Universita' degli studi di Genova

Facolta' di Ingegneria

Laurea Specialistica in Ingegneria Informatica

Sistema di Supporto Mnemonico

Giorgio Ravera

Relatore:
Chiar.mo Prof. Ing. Mauro Migliardi

Anno Accademico 2007/2008

Indice

Ringraziamenti

Nell'arco dei cinque anni in cui ho frequentato la Facoltà di Ingegneria dell'Università di Genova la mia vita è cambiata. Mi sono iscritto nel 2003 dopo aver terminato il liceo scientifico di Savona, di cui ho un pessimo ricordo, sia sul piano umano, sia su quello didattico. Da questo ho tratto forza e voglia di riscatto che mi hanno permesso di poter frequentare l'università con spirito di conoscenza e rivincita.

Qui sono rinato, imparando cosa sia l'impegno, la dedizione e l'amicizia di persone straordinarie che mi hanno dato speranza e gioia anche nei momenti più tristi. Da quando sono entrato in Openlab ho imparato cosa vuol dire responsabilità e ho cercato di ricambiare, a modo mio, la fiducia che molte persone hanno riposto in me. Essere stato presidente del club per tre anni è stata un'esperienza indimenticabile. Non è sempre stato tutto facile. Le difficoltà mi hanno aiutato a crescere, a capire i miei limiti e a superarli. E' proprio a tutti i membri dell'Openlab che va il mio pensiero e ringraziamento per avermi sopportato e sostenuto, difeso e incoraggiato in ogni momento. Non dimenticherò mai i giorni passati insieme in aula e5 tra studio e sano divertimento, le discussioni su tematiche informatiche e non, i seminari da noi tenuti, le edizioni del Linux Day a cui abbiamo partecipato e i nostri interventi a fiere quali Tecno-Acqui e la mostra internazionale del cinema indipendente.

Tra tutti non posso fare a meno di citare le persone che sono state più presenti nel mio percorso e fondamentali: Alessandro (Cerve), Andrea, Nicolò, Emanuele, Andrea (Teddy), Fabio (il Vecchio), Alessandro (Lomba) e consorte, Simone, Davide (Fuia), Gabriele (Palma), Mauro (Nobile), Fabio (Igno), Andrea (Ros), Gianluigi, Tommaso, Stefano (Benve), Massimiliano e Sara. Un pensiero speciale va a Gianluca con il quale ho condiviso momenti imortanti, in particolare durante il tirocinio triennale.

Un ringraziamento speciale è dedicato ai miei amici di Savona e Genova con i quali ho passato serate molto belle e importanti: Pier, Orges, Isida, Matteo (Colo), Davide, Luca, Matteo (Pacchia), Diego (Giaddi), Valerio, Lara, Simone, Alessia, Luca, Ambra, Francesca, Alessandro (Suzzi), Cecilia, Alessandro (Delfo), Alessandro (Tixe) e alla mia compagnia estiva di Lonano: Ivan, Simone, Simone. Un pensiero particolare per Angela e Beatrice con le quali ho intrattenuto splendide conversazioni riflessive che mi hanno permesso di capire molti errori ed evitare di commetterli nuovamente ma, soprattutto, mi hanno tenuto compagnia in momenti difficili e mi hanno sempre motivato a completare i miei studi. Il ricordo dei momenti vissuti insieme è un elemento fondamentale che mi accompagnerà sempre.

Ringrazio anche il Dott. Stefano Lassi e Dott. Alessandro Pozzi per avermi permesso

di imparare moltissimo nell'ambito delle reti e per avermi dato sempre buoni consigli su come risolvere situazioni spiacevoli.

Un ringraziamento speciale va al Prof. Renato Zaccaria che ha sempre creduto in me, sostenuto e difeso. E' stato sempre presente negli ultimi anni dandomi consigli e lezioni di vita di cui ho fatto tesoro e mi serviranno negli anni futuri. Ringrazio anche i Prof. E. Giunchiglia, Prof. A. Tacchella, Prof. A. Armando, Prof. A. L. Frisiani, Prof. S. Zappatore, Prof T. Vernazza, Prof. G. Cainarca, Prof. A. Morro e Prof.sa C. Zordan e la manager didattica di informatica Dott.sa V. Resaz che sono stati dei riferimenti importanti e hanno avuto la pazienza di rispondere alle mie domande dandomi buoni suggerimenti nell'arco della mia carriera universitaria.

Negli ultimi tre anni, a partire dal tirocinio triennale fino a quello specialistico, la figura che è stata, di fatto, il mio maestro e riferimento, sia umano che didattico, è il Prof. Mauro Migliardi che ringrazio sia per la infinita pazienza dimostrata nel tempo trascorso a lavorare insieme, sia per tutti i suoi insegnamenti, senza i quali non avrei potuto terminare entrambe le tesi ed apprendere molti concetti del mondo dell'informatica, sia per la grande fiducia riposta in me. Ricorderò sempre, con molto piacere, le interessanti conversazioni svolte su argomenti di ogni genere, tutti i saggi consigli e gli aiuti che non mi ha fatto mai mancare. A lui va anche il merito di avermi mostrato il vero significato del termine *nerd* e, con orgoglio e fierezza, posso dichiarare di essere stato suo allievo.

Ringrazio anche mia zia Assunta, mia nonna Antonia, scomparsa nel 2005, con la quale sono cresciuto ed ho condiviso momenti splendidi, mio nonno Carlo che, pur non avendolo mai conosciuto ha rappresentato per me una figura importante da imitare, mio nonno Libero, mia nonna Anna, Nastasia e mio cugino Piero che mi ha tenuto molta compagnia in momenti difficili conversando su svariati argomenti relativi ai moltissimi interessi in comune.

Non posso fare a meno di ricordare mio padre, Giuseppe Ravera, scomparso nel 2005, che mi ha insegnato cosa vuol dire essere un Ingegnere e alcuni valori fondamentali quali l'onestà, la correttezza e l'umanità sia nei momenti trascorsi insieme, sia nel modo con cui ha affrontato la sua lunga e difficile malattia.

Infine ringrazio mia mamma, Anna Bonelli, a cui dedico l'intero testo. Mi è stata vicino sempre da quando sono nato, è stata un riferimento importante, capace di trovare il meglio di me e tirarlo fuori, anche in situazioni difficili dandomi l'appoggio e l'amore di cui avevo bisogno. E' stata una guida e un supporto in ogni situazione e, in particolare dalla scomparsa di mio padre, ha sacrificato molto per permettermi di arrivare a questo traguardo. Senza di lei non sarei la persona che sono oggi.

Introduzione

In campo psico-fisiologico recenti studi correlano lo stress con la difficoltà a trasferire informazioni dalla memoria a breve termine a quella a medio termine. Questa difficoltà spesso ingenera una riduzione dell'efficienza personale poiche' le cose da fare vengono in mente in modo disordinato e non pianificato. La forma estrema è *Activity Thrashing* cioè l'incapacità di concludere alcunchè in quanto qualcosa di diverso da fare continua a venire in mente. E' nostra opinione che questo fenomeno potrebbe essere tenuto sotto controllo fornendo ai soggetti puntuali informazioni relative a quali attività possono essere efficientemente svolte nel suo attuale contesto.

Questa tesi si pone all'interno di un progetto che mira a studiare e sviluppare un sistema che analizzi il linguaggio parlato dell'utente e lo traduca in attività prioritarizzate. Queste attività verranno poi tradotte in interrogazioni ad un sistema informativo geografico per fornire all'utente indicazioni sulle attività che possono essere efficacemente svolte nel suo contesto.

Il sistema progettato all'interno di questa tesi ha struttura modulare sia per tenere sotto controllo la complessità, sia per facilitare il refactoring dei diversi componenti. Per poter effettuare con successo l'integrazione di componenti e sottosistemi diversi come quelli che compongono il sistema scopo di questo progetto, è necessario averne una buona conoscenza. Per questo motivo la tesi si sviluppa in diverse sezioni, le prime dedicate a introdurre le tematiche generali affrontate e quelle seguenti a descrivere nel dettaglio il sistema sviluppato e gli esperimenti effettuati.

Lo sviluppo di tale software ha reso necessario un approfondimento di teorie riguardanti l'ingegneria della conoscenza in alcuni suoi aspetti quali l'intelligenza artificiale e modelli computazionali, tecniche di elaborazione del linguaggio naturale e di rappresentazione della conoscenza.

L'idea è quella di presentare prima le basi teoriche che stanno dietro all'intero progetto (prima parte), per poi averne riscontro durante l'analisi del codice (seconda parte). Tale approccio porta a una maggiore comprensione del lavoro svolto.

Il primo aspetto che verrà approfondito riguarda gli agenti e il loro utilizzo. Rifacendosi alla definizione di IBM:

Intelligent agents are software entities that carry out some set of operations on behalf of a user or another program with some degree of independence or autonomy, and in so doing, employ some knowledge or representation of the user's goals or desires.[1]

[1] `http://www.msci.memphis.edu/~franklin/AgentProg.html`

Capire cosa sia un agente intelligente è fondamentale per il proseguo del lavoro: ogni singolo componente del sistema è identificabile come un agente software che svolge un ruolo fondamentale per lo scopo finale del sistema.

Il secondo aspetto presentato sarà quello riguardante l'analisi del linguaggio naturale. Verranno presentate le teorie alla base dell'elaborazione del linguaggio al fine di realizzare traduttori automatici, spesso applicate nella scrittura di compilatori e interpreti di linguaggi informatici. Nel nostro caso saranno utili al fine di elaborare il linguaggio parlato per la traduzione in costrutti più facili da gestire per un analisi semantica.

Infine, verranno presentate le tecniche di rappresentazione della conoscenza, utili al fine di poter dare significato a un discorso e poter effettuare ragionamenti utilizzando una memoria opportunamente realizzata.

Schema del Progetto

In figura 1 è riportato lo schema dell'intero progetto.

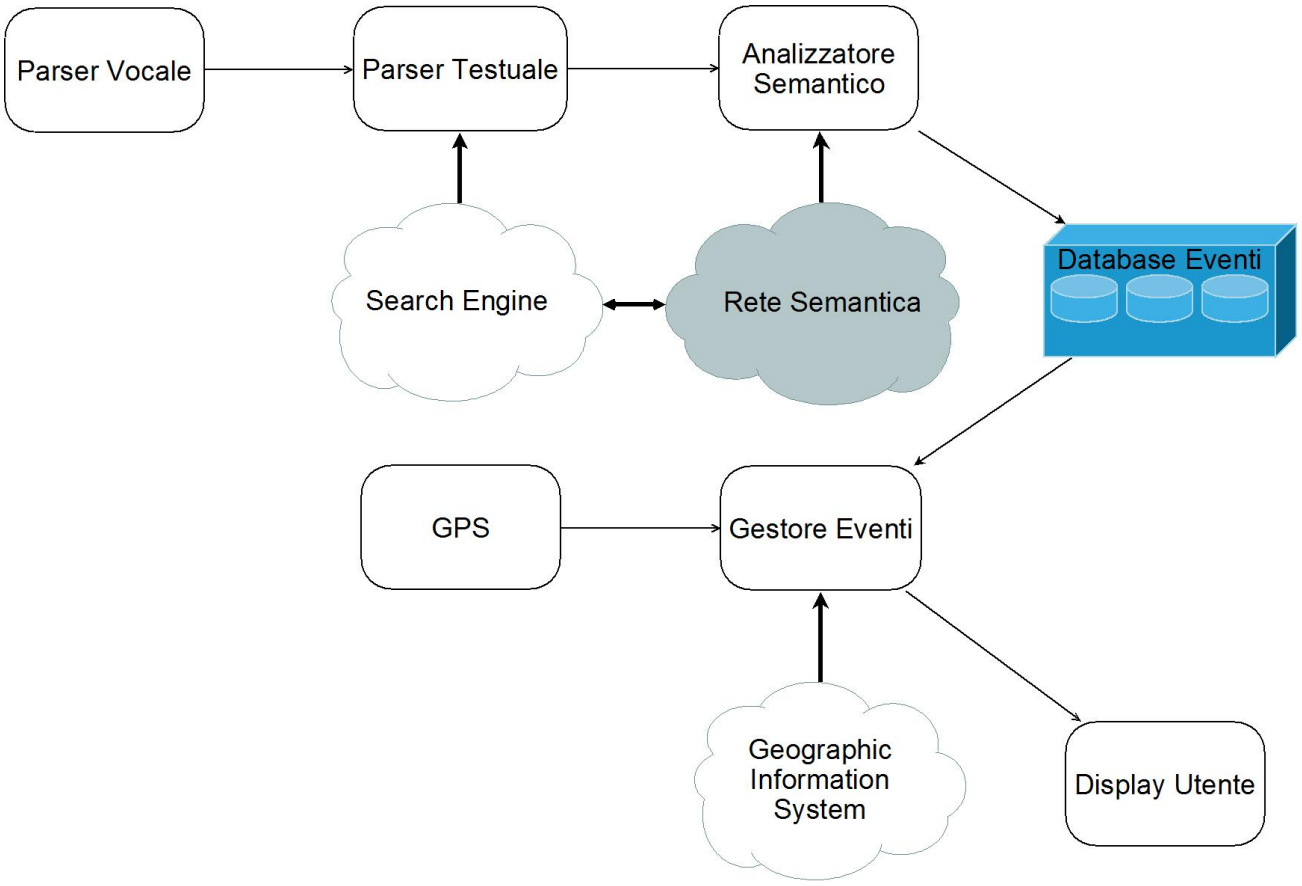


Figura 1: Schema del progetto

Come si può notare il software è composto da diversi elementi:

- **Riconoscitore Vocale**: traduce la voce in uno stream di stringhe
- **Parser Testuale**: codifica le stringhe in istanze di classi per l'analisi semantica e, per ogni frase, identifica soggetto, verbo e complemento oggetto con eventuali aggettivi.
- **Analizzatore Semantico**: dato un albero semantico cerca di identificare l'informazione che trasporta. Utilizza la rete semantica per aumentare la potenza di analisi cercando relazioni tra concetti al fine di individuare le categorie d'appartenenza dei concetti. Una volta identificate, vengono caricati nel database le chiavi di ricerca ad esse associate.
- **Database**: raccoglie tutte le chiavi di ricerca trovate, indicizzate secondo criteri di notifica e priorità.
- **GPS**: Global Positioning System: identifica la posizione dell'utente e invia tali informazioni al gestore di eventi.
- **Gestore delle Interrogazioni**: data la posizione dell'utente provvede ad estrarre dal database le chiavi di ricerca e creare la query di interrogazione al *GIS*. Il risultato viene processato e organizzato sulla base della distanza dal punto corrente dell'utente e presentato sul display.
- **Display Utente**: si occupa di mostrare all'utente le informazioni trovate sul *GIS* opportunamente processate sulla base della distanza e della priorità.
- **Motore di ricerca**: strumento utilizzato per effettuare ricerche su web. Viene utilizzato sia per identificare termini non riconosciuti dal parser, sia per aggiornare la rete semantica
- **Geographic Information System**: viene usato per ottenere informazioni precise riguardanti la posizione dell'utente, calcolare percorsi definendo sorgente e destinazione e per ottenere mappe dettagliate della zona circostante alla posizione dell'utente.
- **Rete Semantica**: si occupa di collezionare le relazioni tra i concetti e le categorie sulla base della definizione di un ontologia.

Parte I

Introduzione all'Ingegneria della Conoscenza

Capitolo 1

Agenti Intelligenti

1.1 Definizione di Agente

Un **agente** è un entità che agisce, cioé svolge un azione. Dev'essere in grado di percepire l'ambiente che lo circonda attraverso dei **sensori** ed eseguire delle azioni attraverso degli **attuatori**.

Un **agente intelligente** o **razionale** è un agente che agisce in modo da ottenere il migliore risultato o, in caso di incertezza, il migliore risultato atteso.

Consideriamo un esempio comune di agente: una persona umana. Quando un soggetto si trova a dover risolvere un problema per prima cosa analizza l'ambiente esterno, attraverso i sensori (occhi e orecchie). In secodo luogo prova ad elaborare, nella sua mente, un idea del problema e prova ad elaborare una soluzione ottimale correlando tra loro le informazioni in suo possesso (pregresse e acquisite). Infine, attraverso gli attuatori (braccia, piedi o, più in generale, muscoli) la applica.

A questa descrizione molto elementare possiamo ricondurre ogni singola attività, dalla soluzione di un problema matematico, alla scelta di quale canale televisivo guardare oppure semplicemente a quale strada percorrere quando si è davanti a un bivio.
Un agente è composto da due elementi fondamentali ed indivisibili:

- **architettura**: il sistema fisico (corpo umano, hardware di un calcolatore) che compone l'agente e gli consente di eseguire i calcoli.
- **programma**: un insieme di istruzioni che guidano l'agente nella risoluzione dei problemi e consentano di interpretare i segnali provenienti dal mondo esterno e ricodificare i risultati per controllare gli attuatori.

Esistono diverse tipologie di agenti che si differenziano in base alla struttura che implementano. Per ogni tipologia vi è un approccio teorico specifico.

Nei paragrafi successivi verranno approfonditi aspetti legati all'ambiente esterno e alla struttura di un agente.

1.2 Ambienti

Un agente interagisce continuamente con l'ambiente che lo circonda:

- preleva informazioni attraverso i sensori
- compie azioni attraverso gli attuatori

Con il termine **percezione** si indicano gli input che l'agente ottiene, tramite i sensori, dall'ambiente esterno in un dato istante. La **sequenza percettiva** è la storia completa di tutto ciò che l'agente ha percepito nella sua esistenza.

In generale la scelta dell'azione di un agente in un qualsiasi istante può dipendere dall'intera sequenza percettiva osservata fino a quel momento. Se possiamo specificare l'azione prescelta dall'agente per ogni possibile sequenza percettiva, allora abbiamo descritto l'agente in modo completo. Ciò implicherebbe avere una conoscenza completa dell'ambiente esterno e questo non è sempre possibile, come vedremo in seguito.

Si indentifica con **funzione agente** l'insieme di azioni che descrivono il comportamento dell'agente in risposta a ciascun elemento della sequenza percettiva. La si può vedere come una tabella che mette in corrispondenza di ogni percezione una delle possibili azioni. Nel caso non si conoscesse tale funzione è possibile ricavarla (interamente o solo parte di essa) provando tutte le possibili sequenze percettive e registrando il comportamento dell'agente.

La tabella appena descritta ha una validità *esterna* all'agente. Al suo interno ci sarà un **programma agente** che implementerà tale funzione. E' importante sottolinearne la differenza: la funzione è una descrizione matematica astratta, il programma è una sua implementazione concreta in esecuzione sull'architettura dell'agente.

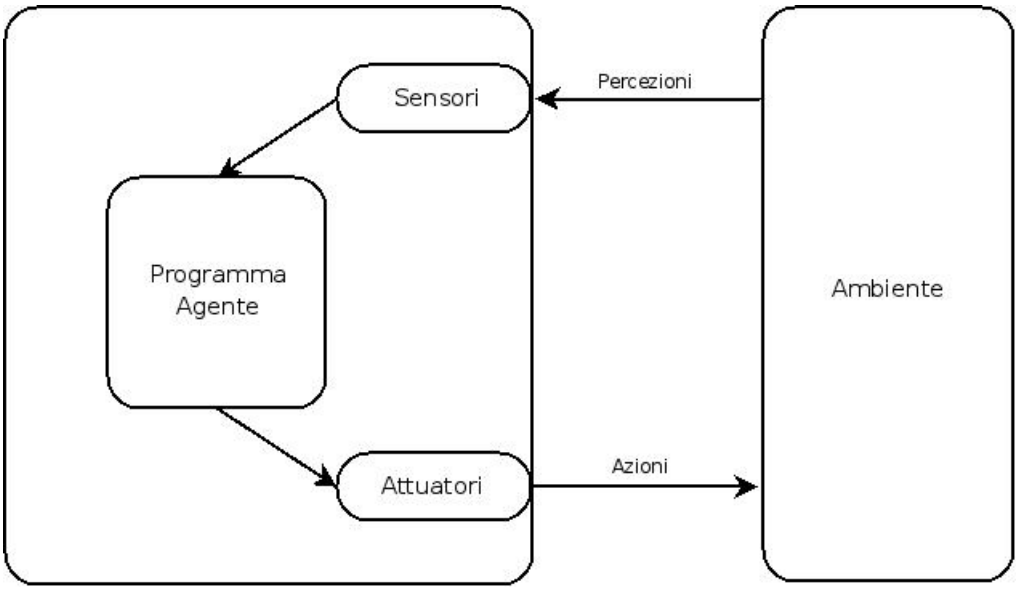


Figura 1.1: Schema di un generico Agente

La figura 1.1 illustra visivamente quanto detto fin ora: come si può vedere viene separato dall'ambiente esterno l'agente che è composto da sensori, attuatori e programma agente. Non è presentato un dettaglio di quest'ultimo per mantenere il più generica possibile la presentazione. Nel paragrafo 1.3 ne verranno mostrate le diverse implementazioni.

1.2.1 Tipologie di Ambienti

Per poter realizzare un agente con un dato obiettivo, è necessario definire l'ambiente che lo circonda. Da esso, infatti, l'agente trarrà tutte le informazioni necessarie per poter effettuare le decisioni e gli elementi per poterle attuare.

E', quindi, necessario identificare delle proprietà in base a cui suddividerli in categorie. Queste proprietà determinano in gran parte la progettazione di agenti appropriati e l'applicabilità delle principali tecniche alla loro implementazione.
Le principali proprietà sono le seguenti:

- **osservabilità completa/parziale**: caratterizza il grado d'accesso dei sensori al mondo esterno. Un ambiente parzialmente osservabile può essere dovuto a sensori inaccurati o imprecisi, alla presenza di rumore oppure all'impossibilità di ottenere alcune informazioni.

- **deterministico/stocastico**: nel caso di determinismo l'agente riesce a prevedere lo stato successivo dell'ambiente sulla base della misura fatta e dell'azione. Qualora ciò non fosse possibile si parla di ambiente stocastico. Nel caso in cui l'ambiente è deterministico ad eccezione delle azioni che altri agenti potrebbero compiere si parla di ambiente **strategico**.

- **episodico/sequenziale**: un ambiente si dice episodico se ogni azione generata non dipende dalle precedenti. Sequenziale nel caso in cui più percezioni influenzano la scelta dell'azione da svolgere.

- **statico/dinamico**: un ambiente è statico nel caso in cui non può cambiare nel periodo in cui l'agente ragiona al fine di prendere la decisione ottima. E' dinamico quando può variare durante la fase di ragionamento. Ciò comporta grossi problemi perché l'azione generata potrebbe non essere più quella corretta. Se l'ambiente non cambia con il tempo, ma la valutazione dell'agente si, allora si dice **semidinamico**

- **discreto/continuo**: tale classificazione è applicata allo *stato* dell'ambiente e a come viene rappresentato il *tempo* alle percezioni e azioni dell'agente. Può essere discreto, campionato ad intervalli regolari, oppure continuo, analizzato con continuità senza salti.

- **agente singolo/multiagente**: si parla di ambiente a singolo agente nel caso in cui eventuali altri elementi autonomi nell'ambiente non influenzano le scelte effettuate. Si parla di multiagente quando altri agenti possono partecipare attivamente al raggiungimento dell'obiettivo.

Si può capire facilmente che il caso più complesso è dato dalla combinazione di ambiente: *parzialmente osservabile, stocastico, sequenziale, dinamico, continuo e multiagente.*

1.3 Struttura di un agente

Come già visto in precedenza vale la seguente legge:

$$agente = architettura + programma \tag{1.1}$$

Con **architettura** si identifica la componente fisica, materiale dell'agente, ovvero l'hardware, i circuiti e le periferiche che lo compongono. E' composta prevalentemente da quattro differenti elementi:

- **strumenti di calcolo**: componenti che si occupano di elaborare i dati per relazionarli ed ottenere gli strumenti per poter effettuare le decisioni. In genere sono rappresentati dal/dai processore/i.
- **sensori**: analizzano l'ambiente esterno e codificano le informazioni raccolte in dati analizzabili dagli apparati di calcolo. Sono in genere microfoni, tastiere, videocamere o misuratori di temperatura.
- **attuatori**: componenti che ricevono istruzioni dagli apparati di calcolo e li traducono in azioni, come ad esempio il movimento di un braccio o la produzione di un segnale visivo/uditivo. Alcuni esempi possono essere display o casse acustiche.
- **memoria**: in alcuni agenti è presente la possibilità di memorizzare informazioni per poter avere più dati da tenere in considerazione nell'elaborazione di decisioni. Possono essere realizzati mediante database memorizzati su dischi o nastri.

