei modelli ontologici ed è parte della pila di tecnologie raccomandate

per il Semantic Web. OWL è un vocabolario, che estende quello di RDF Schema, per la descrizione di classi e proprietà delle risorse RDF.

Per la spiegazione della sintassi completa di OWL si rimanda alle specifiche del W3C; qui verranno descritti solo alcuni concetti fondamentali per fornire una visione d'insieme il più possibile compatta e precisa riguardo alle caratteristiche ed alle potenzialità del linguaggio.

Alla base di ogni modello ontologico sta il concetto di classe, la quale rappresenta una categoria, insieme o collezione (ad esempio Paziente, Medico, ma anche Tempo, Spazio).

OWL permette di definire classi, vedi Fig. 2.3, specificandone il nome, utilizzando costrutti di tipo insiemistico, enumerando gli elementi che ne fanno parte oppure applicando una restrizione su di una proprietà. È inoltre possibile mettere in relazione le classi attraverso legami di ereditarietà, equivalenza o disgiunzione.

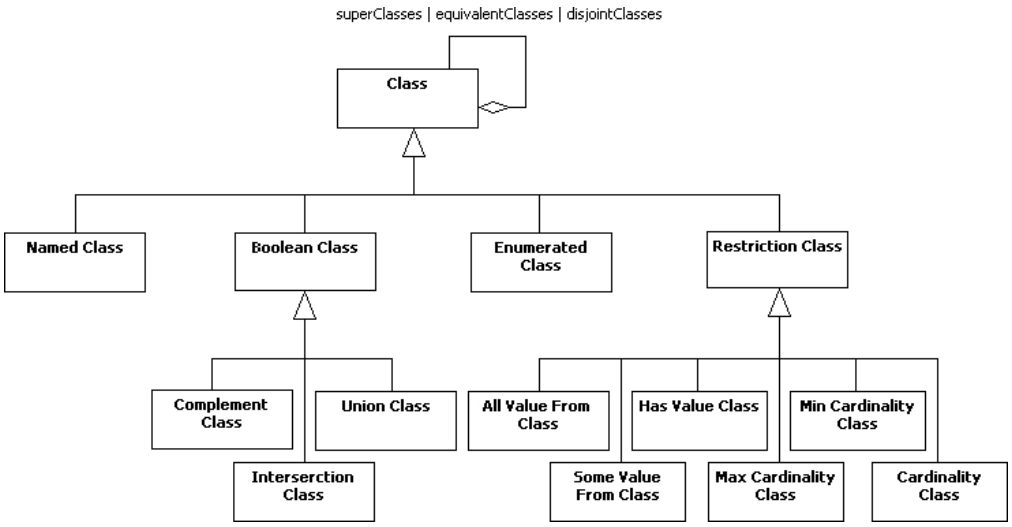

Figura 2.3. Modello semplificato delle classi in OWL

Il secondo elemento fondamentale dei modelli ontologici sono le proprietà, che descrivono la struttura interna degli elementi di una classe o la relazione con elementi di altre classi.

OWL permette di definire le proprietà con le loro caratteristiche quali l'essere simmetriche o transitive, vedi Fig. 2.4. Anche tra le proprietà, al pari delle classi, si possono stabilire legami di eredità ed equivalenza.

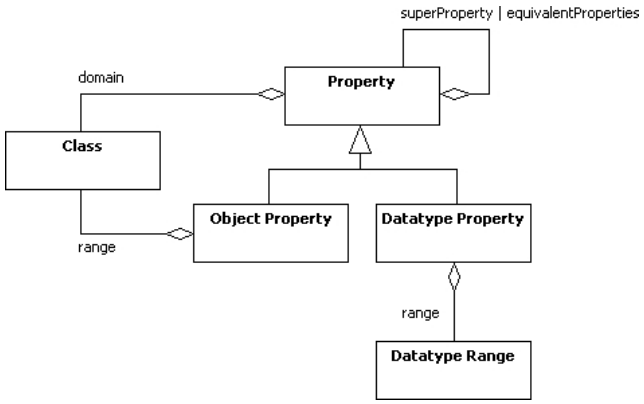

Figura 2.4. Modello semplificato delle proprietà in OWL

Classi e proprietà rappresentano i tipi di un dominio e le loro relazioni, costituendo la parte intensionale di un modello. Su questa si sviluppa la parte estensionale che invece definisce gli individui che istanziano le classi.

OWL permette di definire individui, vedi Fig. 2.5, che siano istanza di una o più classi, ciascuno caratterizzato da attributi semplici e relazioni verso altri individui.

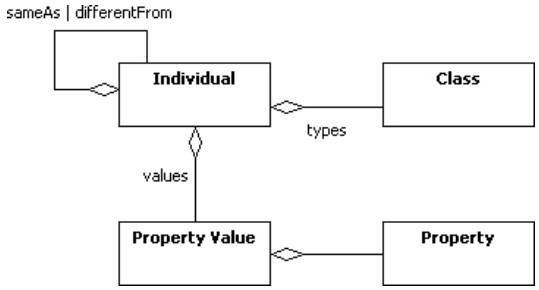

Figura 2.5. Modello semplificato degli individui in OWL

OWL fornisce tre sottolinguaggi dal potere espressivo crescente che sono progettati per l'utilizzo da parte di specifiche comunità di utenti:

- OWL Lite supporta quegli utenti che necessitano gerarchie di classificazione con vincoli semplici come ad esempio i thesaurus e le tassonomie.

- OWL DL fornisce supporto a quegli utenti che vogliono la massima espressività mantenendo la completezza computazionale (tutte le conclusioni sono computabili) e la decidibilità (tutte le computazioni termineranno in un tempo finito). OWL DL include tutti i costrutti del linguaggio OWL ma con alcune restrizioni (per esempio mentre una classe può essere sottoclasse di più classi, una classe non può essere istanza di un'altra classe). OWL DL è così chiamato per la sua corrispondenza con le logiche descrittive (Description Logic).

- OWL Full serve a quegli utenti che vogliono la massima espressività e la libertà sintattica di RDF senza garanzie sulla computabilità. Per esempio, in OWL Full una classe può essere trattata contemporaneamente come collezione di individui e come individuo essa stessa.

Ciascun sottolinguaggio è un'estensione del suo predecessore più semplice sia per quel che riguarda ciò che può essere legalmente espresso sia per quello che può essere validamente concluso.

Recentemente è stata rilasciata una nuova versione di OWL (OWL2) che estende la versione precedente, garantendo la piena compatibilità sia per quel che riguarda la semantica che l'espressività . In questo lavoro faremo comunque riferimento ad OWL 1, se non diversamente specificato, in quanto il numero di componenti software che lo supportano è ad oggi più numeroso.

2.1.3 SPARQL

SPARQL (acronimo ricorsivo di: SPARQL Protocol and RDF Query Language) [110] è un linguaggio di interrogazione per RDF reso raccomandazione dal W3C.

SPARQL può essere usato per esprimere interrogazioni su diverse fonti di dati: sia su quelle che nativamente rappresentano l'informazione con RDF, sia su quelle in cui l'informazione può essere vista come RDF soltanto attraverso un middleware. SPARQL permette la costruzione di query principalmente attraverso la definizione di pattern sui grafi (graph pattern) che possono essere

richiesti od opzionali e venire combinati con gli operatori booleani di congiunzione e disgiunzione. Le query SPARQL sono essenzialmente un insieme di *triple pattern* che è la forma più semplice di graph pattern. Un triple pattern è del tutto simile ad una tripla RDF eccetto il fatto che subject, predicate ed object possono essere variabili.

Un graph pattern è verificato da un sottografo di un modello RDF quando i termini estratti dal sottografo possono essere sostituiti alle variabili del graph pattern: il risultato sarà in quel caso un grafo equivalente al sottografo.

Ad esempio supponiamo di voler scrivere una semplice query per trovare il titolo di un libro dato un grafo di informazioni. Il listato 2.1 mostra il codice SPARQL corrispondente.

Listing 2.1. Esempio di query SPARQL

```
SELECT ?title
WHERE {
        <http://example.org/book/book1> <http://example.org/title> ?title .
}
```

L'interrogazione è costituita da due parti: la clausola SELECT che identifica le variabili che devono comparire nel risultato della query e la clausola WHERE che fornisce il graph pattern da verificare sul grafo dei dati. Il graph pattern è in questo esempio costituito da un solo triple pattern con una sola variabile (?title) nella posizione di object.

Supponiamo di eseguire tale query sul modello di dati RDF descritto nel listato 2.2.

Listing 2.2. Modello RDF su cui viene eseguita la query SPARQL

```
<http://example.org/book/book1> <http://example.org/title> ‘‘First example’’
```

Il risultato dell'esecuzione della query corrisponderà alle informazioni riportate in tabella 2.1.

title
“First Example”

Tabella 2.1. Risultato corrispondente all'esecuzione della query SPARQL

2.1.4 SWRL

SWRL (Semantic Web Rule Language) [111] è un linguaggio proposto dal W3C per la scrittura di regole di inferenza ed è stato ottenuto dalla combinazione di OWL-DL con il Rule Markup Language. SWRL estende il modello teorico di OWL aggiungendo la logica relativa alle clausole di Horn e rendendo possibile l'applicazione di regole di deduzione ed inferenza sui modelli ontologici. Ogni regola scritta con SWRL si presenta nella forma di implicazione tra un antecedente ed un conseguente e corrisponde all'affermazione che se le condizioni specificate nell'antecedente valgono allora anche quelle specificate nel conseguente devono valere.

Sia antecedente che conseguente consistono di zero o più atomi in congiunzione tra di loro, ciascuno dei quali farà riferimento a classi o proprietà del modello OWL cui la regola si riferisce. Antecedenti e conseguenti vuoti sono rispettivamente interpretati come trivialmente vero e trivialmente falso.

Ad esempio con SWRL sarà possibile scrivere una semplice regola per asserire che la combinazione delle proprietà hasParent e hasBrother implicano la relazione hasUncle, vedi listato 2.3.

Listing 2.3. Esempio di regola

```
hasParent(?x1,?x2) and hasBrother(?x2,?x3) -> hasUncle(?x1,?x3)
```

Tale regola corrisponderà all'espressione SWRL mostrata nel listato 2.4.

Da notare che alcune regole che si possono scrivere con SWRL corrispondono ad assiomi definibili direttamente con OWL e finirebbero quindi per essere un duplicato di OWL stesso.

Listing 2.4. Esempio di regola SWRL

```
Implies(
        Antecedent(
                hasParent(I−variable(x1) I−variable(x2))
                hasBrother(I−variable(x2) I−variable(x3)))
        Consequent(
                hasUncle(I−variable(x1)I−variable(x3))))
```

2.2 Architettura Ontologica

Gli sforzi della Semantic Web Initiative insieme all'avanzamento della pratica del software hanno portato alla nascita di numerosi standard e componenti che delineano un vero e proprio paradigma ontologico per la rappresentazione ed elaborazione della conoscenza. Tuttavia la sostanziale mancanza di indicazioni di livello architetturale per quel che riguarda la realizzazione di applicazioni che impieghino il paradigma ontologico limita fortemente la possibilità di questa tecnologia di affermarsi non solo nei laboratori di ricerca ma come approccio valido per la soluzione di problemi concreti. Abbiamo ritenuto quindi opportuno fare considerazioni architetturali riguardanti le applicazioni che impiegano attualmente le ontologie per arrivare successivamente alla descrizione di quella che chiamiamo Architettura Ontologica e che potrebbe essere propriamente considerata un pattern architetturale per la strutturazione di applicazioni.

Più in particolare chiamiamo Architettura Ontologica un'architettura software in cui vengono combinate la capacità di rappresentazione e classificazione delle ontologie con la capacità di elaborazione di un linguaggio ad oggetti ed in cui in particolare siano presenti almeno tre componenti:

- Domain Layer realizzato come modello ad oggetti che codifichi la logica applicativa del sistema e ne implementi le elaborazioni.
- Data Layer realizzato come modello ontologico responsabile di dare persistenza ai dati su cui opera la logica di dominio.

- Mapping Layer che superando l'impedance mismatch, cioè la distanza concettuale tra il modello ad oggetti e quello ontologico, consenta alla logica applicativa di operare sia sulla parte estensionale che su quella intensionale dei dati ontologici

In una architettura ontologica il Domain Layer può quindi delegare al Data Layer non solo la rappresentazione dei dati ma anche quella delle loro concettualizzazioni. Così facendo, diventa possibile generalizzare la logica applicativa in modo da adattarla non solo al variare delle istanze dei dati ma anche al variare dei concetti rispetto ai quali esse vengono descritte. Questo fornisce un mecccanismo per cambiare dinamicamente la struttura ed il comportamento di un sistema ed in ciò richiama il pattern architetturale Reflection [84].

2.3 Motivazioni per l'Architettura Ontologica

Nella sezione precedente è stata data la definizione di architettura ontologica mentre nella prossima ne verrà descritta dettagliatamente la struttura insieme alle giustificazioni che hanno portato a determinate scelte architetturali.

In questa sezione è invece opportuno riportare le motivazioni che giustificano lo sforzo di definire un'architettura ontologica e, in sintesi, i vantaggi che una tale architettura porta allo sviluppo di un software [44][98].

L'elenco di qualità positive riportato è da considerarsi non esaustivo ma comunque rappresentativo. Per ciascuna qualità verrà data una definizione e la spiegazione del perché si ritiene che l'architettura ontologica ne goda. Saranno inoltre forniti esempi esplicativi provenienti dall'area dell'helthcare perché molte delle sperimentazioni fatte si sono svolte proprio in questo ambito e per il motivo più generale che proprio l'area dell'informatica per la medicina ha sentito per prima e maggiormente la necessità di nuovi paradigmi di elaborazione della conoscenza e vede concentrati numerosi sforzi di ricerca relativi alle ontologie.

Nella presentazione dei vantaggi dell'architettura ontologica verranno talvolta attribuite ad essa le qualità positive delle ontologie. Ciò rappresenta una

semplificazione in quanto si ritiene che la transitività di tali caratteristiche non sia automatica ma in larga parte legata alla struttura dell'architettura ontologica stessa e che verrà presentata successivamente.

Le note relative alle ontologie mostrano che questa tecnologia serve principalmente alla rappresentazione della conoscenza e la definizione di architettura ontologica data mostra come le ontologie vengano impiegate per realizzare il Data Layer. Poiché in architetture software evolute ma comunque più tradizionali e consolidate questo layer viene solitamente realizzato per mezzo di database relazionali, nello spiegare i vantaggi dell'architettura ontologica verranno spesso confrontate le ontologie con i database relazionali.

Infine, nel presentare le qualità positive dell'architettura ontologica, si cercherà di mettere queste ultime in relazione con quelle del modello di qualità descritto dalla normativa ISO 9126 ritenendo in questa maniera di rendere la spiegazione più chiara e di agevolare il confronto dell'architettura ontologica con altre architetture evolute.

2.3.1 Espressività

Le ontologie hanno un alto potere espressivo intendendo con ciò sottolineare il fatto che permettono di codificare adeguatamente la conoscenza di qualunque dominio applicativo ed il fatto che operano ad un livello di astrazione maggiore rispetto ad altre tecnologie concorrenti.

Le ontologie mettono a disposizione costrutti che permettono ad esempio di rappresentare in maniera del tutto naturale legami di ereditarietà e relazioni, anche complesse, tra entità. In questo modo i modelli ontologici riescono a descrivere in maniera precisa, completamente formalizzata e priva di dettagli implementativi le entità e le relazioni del dominio applicativo.

Ad esempio, l'ontologia di Fig. 2.6 che viene presentata graficamente per mezzo di un class diagram UML [6] che rispetta le convenzioni stabilite in [8], permette di modellare il concetto di persona (`Person`) collegato ad un insieme di fattori di rischio (`RiskFactor`) di cui alcuni sono ereditari ed altri no. L'ontologia descrive anche aspetti quali il fatto che `HereditaryRiskFactor` e

`NotHereditaryRiskFactor` sono specializzazioni della classe `RiskFactor` tra loro disgiunte ed il fatto che le persone possono essere in relazione tra di loro attraverso i predicati `hasParent` ed `hasChild` e che i due predicati sono uno l'inverso dell'altro rendendo ad esempio possibile dedurre che `Mario` (istanza di `Person`) ha un figlio (`hasChild`)) che è `Giovanni` (istanza di `Person`) dall'asserzione che `Giovanni` ha un padre (`hasParent`) che è `Mario`.

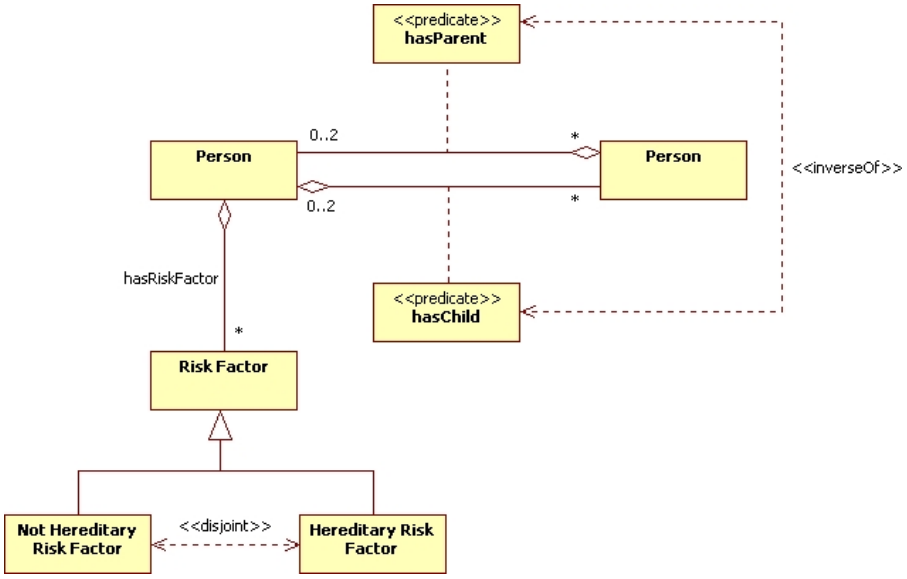

Figura 2.6. Modello per persone e fattori di rischio

Il corrispondente database relazionale avrebbe dovuto contenere le tabelle Person e Risk Factor insieme ad altre tabelle accessorie necessarie per modellare la gerarchia dei fattori di rischio con una strategia quale: Single Table Inheritance, Class Table Inheritance o Concrete Table Inheritance [36]. Sarebbero stati necessari campi identificativi (chiavi surrogate) per ogni tabella al fine di modellare le relazioni tra record e sarebbero state necessarie tabelle di laccio per definire i legami molti a molti tra persone. Non sarebbe stato facile rappresentare il fatto che i fattori di rischio ammettono due specializzazioni disgiunte tra di loro ed il fatto che i predicati avere padre ed avere figlio sono uno l'inverso dell'altro.

Realizzando il Data Layer per mezzo di un'ontologia l'applicazione avrà a disposizione un modello dei dati ricco e, non contenendo dettagli imple-

mentativi, sintetico e in grado, tra l'altro, di contenere informazioni quali la transitività o la riflessività delle relazioni tra entità, informazioni difficilmente rappresentabili attraverso i normali database relazionali.

Il processo realizzativo del software trae vantaggio dall'impiego delle ontologie [121]. Infatti l'alto livello di astrazione permette la condivisione dello stesso modello tra tutte le parti interessate al software siano essi tecnici o meno, riducendo in questa maniera il rischio di fraintendimenti tra sviluppatori e committenti. La possibilità di usare lo stesso formalismo sia durante la fase di analisi che quella di design consente di ottimizzare le risorse e ridurre il rischio di introduzione di bug.

La realizzazione di un Data Layer espressivo attraverso un processo produttivo che veda esperti di dominio ed informatici lavorare in maniera coesa attraverso strumenti condivisi finisce per incidere positivamente sulle seguenti caratteristiche: appropriatezza ed accuratezza del gruppo funzionalità; comprensibilità del gruppo usabilità; analizzabilità, modificabilità e testabilità del gruppo manutenibilità.

2.3.2 Interrogabilità

I modelli ontologici possono essere consultati per mezzo di linguaggi di interrogazione che consentono la formulazione, in maniera relativamente semplice, di espressioni di ricerca estremamente complesse che prendono il nome di query semantiche e che non sono disponibili nella stessa maniera per altre tecnologie concorrenti delle ontologie.

Molti linguaggi di interrogazione per ontologie, tra cui SPARQL, permettono di costruire espressioni di ricerca facendo riferimento ai modelli RDF dei modelli ontologici e rendendo in questa maniera semplice definire vincoli e legami tra entità differenti.

Supponiamo ad esempio di voler cercare i fattori di rischio di Mario Rossi nel modello ontologico di figura 2.6: l'espressione SPARQL corrispondente sarebbe quella mostrata nel listato 2.5.

Listing 2.5. SPARQL per cercare i fattori di rischio di Mario Rossi

```
PREFIX health: <http://www.ing.unifi.it/healt.owl#>
SELECT ?risk
WHERE {
        health:mario_rossi health:hasRiskFactor ?risk.
}
```

Nella clausola WHERE di tale espressione è possibile riconoscere la struttura di una tripla RDF (subject predicate object) nella quale: subject è il valore costante `<http://www.ing.unifi.it/health.owl#mario_rossi>`; predicate è il valore costante `<http://www.ing.unifi.it/health.owl#hasRiskFactor>`; ed object è la variabile `?risk` obiettivo della ricerca.

Supponiamo ancora di voler cercare i fattori di rischio del padre di Mario Rossi: l'espressione SPARQL corrispondente sarebbe quella mostrata nel listato 2.6.

Listing 2.6. SPARQL per cercare i fattori di rischio del padre di Mario Rossi

```
PREFIX health: <http://www.ing.unifi.it/healt.owl#>
SELECT ?risk
WHERE {
        health:mario_rossi health:hasParent ?parent.
        ?parent health:hasRiskFactor ?risk.
}
```

La semplicità del modello concettuale su cui si basano i linguaggi di interrogazione unita al fatto che i modelli ontologici non contengono dettagli implementativi rende le espressioni di query più compatte e facili da leggere rispetto a quelle equivalenti scritte (ad esempio) con SQL.

Con i linguaggi di interrogazione per ontologie si riescono inoltre a formulare espressioni di query che sarebbero invece molto difficili se non impossibili da

scrivere con SQL perché relative allo schema concettuale e non alle sue istanze oppure perché contenenti legami tra istanze parzialmente definiti.

Ad esempio, necessitano di espressioni di interrogazione di questo tipo: la ricerca delle relazioni che legano le persone ad altre entità oppure la ricerca di quelle coppie di persone che siano legate dallo stesso tipo di relazione che lega Mario e Giovanni.

Le espressioni SPARQL corrispondenti sarebbero invece rispettivamente quelle dei listati 2.7 e 2.8.

Listing 2.7. SPARQL per cercare i predicati applicabili alle persone

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX health: <http://www.ing.unifi.it/healt.owl#>
SELECT ?predicate
WHERE {
        ?predicate rdfs:domain healt:Person.
}
```

Listing 2.8. SPARQL per cercare coppie di persone come Mario e Giovanni

```
PREFIX health: <http://www.ing.unifi.it/healt.owl#>
SELECT ?p1, ?p2
WHERE {
        health:mario_rossi ?relation healt:giovanni_bianchi.
        ?p1 ?relation ?p2.
}
```

Realizzando il Data Layer per mezzo di un'ontologia l'applicazione potrà interrogare i dati con query semantiche molto espressive e difficilmente rappresentabili con altri linguaggi di interrogazione, consentendo in definitiva la realizzazione di funzionalità maggiormente rispondenti alle esigenze degli utenti.

Più in generale rendere disponibile al Domain Layer la possibilità di interrogare i dati del Data Layer per mezzo di query semantiche incide positivamente sulle seguenti caratteristiche: appropriatezza ed accuratezza del gruppo funzionalità; comprensibilità ed operabilità del gruppo usabilità; analizzabilità, modificabilità e testabilità del gruppo manutenibilità.

2.3.3 Ragionamento Deduttivo

I modelli ontologici rappresentano la conoscenza in un modo tale da consentire ad appositi componenti software (i reasoner) la derivazione di nuova conoscenza da quella direttamente asserita, applicando opportune regole di ragionamento. Tali deduzioni possono essere impiegate col duplice scopo di estendere la base di conoscenza disponibile per interrogazioni ed elaborazioni e per validare la base di conoscenza stessa.

Le ontologie appartengono al gruppo di formalismi delle logiche descrittive e sono quindi riconducibili alla logica del primo ordine. In particolare OWL nella versione OWL-DL corrisponde ad un frammento della logica del primo ordine completo e decidibile. Le inferenze ed i ragionamenti sulle ontologie corrispondono a trovare nel modello ontologico le conseguenze implicite rispetto a quelle esplicite rappresentate.

Ad esempio, considerando di nuovo il modello di Fig. 2.6 sarebbe possibile dedurre che Giovanni è padre di Mario dai fatti che Giovanni ha un figlio che è Mario e che avere padre è la relazione inversa di avere figli.

Numerosi studi tentano di estendere il potere espressivo di OWL mantenendo la completezza e la decidibilità dei ragionamenti. Il linguaggio SWRL, ad esempio, estende la semantica di OWL aggiungendo a questo le clausole di Horn. Utilizzando SWRL insieme ad OWL aumenta il numero di ragionamenti e deduzioni sulla base di conoscenza. Ad esempio diventa possibile specificare la regola che se una persona ha fattori di rischio ereditari allora anche i suoi figli devono averli. Applicando tale regola sarebbe poi possibile dedurre che Mario è affetto da un determinato fattore di rischio dai fatti che Mario è figlio di Giovanni e Giovanni è affetto dallo stesso fattore di rischio. La formula-

zione della regola precedente per mezzo di SWRL corrisponde all'espressione mostrata nel listato 2.9.

Listing 2.9. SWRL per la deduzione dei fattori di rischio ereditari