Il **programma agente** è un software che deve essere in grado di:

- acquisire correttamente i dati provenienti dai sensori
- codificare tali dati per ottimizzare la funzione di decisione/ragionamento
- inviare agli attuatori azioni da svolgere compatibili con la loro progettazione

Come si può osservare dalla sua definizione, tale programma deve essere progettato per una specifica architettura e, quindi, risultare compatibile con i sensori e gli attuatori scelti.

E' importante sottolinearne la differenza tra il programma agente che prende come input solamente la percezione corrente e la funzione agente il cui input è costituito dall'intera storia delle percezioni. Il programma agente si basa sulla sola percezione corrente perché l'ambiente non può fornirgli nulla di più. Se le sue azioni dipendono dalla sequenza percettiva precedente è necessario utilizzare un sistema di memorizzazione.
Il seguente pseudocodice dovrebbe aiutare a capire la struttura base di un programma:

```
class agente-semplice
{
        map<perception, action> m;

        action calcola(perceptions p)
        {
                return m.lookup(p);
        }
}
```

Esistono sei tipologie diverse di agenti associate a una opportuna tipologia di programma agente che rappresentano i principi alla base di quasi tutti i sistemi intelligenti:

- agenti reattivi semplici
- agenti basati su modello
- agenti basati su obiettivi
- agenti basati sull'utilità
- agenti che apprendono
- agenti basati sulla conoscenza

1.3.1 Agenti Reattivi semplici

Un *agente reattivo semplice* è un agente che sceglie le azioni sulla base della sola percezione corrente, ignorando completamente quelle passate. In altre parole non ha memoria.

Un tale agente è, appunto, molto semplice da realizzare ma ha una intelligenza molto limitata. Si può schematizzare il suo operato attraverso il seguente pseudocodice:

```
class agente-reattivo-semplice
{
        map<state, list<rule>> rules;
        map<rule, action> actions;

        action calcola(perceptions p)
        {
                state s = code_percption(p);
                rule r1 = (rules.lookup(s, p)).getFirst();
                return actions.lookup(r1);
        }
}
```

Per prima cosa, viene codificata in forma astratta la percezione attraverso una sua elaborazione (funzione code_perception). Da qui si potrà ottenere uno stato dell'ambiente esterno. A questo punto è necessario interpretare lo stato e capire che **regola condizione-azione** attribuire a ciò che si è osservato. Nel caso in cui ci fossero più regole per lo stesso stato verrà considerata sempre e solo la prima. A questo punto si procede cercando l'azione da compiere.

Interpretiamo con un esempio quanto appena descritto. Consideriamo un sistema di guida autonomo: ci troviamo dietro ad un'automobile e dobbiamo capire quando questa frena per fare la stessa cosa. I sensori sono una serie di telecamere montate sul parabrezza, gli attuatori sono i freni. Ad intervalli regolari viene catturata un immagine: dall'immagine (percezione) si studiano le luci di segnalazione e in particolare se sono accese o spente (stato). In base al valore appena elaborato si cerca una corrispondenza tra questo e l'azione da svolgere. Si estrarrà, in caso di luci accese una regola condizione azione del tipo:

if (**light_on**) then **start_to_break**

Con questa regola diventa semplice capire che azione svolgere, nel caso corrente frenare.

Anche gli esseri umani fanno uso di connessioni analoghe, alcune basate sull'apprendimento, altre sui riflessi. Questi aspetti verranno approfonditi nei capitoli successivi.

Un tale agente funziona solo e unicamente nel caso in cui l'ambiente sia **completamente osservabile**. Anche una minima parte di inosservabilità potrebbe causare errori molto gravi. Nel caso in esempio la struttura potrebbe non prevedere la possibilità che le luci di segnalazione della macchina davanti non funzionino, oppure non differenziarli dalle luci di posizione. Limitandosi, quindi, alla sola analisi delle luci il sistema non funziona.

Infine un tale agente tende ad eseguire **cicli infiniti** se l'azione da compiere è sempre la stessa per ogni circostanza e lo stato non può mutare per peculiarità proprie dell'azione stessa. Supponiamo di avere un robot che deve muoversi in un terreno in relazione al colore delle piastrelle: se le piastrelle sono bianche si muove a destra o in basso, se sono nere a sinistra o in alto. A un tratto arriva in una piastrella nera con alla sua sinistra un muro. L'azione che dovrebbe essere eseguita sarà il movimento verso sinistra (prima regola) ma non sarà possibile e alla percezione successiva lo stato sarà analogo al precedente.

Per ovviare a questo problema è possibile scegliere in maniera casuale le azioni da svolgere. Ciò dovrebbe evitare cicli infiniti.

1.3.2 Agenti basati su Modello

Il modo più efficace di gestire l'osservabilità parziale, per un agente, è tener traccia della parte del mondo che non può vedere nell'istante corrente. Questo significa che l'agente deve memorizzare una sorta di stato interno che dipende dalla storia delle percezioni e che quindi riflette almeno una parte degli aspetti non osservabili dello stato corrente.

Ritornando al problema della frenata, lo stato interno non è troppo complesso: basta mantenere in memoria il fotogramma precedente scattato dalla telecamera, permettendo cosi all'agente di accorgersi quando due luci rosse, alle estremità laterali del veicolo, si accendono o spengono contemporaneamente.

Aggiornare l' informazione di stato al passaggio del tempo richiede che il programma agente possieda due tipi di conoscenza:

- informazioni sull'evoluzione del mondo indipendente dalle sue azioni
- informazioni sull'effetto che hanno sul mondo le azioni dell'agente stesso

Questa conoscenza sul funzionamento del mondo, viene chiamata **modello del mondo**. Un agente che si appoggia a un simile modello prende appunto il nome di **agente basato su modello**.
Si può schematizzare il suo operato attraverso lo pseudocodice sottostante:

```
class agente-reattivo-con-stato
{
        stete s;
        actions last_action;
        map<state, list<rule>> rules;
        map<rule, action> actions;

        action calcola(perceptions p)
        {
                s.update-state(p,last_action);
                rule r1 = (rules.lookup(s, p)).getFirst();
                last_action = actions.lookup(r1);
                return last_action;
        }
}
```

La parte interessante è la funzione update-state(), responsabile della creazione del nuovo stato interno. Oltre a interpretare la nuova percezione alla luce della conoscenza preesistente dello stato, la funzione utilizza le informazioni sull'evoluzione del mondo per tener traccia delle parti di esso che non sono presentemente visibili. La funzione deve anche conoscere l'effetto delle azioni dell'agente sullo stato del mondo.

1.3.3 Agenti basati su Obiettivi

Conoscere lo stato corrente dell'ambiente non sempre basta a decidere che cosa fare. Oltre che della descrizione corrente dello stato l'agente ha bisogno di qualche tipo di informazione riguardante il suo obiettivo. Il programma agente può unire quest'informazione a quella che riguarda i risultati delle possibili azioni (la stessa usata anche per aggiornare lo stato interno in un agente reattivo) per scegliere quelle che portano al soddisfacimento dell'obiettivo.

Talvolta scegliere un'azione in base a un obiettivo è molto semplice, quando questo può essere raggiunto in un solo passo. Altre volte è più difficile, quando l'agente deve considerare lunghe sequenze di azioni alternative per trovare il cammino che porta al risultato

desiderato. La ricerca e la pianificazione sono dei sottocampi dell'Intelligenza Artificiale dedicati proprio a identificare le sequenze di azioni che permettono a un agente di raggiungere i propri obiettivi.

Notate che questo tipo di decisioni non ha nulla a che vedere con le regole condizione-azione che abbiamo descritto prima, perchè ora dobbiamo prendere in considerazione il futuro sotto due aspetti:

- cosa accadrà se si esegue l'azione prescelta?
- il risultato dell'azione sarà soddisfacente?

Nella progettazione degli agenti reattivi quest'informazione non viene rappresentata esplicitamente, perchè le regole interne mettono direttamente in corrispondenza percezioni e azioni. L'agente reattivo frena quando vede le luci di frenata. Un agente basato su obiettivi, in via di principio, potrebbe ragionare che se la macchina davanti ha le luci di frenata accese starà rallentando. Dato il modo in cui evolve normalmente il mondo, l'unica azione che permetterebbe di raggiungere l'obiettivo di non tamponare altre macchine sarà allora quella di frenare.

Benchè un agente basato su obiettivi sembri meno efficiente, d'altra parte è più flessibile, perchè la conoscenza che motiva le sue decisioni è rappresentata esplicitamente e può essere modificata. Se comincia a piovere, l'agente può aggiornare la propria conoscenza circa l'efficienza dei propri freni. Nel caso dell'agente reattivo semplice sarà necessario riscrivere molte regole condizione-azione.

1.3.4 Agenti basati sull'Utilità

Nella maggior parte degli ambienti gli obiettivi, da soli, non bastano a generare un comportamento di alta qualità. Ci sono molte sequenze di azioni che porteranno al raggiungimento dell'obiettivo ma alcune potrebbero risultare migliori, in termini di costo, sicurezza o affidabilità, rispetto ad altre.

Gli obiettivi forniscono solamente una distinzione binaria tra stati soddisfacenti o meno, Occorre una misura di prestazione più generale permettendo di confrontare stati del mondo differenti e misurare precisamente il grado di contentezza che si otterrebbe se l'agente riuscisse a raggiungerli.

Una funzione di utilità assegna a uno stato (o a una sequenza di stati) un numero reale che quantifica il grado di preferibilità ad esso associato. Una specifica completa della funzione di utilità permette decisioni razionali in due categorie di casi in cui non bastano gli obiettivi:

- **conflitti**: quando ci sono più obiettivi da raggiungere che non si possono ottenere insieme. La funzione di utilità specifica quali privilegiare.
- **incertezza**: quando ci sono più obiettivi raggiungibili ma nessuno può essere ottenuto con certezza. Il concetto di utilità fornisce un mezzo per confrontare le probabilità di successo e l'importanza degli obiettivi.

Ogni agente razionale deve comportarsi come se possedesse una funzione di utilità di cui cerca di massimizzare il valore atteso. Un agente che possiede una funzione di utilità esplicita può quindi prendere decisioni razionali, e lo può fare seguendo un algoritmo generale che non dipende dalla specifica funzione di utilità che si desidera massimizzare. In questo modo la definizione globale di razionalità, che definisce razionali quelle funzioni agente che hanno le prestazioni migliori, viene trasformata in un vincolo locale in progetti di agenti razionali che possono esseri espressi in un semplice programma.

1.3.5 Agenti che Apprendono

Secondo Turing, in un articolo pubblicato nel 1950, è possibile costruire macchine capaci di apprendere e poi addestrarle. L'apprendimento presenta un enorme vantaggio: permette agli agenti di operare inizialmente in ambienti sconosciuti, diventando col tempo più competenti di quanto fossero all'inizio, allorchè si basavano sulla sola conoscenza iniziale. Un agente capace di apprendere può essere diviso in quattro componenti astratti:

- elemento di apprendimento
- elemento esecutivo
- elemento critico
- generatore di problemi

La distinzione più importante è tra: **elemento di apprendimento** (learning element), responsabile del miglioramento interno, e **l'elemento esecutivo** (performance element) che si occupa della selezione delle azioni esterne. Quest'ultimo è ciò che abbiamo considerato fin qui come se costituisse l'intero agente: prende in input le percezioni e decide le azioni. L'elemento di apprendimento utilizza informazione proveniente dall'elemento critico riguardo le prestazioni correnti dell'agente e determina se e come modificare l'elemento esecutivo affinchè in futuro si comportino meglio. Il progetto dell'elemento di apprendimento dipende molto da quello dell'elemento esecutivo. Quando si cerca di progettare un agente che impara a svolgere una certa attività, la prima domanda da porsi non è *come faccio a fargli imparare questa cosa?* ma *quale tipo di elemento esecutivo permetterà al mio agente di fare questa cosa, una volta l'avrà imparata?*. Qualsiasi progetto di agente può essere migliorato in ogni sua parte dall'aggiunta di meccanismi di apprendimento.

L'**elemento critico** dice a quello di apprendimento come si sta comportando l'agente rispetto a uno standard di prestazione fissato. Quest'elemento è necessario perchè le percezioni, in sé, non forniscono alcuna indicazione del successo dell'agente. Un programma di scacchi potrebbe ricevere una percezione che indica che ha dato scacco matto all'avversario, ma ha bisogno di uno standart di prestazione per sapere che questa è una cosa buona; la percezione da sola non basta. E importante che lo standart sia prefissato: concettualmente lo si può pensare come un'entità del tutto separata dall'agente, dato che quest'ultimo non lo deve modificare per adattarlo al suo comportamento.

L'ultimo componente di un agente capace di apprendere è il **generatore di problemi**, il cui scopo è suggerire azioni che portino a esperienze nuove e significative. L'idea è che se si lasciasse mano libera all'elemento esecutivo, esso continuerebbe a ripetere le azioni che ritiene migliori date le conoscenze attuali. Ma se l'agente è disposto a esplorare qualche altra possibilità, e magari eseguire nel breve termine qualche azione subottima, potrebbe scoprire l'esistenza di azioni molto superiori a lungo termine. Scopo del generatore di problemi è di suggerire tali azioni esplorative.

Provando ad applicare lo schema dell'agente che apprende al problema di guidare un taxi, si può sostenere che l'elemento esecutivo corrisponde alla collezione di conoscenze ed procedure usate dal taxi per selezionare le possibili azioni di guida. L'elemento critico osserva il mondo e passa delle informazioni a quello di apprendimento. Da esse l'elemento di apprendimento può formulare una regola che viene aggiunta all'insieme. A questo punto il generatore di problemi potrebbe identificare certe aree di comportamento che necessitano di qualche miglioria e suggerire esperimenti da effettuare.

Tale tipologia di agente è applicabile a tutti quelli analizzati in precedenza. L'elemento di apprendimento, infatti, può modificare uno qualsiasi dei comportamenti degli agenti appena descritti.

1.3.6 Agenti basati sulla Conoscenza

Gli agenti visti fin ora appartengono all'insieme degli agenti reattivi che operano attraverso il calcolo delle conseguenze delle azioni. Questo porta ad ottenere una conoscenza limitata non deduttiva della realtà. Consideriamo gli scacchi: un agente reattivo sa calcolare le mosse di ogni singola pedina e le conseguenze di ogni azione ma, in senso generale, non sa giocare a scacchi.

Gli agenti basati sulla conoscenza possono trarre beneficio da conoscenze espresse in forma generale, combinando e ricombinando le informazioni per adattarle a una varietà di scopi. Spesso, tale procedimento può non avere un legame diretto con le necessità immediate. La conoscenza la si può suddividere in due elementi fondamentali:

- **regole apprese**: procedure, informazioni e, in generale, conoscenza acquisita da uno studio o trasmessa da agenti o persone
- **schemi di associazione**: collegamenti tra le informazioni

Il componente più importante degli agenti basati sulla conoscenza è appunto la **base di conoscenza**, o KB (dall'inglese knowledge base). Informalmente, la base di conoscenza è costituita da un insieme di formule, espresse mediante un linguaggio di rappresentazione della conoscenza. Ogni formula rappresenta un'asserzione sul mondo.

La base di conoscenza deve prevedere meccanismi per raggiungere nuove formule e per le interrogazioni. I nomi standard per queste due azioni sono rispettivamente TELL (asserisci) e ASK (chiedi): entrambe possono comportare un processo di inferenza, ovvero la derivazione di nuove formule a partire da quelle conosciute. La risposta a ogni richiesta (ASK), posta alla base di conoscenza, sia una conseguenza di quello che le è stato detto

(attraverso TELL) in precedenza.
Si può schematizzare quanto detto attraverso lo pseudocodice sottostante:

```
class KB-agent
{
        database knowledge;

        action calcola(perceptions p)
        {
                knowledge.tell(p);
                action a = knowledge.ask(p);
                knowledge.tell(a);
                return a;
        }
}
```

L'agente mantiene in memoria una base di conoscenza, KB, che può contenere conoscenza iniziale (background knowledge). Ogni volta che viene invocato, il programma agente fa due cose

- comunica le sue percezioni alla base di conoscenza tramite la funzione TELL.
- chiede quale azione eseguire tramite la funzione ASK

Rispondere a questa domanda può comportare un esteso processo di ragionamento sullo stato corrente del mondo, le conseguenze delle possibili azioni e cosi via. Una volta che è stata scelta una azione l'agente la registra con TELL prima di eseguirla. Il secondo TELL è necessario per dire alla base di conoscenza che l'ipotetica azione è stata effettivamente eseguita.

I dettagli del linguaggio di rappresentazione sono racchiusi nelle tre funzioni che implementano l'interfaccia tra i sensori e attuatori da una parte e il sistema interno di rappresentazione e ragionamento dall'altra. Un agente siffatto si presta bene a essere descritto a livello di conoscenza, in cui per fissare il comportamento basta specificare solamente ciò che l'agente conosce e i suoi obiettivi. Ad esempio, un taxi automatico potrebbe avere l'obiettivo di portare un passeggero in Via Opera Pia 13, e potrebbe sapere di trovarsi a Genova. Allora ci aspetteremo che il taxi identifichi la via migliore per raggiungere la destinazione, perchè sa che cosi facendo raggiungerà il suo obiettivo.

Notate che quest'analisi è del tutto indipendente dal funzionamento del taxi a livello di implementazione: non importa se la sua conoscenza della geografia è realizzata con liste dinamiche o mappe di pixel, o se per ragionare manipola stringhe di simboli memorizzate in registri o propaga segnali in una rete neurale.

Si possono costruire agenti basati sulla conoscenza semplicemente dicendo loro quello che devono sapere. Il programma agente iniziale, prima di cominciare ad analizzare gli input sensoriali, dovrà aggiungere nella sua base di dati tutte le formule che rappresentano la conoscenza dell'ambiente circostante, opportunamente realizzate dal progettista.

Per fare ciò sarà necessario ricorrere ad un linguaggio di programmazione che rende facile esprimere la conoscenza per mezzo di formule. Tale approccio prende il nome di **approccio dichiarativo**. La base di conoscenza potrà essere ampliata attraverso una fase di apprendimento.

Capitolo 2

Elaborazione del Linguaggio Naturale

2.1 Comunicare informazioni

La **comunicazione** consiste nello scambio intenzionale di informazioni attraverso la produzione e percezione di **segni** appartenenti a un sistema convenzionale e condiviso. E' uno strumento attraverso il quale è possibile condividere informazioni.

Il **linguaggio** è il complesso sistema di messaggi strutturati che permettono a due o più entità di comunicare con chiarezza una parte della loro conoscenza. Solitamente un linguaggio prevede *regole sintattiche* e *semantiche* che consentono a due o più interlocutori di capirsi. Appare ovvio, da questa definizione che due o più agenti che vogliano comunicare tra loro devono condividere il linguaggio utilizzato.

Un agente, per produrre un linguaggio, compie un azione che prende il nome di **atto linguistico**. Con questo termine non si intende solamente la produzione di linguaggio parlato ma, più in generale, tutto ciò che può costituire una comunicazione. Ciò che viene prodotto è un insieme di segni comunicativi chiamati **parole**.

Un agente può intraprendere differenti tipologie di atti linguistici:

- interrogare altri agenti su aspetti dell'ambiente esterno
- informare altri agenti riguardo aspetti dell'ambiente esterno
- richiedere ad altri agenti di eseguire azioni
- acconsentire alle richieste
- promettere o impegnarsi di svolgere azioni
- dichiarare situazioni o proprietà del mondo esterno

Il riconoscimento della tipologia dipende dalla frase stessa. E' simile a qualsiasi altro **problema di comprensione**, come il riconoscimento di immagini o la diagnosi medica. L'agente riceve un insieme di input ambigui e deve procedere all'indietro per decifrarne il significato.

2.2 Fondamenti del linguaggio

Un **linguaggio formale** è definto da un insieme di **stringhe**, ciascuna costituita da **simboli terminali** chiamati **parole**. Hanno una sintassi molto rigorosa e sono facilmente interpretabili.

Un **linguaggio naturale**, invece, è definto da un insieme di stringhe costituite da **simboli terminali** ma non presentano una sintassi rigorosa. Sono i linguaggi utilizzati dalle persone per esprimersi.

Una **grammatica** è un insieme finito di regole che specifica un linguaggio. I linguaggi formali possiedono una grammatica ufficiale e ben definita mentre per i linguaggi naturali non è così.

Si possono approssimare linguaggi naturali a linguaggi formali imponendo una sintassi rigorosa basata su uno studio delle forme di comunicazione più comuni gestendo le eccezzioni attraverso aggiornamenti alla grammatica stessa.

Ad ogni stringa valida, ovvero che appartiene al linguaggio, è associato un significato, detto **semantica**. Nei linguaggi naturali è importante anche comprendere la **pragmatica** di una stringa, ovvero il suo significato in relazione al contesto (spazio-tempo) in cui viene prodotta.

I formalismi su cui si fondano le principali grammatiche sono basati sull'idea di **struttura sintagmatica**, secondo cui le stringhe sono composte da sottostringhe, chiamate **sintagmi** e suddivise in categorie. Tale suddivisione è utile per diverse ragioni. Innanzitutto i sintagmi corrispondono ad elementi semantici naturali da cui si può ricostruire il significato di un enunciato. In secondo luogo, una tale categorizzazione aiuta a descrivere quali sono le stringhe accettabili nel linguaggio.
Consideriamo ad esempio una frase:

Il re *è nudo*

Si può evidenziare chiaramente che la parte sottolineata corrisponde a un sintagma nominale, la parte in corsivo a un sintagma verbale.
Da ciò si può generalizzare che una frase è così ottenuta:

```
< frase > = <  sintagma nominale> < sintagma verbale >
```

Questa, di fatto, rappresenta una **regola di produzione** di una grammatica. I simboli in essa presenti sono detti **simboli non terminali**. Si possono intendere come dei segnaposto che dovranno essere sostituiti con altri simboli. A partire da un **assioma**, ovvero un simbolo non terminale, e quindi astratto, si iniziano ad applicare regole di produzione per ottenere, una frase sintatticamente corretta appartenente al linguaggio. Tale processo prende il nome di **derivazione**.

Si tenga presente che non tutte le frase sintatticamente corrette hanno un senso. In altre parole la correttezza sintattica non implica la correttezza semantica.

Le regole delle grammatiche possono essere utilizzate sia per l'**analisi** (accettare/rifutare stringhe, associare alberi che descrivono la composizione di stringhe) sia per la **generazione** di stringhe appartenenti al linguaggio.