```
Implies(
        Antecedent(
                hasRiskFactor(I−variable(x) I−variable(y))
                HereditaryRiskFactor(I−variable(y))
                hasChild(I−variable(x) I−variable(z)))
        Consequent(
                hasRiskFactor(I−variable(x)I−variable(y))))
```

Realizzando il Data Layer per mezzo di un'ontologia diventa possibile applicare ai dati dell'applicazione regole di ragionamento, addirittura specificabili a run-time, per dedurre nuova conoscenza da quella asserita ed effettuare in modo automantico controlli di validità e consistenza. In questo modo il Domain Layer potrà disporre di un insieme più ricco e corretto di dati e quindi offrire agli utenti servizi migliori od alternativamente offrire servizi necessitando di un insieme minore di input.

La possibilità di applicare regole di ragionamento ed inferenza ai dati finisce per incidere positivamente sulle seguenti caratteristiche: appropriatezza ed accuratezza del gruppo funzionalità; apprendibilità, operabilità e attrattiva del gruppo usabilità; analizzabilità, modificabilità e testabilità del gruppo manutenibilità; adattabilità del gruppo portabilità.

2.3.4 Interoperabilità RDF

Le ontologie permettono di federare sorgenti di dati differenti al fine di realizzare basi di conoscenza distribuite.

I modelli ontologici OWL possono essere trattati, ad un livello di astrazione più basso, come modelli RDF che a loro volta, indipendentemente dai concetti rappresentati, possono esser trasformati in file XML, facili da archiviare e

trasportare. Con RDF inoltre, dati due modelli distinti, è possibile costruire un nuovo modello, unione dei precedenti, semplicemente unendo le triple dei modelli di partenza. Diventa molto semplice per un'applicazione raccogliere file RDF che rappresentano porzioni distribuite di una base di conoscenza ed unirli per ottenere, attraverso l'unione dei vari frammenti, la base di conoscenza complessiva. Si pensi invece a quanto è macchinoso unire le informazioni contenute in database diversi.

Ad esempio, vedi Fig. 2.7, supponiamo che le informazioni cliniche relative ai pazienti siano distribuite tra due sottosistemi (`Sys-1` e `Sys-2`) e supponiamo di voler realizzare un portale (`Portal`) che sia in grado di mostrare per ogni paziente l'informazione complessiva. A seguito della richiesta di un utente, il portale potrà interrogare i sottosistemi ottenendo da ciascuno un frammento di informazione (`data-1.rdf` e `data-2.rdf`), frammenti che poi unirà per restituire all'utente l'informazione complessiva.

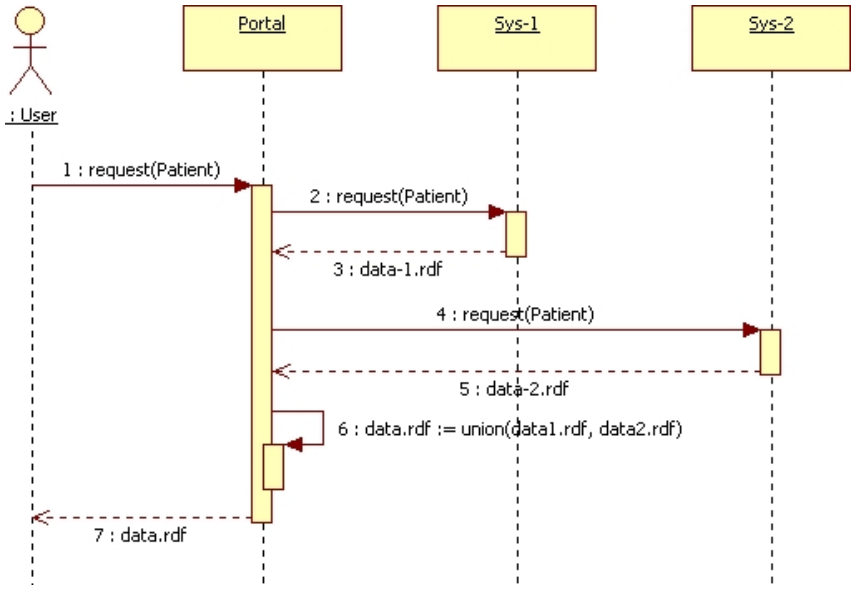

Figura 2.7. Componente RDF

Realizzando il Data Layer per mezzo di un'ontologia e quindi con RDF, si rendono disponbili all'applicazione le funzionalità e la flessibilità di quest'ultimo. Queste potenzialità di RDF sono particolarmente utili nella realizzazione

di applicazioni che operino su basi di conoscenza distribuite o collaborino con altre applicazioni attraverso architetture software evolute quali Publisher and Subscriber, Message Bus e SOA.

La realizzazione del Data Layer per mezzo di RDF finisce per incidere positivamente sulle seguenti caratteristiche architetturali: interoperabilità del gruppo funzionalità.

2.3.5 Interoperabilità OWL

Le ontologie permettono la realizzazione di veri e propri framework per la modellazione ed il riuso di basi di conoscenza costruite in modo distribuito e collaborativo.

Ciascun modello OWL è identificato per mezzo di una URI attraverso la quale può essere riferito per venire importato all'interno di altri modelli OWL. Questo vuol dire che quando si costruisce un'ontologia si può fare riferimento ad altre ontologie già esistenti che, a loro volta, possono fare riferimento ad altre ontologie ancora. Importare un'ontologia in un modello significa aggiungere al modello tutte le definizioni relative a classi, proprietà ed istanze, contenute nell'ontologia importata. Le informazioni importate sono del tutto indistinguibili da quelle direttamente definite nel modello se non per il fatto che sono necessariamente in sola lettura al fine di mantenere la responsabilità delle eventuali modifiche al modello che realmente le ha definite. Le classi e le proprietà importate possono comunque essere istanziate od estese e gli oggetti riferiti. Attraverso il meccanismo dell'importazione la conoscenza può essere organizzata in gerarchie di moduli OWL legati tra di loro, in cui ciascun modulo ha finalità e può riferirsi ad ambiti delimitati e precisi.

Supponiamo ad esempio, vedi Fig. 2.8, di voler costruire un'ontologia (`Master Patient Record Ontology`) su cui basare l'applicazione di cartella clinica di un ospedale. Supponiamo inoltre che esistano già le ontologie: `Person Ontology` che definisce le caratteristiche e le relazioni tra persone; `Pathology Ontology` che descrive le patologie in ambito medico; `Patient Ontology` che estendendo `Person Ontology` descrive per le persone anche que-

gli aspetti rilevanti per un ospedale (ad esempio i dati amministrativi delle ammissioni e dimissioni); `Social Network Ontology` che estendendo `Person Ontology` aggiunge alle persone quelle informazioni tipiche delle applicazioni di social networking. Sarà possibile costruire `Master Patient Record Ontology` importando le informazioni contenute nei moduli: `Pathology Ontology` e `Patient Ontology` che a sua volta importerà le definizioni contenute in `Person Ontology`. Occorre notare che con questo schema di dipendenze `Master Patient Record Ontology` non importerà le definizioni di `Social Network Ontology` mantenendosi maggiormente centrata sugli argomenti per i quali è costruita.

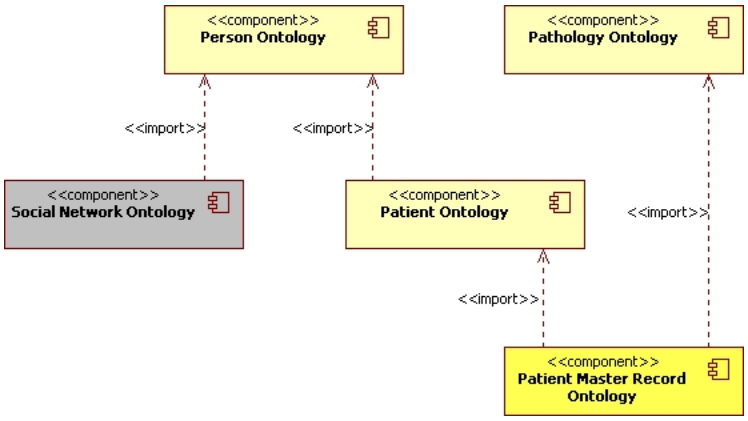

Figura 2.8. Esempio di gerarchia di importazioni tra modelli ontologici

La struttura modulare delle ontologie OWL è legata all'intento di realizzare moduli riusabili e condivisi per abilitare l'interoperabilità tra applicazioni a livello semantico. OWL di per sé garantisce l'interoperabilità solo a livello sintattico. Supponiamo di costruire un'ontologia importandone due esistenti nelle quali siano state definite rispettivamente le classi: Persona e Person per rappresentare di concetto di persona. La nuova ontologia conterrà la definizione di entrambe le classi e sarà perfettamente valida. Non esiste però alcun modo automatico per rilevare che le due classi servono a rappresentare lo stesso concetto. OWL infatti mette semplicemente a disposizioni il costrutto `equivalentTo` per consentire ad un utente di esplicitare le equivalenze che manualmente ha rilevato. Affinché applicazioni differenti condividano la stessa

semantica è necessario che esse operino su modelli condivisi. Numerosi sono oggi gli sforzi da parte della comunità scientifica volti a realizzare o potenziare gli strumenti teorici e pratici per il mapping tra ontologie e per arrivare alla definizione di robuste ontologie fondazionali, cioè di ontologie contenenti concetti basilari e generali cui riferire le ontologie di dominio e quelle applicative [43]. La diffusione di ontologie pubbliche e standardizzate, peraltro già disponibili per alcuni domini, inciderà significativamente sulla possibilità per le applicazioni di integrarsi a livello semantico ed è a questa diffusione che è fortemente legata la possibilità di affermarsi del Semantic Web.

Supponiamo che l'ontologia `Master Patient Record Ontology` dell'esempio precedente sia pubblica e standard. Ogni programma di cartella clinica basato su di essa riuscirà automaticamente a condividere le informazioni della sua base di conoscenza con gli altri applicativi.

Realizzando il Data Layer di un'applicazione per mezzo di modelli ontologici si riesce ad organizzare la base di conoscenza attraverso una gerarchia di moduli coesi e riusabili. Inoltre è possibile fare riferimento a modelli standard abilitando l'interoperabilità semantica e diminuendo il carico di lavoro legato all'analisi del problema. Diventa infatti possibile progettare la base di conoscenza estendendo soluzioni già validate e diffuse anziché partendo da zero.

La strutturazione del Data Layer attraverso moduli specializzati e collegati permette la strutturazione della conoscenza in modo che questa possa: essere estesa, evolvere nel tempo, aderire a standard e convenzioni. Anche il processo realizzativo del software trae vantaggio da una tale organizzazione della conoscenza perché è favorito il riuso e sono potenzialmente alleggerite le fasi di analisi. Tutto ciò finisce per incidere positivamente sulle seguenti caratteristiche architetturali: appropriatezza, interoperabilità e conformità del gruppo funzionalità; analizzabilità, modificabilità, testabilità del gruppo manutenibilità; adattabilità, conformità e sostituibilità del gruppo portabilità.

2.3.6 Reflection

Le ontologie abilitano la realizzazione di architetture software orientate al cambiamento, in grado cioè di modificare in maniera agevole e dinamica aspetti fondamentali del sistema quali la struttura dei tipi ed il comportamento. Le ontologie agevolano quindi l'implementazione del pattern Reflection descritto in [84]. I modelli ontologici, ad un livello di astrazione diversa, possono essere gestiti come modelli RDF per i quali la manipolazione di triple riguardanti classi, proprietà od istanze è del tutto indistinguibile. Diventa quindi parimenti semplice per un'applicazione accedere o modificare le istanze di una concettualizzazione ed accedere o modificare la concettualizzazione stessa.

Supponiamo ad esempio di realizzare il software per la gestione di una cartella clinica. In tale ambito è molto difficile riuscire ad esplicitare completamente il modello durante la fase di analisi perché le conoscenze mediche sono molto ampie ed evolvono in fretta. Adoperando le ontologie sarebbe però in ogni momento possibile aggiungere al modello nuove classi, ad esempio di sintomi o patologie.

Con i database relazionali invece, mentre è semplice consultare od elaborare record, non è altrettanto semplice aggiungere o rimuovere tabelle oppure modificare la loro struttura.

Ai modelli OWL, inoltre, si possono applicare regole, ad esempio SWRL, per derivare nuova conoscenza od eseguire validazioni ed è proprio all'applicazione di regole specificate a run-time che si può immaginare di delegare alcune delle funzionalità dell'applicazione.

Ad esempio, sempre nella realizzazione di un software di cartella clinica si potrebbe immaginare di adoperare un insieme di regole, specificate da specialisti medici, per l'individuazione delle patologie che potenzialmente affliggono un paziente dato un preciso quadro clinico.

Infine, solo per citare un altro scenario di modificabilità dinamica del comportamento di un'applicazione, per mezzo di OWL si può dare una descrizione ontologica di un insieme di web-servicies [107], descrizione che può venire adoperata: per abilitare l'utilizzo di query semantiche per la scelta del servizio

da invocare oppure per costruire automaticamente composizioni di più web servicies attraverso strategie di planning.

Realizzando il Data Layer di un'applicazione per mezzo di un'ontologia si ottiene più facilmente rispetto all'impiego di altre tecnologie che il Domain Layer dell'applicazione sia altamente configurabile e modificabile.

L'utilizzo di ontologie per la realizzazione del Data Layer fornisce una leva formidabile per affrontare i requisiti di quelle applicazioni che operano in ambiti che per loro natura mutano nel tempo, che sono soggetti a sviluppo incrementale, distribuito e spesso concorrente. Incidendo in modo positivo sulle seguenti caratteristiche architetturali: appropriatezza ed accuratezza del gruppo funzionalità; operabilità del gruppo usabilità; modificabilità del gruppo manutenibilità; adattabilità del gruppo portabilità.

2.4 Implementazione dell'Architettura Ontologica

In questa sezione verrà presentata la struttura dell'Architettura Ontologica proposta insieme alle argomentazioni che hanno portato alle scelte progettuali prese. La descrizione dell'architettura avverà in modo incrementale per passi successivi ed al fine di rendere più chiara l'esposizione, l'Architettura Ontologica verrà descritta confrontandola con altre architetture più tradizionali da cui trae spunto e di cui rapprensenta un'evoluzione.

Nel fare valutazioni ed esempi ci si riferirà all'insieme, piuttosto maturo e diffuso, di standard e raccomandazioni proposti dal W3C (RDF, OWL, SPRQL, SWRL) con la precisazione che molte delle considerazioni continuerebbero a valere pur adottando tecnologie ontologiche differenti.

In Fig. 2.9 viene mostrata la struttura, attualmente più diffusa, per le Enterprise Application. Sono distinguibili: un `Data Layer` che si occupa dell'accesso ai dati e della persistenza delle informazioni; un `Domain Layer` che implementa la logica applicativa; ed un `Presentation Layer` che implementa le interfacce e la logica per l'interazione con gli utenti. In un'architettura a Layer ogni strato offre servizi quello immediatamente superiore impiegando le

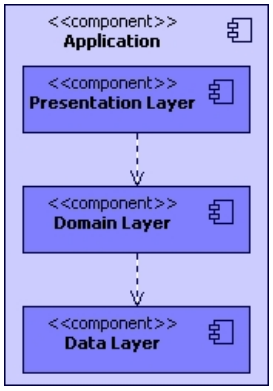

Figura 2.9. Architettura 3 Layers

funzionalità di quello immediatamente sottostatne. In partolcolare per l'architettura mostrata il `Presentation Layer` fornisce funzionalità agli utenti adoperando quelle del `Domain Layer` che a sua volta utilizza quelle del `Data Layer`.

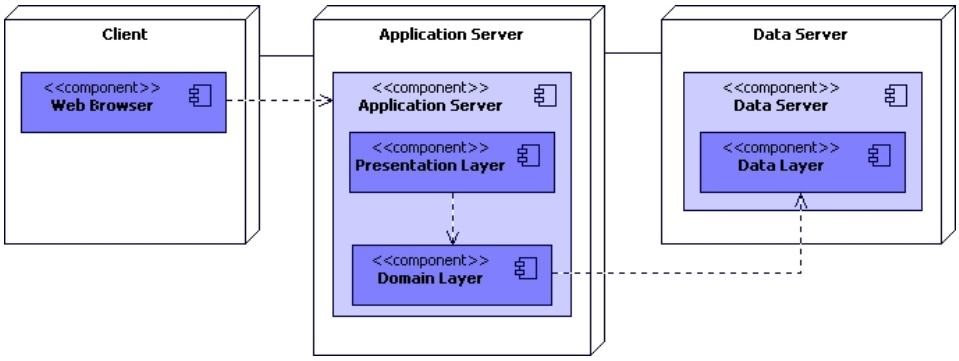

Figura 2.10. Esempio di architettura 3-Tier

È molto frequente, per le odierne applicazioni, che i diversi layer dell'applicazione vengano installati ed e fatti funzionare su computer differenti arrivando all'architettura N-Tier di cui è possibile vedere un esempio, per una tipica applicazione web, in Fig. 2.10.

Dell'architettura mostrata in Fig. 2.10, occorre considerare, per da discussione seguente, il frammento mostrato in Fig. 2.11 contente il `Domain Layer` ed il `Data Layer` insieme al componente che lo gestisce: il `Data Server`.

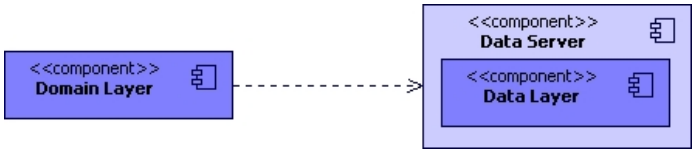

Figura 2.11. Frammento di dell'architettura 3-Tier

Nella descrizione dell'Architettura Ontologica infatti ci soffemeremo sulla descrizione di come il `Domain Layer` ed id `Data Layer` debbano essere implementati; di come possano interagire tra di loro; e di quali componenti software realizzino il `Data Server`.

Ad esempio, per un'applicazione sviluppata con Java che adoperi un database MySql attraverso JDBC, l'architettura corrisponderebbe più precisamente a quelle mostrata in Fig. 2.12, in cui viene evidenziato che: il `Data Layer` è realizzato per mezzo di oggetti java (Pojo); il `Data Server` è costituito dal server MySql; ed il `Data Layer` è un database gestito da MySql e accedibile attraverso l'intefaccia applicativa `JDBC`.

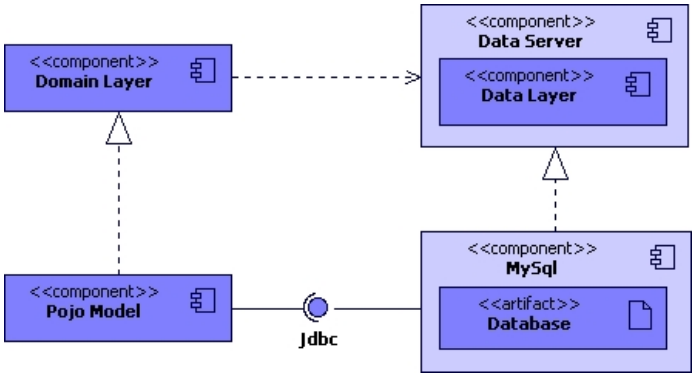

Figura 2.12. Esempio di architettura per un'applicazione Java che usa un database Mysql

In primo passo per arrivare alla descrizione della struttura dell'Architettura Ontologica consiste nella precisazione che questa rappresenta una specializzazione ed una evoluzione del frammento mostrato in Fig. 2.11. Tale precisazione serve ad evidenziare che i linguaggi di modellazione ontologica, pur consentendo di rappresentare la conoscenza, non sono linguaggi di programmazione. OWL non mette a disposizione costrutti per la definizione di metodi e fun-

zioni, né per l'accesso alle periferiche e nemmeno più semplicemente permette la definizione e la manipolazione di variabili. OWL è un formalismo di tipo dichiarativo che difficilmente si presta alla codifica di algoritmi.

Ad esempio si potrebbe costruire un'ontologia contente le classi necessarie a descrivere il concetto di `Misurazione` e di `Unità di Misura`, insieme a quelli di `Gradi Celsius` e `Gradi Fahrenheit`. Ma sarebbe difficile, senza ricorrere a qualche artificio, riuscire a descrivere la semplice formula di conversione di temperature tra le due unità di misura.

Risulta a questo punto evidente che le ontologie non possono essere nient'altro che uno dei componenti necessari alla realizzazione di un'applicazione ed in particolare il loro ruolo è relativo al componente `Data Layer` mentre servirà ricorrere a normali linguaggi di programmazione per realizzare la logica applicativa e quindi il `Domain Layer`.

Il passo successivo nella descrizione dell'Architettura Ontologica consiste nell'osservare, vedi Fig. 2.13, che il componente `Data Server` è in questo caso realizzato da Repository Ontologici (`Ontological Repository`).

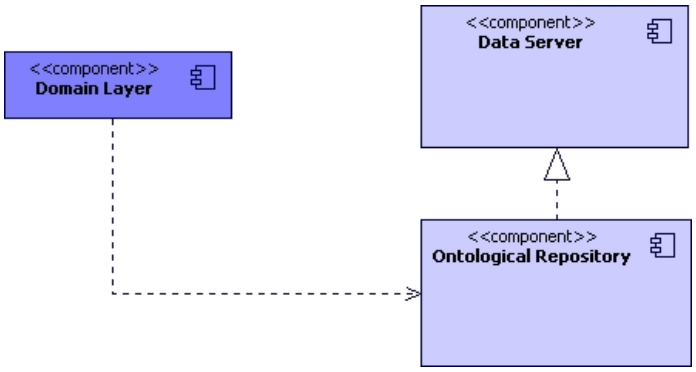

Figura 2.13. Data Server realizzato per mezzo di un Ontological Repository

Nella discussione fatta fino ad ora riguardo alle ontologie ci si è concentrati sulle loro caratteristiche e sulle loro potenzialità ma niente è stato detto riguardo a quali sono i componenti software che servono a manipolarle. È l'Ontological Repository, che archiviando i modelli OWL, spesso in maniera più efficiente rispetto a semplici file XML, riesce ad esempio ad eseguire

le query SPARQL oppure, se provvisto di reasoner, a calcolare le deduzioni corrispondenti all'applicazione di un insieme di regole SWRL.

In Fig. 2.14, viene mostrata la struttura interna tipica di un'`Ontological Repository` in cui sono riconoscibili: l'`RDF Repository`, un `OWL/SWRL Reasoner` e la `Repository API`. Vale la pena di osservare che l'`RDF Repository` può archiviare i modelli RDF: in strutture dati specifiche, in database relazionali oppure fare il mapping di database legacy. Si osservi inoltre che SWRL ammette una rappresentazione in OWL delle sue regole che pertanto possono, al pari dei modelli ontologici, venire archiviate all'interno dell'`RDF Repository`.

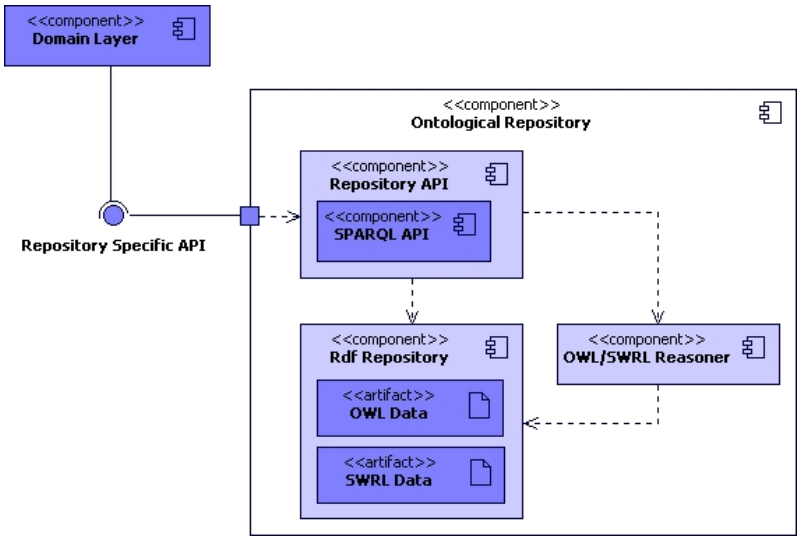

Figura 2.14. Struttura interna degli Ontological Repository

La Fig. 2.14 corrisponde all'architettura attualmente più comune per le applicazioni che facciano uso di ontologie: la logica applicativa, scritta con qualche linguaggio di programmazione, accede alle funzionalità dell'`Ontological Repository` attraverso l'interfaccia applicativa di programmazione fornita dal repository stesso.

Ad oggi sono disponibili, spesso in open source, numerosi `Ontological Repository`, ad esempio, solo per citarne alcuni: Jena [17], Sesame [1], Mulgara [72], Protégé [91], Kaon [7].

Purtroppo la crescente maturità degli standard sulle ontologie non corrisponde ancora ad un'equivalente maturità degli `Ontological Repository`. La differenza di servizi offerti, la mancanza di standardizzazione delle interfacce applicative e le prestazioni non sempre eccellenti, rendono la scelta dell'`Ontological Repository` una decisione estremamente critica per la realizzazione di un'applicazione ontologica.

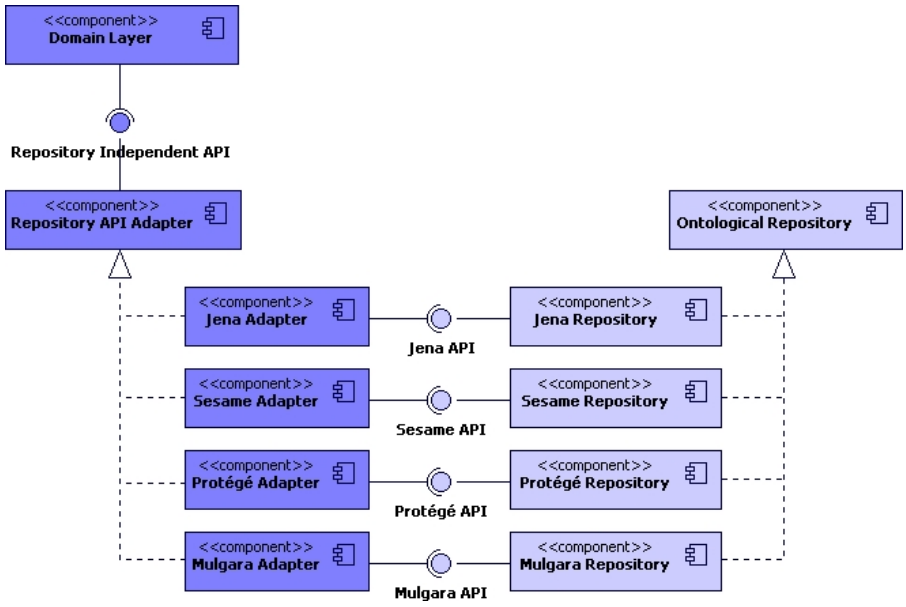

Figura 2.15. Repository API Adapter

Nell'Architettura Ontologica che proniamo, vedi Fig. 2.15, è presente un componente che svolgendo il ruolo di Adapter, permette di astrarre l'`Ontological Repository` concretamente impiegato consentendo di sostituire un'implementazione con un'altra senza bisogno di modificare il `Domain Layer`.

Le considerazioni architetturali fin qui svolte portano a definire il frammento di Architettura Ontologica mostrato in Fig. 2.16.

Occorre considerare di nuovo il modello descritto in Fig. 2.13 ed approfondire i dettagli architetturali relativi al `Domain Layer`.

La pratica attuale dell'ingegneria del software ha evidenzionato che è utile organizzare la logica applicativa attraverso modello ad oggetti al fine di rendere gestibili, da parte degli sviluppatori, anche problemi applicativi di

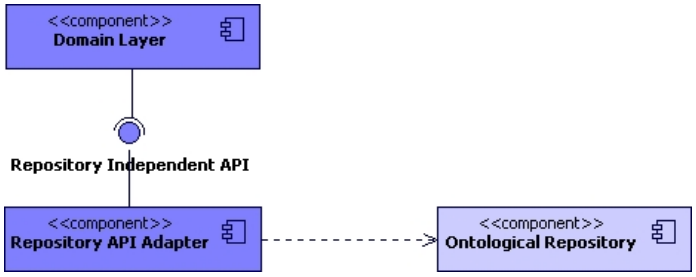

Figura 2.16. Architettura n-tier

grandi dimensioni. Suddividendo le responsabilità di un software in oggetti autosufficienti ed individualmente riusabili, ciascuno contenente i dati ed il comportamento rilevante per l'oggetto, si riducono, in questo modo, le possibità di errore ed i tempi di sviluppo. Da notare che molti `Ontological Repository` offrono, purtroppo, interfacce di programmazione che, pur essendo scritte con linguaggi di programmazione ad oggetti, impongono uno stile di programmazione di tipo procedurale.

Nell'Architettura Ontologica si ritiene utile organizzare la logica applicativa attraverso l'uso di modelli ad oggetti che abilitano tra l'altro l'impiego dei numerosi design pattern esistenti [39]. Si osservi che molti `Ontological Repository` forniscono un'API di programmazione che impone l'adozione di modelli anemici per la manipolazione dei dati [36].

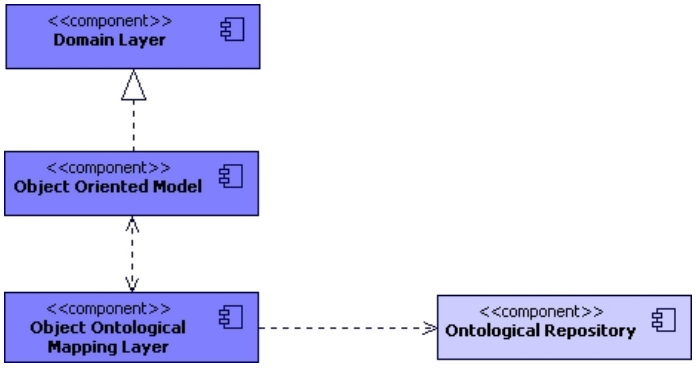

Figura 2.17. Architettura n-tier

Per trasportare dati tra modelli ad oggetti e database relazionali sono descritti in letteratura numerosi approcci e pattern [36]. Attingendo da questi e

generalizzandone l'uso pare conveniente adottare il pattern DataMapper [36] per far colloquiare il `Data Layer` ontologico col `Domain Layer` ad oggetti. Occorre rilevare che molte delle librerie attualmente esistenti per questo scopo, ad esempio Protégé [91], applicano invece un approccio riconducibile ad Active Record [36] imponendo in questa maniera di "sporcare" il modello di dominio con porzioni di codice riguardanti la persistenza.

Si arriva dunque al frammento di Architettura Ontologica descritto in Fig. 2.17

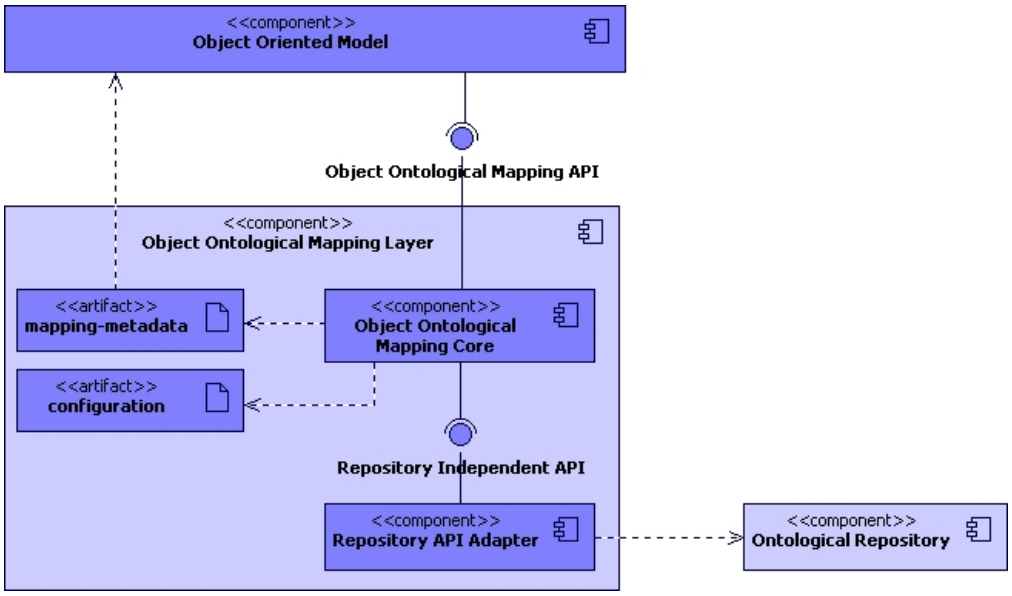

Figura 2.18. Architettura n-tier

Riconciliando i due frammenti di architettura si ottiene la schematizzazione mostrata in Fig. 2.18 nella quale è possibile individuare: il componente `Object Oriented Model` che realizza il `Domain Layer` come modello ad oggetti e che si interfaccia con l'`Object Ontological Mapping Layer` e più in particolare con il componente `Object Ontological Mapping Core` attraverso l'interfaccia applicativa di programmazione `Object Ontological Mapping API`. Si può osservare inoltre che il componente `Object Ontological Mapping Core` è configurabile attraverso attraverso i parametri contenuti in `mapping-metadata` e `configuration`. `Object Ontological Mapping Core` interagisce

con `Ontological Repository` e quindi con il `Data Layer` ontologico attraverso il componente `Repository API Adapter` con il quale colloquia per il tramite della `Repository Independent API`.

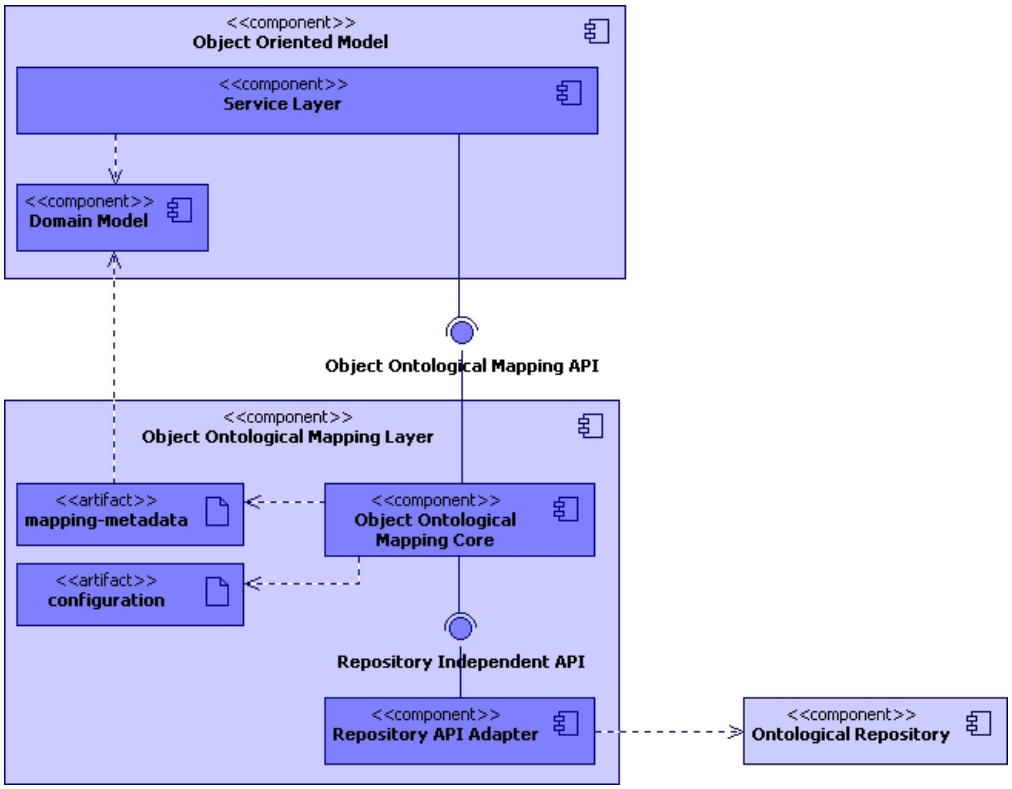

Figura 2.19. Struttura dell'Architettura Ontologica

Infine, in Fig. 2.19, il componente `Object Oriented Model` viene suddiviso in due sotto-componenti: il `Domain Model`, composto da oggetti con interfacce a granularità fine e totalmente indipendente dalla logica di persistenza; ed il `Service Layer`, composto da oggetti con interacce a grana grossa che compongono i servizi del `Domain Model` con quelli di persistenza dell'`Ontological Mapping Core`.

L'Architettura Ontologica proposta è fortemente caratterizzata dal componente `Object Ontological Mapping Layer` che sarà descritto in maggior dettaglio e di cui verrà descritta un'implementazione di riferimento nel prossimo capitolo.

Capitolo 3

Object Ontology Mapping

In questo capitolo verrà presentato Loom, cioè la libreria che è stata progettata ed implementata per realizzare il componente di Object Ontology Mapping individuato dalle considerazione sull'architettura ontologica.

Il problema di mettere in comunicazione un Domain Layer object-oriented con un Data Layer ontologico, è stato affrontato da vari autori. In [40] viene confrontata la modellazione object-oriented con quella ontology-oriented, evidenziando le considerevoli sovrapposizioni come pure le significative differenze. Viene inoltre investigato il problema di mappare le classi di un'ontologia nelle classi di linguaggi di programmazione object-oriented. In [57] un modello di dominio ontologico viene manipolato attraverso componenti Java realizzati impiegando Protégé [91] e Jena [17] (vedere anche [56] e [58]). La trasformazione di un'ontologia RDF Schema in familiari oggetti Java è descritta in [104]. In [54] le classi di OWL sono fatte corrispondere ad interfacce e classi Java. In [2] viene descritto un framework che fa corrispondere i costrutti ontologici a componenti EJB resi persistenti attraverso uno strato di object-relational mapping scritto con Hibernate. Ad oggi sono inoltre disponibili sul web un

numero di software per l'object-ontology mapping [20], solo per citarne alcuni si ricordano: Jenabean [18], Owl2Java [120] e So(m)mer [3] .

Con Loom, tenendo presenti le numerose pubblicazioni sul tema dell'objet-ontology mapping, si è cercato di recuperare dalle librerie esistenti gli spunti migliori e, aggiungendo inoltre nuove funzionalità e caratteristiche, si è arrivati a definire un framework che offre alcuni rilevanti benefici.

In particolare Loom:

- garantisce l'indipendenza del Domain Layer dal Data Layer: è infatti possibile cambiare il Repository ontologico impiegato dall'applicazione semplicemente modificando le impostazioni in un file di configurazione;
- permette di tenere la logica applicativa separata da quella per la persistenza: le informazioni per il mapping sono archiviate in file .xml esterni al codice del modello ad oggetti, consentendo la definizione di più strategie di mapping per i medesimi oggetti e permettendo la realizzazione di un'applicazione più modulare;
- il meccanismo di mapping è estremamente flessibile lasciando ampi margini di scelta all'utente in modo da rendere semplice l'applicazione di pattern complessi come Reflection.

Di seguito verrà descritto come configurare Loom per adoperarlo in un'applicazione, come utilizzare i file di mapping per stabilire una corrispondenza tra modello ontologico e modello ad oggetti, come impiegare la API per svolgere le operazioni relative alla persistenza (lettura e scrittura di oggetti) ed infine saranno date alcune informazioni relativamente alla sua struttura interna.

3.1 Configurazione di Loom

Per utilizzare Loom all'interno di un'applicazione è necessario aggiungere il file `loom.jar` al progetto e configurare i file: `persistence.xml`, `repository.xml` ed `oom-mapping.xml`.

Il file `persistence.xml`, che può in realtà avere un nome diverso a patto che continui a rispettare le specifiche di `persistence-unit.xsd`, serve ad indicare quale configurazione di repository e quali file di mapping l'applicazione dovrà adoperare.

Listing 3.1. Esempio di file persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence-unit
        xmlns="http://www.ing.unifi.it/loom"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.ing.unifi.it/loom persistence-unit.xsd"
        base-path="/home/myapp">


        <!-- Indicazione del repository da adoperare -->
        <rdf-persistence-unit ref="/resources/repository.xml"/>


        <!-- Indicazione dei file di mapping da adoperare -->
        <oom-mappings ref="/resources/oom-mappings1.xml"/>
        <oom-mappings ref="/resources/oom-mappings2.xml"/>

        ...
        <oom-mappings ref="/resources/oom-mappings2.xml"/>

</persistence-unit>
```

Il listato 3.1 mostra un esempio di file `persistence.xml` in cui viene specificato di adoperare per l'applicazione la configurazione del repository descritta nel file `/home/myapp/resources/repository.xml` e come risorse contenenti informazioni di mapping i file: `/home/myapp/resources/oom-mappings1.xml`, `/home/myapp/resources/oom-mappings2.xml` e `/home/myapp/resources/oom-`

`mappings3.xml`. Si osserva che l'indicazione della configurazione di repository è unica mentre possono venire specificate più risorse di mapping.

Il file `repository.xml`, che può chiamarsi anche in altro modo ma che comunque deve rispettare lo schema stabilito in `rdf-persistence-unit.xsd`, serve ad impostare le caratteristiche del repository ontologico che l'applicazione dovrà impiegare.

Listing 3.2. Esempio di file repository.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf-persistence-unit
        xmlns="http://www.ing.unifi.it/loom"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.ing.unifi.it/loom rdf-persistence-unit.xsd">

        <sesame>
                <driverClass>org.gjt.mm.mysql.Driver</driverClass>
                <database>jdbc:mysql://localhost:3306/sesamedb</database>
                <user>sesame_user</user>
                <password>sesame_password</password>
        </sesame>

</rdf-persistence-unit>
```

Nel listato 3.2 viene mostrato un esempio di file `repository.xml` che descrive la configurazione necessaria affinché l'applicazione possa impiegare un repository ontologico Sesame che renda persistente l'informazione all'interno del database MySql `jdbc:mysql://localhost:3306/sesamedb` accedibile dall'utente `sesame_user` con la password `sesame_password`.

I file `oom-mapping.xml`, che possono avere anche altro nome ma che devono rispettare lo schema stabilito in `oom-mappings.xsd`, servono a descrivere il mapping tra modello ad oggetti e modello ontologico cioè l'insieme di cor-

rispondenze tra classi, attributi ed istanze del modello ad oggetti con classi, predicati ed individui del modello ontologico.

Listing 3.3. Esempio di file repository.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<oom-mappings
        xmlns="http://www.ing.unifi.it/loom"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.ing.unifi.it/loom oom-mappings.xsd "
        default-package="myapp.domain">

        <!--
                Informazioni di mapping relative alle classi:
                Class1, Class2, ... del modello ad oggetti
        -->
        <class name="Class1"...
        </class>

        <class name="Class2"...
        </class>

        ...

</oom-mappings>
```

Il listato 3.3 mostra un esempio di file `oom-mappings.xml`. I dettagli relativi al mapping verranno trattati nella prossima sezione ma vale comunque la pena di osservare che ogni file `oom-mappings.xml` può contenere le informazioni relative al mapping di una o più classi.

3.2 Mapping

Fare il mapping significa stabilire una corrispondenza tra le entità del modello ad oggetti, scritto in Java, e costituente il Domain Layer e gli elementi del modello ontologico, scritto in OWL, e costituenti il Data Layer.

Nel fare il mapping occorre tenere presenti un numero di differenze, note anche come impedence mismatch [89][40][54], che esistono tra le due differenti astrazioni e di cui riportiamo brevemente le più importanti.

La prima differenza è costituita dal fatto che nei modelli OWL sono ammissibili legami di ereditarietà multipla ed è possibile definire nuove classi, non solo specificandone il nome ma anche attraverso costrutti di tipo insiemistico, per enumerazione degli individui o utilizzando espressioni di restrizione sui predicati. Con Java è possibile definire classi ed interfacce solo attraverso la specificazione di un nome e l'ereditarietà multipla vale soltanto per le interfacce.