2.3 Fasi della comunicazione

La comunicazione tra due agenti è suddivisa in diverse fasi. Ciascuna si occupa di realizzare una specifica funzione, superando i problemi che si possono generare. L'obiettivo è quello di inviare un informazione o una richiesta ad un altro agente.
Consideriamo due agenti A e B. A decide di intraprendere un atto linguistico con B per comunicare la frase:

il re è nudo

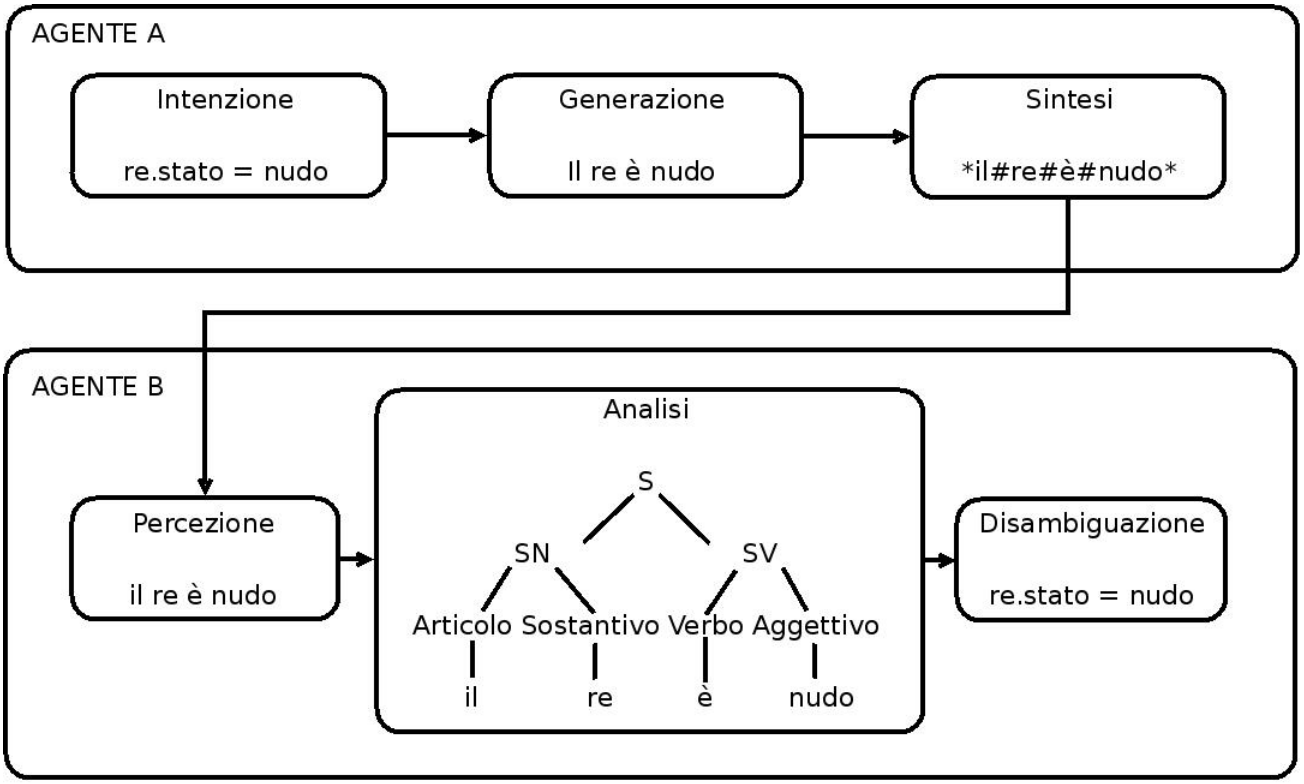


Figura 2.1: Schema di una comunicazione tra agenti

La figura 2.1 mostra schematicamente le fasi che si susseguono al fine di realizzare una comunicazione completa tra i due agenti:

1. **Intenzione**: l'agente A decide di comunicare con B.
2. **Generazione**: l'agente A trasforma l'informazione, codificata nella sua base dati in un enunciato che, una volta percepito dall'ascoltatore nella situazione corrente, con buona probabilità gli farà dedurre il significato.

3. **Sintesi**: il parlante produce la realizzazione fisica delle parole. Questo può essere fatto in svariati modi a seconda di ciò che gli attuatori consentono di fare. Ad esempio inchiostro su carta, vibrazioni nell'aria (suono), variazioni di luminosità o di tensione. Nell'esempio si aggiugono i simboli # e per segnalare il delimitarsi di una parola o di una frase.

4. **Percezione**: l'agente B, tramite i sensori, percepisce la realizzazione fisica delle parole e le decodifica. Se la comunicazione è vocale, tale passo prende il nome di **riconoscimento del parlato**, se è scritto **riconoscimento ottico**.

5. **Analisi**: l'agente B applica alla frase ottenuta la grammatica del linguaggio e procede alla costruzione dell'**albero sintattico** di cui parleremo nel paragrafo 2.4.

6. **Disambiguazione**: può capitare che alla stessa frase corrispondano più allberi sintattici diversi e quindi differenti possibili significati. In questi casi è necessario procedere a considerare gli aspetti pragmatici citati in precedenza. La scelta è guidata dalla probabilità con cui i diversi alberi possono corrispondere alla frase letta.

7. **Assimilazione**: una volta interpretato il significato di una frase, l'agente B può decidere se considerare valida o meno la comunicazione, vale a dire aggiungere o meno l'informazione alla base di conoscenza oppure eseguire o meno l'azione richiesta.

Per completezza riportiamo di seguito una grammatica tale da riconoscere la frase del linguaggio vista in precedenza:

```
< frase > = < SN > < SV > |
                    < SV > < SN > |
                    < S > <Congiunzione > < S >
< congiunzione > = e | oppure | ma ...
< SN > = < Pronome > |
                 < NomeProprio> |
                 < Sostantivo > |
                 < Articolo > |
                 < Cifra > |
                 < SN > < PP > |
                 < SN > < PropCond >
< SV > = < Verbo > |
                 < SV > < SN > |
                 < SV > < Aggettivo > |
                 < SV > < PP > |
                 < SV > < Avverbio >
< PP > = < Preposizione > < SN >
< Preposizione > = di | a | da | in | con | su | per | tra | fra
< PropCond > = che < SV >
< Pronome > = io | me | tu | te | lui | lei | egli | noi | voi | essi | loro
```

```
< Articolo > = il | lo | la | i | gli | le
< Cifra > = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
< NomeProprio > = ... | re | ...
< Aggettivo > = ... | nudo | ...
```

2.4 Analisi Sintattica

L'**analisi sintattica** (o **parsing**) è il processo atto ad analizzare uno stream continuo in input (letto per esempio da un file o una tastiera) in modo da determinare la sua struttura grammaticale.

Si può pensare al parser come a un agente software che ha per input un file di testo e produce in output un albero sintattico. La costruzione di tale albero dipende dalla grammatica che si utilizza per il riconoscimento del testo.

Il *parsing* può essere considerato un processo di ricerca di un albero sintattico. Ad ogni passo, vengono, infatti, esplorate le regole di produzione fornite dalla grammatica al fine di scegliere quella più opportuna da utilizzare. Esistono due metodologie nella scelta delle regole da utilizzare ad ogni passo di derivazione:

- **derivazione canonica sinistra**: si sceglie sempre di sviluppare il simbolo non terminale più a sinistra. Tecnica più usata perché riflette il metodo di lettura più diffuso (da destra verso sinistra).
- **derivazione canonica destra**: si sceglie sempre di sviluppare il simbolo non terminale più a destra.

Scegliere una delle due tecniche è fondamentale perché evita la creazione di alberi sintattici errati e migliora l'efficienza del parsing stesso.

Le grammatiche che si utilizzano per la rappresentazione della sintassi sono **grammatiche non contestuali** (CFG) che si differenziano dalle altre per la tipologia di regole di derivazione. Queste ultime dovranno avere la seguente caratteristica: *la parte sinistra della regola deve essere un unico simbolo non terminale e la parte destra una sequenza qualsiasi di simboli terminali e non senza vincoli sulla loro disposizione.*

Esistono due diversi approcci nella realizzazione di un parser che dipendono dal modo con cui viene costruito, durante l'analisi dell'input, l'albero sintattico:

- **top-down**: l'albero viene costruito a partire da un simbolo non terminale, assioma, e si aggiungono nodi in relazione alla parola dell'input puntata. Per ogni simbolo terminale che si andrà a trovare nelle regole di produzione si sposterà il puntatore di lettura sulla parola successiva.
- **bottom-up**: l'albero viene costruito leggendo un simbolo terminale per scegliere quale regola di produzione utilizzare per creare i nodi dal basso verso l'alto fino ad arrivare al nodo radice, l'assioma.

La realizzazione del parser è una semplice applicazione di sintesi di codice. Il suo funzionamento è intrinsecamente legato alla definizione della grammatica stessa. Esistono molti tools in grado di generare un parser dalle specifiche della grammatica.
I più comuni sono:

- antlr
- javacc
- bison
- flex
- yacc
- lex

2.4.1 Parsing Top-Down

Un parser **top-down** è un parser che costruisce l'albero sintattico a partire dalla radice. Lo si può pensare come un software di ricerca che opera nel seguente modo:

1. **stato iniziale**: si parte da un albero avente un unico nodo, l'assioma della grammatica. Lo stato è rappresentato da un albero sintattico che si evolve nel tempo.
2. **funzione successore**: sceglie, nell'albero, il nodo più a sinistra da sviluppare (ovvero un nodo corrispondente a un simbolo non terminale senza figli), quindi cerca le regole nella grammatica che hanno come parte sinistra tale nodo. Per ogni regola trovata vengono creati tanti nodi figli quanti sono gli elementi nella parte destra della regola. La scelta della regola è un problema molto importante e dipende da differenti fattori.
3. **test obiettivo**: verifica se le foglie dell'albero sintattico corrispondono esattamente alla stringa in input.

2.4.2 Parsing Bottom-Up

Un parser **bottom-up** è un parser che costruisce l'albero sintattico a partire dalle foglie per arrivare al nodo radice. Lo può pensare come un software di ricerca che opera nel seguente modo:

1. **stato iniziale**: si parte dalla lista di parole che compongono la stringa di input. Lo stato, in questo caso, è rappresentato da una lista di alberi sintattici che dovranno convergere tutti al nodo radice.
2. **funzione successore**: considera la posizione i-esima nella lista di alberi e la parte destra di ogni regola di produzione. Se la sottosequenza della lista di alberi che comincia con i corrisponde alla parte destra, viene sostituita da un nuovo albero la cui categoria è la parte sinistra della regola e i cui figli sono la sequenza stessa.

3. **test obiettivo**: verifica se le foglie dell'albero sintattico corrispondono esattamente alla stringa in input.

2.4.3 Problematiche del parsing

Esistono alcune problematiche riguardanti l'analisi sintattica che possono portare a seri problemi:

- inefficienza
- ambiguità
- ricorsione a sinistra

Efficienza del parsing

Entrambe le tipologie di parser analizzate possono risultare inefficienti a causa di molteplici modi in cui si possono combinare più analisi sintattiche di sintagmi differenti. E' possibile perdere tempo cercando in porzioni non rilevanti dello spazio di ricerca. Il parsing top-down può generare noti intermedi che non potranno mai legare con nessun simbolo terminale nella stringa di input, mentre il bottom-up può generare sottoalberi che non si potranno unire tra loro per arrivare all'assioma.

Questo aspetto appare evidente se si pensa al **backtrace**. Nel caso in cui venga provata una derivazione e si arrivasse a un simbolo terminale differente da quello puntato in input, è necessario tornare indietro e provare un altra regola di produzione.

Esiste una tecnica di parsing che si ispira alla **programmazione dinamica** che può migliorare l'efficienza dell'analisi. Il concetto è il seguente: *ogni volta che si analizza una sottostringa, si memorizzano i risultati in modo da non doverla rianalizzare in seguito.* Questa tecnica prende il nome di **chart parser**.

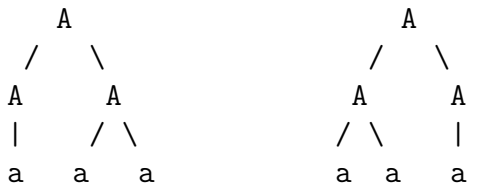
Ambiguità

Si genera un **ambiguità** quando alla stessa stringa di input corrispondono più alberi sintattici e quindi più interpretazioni semantiche diverse. Il problema non dipende dal parser ma dalla grammatica che viene utilizzata. Una grammatica è ambigua se possiede più derivazioni canoniche (destre o sinistre) per la stessa frase.

L'ambiguità di una grammatica è una **proprietà indecidibile**, pertanto non è possibile costruire un algoritmo che identifichi automaticamente se una grammatica è o no ambigua e l'eliminazione delle ambiguità dalla grammatica è un procedimento necessariamente manuale. Consideriamo il seguente esempio:

```
input: aaa
```

```
< A > = < A > < A > | a
```



Tale grammatica è ambigua. Dallo stesso input (aaa) si possono estrarre due alberi sintattici (destra e sinistra). Si può riscrivere la grammatica per renderla non ambigua e ottenere la seguente:

```
< A > = < A > < B > | < B >
< B > = a
```

oppure

```
< A > = < B > < A > | < B >
< B > = a
```

Ci sono casi in cui non si possono eliminare le ambiguità. Si utilizzano **grammatiche non contestuali probabilistiche** (PCFG) in cui si pesano con le probabilità le regole di produzione. La scelta dell'albero sintattico, quindi, è un applicazione di un problema probabilistico. E', infine, possibile modificare le probabilità attraverso tecniche di apprendimento.

Ricorsione a sinistra

Un problema tipico dei parser top-down è la ricorsione a sinistra. Questo può portare a loop infiniti dal quale il parser non potrà più uscire.
Consideriamo le seguenti regole:

1. `< A > = < A > x | y`
2. `< B > = < C >`
3. `< C > = < B > | y`

Consideriamo la prima regola: se si applica la procedura di derivazione canonica a sinistra il processo continuerà a sostituire la regola con se stessa senza uscire da questo ciclo infinito. Questa ricorsione prende il nome di **ricorsione diretta**. Le regole 2 e 3, invece, analizzate da sole non generano ricorsione infinita ma, quando la 2 viene applicata, dovrà essere applicata la 3 successivamente, che a sua volta richiamerà la 2. Questa ricorsione prende il nome di **ricorsione indiretta**.
Rimuovere le ricorsioni è facile e si può applicare un algoritmo che va a modificare le regole di produzione come segue:

```
< A > = < A > x | y

si trasforma in:

< A > = y < B >
< B > = x < B > | < NULL >
```

2.5 Analisi Semantica

L'analisi semantica ha il compito di attribuire un significato all'albero sintattico ottenuto. In altre parole deve trovare un significato degli enunciati.

I significati, ovvero i sensi, spesso sono rappresentati tramite collezioni di sinonimi o synset. Un synset definisce un concetto noto all'uomo e classificato in una gerarchia di concetti. Un concetto può essere espresso tramite una o piu parole, note come lessicalizzazioni.

Attribuendo ai vari synset i codici univoci, si possono classificare i concetti in una struttura reticolare con le relazioni, detta **ontologia**, ed arrivare a traduzione automatica che permette di passare da una lessicalizzazione ad altra.

La semantica è direttamente collegata con la rappresentazione della conoscenza, di cui parleremo nel capitolo 3. La si può vedere come una fase di traduzione volta a convertire un albero sintattico nel formalismo con cui vengono rappresentate le informazioni, obiettivi e azioni nell'agente stesso.

Spesso, l'analisi semantica contribuisce alla generazione di linguaggi attraverso l'uso di **grammatiche aumentate** o grammatica a clausole definite (DCG), che si occupano di etichettare le regole per poterle scegliere correttamente al fine di generare una frase che, oltre ad essere corretta sintatticamente, lo è anche semanticamente. Se ad esempio si volesse generare:

Io voglio il budino

si applicherebbe una struttura grammaticale simile alla seguente:

```
< pronome > < verbo > < nome >
```

Budino è chiaramente un nome e ne indica l'oggetto desiderato. Il verbo è espresso alla prima persona ed implica una volontà, quindi si sceglie voglio. Per il pronome è necessario interrogarsi sulla differenza tra Io e Me: il primo è utilizzato in caso di soggetto, il secondo in caso di oggetto. Si potrebbe scegliere di utilizzare regole aggiuntive ma ciò renderebbe molto complicata la grammatica. L'alternativa è quella di effettuare annotazioni sulle regole stesse. Riprendendo la grammatica vista in precedenza, si modificherebbe come segue:

```
< frase > = < SN(Soggetto) > < SV > | ...
< SN(Type) > = < Pronome(Type) > |
                    ...
```

```
                    < SN > < PP(Type) >
< SV > = < Verbo > |
                    < SV > < SN(Oggetto) > |
                    ...
                    < SV > < PP(Oggetto) >
< PP > = < Preposizione > < SN(Oggetto) >
< Pronome(Soggetto) > = io | tu ...
< Pronome(Oggetto) > = me | te ...
```

Un metodo alternativo è rappresentato dagli **schemi di traduzione** che rappresentano una notazione per collegare particolari azioni, sotto forma di istruzioni, in un qualche linguaggio di programmazione, alle regole di produzione di una grammatica. Quando viene scelta una regola durante l'analisi della frase, viene svolta l'azione ad essa associata. Il risultato della generazione è, quindi, il risultato dell'esecuzione di tutte le azioni combinate fra di loro nell'ordine indotto dai costrutti della grammatica.

2.6 Induzione di grammatiche

L'**induzione di grammatiche** è il processo di apprendimento di una grammatica partendo dai dati. E' un'attività che viene spontaneo tentare, visto che costruire grammatiche a mano è molto faticoso mentre sulla rete sono disponibili miliardi di enunciati esemplificativi gratuiti. Il compito è difficile poiché lo spazio delle possibili grammatiche è infinito e verificare che una data grammatica generi un particolare insieme di frasi è computazionalmente molto costoso.

Un modello interessante è quello del sistema **SEQUITUR** (Nevill-Manning e Witten, 1997). Non è richiesto alcun input tranne un singolo testo (che non dev'essere precedentemente suddiviso in frasi). Il sistema produce una grammatica in forma molto specializzata, e precisamente quella che genera una singola stringa: il testo originale. Si può dire che SEQUITUR apprende tanta grammatica quanta è strettamente necessaria ad eseguire l'analisi sintattica del testo di input.

SEQUITUR è basato sull'idea che una grammatica è buona quando è compatta. In particolare sono sempre rispettati i seguenti due vincoli:

1. Nessuna coppia di simboli adiacenti deve comparire più d' una volta nella grammatica. Se la coppia A B compare nella parte destra di più regole, la si deve sostituire con un nuovo simbolo non terminale C e aggiungere la regola C AB.

2. Ogni regola dev'essere usata almeno due volte. Se un simbolo non terminale C compare una sola volta in tutta la grammatica, si deve eliminare la regola in questione e sostituire il suo singolo uso con la parte destra.

Questi due vincoli sono applicati all'interno di una ricerca golosa che esamina l'input da sinistra a destra, costruendo incrementalmente la grammatica e imponendo il rispetto dei vincoli il più presto possibile.

Capitolo 3

Rappresentazione della Conoscenza

3.1 Ingegneria della conoscenza

Il processo generale di costruzione di una base di conoscenza prende il nome di **ingegneria della conoscenza** (knowledge engineering). Un ingegnere della conoscenza investiga un particolare dominio, impara quali concetti sono importanti e scrive una rappresentazione formale degli oggetti al suo interno e delle relazioni tra essi. In altre parole è la disciplina che si occupa di come rendere le conoscenze disponibili ai sistemi informatici.

3.1.1 Conoscenza e uso della Conoscenza

I termini *conoscenza*, *informazione* e *dato* hanno molteplici significati e non è facile tracciare un confine largamente accettato. cerchiamo di dare un significato consono a ciò che si intende realizzare. Un **dato** è un elemento atomico ricavato dall'osservazione di un certo fenomeno, mentre un informazione è il risultato di un elaborazione di più dati. Il problema ora è come identificare la conoscenza. In letteratura si distinguono due tipi di conoscenza:

- **conoscenza procedurale**: saper svolgere un operazione (parlare, scrivere, programmare)
- **conoscenza dichiarativa**: avere informazioni su eventi, relazioni, aspetti dell'ambiente esterno

La nostra trattazione rifletterà prevalentemente le conoscenze dichiarative che ci consentono di rappresentare le informazioni, da poter analizzare ed elaborare, riguardanti l'ambiente esterno. Queste ultime possono essere classificate in due categorie principali:

- **conoscenze terminologiche**: conoscenza del lessico di una lingua, sapere a che cosa corrispondono i termini
- **conoscenze nomologiche**: conoscenza di regolarità, leggi e relazioni che governano l'ambiente

Le conoscenze provengono da diverse fonti. La principale deriva dall'interazione diretta tra agente ed ambiente stesso. Tuttavia la maggior parte della conoscenza umana deriva dal **ragionamento**, che può essere deduttivo (dalle premesse alle conclusioni) o induttivo (dai fatti alle regole generali). Qualunque sia la fonte è essenziale la funzione della memoria, intesa come la capacità di conservare elementi di conoscenza nel tempo e reperirli quando servono.

Il problema principale è capire come poter rappresentare la conoscenza attraverso un linguaggio che può essere comprensibile ed elaborabile da un agente software. Quest'ultimo dovrà essere in grado di gestire le informazioni e utilizzarle tramite, procedure di ragionamento automatico, per interpretare la realtà, prevederne l'evoluzione, intraprendere azioni per modificarla ed interagire con altre entità.

3.1.2 Realtà e sua Rappresentazione

Il primo passo per capire come la conoscenza possa essere resa comprensibile alle macchine è domandarsi che rapporto c'è tra realtà e sua rappresentazione. Consideriamo un qualsiasi enunciato, ad esempio:

c'è un asino che vola

Tale frase è sintatticamente corretta ma non ha alcun significato in quanto si sa perfettamente che un asino non può volare. Ciò viene dedotto dalle caratteristiche specifiche dell'animale. La frase:

c'è un asino in cortile

potrebbe essere corretta, ma il suo significato dipende dal contesto in cui ci troviamo. Appare, quindi, evidente che un enunciato rappresenta o descrive uno *stato di cose* nel mondo del discorso. E' *vero* se ciò che descrive sussiste (l'asino è effettivamente nel cortile), è *falso* se non sussiste (l'asino è da un altra parte o non esiste). Tale definizione deriva da Aristotele ed è nota come **definizione corrispondentista della verità**.

Tale definizione assume che esistano solamente due classi di verità: vero e non vero. Purtroppo la realtà non può essere rappresentata con un assunzione binaria perché molte verità che affermiamo dipendono da punti di vista o non sono completamente vere o false. La si può approssimare se la si descrive molto dettagliatamente, cercando di identificare le eccezioni in modo da gestirle in maniera specifica. Altrimenti è necessario ricorrere a tecniche che si basano sulla *teoria della probabilità* che consentono di esprimere diversi gradi di credenze.

Entrando più nel dettaglio è necessario definire formalmente il concetto di **enunciato** . Esso è costituito da una frase in una lingua, proferita in uno specifico contesto, *contesto di enunciazione*. Quest'ultimo determina il *mittente* e il *destinatario* dell'enunciato, il *luogo* e il *tempo* dell'enunciazione e altre informazioni. Una **frase**, invece, è una sequenza di parole che rispetta la grammatica di una lingua. La differenza risiede nel contesto stesso. Un enunciato è una frase contestualizzata.

Il **mondo del discorso** è la porzione di realtà di cui si parla negli enunciati. Può essere limitata a una stanza o essere estesa a tutto l'universo, può ricoprire un periodo temporale definito o essere spalmato su un intera era o più. Il mondo del discorso non va confuso con il contesto di enunciazione. Quest'ultimo è la situazione in cui si parla, mentre il mondo del discorso è la situazione di cui si parla.

Un mondo del discorso è costituito da **stati di cose**. In prima approssimazione, quest'ultimo, è composto da determinati individui, dalle loro proprietà e dalle relazioni che sussistono tra di essi. Per **individuo**, che in latino vuol dire indivisibile, si intende qualunque entità, concreta o astratta, vivente o non vivente, animata o inanimata, che possa essere trattata come un unico elemento a cui si possa fare riferimento. In generale agli individui si fa riferimento con *sintagmi nominali*. In genere le *proprietà* di individui sono espresse da predicati con un argomento, le *relazioni* da predicati a due o più argomenti.

Infine occorre definire con precisione la **valutazione di un enunciato**. Esso è un procedimento complesso che richiede di gettare un ponte fra il piano del linguaggio e quello della realtà. Consideriamo l'enunciato:

A è un animale

è vero nel mondo del discorso, secondo la definizione corrispondentista della verità, quando siamo in grado di *identificare* A e quando *comprendiamo* il significato della proprietà essere un animale. Ovvero quando siamo in grado di gestire il problema del **symbol grounding**.

Un modo intuitivo di capire il problema è considerare il problema proposto da Van Harnad nel 1990, ovvero domandarsi se sia possibile imparare il cinese stando chiusi in una stanza e avendo a disposizione solo un dizionario cinese-cinese. Il tentativo di capire un simbolo porta solo a passare in rassegna altri simboli senza dare luce al significato del simbolo. Il dizionario lega ogni simbolo ad altri in un sistema completamente autoreferenziale e non è possibile andare al di la dei simboli, perché non ci sono legami con la realtà. Non basta quindi possedere un dizionario, occorre anche mettere in giusta relazione i termini con la realtà e con il loro significato.