Altra differenza è costituita dal fatto che nei modelli OWL le classi non hanno attributi, si ricorre ai predicati per aggiungere relazioni tra classi e tipi di dati primitivi. Classi e predicati formano due gerarchie di entità distinte. In Java il concetto di predicato non esiste e deve essere simulato all'occorrenza come attributo semplice o come classe di associazione.

Infine nei modelli OWL il tipo di un individuo può variare nel tempo, ogni individuo può essere contemporaneamente istanza di più classi, ed il tipo degli individui può venire dedotto dalle loro caratteristiche. In Java ogni oggetto è istanza di una sola classe ed il suo tipo, definito esplicitamente all'atto della creazione, non può variare nel tempo.

Loom, tenuto conto delle criticità derivanti dall'impedence mismatch esistente tra modelli ad oggetti e modelli ontologici, permette di fare il mapping attraverso l'utilizzo, nei file `oom-mappings.xml`, di un opportuno insieme di costrutti.

3.2.1 <class>

Il tag `<class>` rappresenta l'elemento fondamentale per la descrizione della corrispondenza tra elementi del modello ad oggetti e di quello ontologico.

Listing 3.4. Esempio di utilizzo del tag <class>

```
<class name="Patient" uri="urn:loom-test:Patient">
...
</class>
```

Il listato 3.4 mostra un semplice esempio di come sia possibile far corrispondere gli oggetti della classe Java `Patient`, specificata nell'attributo `name`, con le istanze della classe ontologica `urn:loom-test:Patient`, specificata nell'attributo `uri`.

Per mappare una gerarchia di classi è sufficiente utilizzare il tag `<class>` in modo annidato.

Listing 3.5. Mapping di una gerachia di classi

```
<class name="RiskFactor" uri="urn:loom-test:RiskFactor">
        <class name="HereditaryRiskFactor"
                uri="urn:loom-test:HereditaryRiskFactor">
        ...
        </class>

        <class name="NotHereditaryRiskFactor"
                uri="urn:loom-test:HereditaryRiskFactor">
        ...
        </class>

...
</class>
```

Nel listato 3.5 si vede ad esempio in che maniera far corrispondere le classi relative ai fattori di rischio del modello Java con quelle del modello OWL.

Affinché una definizione di mapping del tipo di quella descritta sia valida occorre che rispetti alcuni vincoli. Più precisamente occorre che la classe ontologica definita nel tag `<class>` sia sottoclasse della classe ontologica definita nel tag `<class>` contenitore, se esistente. In maniera simile la classe Java definita in ogni tag `<class>` deve essere sottoclasse di quella parent se quest'ultima non dichiara interfacce e deve definire interfacce che siano estensione di quelle della classe Java parent. Si veda ad esempio il modello descritto in Fig. 3.1.

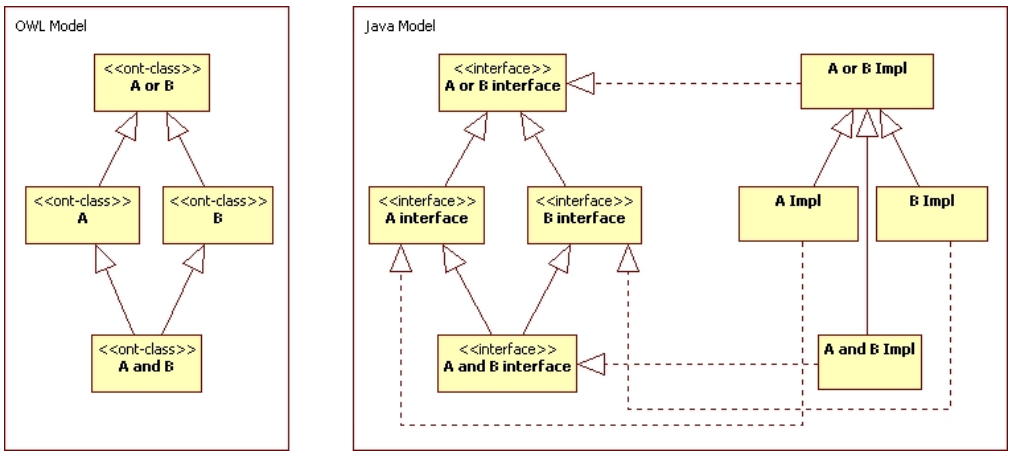

Figura 3.1. Modello ontologico e ad oggetti con ereditarietà multipla

Supponiamo di voler mettere in corrispondenza le classi ontologiche: `AorB`, `A`, `B`, `AandB`, con quelle di un modello Java. Poiché nel modello ontologico esiste una relazione di ereditarietà multipla, quella di `AandB` verso `A` e `B`, nel modello Java occorrerà ricorrere ad un insieme di interfacce: `AorBinterface`, `Ainterface`, `Binterface` e `AandBinterface`, e ad un insieme di classi che implementano le interfacce: `AorBimpl`, `Aimpl`, `Bimpl` e `AandBimpl`. Da notare che le classi Java potranno avere tra di loro rapporti di ereditarietà di implementazione ma non potranno esserci rapporti di ereditarietà multipla. Si veda infatti che `AandBimpl` non eredita da `Aimpl` e nemmeno da `Bimpl`.

Listing 3.6. Mapping di una gerachia di classi con ereditarieà multipla

```
<class name="AorBimpl" interfaces="AorBinterface" uri="urn:AorB">
        <class name="Aimpl" interfaces="Ainterface" uri="urn:A">

                <class name="AandBimpl"
                        interfaces="AandBinterface"
                        uri="urn:AandB">

                </class>

        </class>

        <class name="Bimpl" interfaces="Binterface" uri="urn:Bimpl">

                <class ref="AandBimpl" />

        </class>
</class>
```

Il frammento di mapping corrispondente viene mostrato nel listato 3.6 ed affinché sia valido dovranno valere le seguenti relazioni: `urn:AandB` sottoclasse di `urn:A`, `urn:A` sottoclasse di `urn:AorB`, `urn:AandB` sottoclasse di `urn:A`, `urn:B` sottoclasse di `urn:AorB`, `urn:AandB` sottoclasse di `urn:AandB` ed inoltre: `AandBinterface` estende `Ainterface`, `Ainterface` estende `AorBinterface`, `Binterface` estende `AorBInterface`. Si noti come nel listato venga impiegato il tag `<class ref=“”/>` per riferire la definizione di una classe data in un altro punto del file di mapping.

Per ogni classe Java può essere data solo una definizione di mapping e non devono essere generati riferimenti circolari attraverso l'uso del tag `<class ref=“”/>`, per ogni classe ontologica, invece, possono essere dati più mapping.

Si noti infine che il meccanismo di mapping è stato descritto nel caso in cui si vogliano trattare come istanze di classi Java le istanze di un modello

ontologico. Utilizzando lo stesso tag `<class>`, con opportuni accorgimenti, sarà possibile trattare le classi ed i predicati ontologici come istanze di classi Java.

Tale possibilità, assai diversa dalle capacità degli ordinari strumenti di ORM (come ad esempio Hibernate) e non sempre disponibile negli attuali strumenti di OOM, abilita il Domain Layer a modificare la struttura del Data Layer con la stessa facilità con cui, più usualmente, si modificano le realizzazioni concrete delle concettualizzazioni.

Gli accorgimenti necessari consistono nel fornire attraverso l'attributo `mapper-class` del tag `<class>` l'indicazione della corretta classe Loom di mapping. Lo stesso attributo può essere utilizzato come punto di estensione di Loom al fine di aggiungere a tale libreria funzionalità di mapping attualmente non previste.

3.2.2 <id>

Il tag `<id>` serve a specificare l'attributo della classe che conterrà l'uri (od il codice del blank-node) necessario ad identificare l'elemento ontologico all'interno del modello OWL.

Listing 3.7. Esempio di utilizzo del tag <id>

```
<class name="Patient" uri="urn:loom-test:Patient">
        <id name="code" />
...
</class>
```

Nel listato 3.7 l'uri identificativo dei pazienti nell'ontologia viene mappato nell'attributo "code", necessariamente stringa, della classe Java.

3.2.3 <predicate>

Il tag <predicate>, che può essere impiegato solo annidandolo all'interno di un tag <class>, serve a specificare il mapping tra gli attributi delle classi Java

ed i predicati data-type dell'ontologia.

Listing 3.8. Esempio di utilizzo del tag <predicate>

```
<class name="Patient" uri="urn:loom-test:Patient">
        <predicate name="name" uri="urn:loom-test:hasName" />
...
</class>
```

Il listato 3.8 mostra, ad esempio, come sia possibile mettere in corrispondenza il predicato `urn:loom-test:hasName` con l'attributo `name` della classe `Patient`. Tale mapping risulterà valido se la classe `Patient` contiene i metodi `getName` e `setName`, nel caso di nome singolo oppure se contiene `addName`, `removeName` e `nameIterator`, nel caso di più nomi.

Supponendo che ogni paziente abbia un solo nome, sarà possibile leggerlo o modificarlo scrivendo codice Java del tutto simile a quello riportato nel listato 3.9.

Listing 3.9. Lettura e scrittura attraverso metodi getter e setter

```
String curName = patient.getName();
patient.setName( "Mario" );
```

Supponendo invece che ogni paziente possa avere più nomi il codice Java per leggere o modificare l'attributo sarà del tutto simile a quello mostrato nel listato 3.10.

Listing 3.10. Lettura e scrittura degli elementi di una collezione

```
patient.addName( "Mario" );
patient.removeName( "Giovanni" );
Iterator<String> it = patient.nameIterator();
```

In generale, dato il mapping di un predicato data-type verso l'attributo <attribute>, verrà verificata la presenza dei metodi: set<attribute> e

get<attribute> oppure quella dei metodi: add<attribute>, remove<attribute> e <attribute>Iterator.

Da notare che le modifiche apportate agli attributi di un oggetto non verranno rese persistenti nell'ontologia, all'atto del salvataggio, se nel mapping è stato specificato `read-only="true"`.

Utilizzando l'attributo `property-class` del tag `<predicate>`, è inoltre possibile fare il mapping tra modelli ontologici e modelli ad oggetti che consentono un maggior grado di indirezione.

Listing 3.11. Esempio di mapping che utilizza l'attributo property-class

```
<class name="Patient" uri="urn:loom-test:Patient">
        <predicate name="generalInfo"
                        uri="urn:loom-test:hasName"
                        property-class="GeneraInfoProperty" />

        <predicate name="generalInfo"
                        uri="urn:loom-test:hasSurname"
                        property-class="GeneraInfoProperty" />

</class>

<class name="GeneralInfoProperty" ...
...
</class>
```

Si consideri, ad esempio, il listato 3.11 in cui i predicati `urn:loom-test:hasName` e `urn:loom-test:hasSurname`, trattati come istanze della classe `GeneralInfoProperty` insieme ai rispettivi eventuali sottopredicati, vengono associati, con modalità indiretta , all'attributo `generalInfo` della classe `Patient`.

In questo caso il codice Java per accedere agli attributi dell'oggetto sarà: quello del listato 3.12, per attributi semplici; quello del listato 3.13 per attributi con più occorrenze.

Listing 3.12. Accesso ad attributi singoli mappati indirettamente

```
//
// nameProperty: instance of GeneralInfo
// that represents urn:loom−test:hasName
//
// surnameProperty: instance of GeneralInfo
// that represents urn:loom−test:hasSurname
//

String curName =
        patient.getGeneralInfo( nameProperty );
patient.setGeneralInfo( nameProperty, "Mario" );

String curSurname =
        patient.getGeneralInfo( surnameProperty );
patient.setGeneralInfo( surnameProperty, "Rossi" );

Iterator<GeneralInfo> patient.generalInfoProperties();
```

Occorre sottolineare che la modalità di mapping con indirezione è da considerarsi una funzionalità avanzata da adoperarsi solo nei casi in cui sia effettivamente necessaria all'implementazione di specifiche funzionalità di Reflection. Utilizzando in maniera non oculata tale modalità di mapping si rischia infatti di ottenere un Domain Layer il cui codice è più verboso, un po' più difficile da leggere e maggiormente orientato all'errore.

3.2.4 <reference>

Il tag `<reference>`, che può essere impiegato solo annidandolo all'interno di un tag `<class>`, serve a specificare il mapping tra gli attributi delle classi Java ed i predicati object-type dell'ontologia.

Il listato 3.14 mostra un esempio in cui fattori di rischio di un paziente,

Listing 3.13. Accesso ad attributi multipli mappati indirettamente

```
//
// nameProperty: instance of GeneralInfo
// that represents urn:loom−test:hasName
//
// surnameProperty: instance of GeneralInfo
// that represents urn:loom−test:hasSurname
//

patient.addGeneralInfo( nameProperty, ”Mario” );
patient.removeGeneralInfo( nameProperty, ”Giovanni” );
Iterator<String> nameIterator =
        patient.generalInfoIterator( nameProperty );

patient.addGeneralInfo( surnameProperty, ”Mario” );
patient.removeGeneralInfo( surnameProperty, ”Giovanni” );
Iterator<String> surnameIterator =
        patient.generalInfoIterator( surnameProperty );

Iterator<GeneralInfo> patient.generalInfoProperties();
```

identificati nell'ontologia dal predicato `urn:loom-test:hasRiskFactor`, vengono mappati nella classe `RiskFactor` ed associati all'attributo `riskFactor` della classe `Patient`. L'insieme di attributi che la classe `Patient` deve possedere affinché il mapping sia corretto, segue le stesse regole introdotte per il tag `<predicate>`.

Per il tag `<reference>`, oltre agli attributi `name`, `uri`, `read-only` e `property-class`, aventi lo stesso significato di quelli del tag `<predicate>`, si possono valorizzare gli attributi: `lazy`, `cascade-persist`, `cascade-refresh`, `cascade-remove`, `cascade-attach` e `cascade-detach`, il cui significato sarà chiarito nella prossima sezione dopo che sarà stato discusso del ciclo di vita degli oggetti Loom.

Listing 3.14. Esempio di utilizzo del tag <reference>

```
<class name="Patient" uri="urn:loom-test:Patient">
        <reference name="riskFactor" uri="urn:loom-test:hasRiskFactor" />
...
</class>

<class name="RiskFactor" ...
...
</class>
```

3.3 Application Programming Interface

Per sfruttare il mapping, definito impiegando i file di configurazione, è necessario adoperare la API di Loom che è stata progettata ispirandosi volutamente alla ben nota specifica JPA. Numerose sono infatti le similitudini tra gli strumenti di object relational mapping ed i componenti per l'object ontology mapping che, rese evidenti dalle scelte progettuali fatte, rendono Loom più facile da usare.

La prima operazione da compiere con Loom è quella di ottenere un'istanza corretta della classe `EntityManger`, essa rappresenta infatti l'interfaccia per mezzo della quale tutti i servizi di Loom sono concretamente raggiungibili.

Listing 3.15. Codice per ottenere un'istanza di EntityManager

```
PersistenceUnit pu =
        new PersistenceUnitImpl( "/myapp/resources/persistence.xml" );

EntityManagerFactory entityManagerFactory =
        pu.createEntityManagerFactory();

EntityManager entityManager =
        entityManagerFactory.createEntityManager();
```

Nel listato 3.15 viene mostrato il frammento di codice Java necessario ad ottenere un'istanza di `EntityManager`. Per prima cosa occorre un'istanza di `PersistenceUnit` creata passandole come parametro il nome del file `persistence.xml` da utilizzare. Bisogna quindi costruire un'istanza di `EntityManagerFactory` invocando il metodo `createEntityManagerFactory` di `PersistenceUnit`. A questo punto è sufficiente invocare il metodo `createEntityManager` di `EntityManagerFactory` per ottenere l'oggetto `EntityManager` desiderato. Si noti che gli oggetti `PersistenceUnit` e `EntityManagerFactory` sono molto onerosi da creare e un'applicazione tipicamente ne istanzierà solo uno. L'`EntityManager` invece, corrispondente alla sessione di lavoro, è molto veloce da istanziare e l'applicazione potrà gestirne anche un numero elevato.

I servizi della API di Loom possono essere raggruppati in due macro categorie: la prima contenente quelle funzionalità che in modi differenti servono a leggere il modello OWL per poi istanziare e valorizzare un insieme di oggetti Java; la seconda contenente quelle funzionalità che servono a leggere le informazioni di un insieme di oggetti Java per renderle poi persistenti nel modello OWL.

Più in particolare `EntityManager` fornisce i metodi:

- `persist` per rendere persistente un nuovo oggetto, vedi listato 3.16;

Listing 3.16. Esempio in cui si rende persistente un nuovo oggetto

```
//
// create a new object
// and then persit
//
// Suppose that Patient is a mapped class
//
Patient patient = new Patient();
...
entityManager.persist( patient );
```

- `load` per caricare un singolo oggetto, vedi listato 3.17;

Listing 3.17. Load di un oggetto

```
Patient patient = entityManager.load( Patient.class, "urn:loom−test:55" );
```

- `delete` per cancellare un oggetto, vedi listato 3.18;

Listing 3.18. Cancellazione di un oggetto

```
entityManager.delete( patient ));
```

- `refresh` per ricaricare i valori dell'oggetto dall'ontologia, vedi listato 3.19;

Listing 3.19. Refresh di un oggetto

```
entityManager.refresh( patient ));
```

- `detach` per "staccare" un oggetto dall'EntityManager affinché le modifiche che lo riguardano non vengano rese persistenti , vedi listato 3.20;

- `attach` per rendere nuovamente gestito un oggetto di cui è stato fatto il detach, vedi listato 3.21.

Tutte le operazioni sugli oggetti devono avvenire nel contesto di una transazione. Le modifiche apportate (aggiunte, modifiche e cancellazioni) agli oggetti Java saranno rese persistenti in blocco al momento dell'invocazione del metodo `commit` sull'oggetto `Transaction` corrente.

Nel listato 3.22 vengono mostrate le chiamate tipiche per la gestione delle transazioni che, c'è da precisare, sono transazioni logiche e non relative al

Listing 3.20. Detach di un oggetto

```
entityManager.detach( patient ));
```

Listing 3.21. Attach di un oggetto

```
entityManager.attach( patient ));
```

repository ontologico. Può dunque trascorrere anche molto tempo tra il loro inizio (invocazione del metodo `begin`) ed il loro termine (invocazione del metodo `commit` o `rollback`) senza che questo si ripercuota sulle prestazioni dell'applicazione.

Utilizzando Loom è, infine, possibile adoperare espressioni SPARQL per interrogare l'ontologia ed ottenere per risultato liste di oggetti, vedi listato 3.23.

Per meglio comprendere le funzionalità offerte da Loom occorre esplicitare quali sono gli stati logici in cui gli oggetti del modello Java mappati possono trovarsi e quali sono le transizioni di stato ammissibili, si veda a tal proposito Fig. 3.2.

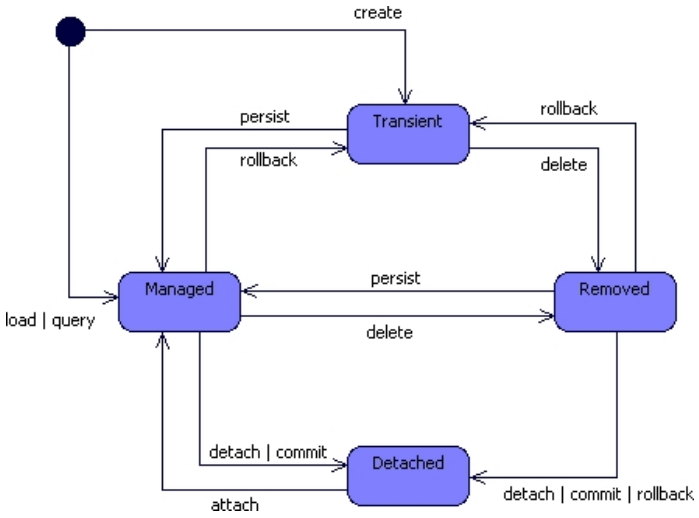

Figura 3.2. Ciclo di vita degli oggetti gestiti da Loom

Listing 3.22. Gestione delle transazioni

```
//
// begin a new transaction
// and then commit
//
entityManager.getTransaction().begin();
...
entityManager.getTransaction().commit();

//
// begin a new transaction
// and then rollback
entityManager.getTransaction().begin();
...
entityManager.getTransaction().commit();
```

Quando si istanzia un nuovo oggetto Java questo si trova, per Loom, nello stato logico *Transient*. Invocando il metodo `persist` della classe `EntityManager` si modifica lo stato dell'oggetto a *Managed*, questo significa che Loom, al commit della transazione, si preoccuperà di rendere persistenti nell'ontologia le modifiche all'oggetto. Un modo alternativo di ottenere oggetti *Managed* è quello di interrogare il modello OWL attraverso il metodo `load` di `EntityManager` oppure eseguendo una Query SPARQL. Se un oggetto viene cancellato con il metodo `delete`, allora il suo stato diventa *Removed*. Applicando il metodo `persist` ad un oggetto *Removed*, questo torna ad essere *Managed*. Quando la transazione termina gli oggetti diventano *Detached* come pure lo sono gli oggetti su cui è stato invocato il metodo `detach`. Gli oggetti *Detached* possono tornare ad essere *Managed* utilizzando il metodo `attach`.

Impostando gli attributi: `cascade-persist`, `cascade-remove`, `cascade-refresh`, `cascade-attach` e `cascade-detach`, del tag `<reference>` del file di mapping si sceglie in che modo far propagare le modifiche agli oggetti. In questa maniera il codice del Domain Layer risulterà più compatto e meno

Listing 3.23. Esempio di query

```
//
// sparql is a string that
// contains a SPARQL query
//
Query query = entityManager.createQuery( Patient.class, sparql );
List<Patient> result = query.execute();
```

soggetto ad errori.

Consideriamo adesso il codice del listato 3.24 in cui viene mostrato un esempio di utilizzo di Loom. In particolare viene aggiunto il fattore di rischio "ansia" al paziente Mario Rossi e modificato la città di residenza. Supponiamo che `Patient` e `RiskFactor` siano classi opportunamente mappate.

Listing 3.24. Esempio di modifica

```
entityManager.getTransaction().begin();

Patient patient = entityManager.load( Patient.class, "urn:loom-test:55" );

RiskFactor riskFactor =
        new NotHereditaryRiskFactor( "Ansia" );
patient.addRiskFactor( riskFactor );

patient.getAddress().setCity( "Firenze" );

entityManager.getTransaction().commit();
```

Nel frammento di codice presentato è possibile notare che inizia con l'avvio di una transazione e termina con la conclusione della stessa. Si osserva poi che per rendere persistenti le modifiche non è necessario, in questo caso, invocare alcun metodo specifico perché esse riguardano oggetti *Managed* (paziente ed indirizzo) ed oggetti riferiti da oggetti *Managed* (il nuovo fattore di rischio).

Il codice risulta piuttosto lineare e, pur archiviando tutta l'informazione in un modello ontologico, non contiene alcun riferimento ad OWL.

Si osservi adesso con particolare attenzione la riga per la modifica dell'indirizzo ed in particolare all'utilizzo del metodo `getAddress` per accedere ad uno degli oggetti riferiti da `patient`.

Loom permette di navigare tra i riferimenti degli oggetti in modo del tutto naturale come se questi formassero un grafo totalmente caricato in memoria. In realtà Loom utilizza un elaborato meccanismo di Proxy al fine di prelevare dall'ontologia, nella maniera più efficiente possibile, solo gli oggetti di cui c'è effettivamente bisogno. È possibile regolare il comportamento di Loom nel caricamento dei riferimenti attraverso l'attributo `lazy` del tag `<reference>`.

Richiedendo a Loom di caricare un oggetto (ad esempio con il metodo `load`) si otterrà un'istanza i cui riferimenti saranno effettivamente presenti in memoria soltanto se nel mapping l'attributo `lazy` corrispondente è impostato a `false`, viceversa i riferimenti saranno costituiti da proxy che preleveranno l'informazione dall'ontologia solo all'atto dell'utilizzo di uno dei loro metodi.

Il modello ad oggetti del Domain Layer accede facilmente al modello ontologico del Data Layer grazie alla API di Loom che nasconde tutte le complessità relative all'ottimizzazione delle operazioni di lettura e scrittura, ottimizzazione che è necessaria per garantire all'applicazione efficienza nelle prestazioni. Questo tema assume particolare rilevanza in quanto gli odierni repository ontologici sono purtroppo piuttosto lenti.

3.4 Analisi e Progettazione

In questa sezione verranno presentate l'analisi ed il design relativi al componente Loom. Saranno mostrati cioè alcuni dettagli relativi alla struttura interna di Loom ed al processo che ha portato alla sua realizzazione.

In particolare è stata applicata la metodologia di sviluppo Iconix [30] in quanto ritenuta sufficientemente rigorosa sebbene piuttosto agile da utilizzare.

Data la complessità del componente di Object Ontology Mapping da realizzare era necessario riferirsi a tecniche di sviluppo software in grado di garantire il pieno soddisfacimento e la perfetta rispondenza del codice prodotto con i requisiti posti.

In Fig. 3.3 si può vedere che Loom è costituito da due sottocomponenti: Oom Api ed Rdf Api, che verranno trattati separatamente.

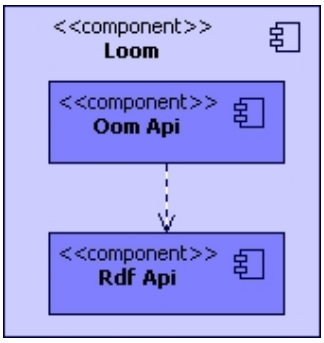

Figura 3.3. Architettura interna di Loom

3.4.1 Rdf Api

In Fig. 3.4 viene mostrato il modello di dominio per la Rdf Api. La classe `RdfNode` serve a rappresentare i vari tipi di risorse che possono trovarsi in un modello RDF, essa pertanto ammette un insieme di specializzazioni costituite dalle classi: `RdfResource` e `Literal`, che a loro volta si specializzano rispettivamente in: `UriRef` e `BlankNode`; `PlainLiteral` e `TypedLiteral`. Le istanze della classe `RdfNode` sono utilizzate per formare le triple RDF rappresentate nel modello con la classe `Statement`. `RdfPersistenceUnitConfiguration` rappresenta infine la classe per mezzo della quale manipolare la configurazione della Rdf Api.

Occorre rilevare che tale modello, costruito nel rispetto della metodologia Iconix, rappresenta il primo passo dell'analisi ed è stato esteso e raffinato nei passi successivi fino a raggiungere un livello di dettaglio necessario a rappresentare delle specifiche per l'implementazione.

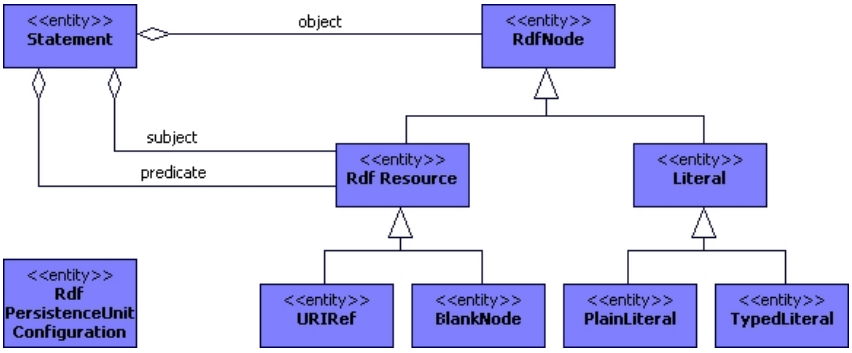

Figura 3.4. Modelli di dominio per la Rdf Api

La metodologia Iconix prevede che i casi d'uso vengano definiti in relazione all'interfaccia che concretamente verrà realizzata. Prima del modello dei casi d'uso viene pertanto presentato, in Fig. 3.5, quello delle classi di interfaccia.

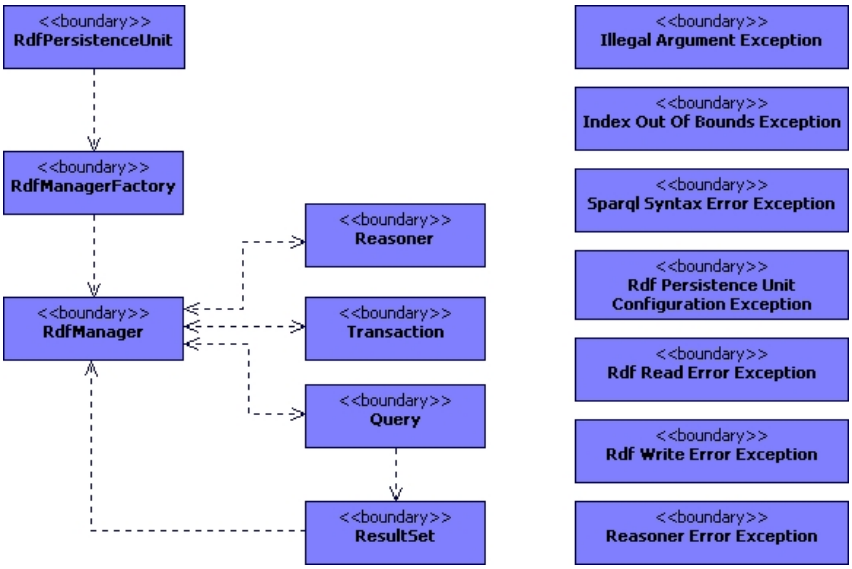

Figura 3.5. Interfacce della Rdf Api

Sia la Rdf Api che la Oom Api presentata più avanti sono state progettate traendo ispirazione da JPA, ritenendo in questo modo di riuscire ad ottenere interfacce di programmazione molto semplici da utilizzare.

In Fig. 3.6 sono mostrati i casi d'uso principali per la Rdf Api. In particolare `CRUD Statement` che si specializza in `Add Statement`, `Remove Statement`,

`Query Statement` e `List Statement`, serve a descrivere le interazioni con la Rdf Api che sono necessarie al fine di manipolare le triple di un modello RDF. `Commit Transaction` riguarda le interazioni relative alla gestione delle transazioni. `Reason on Statements` descrive in che modo accedere alle funzionalità di un reasoner che eventualmente operi sul repository ontologico. `Get RdfManager` infine si riferisce all'interazione necessaria per ottenere un'istanza valida di `RdfManager` cioè un'istanza di quella classe per mezzo della quale si può avere accesso alle funzionalità della Rdf Api.

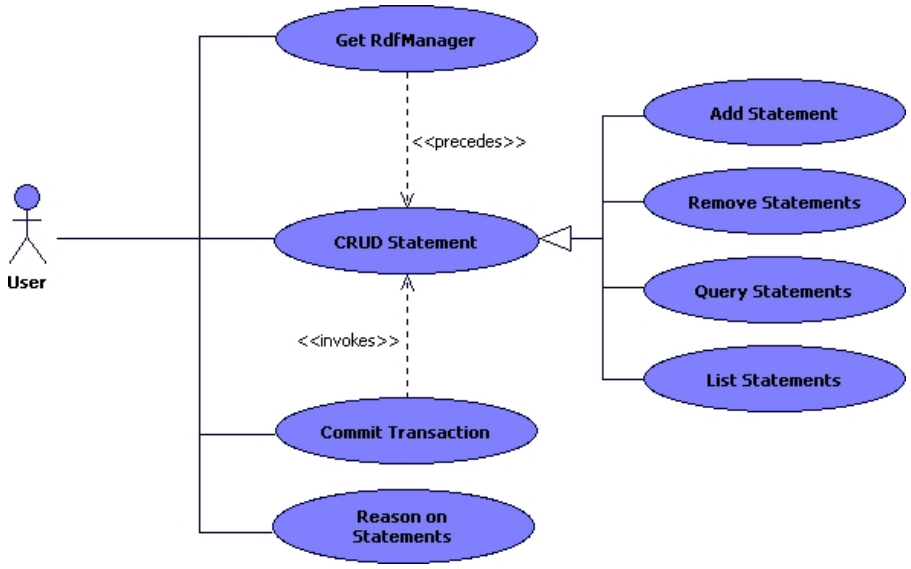

Figura 3.6. Casi d'uso principali per la Rdf Api

Il processo che ha portato alla definizione del modello, delle interfacce e dei casi d'uso, presentato qui in modo sequenziale, è stato in realtà di tipo iterativo ed ha previsto numerose correzioni e cambiamenti per concludersi con la scrittura del testo corrispondente ai casi d'uso e successivamente con l'analisi di robustezza di cui riportiamo due esempi relativi ai casi d'uso: `Add Statement` (Fig. 3.7) e `Query Statements` (Fig. 3.8).

L'analisi di robustezza ha permesso di verificare la coerenza e la non ambiguità dei casi d'uso in modo che questi potessero essere impiegati per guidare la fase di progettazione che è stata eseguita, considerando un diagramma di

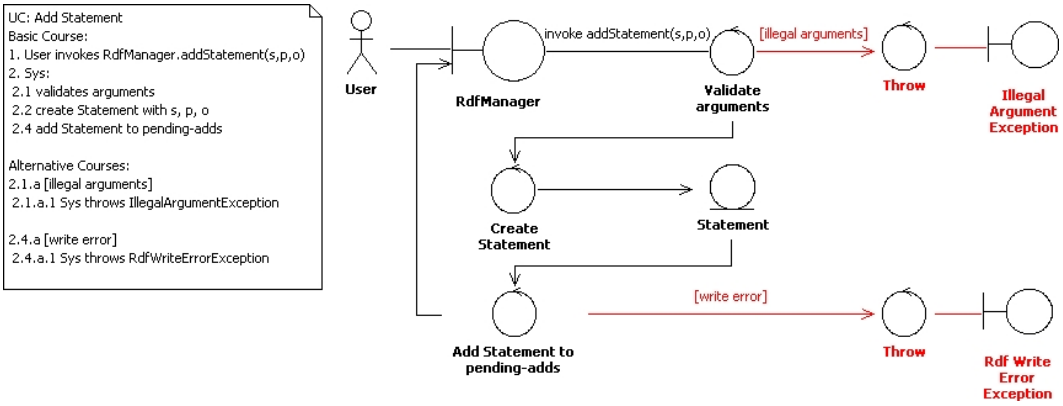

Figura 3.7. Diagramma di robustezza relativo ad Add Statemnt

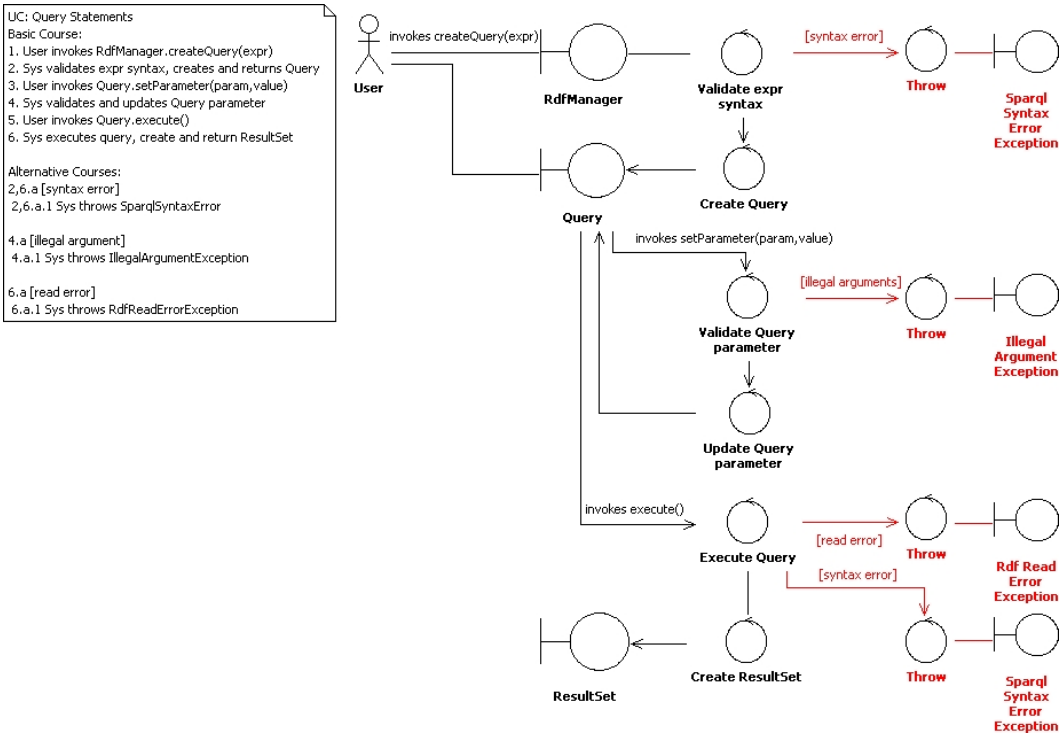

Figura 3.8. Diagramma di robustezza relativo a Query Statements

robustezza per volta, definendo i diagrammi di sequenza corrispondenti al fine di raffinare il modello di dominio fino a farlo diventare un modello di specifica.

In Fig. 3.9 viene mostrato il diagramma di sequenza relativo al caso d'uso `Add Statement`.

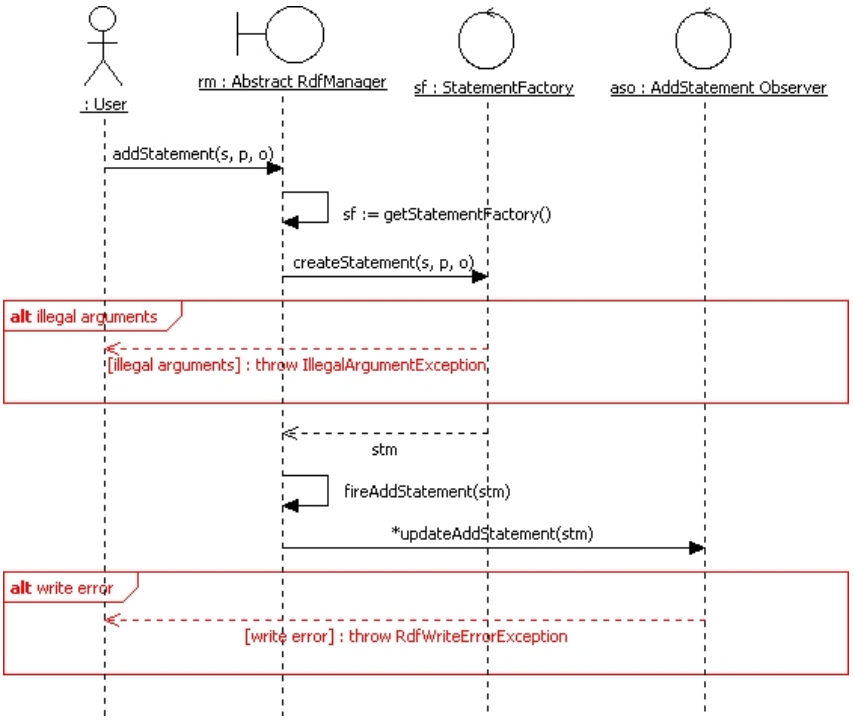

Figura 3.9. Diagramma di sequenza per Add Statement

In Fig. 3.10 viene mostrato un frammento del modello di specifica per la Rdf Api in cui sono evidenziati alcuni dei Design Pattern [36][39][84] adoperati. Occorre osservare che i diagrammi di robustezza sono stati impiegati, sempre come descritto nella metodologia Iconix, anche per individuare l'insieme minimo di test da implementare per verificare il buon funzionamento del software [47][4].

La fase di progettazione si è conclusa con l'individuazione precisa delle classi che nella fase successiva della codifica sono state implementate fino a realizzare il componente Rdf Api mostrato in Fig. 3.11.

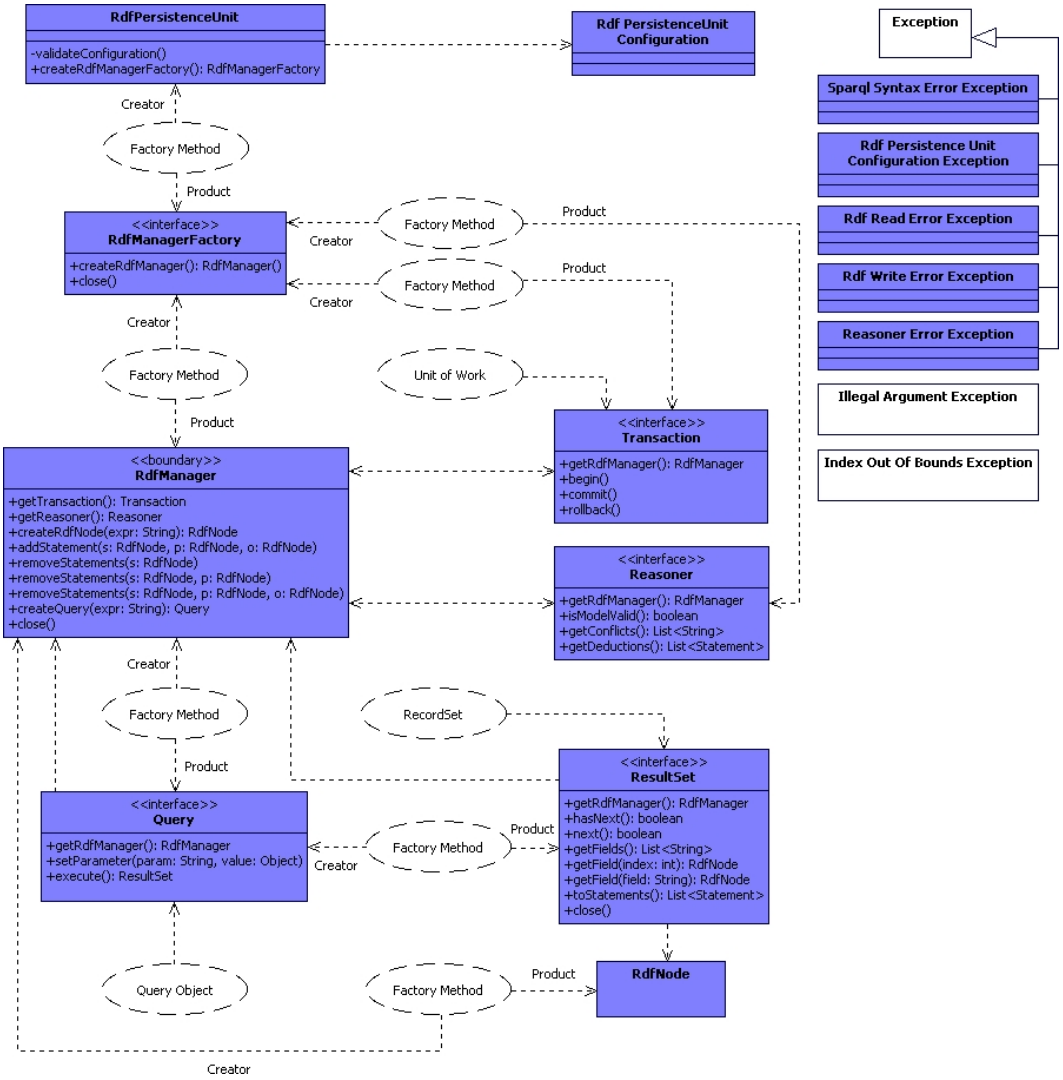

Figura 3.10. Frammento del modello di specifica di Rdf Api

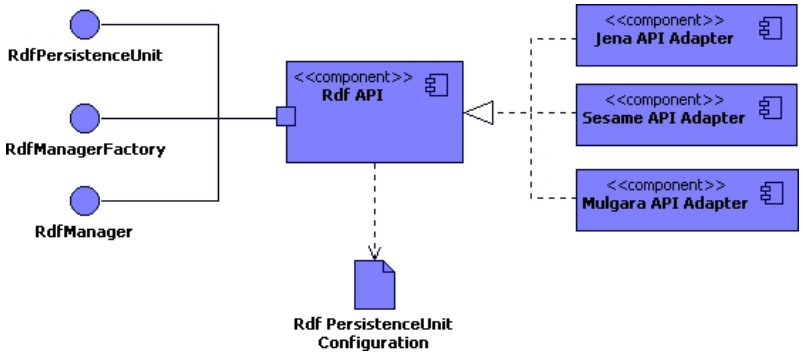

Figura 3.11. Rdf Api

3.4.2 Oom Api

In Fig. 3.12 viene descritto il modello di dominio del componente Oom Api. In particolare le classi `Object Type` ed `Object Status`, insieme alle relative specializzazioni, servono a gestire lo stato logico delle entità manipolate, come descritto nelle sezione relativa al ciclo di vita degli oggetti gestiti da Loom. `ClassMapping`, `IdMapping` e `PropertyMapping` servono invece a rappresentare le informazioni di mapping all'interno della libreria. La classe `Proxy`, infine, rappresenta il tipo di entità che viene prodotta da Loom a seguito di un'operazione di load o di query.

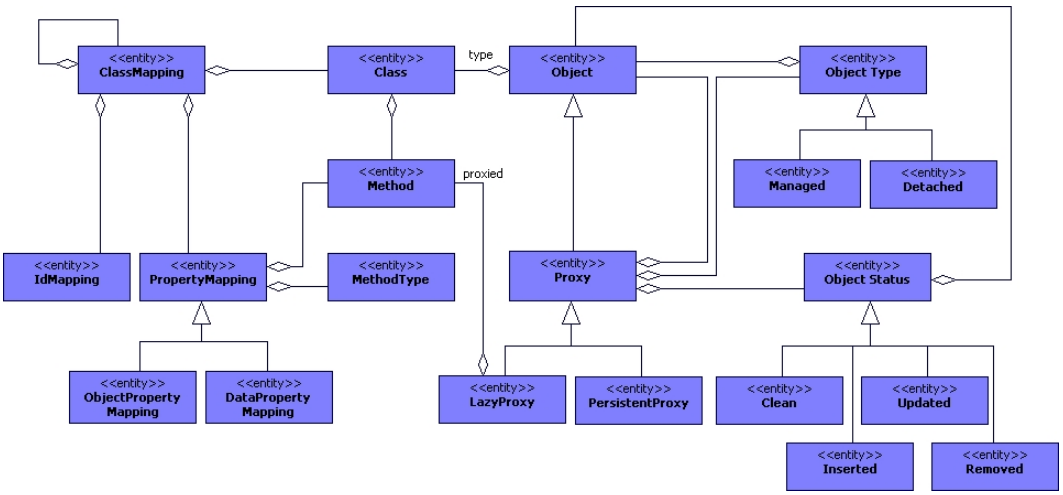

Figura 3.12. Componente OOM

In Fig. 3.13 sono descritte le interfacce del componente Oom Api che corrispondono alle interfacce di Loom percepite dall'esterno e già descritte precedentemente.

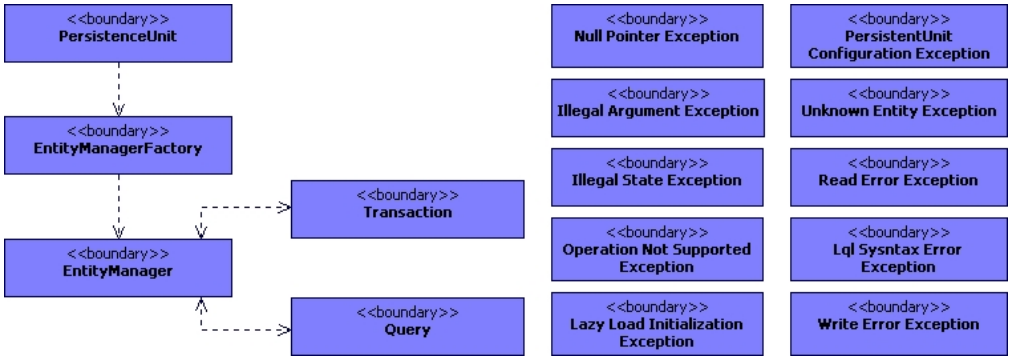

Figura 3.13. Interfacce del componente Oom Api

In Fig. 3.14 viene mostrato il diagramma dei casi d'uso del componente Oom Api.

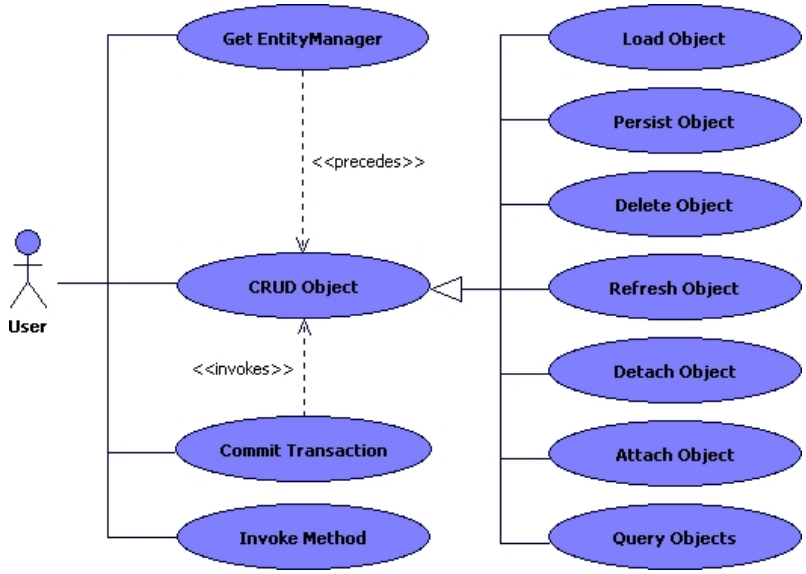

Figura 3.14. Casi d'uso del componente Oom Api

Anche per il componente Oom Api, come per Rdf Api, la fase di analisi è stata seguita dalla verifica della stessa attraverso opportuni diagrammi di

robustezza. Riportiamo a titolo esemplificativo quelli relativi ai casi d'uso: `Load Object` (Fig. 3.15) e `Delete Object` (Fig. 3.16).

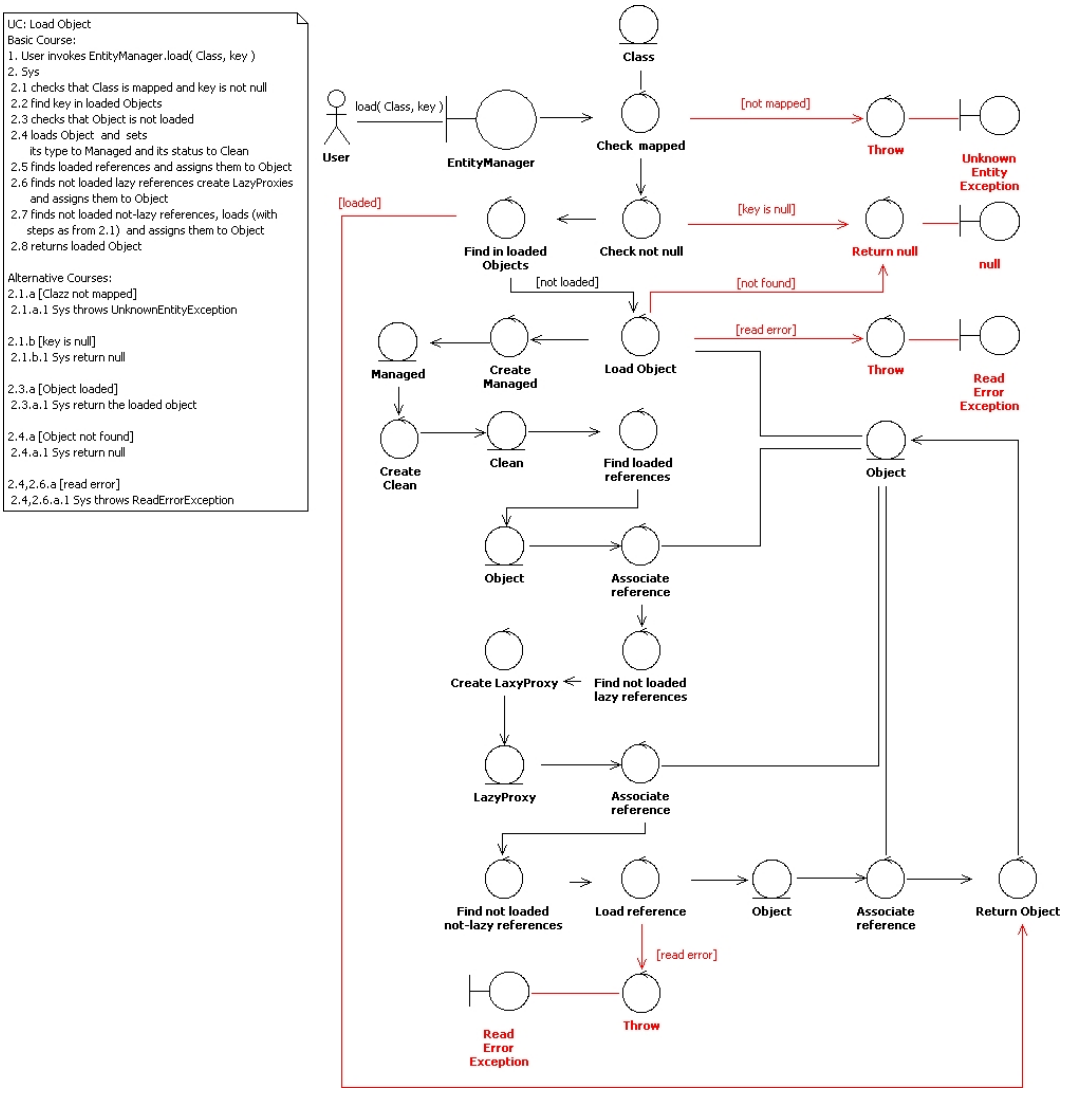

Figura 3.15. Diagramma di robustezza per il caso d'uso Load Object

All'analisi di robustezza è seguita la fase di progettazione attraverso l'individuazione delle classi necessarie con i relativi attributi per mezzo dei diagrammi di sequenza di cui vengono riportati a titolo esempificativo due di quelli relativi al caso d'uso `Load Object`, vedi Fig. 3.17 e Fig. 3.18.

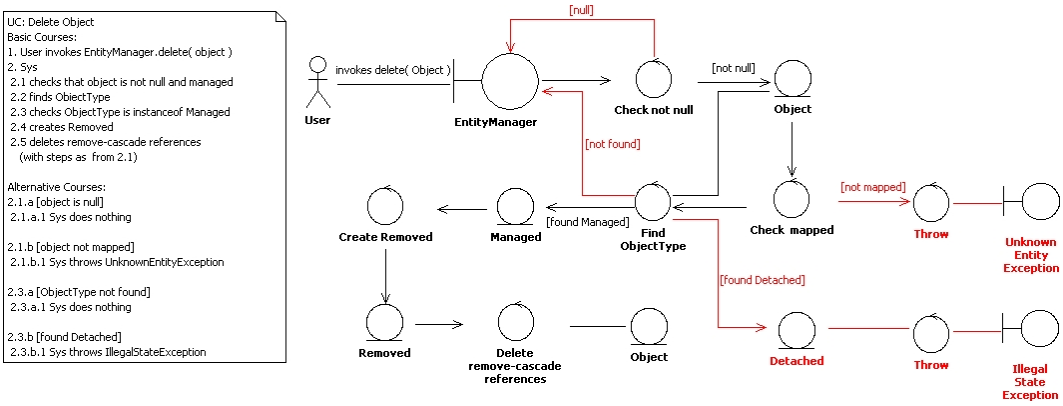

Figura 3.16. Diagramma di robustezza per il caso d'uso Delete Object

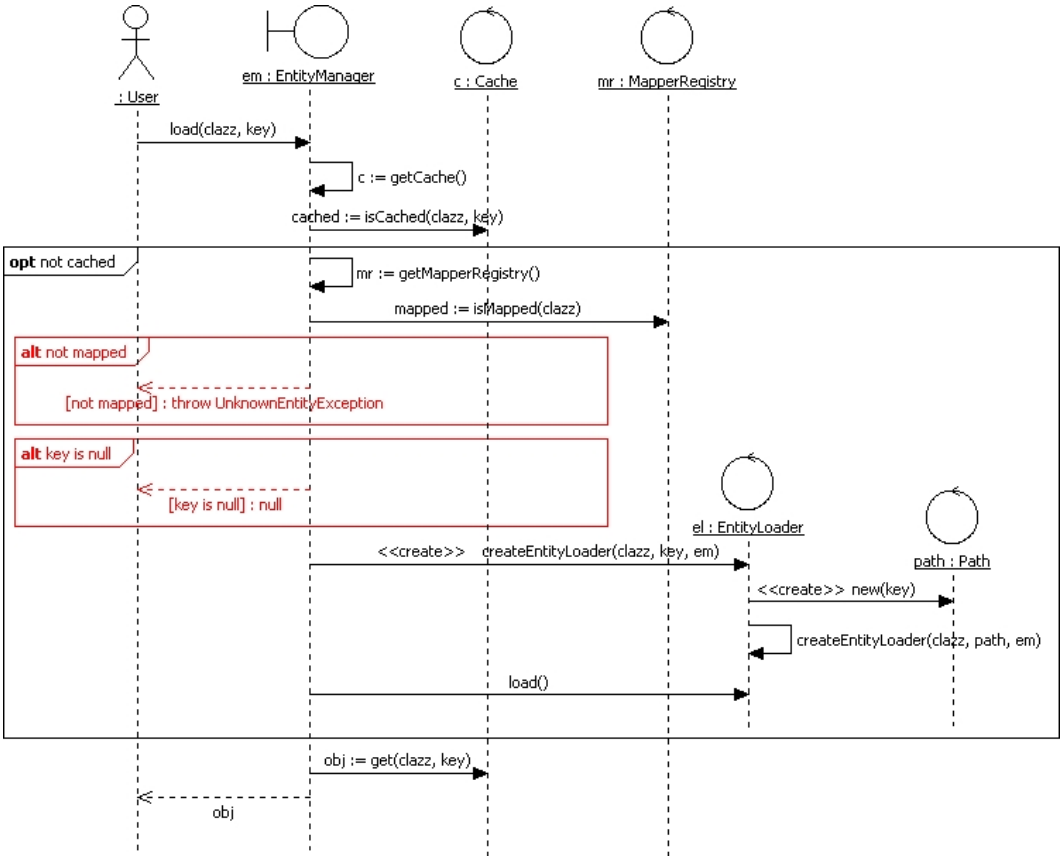

Figura 3.17. Diagramma di sequenza per il caso d'uso Load Object

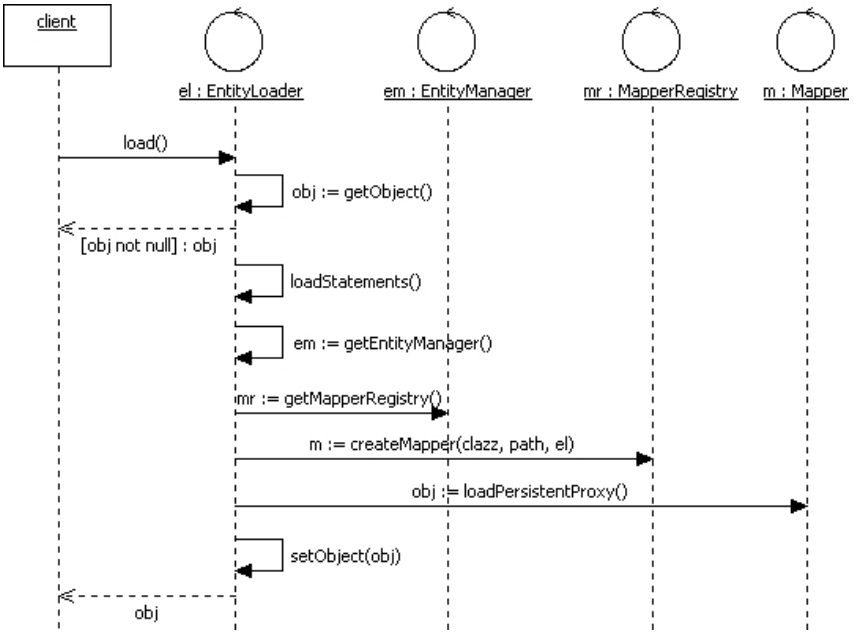

Figura 3.18. Altro diagramma di sequenza per il caso d'uso Load Object

Alla progettazione è seguita la fase di implementazione e testing fino alla realizzazione del componente Oom Api descritto in Fig. 3.19.

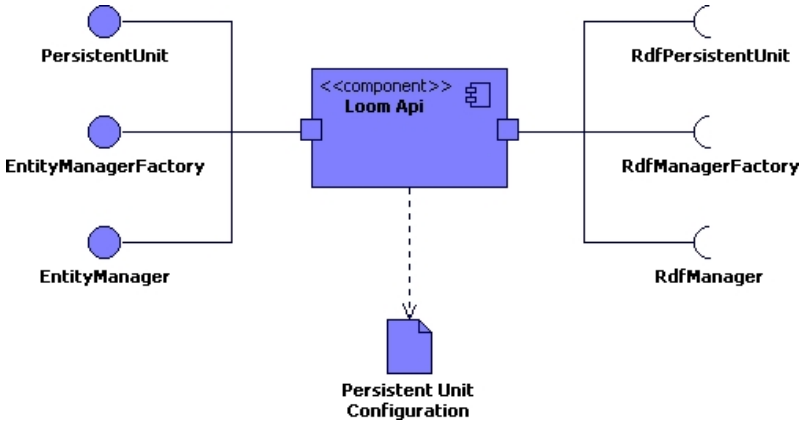

Figura 3.19. Componente OOM

Capitolo 4

Applicazioni

In questo capitolo verranno presentate alcune delle sperimentazioni che abbiamo svolto con l'obiettivo primario di applicare le ontologie a contesti operativi diversi per valutarne le effettive potenzialità e modi di impiego. Le sperimentazioni, che si sono concretizzate nella realizzazione di prototipi funzionanti, sono servite inoltre ad individuare più precisamente le caratteristiche architetturali che le applicazioni ontologiche dovrebbero possedere.

Molte delle sperimentazioni si sono svolte nel contesto della collaborazione tra l'Università di Firenze e l'Azienda Ospedaliero Universitaria di Careggi (AOUC) permettendoci di affrontare problemi reali, complessi e talora imprevedibili. I forti requisiti di robustezza ed efficienza legati all'attività medico ospedaliera hanno talvolta imposto la realizzazione di applicativi implementati con tecnologie più consolidate parallelamente a quella di prototipi più innovativi implementati con le ontologie. Questo, pur determinando un maggior carico di lavoro, ha permesso di confrontare approfonditamente le ontologie con le tecnologie che sono invece lo stato dell'arte attuale.

Avendo delimitato l'ambito di indagine della tesi alle ontologie, verrà necessariamente data maggiore evidenza ai risultati ottenuti con i prototipi ontolo-

gici rispetto a quelli delle applicazioni realizzate con tecnologie più tradizionali.

4.1 Owl2Xml

Owl2Xml è un componente che permette la rappresentazione ontologica delle corrispondenze tra modelli OWL e schemi XSD al fine di eseguire query semantiche sugli XSD e di convertire i messaggi XML circolanti su un bus in equivalenti messaggi di tipo ontologico.

4.1.1 Enterprise Application Integration

Nell'ultima decade, l'Enterprise Application Integration (EAI) è diventata uno dei settori più rilevanti dell'Information Technology (IT). L'obiettivo dell'EAI è quello di interconnettere più Enterprise Information System (EIS) per estendere i processi di business attraverso tutta l'organizzazione. Alcuni tra i fattori che giustificano gli sforzi nell'area dell'EAI sono gli elevati costi che lo sviluppo di nuove applicazioni potrebbe comportare, la forte fiducia da parte degli operatori nei sistemi di legacy [10] in uso e la necessità di integrare velocemente differenti sistemi informativi in seguito, ad esempio, a fusioni od acquisizioni aziendali. Quasi invariabilmente, gli EIS da integrare sono il prodotto di sviluppi occorsi nel tempo e sono caratterizzati dall'essere HAT (Heterogenous, Autonomous and Distributed)[15][45]. Ogni sistema applicativo opera cioè isolatamente andando a determinare isole di informazione [93] ed isole di automazione [62]. Le isole di informazione archiviano dati che possono sovrapporsi con i contenuti di altre isole e solitamente nessuna di esse fornisce un'immagine completa dei dati dell'organizzazione. Creare una vista unificata dei dati dell'impresa necessita di informazioni di integrazione provenienti da molteplici risorse di dati; inoltre accade spesso che applicazioni e processi di business siano duplicati attraverso più isole. Questo richiede la sincronizzazione dei cambiamenti tra più applicativi garantendo la consistenza ed il rispetto delle regole e dei processi di business. Il crescente bisogno di realizzare integrazioni ha determinato la nascita di un numero rilevante di EAI platform introdotte

da industrie ben conosciute (incluse IBM, Microsoft, Oracle e Sun). I software commerciali per EAI si basano su un ampio spettro di tecnologie [99][55], e sono talvolta piuttosto differenti gli uni dagli altri, tanto che sono state introdotte tassonomie per valutare e confrontare i diversi pacchetti [78][88]. Quel che purtroppo le piattaforme EAI odierne hanno in comune è l'assenza di strumenti per la gestione delle integrazioni a livello semantico. L'assenza cioè di strumenti per la definizione della semantica dei messaggi scambiati e delle funzionalità offerte. L'avvento del Semantic Web e di tecnologie quali RDF, RDF/S ed OWL abilita la descrizione formale della semantica delle strutture scambiate tra gli EIS, aprendo la via alla validazione e riconciliazione automatica dei dati scambiati e più in generale alla realizzazione di strumenti di ausilio all'integrazione di tipo semantico [77][15][102][80][70][119][83].

Il tema dell'utilizzo di tecnologie ontologiche per la risoluzione di problemi di integrazione è stato trattato da numerosi autori [103][52][51][94][113]; il contesto della collaborazione con AOUC ci ha permesso di proseguire questa linea di ricerca attraverso sperimentazioni ed approfondimenti che hanno portato alla realizzazione di componenti di integrazione di tipo ontologico in grado di operare sugli odierni Enterprise Service Bus fornendo nuovi strumenti di elaborazione e di governance per i problemi di EAI [13].

4.1.2 Enterprise Integration Architectures

Dal punto di vista tecnico ci sono essenzialmente quattro approcci per l'integrazione [48]:

- File Transfer: ciascuna applicazione produce file di dati consumati dalle altre applicazioni e consuma i file prodotti dalle altre applicazioni.
- Shared database: le applicazioni devono archiviare in un database comune i dati che desiderano condividere.
- Remote Procedure Invocation: ciascuna applicazione espone un insieme di servizi che può essere invocato dalle altre applicazioni al fine di condividere dati e funzionalità.

- Messaging: ciascuna applicazione si connette ad un comune sistema per lo scambio di messaggi e scambia dati e funzionalità usando messaggi.

Ciascun approccio risolve essenzialmente il medesimo problema e presenta specifici vantaggi e svantaggi. In molte situazioni l'approccio migliore risulta comunque essere quello dello scambio di messaggi asincroni, approccio che è stato approfondito in questo lavoro. Esso favorisce infatti un basso accoppiamento tra le applicazioni integrate (loose-coupling) ed un alta reliability superando le limitazioni delle comunicazioni remote come la latency ed unreliability [95][25][68].

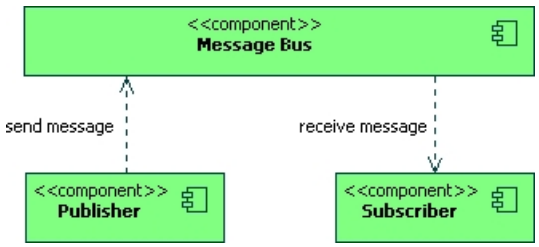

Figura 4.1. Architettura Publisher & Subscriber

In Fig.4.1 viene mostrata l'architettura tipica per lo scambio di messaggi asincroni. In particolare Message Bus rappresenta il componente software in grado di veicolare i messaggi, spediti sul Message Bus, a coloro i quali ne hanno fatto richiesta. Publisher rappresenta un generico componente software in grado di spedire messaggi sul bus. Subscriber rappresenta un generico componente software che riceve dal Message Bus i messaggi richiesti.

Per realizzare l'integrazione tra applicazioni differenti attraverso l'uso di messaggi asincroni occorre che esse, comportandosi da publisher e/o da subscriber, spediscano e/o ricevano messaggi. In realtà la costruzione di una buona soluzione di integrazione prevede spesso, oltre allo scambio di semplici messaggi, anche l'implementazione di una porzione di logica (ad esempio convertitori di formato, filtri, router, etc.) che solo se tenuta fuori dagli applicativi integrati si rende più facilmente riutilizzabile. Le applicazioni si alleggeriscono di molti dettagli relativi all'integrazione che, prendendo la forma di Integration Components (IC), vanno a far parte del middleware. L'architettura di

integrazione, a questo punto costituita da Message Bus e IC, prende il nome di Enterprise Service Bus (ESB) [48][79], vedi Fig.4.2.

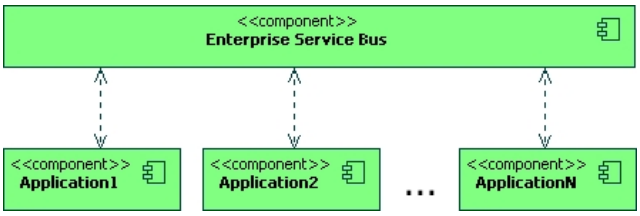

Figura 4.2. Architettura Publisher & Subscriber

Esistono ad oggi vari software per la realizzazione di ESB (es. Mule [71], ServiceMix [34], OpenESB [75]) ognuno dei quali, pur con le proprie caratteristiche differenziali, è comunque in grado di fornire almeno il seguente insieme di funzionalità: location transparency, trasport protocol conversion, message transformation, message routing, message enhancement, security, monitoring and management. In Fig. 4.3 sono mostrati alcuni dettagli relativi all'architettura di integrazione con ESB. In particolare si noti come quest'ultimo contenga al suo interno i vari componenti di integrazione e sia in grado di supportare differenti tecnologie di comunicazione (es. http, jms, jdbc, etc). Ciascuna applicazione avrà al suo interno un componente (Endpoint) che le consentirà di spedire e ricevere dati verso e dall'esterno.