Se l'enunciato diventa più complesso, vale il **principio di composizione**: il valore di verità di un enunciato complesso è funzione dei valori di verità delle sue parti. Ad esempio, consideriamo la frase:

Il panettone è grande, con le mandorle ed è sul tavolo

La frase è vera se lo sono le sue componenti:

```
X è un panettone
Y è un tavolo
X è grande
X ha le mandorle
X è su Y
```

Le prime formule derivano dalla capacità di identificare i due oggetti, panettone, tavolo, grande, mandorle. L'ultima deriva da una relazione tra i due oggetti. La validità dell'enunciato globale dipende quindi da:

- capacità di identificare gli oggetti
- capacità di identificare la relazione tra loro

E' sempre necessario tenere presente che un enunciato è vero o falso relativamente al mondo del discorso. E può essere contemporaneamente vero in un contesto e falso in un altro.

3.1.3 Fasi di Ingegnerizzazione della Conoscenza

Tra i diversi progetti di ingegneria della conoscenza ci sono grandi differenze di contenuto, dimensione e difficoltà, ma tutti includono i seguenti passi.

1. **Identificare il compito della base di conoscenza**. L'ingegnere della conoscenza deve delineare la gamma di domande a cui la *base di conoscenza*, *Knowledge Base (KB)*, dovrà rispondere e le categorie di fatti disponibili per ogni specifica istanza di problema.

2. **Raccogliere la conoscenza rilevante**. L'ingegnere della conoscenza potrebbe già essere un esperto del dominio, o potrebbe lavorare insieme ad altri esperti per estrarre quello che sanno: quest'ultimo processo prende il nome di *acquisizione della conoscenza*. In questa fase non viene utilizzata una rappresentazione formale; lo scopo è comprendere l'entità della base di conoscenza, determinata dal suo compito, e capire come funziona effettivamente il dominio.

3. **Definire un vocabolario di predicati, funzioni e costanti**. A questo punto occorre tradurre i concetti importanti del dominio in nomi di simboli logici. Nel far questo sorgono molti problemi di stile: come lo stile di programmazione, anche quello di ingegneria della conoscenza può avere un impatto significativo sul successo di un progetto. Una volta fatte queste scelte, il risultato sarà un vocabolario che prende il nome di ontologia del dominio. Con la parola **ontologia** si intende una particolare teoria riguardante la natura dell'esistenza. Un'ontologia definisce le categorie di oggetti esistenti, ma non le loro specifiche caratteristiche e le relazioni tra esse.

4. **Codificare la conoscenza generale riguardante il dominio**. L'ingegnere della conoscenza scrive gli assiomi per tutti gli elementi del vocabolario. Questo definisce precisamente (nei limiti del possibile) il significato di ogni termine permettendo all'esperto del dominio di verificare il contenuto. Spesso questo passo rivela malintesi o lacune nel vocabolario, che dovranno essere colmate tornando al passo precedente e ripetendo il processo iterativamente.

5. **Codificare una descrizione della specifica istanza del problema**. Se l'ontologia è ben definita, questo passo dovrebbe risultare semplice: si tratta di scrivere semplici formule atomiche che riguardano istanze di concetti che fanno già parte dell'ontologia. Per un agente logico le istanze dei problemi sono fornite dai sensori, e la base di conoscenza iniziale è riempita di formule aggiuntive allo stesso modo con cui i programmi tradizionali ricevono dati di input.

6. **Interrogare la procedura di inferenza e ottenere da essa risposte**. A questo punto si possono raccogliere i frutti del proprio lavoro: la procedura di inferenza, partendo dagli assiomi e dai frutti specifici del problema, sarà in grado di derivare i fatti che ci interessa conoscere.

7. **Fare il debugging della base di conoscenza**. Purtroppo le risposte saranno raramente corrette al primo tentativo. Per essere più precise le risposte saranno *quelle giuste per la base di conoscenza così com'è scritta*, presumendo che la procedura di inferenza sia corretta, ma potranno essere diverse da quelle che ci saremmo aspettati. Se mancasse un assioma (traducibile in assenza di un informazione), alcune interrogazioni non potranno avere risposta. L'assenza di assiomi può essere rilevata facilmente cercando interruzioni della catena di ragionamento.

3.2 Caratteristiche della Conoscenza

Abbiamo visto che un ontologia organizza tutti gli elementi del mondo in una gerarchia di categorie. Per realizzare un ontologia è necessario capire alcune caratteristiche della conoscenza che ci aiuteranno nell'identificare una serie di elementi utili al fine di ottenere una sua rappresentazione.

3.2.1 Classi

Una **classe** è l'insieme di tutte le entità che hanno caratteristiche comuni. Tale definizione pone l'accento sul fatto che si tratta di un *insieme*, non di una singola entità e, come tale, viene utilizzato per discriminare gli elementi ad essa appartenenti rispetto a tutti quelli che possono essere presi in considerazione.

Per poter effettuare questa discriminazione è essenziale che all'insieme sia associata una modalità che permetta di valutare l'appartenenza di un elemento generico all'insieme stesso. Questa modalità è generalmente chiamata **funzione d'appartenenza**, in quanto è pensata come una funzione che, applicata ad un elemento generico dell'universo del discorso (nel nostro caso un'entità), fornisce un grado di appartenenza di tale elemento all'insieme (nel nostro caso una classe). Quando si parla di classi, la funzione d'appartenenza è funzione delle proprietà della classe. In altri termini: tanto più un'entità ha proprietà coerenti con quelle che descrivono la classe, quanto più tale entità appartiene alla classe stessa.

Il fatto che una classe permetta di definire delle proprietà comuni ad un insieme di elementi è di importanza fondamentale per un'efficace gestione della conoscenza. In questo modo si riesce a raggruppare diverse entità in una unica (la classe appunto) e a ragionare in modo efficiente ed in termini generali su di essa.

Tramite questo concetto è anche possibile effettuare un particolare processo detto **classificazione**, cioè di ricondurre un'entità ad un certo insieme. Questo permette di aumentare la conoscenza a disposizione, in quanto, nel momento in cui si sa che un'entità appartiene ad una certa classe, si può assumere, finché non viene provato il contrario,

che quell'entità possegga tutte le proprietà che descrivono la classe, anche se, in realtà, si conoscono direttamente solo alcune di esse.

Un concetto legato alle classi è quello dell'**ereditarietà**: quando si cerca di rappresentare la conoscenza, solitamente si iniziano a classificare le entità in relazione a caratteristiche comuni. Man mano che il livello di dettaglio aumenta si creano dei *sottoinsiemi* sempre più piccoli di entità. Ciascun elemento di tale sottoinsieme ha proprietà comuni con elementi di altri sottoinsiemi ma altre differenti. Si può dire, quindi, che un elemento appartenente ad un sottoinsieme, che prende il nome di classe derivata, possieda tutte le proprietà che caratterizzano l'insieme principale, con l'aggiunta di altre peculiari a tale raggruppamento.

Consideriamo, per esempio, un cane: sappiamo con certezza che questo è un mammifero. Di conseguenza possiamo dire che possiede tutte le caratteristiche peculiari di tale insieme di esseri viventi. Un cane, però non è una scimmia. Ciò porta ad ottenere differenti classi di animali che fanno tutti parte della classe *Mammifero* ma differiscono tra loro per caratteristiche comuni. Il cane, ad esempio, ha una forma e un verso completamente differenti rispetto alle medesime caratteristiche della scimmia.

L'uso delle classi è stato integrato completamente nei linguaggi di programmazione per la semplicità di utilizzo e per le caratteristiche già elencate in precedenza.

3.2.2 Esemplari

Un **esemplare** è un'entità singola, descritta da sue proprietà caratteristiche. Di solito gli esemplari sono messi in relazione con una o più classi. Il fatto che un esemplare possa appartenere a più classi è possibile solo quando le classificazioni sono fatte secondo criteri diversi. Ad esempio, uno struzzo può essere classificato come uccello, ma anche come corridore, o come animale del deserto. Le tre classi nascono da criteri di classificazione diversi: la prima nasce da una classificazione zoologica, mentre la seconda e la terza nascono da una classificazione basata sulle caratteristiche degli esemplari (nel nostro caso caratteristiche di locomozione e di luogo di presenza).

Il concetto di esemplare serve per poter descrivere singole entità con le loro caratteristiche specifiche. Le proprietà che descrivono un esemplare possono essere le stesse che si trovano nella sua classe o in un prototipo, oppure possono essere ulteriori proprietà caratteristiche dell'esemplare specifico.

Consideriamo, ad esempio, Pippo: possiamo dire che è un cane (e quindi, è un mammifero, si nutre, respira, abbaia ecc), abita in una cuccia, ed è un amico di Topolino.

Da quanto visto fin ora emergono due problematiche riguardanti gli esemplari:

- identificazione
- numero di esemplari

Il primo problema è relativo alla presenza di più esemplari distinti dal solo nome, ma descritti tutti nello stesso modo. Questa situazione è caratteristica in carenza di informazione, quando cioè la conoscenza che permetterebbe di distinguere diversi esemplari non è disponibile. Quando si verifica una situazione di questo genere, il progettista dovrebbe

chiedersi se effettivamente ha senso avere diversi esemplari, oppure perchè ritiene che debbano esserci diversi esemplari distinti dal solo nome. Così operando si riesce solitamente a far emergere caratteristiche che permettano di rappresentare conoscenza che a sua volta permetta di distinguere i due esemplari.

Il secondo problema citato si riferisce al numero di esemplari. Nelle applicazioni, avere un alto numero di esemplari per una certa classe può essere indice di carenza di conoscenza che permetta di specificare meglio la classe. In questo caso occorre chiedersi se non sia possibile o conveniente identificare ulteriori elementi comuni ad un sottoinsieme degli elementi appartenenti alla classe, in modo da partizionare ulteriormente la classe in sottoclassi.

Nella programmazione orientata agli oggetti si identifica questo concetto con **istanza di classe**. Solitamente un istanza appartiene ad una sola classe. Questa però può essere ereditata da più classi fornendo alle sue istanze tutte le caratteristiche peculiari di ciascuna di esse. Su queste tipologie di ereditarietà possono nascere problemi tali per cui si preferisce evitare ereditarietà multiple.

3.2.3 Proprietà

Le **proprietà** permettono di descrivere le entità e differenziarle tra loro. Per quanto detto finora è evidente che il concetto di proprietà è fondamentale per la rappresentazione della conoscenza: permette non solo di descrivere entità, ma anche di creare relazioni fra di loro.

Occorre notare che, trattandosi di un concetto di cui si può parlare, la proprietà è a sua volta un'entità, descrivibile da proprietà per essa caratteristiche. Ad esempio il range di valori tipico per una proprietà, il livello di affidabilità che abbiamo circa il fatto che un esemplare abbia davvero una proprietà con un certo valore, ecc.

Tramite il concetto di proprietà, quindi, andiamo a realizzare una classe: tutte le entità che posseggono tali caratteristiche sono entità di una specifica classe. Un cane, ad esempio, ha, come un gatto e un topo, una proprietà che è l'*età*. Tale proprietà può assumere un range di valori tra 0 e 30 (valore indicato per solo scopo esemplificativo non ottenuto da alcun tipo di analisi).

Non sempre è facile identificare una classe e le sue proprietà. Lo stesso concetto può essere espresso in svariati modi utilizzando una sola classe con tantissime proprietà, oppure molte classi con poche proprietà, oppure classi che usano istanze di altre classi come proprietà. Non esiste un unica rappresentazione assoluta e universale. Tutto dipende dall'obiettivo che ci si pone e, una volta definito in maniera formale ed esplicita, dalla parte di realtà che si vuole utilizzare per tale fine.

3.2.4 Relazioni e gerarchie

Una **relazione** è un legame che coinvolge due o più entità ed è caratterizzata da una proprietà che ne identifica il rapporto. Un padre è legato a un figlio da un rapporto di parentela

Una **gerarchia** è una generica relazione di ordinamento in relazione ad una certa proprietà. Le gerarchie a cui siamo abituati a pensare (si pensi alle gerarchie di classificazio-

ne tradizionali come quelle delle Scienze Naturali), sono gerarchie sviluppate secondo le proprietà di specializzazione e generalizzazione.

Altre gerarchie possono essere definite, ad esempio, secondo la proprietà *PartOf*, che crea una relazione tra le parti di un'entità e l'entità stessa. Altra gerarchia comunemente usata è quella definita attraverso la proprietà *instanceOf*, tra esemplari e classi.

Il concetto di relazione gerarchica serve per realizzare il meccanismo dell'eredità già visto in precedenza.

In generale, un'entità può essere in relazione gerarchica con diverse altre entità. Di solito, nel meccanismo di eredità, è adottata una politica in cui le proprietà definite ai livelli inferiori della gerarchia hanno la meglio in caso di conflitto, rispetto alle analoghe proprietà definite ai livelli superiori. Questo meccanismo si chiama sovrascrittura (**overriding**) dei valori e viene attuato quando le proprietà considerate non sono quelle caratteristiche, ma quelle tipiche delle classi.

3.3 Sistemi di ragionamento per categorie

Le categorie sono gli elementi principali per la costruzione di schemi di rappresentazione su grande scala. Questo paragrafo descrive i sistemi esplicitamente progettati per organizzare e ragionare sulle categorie. Ci sono due categorie di famiglie di sistemi, strettamente imparentate:

- **reti semantiche**: permettono di visualizzare graficamente una base di conoscenza e forniscono algoritmi efficienti per inferire le proprietà di un oggetto sulla base della sua appartenenza a una categoria
- **logiche descrittive**: rappresentano un linguaggio formale per costruire e combinare definizioni di categorie e forniscono algoritmi efficienti per determinare le relazioni di sottoinsieme e superinsieme tra categorie.

In questo testo ci soffermeremo sulla trattazione delle prime.

3.3.1 Reti semantiche

Nel 1909 Charles Peirce propose una notazione grafica basata su nodi e archi che chiamò **grafi esistenziali** e definì la logica del futuro. Cominciò cosi un lungo dibattito tra i paladini della logica e quelli delle reti semantiche. Sfortunatamente, le diatribe nascosero il fatto che le reti, almeno quelle la cui semantica è ben definita, sono una formula di logica. Per certi tipi di formule la notazione offerta dalle reti semantiche è spesso più comoda, ma se non consideriamo le questioni di interfacciamento con gli esseri umani, i concetti sottostanti (oggetti, relazioni, quantificazioni ecc.) sono identici.

Ad una prima approssimazione, le reti semantiche sono dei formalismi per rappresentare la conoscenza aventi una struttura a grafo. Solitamente i nodi rappresentano oggetti, concetti, stati, mentre gli archi, eventualmente etichettati, rappresentano le relazioni tra

i nodi. Esse condividono l'idea di base che strutture complesse possano essere descritte come una collezione di attributi, e valori associati, e che il ragionamento è, nella sua forma più astratta, una realizzazione di collegamenti, la cui giustificazione è riconducibile al buon senso: esperienza, somiglianza, dipendenza, tipicità.

Sono state spesso proposte come modello della memoria umana, largamente utilizzate per la comprensione del linguaggio naturale e sono attualmente considerate un tipo comune di dizionario leggibile da una macchina.

Ci sono molte varianti di reti semantiche, ma tutte sono capaci di rappresentare oggetti singoli, categorie di oggetti e relazioni. Una notazione grafica tipica rappresenta i nomi degli oggetti e delle categorie racchiusi in ovali o rettangoli, collegati con archi etichettati.

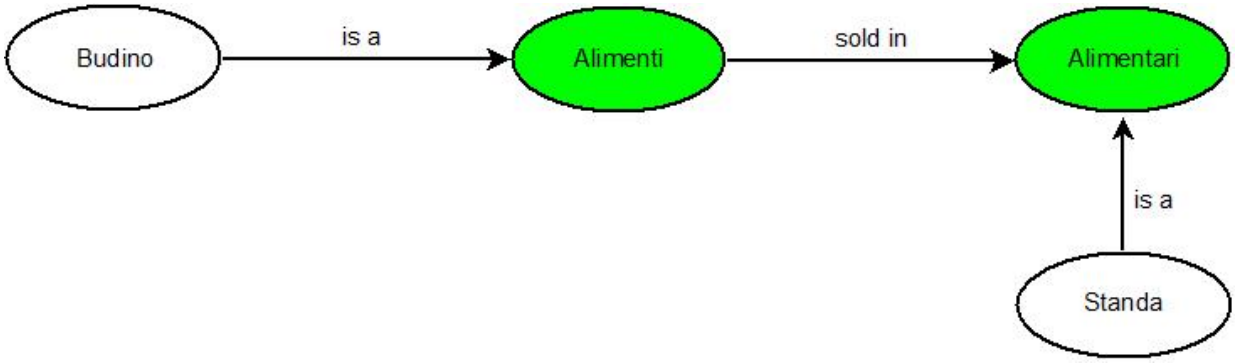


Figura 3.1: Semplice rete semantica

La figura 3.1 mostra un semplice esempio di rete semantica ove il concetto di *budino* è legato alla categoria *alimenti* da una relazione di *is a*, vale a dire il *budino* è un *alimento*. La categoria *alimenti*, a sua volta, è legata alla categoria *alimentari* da una relazione di *sold in*. Ciò implica che se si trova il concetto di *budino* si può dire che è un *alimento* ed è venduto negli *alimentari*. La *Standa*, legata alla categoria *Alimentari* da un link *is a* probabilmente vende tale oggetto.

La notazione delle reti semantiche rende molto semplice ragionare sull'**ereditarietà delle proprietà**. Consideriamo, per esempio, un *Cavallo*: questo è un animale e, come tale, condivide alcune caratteristiche con una scimmia e un delfino, ma ne ha alcune di proprie che ne identificano univocamente l'appartenenza a questa categoria. L'eredità diventa complicata quando un oggetto può appartenere a più di una categoria, o quando una categoria può essere un sottoinsieme di più di un'altra categoria; in questo caso si parla di ereditarietà multipla. Ora un algoritmo di ricerca potrebbe trovare , in risposta a una interrogazione, due o più valori in conflitto. Per questa ragione alcuni linguaggi di programmazione orientati agli oggetti (OOP), come Java, proibiscono l'uso dell'eredità multipla nella gerarchia delle classi. Nelle reti semantiche, comunque, essa è generalmente consentita.

Consideriamo, ad esempio, un *cavallo alato*. E' chiaro che dovrà avere i caratteri pecu-

liari degli uccelli ma anche dei cavalli. Ma tutte e due le categorie derivano direttamente da *Animali*. Ciò porta ad una rappresentazione simile a quella presentata in figura 3.2.

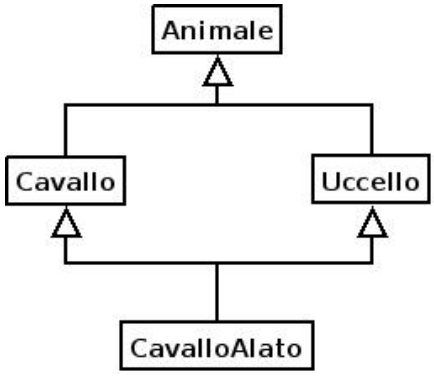


Figura 3.2: Schema UML dell'albero di eredità di un Cavallo Alato

Uno degli aspetti più importanti è la capacità di rappresentare **valori di default** per le categorie. Consideriamo un individuo, *Polifemo*, che ha un solo occhio. Solitamente tutte le persone posseggono 2 occhi. In una KB strettamente logica, questa sarebbe una contraddizione. In una rete semantica, tuttavia, l'asserzione che tutte le persone hanno 2 occhi è solo un valore di default. Questo significa che si presume sempre che le persone abbiano sempre 2 occhi a meno che ciò non sia contraddetto da informazione più specifica. La semantica dei valori di default è supportata in modo naturale dall'algoritmo di ereditarietà, perchè quest'ultimo segue i link verso l'alto partendo dall'oggetto (*Polifemo* in questo caso) e fermandosi non appena trova un valore. Si dice che il valore di default è sovrascritto (*overridden*) dal valore più specifico. Notate che è possibile, anche, sovrascrivere il numero di occhi di default creando una categoria di *PersoneConUnOcchio*, un sottoinsieme di Persone di cui *Polifemo* sarebbe membro. Per mantenere la stretta correttezza logica della rete, possiamo dire che l'asserzione *Occhi* delle *Persone* include un'eccezione nel caso di *Polifemo*. Per una rete di struttura prefissata questa semantica è accettabile, ma nel caso ci siano molte eccezioni le espressioni logiche saranno molto meno concise della notazione grafica. Nel caso che la rete debba essere aggiornata con nuove asserzioni, quest'approccio fallirà completamente: dovremmo dire che anche tutte le persone (presentemente sconosciute) con un occhio in meno costituiscono un'eccezione alla regola.

Una limitazione delle reti semantiche è data dal fatto che i collegamenti tra le bolle possono rappresentare solo relazioni binarie. Tale restrizione ha come conseguenza la creazione di una ricca ontologia di concetti reificati. La *reificazione*, ovvero il procedimento di creazione di un modello di dati basato su un concetto astratto predefinito, delle proposizioni rende possibile rappresentare ogni formula atomica della logica del primo ordine, priva di variabili e di funzioni, con la notazione delle reti. Alcuni tipi di formule universalmente quantificate possono essere espresse usando collegamenti inversi e con frecce etichettate con rettangoli singoli e doppi applicati alle categorie. Questo tuttavia ci lascia ben lontani dalla potenza espressiva della logica del primo ordine completa. Tra le altre cose non è

possibile esprimere la negazione, la disgiunzione, le funzioni annidate e la quantificazione esistenziale. E possibile estendere la notazione per renderla del tutto equivalente alla logica del primo ordine, come hanno fatto gli stessi grafi esistenziali di Peirce o le reti semantiche partizionate di Hendrix (1975), questo però elimina uno dei vantaggi principali delle reti semantiche: la semplicità e la trasparenza dei processi di inferenza.
I progettisti possono costruire una grande rete e continuare ad avere una buona idea di quali interrogazione risulteranno efficienti, perchè:

- è facile visualizzare i passi della procedura di inferenza
- in alcuni casi il linguaggio di query è cosi semplice che non si possono proprio esprimere interrogazioni complesse.

Nei casi in cui la potenza espressiva sia troppo limitata, per colmare le lacune molte reti semantiche sfruttano il cosiddetto meccanismo del collegamento procedurale (*procedural attachment*). Questa tecnica fa si che una query (e talvolta anche una asserzione) che riguarda una certa relazione abbia come risultato la chiamata di una procedura speciale appositamente progettata, senza utilizzare l'algoritmo generale di inferenza.

Parte II

Implementazione del Sistema

Capitolo 4

Parser Testuale

4.1 Riconoscimento Vocale

Il **riconoscimento vocale** è il processo mediante il quale il linguaggio orale umano viene riconosciuto e successivamente elaborato attraverso un computer o più specificatamente attraverso un apposito sistema di riconoscimento vocale.

Sistemi di riconoscimento vocale vengono utilizzati per applicazioni vocali automatizzate nel contesto delle applicazioni telefoniche, ad esempio call center automatici, per sistemi di dettatura (in inglese dictation systems), che consentono di dettare discorsi al computer, oppure per sistemi di controllo del sistema di navigazione satellitare o del telefono in auto tramite comandi vocali.

Molte aziende di software e gruppi di ricerca hanno affrontato questo problema fornendo soluzioni valide, sia proprietarie che gratuite. Questi programmi funzionano con algoritmi adattativi di tipo statistico inferenziale, che ricostruiscono il linguaggio in base alla frequenza delle associazioni fra parole. Alcuni possiedono anche un vocabolario fonetico con le pronuncie base di numerose parole nella lingua selezionata. Per ottenere un migliore riconoscimento del parlato è necessaria una procedura di addestramento che consente al sistema di adattarsi alle peculiarità fonetiche del parlatore. Tale procedura consiste nella lettura da parte dell'utente di un brano davanti al microfono, per abituare il programma a riconoscere la voce, che viene registrata e analizzata per costruire una libreria di file vocali.