Nel disegnare ed implementare IC è importante riferirsi il più possibile agli Enterprise Integration Patterns (EIP) che sono una collezione di soluzioni robuste per un notevole numero di problemi di integrazione. Alcuni IC sono composti di un solo EIP mentre altri di più di uno. Citando [48]: "Patterns are not copy-paste code samples or shrink-wrapped components, but rather nuggets of advice that describe solutions to frequently recurrent problems. Used properly, EIPs can fill the wide gap between high-vision of integration and the actual system implementation".

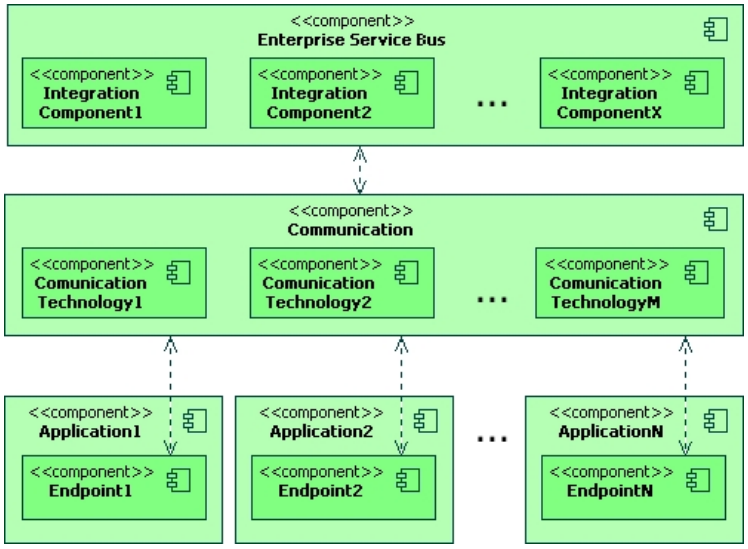

Figura 4.3. Architettura Publisher & Subscriber

4.1.3 Ontologie nelle Enterprise Integration Architecture

Riferendosi allo scambio di dati per l'integrazione tra applicazioni, il progettista si trova di fronte al problema di riconciliare differenti formati utilizzati da differenti applicazioni. Se il numero di formati è basso pochi convertitori di formato possono bastare, al crescere dei formati il numero di convertitori rischia però di aumentare esponenzialmente. Una migliore soluzione risulta essere quella di ricorrere ad un "modello canonico"[48] che è la forma di rappresentazione da e verso cui tutti i differenti formati sono convertiti. L'impiego di ontologie per la realizzazione di detto modello è particolarmente appropriato in quanto consentono di rappresentare la conoscenza in maniera facilmente riusabile e condivisibile tra applicazioni [90]. Un modello ontologico è infatti una rappresentazione formale di un insieme di concetti che afferiscono al dominio di interesse. Le ontologie hanno un alto potere espressivo che, per loro natura, si adatta ai bisogni di rappresentare domini complessi, sono automaticamente elaborabili, inerentemente estendibili e rendono esplicita ed omogenea anche la conoscenza tacita ed embedded. I modelli ontologici nascondono i dettagli di implementazione permettendo a stakeholder, esperti di dominio e sviluppatori di condividere esattamente la stessa rappresentazione del dominio di interes-

se. Nell'implementare IC risulta vantaggioso usare ontologie per derivare il contenuto dei messaggi trasmessi come pure per supportare meccanismi per la validazione del contenuto dei messaggi ricevuti. In altre parole gli IC si occupano degli aspetti tecnici collegati allo scambio di informazioni mentre le ontologie forniscono il significato per validare la loro semantica.

Date le loro caratteristiche, le ontologie possono essere utilizzate non solo per descrivere il modello canonico di una integrazione ma per guidare tutte le fasi di analisi e progettazione; in termini pigenerali possono costituire un valido strumento per la gestione della governance dell'architettura di integrazione.

Nel contesto del lavoro svolto in collaborazione con AOUC, l'esperienza riguardante la cooperazione applicativa si è concretizzata nella realizzazione di un numero di integrazioni (ad esempio quella tra la cartella clinica oncologica ed il sistema di prenotazione delle visite) ed è tuttora in corso per l'implementazione di un sistema di anagrafi unificate attraverso l'ESB ServiceMix. Tali attività hanno evidenziato la necessità di strumenti di governance, principalmente nella forma di strumenti per il controllo e la gestione del formato dei messaggi presenti sull'ESB.

Per realizzare questi strumenti attraverso l'impiego di ontologie e per abilitare lo sviluppo di componenti di integrazione ontologici in grado di operare su piattaforme più tradizionali, si è reso indispensabile creare un componente software in grado di mettere in corrispondenza il formato dei messaggi, oggi definito attraverso file XML Schema, con un sistema di astrazioni di tipo ontologico cioè di uno strumento per la conversione tra OWL ed XML che abbiamo chiamato Owl2Xml.

4.1.4 Modello Ontologico

In Fig. 4.4 viene mostrato il modello dell'ontologia su cui si basa l'applicazione Owl2Xml (Owl2XML Ontology). Tale ontologia può inoltre essere utilizzata per eseguire query semantiche riguardanti gli schemi XSD che descrivono la struttura dei messaggi che circolano su un ESB. Utilizzando l'ontologia Owl2Xml è possibile, ad esempio, richiedere gli XSD che trattano di un cer-

to argomento o fare l'analisi di impatto (quanti schemi di messaggi dovranno essere corretti) relativa alla modifica di un XSD.

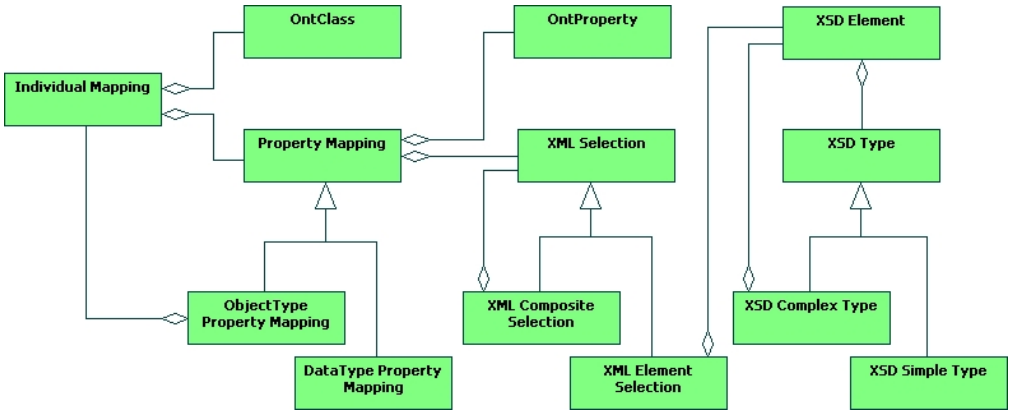

Figura 4.4. Modello di Owl2Xml

Nel modello di Fig. 4.4, la classe `IndividualMapping` serve a rappresentare le informazioni di mapping di un individuo dell'ontologia il cui tipo è determinato dal valore della classe `OntClass` associata. `IndividualMapping` è collegata alle informazioni di mapping delle proprietà dell'individuo, descritte per mezzo della classe `PropertyMapping` che si specializza in `DataTypePropertyMapping` e `ObjectTypePropertyMapping`. `PropertyMapping` è inoltre collegata alla proprietà ontologica (`OntPredicate`) mappata ed al selettore della porzione di un file XML, `XML Selection`, da considerare nel mapping. `XML Selection` ècollegata a `XSD Element`. `XSD Element` e `XSD Type`, infine, servono a rappresentare all'interno dell'ontologia le informazioni contenute nei file XSD.

4.1.5 Casi d'uso

Il diagramma dei casi d'uso di Fig. 4.5 mostra che l'applicazione `Owl2Xml Application` dipende dal modello ontologico `Owl2Xml Ontology` (ne costituisce la parte più rilevante del suo Data Layer). `Owl2Xml Application` offre funzionalità relative alla conversione di messaggi XML in OWL e viceversa. È possibile, ad esempio, configurare `Owl2Xml Application` come subscriber, su

un ESB, di quei messaggi XML che si voglio convertire in OWL e fare in modo che l'ontologia prodotta dalla trasformazione venga serializzata in RDF/XML e spedita da `Owl2Xml Application`, nel ruolo di publisher, sull'ESB. In questo modo si troveranno sul bus non solo i messaggi XSD/XML originari ma anche quelli OWL/RDF prodotti da `Owl2Xml Application` che abiliteranno il funzionamento degli IC ontologici eventualmente realizzati.

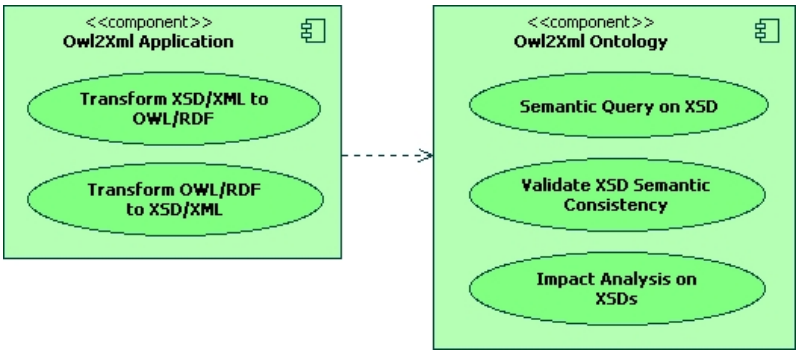

Figura 4.5. Casi d'uso relativi ad Owl2Xml Ontology & Application

Le funzionalità di `Owl2Xml Application` sono un piccolo sottoinsieme dei servizi che si possono implementare basandosi sulla `Owl2Xml Ontology`. Utilizzando quest'ultima diventa ad esempio possibile offrire servizi di validazione e ricerca semantica sugli schemi XSD dei messaggi circolanti su un bus. È facile costruire modelli ontologici che, estendendo `Owl2Xml Ontology` (import di ontologie) [13], riescano a rappresentare, oltre ai tipi dei messaggi, anche tutti gli altri aspetti eventualmente rilevanti per i problemi di EAI, quali ad esempio: l'elenco dei componenti IC esistenti e l'elenco dei software e delle apparecchiature del contesto in cui le integrazioni devono avvenire. L'ontologia `Owl2Xml Ontology`, cioè, può facilmente venire estesa fino a diventare lo strumento più idoneo a rappresentare l'intera base di conoscenza necessaria al processo di governance di un'architettura per l'EAI.

4.1.6 Sperimentazione

L'ontologia Owl2Xml e la corrispondente applicazione per le conversioni sono state sperimentata riferendosi alla RFC Comet di Regione Toscana [101] che descrive il formato XSD dei messaggi che circolano sul bus applicativo regionale CART [100] e che riguardano le sperimentazioni cliniche. Studiando gli XSD è stato possibile definire l'ontologia per rappresentare i concetti descritti nella RFC ed utilizzare tale astrazione come modello verso cui fare il mapping dei messaggi XML.

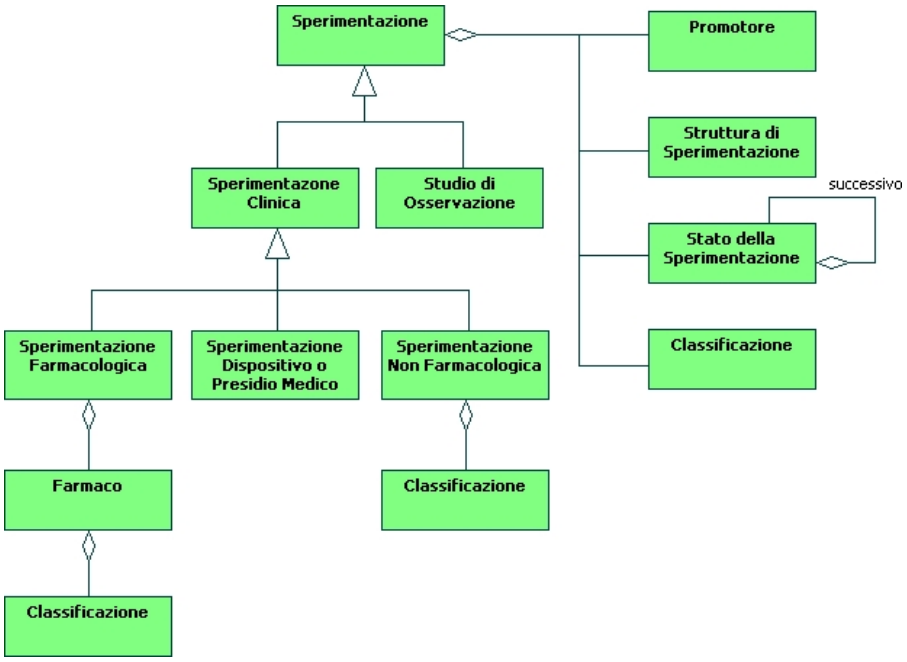

Figura 4.6. Modello delle Sperimentazioni in COMET

In Fig. 4.6 viene mostrato il frammento dell'ontologia che descrive il concetto di `Sperimentazione` che può specializzarsi in: `Studi di Osservazione` e `Sperimentazioni Cliniche`, le quali possono ulteriormente specializzarsi in: `Sperimentazioni Farmacologiche`, `Sperimentazione di Dispositivi o Presidi Medici`, e `Sperimentazioni Non Farmacologiche`.

Ogni `Sperimentazione` ha un `Promotore` ed una `Struttura di Sperimentazione`, il processo di sperimentazione passa attraverso una successione

di stati logici (`Stato della Sperimentazione`). Le `Sperimentazioni Farmacologiche` riguardano un `Farmaco`. `Farmaci`, e `Sperimentazioni` possono essere classificati in più modi, cioè rispetto a più criteri di classificazione.

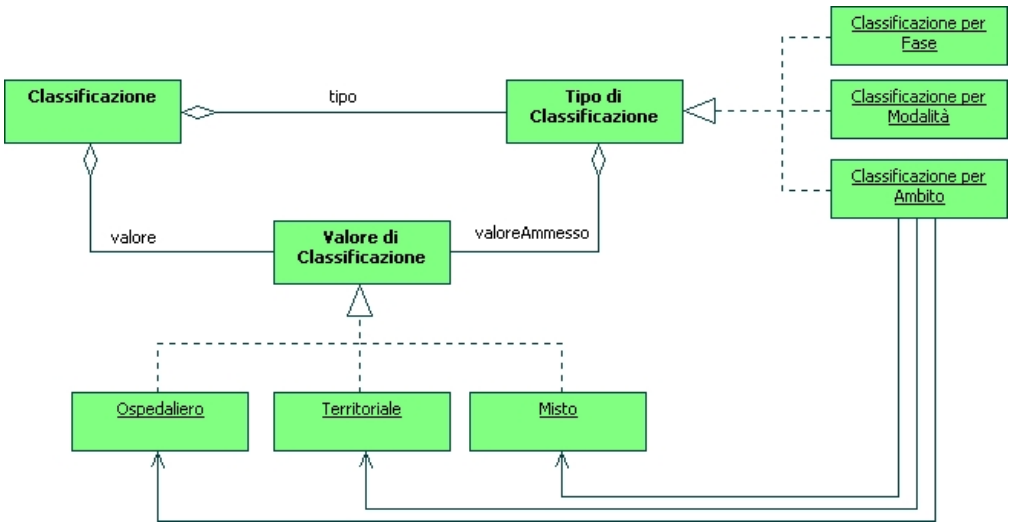

Figura 4.7. Modello per la classificazione di entità in COMET

In Fig. 4.7 viene mostrato il modello per descrivere la classificazione di entità in COMET. `Classe` è la misurazione di un fenomeno qualitativo, il `Tipo di Classificazione`, e pertanto è caratterizzata dal tipo e dal valore assunto (l'unità di misura ed il valore di misura). Ogni `Tipo di Classificazione` ammette un ben determinato insieme di valori ammissibili. Nell'esempio vengono mostrati i tipi di classificazione: per fase, per modalità e per ambito. In particolare per quest'ultimo tipo di classificazione viene mostrato anche l'insieme di valori ammissibili: ambito ospedaliero, ambito territoriale e misto.

In Fig. 4.8 viene mostrata la classe `Stato della Sperimentazione` che permette di descrivere l'automa per rappresentare gli stati logici in cui può trovarsi una `Sperimentazione` e le transizioni valide tra di essi. In particolare una sperimentazione inizialmente si trova nello stato di proposta ed eventualmente passa a quello di richiesta integrazioni e poi di integrazioni ricevute, fino a concludersi con l'approvazione o meno.

Infine la Fig. 4.9 mostra le classi relative agli `Enti`, pubblici (`Ente Pubblico`) o privati (`Ente Privato`) ed ai ruoli, `Promotore` e `Struttura di`

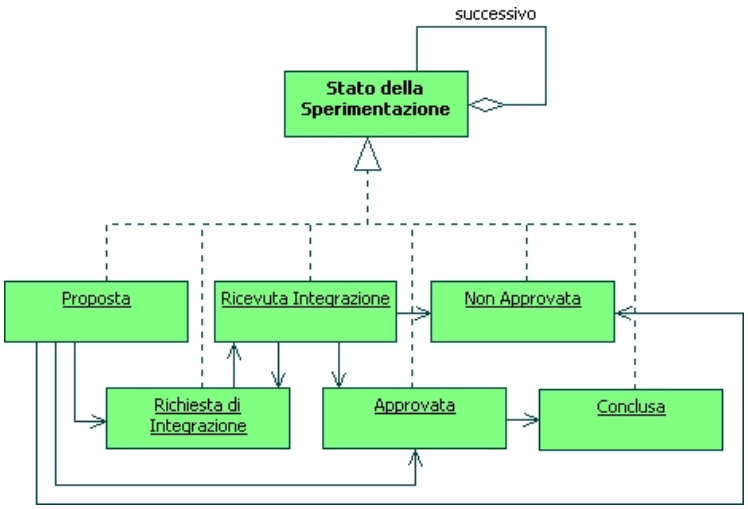

Figura 4.8. Modello per la descrizione degli stati logici delle Sperimentazioni

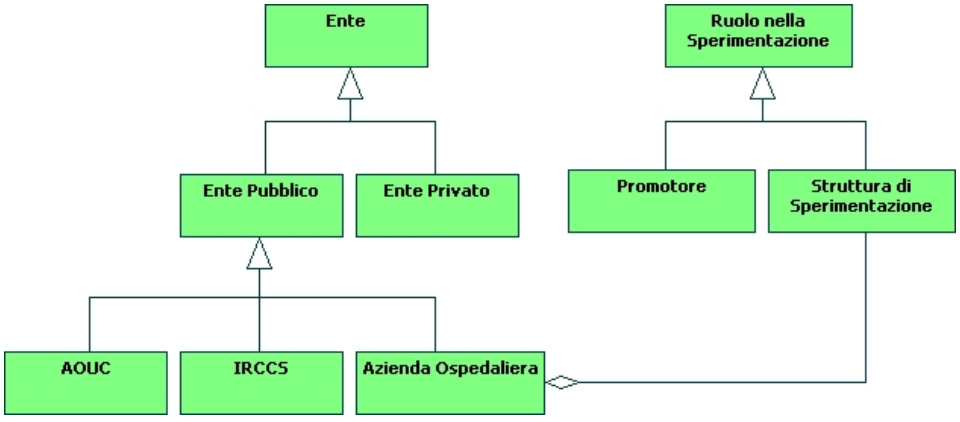

Figura 4.9. Modello per la descrizione degli stati logici delle Sperimentazioni

`Sperimentazione` che sono coinvolti nella `Sperimentazione`.

La modellizzazione ontologica ha permesso di ottenere una rappresentazione compatta e leggibile dei concetti della RFC Comet evidenziando come le ontologie ben si prestino a descrivere modelli da condividere con gli esperti di dominio. L'ontologia Owl2Xml ha consentito di rappresentare agevolmente le relazioni di mapping tra il modello OWL di COMET ed i suoi XSD.

4.2 Empedocle

Empedocle è il software di gestione informatizzata di cartelle cliniche che abbiamo realizzato nel contesto della collaborazione con l'AOUC e che a breve sarà completato per essere utilizzato nella pratica quotidiana dal personale medico .

Empedocle trae spunto dall'implementazione di un nostro prototipo per la gestione di cartelle cliniche con tecnologie ontologiche [12]. È di tale prototipo che viene riportata l'esperienza.

4.2.1 Diagnosi Medica

La diagnosi medica è il processo che cerca di identificare le malattie di cui soffre un paziente sulla base di sintomi, segni e della sua storia medica (col termine segni in medicina si intendono quei sintomi che sono oggettivamente rilevabili e misurabili secondo determinati parametri, ad esempio uno stato febbrile dalla misura di temperatura).

Di solito il medico conduce una prima valutazione della situazione clinica del paziente considerando le patologie compatibili con determinati sintomi e segni, poi procede con la diagnosi differenziale che è un processo sistematico ed iterativo di pesatura della probabilità di una patologia rispetto a quella di altre ugualmente compatibili fino alla identificazione della diagnosi più probabile relativa al quadro clinico presentato. Durante questo processo il medico può aver bisogno di far eseguire al paziente uno o più accertamenti.

La quantità di informazioni che deve essere presa in considerazione nella diagnosi medica è enorme [81] e sono stati introdotti nella prassi clinica molti sistemi informativi basati sull'intelligenza artificiale per aiutare il medico [61], [67]. I cosidetti Sistemi Esperti sono software capaci di effettuare deduzioni facendo riferimento alla conoscenza (anche incompleta) fornita da esperti di dominio. Validare e mantenere aggiornata la conoscenza può tradursi in un pesante onere, dal momento che il settore medico è estremamente vasto ed in continua evoluzione. Non è ragionevole aspettarsi che, raggruppando un insieme di esperti, questi riescano a fornire un insieme di dati completo e definito su cui basare un sistema informatico. Occorre un approccio diverso: è necessario costruire sistemi che permettano ad utenti qualificati di contribuire al software con la loro conoscenza in modo continuativo e senza bisogno dell'intervento degli sviluppatori. La contribuzione di nuova conoscenza deve poter riguardare sia l'aspetto intensionale che quello estensionale e deve avvenire nel rispetto delle modellizzazioni e dei dati precedentemente immessi.

Questo è reso possibile attraverso l'uso di ontologie [115][74][23] che , favorendo un approccio incrementale che accompagni l'evoluzione verso modelli condivisi, soddisfano le esigenze crescenti di dati e di integrazione in campo biomedico. Sono infatti disponibili un certo numero di ontologie di dominio [74] come pure sono disponibili strumenti off-the-shelf per ricerca, validazione ed inferenza su modelli ontologici [1][7][17][69][91][38]. La logica del primo ordine fornisce le basi teoriche per l'analisi inferenziale ma purtroppo questo porta ad algoritmi che sono deterministici per natura. Il limite delle ontologie è in realtà la loro incapacità di trattare con l'incertezza che è invece un aspetto tipico dei processi diagnostici; le Bayesian Networks (BN) offrono una rappresentazione coerente e intuitiva di conoscenza di dominio incerta[114]. Una BN è costituita da una struttura a grafo che codifica le variabili aleatorie provenienti dal dominio dell'applicazione insieme alle relazioni di influenza tra di loro ed alle tabella di probabilità condizionata su queste variabili. Costruire una BN prevede tre compiti fondamentali[31]. Il primo è di individuare le variabili che sono importanti nel dominio, insieme ai loro valori possibili. Il secondo è di individuare le relazioni tra queste variabili e di esprimerle in

una struttura a grafo. L'ultimo compito è quello di ottenere le probabilità che sono necessarie per la determinazione quantitativa della rete. Queste attività vengono in genere eseguite con l'ausilio di esperti di dominio.

In [46] viene proposta una metodologia di ingegneria della conoscenza per la costruzione ed il mantenimento di BN. Secondo gli autori un'ontologia costituisce la base di conoscenza di dominio condivisa e distribuita che attraverso una successione di passi serve a derivare la struttura del grafo della BN. A causa della complessità e dell'importanza delle scelte legate alla strutturazione della BN, l'ingegnere deve effettuare manualmente la derivazione del grafo dall'ontologia e la metodologia in oggetto gli fornisce indicazioni e supporto nel prendere le decisioni. Questa metodologia è stata applicata al dominio del cancro esofageo.

La costruzione automatica delle BN è stata trattata in [26] con riferimento al dominio delle telecomunicazioni. La costruzione della rete comprende essenzialmente le stesse attività descritte in [31] e vengono sfruttati il potere espressivo e le capacità di inferenza delle ontologie per generare automaticamente BN che possano essere di supporto a processi decisionali. Il modello ontologico della conoscenza di dominio viene esteso con concetti relativi alle BN e per creare una BN sono definite opportune istanze della classe <BayesianNetworkNode>.

In [118] viene presentata un'ontologia relativa alle Linee Guida nella Pratica Clinica. Questa ontologia contiene anche dati che permettono la generazione automatica delle tabelle di probabilità condizionata delle corrispondenti BN. L'approccio non è inteso per generare BN generiche ma piuttosto riguarda la costruzione automatica di reti nella forma richiesta dalla specifica applicazione.

Fenz et al. [33] seguono un approccio simile a quello di [26] costruendo però la BN direttamente da ontologie pubbliche di dominio esistenti, evitando di sviluppare un'ontologia specifica contenente estensioni Bayesiane. Le funzioni per il calcolo delle tabelle di probabilità condizionata non sono fornite dall'ontologia e devono essere modellate esternamente; l'intervento umano è inoltre necessario se l'ontologia pubblica descrive un modello che non si adatta perfettamente al dominio di interesse.

In [21] e [22] vengono proposte estensioni di OWL al fine di rendere possibilie la rappresentazione di conoscenza probabilistica. L'idea e quella di costruire un framework che permetta la rappresentazione di conoscenza incerta nel campo dell'ingegneria delle ontologie. Il linguaggio, chiamato PR-OWL, estende sia la sintassi che la semantica di OWL, rendendo possibile esprimere l'incertezza riguardo a tutte le forme di conoscenza contenuta nel modello ontologico. L'estensione proposta permetterebbe alle ontologie legacy di interoperare con le nuove ontologie probabilistiche. Gli autori sostengono che le estensioni proposte dovrebbero far parte di una nuova versione di OWL. Certamente questo implicherebbe modifiche agli strumenti correnti, dagli editor ai reasoner.

Estensioni ad OWL per includere le BN sono proposte in [27] [28]. Con questo fine viene definito un linguaggio di markup probabilistico. Le ontologie OWL annotate con informazioni probabilistiche sono convertite in BN attraverso un insieme di regole strutturali di conversioni. La rete ottenuta può riprodurre accuratamente i ragionamenti tipici sui modelli ontologici e trattare quelli probabilistici come inferenze Bayesiane.

Una proposta simile ma meno invasiva viene fatta in [116] e [117] dove le annotazioni sono usate per specificare le probabilità associate alle classi e alle proprietà. Le relazioni di dipendenza tra variabili aleatorie sono esplicitamente specificate attraverso la proprietà <dependsON> appositamente introdotta per marcare l'informazione delle dipendenze in una ontologia OWL.

Come mostrato, numerosi articoli propongono l'uso di BN insieme a quello di ontologie. Sinteticamente, essi tendono a seguire un approccio in cui i costrutti dell'ontologia sono interpretati in maniera probabilistica così da esprimere l'incertezza riguardo alla conoscenza contenuta nel modello. Le BN sono solitamente generate in maniera semi-automatica dai modelli ontologici [33][46]. Molti autori [22][27] propongono estensioni al linguaggio usate per descrivere l'ontologia (ad esempio OWL).

In questa sezione discutiamo dell'uso delle ontologie e delle BN per la diagnosi medica seguendo un approccio in qualche modo diverso che è stato delineato dopo la considerazione che un algoritmo di generazione delle BN specifico per il dominio conduce a modelli ontologici più semplici ed a BN su misura

per gli obiettivi applicativi. Oltre a definire il modello per lo specifico dominio abbiamo definito anche un template per le reti necessarie al processo diagnostico. Così facendo le BN sono generate in maniera completamente automatica mentre l'ontologia è gestita in maniera usuale adottando OWL standard.

Abbiamo sviluppato un prototipo in cui la conoscenza di dominio può essere progressivamente estesa da utenti accreditati che contribuiscono con conoscenza aggiuntiva. All'ontologia non viene richiesto di modellare quegli aspetti che sono necessari per costruire il grafo poiché le BN sono prodotte in una forma predefinita.

Tale prototipo è in grado di stimare la probabilità che un paziente sia affetto da una data patologia ed è anche capace di suggerire test aggiuntivi come pure di calcolare la loro utilità ai fini diagnostici (per questa parte sono generate *decision network* come estensione di BN) [12].

4.2.2 Modello Ontologico

In Fig. 4.10 viene mostrato il modello che, nel nostro prototipo, rappresenta parte dell'ontologia rilevante ai fini della diagnosi medica.

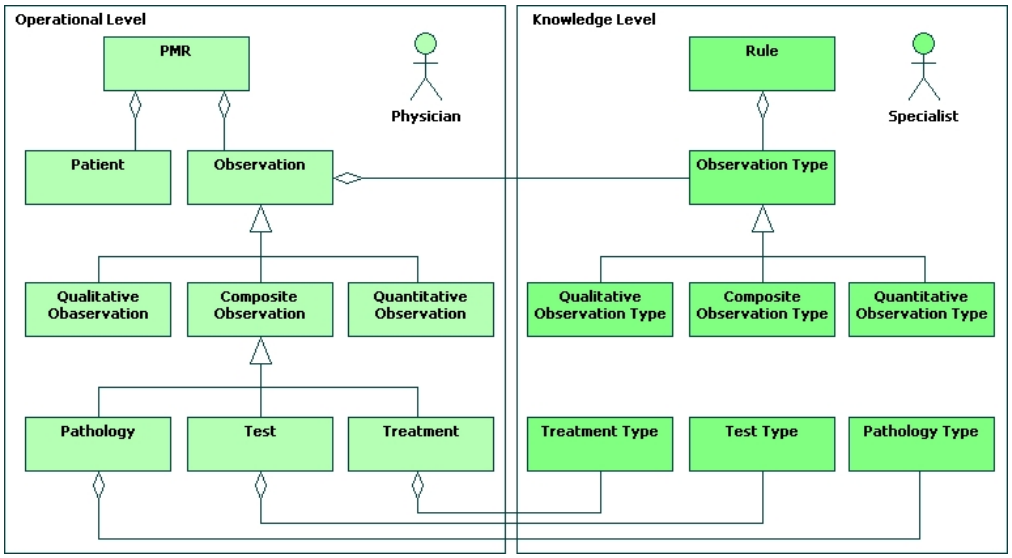

Figura 4.10. Modello ontologico per la diagnosi medica

La schematizzazione corrisponde al pattern "Observation" degli Analysis Pattern [35]. Questo pattern ha una caratteristica importante: separa la parte operazionale da quella di conoscenza, fornendo in questo modo un paradigma per implementare un'ontologia in cui le osservazioni rappresentano la parte estensionale mentre i tipi di osservazione rappresentano la parte intensionale dell'ontologia stessa. Gli Analysis Pattern offrono soluzioni pronte all'uso per diversi problemi di modellazione e quindi riferirsi ad essi riduce la complessità inerente la costruzione di un'ontologia in ambito medico. In 4.10 `PMR` sta per Personal Medical Record, mentre `Observation` è un concetto generico per rappresentare sintomi, segni ed altri fattori e fatti relativi alla salute del paziente inclusi i trattamenti passati e l'eventuale predisposizione a malattie. I due attori, il medico (`Physician`) e lo specialista clinico (`Specialist`) interagiscono con la base di conoscenza.

Lo specialista è responsabile della contribuzione di nuova conoscenza che estenda la parte intensionale dell'ontologia. Ad esempio, può aggiungere un nuovo tipo di osservazione come pure nuove patologie a seguito della crescita delle conoscenze mediche.

Lo specialista può inoltre definire regole di inferenza per un reasoner. Ad esempio una regola riguardante il fatto che "se il paziente ha una temperatura maggiore di 39°C, allora ha febbre alta" può essere espressa (in linguaggio pseudo-formale) come:

Patient(?p) $\wedge$ hasObservation(?p,?t)
$\wedge$ TemperatureObservation(?t) $\wedge$?t.value $>$ 39°C
$\rightarrow$ hasObservation(?p,?f) $\wedge$ FeverObservation(?f)
$\wedge$?f.value $=$ HIGH

Il medico è responsabile per l'esecuzione della diagnostica e questa attività conduce ad aumentare la parte estensionale dell'ontologia con nuovi fatti (osservazioni).

Il medico può inoltre interrogare la base di conoscenza (estesa con le deduzioni che il reasoner ha fatto applicando le regole definite dallo specialista) per ottenere i dati che considera utili per formulare una diagnosi.

Le regole di inferenza possono essere usate dal reasoner per: (a) validare la

consistenza delle osservazioni introdotte dal medico; (b) dedurre nuovi fatti (osservazioni) che vadano ad estendere la base di conoscenza; (c) eseguire automaticamente quelle diagnosi che dipendono dalla base di conoscenza.

Con riferimento al punto c), i tipi di diagnosi che è possibile fare con l'ontologia descritta sono quelle che derivano da ragionamenti deterministici. Questo è sicuramente utile in quelle situazioni in cui dati alcuni sintomi è direttamente possibile dedurre la presenza od assenza di una patologia, ma questo accade molto di rado in medicina. Citando da [82]:

"Il tentativo di usare la logica del primo ordine per gestire il dominio di diagnosi medica fallisce per tre motivi principali: (a) Pigrizia: l'elencazione dell'insieme di tutti gli antecedenti o conseguenti necessari per garantire una regola senza eccezioni è troppo laborioso e, la regola risultante, troppo difficile da usare. (b) Ignoranza teorica: la scienza medica non ha una teoria completa per il domino. (c) Ignoranza pratica: anche se conoscessimo tutte le regole, potremmo essere incerti a proposito di un paziente particolare perché tutti i test necessari non sono stati fatti o non è stato possibile farli."

Riassumendo, le ontologie sono eccellenti per rappresentare domini ampi e complessi come quello della medicina ma falliscono quando sono necessari ragionamenti probabilistici. In altre parole un sistema che gestisca diagnosi mediche deve essere capace di gestire l'incertezza. Le BN sono usato per questo scopo.

4.2.3 Bayesian Networks

Riferendosi al modello di figura 4.10 la domanda è: qual'è l'informazione che deve essere inclusa nella base di conoscenza per permettere la generazione e l'uso di BN per la diagnosi medica?