Nel progetto presentato non ci si è posta molta attenzione a questo aspetto perché le soluzioni presenti sul mercato sono valide e funzionano a dovere. La scelta di quale software di riconoscimento vocale utilizzare dipende dalle necessità del momento e non è vincolante per il corretto funzionamento del sistema globale.

4.2 Linea guida

Il parser testuale è l'oggetto che si occupa della prima fase dell'analisi, ovvero l'**analisi sintattica**. Tale operazione è suddivisa nei seguenti passi:

- identificazione delle singole stringhe

- identificazione del verbo
- identificazione del soggetto dell'azione
- identificazione dell'oggetto dell'azione (non sempre presente)

Alla base del parser si è definita una specifica grammatica che si basa sul seguente modello:

```
< frase > = < SN > < SV > |
                    < SV > < SN > |
                    < S > <Congiunzione > < S >
< congiunzione > = e | oppure | ma ...
< SN > = < Pronome > |
                 < NomeProprio> |
                 < Sostantivo > |
                 < Articolo > |
                 < Cifra > |
                 < SN > < PP > |
                 < SN > < PropCond >
< SV > = < Verbo > |
                 < SV > < SN > |
                 < SV > < Aggettivo > |
                 < SV > < PP > |
                 < SV > < Avverbio >
< PP > = < Preposizione > < SN >
< Preposizione > = di | a | da | in | con | su | per | tra | fra
< PropCond > = che < SV >
< Pronome > = io | me | mi | tu | te | ti | lui | gli | le | lei | egli |
                 noi | ci | voi | vi | essi | loro
< Articolo > = il | lo | la | i | gli | le
< Cifra > = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
< NomeProprio > = ....
< Aggettivo > = ...
```

Come si può notare, tale grammatica dovrebbe essere in grado di riconoscere frasi del linguaggio italiano. Il problema di applicarla è legato, però, alla sua natura ambigua e a problemi di efficienza nell'analisi stessa. Un linguaggio parlato, infine, non è un linguaggio che segue una sintassi rigorosa e formale: spesso vengono espresse frasi in cui non compare un verbo o non sia indicato neppure un soggetto ma utili al fine di identificare una necessità dell'utente.

Frasi del tipo *che fame* non rispecchiano alcuna costruzione derivabile dalla grammatica mostrata in precedenza ma è utilissima al fine di identificare un bisogno specifico dell'utente.

Il parser si deve, quindi, occupare di identificare alcuni elementi fondamentali dai quali si potrebbe dedurre necessità o bisogni. Per fare questo si è scelto un'implementazione alternativa che non segue i modelli visti in precedenza nei paragrafi 2.4.1 e 2.4.2. Alla sua base vi è, comunque, la grammatica elencata in precedenza, ma si cerca di non vincolare l'utente a generare frasi ben formate.

Si può, quindi, affermare che la grammatica non è *la* regola di riconoscimento ma *una* linea guida. Le frasi vengono riconosciute anche se non sono propriamente conformi alla sintassi ad essa associata.
Il risultato che si ottiene è rappresentato in figura 4.1:

Figura 4.1: Schema del parser

Come si può notare il parser è un oggetto che prende frasi in ingresso, le processa e da esse estrae un oggetto che contiene una lista di oggetti corrispondenti alle stringhe generate in ordine di lettura, il soggetto, il verbo ed, eventualmente, l'oggetto dell'azione. Tale rappresentazione è in contrapposizione all'albero sintattico generato da parser top-down e bottom-up.
La figura 4.2 ne mostra un esempio.
Tale oggetto è l'output fornito dal parser sintattico in corrispondenza della frase:

Io ho finito la vernice rossa

Come si può osservare, la frase ha un soggetto esplicito (*Io*), un verbo (*ho finito*) e un oggetto (*vernice*), quest'ultimo arricchito da un aggettivo (*rossa*). Al fine di una corretta analisi semantica sono necessari prevalentemente i primi tre elementi, identificati e isolati dal resto della frase. Il fatto che l'attributo soggetto e oggetto siano liste è dovuta al fatto che una frase può avere più soggetti e più oggetti come ad esempio:

Immanuel e Checilia sono colleghi
mi serve il pane e la carne

In entrambe le frasi abbiamo un soggetto multiplo (*Immanuel e Checilia* nella prima e *pane e carne* nella seconda) legati dalla congiunzione *e*.

Tuttavia, viene raccolta in una lista l'intera sequenza di stringhe in ordine di produzione. Ciò viene fatto per poter analizzare eventuali elementi aggiuntivi, nell'esempio riportato

l'aggettivo *rossa*, in grado di aumentare il contenuto informativo dell'azione stessa. Se non venissero considerati, il sistema non sarebbe in grado di essere specifico. Ciò porta a un maggiore grado di dettaglio e di precisione riguardo al supporto fornito all'utente. Se non si identificasse, nell'esempio, che è finita la vernice rossa, l'utente non saprebbe che colore acquistare una volta giunto al negozio più vicino.

Grazie a questo oggetto è possibile semplificare il riconoscimento di necessità o bisogni dell'utente tramite l'analizzatore semantico che verrà analizzato nel capitolo 5.

4.3 Pacchetti e Classi Java

Per capire il funzionamento del parser è necessario, innanzitutto, mostrare la struttura delle classi e le strutture dati utilizzate. Per semplicità si è scelto di realizzare tre diversi pacchetti nei quali sono raccolte le varie classi sulla base della loro utilità.
I pacchetti sono:

- **types**
- **databases**
- **parser**

4.3.1 Pacchetto types

All'interno di questo pacchetto sono definite le classi che corrispondono ai tipi di simboli che compaiono nel linguaggio, in questo caso adattati alla lingua italiana. Ciascuna classe rappresenta un *simbolo non terminale* della grammatica vista in precedenza. Si noti il forte

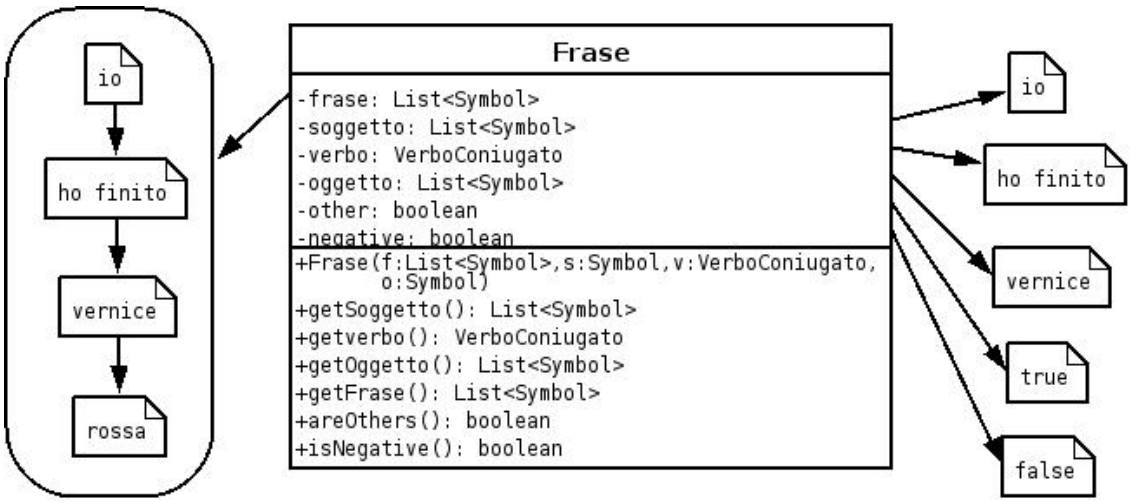


Figura 4.2: Schema UML della classe Frase per la frase *io ho finito la vernice rossa*

legame tra grammatica e classi: ogni volta che viene analizzato un simbolo in ingresso, rappresentato da un oggetto della classe String (fornita dal linguaggio Java) il parser cerca di identificare l'oggetto ad esso associato. Vale a dire cerca di capire se l'oggetto appena letto è una parola, un verbo, un aggettivo ecc. Sulla base di tale identificazione il parser può scegliere la regola di produzione opportuna da applicare e, di conseguenza, ciò influenza il riconoscimento della frase.

Le classi contenute in questo pacchetto non sono altro che una forma di rappresentazione dei *simboli non terminali* della grammatica, mentre le funzioni fornite dalla classe Parser si occuperanno di applicare le opportune regole grammaticali per l'identificazione di soggetto, verbo e oggetto. Gli oggetti di classe String letti, sono i *simboli terminali*.

Il fatto di utilizzare oggetti specifici per ogni tipologia di simbolo consente di evidenziare, tramite valori ai dati membro della classe, caratteristiche e proprietà peculiari che aiutano il parser nel riconoscimento e ricostruzione della sintassi della frase. Un verbo, ad esempio, possiede la caratteristica di essere coniugato in un certo tempo e una certa persona. Sulla base di questi dati si è in grado di identificare, all'interno della frase chi o quali potrebbero essere i soggetti dell'azione da esso rappresentata.

La figura 4.3 mostra l'intero schema UML del pacchetto types, sottolineando l'albero di derivazione delle classi. Da questa rappresentazione è omessa la derivazione implicita dalla classe Object perché priva di contenuto informativo.

Classe Symbol

La classe principale è la classe **Symbol** che contiene una stringa corrispondente a ciò che viene letto. Tale classe dispone di un *costruttore*, una funzione *equals* che si occupa di verificare se due oggetti sono identici e una funzione **getType()** che ritorna il codice corrispondente alla tipologia di oggetto di cui si ha l'istanza.

Il costruttore accetta come argomenti:

- String s: stringa corrispondente alla parola letta
- int t: tipo di oggetto

Il tipo di oggetto è un intero che corrisponde a una costante fornita dalla classe *Type*.

Classe Type

Per rendere più parametrizzabile possibile il codice si è introdotta una classe, **Type**, che contiene delle costanti, una per ogni tipologia di oggetto che il parser riesce a riconoscere. questi sono:

- parola = 1;
- verbo = 2;
- pronome = 3;

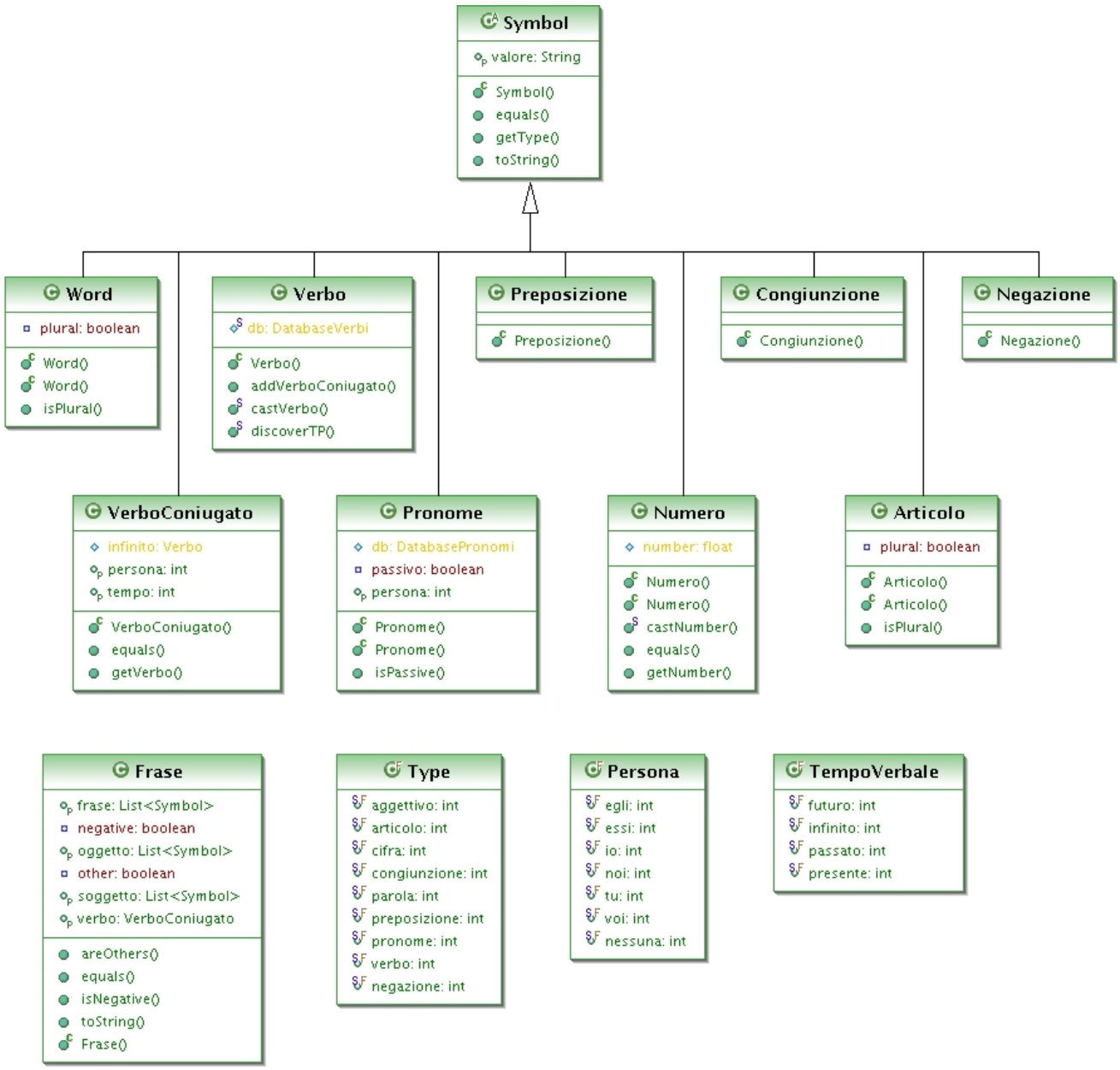


Figura 4.3: Schema UML del pacchetto types

- articolo = 4;
- cifra = 5;
- aggettivo = 6;
- preposizione = 7;
- congiunzione = 8;
- negazione = 9;

L'utilizzo di tali costanti aiuta, durante l'esecuzione del codice, a capire di quale simbolo si tratti evitando di avere problemi derivati all'ereditarietà. Maneggiando puntatori alla classe base, l'invocazione del metodo *instanceof* può generare ambiguità dovute al fatto che una parola (*Word*) è sempre un Symbol come lo è un aggettivo (*Aggettivo*). L'uso di getClass e il conseguente controllo su un oggetto di tipo Class eviterebbe simili problemi ma computazionalmente è molto oneroso.

Classe Persona

Come visto in precedenza per la classe *Type*, per parametrizzare pronomi e dare una congruenza con i verbi, si è scelto di usare una classe, **Persona**, che si occupa di definire delle costanti da condividere in ogni parte del codice:

- nessuna = 0;
- io = 1;
- tu = 2;
- egli = 3;
- noi = 4;
- voi = 5;
- essi = 6;

Da notare la prima costante: *nessuna* che indica, appunto, nessuna persona. Il suo utilizzo è dedicato a casi in cui i le coniugazioni verbali non possiedano una persona. Si considerino, ad esempio, i tempi *infinito*, *participio* e *gerundio*.

Classe TempoVerbale

Anche per i tempi dei verbi è stata adottata una scelta di parametrizzazione. Questa volta il range non ricopre tutti i possibili tempi verbali ma esprime la seguente classificazione:

- infinito=0;
- presente = 1;
- passato = 2;
- futuro = 3;

Per scelta i verbi espressi hanno solo 3 tempi, infinito escluso. Tali tempi rappresentano le forme prevalenti di collocamento temporale di un azione. In italiano, come anche in inglese e in francese, i tempi sono molti di più e vengono usati per esprimere relazioni tra le varie frasi oppure per indicare azioni in corso ma si possono approssimare tutte a uno di questi tre tempi. La perdita di informazione non è rilevante al fine di identificare i bisogni dell'utente.

Un altro vantaggio riguarda il supporto multilingua. Non tutti i linguaggi esprimono i tempi verbali allo stesso modo ma tutti intendono riferire di fatti collocati al presente, al passato o al futuro. Ciò porterebbe a modificare solo alcuni elementi, specifici della classe Parser nell'individuazione del tempo e non implicherebbe sconvolgimenti di codice in altre parti.

Classe Word

La classe **Word** (parola) si occupa di rappresentare tutte quelle parole corrispondenti a oggetti e persone. Rispetto alla classe base Symbol prevede l'aggiunta di un valore booleano che indica se tale oggetto è singolare o plurale. Di default l'oggetto prevede il singolare. Il parser, come vedremo più avanti, decide che la stringa corrisponente alla parola sia plurale in relazione a tre condizioni:

- sia preceduta da un articolo plurale
- sia preceduta da un numero maggiore di 1
- sia esplicitamente plurale in relazione alla ricerca su un portale web

La funzione **isPlural()**, infine, si occupa di informare se si tratti di un istanza corrispondente a una parola plurale o singolare.

Classe Articolo

La classe **Articolo** si occupa di rappresentare gli articoli, distinguendoli tra singolari e plurali. Tale distinzione è necessaria per identificare le parole che questi precedono. Il funzionamento è analogo a quello della classe *Word*.

Classe Congiunzione

La classe **Congiunzione** si occupa di gestire parole tipo *e*, *oppure* che legano parole o frasi tra loro. E' usata nel Parser per identificare soggetti o oggetti multipli. Attualmente è supportata solo la congiunzione o disgiunzione di parole o aggettivi, non riesce a legare più frasi tra loro.

Classe Negazione

La classe **Negazione** si occupa di rappresentare tutti quei simboli che negano un azione e danno un accezione negativa alla frase. Se all'interno della frase compare una negazione viene impostato a true un apposito campo all'interno dell'oggetto Frase. In questo modo l'analizzatore semantico sarà in grado di identificare l'accezione dell'azione espressa dal verbo in essa contenuto.

Classe Preposizione

La classe **Preposizione** si occupa di gestire le preposizioni, in italiano i termini *di, a, da, in, con, su, per, tra, fra.* Tali preposizioni hanno il compito di legare gli elementi tra loro e aggiungere precisione nell'identificare oggetti, posizioni e azioni stesse. Vengono usate dall'analizzatore semantico al fine di poter effettuare ricerche più mirate.
Si consideri la frase:

```
Mi serve il vaso di ceramica
Ho voglia di gelato
```

La preposizione *di* in entrambi le frasi aggiunge dettaglio all'oggetto del desiderio/necessita'.

Classe Pronome

La classe **Pronome** si occupa di gestire i pronomi. Tale classe è necessaria per il riconoscimento dei soggetti/oggetti all'interno della frase a partire dal verbo. Ogni pronome è associato a una Persona tramite un dato membro intero che richiama una costante definita dalla classe *Persona*.

Ogni oggetto della classe *Persona* ha un riferimento a un oggetto statico di tipo *DatabasePronomi*. Questo serve per raccogliere i pronomi in una mappa indicizzabile tramite la persona che essi rappresentano. E' usata per ricavare il soggetto corretto dal verbo in frasi in cui non è esplicitato.
Si consideri, ad esempio, la seguente frase:

ho finito il latte

In tale frase il soggetto non è presente ma lo si può dedurre dal verbo che, essendo coniugato alla prima persona singolare, si può dedurre essere *io*.

Ogni volta che un oggetto di tipo *Pronome* viene costruito, viene creata un entry opportuna nel database dei Pronomi che, essendo pensato come singleton, si istanzia non

appena viene invocato per la prima volta il suo metodo *getInstance()*, altrimenti la funzione restituirebbe un riferimento all'oggetto già allocato.

Infine si ponga l'attenzione sul dato membro *passivo* di tipo boolean: tale attributo riflette la proprietà di alcune forme di pronomi di subire una specifica azione. Si pensi alla frase:

io sono un nerd
e
ci piace la musica di Johannes Brahms

La prima frase indica che *io* è un pronome soggetto che compie l'azione di *essere nerd*, nel secondo caso il soggetto è la *musica* che piace a chi produce la frase. In questo caso il pronome *ci* subisce l'azione. La frase la si può capire meglio se in forma esplicita:

La musica di Immanuel Casto piace a noi

Entrambi le frasi indicano lo stesso concetto. In questo caso, però, è evidente che il soggetto è la musica e non chi produce la frase.

Classe VerboConiugato

La classe **VerboConiugato** si occupa di rappresentare le coniugazioni dei verbi. Oggetti di tale istanza, derivando dalla classe Symbol possiedono un dato membro di classe String che identifica il termine (ad esempio *ho mangiato*) con l'aggiunta di due dati membro:

- int tempo
- int persona
- Verbo infinito

Il primo assume valori in relazione alle costanti definite dalla classe *TempoVerbale*, il secondo in relazione a quelli fornite dalla classe *Persona*. L'ultimo dato membro identifica l'appartenenza della forma coniugata a un ramo di coniugazioni a partire da una forma infinita. *Ho mangiato* appartiene al verbo *mangiare*.

Vengono forniti i metodi di get per ottenere tali valori: *getTempo()*, *getPersona()* e *getVerbo()*. Essi sono usati dal parser e dall'analizzatore semantico per capire e identificare il senso della frase e/o i possibili soggetti/oggetti dell'azione descritta dal verbo.

Il costruttore, unico, accetta come argomenti: la stringa corrispondente alla coniugazione, il tempo, la persona e un riferimento all'oggetto di tipo *Verbo* a cui tale coniugazione appartiene. Per capire come vengono ricavati questi dati si guardi la descrizione della classe *Verbo*.

Classe Verbo

La classe **Verbo** si occupa di rappresentare le forme infinite dei verbi. L'idea alla base è quella di avere un unico oggetto che descriva il verbo e un insieme di oggetti di tipo *VerboConiugato* che descrivano le coniugazioni nelle varie persone e tempi. In altre parole, il verbo *mangiare* possiede un istanza di classe *Verbo* corrispondente all'infinito e una per *io mangio*, *tu mangi* ecc. di tipo *VerboConiugato*.

Questa classe, derivando direttamente da Symbol, possiede una stringa che identifica la forma infinita del verbo. A ciò viene aggiunto un riferimento a un oggetto di tipo *DatabaseVerbi* che si occupa di raccogliere le varie coniugazioni dei verbi. Il parser è in grado di identificare tempo e persona dei verbi regolari grazie a un algoritmo che studia le terminazioni. Per i verbi irregolari, tale procedura è impossibile e diviene fondamentale possedere un database con le coniugazioni.

Abbiamo visto prima, nel *VerboConiugato*, che, per istanziarne un oggetto, è necessario sapere tempo e persona in cui la forma verbale è coniugata. La classe Verbo, tramite la funzione **discoverTP(String s)**, si occupa proprio di effettuare tale operazione. Il funzionamento è molto semplice: si verifica se la stringa passata per argomento termina con un postfisso appartenente a una particolare coniugazione. L'appartenenza identifica (quasi) univocamente tempo e persona della forma coniugata analizzata. Il valore di ritorno è un array contenente i seguenti valori:

1. prefisso: parte immutata del verbo comune a tutte le coniugazioni
2. tempo: codice corrispondente ai valori definiti dalla classe *TempoVerbale*
3. persona: codice corrispondente ai valori definiti dalla classe *Persona*

Si consideri, per esempio, la seguente stringa:

```
stringa: mangio

prefisso: mangi
coniugazione: are
tempo: presente
persona: prima singolare
```

Questa funzione è usata dalla funzione statica **castVerbo(String s)**. L'idea è quella di avere una funzione, indipendente dalle istanze, che verifichi se una stringa è una forma coniugata di un verbo. Il funzionamento è molto semplice e riassumibile nei seguenti passi:

1. verifica se la stringa è contenuta nel database dei verbi. In caso affermativo ritorna l'oggetto corrispondente.
2. verifica se la stringa termina in *are*, *ere*, *ire* (forme infinite). In caso affermativo viene creato un oggetto Verbo e viene ritornata la forma corrispondente di istanza VerboConiugato

3. verifica se la stringa può essere un verbo richiamando la funzione *discoverTP(String s)*. Nel caso in cui si individua persona e tempo viene creato un oggetto di tipo VerboConiugato corrispondente alla stringa rilevata e viene ritornata

4. se nessuna delle precedenti verifiche ha avuto successo, viene ritornato null. Ciò vuol dire che la stringa non è un verbo riconosciuto.

Classe Numero

La classe **Numero** si occupa di rappresentare i numeri, intesi in virgola mobile. Una loro rappresentazione in forma di stringa non garantisce una efficenza in fase di elaborazione (causa problemi di cast e rallentamenti). Alla classe *Symbol* viene aggiunto un dato membro di tipo float che contenga il valore corrispondente.

Anche in questo caso, come nella classe *Verbo* esiste una funzione di cast opportuna: **castNumber(String s)**. Questo metodo prende la stringa e, tramite la funzione **Float.parseFloat(String s)** si prova a convertirla in un numero float. In caso di fallimento (vale a dire che si tratti di una vera e propria stringa alfanumerica) viene raccolta l'eccezzione e ritornato null.

Classe Frase

Come già visto in precedenza, la classe **Frase** raccoglie il risultato dell'analisi sintattica, evidenziando il soggetto, l'oggetto e il verbo della frase analizzata. Tale classe non deriva da Symbol.
Un qualsiasi oggetto della classe *Frase*, possiede i seguenti dati membro:

- List<Symbol> frase: contiene l'intera lista di simboli che costituiscono la frase, priva di articoli.
- List<Symbol> soggetto: contiene la lista di soggetti che compiono l'azione espressa dalla frase.
- VerboConiugato verbo: contiene il verbo coniugato che esprime l'azione che si vuole rappresentare con la frase.
- List<Symbol> oggetto: contiene la lista di elementi che subiscono l'azione espressa.
- boolean other: indica se ci sono altri elementi nella frase che non sono stati identificati come soggetti oppure oggetti.
- boolean negative: indica se la frase è espressa con accezione negativa (mancanza).

Grazie all'utilizzo della variabile booleana other e della possibilità di avere la frase originaria completa, l'analizzatore semantico è in grado di identificare elementi aggiuntivi che caratterizzerebbero meglio e aggiungerebbero più dettaglio alla frase stessa. Ciò porta a identificare con maggiore precisione eventuali bisogni o mancanze dell'utente, supportandolo con maggiore precisione nelle fasi decisionali.

4.3.2 Pacchetto databases

Il pacchetto **databases** fornisce le classi che si occupano di gestire i database e, più in particolare, quello relativo ai *Verbi* e ai *Pronomi*. I database *DatabaseRelazioni* e *DatabaseRicerche* verranno usati per l'analisi semantica e saranno analizzati nel capitolo 5 mentre quello *DatabaseLuoghi* verrà usato per l'interrogazione al *GIS* e sarà trattato nel capitolo 6. Grazie a tali database è possibile identificare le stringhe lette e semplificare il riconoscimento testuale. In figura 4.4 è possibile vedere lo schema UML delle classi appartenenti al pacchetto. Tutti gli oggetti istanziati sulla base delle classi contenute in tali pacchetti sono

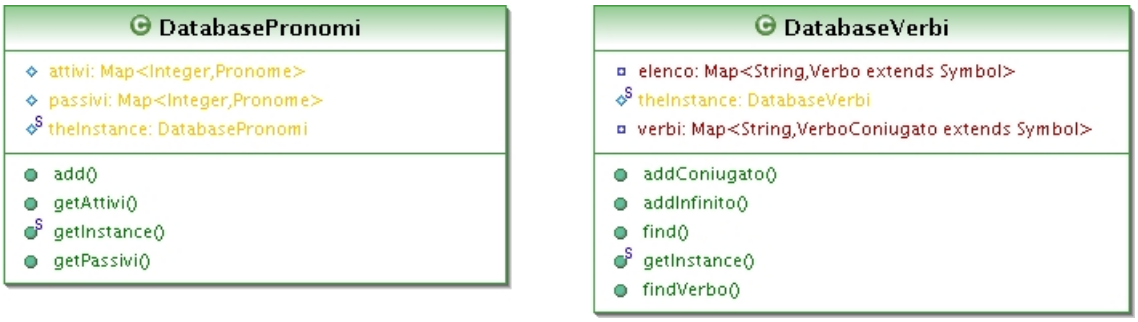


Figura 4.4: Schema UML del pacchetto databases

dei **singleton**, ovvero un oggetto che viene istanziato la prima volta che viene richiesto e da quel momento resta disponibile per chi ne fa richiesta.

La seguente classe fornisce un modello per tutti i database, istanziati come singleton, che il software richiede:

```
public class Database
{
    private static Database theInstance = null;

    // Data Members
    private Database() {}

    public static Database getInstance()
    {
        if (theInstance == null)
        {
            theInstance = new Database();
        }
        return theInstance;
    }

    // Functions Members
```

```
}
```

Come si può notare la classe possiede un dato membro statico, quindi comune a tutte le istanze, appartenente alla classe stessa. In questo modo per ogni riferimento di classe, esiste un unico oggetto vero e proprio sul quale vengono effettuate le operazioni. Il costruttore è privato, vale a dire che non può essere invocato se non tramite funzioni appartenenti alla classe stessa.

Per ottenere il riferimento al database vero e proprio è necessario, quindi, richiamare la funzione **getInstance()**. Ciò è possibile perché la funzione è statica e quindi, oltre a poter elaborare dati membri statici (comuni a tutte le classi) non richiede un oggetto sul quale invocarli ma bensi basta conoscere il nome della classe. Si veda l'esempio seguente:

```
Database d = Database.getInstace();
d.method();
```

La funzione, appena viene invocata, verifica se è già stato istanziato l'oggetto statico **theInstance**. In caso affermativo, ne ritorna un riferimento. In caso negativo, vale a dire la prima volta che un componente ne richiede l'utilizzo, lo istanzia e ne ritorna il riferimento.

Classe DatabasePronomi

La classe **DatabasePronomi** fornisce un sistema per immagazzinare e consultare i pronomi memorizzati e riconosciuti dal sistema. I dati membro sono i seguenti:

- Map<Integer, Pronome> attivi
- Map<Integer, Pronome> passivi

La differenza è dovuta al diverso modo di concepire il pronome nel caso di attivo, soggetto dell'azione, oppure passivo, oggetto dell'azione. Tutti i pronomi vengono automaticamente inseriti in tale database già nel costruttore. L'inserimento avviene grazie alla funzione **add(Pronome p)** che verifica se il pronome è da inserire nella prima o nella seconda lista.

I pronomi sono inseriti nella mappa secondo una chiave che corrisponde al tag assegnato ai pronomi per indicarne la persona. In questo modo la ricerca può avvenire proprio attraverso questo parametro getAttivi(int i), per ottenere il pronome attivo corrispondente, e getPassivi(int i) per ottenere il corrispondente pronome passivo.

Questa rappresentazione risulterà molto più chiara in seguito. L'utilizzo è reso necessario in quei casi in cui la frase non presenta un soggetto esplicito:

ho fame

Tale frase, infatti, non ha un soggetto ma è chiaro che chi ha fame è colui che parla, quindi *io* è soggetto.

Classe DatabaseVerbi

La classe **DatabaseVerbi** si occupa di organizzare e gestire i verbi riconosciuti e le loro coniugazioni. E' anch'essa un singleton e presenta i seguenti dati membri:

- Map<String, Verbo> elenco
- Map<String, VerboConiugato> verbi

La mappa **elenco** si occupa di raccogliere l'elenco di tutti i verbi, espressi in forma infinita da oggetti di classe *Verbo*. La chiave di inserimento e ricerca è il prefisso del verbo, la parte di stringa che rimane immutata in tutte le coniugazioni di verbi regolari. Si consideri il seguente esempio:

```
forma infinita: amare
prefisso: am
desinenza: are

esempi di coniugazione
io amo: (am + o)
ho amato: (am + ato)
```

La mappa **verbi** associa la stringa corrispondente alla coniugazione a un oggetto di tipo VerboConiugato che lo rappresenta.

L'idea è quella di avere un unico oggetto che contenga tutti i verbi e tutte le coniugazioni e l'uso delle mappe aiuta a ridurre la complessità computazionale dell'algoritmo di ricerca.

La funzione **addInfinito(Verbo v)** si occupa di aggiungere un nuovo verbo in forma infinita nella lista. E' richiamata dal costruttore della classe Verbo.

La funzione **addConiugato(String s, VerboConiugato v)** aggiunge una forma coniugata di un verbo e la sua stringa.

La funzione **find(String s, boolean plural)** restituisce un oggetto di tipo *VerboConiugato* se la stringa passata come argomento corrisponde a una forma coniugata già vista in precedenza. I database vengono popolati ogni volta che viene istanziato un oggetto corrispondente. La loro funzione è quella di evitare l'allocamento di ulteriori oggetti per l'elaborazione semantica e velocizzare il riconoscimento.

Infine, la funzione **findVerbo(String s)** verifica se è presente un verbo, in forma infinita, corrispondente alla parte di prefisso fisso per tutte le coniugazioni (es: amare = am + are).

4.4 Funzionamento del Parser

Per capire il funzionamento del parser è necessario iniziare con una descrizione molto generale per poi aumentare sempre di più il livello di dettaglio. La figura 4.5 mostra lo schema UML della classe **Parser**
Analizziamo, per prima cosa, i dati membro:

- Scanner read: si occupa di processare l'input per estrarre le stringhe provenienti dal parser vocale.
- LinkedHashMap<String, Symbol> dictionary: è un dizionario di termini conosciuti. Consente di associare un oggetto *String* (letto in input) ad un oggetto derivato da *Symbol* corrispondente. I verbi compaiono solo in forma infinita.
- DatabasePronomi pronomi: è un riferimento al singleton che si occupa di gestire i pronomi4.3.2)
- DatabaseVerbi dbverbi: è un riferimento al singleton che si occupa di gestire i verbi (4.3.2)
- Verbo avere: è un riferimento a un oggetto Verbo corrispondente al verbo ausiliare *avere*
- Verbo essere: è un riferimento a un oggetto Verbo corrispondente al verbo ausiliare *essere*
- List<Symbol> frase: è una lista di simboli corrispondente alla frase che si sta leggendo e analizzando.

Passiamo ora all'analisi dei metodi:

- Costruttore
- Frase nextFrase(): si occupa di leggere un intera frase e restituire un oggetto di tipo *Frase*
- Symbol nextSymbol(boolean plural): si occupa di leggere un simbolo dall'oggetto *Scanner*, identificarne il tipo corretto e inserire nella lista *frase* l'oggetto derivato da *Symbol* corrispondente.

Figura 4.5: Schema UML della classe Parser

- Symbol find(String s, boolean plural): si occupa di interrogare opportuni sistemi informativi che si occupano di catalogare i simboli per associarli a un corrispondente oggetto derivato da *Symbol*.
- Frase generateFrase(): si occupa di analizzare la lista di simboli *frase* per identificare i soggetti, il verbo e gli oggetti della frase, isolando anche gli aggettivi ad essi associati.

Per comprendere meglio il funzionamento del parser è necessario analizzare più approfonditamente le funzioni principali.

4.4.1 Costruttore

Il costruttore del *Parser* non ha il compito di inizializzare i dati membro, i quali vengono inizializzati in automatico in fase di creazione della classe per richiesta esplicita nella loro definizione, ma si occupa di leggere il file di testo con i dettagli per istanziare alcuni oggetti derivati da *Symbol* per la configurazione iniziale ed ottenere una migliore efficienza. I dati necessari sono:

- Completa coniugazione dei verbi ausiliari (*essere* ed *avere*)
- Elenco di tutti i Pronomi, sia attivi che passivi
- Termini di negazione (*non*) e di congiunzione (*e*)
- Articoli, sottolineando quelli singolari e plurali
- qualsiasi altro termine utile ad evitare la ricerca su web

4.4.2 Funzione nextFrase()

Questa funzione si occupa di generare un oggetto di tipo *Frase* in corrispondenza della lettura di una serie di stringhe. Il sistema è in grado di delimitare le singole frasi attraverso dei delimitatori inviati dal parser vocale.

Il funzionamento è molto semplice: ogni stringa letta viene identificata ed inserita in una lista (*List<Symbol> frase*) fino a quando non viene notificato dal parser vocale la fine di una possibile frase. Ciò si verifica nei seguenti casi:

- pausa significativa nella generazione di parole
- cambio di parlatore

Appena viene ricevuto tale messaggio, implementato attraverso un simbolo di delimitazione (#) viene eseguita la funzione *generateFrase()* che processa la lista di stringhe più volte al fine di isolare gli elementi principali utili all'analisi semantica: soggetto, verbo, oggetto dell'azione.
Per ogni simbolo letto attraverso l'oggetto *Scanner*, vengono eseguiti diversi passi:

1. viene identificato l'oggetto Symbol appartenente a tale stringa invocando la funzione *nextSymbol(boolean plural)*
2. si verifica se il simbolo letto corrisponde a un evento di fine programma o di fine frase:
   - fine programma: si ritorna con un valore *null*
   - fine frase: si esegue la funzione *generateFrase()* e si ritorna l'oggetto *Frase* da essa generato.
3. se non si è individuata una condizione sopra citata si procede con l'analisi del simbolo trovato prestando attenzione alla sua singolarità o pluralità
   - se l'oggetto è un articolo viene verificato se anticipa una parola singolare o plurale e si procede al simbolo successivo senza aggiungerlo alla lista. Ciò viene fatto perché l'articolo non è una componente importante ai fini dell'analisi ma può aiutare a capire se ciò che lo segue è una parola singolare o plurale.
   - se l'oggetto è un numero vale il discorso precedente e la pluralità viene calcolata sulla base del valore del numero. Una cifra maggiore di uno indica che la parola successiva è plurale. In questo caso l'oggetto viene aggiunto perché la quantità è un elemento ad alto contenuto informativo.
   - tutte le altre tipologie di oggetti vengono aggiunti alla lista di elaborazione *List<Symbol> frase*).

Nel caso in cui si identifica una condizione che induce a pensare che il simbolo successivo che verrà letto è plurale (articolo plurale o numero maggiore di uno) viene memorizzata tale informazione in una variabile booleana che, una volta letto il simbolo successivo, verrà passata come argomento alla funzione *nextSymbol(boolean plural)*. In questo modo è possibile riconoscere se un simbolo mai letto, e quindi non presente nel dizionario, è singolare o plurale. Una volta consumata la variabile il suo valore viene rimesso a *false*.
Si consideri ora la seguente frase:

Io ho mangiato i pasticcini
Vorrei 3 fragole

La prima frase ha un soggetto, *io*, un verbo di prima persona al tempo passato, *ho mangiato* e un oggetto, i pasticcini. Il sistema deve poter essere in grado di capire che *pasticcini* è plurale. Supponendo di non aver alcun elemento distintivo non si potrebbe generalizzare limitandosi a verificarne la terminazione. Si consideri la parola *dito*, il suo plurale non è *diti*, ma *dita*. Se si analizzasse solo la terminazione in *i* ed *e* per il plurale si considererebbe *dita* come un termine singolare. Considerare plurale le parole che finiscono in *a*, invece, sarebbe un errore perché tutti i termini femminili sarebbero riconosciuti come plurali. *Palla*, *foglia* e molte altre parole, pur essendo singolari, sarebbero classificate diversamente. L'analisi dell'articolo è precisa: *un dito* indica, inequivocabilmente che il termine *dito* è singolare,

le dita indica che *dita* è plurale. Stesso discorso per *la foglia* e *le foglie*. Classificare gli articoli non genera ambiguità perché il loro uso è esclusivo: o sono singolari o plurali.

Per i numeri, seconda frase, il ragionamento è analogo. Si pensi a *1 dito* e *3 dita*. Il fatto che il termine *dita* è plurale è determinato dal numero 3 da esso preceduto.

Capire se un termine è plurale o singolare è un elemento fondamentale per la ricerca di un soggetto o di un oggetto in relazione alla persona con cui è coniugato il verbo principale della frase letta.

Questa funzione, insieme al costruttore, è l'unico metodo ad essere pubblico. Tale scelta è dovuta al fatto che le altre funzioni si occupano di analizzare simboli o di generare una frase ma dall'esterno, e quindi dall'analizzatore semantico, non devono essere accessibili perché operano su variabili locali e sono chiamate solo in particolari condizioni che dall'esterno dell'oggetto *Parser* non si possono verificare.

4.4.3 Funzione nextSymbol(boolean plural)

La funzione *nextSymbol(boolean plural)* si occupa di estrarre una stringa da uno stream di input, utilizzando l'oggetto *input* della classe *Scanner* e ritornare l'oggetto derivato dalla classe *Symbol* corrispondente. La sua funzione, in pratica, è quella di identificare a quale elemento sintattico del linguaggio appartiene la stringa letta (articolo, verbo, nome ecc). Il funzionamento si basa sulla seguente procedura:

1. ricerca nel dizionario
2. elaborazione della stringa

Nel dizionario sono contenuti tutti gli oggetti derivati da Symbol che il sistema conosce o ha imparato. E' implementato attraverso una *LinkedHashMap*, ovvero un evoluzione di una *HashMap*. L'idea di base è quella di avere una collezione di dati efficiente nella procedura di inserimento e ricerca in cui i record inseriti non possono comparire più volte. L'indicizzazione, e quindi la ricerca, viene fatta sulla base di un oggetto che prende il nome di *chiave* (di cui viene calcolato l'*hashcode*). In media il costo computazionale della ricerca o dell'inserimento di un elemento in tale struttura dati è lineare e dipende dalla quantità di elementi nella lista e dalla complessità della procedura di calcolo dell'hashcode della chiave. Nel nostro caso si è scelto di usare la stringa come chiave e l'oggetto di classe *Symbol* ad essa associato come valore.

Nel caso in cui la procedura di ricerca termina con un oggetto diverso da null, vuol dire che la stringa è associata a un elemento già riscontrato in precedenza o di cui il *Parser* ha una conoscenza congenita. Il ritorno, pertanto, sarà immediato e assumerà il valore dell'oggetto estratto dalla collezione.

In caso contrario, invece, si procede alla sua identificazione utilizzando metodi statici di cast definite per alcune classi del pacchetto *type*. L'ordine con cui vengono fatti i test si basa su questo ordine:

1. verbi

2. numeri

3. parole. nomi propri e aggettivi

Tutti gli altri simboli (Articoli, Congiunzioni e Pronomi) devono essere già inseriti nel dizionario. Ciò viene fatto attraverso una fase di addestramento automatico effettuato dal costruttore tramite la lettura di un particolare file di testo in cui sono rappresentati i sintagmi e le loro proprietà.

4.4.4 Riconoscimento verbi

La prima tipologia di elementi che vengono identificati sono i verbi. per fare ciò si esegue la funzione *castVerbo(String s*, metodo statico della classe *Verbo*, sulla stringa letta. Se il verbo è regolare oppure è già presente nel database il verbo viene identificato subito. In caso contrario la funzione termina.

Se il verbo letto appartiene ad una coniugazione di *essere* o *avere* viene salvato l'oggetto VerboConiugato appena trovato in una variabile temporanea e viene estratta dall'oggetto *input* della classe *scanner* una nuova stringa. e viene riapplicata la procedura di identificazione. Se anch'essa risulta essere un verbo e, in particolare, coniugato in gerundio o participio, allora vuol dire che il verbo precedente non era il verbo della frase ma svolgeva una funzione ausiliaria per ciò che lo seguiva. Grazie a questo accorgimento è possibile individuare coniugazioni appartenenti a tempi composti e le forme passive.
Si considerino i seguenti esempi:

- avrò mangiato
- ho mangiato
- è finito

La prima coniugazione appartiene al verbo *mangiare*, è di prima persona singolare (*io*) e il tempo è futuro. Queste ultime due informazioni sono ricavate dall'analisi del verbo ausiliario. Si tenga presente che l'ausiliario non indica con precisione il tempo. Se si studia la seconda frase, l'analisi del verbo avere indicherebbe che si tratta di tempo presente. Una frase del tipo:

Ieri ho mangiato la ratatouille

vuole descrivere un azione svolta in passato. E' sintatticamente e semanticamente corretta ma occorrerebbe un analisi semantica per identificarne la congruenza. In caso di forme composte, se il verbo ausiliario risulta essere, quindi, di tempo presente, viene forzato ad essere inserito come passato. Ciò è perfettamente congruente con quanto detto fin ora.

La terza frase è espressa in forma passiva. Viene anch'essa riconosciuta come verbo composto, nel caso corrente appartenente alla coniugazione di *finire* e di tempo passato. Nel monento esatto in cui si pronuncia tale verbo, l'azione o l'evento di cui si parla ha già subito l'azione di essere terminato.

Nel caso in cui il secondo elemento non risultasse essere un verbo, allora, il verbo salvato nella variabile temporale viene aggiunto alla lista di simboli identificati (*List<Symbol> frase*) e, per l'oggetto corrente, si prosegue con le fasi succesisve di riconoscimento.

In caso di riconoscimento positivo di una forma complessa, viene ritornato un nuovo oggetto di tipo *VerboConiugato* che corrisponde al verbo completo di ausiliario. Tale verbo viene aggiunto nel database dei verbi attraverso la funzione *addConiugato(String s, int t, int p)* eseguita sull'oggetto *Verbo* corrispondente alla coniugazione trovata.

Si tenga presente che i verbi sono presenti nel dizionario del parser solo nella sua forma infinita. La ricerca della coniugazione viene fatta nella funzione *castVerbo(String s)*, metodo statico della classe Verbo.

4.4.5 Riconoscimento numeri

Una volta esclusa la possibilità apparente che la stringa possa essere un verbo, il parser si occupera di verificare se la stringa è un numero. Per fare questo si eseguirà il metodo statico *castNumber(String s)* della classe Number che si occupa di eseguire il cast della stringa ad un numero di tipo float. Se il parser vocale ha generato un simbolo numerico (es: 42) tale oggetto verrà riconosciuto e verrà restituita un istanza di Number ad esso associato. Nel caso in cui il parser restituisce un numero in forma alfabetica (es: quarantadue), la funzione non sarà in grado di rilevare tale identificativo. Sarà comunque possibile classificare tale stringa come numero grazie alla funzione find(String s, boolean plural) di cui discuteremo in seguito.

4.4.6 Riconoscimento parole, nomi propri e aggettivi

Il riconoscimento di parole, nomi propri e aggettivi è complesso e non può essere attraverso algoritmi specifici che elaborano la parola stessa. A priori, infatti, non è possibile dire che *Giuseppe* è un nome proprio, un aggettivo o il nome di un oggetto. Lo si può capire se al sistema viene riferito. La fase di apprendimento iniziale sarà pertanto fondamentale: definendo un dizionario iniziale di simboli è possibile riuscire ad avere un buon livello di riconoscimento che dipende dalla dimensione iniziale del database.

Inizialmente si è pensato di classificare parole, nomi propri e aggettivi come unici oggetti. Tale approssimazione può andare bene nel caso di un prototipo operante in un ambiente statico con frasi molto semplici e richieste, da parte dell'utente, molto precise e ben formate. In casi più generici e complessi ciò risulterebbe un problema notevole.
Si pensi alla seguente frase:

Io voglio il tappeto volante

Volante è un aggettivo che qualifica il tappeto. Se si generalizzasse convergendo aggettivi e parole in un unico elemento, il parser e, di conseguenza, l'analizzatore sintattico potrebbero pensare che il soggetto (*io*) voglia due oggetti distinti: *tappeto* e *volante*. In questo caso l'ultimo termine è anche un oggetto vero e proprio. Se si fosse detto al sistema che *volante* è un aggettivo che qualifica *tappeto*, il soggetto sarebbe uno ed univoco.

Pertanto, è fondamentale una conoscenza pregressa o disporre di un meccanisco di aggiornamento e addestramento. La funzione *find(String s, boolean plural)* è pensata proprio per questo scopo.

4.4.7 Funzione find(boolean plural)

Il metodo find si occupa di interrogare un database, identificare a cosa corrisponde un particolare elemento ed istanziare l'oggetto. ad esso associato utilizzando un motore di ricerca opportuno. Nel prototipo si è scelto di utillizzare *Coniuga.com*.

Una volta identificato il codice associato alla tipologia di elemento viene selezionato il ramo corrispondente dell'istruzione *switch* e viene ritornato il corretto oggetto derivato da Symbol al quale la stringa è associata.