Per rispondere a questa domanda devono essere considerati due aspetti: a) la struttura della rete, cioè la topologia del grafo rappresentante la rete; b) i parametri caratterizzanti le entità che sono comprese nella rete.

Riguardo al punto a), i requisiti di modellazione posti dai medici durante la fase di analisi ci hanno portato a proporre una semplice rete a due livelli

(da sintomi a patologie). In questa rete la probabilità che una data patologia sia presente è condizionata dalla probabilità delle osservazioni. Le osservazioni possono rappresentare fattori rilevanti per il processo diagnostico inclusi sintomi, segni, fattori di predisposizione, risultati di accertamenti, effetto di trattamenti, condizioni di patologie multiple e fenomeni che evolvono nel tempo. L'importante conseguenza di scegliere una rete con struttura predefinita nel nostro sistema è che la generazione delle BN è delegata ad un semplice algoritmo, descritto sotto, evitando di rappresentare ogni aspetto strutturale nell'ontologia.

Riguardo al punto b), poiché la struttura della rete è demandata all'algoritmo predefinito, quel che è necessario per popolare l'ontologia e permettere la produzione di BN si riduce alle correlazioni tra patologie ed osservazioni che sono date attraverso tabelle di probabilità a priori e condizionate.

L'algoritmo lavora come segue:

1. Aggiunge le osservazioni relative al paziente alla base di conoscenza ed esegue il reasoner per derivare eventuali ulteriori osservazioni in accordo alle regole di inferenza

2. Per tutti i tipi di osservazioni corrispondenti all'insieme di osservazioni asserite e derivate estrae tutti i tipi di patologie compatibili e così costruisce la BN a due livelli. Aggiunge poi ai nodi della rete le probabilità a priori e condizionate contenute nella base di conoscenza.

3. Cambia la probabilità dei nodi in accordo alle osservazioni asserite e derivate e calcola le probabilità a posteriori.

4.2.4 Decision Networks

Nell'eseguire la diagnosi il medico può richiedere al paziente di sottoporsi ad accertamenti medici aggiuntivi. Questo pone il problema di valutare qual è il miglior accertamento da fare. La ricerca dell'accertamento più utile è fatta per mezzo di "Decision Networks" (DN), un'estensione delle BN che aggiunge nodi

di decisione e di utilità. Graficamente i nodi di decisione sono rappresentati da rettangoli mentre i nodi di utilità sono nella forma di rombi.

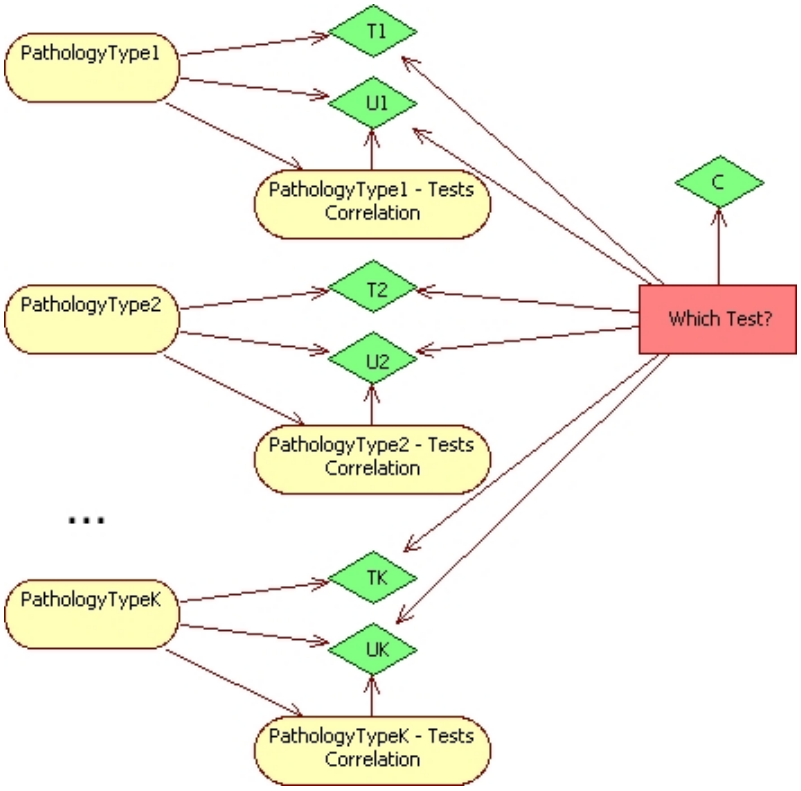

Figura 4.11. Rete di decisione per una patologia

L'utilità di un accertamento è valutata rispetto a:

a) indici di utilità calcolati attraverso una semplice funzione di costo, dipendenti solo dal tipo di test, che includono costo ed invasività dell'accertamento. Il corrispondente nodo di utilità è etichettato C in Fig. 4.11;

b) indici di utilità, calcolati tenendo conto sia della patologia che del tipo di accertamento, che includono l'abilità di identificare le patologie più serie tra quelle prese in considerazione ed il tempo per ottenere i risultati. I nodi di utilità sono etichettati Ti in Fig. 4.11;

c) indici di utilità collegati alla sensitività ed alle specificità di un test rispetto ad una patologia. I corrispondenti nodi di utilità sono etichettati come Ui in Fig. 4.11.

In Fig. 4.11 i nodi PathologyType (1, 2, ... k) rappresentano la probabilità che un dato paziente abbia quelle patologie. Queste probabilità risultano dall'inferenza probabilistica fatta attraverso le BN già discusse. Il rettangolo `Which Test?` è il nodo di decisione per gli accertamenti medici; in pratica questa è una variabile i cui valori identificano gli accertamenti che sono rilevanti date le patologie considerate e l'insieme di valori ammessi è ottenuto interrogando il modello ontologico. I nodi PathologyType - Test Correlation dipendono dal tipo di patologia e dal tipo di accertamento e modellano: (a) sensitività: la probabilità che un accertamento produca un risultato positivo quando eseguito su pazienti affetti da una determinata patologia; (b) specificità: la probabilità che un test produca risultato negativo quando eseguito su pazienti non affetti da quella stessa patologia. Notare che in Fig. 4.11 i test sono associati ad un nodo di utilità che dipende solo da essi (rombo C); i nodi di utilità Ti quantificano l'utilità degli accertamenti relativamente alla loro urgenza rispetto ad una patologia; infine i rombi Ui, che dipendono dalla presenza/assenza di una patologia come pure dalla sensitività e specificità di un accertamento, misurano l'efficacia di un test per le varie patologie. Nel nostro sistema, la struttura delle DN è predefinita per essere come quella di Fig. 4.11 (la sola possibile variazione è l'assenza di uno o più nodi di utilità se il relativo indice di utilità non è considerato di interesse).

Il medico interagisce col sistema specificando il quadro clinico di interesse e richiedendo il calcolo dell'utilità degli accertamenti. Il sistema, utilizzando l'informazione fornita dal medico, formula un'insieme di interrogazioni sull'ontologia aumentata con le deduzioni del reasoner e impiega i risultati ottenuti per generare una DN che adoprerà per svolgere i ragionamenti probabilistici e quantitativi. L'algoritmo per la generazione delle DN è simile a quello per la generazione delle BN.

4.2.5 Modello per la Diagnosi Medica

Per costruire le BN, l'ontologia deve comprendere le correlazioni tra patologie ed osservazioni. Prima di introdurre queste correlazioni occorre spendere alcune parole su osservazioni qualitative e quantitative (quelle composite sono un mix delle due). In Fig. 4.10 le osservazioni sono classificate come qualitative o quantitative. Un'osservazione quantitativa può essere "la temperatura del paziente è 37.5°C", mentre un'osservazione qualitativa può essere "il paziente ha la febbre" o più precisamente "il paziente ha la febbre bassa". Nella pratica clinica, il valore esatto di un parametro ha meno significato di quello dell'intervallo di valori in cui il parametro cade. Queste considerazioni, ed il fatto che il trattamento di osservazioni quantitative richiederebbe l'introduzione di distribuzioni di probabilità (invece di probabilità discrete), sono le motivazioni per discretizzare le osservazioni quantitative. Questo viene fatto attraverso regole di inferenza definite dallo specialista così da eseguire una discretizzazione basata sulle correlazioni con le patologie. Per esempio una temperatura di 37.5°C può essere classificata bassa o media in base alla patologia. In altre parole, l'interpretazione delle osservazioni quantitative produce osservazioni qualitative, motivo per cui nel modello di figura 4.12 ci sono solo osservazioni di tipo qualitativo.

In accordo con [35], in Fig. 4.12, è stato introdotto il concetto di fenomeno. Fenomeno rappresenta l'insieme di valori discreti che un'osservazione qualitativa può prendere. La correlazione è quindi tra `Pathology Type` e `Phenomenon`; questa non è altro che una tabella di probabilità condizionata. In figura viene mostrata anche la correlazione tra `Treatment Type` e `Phenomenon` che non è però oggetto di questa tesi. Inoltre la figura modella la correlazione tra patologie ed accertamenti che è adoperata per rilevare sensitività e specificità.

4.2.6 Sperimentazione

Il prototipo realizzato utilizza un'ontologia come repository di conoscenza relativa all'ambito medico. L'ontologia può essere estesa dagli specialisti nel corso dell'uso del sistema. A questo fine, agli utenti accreditati viene mostrata un'in-

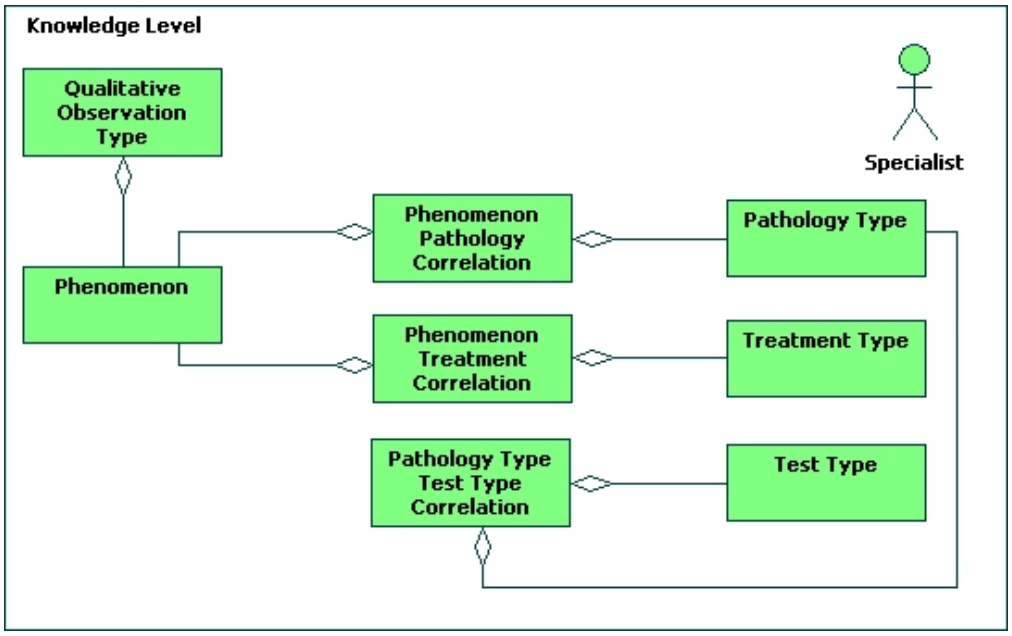

Figura 4.12. Correlazioni per l'analisi Bayesiana

terfaccia attraverso la quale è possibile aggiungere nuove patologie e sintomi insieme alle probabilità che caratterizzano le loro dipendenze. Un reasoner fornisce il supporto per validare la consistenza dei dati introdotti come pure deriva proprietà e relazioni tra essi, permettendo in questo modo un'espansione della conoscenza. L'ontologia è trattata in maniera convenzionale così da impiegare gli standard esistenti come OWL. Per eseguire l'analisi bayesiana sono state introdotte correlazioni probabilistiche tra osservazioni e patologie. Abbiamo deciso di utilizzare sempre una BN a due livelli e DN con struttura predefinita. Questo semplifica la generazione delle reti e riduce la quantità di informazione da archiviare nella base di conoscenza. Per generare e valutare BN e DN sono state impiegate librerie off-the-shelf le cui funzionalità sono richiamate sulla base del tipo di rete che il medico vuol usare.

In Fig. 4.13, 4.14 e 4.15 sono mostrati gli screenshot delle pagine che permettono l'immissione dei sintomi corrispondenti al quadro clinico del paziente.

In Fig. 4.16 viene mostrato uno screenshot della pagina che, dato il quadro clinico di un paziente, presenta: l'elenco delle patologie compatibili ognuna col

Figura 4.13. Pagina per la scelta del sintomo da aggiungere al quadro clinico

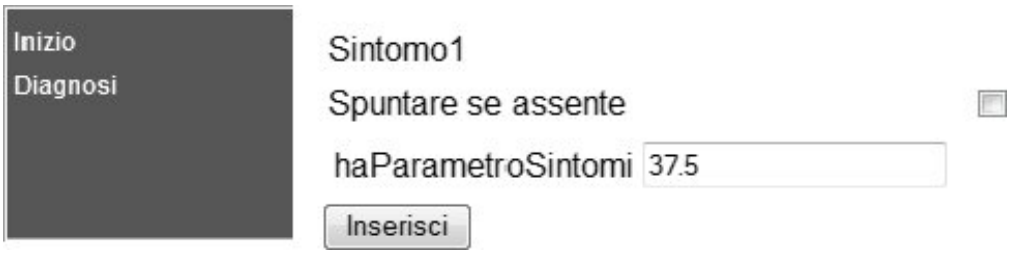

Figura 4.14. Pagina per l'immissione dei parametri caratterizzanti un sintomo

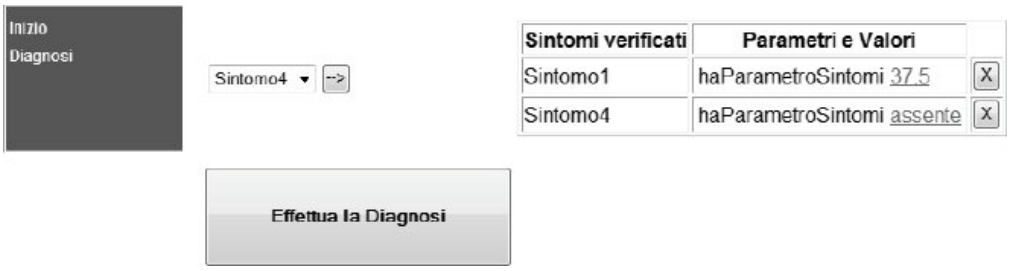

Figura 4.15. Pagina che mostra l'elenco dei sintomi immessi

Figura 4.16. Pagina che mostra la probabilità delle patologie e l'utilità degli accertamenti, dato il quadro clinico

relativo valore di probabilità, l'elenco degli accertamenti che potrebbero servire a procedere con la diagnosi differenziale (col corrispondente indice di utilità) e l'elenco di sintomi che, calcolate le patologie più probabili, potrebbero essere presenti ma che ancora non sono stati rilevati.

Infine in Fig. 4.17, 4.18 e 4.19 sono mostrati alcuni screenshot corrispondenti a pagine del software di cartella clinica informatizzata che è stato sviluppato grazie all'analisi ed all'implementazione del prototipo ontologico.

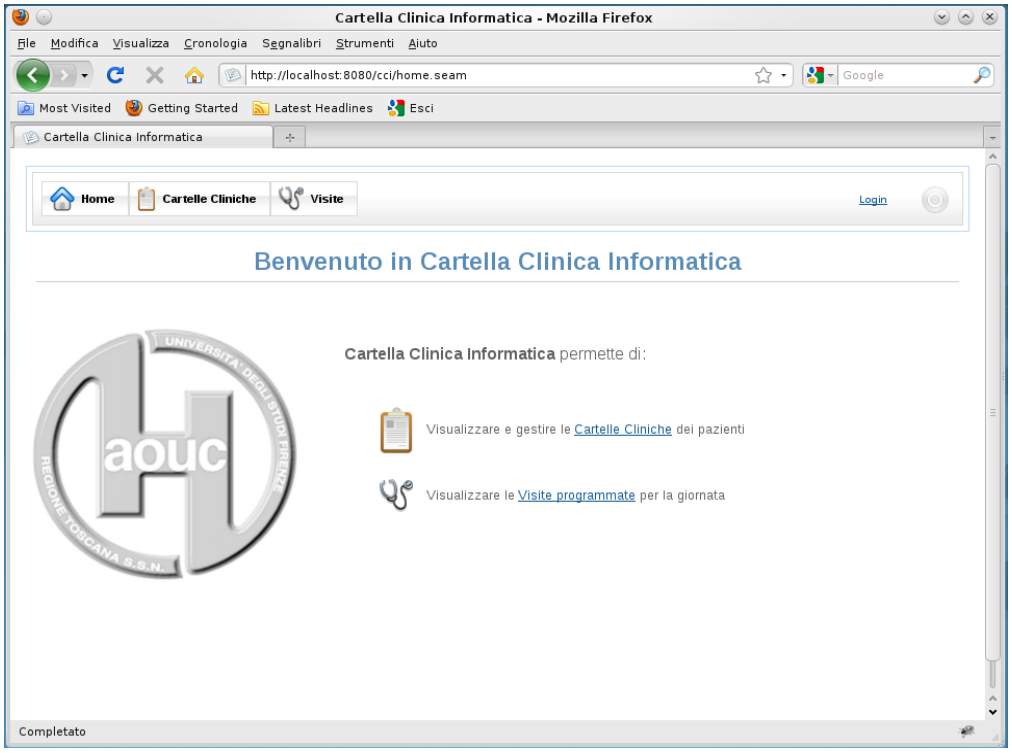

Figura 4.17. Pagina Home di Empedocle

4.3 Carepedia

Carepedia è il software che serve a costruire un portale web basato su di un repository di documenti indicizzati, navigabili e ricercabili in base al lo-

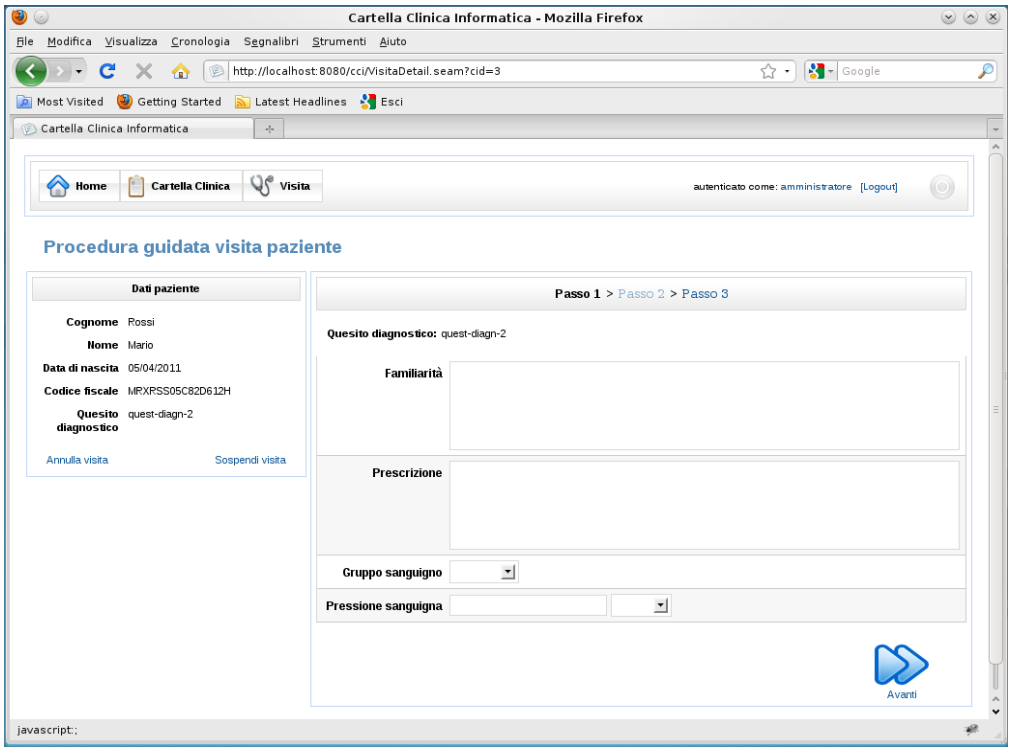

Figura 4.18. Pagina per l'erogazione di una visita con Empedocle

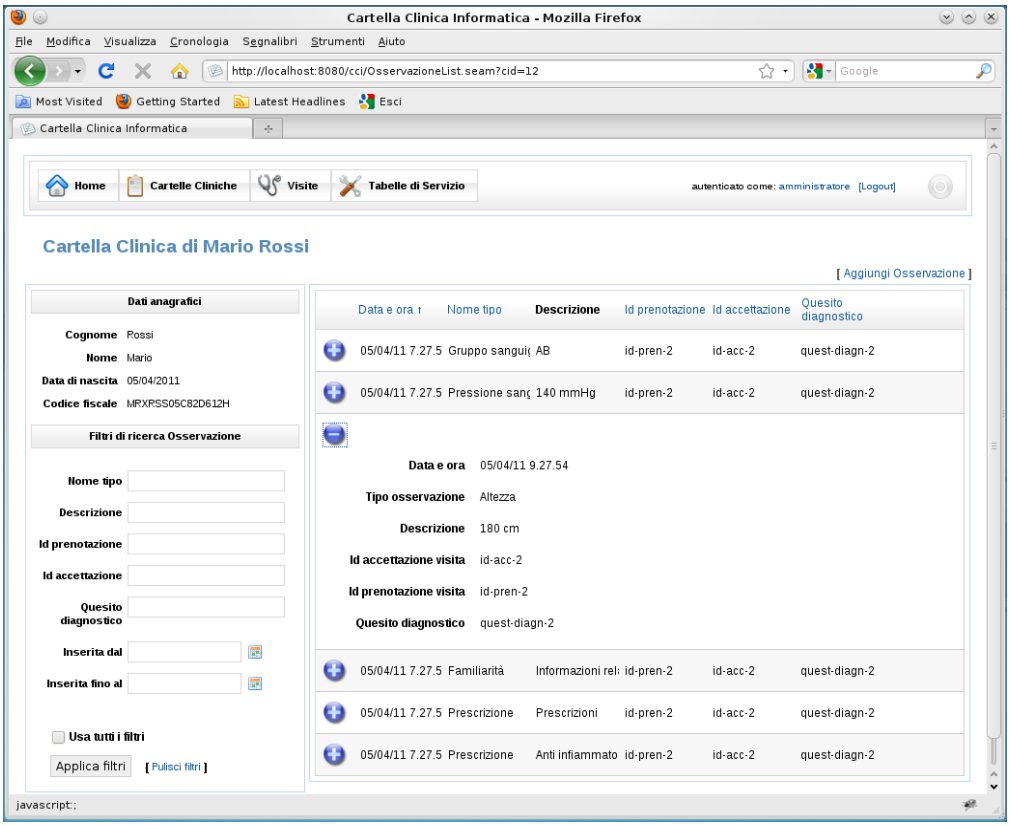

Figura 4.19. Pagina per la consultazione della cartella clinica di Empedocle

ro contenuto semantico che viene descritto da un'ontologia a cui gli utenti contribuiscono in modo distribuito e collaborativo.

Carepedia, sviluppato in contesto ospedaliero nella nostra collaborazione con l'AOUC, non è limitato al pur ampio dominio applicativo della medicina e qui ne riportiamo un esempio d'uso relativo all'indicizzazione e alla ricerca semantica di documenti ed articoli scientifici sulle Reti di Petri.

4.3.1 Semantic Web Portals

Le tecnologie per la modellazione ed il ragionamento ontologico delineano un nuovo paradigma per l'organizzazione di architetture software con alto grado di interoperabilità, manutenibilità ed adattabilità [63][49][86][73][85].

Per loro natura, le ontologie permettono la rappresentazione di modelli semantici combinando la non ambiguità della specifica tecnica con la comprensibilità richiesta per colmare la distanza tra i tecnici e gli altri soggetti interessati; le ontologie abilitano l'applicazione di metodi e strumenti per il reasoning necessario per la ricerca di informazione, la validazione di modelli, l'inferenza e la deduzione di nuova conoscenza[7]; le ontologie si adattano ad un contesto distribuito, abilitando creazione di modelli riusabili, composizione e riconciliazione di frammenti sviluppati in maniera concorrente e distribuita [97][96]; ultimo, non per importanza, le ontologie possono modellare domini che evolvono, incoraggiando in questo modo un approccio incrementale che può condurre all'evoluzione verso modelli condivisi.

In particolare, questo potenziale appare adatto all'organizzazione di portali web, dove le ontologie forniscono un paradigma per progettare, gestire e mantenere information architecture, site structure, and page layout [37][50][14][9]. Questo ha una rilevanza cruciale nella costruzione di portali (vedere [65] e [29]) con una organizzazione di tipo web [76] in cui le pagine sono organizzate secondo il pattern del grafo generico e la navigazione è guidata dalla semantica inerente il contesto più che da una classificazione gerarchica definita a priori [11].

In [87] viene presentato un portale che supporta un accesso unificato ad una varietà di risorse di cultural heritage classificate in accordo ad una ontologia pubblica unificante. Il portale è sviluppato usando tecnologie ontologiche standard e SWI-prolog per supportare la ricerca semantica e la presentazione dei dati recuperati.

In [92] un portale basato su tecnologie ontologiche è usato come base per supportare la contribuzione di contenuti da parte di una comunità distribuita. L'architettura proposta assume che la contribuzione sia limitata agli individui e sono inoltre studiate differenti tecniche per la determinazione di un indice di rilevanza per i result-set di query semantiche.

In [53] viene proposto un approccio dichiarativo alla costruzione di portali semantici che si basa sulla definizione di un insieme di ontologie create dal programmatore del portale per definire i concetti e i contenuti di dominio, la struttura di navigazione e lo stile di presentazione.

In [19] l'approccio dichiarativo di [53] è esteso in un framework basato su ontologie che supportano la costruzione di applicazioni web combinando informazione e servizi. Il framework implementa il pattern architetturale Model View Controller [84]. Mentre il modello è dichiarato usando le ontologie, le viste sono implementate con tecnologie di presentazione esistenti ed in particolare JSP, che principalmente si basa sul paradigma OO. Per colmare la distanza tra i due paradigmi [89] allo sviluppatore viene fornito un insieme di componenti JSP predefiniti che si assumono la responsabilità dell'adattamento.

4.3.2 Partecipanti del Processo di Sviluppo

La struttura di Carepedia supporta la separazione di interessi tra quattro differenti ruoli: Domain Expert, Ontology Expert, Stakeholder ed IT Expert. Questi naturalmente si adattano ad un contesto realistico di sviluppo [16] e fondamentalmente corrispondono ai ruoli identificati in [97].

Domain Expert è quella persona che conosce il dominio del portale e condivide modelli parzialmente formalizzati con la comunità cui appartiene. I Domain Expert solitamente usano strumenti specifici per fare analisi e produrre

risultati di ricerca ma spesso accade che non sappiano niente di ontologie e non abbiano nemmeno l'opportunità (tempo non disponibile) di imparare qualcosa di esse.

Ontology Expert è quella persona che è capace di usare gli strumenti di modellazione semantica e può descrivere la conoscenza contribuita dai Domain Expert con un Modello Ontologico. In questo modo l'informazione che era eterogenea, e qualche volta anche tacita o embedded, diventa formalizzata, esplicita, omogenea e consistente [59].

Stakeholder è quella persona che è interessata alla logica di dominio ma non è necessariamente esperta. In questo caso è utile avere un modello ontologico che può essere letto e studiato e che può essere usato per navigare attraverso i documenti dei Domain Expert.

Infine IT Expert è quella persona che deve sviluppare gli strumenti software necessari agli altri soggetti così da permettere loro di leggere e scrivere risorse e modelli ontologici.

4.3.3 Modello di Dominio

L'obiettivo principale di Carepedia è quello di impiegare l'informazione contenuta nel modello ontologico direttamente contribuito dagli utenti per descrivere i metadati relativi ai documenti forniti dagli utenti stessi, al fine di abilitarli a ricercare i documenti per mezzo di query semantiche e a navigare da un documento all'altro per mezzo di link derivanti dall'astrazione ontologica definita.

Il modello di dominio descrive le classi che fanno parte del modello ad oggetti costituente il Domain Layer dell'applicazione ed è formato da due parti, vedi Fig. 4.20: Carepedia Specific Concepts e User Defined Concepts and Realizations. Carepedia Specific Concepts contiene le classi del modello che corrispondono a quei concetti dell'ontologia che sono da considerarsi predefiniti ed immutabili per l'applicazione, come ad esempio: `Document`, `Permission` o `User`. User Defined Concepts and Realizations contiene invece quelle classi che servono a manipolare le porzioni di ontologia che sono direttamente ge-

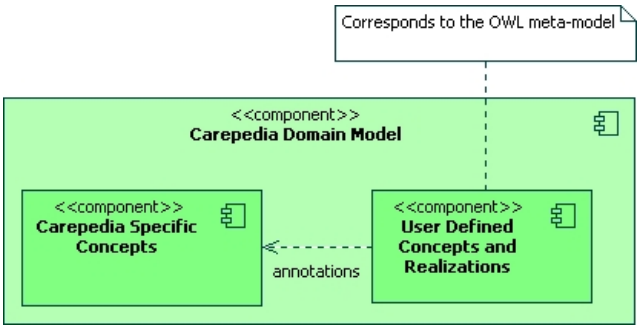

Figura 4.20. Carepedia

stibili dagli utenti. Questa porzione di modello corrisponde essenzialmente al meta-modello di OWL ed in linea di principio potrebbe essere adoperata per manipolare tutta l'informazione del Data Layer di Carepedia compresi i concetti e le realizzazioni contenute in Carepedia Specific Concepts. L'esistenza di due moduli è giustificata dal fatto che, col solo meta-modello di OWL, la logica applicativa finirebbe per essere organizzata secondo il paradigma procedurale anziché quello object oriented ed il codice del Domain Layer risulterebbe in molte parti più verboso e maggiormente orientato all'errore.

Da qui in poi, indicheremo col termine "annotazione" ogni riferimento fatto da entità definite dagli utenti verso concetti predefiniti in Carepedia. Di particolare interesse saranno le annotazioni relative ai documenti in quanto andranno a costituire l'insieme di metadati che abiliteranno gli utenti ad eseguire query semantiche per ricercare i documenti stessi.

La porzione di modello mostrata in Fig. 4.21 fa parte di Carepedia Specific Concepts e serve per la profilazione degli utenti e per la descrizione dei permessi associati a ciascuna risorsa. Tali funzionalità rivestono un'importanza strategica nella definizione di un software per la produzione di concettualizzazioni di tipo distribuito e collaborativo. In particolare `Resource`, che rappresenta una generica risorsa gestita da Carepedia, è associata ad un insieme di permessi (`Permission`), ognuno dei quali relativo ad una ben precisa azione applicabile alla risorsa stessa (`PermissionType`) e relativo ad un determinato contesto (`PermissionScope`). Ogni risorsa è collegata all'autore che l'ha creata (`User`) il quale può essere membro di uno o più gruppi (`Group`). Il meccanismo di

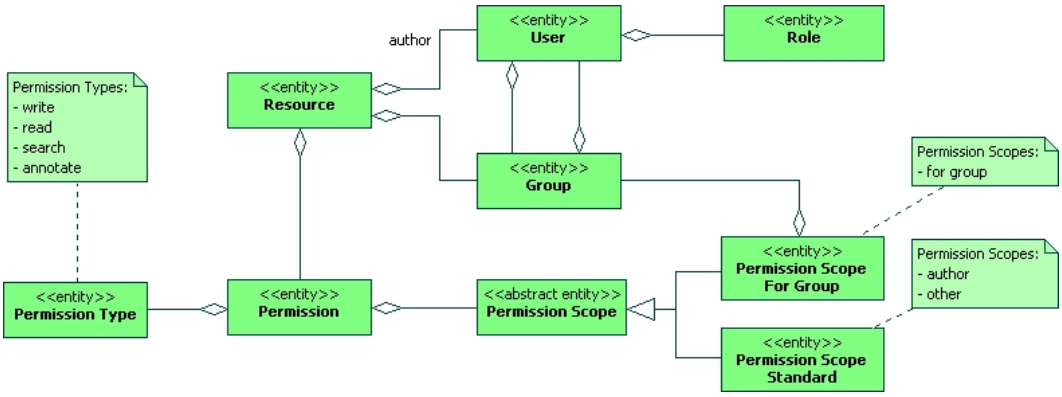

Figura 4.21. Modello relativo a risorse, utenti e permessi

Carepedia per la gestione dei permessi delle risorse, volutamente, per molti versi richiama le funzionalità che i sistemi operativi mettono a disposizione per regolare l'accesso ai file. In Carepedia ogni utente è inoltre associato ad uno o più ruoli (`Role`) che servono a specificare l'insieme di funzionalità per le quali è abilitato.

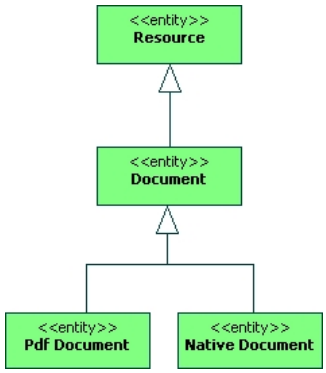

Figura 4.22. Modello dei documenti di Carepedia

In Fig. 4.22 viene mostrata la classe `Document` che serve a descrivere qualunque documento gestito per mezzo di Carepedia; allo stato attuale sono considerati tali i file PDF (`PDF Document`) ed i documenti che direttamente siano stati contribuiti dagli utenti per mezzo dell'interfaccia wysiwyg di Carepedia (`Native Document`). Il numero di tipologie di documenti supportate può comunque essere esteso in base alle esigenze (ad esempio file Microsoft

Word).

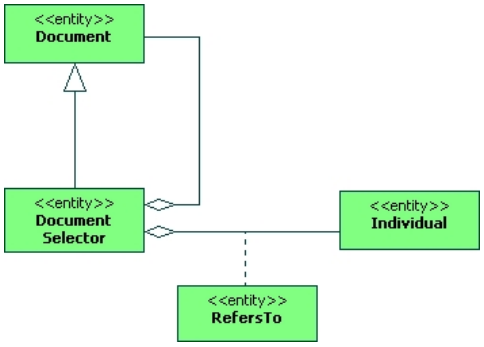

Figura 4.23. Modello per le annotazioni di Carepedia

In Fig. 4.23 viene mostrato il modello corrispondente al sottoinsieme di annotazioni considerato ad oggi in Carepedia. In particolare interessano i collegamenti tra documenti (o parti di essi) verso generici individui attraverso la prorietà `RefersTo` o una delle sue eventuali specializzazioni definite dagli utenti.

La necessità di limitare il numero di tipologie di annotazioni gestite deriva dalla volontà di ridurre l'ampiezza del problema, per arrivare più velocemente alla definizione di interfacce utente sufficientemente semplici ed usabili.

Carepedia è sviluppata seguendo una metodologia di tipo iterativo ed incrementale che mira a produrre una successione di rilasci ognuno dei quali, una volta sperimentato da una comunità di utenti, può essere utilizzato, attraverso feedback e commenti, per guidare l'analisi e l'implementazione del rilascio successivo.

Le Fig. 4.24, 4.25 e 4.26 descrivono la porzione di modello che fa parte del componente User Defined Concepts and Realizations e che corrisponde sostanzialmente al meta-modello di OWL permettendo agli utenti, in questa maniera, di definire nuove classi, predicati ed individui, indipendentemente dal dominio trattato e senza alcuna limitazione. Si osservi inoltre che, in Carepedia, `Class`, `Property` ed `Individual` sono specializzazioni di `Resource`, pertanto può essere loro applicato il meccanismo di gestione dei permessi precedentemente introdotto.

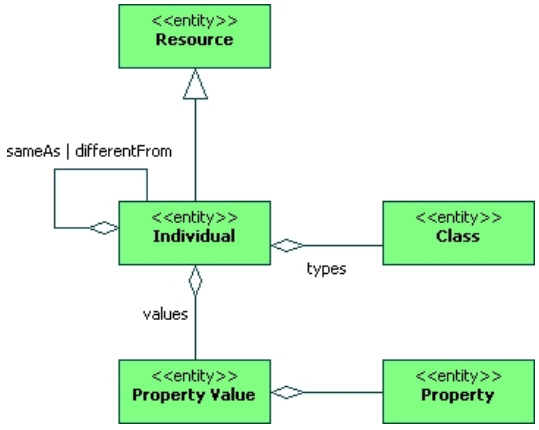

Figura 4.24. Modello per gli Individual

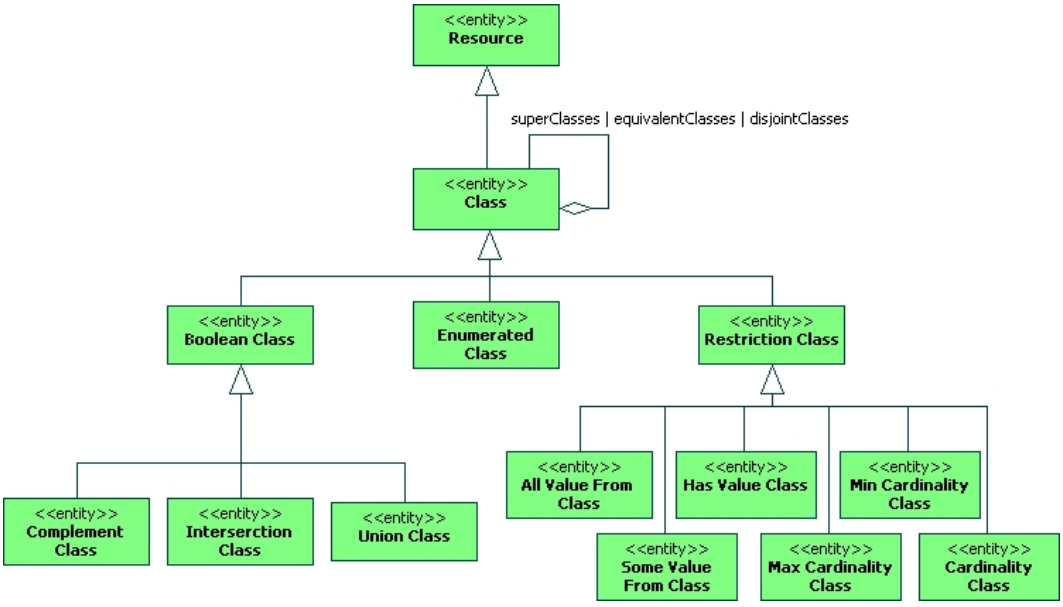

Figura 4.25. Modello delle classi ontologiche

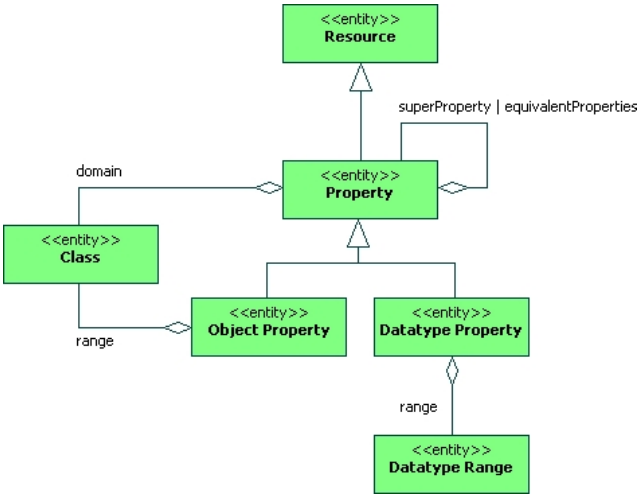

Figura 4.26. Modello delle Properties

4.3.4 Casi d'uso

In Fig. 4.27 viene mostrato il modello dei principali casi d'uso dell'attuale versione di Carepedia.

In particolare si può osservare che gli utenti Guest possono ricercare i documenti e visualizzare i risultati (dipendentemente dalle impostazioni dei permessi definite per i documenti stessi). La classe di utenti User è inoltre abilitata a contribuire nuova conoscenza sia nella forma di nuovi documenti ed annotazioni che in quella di nuove porzioni del modello ontologico.

Occorre rilevare che la difficoltà concettuale, intrinsecamente legata alla definizione di nuove classi e proprietà di un'ontologia, è maggiore rispetto a quella di definire nuove istanze, documenti od annotazioni. Si ritiene pertanto che solo un numero ridotto di utenti, pur avendone i permessi, sarà concretamente in grado di fornire nuova conoscenza nella forma di schemi ontologici. Gli utenti, infine, che abbiano i privilegi di Admin, sono abilitati a svolgere, oltre alle operazioni precedenti, anche quelle relative alla definizione di nuovi utenti o gruppi.

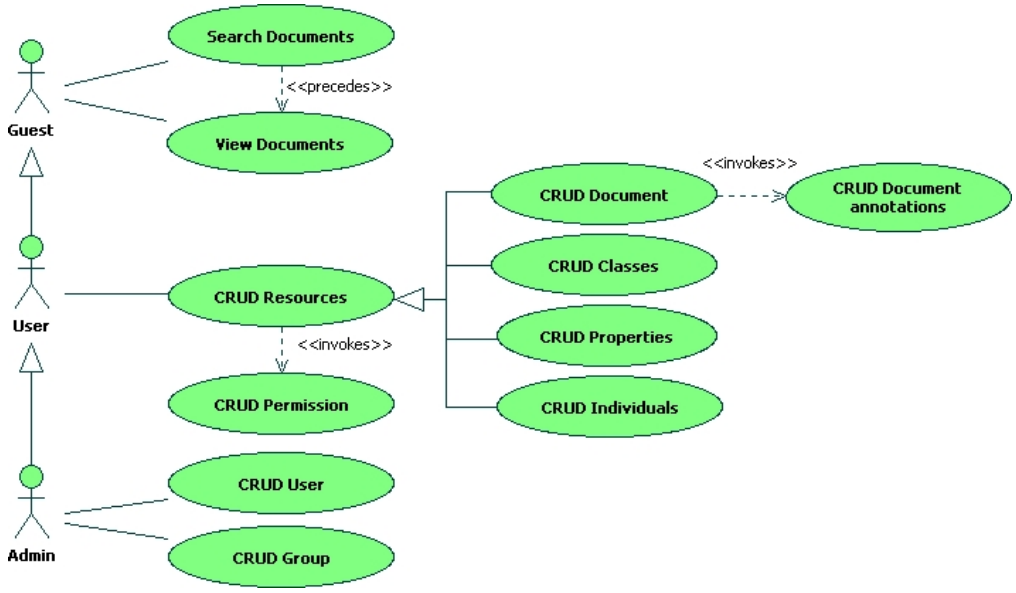

Figura 4.27. Casi d'uso principali di Carepedia

4.3.5 Sperimentazione

Verrà adesso presentata una sperimentazione eseguita con l'attuale versione di Carepedia riguardante il dominio applicativo delle Reti di Petri. Per prima cosa verrà presentato il modello relativo a tale argomento e successivamente, una volta definiti alcuni scenari d'uso, saranno mostrati i corrispondenti screenshot dell'applicazione.

Gli scenari presentati riguarderanno la contribuzione di nuova conoscenza e le ricerche semantiche, in quanto tali operazioni si immaginano essere le più caratteristiche per la versione attuale di Carepedia.

Occorre osservare che le classi, le proprietà e le istanze del modello delle Reti di Petri corrispondono a concettualizzazioni e realizzazioni definite dagli utenti di Carepedia e che pertanto vengono manipolate all'interno dell'applicazione attraverso le classi del modello di dominio facenti parte del componente User Defined Concepts and Realizations.

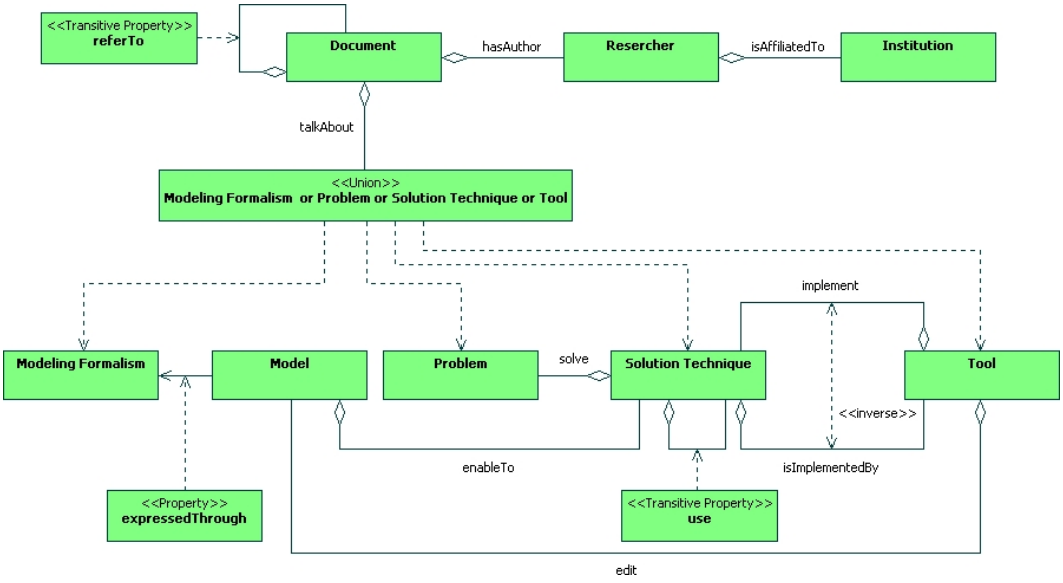

Figura 4.28. Modello delle Reti di Petri

4.3.5.1 Modello per le Reti di Petri

In Fig. 4.28 sono mostrate le classi principali, insieme alle loro reciproche relazioni, che servono a descrivere il dominio delle Reti di Petri. In particolare ogni modello (`Model`) è espresso attraverso un formalismo (`Modeling Formalism`) ed abilita ad un certo numero di tecniche di risoluzione (`Solution Technique`) le quali servono a risolvere una determinata classe di problemi (`Problem`). Ogni tecnica risolutiva può far riferimento ad altre e può essere implementata da uno o più software (`Tool`). Infine i documenti (`Document`), scritti da un ricercatore (`Researcher`) affiliato ad una ben determinata istituzione (`Istitution`), possono parlare di uno o più argomenti tra modelli, formalismi di modellazione, tecniche risolutive, problemi o tool, e possono riferirsi ad altri documenti.

In Fig. 4.29 viene mostrato che la classe `Stochastic Model` è un particolare tipo di modello che sottendendo un processo stocastico (`Stochastic Process`) può essere espresso solo attraverso una determinata categoria di formalismi (`Stochastic Modeling Formalism`).

In Fig. 4.30 sono mostrate alcune specializzazioni della classe `Modeling Formalism` insieme alle corrispondenti istanze.

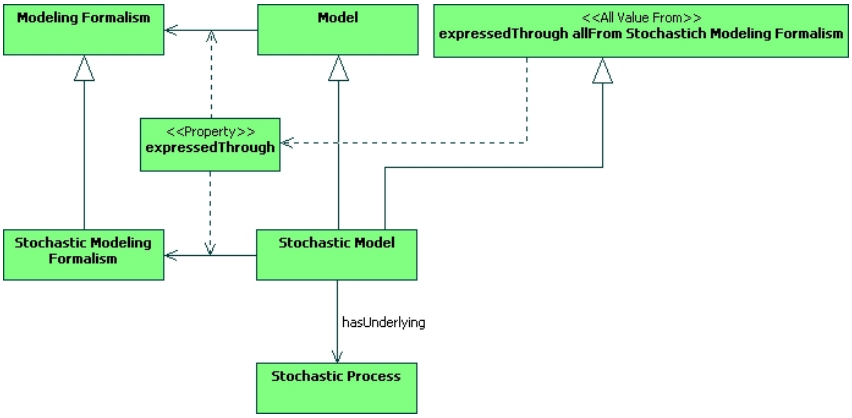

Figura 4.29. Modello dei processi stocastici

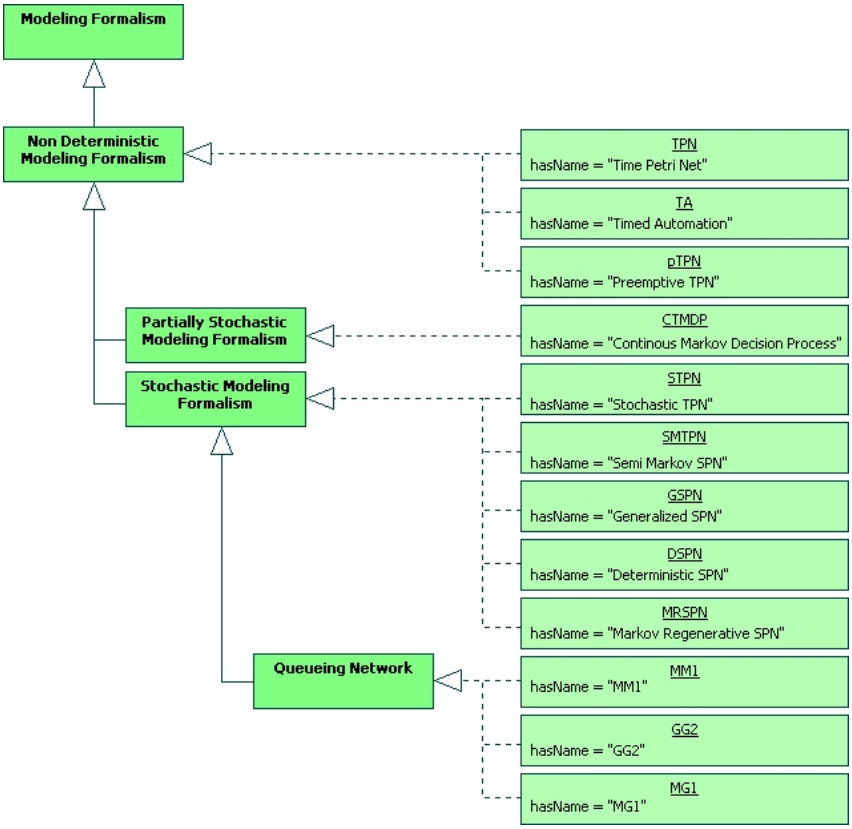

Figura 4.30. Dettagli relativi a formalismi di modellazione

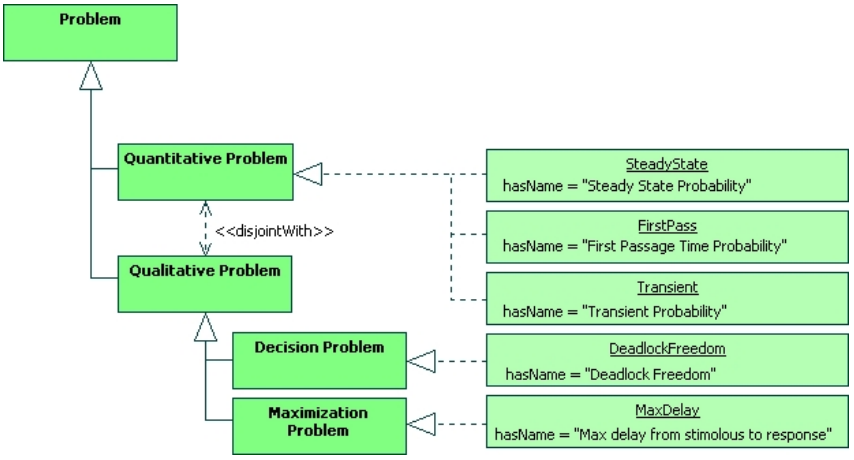

Figura 4.31. Dettaglio relativo ai problemi

In Fig. 4.31 sono mostrate alcune specializzazioni della classe `Problem` insieme alle corrispondenti istanze.

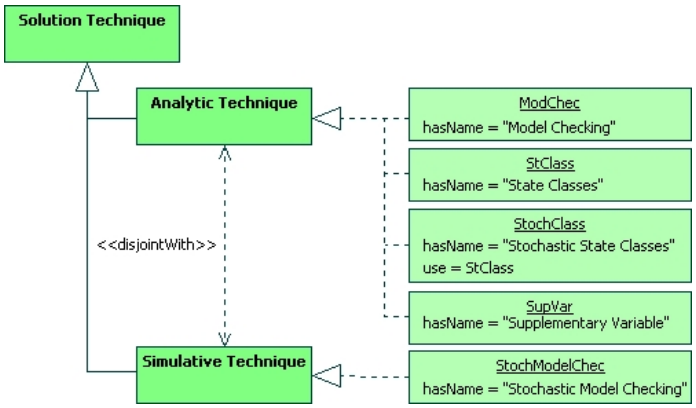

Figura 4.32. Dettaglio relativo alle tecniche risolutive

In Fig. 4.32 sono mostrate alcune specializzazioni della classe `Solution Technique` insieme alla corrispondenti istanze.

In Fig. 4.33 sono infine mostrate altre classi ed istanze che sono necessarie per sperimentare gli scenari d'uso che seguono.

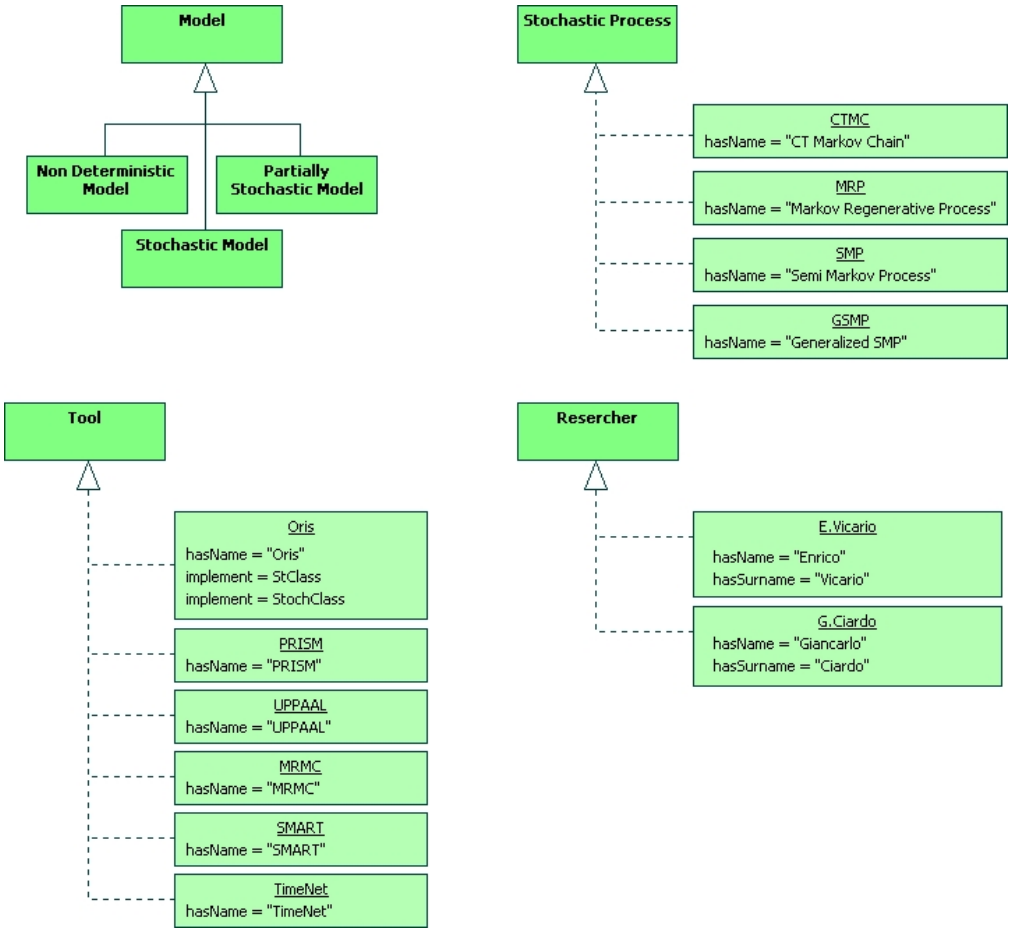

Figura 4.33. Classi e istanze necessarie alla simulazione

4.3.5.2 Simulazione d'uso

Scenario 1

L'utente definisce prima la classe Tool e poi la proprietà implement relativa a tale classe. Successivamente crea l'individuo Oris che è un Tool che implementa le Solution Technique: State Classes e Stochastic State Class (già presenti nell'ontologia). Vedere Fig. 4.34, 4.35, 4.36, 4.37, 4.38 e 4.39 per gli screenshot corrispondenti.

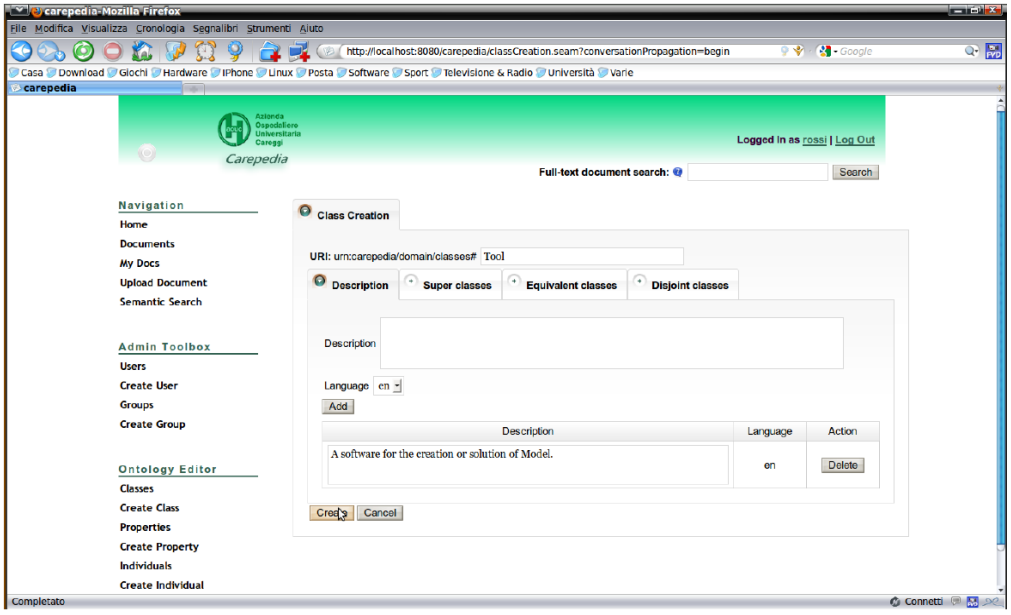

Figura 4.34. Creazione della classe `Tool`

Scenario 2

L'utente, che sta visualizzando un documento, decide di aggiungere una nuova annotazione (tag o metadata). In particolare si sposta sulla Tab Tag type dove definisce il nuovo tipo di annotazione `hasAuthor` (che supponiamo non esistere). Si osservi che l'utente sta in questo modo definendo una specializzazione della proprietà `RefersTo` introdotta in Fig. 4.23. L'utente si sposta quindi sulla Tab Tag dove può specificare che il documento corrente ha un autore che

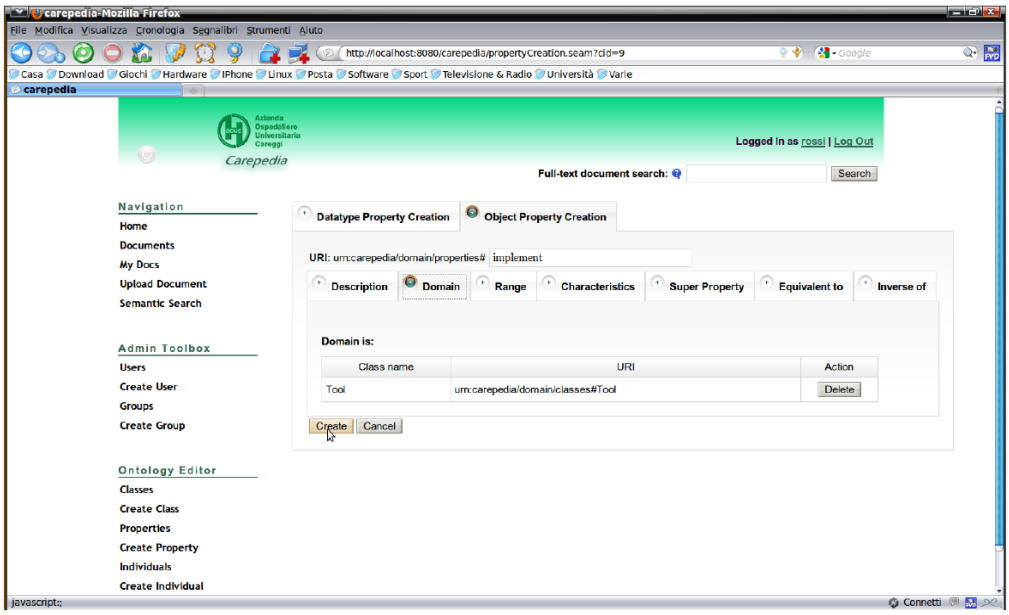

Figura 4.35. Creazione della proprità `implement` (Tab Domain)

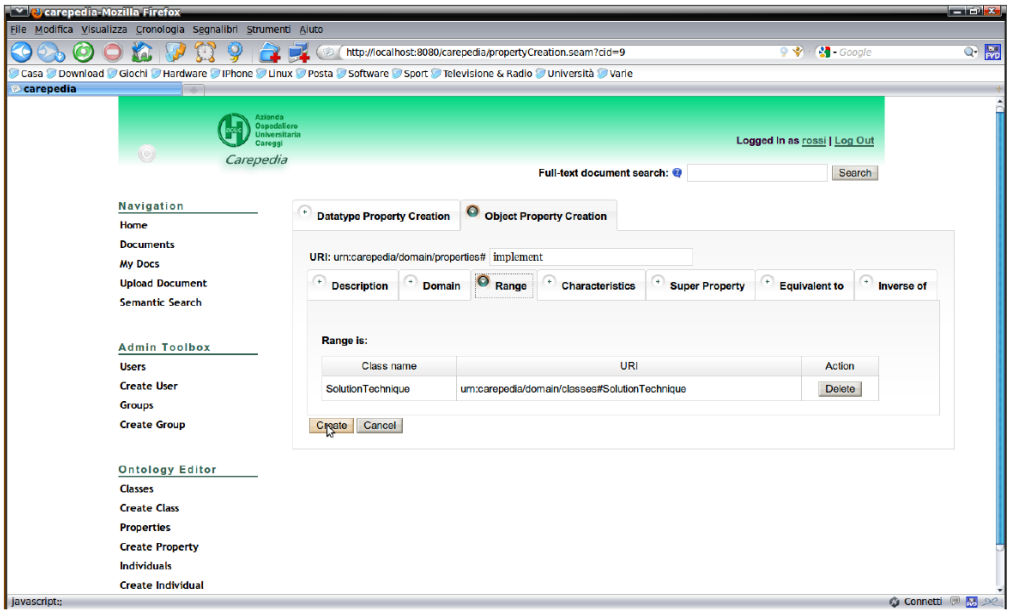

Figura 4.36. Creazione della proprità `implement` (Tab Raange)

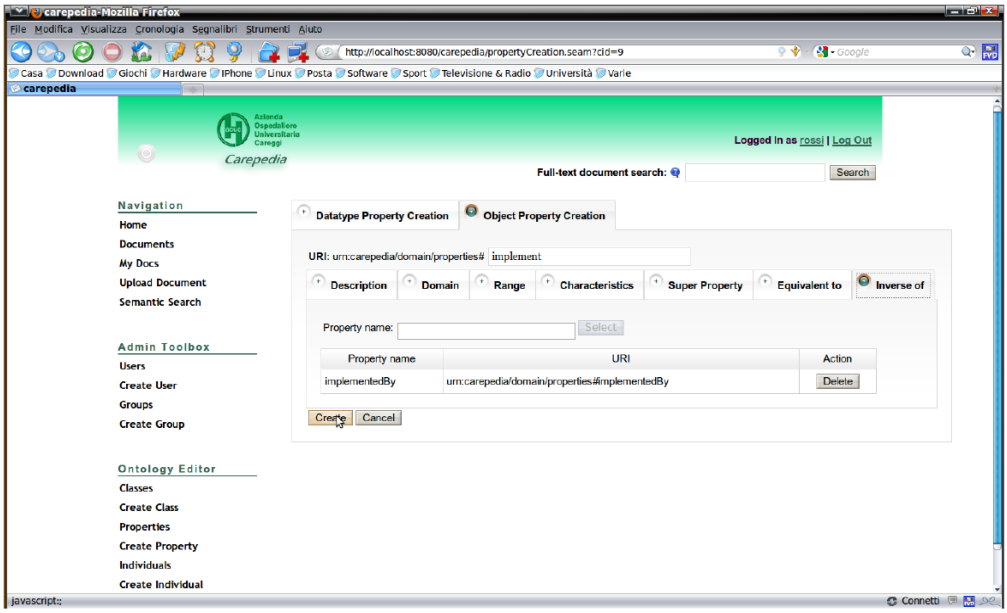

Figura 4.37. Creazione della proprità `implement` (Tab InverseOf)

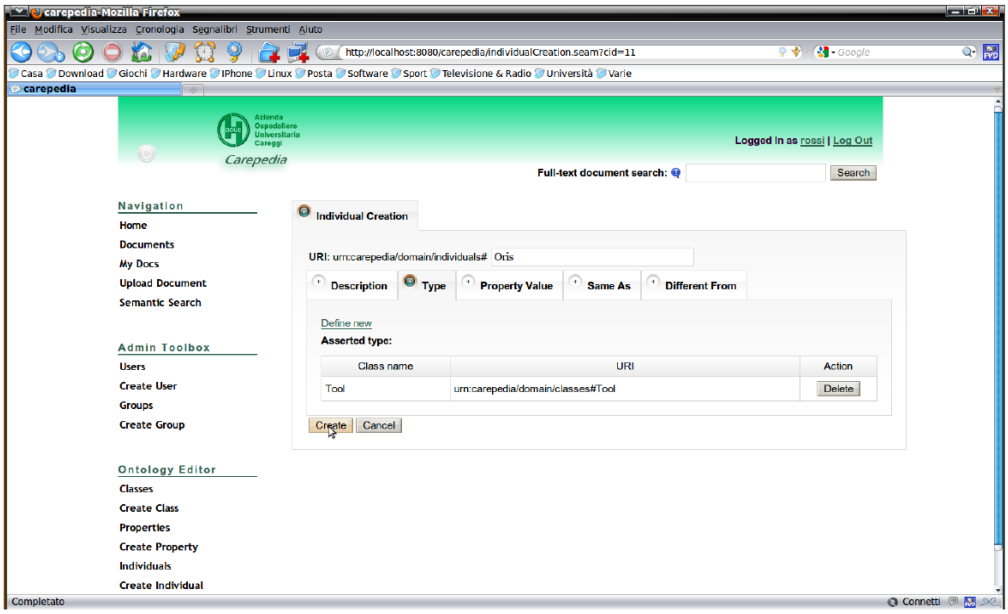

Figura 4.38. Creazione dell'individuo `Oris` (Tab Type)

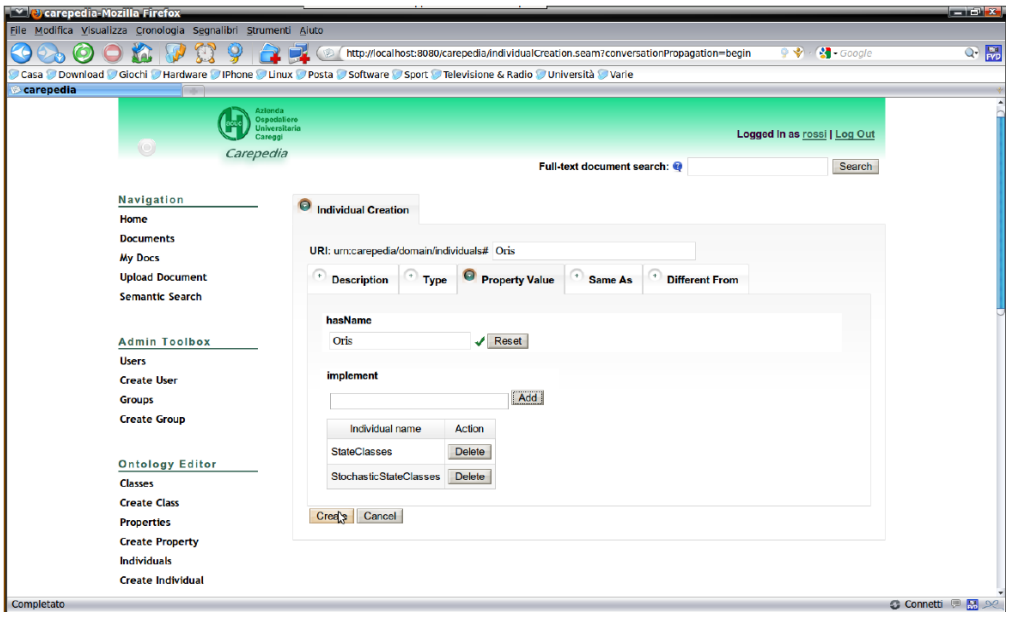

Figura 4.39. Creazione dell'individuao `Oris` (Tab Property Value)

è Enrico Vicario (che supponiamo essere già presente nell'ontologia). Vedere Fig. 4.40, 4.41 per gli screenshot corrispondenti.