Si noti che questa procedura non si applica ai verbi che dispongono autonomamente di un meccanismo di interrogazione attraverso la procedura vista in precedenza nel paragrafo 4.3.1.

4.4.8 Funzione generateFrase()

La funzione *generateFrase()* si occupa di analizzare il contenuto della lista che contiene gli oggetti derivati da Symbol fin ora letti (List<Symbol> frase) per identificare verbo, soggetti e oggetti al fine di ottenere un oggetto di tipo *Frase*. In questa fase assume un ruolo fondamentale la grammatica: sulla base di essa, infatti, vengono ricercati gli elementi della frase in relazione alla posizione del verbo.

Questa operazione può essere classificata come una prima e rudimentale analisi semantica. Per semplicità viene svolta dal parser e, quindi, viene trattata in questo capitolo.

La fase di riconoscimento è progettata per lavorare al fine di identificare i soggetti e gli oggetti a partire dal verbo sulla base del loro posizionamento nella frase rispetto a quest'ultimo. Al fine di un ottimo riconoscimento il parser implementa le seguenti funzionalittà:

- riconoscimento di soggetti e oggetti multipli, separati da congiunzioni
- associazione di qualificativi (aggettivi in particolare) agli elementi rilevati, gestiti attraverso collezioni di dati di tipo lista
- identificazione dell'accezione, positiva o negativa, di una frase
- aggiunta di soggetto in caso di sua assenza

La procedura di generazione viene eseguita attraverso diverse fasi:

1. riconoscimento del verbo
2. identificazione di pronomi
3. identificazione di elementi mancanti

4. aggiunta del soggetto sottointeso

Riconoscimento del verbo

Viene scorsa la lista dei simboli letti (List<Symbol> frase) alla ricerca di un oggetto di tipo *VerboConiugato*. In questa fase, viene spezzata la lista in due sottoliste per identificare gli elementi che si trovano prima e dopo il verbo. Questa operazione è fondamentale per la ricerca del soggetto.

Una volta trovato il verbo si identifica il tempo e, in particolare, la persona. Da un analisi di questa informazione si può andare a ricercare il soggetto dell'azione, se esiste, oppure ipotizzarlo. Se non dovesse essere presente alcun verbo la frase viene ritornata. In caso contrario si procede alla fase successiva.

Identificazione di pronomi

Una volta trovato il verbo si procede alla rianalisi della frase al fine di identificare pronomi. Solitamente i pronomi possono essere soggetti o oggetti di una frase. Per ogni pronome rilevato vengono considerati tre casi differenti:

1. la persona del verbo corrisponde alla persona del pronome e il pronome non è in forma passiva: si è trovato un soggetto dell'azione e lo si aggiunge alla lista dei soggetti. Si pensi alla frase:

   Io ho fame

2. se il verbo ha, come soggetto, una persona plurale e il pronome rilevato non è passivo: si aggiunge momentaneamente il soggetto alla lista e si procede all'analisi del simbolo successivo. Se è presente una congiunzione seguita da un pronome o un nome proprio, allora essi sono soggetti, altrimenti il pronome trovato viene rimosso dalla lista di soggetti. Ogni volta che viene trovata una congiunzione, la procedura viene iterata. Si pensi alle seguenti frasi:

   Io e te abbiamo vinto
   Io e Luca siamo stanchi

3. se non è vera una delle due condizioni precedenti e il pronome è passivo allora non si tratta di soggetto ma di oggetto. Ogni pronome verrà aggiunto alla lista degli oggetti tenendo conto della presenza di altri possibili oggetti uniti da una congiunzione. In questo caso non ci sono relazioni con la persona del verbo perché l'oggetto subisce l'azione e il verbo è coniugato in relazione al soggetto. Si pensi alla frase:

   Mi piace la Ferrari

   In questo caso l'azione la compie *la Ferrari* che piace a chi formula la frase. *Mi* è un pronome classificato come passivo, ovvero che subisce l'azione.

Identificazione di Elementi Mancanti

Terminata la fase precedente, potrebbero non essere stati identificati soggetti e oggetti dell'azione. In questo caso ci sarebbero degli elementi ancora presenti nella lista di simboli letti. Si ricordi che ad ogni identificazione, il simbolo viene rimosso.

Pertanto, se la lista contiene ancora simboli sono possibili due scenari:

1. assenza di soggetto
2. assenza di oggetto

Si consideri il primo caso: se fino al passo corrente il soggetto non è stato identificato può voler dire due cose:

- non è presente: allora si vedrà in seguito come risolvere questo aspetto
- il soggetto è di terza persona (singolare o plurale) ed è costituito da un termine che non è un pronome.

In quest'ultima condizione, si esamina una sottolista di frase, quella in cui potrebbero essere presenti i soggetti. Siccome sono state identificate due sottoliste, una contenente i termini prima del verbo, e una contenente quelli posti dopo, è necessario scegliere in quale delle due si andrà a ricercare i soggetti. Per questa scelta assume un ruolo determinante la presenza di un pronome attivo o passivo. Si considerino le seguenti frasi:

Marco ha fame
Ci piace la nutella

La prima frase presenta un soggetto nella prima parte. Il soggetto è un nome proprio di persona. La seconda frase presenta un soggetto nella seconda parte della frase e l'oggetto è rappresentato dal *Ci*, pronome di prima persona singolare. Da ciò si può dedurre che se è presente un pronome passivo, si ricerca il soggetto nella seconda parte della frase e l'oggetto nella prima, altrimenti l'esatto contrario.

Anche in questa fase di analisi è presente la classica analisi del simbolo successivo, alla ricerca di congiunzione o qualificativo per migliorare la precisione dell'analisi sintattica.

Nel secondo caso, ovvero assenza di oggetto, la procedura è analoga a quella vista in precedenza. L'unica differenza risiede nelle sottoliste esaminate. Sono l'opposto del caso precedente.

Aggiunta del Soggetto Sottointeso

Una volta che le analisi precedenti sono terminate, si verifica che il soggetto sia effettivamente presente o meno. In caso non lo sia, è possibile che sia stato omesso. E' tipico del linguaggio italiano, infatti, omettere alcuni pronomi. Si consideri la frase:

Ho fame

Tale affermazione è priva di soggetto esplicito ma è deducibile dal verbo. Si può quindi calcolare, a partire dalla persona, prima nel caso dell'esempio, il soggetto e aggiungerlo. In questo caso la frase corretta sarebbe:

Io ho fame

L'aggiunta di *Io* è possibile attraverso una funzione della classe *DatabasePronomi*, *getAttivi(int persona)*, che, specificata una persona tramite una costante fornita dalla classe *Persona*, ritorna un oggetto di tipo *Pronome*. Il risultato è quello di ottenere un pronome corrispondente alla persona in cui è coniugato il verbo.

4.5 Conclusioni

Il parser così pensato è in grado di riconoscere verbi regolari in maniera automatica, conoscendo la sua forma infinita, e quelli irregolari, attraverso l'interrogazione di un database su web.

Per la prima fase di analisi semantica è possibile identificare soggetti e oggetti multipli ed eventuali aggettivi o termini qualificativi ma non si è ancora in grado di identificare, questi ultimi grazie ad un opportuna elaborazione delle preposizioni.

Infine sono implementate opportunamente le forme congiuntive ma non si è ancora in grado di analizzare frasi multiple legate da essi. Si possono però spezzare le frasi e renderle indipendenti: nel momento in cui vengono riconosciuti due verbi coniugati, le frasi vengono spezzate in relazione alla presenza di una congiunzione ed analizzate separatamente. C'è la possibilità che alcune volte non vengano spezzate in modo opportuno per la difficile distinzioni tra la congiunzione di termini e quella di frasi.

L'ultimo problema è legato alla necessità di definire preposizioni, congiunzioni e articoli tramite file di testo. E' possibile che tale riconoscimento sia svolto tramite la funzione *find(boolena plural)*. Grazie a quest'ultima è, infine, possibile riconoscere articoli e nomi propri. La precisione del riconoscimento dipende da quale database su web si è scelto di usare.

Capitolo 5

Analizzatore Semantico

5.1 Linea Guida

L'obiettivo dell'**analizzatore semantico** è quello di leggere un oggetto di tipo *Frase* in input e tradurlo in una chiave di ricerca da inserire nel *DatabaseRicerche*. In questa fase sarà fondamentale definire un opportuna *ontologia* al fine di poter correlare le parole presenti nelle frase. Nella tesi non ci siamo dedicati molto a definire l'ontologia perché è un lavoro di semplice classificazione della realtà e di inserimento all'interno di un database. La parte più importante del lavoro è stata dedicata a realizzare una struttura di classi e strutture dati che consentano di porre in relazione tra loro le istanze con le categorie a cui potrebbero corrispondere e le categorie tra loro.

Figura 5.1: Schema dell'Analizzatore Semantico

La figura 5.1 mostra quanto detto tramite uno schema chiarificatore. Come si può osservare, l'oggetto di tipo *Frase* viene elaborato al fine di generare delle chiavi di ricerca per comporre la query di interrogazione al *GIS*.

Per capire meglio quanto detto è necessario considerare un esempio. Supponiamo che un utente in un determinato momento della giornata pronunci la frase:

Io ho fame

Il *Parser* si occuperà di generare un oggetto *Frase* ad esso associato isolando il soggetto, *io*, il verbo, *ho* e l'oggetto, *fame*. A questo punto l'analizzatore semantico inizia l'analisi verificando cosa vuol dire *avere fame*. Interrogando il suo database, scoprirà che la parola *fame* è una condizione di un utente ed è in relazione con una categoria, *Alimenti*. Questa relazione è dovuta al fatto che se uno ha fame, ha bisogno di alimenti. Gli *Alimenti*, però, non costituiscono un entità ricercabile su un *GIS*. Sarà necessario quindi verificare la categoria *Alimenti* a quali altre categorie è collegata. Sicuramente esisterà un link che la unisce alla categoria *Alimentari*, ovvero negozi che vendono *Alimenti*. Quest'ultima è rintracciabile in un *GIS*. Verrà quindi generata una chiave di ricerca corrispondente alla categoria appena rilevata.

Per il corretto funzionamento del sistema è necessario definire un'ontologia che tenga conto di tutte le possibili relazioni, sia che legano le istanze alle categorie, sia quelle tra le categorie. Completata questa fase sarà necessario fare una mappatura tra le categorie appena individuate e quelle utilizzate dal *GIS* per rappresentare luoghi ed entità. Considerando che quest'ultimo avrà una ontologia molto più semplice rispetto a quella definita per un analisi semantica di un linguaggio parlato, si può dire che solo un sottoinsieme delle categorie pensate sarà riconosciuto dal *GIS*. La ricerca avrà senso eseguirla solo se esiste una relazione, diretta o indiretta, che lega il concetto estratto dall'analisi della frase pronunciata dall'utente con una categoria appartenente a tale sottoinsieme. In caso contrario non si otterrebbero risultati validi.

Nel nostro caso ci siamo limitati a definire una semplice ontologia, realizzata dalle classi presenti nel pacchetto *knowledge_members* da utilizzare nei test di validazione del prototipo. Maggiormente sarà precisa e consistente, maggiore potenza analitica avrà il sistema.

5.2 Tipologie di Relazioni

Prima di procedere con l'analisi dei pacchetti Java sviluppati, è importante definire cosa si intende con relazione e quali sono le sue tipologie. Una **relazione** è un legame che unisce due entità sulla base di un rapporto di:

- somiglianza
- derivazione/deduzione

Tornando al caso del *budino* è possibile dire che, essendo un *alimento*, è collegato alla categoria *Alimentari* perché è li che lo si può trovare. Tale affermazizone si basa su un semplice ragionamento deduttivo:

1. il *budino* è un *alimento*
2. gli *alimentari* vendono *alimenti*
3. il *budino* lo si vende negli *alimentari*

Una volta interiorizzato il concetto di relazione è possibile identificarne due tipologie:

- relazione tra istanza e categoria
- relazione tra categoria e categoria

Nel caso di *budino* si può parlare di relazione tra categoria e categoria (*Alimenti* con *Alimentari*) perché facendo parte della categoria *Alimenti*, il budino eredita tutte le relazioni in cui tale categoria è coinvolta.

Consideriamo *condizioni* del nostro corpo: *essere affamati* ed *essere malati*. Entrambe sono istanze della classe *Condizione* ma sono in relazione a categorie diverse: se una persona ha fame, probabilmente è interessata a sapere quali sono i negozi di alimentari vicini e non necessariamente le farmacie, a cui è legata la seconda condizione citata.

5.3 Pacchetti e Classi Java

Per capire il funzionamento dell'analizzatore semantico è necessario, innanzitutto, mostrare la struttura delle classi e le strutture dati utilizzate. Per semplicità si è scelto di realizzare tre diversi pacchetti nei quali sono raccolte le varie classi sulla base della loro utilità.
I pacchetti sono:

- **databases**
- **knowledge_members**
- **semantica**

5.3.1 Pacchetto knowledge_members

Questo pacchetto definisce una banale ontologia realizzata con lo scopo di ottenere un prototipo del sistema funzionante. In figura 5.2 si può vedere l'albero di eredità delle classi che rappresentano le categorie individuate.
Ci dedicheremo ad analizzare solo tre classi:

- KnowledgeElement
- Alimenti
- Alimentari

Si è scelto di approfondire solo queste tre perché la prima rappresenta la classe base da cui tutte le altre classi derivano, la seconda una generica classe rappresentante una categoria, la terza una categoria che è presente anche nel *GIS*.

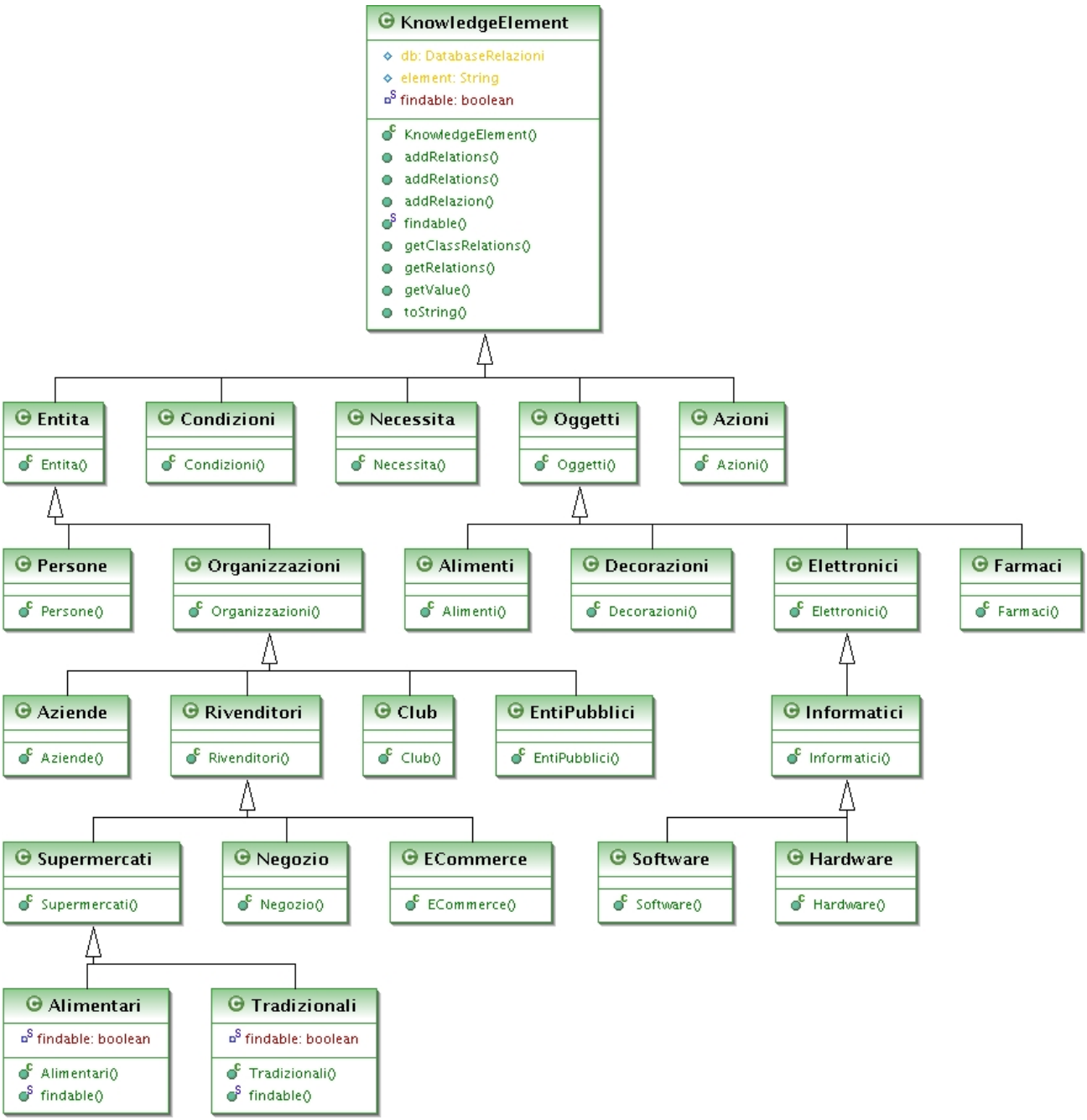


Figura 5.2: Schema UML del pacchetto knowledge_members

Classe KnowledgeElement

La classe **KnowledgeElement** è la classe base da cui derivano tutte le categorie di elementi che rappresentano la conoscenza del sistema. Come si può osservare dallo schema UML in figura 5.2 la classe possiede i seguenti dati membro:

- **DatabaseRelazioni db**: costituisce un riferimento al database che tiene conto delle diverse relazioni tra le categorie e quelle che legano le singole istanze a categorie specifiche.
- **String element**: rappresenta la stringa a cui l'oggetto è associato.
- **boolean findable**: indica se la classe rappresenta una categoria che consente di generare una query di ricerca da inviare al GIS

I primi due dati membro hanno visibilità protetta, vale a dire possono essere modificati e interrogati direttamente tramite metodi delle classi derivate. Il terzo elemento è privato e dipende dall'istanza. Di default assume valore *false* e viene ereditato dalle classi derivate. Per modificarne il valore sarà necessario ridefinirlo con procedura di *override*. La sua ridefinizione comporta l'obbligo di ridefinire anche il metodo di verifica (*findeable()*). Se non lo si ridefinisse si utilizzerebbe quello fornito dalla classe *KnowledgeElement* che mostrerebbe il suo valore locale (*false*) e non quello ridefinito.

Oltre ai dati membro sopra citati, la classe fornisce dei metodi di elaborazione e gestione delle relazioni:

- **addRelation(...)**: consente di aggiungere una relazione che lega l'istanza corrente a una categoria, questa rappresentata con un oggetto di tipo *Class*. Ne esistono tre versioni: una che aggiunge una sola relazione e due che aggiungono un insieme di relazioni rappresentate da una lista nel primo caso e da un array monodimensionale nel secondo.
- **getValue()**: ritorna un oggetto *String* corrispondente al valore da esso rappresentato
- **findeable()**: ritorna *true* se la categoria a cui appartiene l'istanza può costituire una keyword di ricerca nel *GIS*.

Classe Alimenti

La classe **Alimenti**, come tutte le classi che costituiscono una categoria di rappresentazione della conoscenza ma non sono ricercabili sul *GIS*, eredita tutti i medoti e dati membro della classe *KnowledgeElement* senza ridefinirne alcuno.

L'unicanota è il costruttore: viene esplicitato per poter richiamare il costruttore della classe base tramite l'operatore **super**.

Classe Alimentari

La classe **Alimentari** è un classico esempio di categoria che è rappresentata nel *GIS*. Sono stati ridefiniti due elementi:

- dato membro statico *findable*: ridefinito a valore *true*
- funzione statica *findeable*: ridefinita per poter accedere al valore di *findeable* della classe derivata e non di quella base

Come già visto per la classe *Alimenti*, è ridefinito il costruttore che va a richiamare il costruttore della classe base.

5.3.2 Pacchetto databases

Come già visto in precedenza nel paragrafo 4.3.2, il pacchetto *databases* contiene gli oggetti necessari per creare una sorta di collezione di dati utili al corretto funzionamento del sistema. L'analizzatore semantico utilizza due database in particolare:

- DatabaseRelazioni
- DatabaseRicerche

Entrambe le classi sono *singleton*. Questo vuol dire che tali oggetti non necessitano di un essere istanziati esplicitamente. Verranno istanziati la prima volta che se ne richiede l'uso e da quel momento saranno utilizzabili da ogni componente che li condivide. [1].
In figura 5.3 è possibile visualizzare uno schema UML delle classi sopra citate.

Classe DatabaseRelazioni

La classe **DatabaseRelazioni** si occupa di tenere traccia di tutte le possibili relazioni sia tra le categoria, sia tra istanza e categoria. Si può considerare una elementare *Rete Semantica*. Si è scelto di implementare una struttura ad-hoc per il fatto che utilizzarne una già esistente sarebbe stato uno strumento troppo sofisticato per l'obiettivo che ci si è prefissi di raggiungere e avrebbe richiesto maggiori studi di integrazione. In futuro si prevede di integrarne una utilizzando le tecnologie già presenti.
Per fare questo vengono utilizzati due particolari dati membro:

- Map<Class, List<Class>> relazioni
- Map<KnowledgeElement, List<Class>> relazioni_istanza

La mappa *relazioni* si occupa di associare ad ogni categoria una lista di categorie con le quale esiste una relazione. Ogni categoria è rappresentata da un oggetto di tipo *Class*.

[1] Si riveda lo schema presentato nel paragrafo 4.3.2

Questo è importante per poter utilizzare la *reflection* di cui si discuterà nell'analisi del codice dell'*Analizzatore semantico*.

La mappa *relazioni_istanza* si occupa di associare un KnowledgeElement, ovvero un'istanza di una categoria descritta nell'ontologia, con una lista di oggetti di tipo *Class* che rappresentano le categorie con cui esso è in relazione.

Al fine del corretto funzionamento è necessario che gli oggetti di tipo *Class* che vengono inseriti in queste mappe appartengano al pacchetto knowledge_member e derivino dalla classe *KnowledgeElement*. Questo vincolo si rende necessario per imporre coerenza alla rete di relazione che si realizza definendo un opportuna ontologia.

Insieme ai dati membri visti fin ora, la classe *DatabaseRelazioni* fornisce dei metodi per l'inserimento e l'interrogazione delle relazioni:

- **addRelations(...)**: consente di aggiungere una relazione che lega un istanza ad una categoria o tra due categorie. Ne esistono tre versioni per entrambe le tipologie di relazioni inserite: una che aggiunge una sola relazione e due che aggiungo un insieme di relazioni rappresentate da una lista nel primo caso e da un array monodimensionale nel secondo.

- **getRelations(...)**: consente di ottenere un iteratore sulla lista di relazioni che coinvolge l'istanza passata come argomento o la categoria. Se si richiedono le relazioni per una particolare istanza viene riportata una lista che contiene anche le relazioni che coinvolgono la categoria a cui l'istanza appartiene.

L'efficienza di una collezione dati così concepita dipende dall'utilizzo estensivo delle collezioni di dati di tipo *Map*. Usare una mappa aiuta a non dover iterare un intera lista per identificare la categoria o l'oggetto *KnowledgeElement* per identificare le relazioni in

Figura 5.3: Schema UML del pacchetto databases

cui è coinvolto. Resta inevitabile, però, che la mappa contenga liste in cui la chiave di ricerca è coinvolta. Più l'ontologia è complessa e peggiore sarà l'efficienza dell'algoritmo. L'operazione di confronto è rapida e non richiede particolari elaborazioni e, quindi, può essere trascurabile al fine di ottenere un indicatore prestazionale. La parte significativa è data, appunto, da due fattori:

- quantità di categorie e istanze
- numero medio di relazioni per categorie e istanze

Sulla base di questi due fattori si può calcolare la complessità computazionale della gestione ed elaborazione delle relazioni e dell'ontologia.