Scenario 3

L'utente decide di cercare: Tutti i documenti (`Document`) che parlano di (`talkAbout`) `Tool` che implementano (`implement`) una tecnica risolutiva (`SolutionTechnique`) il cui nome (`hasName`) contiene la stringa "State Classes". Tale query è formata dalla congiunzione di un insieme di clausole ed è esprimibile attraverso l'interfaccia semplificata per l'immissione di query semantiche. L'utente seleziona la voce di menu "Semantic Search", compila i campi della pagina e preme il pulsante "Execute", vedere Fig. 4.42.

La query definita per mezzo della pagina di costruzione guidata corrisponde in realtà all'espressione SPARQL mostrata nel listato 4.1.

Scenario 4

L'utente decide di cercare: Tutti i documenti (`Document`) che parlano di (`talkAbout`) una tecnica risolutiva (`SolutionTechnique`) applicabile (`amenableTo`) a modelli stocastici (`StochasticModel`) che sottendono (`hasUnderlying`) un

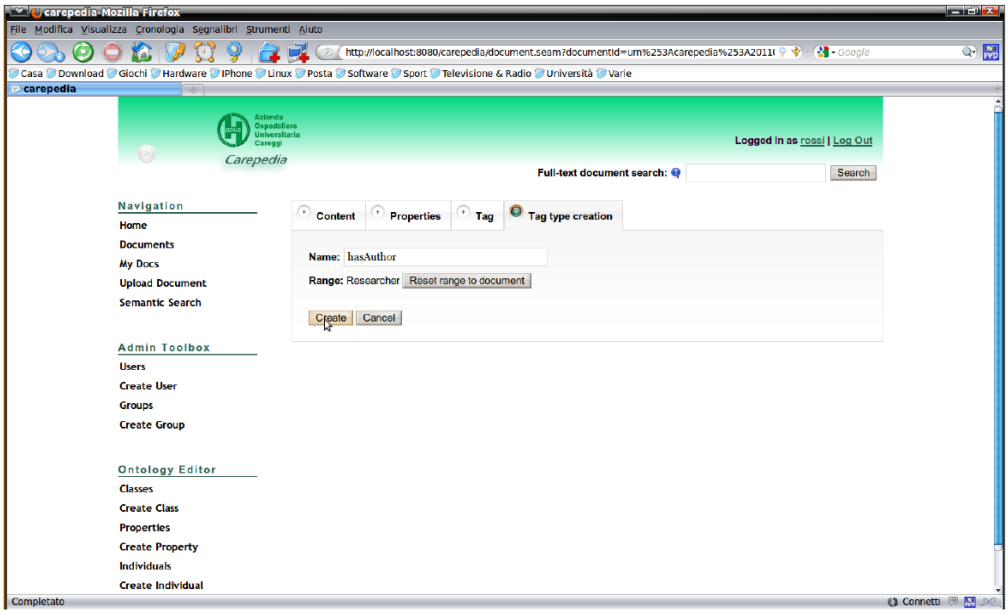

Figura 4.40. Creazione del nuovo tipo di tag `hasAuthor`

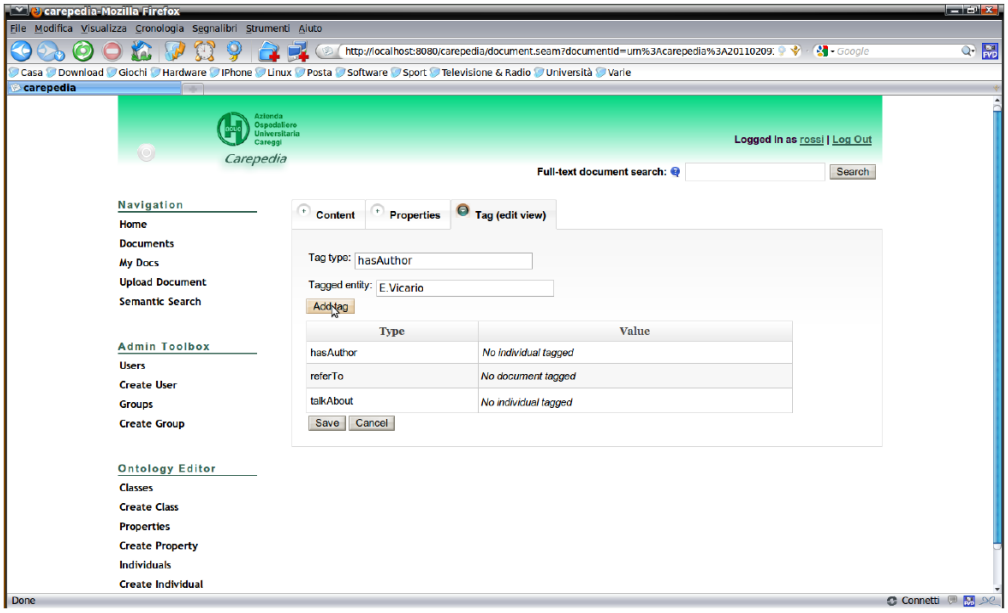

Figura 4.41. Creazione della annotazione relativa a `hasAuthor`

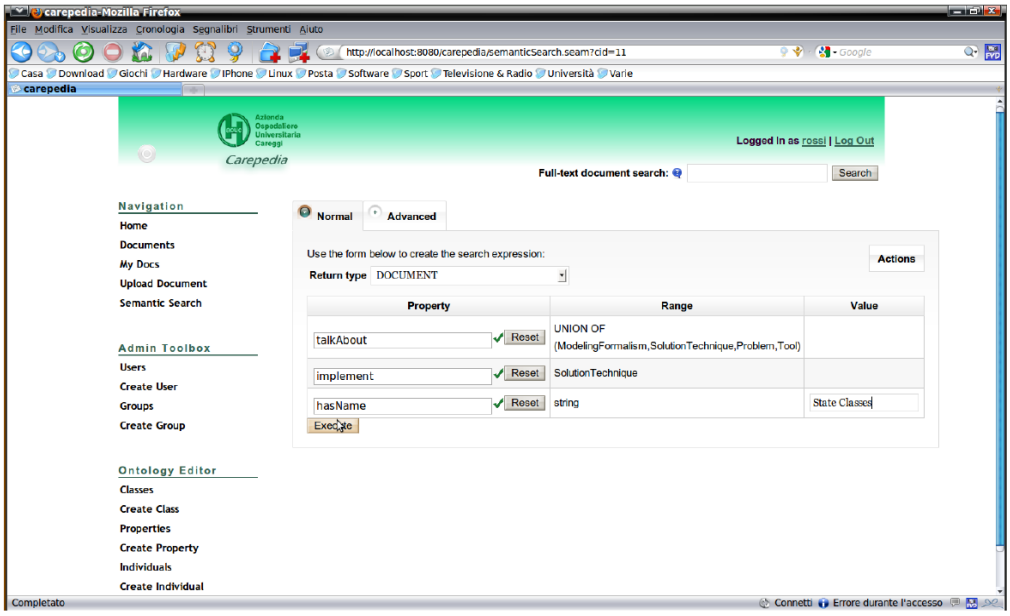

Figura 4.42. Maschera per la definizione di query semantiche

processo stocastico (`StochasticProcess`) non Markoviano (dove per non Markoviano si intende un processo stocastico diverso da CTMC). Nella versione attuale di Carepedia tale interrogazione non può essere formulata attraverso l'interfaccia di immissione guidata di query semantiche. L'utente dovrà quindi utilizzare il pannello di ricerca avanzata dove potrà specificare direttamente l'espressione SPARQL, vedere Fig. 4.43.

Listing 4.1. SPARQL corrispondente allo Scenario3

```
PREFIX rdf: <http://www.w3c.org/1999/02/22-rdf-syntax-ns#>
PREFIX cp: <urn:carepedia#>
SELECT ?subject
WHERE {
  ?subject rdf:type cp:Document.
  ?subject cp:talkAbout ?x1.
  ?x1 cp:implement ?x2.
  ?x2 cp:hasName ?value.
  FILTER regex( str( ?value ), "State Classes", "i" ).
}
```

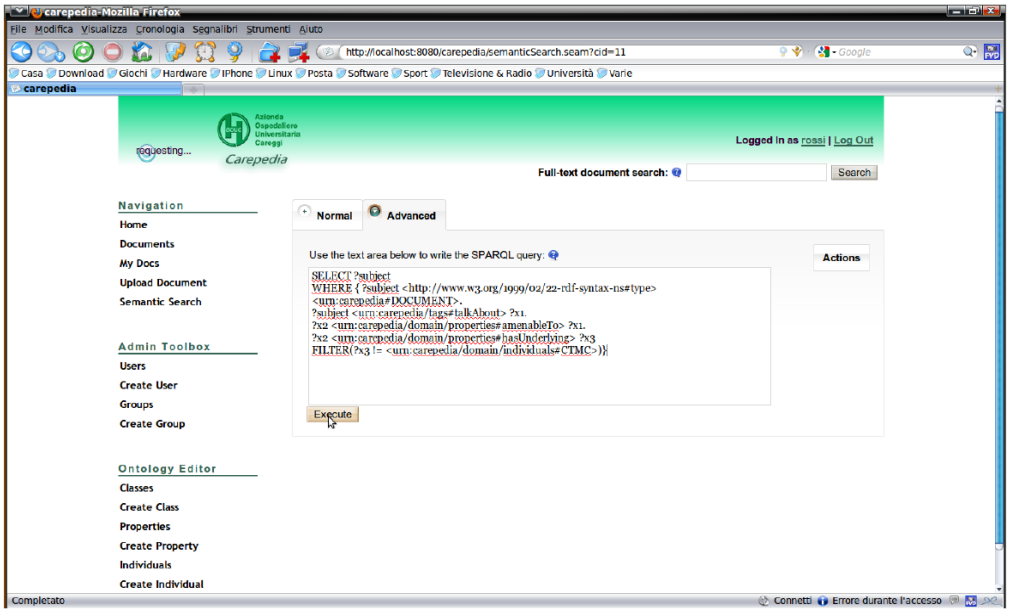

Figura 4.43. Carepedia

Conclusioni

In questo lavoro abbiamo proposto un'architettura ontologica che si basa sul componente software di object-ontology mapping (Loom) da noi realizzato.

Loom risolve il problema di impedance mismatch esistente tra modelli ad oggetti e modelli ontologici rendendo agevole lo sviluppo di un layer di mapping nelle applicazioni. Impiegando Loom diventa possibile interrogare e manipolare tutte le entità dei modelli OWL attraverso modelli Java opportunamente mappati. Il potente linguaggio di mapping consente di mettere in relazione classi, proprietà ed istanze ontologiche con corrispondenti classi Java abilitando, tra l'altro, l'implementazione del pattern Reflection. Loom rende il codice dell'applicazione indipendente dal particolare repository ontologico impiegato che può essere modificato cambiando le impostazioni di un file di configurazione. Attraverso un complesso meccanismo di proxy viene inoltre ottimizzato l'accesso al repository ontologico sia per quel che riguarda le operazioni di lettura che per quelle di scrittura, in modo da rendere più efficiente l'impiego delle ontologie da parte dell'applicazione. Nel futuro si immagina di estendere Loom per supportare nuove tipologie di repository ontologici, per abilitare l'esecuzione delle query per mezzo di linguaggi diversi da SPARQL (ad esempio JPQL) e per migliorare ulteriormente le prestazioni nell'accesso ai repository.

L'esperienza maturata con lo studio della letteratura e con lo svolgimento di numerose sperimentazioni ha permesso di valutare l'effettivo potenziale delle ontologie per la realizzazione di complesse architetture software evidenziando

la loro adeguatezza nel costruire sistemi con alto grado di interoperabilità manutenibilità ed evolvibilità.

La realizzazione di prototipi funzionanti ha messo in evidenza anche alcune criticità. In particolare è emersa la scarsa maturità dei repository ontologici che ad oggi non riescono a competere quanto a prestazioni e stabilità con le più tradizionali piattaforme per i database relazionali. È del tutto ragionevole aspettarsi che tali problemi verranno superati in un futuro prossimo dalla normale evoluzione tecnologica e per il momento possono essere mitigati attraverso l'impiego di uno strumento di object-ontology mapping quale quello realizzato.

Bibliografia

[1] Aduna. Sesame: Rdf schema querying and storage . `http://www.openrdf.org/`, accessed 2010.

[2] Ioannis N. Athanasiadis, Ferdinando Villa, and Andrea-Emilio Rizzoli. Ontologies, javabeans and relational databases for enabling semantic programming. In *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference*, pages 341–346, Washington, DC, USA, 2007. IEEE Computer Society.

[3] bblfish. Binding java objects to rdf . `http://java.net/projects/sommer`, accessed 2010.

[4] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Professional, November 2002.

[5] Tim Berners-Lee. Semantic web roadmap . `http://www.w3.org/DesignIssues/Semantic.html`, accessed 2010.

[6] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley Professional, September 1998.

[7] E. Bozsak, Marc Ehrig, Siegfried Handschuh, Andreas Hotho, Alexander Maedche, Boris Motik, Daniel Oberle, Christoph Schmitz, Steffen Staab, Ljiljana Stojanovic, Nenad Stojanovic, Rudi Studer, Gerd Stumme, York Sure, Julien Tane, Raphael Volz, and Valentin Zacharias. Kaon tool suite . `http://kaon.semanticweb.org/frontpage`, accessed 2010.

[8] Sara Brockmans, Raphael Volz, Andreas Eberhart, and Peter Löffler. Visual modeling of OWL DL ontologies using UML. In *International Semantic Web Conference*, pages 198–213, 2004.

[9] Giacomo Bucci, Marco Rufino, Valeriano Sandrucci, and Enrico Vicario. An ontological sw architecture supporting annotation and retrieval of digital resources. In *Eva09*, 2009.

[10] Giacomo Bucci, Valeriano Sandrucci, and Enrico Vicario. An incremental approach to software reengineering bases on object-data mapping. In *ICSoft08*, 2008.

[11] Giacomo Bucci, Valeriano Sandrucci, and Enrico Vicario. An ontological sw architecture supporting agile development of semantic portals. *Book Series Communications in Computer and Information Science*, 22:185–200, November 2008.

[12] Giacomo Bucci, Valeriano Sandrucci, and Enrico Vicario. Ontologies and bayesian networks in medical diagnosis. In *HICSS44*, 2010.

[13] Giacomo Bucci, Valeriano Sandrucci, and Enrico Vicario. Ontology-driven enterprise application integration. In *SEKE*, 2010.

[14] Giacomo Bucci, Valeriano Sandrucci, Enrico Vicario, and Saverio Mecca. An ontological sw architecture for the development of cooperative web portals. In *ICSoft07*, 2007.

[15] Christoph Bussler. The role of semantic web technology in enterprise application integration. *The Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, pages 62–68, 2003.

[16] Alistair Cockburn. The interaction of social issues and software architecture. *Commun. ACM*, 39(10):40–46, 1996.

[17] Hewlett-Packard Development Company. Jena a semantic web framework for java
. `http://jena.sourceforge.net/`, accessed 2010.

[18] Hewlett-Packard Development Company. Jenabean - a library for persisting java beans to rdf
. `http://code.google.com/p/jenabean/`, accessed 2010.

[19] Oscar Corcho, Angel López-Cima, and Asunción Gomez-Pérez. A platform for the development of semantic web portals. In *ICWE '06: Proceedings of the 6th international conference on Web engineering*, pages 145–152, New York, NY, USA, 2006. ACM Press.

[20] Taylor Cowan. Binding java objects to rdf . `http://semanticweb.com/binding-java-objects-to-rdf_b10682`, accessed 2010.

[21] Paulo Cesar G. da Costa and Kathryn B. Laskey. Multi-entity bayesian networks without multi-tears. `http://hdl.handle.net/1920/456`, 2006.

[22] Paulo Cesar G. da Costa, Kathryn B. Laskey, and Kenneth J. Laskey. Pr-owl: A bayesian ontology language for the semantic web. In *ISWC-URSW*, pages 23–33, 2005.

[23] Jiangbo Dang, Amir Hedayati, Ken Hampel, and Candemir Toklu. An ontological knowledge framework for adaptive medical workflow. *J. of Biomedical Informatics*, 41(5):829–836, 2008.

[24] Mary Shaw David Garlan. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering*, pages 1–39. Publishing Company, 1993.

[25] Vladan Devedzić. Understanding ontological engineering. *Commun. ACM*, 45(4):136–144, 2002.

[26] Ann Devitt, Boris Danev, and Katarina Matusikova. Constructing bayesian networks automatically using ontologies. *Applied Ontology, IOS Press*, 1:1–11, 2006.

[27] Zhongli Ding and Yun Peng. A probabilistic extension to ontology language owl. In *HICSS*, 2004.

[28] Zhongli Ding, Yun Peng, and Rong Pan. A bayesian network approach to ontology mapping. In *International Semantic Web Conference*, pages 563–577, 2005.

[29] M. Domingo. Managing healthcare through social networks. *Computer*, 43(7):20–25, July 2010.

[30] Matt Stephens Doug Rosenberg. *Use Case Driven Object Modeling with UML*. Apress, 2007.

[31] Marek J. Druzdzel and Linda C. van der Gaag. Building probabilistic networks: where do the numbers come from? guest editors' introduction. *IEEE Trans. Knowl. Data Eng.*, 12(4):481–486, 2000.

[32] Nick Rozanski Eoin Woods. *Software Systems Architecture*. Addison-Wesley Professional, 2005.

[33] Stefan Fenz, A Min Tjoa, and Marcus Hudec. Ontology-based generation of bayesian networks. In *International Conference on Complex, intelligent and Software Intensive Systems*, 2009.

[34] Apache Software Foundation. Servicemix . `http://servicemix.apache.org/home.html`, accessed 2010.

[35] Martin Fowler. *Analysis Patterns, Reusable Object Models*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[36] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[37] Paolo Paolini Franca Garzotto, Luca Mainetti. Hypermedia design analysis and evaluation issues. incomm. of the acm. Communications of the ACM, 1995.

[38] Franz. Racer pro . `http://www.franz.com/agraph/racer/`, accessed 2010.

[39] Johnson Ralph Vlissides John Gamma Erich, Helm Richard. *Design Patterns*. Addison-Wesley Professional, January 1995.

[40] Neil M. Goldman. Ontology-oriented programming: Static typing for the inconsistent programmer. In *2nd International Semantic Web Conference (ISWC 2003)*, pages 850–865, Sanibel Island, FL, 2003.

[41] T. R. Gruber. A translation approach to portable ontologies. *Knowledge Acquisition*, 5(2):199–220, 1993.

[42] Edited N. Guarino, R. Poli, and Nicola Guarino. Formal ontology, conceptual analysis and knowledge representation. *International Journal of Human and Computer Studies*, 43:625–640, 1995.

[43] Abdolreza Hajmoosaei and Sameem Abdul-Kareem. An approach for mapping of domain-based local ontologies. In *CISIS '08: Proceedings of the 2008 International Conference on Complex, Intelligent and Software*

Intensive Systems, pages 865–870, Washington, DC, USA, 2008. IEEE Computer Society.

[44] Hans J. Happel and Stefan Seedorf. Applications of ontologies in software engineering. In *Workshop on Sematic Web Enabled Software Engineering (SWESE) on the 5th International Semantic Web Conference (ISWC 2006)*, Athens, Georgia - USA, November 2006.

[45] Wilhelm Hasselbring. Information system integration. *Commun. ACM*, 43(6):32–38, 2000.

[46] Eveline M. Helsper and Linda C. van der Gaag. Building bayesian networks through ontologies. In *Proceedings of the 15th Eureopean Conference on Artificial Intelligence, ECAI'2002*, pages 680–684, Lyon, France, July 2002.

[47] Jim Heumann. Generating test cases from use cases. The Rational Edge, 2001.

[48] Gregor Hohpe and Bobby Woolf. *Entreprise Integration Patterns*. Addison Wensley, 2008.

[49] ISO. Iso 9126 "Software engineering - Product quality" . `http://www.iso.org/iso/en/ISOOnline.frontpage`, accessed 2010.

[50] ISO. Iso 9241-11 "Guidance on Usability" . `http://www.iso.org/iso/en/ISOOnline.frontpage`, accessed 2010.

[51] Saïd Izza, Lucien Vincent, and Patrick Burlat. Ontology urbanization for semantic integration: Dealing with semantics within large and dynamic enterprises. In *EDOC '05: Proceedings of the Ninth IEEE International EDOC Enterprise Computing Conference*, pages 83–94, Washington, DC, USA, September 2005. IEEE Computer Society.

[52] Saïd Izza, Lucien Vincent, and Patrick Burlat. A unified framework for application integration - an ontology-driven service-oriented approach. In *ICEIS (1)*, pages 165–170, Miami, Florida - USA, 2005.

[53] Yuhui Jin, Stefan Decker, and Gio Wiederhold. Ontowebber: Model-driven ontology-based web site management. In *In Semantic Web Working Symposium (SWWS*, 2001.

[54] Aditya Kalyanpur, Daniel J. Pastor, Steve Battle, and Julian Padget. Automatic mapping of owl ontologies into java. In *16th Int'l Conference on Software Engineering and Knowledge Engineering (SEKE 2004)*, Banff, Canada, June 2004.

[55] Khalil Khoumbati, Marinos Themistocleous, and Zahir Irani. Integration technology adoption in healthcare organisations: A case for enterprise application integration. In *HICSS '05: Proceedings of the Proceedings of the 38th Annual Hawaii International Conference on System Sciences (HICSS'05) - Track 6*, page 149.1, Washington, DC, USA, 2005. IEEE Computer Society.

[56] Holger Knublauch. An ai tool for the real world: Knowledge modeling with protégé. *JavaWorld, June 20, 2003.*, June 2003.

[57] Holger Knublauch. Ontology-driven software development in the context of the semantic web: An example scenario with protégé/owl. In David S. Frankel, Elisa F. Kendall, and Deborah L. Mcguinness, editors, *1st International Workshop on the Model-Driven Semantic Web (MDSW2004)*, 2004.

[58] Holger Knublauch, Matthew Horridge, Mark Musen, Alan Rector, Robert Stevens, Nick Drummond, Phil Lord, Natalya F. Noy, Julian Seidenberg, and Hai Wang. The protégé owl experience. In *Workshop on OWL: Experiences and Directions*, Galway, Ireland, 2005.

[59] Victor V. Kryssanov, V. A. Abramov, Yoshiro Fukuda, and K. Konishi. The meaning of manufacturing know -how. In *PROLAMAT '98: Proceedings of the Tenth International IFIP WG5.2/WG5.3 Conference on Globalization of Manufacturing in the Digital Communications Era of the 21st Century*, pages 375–388, Deventer, The Netherlands, The Netherlands, 1998. Kluwer, B.V.

[60] Rick Kazman Len Bass, Paul Clements. *Software Architecture in Practice*. Addison-Wesley Professional, 1997.

[61] Lifecom. Lifecom, bridging intelligence to the medical record . `http://www.lifecomhealth.com/index.html/`, accessed 2010.

[62] Boris Lublinsky. Achieving the ultimate EAI implementation. *eAI Journal*, February 2001.

[63] M.P. Cline M. Fayad. Aspects of software adaptability. COMMUNICATIONS OF THE ACM, 1996.

[64] N. Juristo M. Ferndndez, A. Gomez-Perez. Methontology: From ontological art towards ontological engineering. In *Proceedings of the AAAI97 Spring Symposium*, pages 33–40, 1997.

[65] S. de Deugd R. Yates M. Weitzel, A. Smith. A web 2.0 model for patient-centered health informatics applications. *Computer*, 43(7):43–50, 2010.

[66] Mark W. Maier, David Emery, and Rich Hilliard. ANSI/IEEE 1471 and systems engineering. *Systems Engineering*, 7(3):257–270, 2004.

[67] MedTech. Diagnosispro, fre online differential diagnosis tool . `http://en.diagnosispro.com/`, accessed 2010.

[68] Brenda M. Michelson. *Event-Driven Architecture Overview*. Patricia Seybold Group, Boston, Ma, USA, 2006. `http://dx.doi.org/10.1571/bda2-2-06cc`.

[69] Clark Mindswap Lab and Parsia. Pellet: The open source owl dl reasoner. http://pellet.owldl.com/, 2007.

[70] Thorsten Möller and Heiko Schuldt. A platform to support decentralized and dynamically distributed p2p composite owl-s service execution. In *MW4SOC '07: Proceedings of the 2nd workshop on Middleware for service oriented computing*, pages 24–29, New York, NY, USA, 2007. ACM.

[71] MuleSoft. Mule enterprise service bus. `http://www.mulesoft.org/`, accessed 2010.

[72] Mulgara. Mulgara semantic store. `http://www.mulgara.org/`, accessed 2010.

[73] Michele Nucci, Michele Barbera, Christian Morbidoni, and Daniel Hahn. A semantic web powered distributed digital library system. In Leslie Chan and Susanna Mornatti, editors, *ELPUB*, pages 130–139, 2008.

[74] OBO. The open biomedical and biological ontologies . `http://www.obofoundry.org/`, accessed 2010.

[75] Oracle. OpenESB. `http://wiki.open-esb.java.net/`, accessed 2010.

[76] S. Horton P.J. Lynch. Web style guide: Basic design principles for creating web sites. Yale University Press, 2002.

[77] Jeffrey T. Pollock. Integration's dirty little secret: It's a matter of semantics. Whitepaper, Modulant, the Interoperability Company, February 2002.

[78] Thomas Puschmann and Reiner Alt. Enterprise Application Integration - the case of the Robert Bosch Group. In *HICSS '01: Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)-Volume 9*, Washington, DC, USA, 2001. IEEE Computer Society.

[79] Tijs Rademakers and Jos Dirksen. *Open-Source ESBs in Action: Example Implementations in Mule and ServiceMix*. Manning Publication Co., September 2008.

[80] Domenico Redavid, Luigi Iannone, and Terry R. Payne. OWL-S Atomic Services Composition with SWRL Rules. In Giovanni Semeraro, Eugenio Di Sciascio, Christian Morbidoni, and Heiko Stoermer, editors, *SWAP*, volume 314 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007.

[81] Alejandro Rodriguez, Myriam Mencke, Giner Alor Hernandez, Ruben Posada-Gomez, and Juan Miguel Gomez. Medboli: Medical diagnosis based on ontologies and logical inference. In IEEE, editor, *International Conference on eHealth, Telemedicine, and Social Medicine - eTELEMED 2009*, Cancun, Mexico, February 2009.

[82] Stuart Russel and Peter Norvig. *Artificial Inteligence: A Modern Approach*. Prentice Hall, 1995.

[83] B. Sarder and S. Ferreira. Developing systems engineering ontologies. *System of Systems Engineering, 2007. SoSE '07. IEEE International Conference on*, 1:1–6, April 2007.

[84] Douglas C. Schmidt, Hans Rohnert, Michael Stal, and Dieter Schultz. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, Inc., New York, NY, USA, 2000.

[85] Kay-Uwe Schmidt, Jörg Dörflinger, Tirdad Rahmani, Mehdi Sahbi, Ljiljana Stojanovic, and Susan Marie Thomas. An user interface adaptation

architecture for rich internet applications. In Sean Bechhofer, Manfred Hauswirth, Jörg Hoffmann, and Manolis Koubarakis, editors, *ESWC*, volume 5021 of *Lecture Notes in Computer Science*, pages 736–750. Springer, 2008.

[86] Guus Schreiber, Alia Amin, Lora Aroyo, Mark van Assem, Victor de Boer, Lynda Hardman, Michiel Hildebrand, Borys Omelayenko, Jacco van Osenbruggen, Anna Tordai, Jan Wielemaker, and Bob Wielinga. Semantic annotation and search of cultural-heritage collections: The multimedian e-culture demonstrator. *Web Semant.*, 6(4):243–249, 2008.

[87] Guus Schreiber, Alia Amin, Mark van Assem, Viktor de Boer, Lynda Hardman, Michiel Hildebrand, Laura Hollink, Zhisheng Huang, Janneke van Kersen, Marco de Niet, Borys Omelayenko, Jacco van Ossenbruggen, Ronny Siebes, Jos Taekema, Jan Wielemaker, and Bob J. Wielinga. Multimedian e-culture demonstrator. In *International Semantic Web Conference*, pages 951–958, 2006.

[88] Rafael Silveira and Joan A. Pastor. A model for enterprise application integration tools evaluation. In *European and Mediterranean Conference on Information Systems (EMCIS) 2006*, Costa Blanca, Alicante, Spain, July 2006.

[89] Woodfield S.N. The impedance mismatch between conceptual models and implementation environments. In *Proceedings of the ER'97 Workshop on Behavioral Models and Design Transformations: Issues and Opportunities in Conceptual Modeling*, Los Angeles, California, November 1997. UCLA.

[90] Guanglei Song, Yu Qian, Ying Liu, and Kang Zhang. Oasis: a mapping and integration framework for biomedical ontologies. In *Proceedings of the 19th IEEE Symposium on Computer-Based Medical Systems (CBMS'06)*, 2006.

[91] Stanford-University. Protégé. `http://protege.stanford.edu`, accessed 2010.

[92] N. Stojanovic, A. Maedche, S. Staab, R. Studer, and Y. Sure. Seal: a framework for developing semantic portals, 2001.

[93] Michael Stonebraker. Integrating islands of information. *eAI Journal*, 1999.

[94] Suzette Stoutenburg, Leo Obrst, Deborah Nichols, Paul Franklin, Ken Samuel, and Michael Prausa. Ontologies in owl for rapid enterprise integration. In *OWLED*, 2007.

[95] Hugh Taylor, Angela Yochem, Les Phillips, and Frank Martinez. *Event-Driven Architecture: How SOA Enables the Real-Time Enterprise*. Addison Wesley, 2009.

[96] Christoph Tempich, H. Sofia Pinto, and Steffen Staab. Ontology engineering revisited: an iterative case study with diligent. In York Sure and John Domingue, editors, *The Semantic Web: Research and Applications: 3rd European Semantic Web Conference, ESWC 2006, June 11-14, 2006 Proceedings*, volume 4011 of *LNCS*, pages 110–124, Budva, Montenegro, JUN 2006. Springer.

[97] Christoph Tempich, H. Sofia Pinto, York Sure, and Steffen Staab. An argumentation ontology for distributed, loosely-controlled and evolving engineering processes of ontologies (diligent). In *Second European Semantic Web Conference, (ESWC 2005)*, volume 3532 of *LNCS*, pages 241–256, Heraklion, Crete, Greece, MAY 2005. Springer.

[98] Phil Tetlow, Jeff Z. Pan, Daniel Oberle, Evan Wallace, Michael Uschold, and Elisa Kendall. Ontology driven architectures and potential uses of the semantic web in systems and software engineering. Technical report, W3C, 2006.

[99] Marinos Themistocleous and Z. Irani. Towards a novel framework for the assessment of enterprise application integration packages. In *Proceedings of the 36th Hawaii International Conference on System Sciences (HICSS'03)*, Hawaii, Usa, 2003.

[100] Regione Toscana. Cooperazione applicativa regionale toscana . `http://www.cart.rete.toscana.it/portal/view/index.jsp`, accessed 2010.

[101] Regione Toscana. Rfc regione toscana . `http://web.rete.toscana.it/eCompliance/portale/cercaRFC`, accessed 2010.

[102] M. Uschold and M. Gruninger. Ontologies and semantics for seamless connectivity. *Sigmod Record*, 33(4):58–64, December 2004.

[103] Ubbo Visser, Heiner Stuckenschmidt, Holger Wache, and Thomas Vögele. Enabling technologies for interoperability. In *TZI, University of Bremen*, pages 35–46, Bremen, Germany, 2000. TZI.

[104] Max Voelkel and York Sure. Rdfreactor - from ontologies to programmatic data access. In *Jena User Conference*, Bristol, UK, May 2006.

[105] W3C. Extensible Markup Language
. `http://www.w3.org/TR/2008/REC-xml-20081126/`, accessed 2010.

[106] W3C. Ontology Web Language (OWL)
. `http://www.w3.org/2004/OWL/`, accessed 2010.

[107] W3C. OWL-S: Semantic Markup for Web Services
. `http://www.w3.org/Submission/OWL-S/`, accessed 2010.

[108] W3C. RDF Vocabulary Description Language 1.0: RDF Schema
. `http://www.w3.org/TR/rdf-schema/`, accessed 2010.

[109] W3C. Resource Description Framework
. `http://www.w3.org/TR/rdf-concepts/`, accessed 2010.

[110] W3C. SPARQL Query Language for RDF
. `http://www.w3.org/TR/rdf-sparql-query/`, accessed 2010.

[111] W3C. SWRL: A Semantic Web Rule Language Combining OWL and RuleML
. `http://www.w3.org/Submission/SWRL/`, accessed 2010.

[112] W3C. XML Schema
. `http://www.w3.org/TR/xmlschema-0/`, accessed 2010.

[113] H. Wache, T. Vogele, U. Visser, H. Stuckenschmidt, G. Schusterand, H. Neumann, and S. Hubner. Ontology-based integration of information – a survey of existing approaches. In *Proceedings of IJCAI-01 Workshop: Ontologies and Information Sharing*, pages 108–117, Seattle, WA, USA, 2001.

[114] H. Wasyluk, A. Onisko, and M J Druzdzel. Support of diagnosis of liver disorders based on causal bayesian network model. *Medical Science Monitor*, 7(Suppl. 1):327–332, May 2001.

[115] Pinar Wennerberg. Aligning medical domain ontologies for clinical query extraction. In *EACL '09: Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics: Student Research Workshop*, pages 79–87, Morristown, NJ, USA, 2009. Association for Computational Linguistics.

[116] Yi Yang and Jacques Calmet. Ontobayes: An ontology-driven uncertainty model. In *International conference on Computational Intelligence for Modeling, Control and Automation, and international conference on Intelligent Agents, Web tecnologies and Internet Commerce, CIMCA-IAWTIC'05*, 2005.

[117] Yi Yang and Jacques Calmet. From the ontobayes model to a service oriented decision support system. In *International conference on Computational Intelligence for Modeling, Control and Automation, and international conference on Intelligent Agents, Web tecnologies and Internet Commerce, CIMCA-IAWTIC'06*, 2006.

[118] Hai Tao Zheng, Bo Yeong Kang, and Hong Gee Kim. An ontology-based bayesian network approach for representing uncertainty in clinical practice guidelines. In *6th International Semantic Web Conference ISWC2007, Workshop on Uncertainty Reasoning for the Semantic Web*, 2007.

[119] Hong Zhou, Feng Chen, and Hongji Yang. Developing application specific ontology for program comprehension by combining domain ontology with code ontology. In *QSIC '08: Proceedings of the 2008 The Eighth International Conference on Quality Software*, pages 225–234, Washington, DC, USA, 2008. IEEE Computer Society.

[120] Michael Zimmermann. Owl2Java - A Java Code Generator for OWL . `http://www.incunabulum.de/projects/it/owl2java`, accessed 2010.

[121] LI Zong-yong, WANG Zhi-xue, YANG Ying-ying, WU Yue, and LIU Ying. Towards a multiple ontology framework for requirements elicitation and reuse. *Computer Software and Applications Conference, Annual International*, 1:189–195, 2007.