Classe DatabaseRicerche

La classe **DatabaseRicerche** si occupa di immagazzinare in una lista tutte le chiavi di ricerca, sotto forma di oggetti di classe *String*. Ogni volta che l'*Analizzatore Sintattico* riconosce una categoria che può essere usata per interrogare il *GIS*, viene aggiunta la stringa che la identifica nel database. Le chiavi vengono rimosse in seguito alle risposte fornite dall'utente e ciò può avvenire per i seguenti motivi:

- scaduta l'età massima di validità della necessità rilevata
- l'utente ha rifiutato di soddisfare la necessità rappresentata
- l'utente ha soddisfatto la necessità rappresentata

Anche questa classe, come le altre del pacchetto databases, è un *Singleton*, pertanto valgono gli stessi discorsi visti in precedenza.[2] L'unico dato membro è un oggetto di tipo LinkedList di chiavi di ricerca. La gestione di tale collezione viene fatta attraverso quattro funzioni fornite dalla classe *DatabaseRicerche*:

- **add(String s)**: aggiunge una stringa alla lista delle chiavi di ricerca.
- **iterator()**: ritorna l'iteratore alla lista di chiavi di ricerca.
- **remove(String s)**: rimuove la chiave di ricerca passata come argomento dalla lista.
- **clear()**: cancella ogni elemento dalla lista.
- **signal()**: impone il flag di segnalamento a true. Serve per controllare il thread di ricerca sul *GIS*.
- **signaled()**: ritorna il valore della variabile *signaled*: se è positivo viene consumato riponendolo a valore *false*. In caso contrario viene ritornato *false*.

[2]Si riveda lo schema presentato nel paragrafo 4.3.2

Si tenga presente che questo oggetto è condiviso da due elementi indipendenti tra loro istanziati, nel sistema generale, come *Thread*:

- *Analizzatore Semantico*
- *Finder*

Per questo motivo, è necessario predisporre un meccanismo di sincronizzazione ogni volta che si vuole accedere ad uno dei metodi presentati sopra. La classe non ne fornisce di suoi per la presenza dell'iteratore: durante tutta la fase di iterazione si dovrà possedere il lock sul database stesso.

Non predisponendo meccanismo di mutuo acceso ai metodi dell'oggetto in questione, si vincola ad implementarli nelle parti di codice che ne richiedono l'uso. Ciò porta ad avere una sincronizzazione più esplicita evitando così possibile errori di acquisizione di lock su oggetti diversi durante le fasi di manipolazione della lista.

5.4 Funzionamento dell'Analizzatore Semantico

Per capire il funzionamento dell'**Analizzatore Semantico** è necessario iniziare con una descrizione molto generale per poi aumentare sempre di più il livello di dettaglio. La figura 5.4 mostra lo schema UML della classe *AnalizzatoreSemantico*: A differenza delle

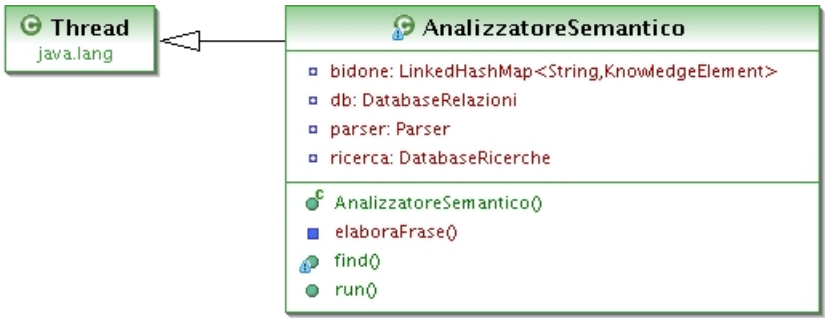


Figura 5.4: Schema UML della classe *AnalizzatoreSemantico*

classi viste fin ora, questa è la prima volta che viene presentata una derivazione da *Thread*. Per come è pensato il sistema, tutte le componenti viste fin ora vengono utilizzate dall'analizzatore semantico che, essendo un thread indipendente, può lavorare in parallelo alle componenti che gestiscono le interrogazioni sul *GIS* e processano le segnalazioni dal GPS (si veda il paragrafo 6).
Analizziamo, per prima cosa, i dati membro della classe:

- **Parser parser**: un oggetto di tipo parser, restituisce un oggetto di tipo *Frase* che viene processato nei metodi forniti dalla classe.

- **LinkedHashMap<String, KnowledgeElement> bidone**: collezione di dati di tipo mappa che contiene l'insieme di oggetti di tipo *KnowledgeElement* conosciuti e ne definisce la relazione con le stringhe che li rappresentano.
- **DatabaseRelazioni db**: un riferimento al singleton di tipo *DatabaseRelazioni* per poter calcolare le relazioni in cui gli oggetti e le categorie sono coinvolti
- **DatabaseRicerche ricerca**: un riferimento al singleton di tipo *DatabaseRicerche* per poter inserire le chiavi di ricerca per l'interrogazione al *GIS*.

A lungo si è discusso sul fatto di integrare la mappa *bidone* all'interno del singleton *DatabaseRelazioni*. Ciò non è stato fatto perché quest'ultimo è pensato solo per gestire le relazioni, non per immagazzinare anche tutti i *KnowledgeElement* riconosciuti. Negli sviluppi futuri è, infatti, prevista una fase di addestramento automatico attraverso la quale è possibile riempire il database di relazioni e di *KnowledgeElement* sfruttando due differenti fonti. Separare le cose ha quindi una rilevanza logica.
I metodi forniti dalla classe *AnalizzatoreSemantico* sono i seguenti:

- **Costruttore**
- **run()**
- **elaboraFrase(Frase f)**
- **find(Iterator<Class> rel)**

Analizziamo ciascun metodo individualmente introducendo, anche, il concetto di *reflection* utile per la corretta comprensione dell'ultimo metodo presentato.

5.4.1 Costruttore

Come per la classe *Parser*, anche in questo caso il costruttore si occupa unicamente di leggere il file di testo che contiene le specifiche per il corretto funzionamento della classe *AnalizzatoreSemantico*. In questo caso i dati riguardano gli oggetti di tipo *KnowledgeElement* e le relazioni tra le categorie e tra le istanza e categorie.

A seconda della quantità di informazioni che viene letta nel file di testo, la procedura può essere più o meno rapida.

5.4.2 Funzione run()

La funzione *run* è un metodo della classe *Thread*. Rappresenta il codice che verrà eseguito una volta mandato in esecuzione il thread stesso. Il funzionamento è molto semplice ed è descritto dai seguenti punti:

1. Viene estratta dal *Parser*

2. Se dal *Parser* viene estratto il valore *null* vuol dire che si è raggiunta una condizione di *halt* (terminazione) dell'algoritmo. In questo caso viene stampata su video la lista di ricerche e il thread termina la sua esecuzione.

3. Nel caso in cui viene estratto un oggetto di tipo *Frase* viene eseguita la funzione *elaboraFrase(Frase f)* al fine di processare l'oggetto passato per argomento identificando le possibili chiavi di ricerca ad esso associate.

La procedura descritta sopra viene iterata fino a che non si ottiene un valore *null* dal *Parser*.

5.4.3 Funzione elaboraFrase()

La funzione *elaboraFrase* costituisce il core dell'algoritmo. Processa l'oggetto di tipo *Frase* ottenuto dal *Parser* per identificare le chiavi di ricerca per interrogare il *GIS*. Il funzionamento è descritto dai seguenti passi:

1. Viene estratto il *verbo* e si verifica se identifica un bisogno o una mancanza:
   - se la verifica è positiva: si procede con l'elaborazione
   - in caso negativo si termina l'analisi
2. Viene selezionata la lista in cui andare a ricercare le chiavi per generare le query di interrogazione al *GIS*:
   - si sceglie la lista di *soggetti* se il primo elemento è un pronome ed è in forma passiva le chiavi andranno calcolate sulla base del soggetto
   - in caso contrario le chiavi saranno calcolate sulla base dell'oggetto
3. Una volta identificata la lista di elementi in cui andare a ricercare le chiavi, la si scorre e, per ogni suo elemento:
   - si estrae la stringa corrispondente all'oggetto di classe *Symbol* appena estratto
   - si cerca nella mappa *bidone* l'oggetto di tipo *KnowledgeElement* corrispondente
   - si estrae da esso la lista di relazioni in cui lui e la categoria di appartenenza sono coinvolti
   - si processano le relazioni attraverso la funzione *find(Iterator<Class> rel)*. Quest'ultima aggiungerà ogni categoria *findable* sul *GIS* nel *DatabaseRicerche*

Come già detto in precedenza, la complessità computazionale dell'analisi dipende fortemente dalla composizione del *DatabaseRicerche*. Maggiormente è complesso, peggiori saranno le performances globali del sistema. Ciò lo si può riscontrare con maggiore dettaglio nella funzione *find*.

5.4.4 Reflection

Prima di procedere all'analisi dell'ultima funzione è necessario approfondire un concetto che prende il nome di *reflection*. Con **reflection** si intende il processo tramite il quale un componente software può ottenere informazioni relative ai tipi di dato in run-time. Il paradigma di programmazione guidato dalla reflection è chiamato programmazione riflessiva.

E' utile nelle moderne architetture di software in cui i dati sono rappresentati e immagazzinati attraverso oggetti e le istruzioni corrispondono a metodi eseguiti su oggetti. In questo contesto può essere utile principalmente per:

- istanziare un oggetto il cui tipo è scelto in run-time
- eseguire metodi scelti in run-time su un tale oggetto
- avere informazioni generali su un tipo di oggetto:
  - quali sono i dati membro
  - quali sono i metodi e i rispettivi argomenti e valori di ritorno

In java ciò è possibile attraverso due classi: *Class*, per avere informazioni sulle classi e manipolarle, e *Method*, per avere informazioni circa i metodi e manipolarli. Per chiarezza osserviamo un esempio:

```
// Classi
// Creazione di un oggetto Class associato a un oggetto passato
// in forma di stringa
Class x =  Class.forName("java.lang.String");

//  Estrazione di informazioni sul tipo
Constructor[] c = x.getConstructors();
Field[] f = x.getDeclaredFields();
Method[] m = x.getDeclaredMethods();
Type i[] = x.getGenericInterfaces();
Type sc = x.getGenericSuperclass();
String name = x.getName();
Package p = x.getPackage();

// Allocazione di un oggetto appartenente a quel tipo di dato
Object o = x.newInstance();

// Verifica se l'oggetto è un istanza della classe
boolean ver = x.isInstance(o);

// Metodi
// Raccolta di informazioni sul primo metodo della classe sopra elaborata
```

```
String namemethod = m[0].getName();
Type tr = m[0].getGenericReturnType();
Type[] e = m[0].getGenericExceptionTypes();
Type[] par = m[0].getTypeParameters();

// Esecuzione del Metodo sull'oggetto prima allocato
// con la creazione di un array per gli argomenti da passare
Object args[] = new Object[par.length];
Object result = m[0].invoke(o, args);
```

Nell'esempio appare evidente come in run-time è possibile instanziare oggetti ed eseguire metodi la cui scelta dipende da un parametro in forma di stringa, che può essere letto, ad esempio, da tastiera. Nella funzione *find*, che vedremo nel paragrafo successivo, verranno utilizzate alcune funzioni appena analizzate.

5.4.5 Funzione find(Iterator rel)

La funzione **find(Iterator<Class> rel)** si occupa di esplorare ricorsivamente una lista di relazioni e aggiungere al *DatabaseRicerche* tutte le chiavi associate a categorie ricercabili tramite interrogazione al *GIS*.
Il funzionamento è descritto dalle seguenti fasi:

1. Viene estratto un elemento dall'iteratore sulla lista di relazioni passato per argomento
2. Tramite *reflection* viene eseguito il metodo statico *findable* sull'oggetto trovato. Ciò può essere fatto perché si verifica che questo oggetto appartenga al pacchetto *knowledge_member*. Come già visto nel paragrafo 5.3.1, tutte le classi appartenenti a tale pacchetto derivano da *KnowledgeElement* e posseggono, quindi, il metodo *findable()*.
3. Viene processato il valore di ritorno della funzione *invoke(Object obj, Objects[] args)*):
   - se è *positivo* la categoria trovata appartiene a quelle che possono generare una query sul *GIS* e, quindi, viene aggiunta la sua chiave di ricerca nel *DatabaseRicerche*.
   - se è *negativo* la categoria trovata non appartiene a quelle che possono generare una query sul *GIS*.
4. In entrambi i casi, viene estratta la lista di relazioni che coinvolgono quella categoria e viene richiamata ricorsivamente la funzione *find(Iterator*<**Class> rel)** passando, come argomento, il suo iteratore.

La procedura sopra citata viene iterata per ogni elemento presente nella lista il cui iteratore viene passato come argomento. Nel caso in cui venisse aggiunta una nuova chiave di ricerca, viene eseguito il segnalamento al thread che si occupa di effettuare le ricerche sul *GIS* tramite una *notify()* eseguita sul *DatabaseRicerche*, condiviso da entrambi i thread.

5.5 Conclusioni

Inizialmente il progetto era nato con l'obiettivo di utilizzare una rete semantica standard che inglobasse le tecnologie che si utilizzano solitamente nel campo del *Semantic Web*, quali *OWL*, *RDF* e similari. In fase di realizzazione si è scelto, però, di realizzare una struttura ad-hoc per ottenere un miglior livello di integrazione con la struttura globale del sistema. Una vera e propria rete semantica possiede, oltre alla struttura di immagazzinamento delle relazioni e al sistema di interrogazione, delle procedure di ragionamento che consentono di apprendere ed aumentare la qualità di conoscenza rappresentata dalla rete. Si prevede che, in futuro, si modificherà la struttura attuale per integrarla con una rete semantica vera e propria.

La struttura semantica realizzata nel prototipo presenta sicuramente il problema della complessità. Si può pensare che la ricerca delle relazioni sia un problema paragonabile alla ricerca su reti. Ciò è avvalorato dal fatto che per ogni istanza di *KnowledgeElement*, si genera un grafo. Tale processo non è attualmente molto efficiente perché alcuni segmenti del grafo possono dover essere ricostruiti piu' volte. L'inserimento, nell'architettura, di una sorta di memoria cache di sottografi di frequente utilizzo, permetterebbe di evitare l'esplorazione ripetuta di sottosezioni della struttura di memorizzazione delle relazioni. E' ovvio, comunque, che tanto maggiore sarà la ricchezza di una frase in termini di *KnowledgeElement*, tanto piu' complessa si rivelera' l'esplorazione della rete di relazioni.

E' importante, infine, sottolineare come, in questa fase del progetto, si è focalizzata l'attenzione sulla struttra di gestione delle relazioni tralasciando la formalizzazione di una vera e propria ontologia. Al contrario, ci si e' limitati a creare una piccola parte di essa utile unicamente alla verifica e validazione del prototipo.

Capitolo 6

Gestore Mappe

6.1 Linee Guida

L'ultimo blocco del sistema che andremo ad analizzare è quello che si occupa di effettuare le query di ricerca sul *GIS*. E' rappresentato da un thread indipendente dal resto del sistema. La struttura è pensata per potersi adattare ad effettuare ricerche su diversi *GIS*. E' diviso, per questo motivo in due elementi principali:

- **Finder**: oggetto che si occupa di gestire la ricerca
- **GISFinder**: oggetto che si occupa di effettuare la ricerca su un particolare *GIS*

Passare da un *GIS* all'altro è molto facile e basta solo definire un oggetto derivato da *GISFinder* con le specifiche di ricerca proprie del sistema scelto.
L'obiettivo generale di questo blocco è quello di ricercare sul *GIS*:

- luoghi presenti nel territorio in un raggio di un massimo di 10 km dalla posizione dell'utente e salvarli in un database
- per ognuno calcolare il percorso che l'utente deve percorrere per raggiungerlo a partire dalla sua posizione locale

Si tenga presente che con il concetto di *luogo* non identifica una posizione ma anche un entità che può essere un ristorante, un centro commerciale, un bar e molto altro ancora.

In questa tesi non viene trattata la realizzazione del GPS perché si pensa di utilizzarne uno già esistente che si limiti a fornire la posizione corrente dell'utente sotto forma di stringa.
Le classi che costituiscono questo componente sono raggruppate nei seguenti pacchetti Java:

- databases
- maps

6.2 Pacchetto databases

In figura 6.1 è riportato lo schema UML del pacchetto *databases* e, in particolare dell'ultima classe che andremo a vedere: *DatabasesLuoghi*

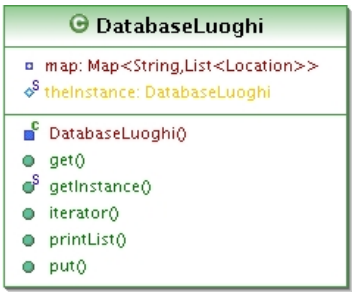


Figura 6.1: Schema UML del pacchetto databases

6.2.1 Classe DatabaseLuoghi

Questa classe si occupa di raccogliere tutti i luoghi trovati tramite l'interrogazione sul *GIS*. Sono collezionati attraverso una struttura dati di tipo *lista* raggruppate all'interno di una struttura dati di tipo *mappa* le cui chiavi corrispondono alle categorie ricercate nel **GIS**.

Anche questo database, come quelli visti in precedenza, è un *singleton*[1]. Valgono, quindi, le stesse considerazioni viste in precedenza.

L'unico dato membro che compare è la mappa che prende il nome di *map*. All'interno di essa sono contenute delle *LinkedList* di oggetti di tipo *Location*, ordinate in base alla distanza dal punto in cui si trova l'utente.

I metodi presenti nella classe sono semplicemente quelli che si occupano di gestire l'inserimento, la rimozione e la consultazione della mappa:

- **put(String s, List<Location> l)**: aggiunge una lista di luoghi trovati nell'*GIS* corrispondenti alla categoria passata come argomento. La lista, prima di essere inserita viene ordinata sulla base della distanza rispetto alla posizione corrente dell'utente in ordine crescente.

- **get(String c)**: ritorna la lista di luoghi trovati corrispondenti alla categoria passata per argomento. La lista, prima di essere restituita, viene ordinata tramite la funzione *Collections.sort()* che può essere applicata a strutture dati che contengono oggetti che implementano l'interfaccia *Comparable*.

- **printList(String c)**: stampa su video tutti i luoghi trovati in relazione alla categoria passata per argomento.

[1]Si riveda lo schema presentato nel paragrafo 4.3.2

- **iterator(String c)**: ritorna un iteratore sulla lista contenente i luoghi trovati per la categoria passata per argomento.

Questo oggetto non è condiviso, pertanto non è necessario predisporre sistemi di accesso in mutua esclusione.

6.3 Pacchetto maps

Questo pacchetto contiene tutti gli elementi necessari al fine di effettuare la ricerca sul *GIS*. In figura 6.2 è possibile osservare uno schema UML dei vari componenti.

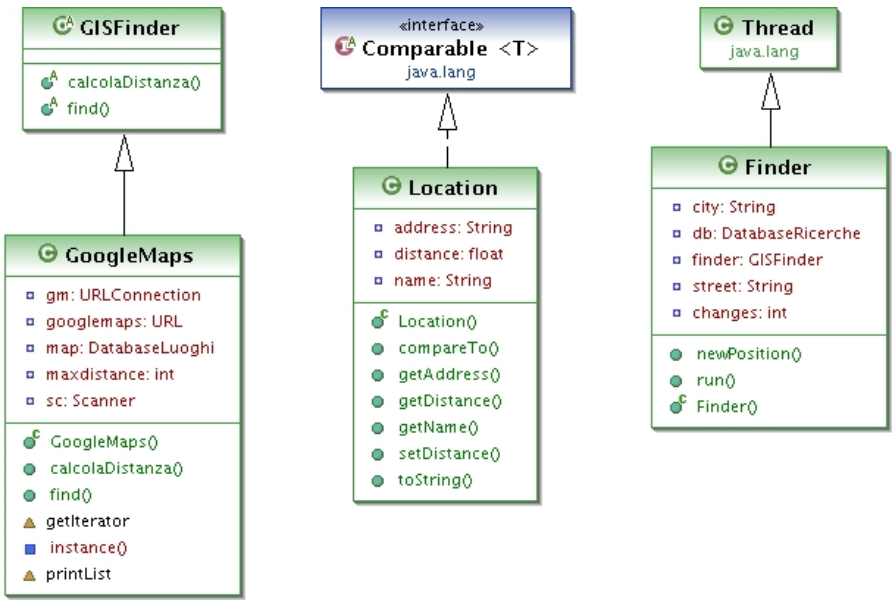


Figura 6.2: Schema UML del pacchetto maps

6.3.1 Classe Location

La classe **Location** si occupa di rappresentare i luoghi trovati tramite l'interrogazione al *GIS*. Come si può osservare dallo schema UML in figura 6.2 implementa l'interfaccia *Comparable*. Questo consente di poter effettuare un ordinamento tra due istanze sulla base della definizione del metodo *compareTo(Object<T> o)* di cui discuteremo in seguito.

Sempre dallo schema UML si può osservare la presenza di tre dati membro fondamentali per la trattazione dei luoghi:

- **String name**: identifica il nome del luogo
- *String address*: identifica l'indirizzo in cui si trova

- *float distance*: indica la distanza dalla posizione dell'utente

I metodi che manipolano tali dati sono i seguenti:

- **Location(String q)**: costruttore che accetta come argomento una stringa così formattata:

  (*ADDRESS*) (*OTHER*) (*NAME*)

  ove *ADDRESS* e *NAME* sono stringhe che possono contenere anche spazi. Il costruttore non inizializza anche il campo *distance* perché prima viene identificato il luogo e poi segue la query che calcola la distanza dal punto in cui si trova l'utente.

- **getName()**: ritorna il nome.

- **getAddress()**: ritorna l'indirizzo.

- **getDistance()**: ritorna la distanza dal punto in cui si trova l'utente.

- **setDistance(float d)**: imposta la distanza dalla posizione dell'utente calcolata attraverso una query al *GIS*.

- **compareTo(Location l)**: permette di confrontare due oggetti di tipo *Location*. Il valore di ritorno è un intero ottenuto come differenza della distanza tra l'oggetto corrente e quello passato per argomento. Grazie a questa funzione si può effettuare un ordinamento. L'idea è quella di avere una collezione di dati di *Location* che contenga i luoghi trovati in ordine di distanza per poter interrompere la ricerca non appena si è fuori dal raggio di 1 km dalla posizione dell'utente.

L'utilità di questo oggetto è quella di immagazzinare una sola volta in memoria i luoghi trovati e di aggiornarli con la nuova posizione. Il prototipo è stato realizzato utilizzando *GoogleMaps* come *GIS* e si è semplicemente effettuata una query calcolando la distanza dal punto in cui si trova l'utente e ricalcolandola ogni volta che quest'ultimo si sposta. In futuro si pensa di organizzare la struttura cercando di calcolare automaticamente i correttivi ed evitando di interrogare ripetutamente il sistema informativo a meno che la posizione non si sia modificata ripetutamenente rendendo obsoleti i dati raccolti.

6.3.2 Classe Finder

La classe **Finder** si occupa di organizzare la ricerca sul *GIS*. E' il componente principale di quest'ultimo blocco del sistema. Come si può notare dallo schema UML riportato in figura 6.2 si nota che la classe deriva da *Thread*. Questo implica, come già visto per il caso dell'*AnalizzatoreSemantico* che si tratta di un entità autonoma che esegue istruzioni concorrentemente ad altri blocchi del sistema. Sarà necessario prestare attenzione alla parte di sincronizzazione e mutua esclusione sull'accesso e manipolazione di variabili condivise.
I dati membro della classe sono i seguenti:

- **String street**: indica l'ultima via in cui il *GPS* ha identificato la presenza dell'utente
- **String city**: indica l'ultima città in cui il *GPS* ha identificato la presenza dell'utente
- **DatabaseRicerche db**: un riferimento al singleton che immagazzina le chiavi di ricerca inserite dall'*AnalizzatoreSemantico*
- **GISFinder finder**: un riferimento a un oggetto che effettua la ricerca sul *GIS* tenendo conto delle specifiche dell'architettura target.
- **int changes**: indica quante volte è cambiata la posizione dell'utente

Per quanto riguarda i metodi, segue la descrizione dettagliata dei seguenti:

- *Finder(String s, String c)*
- *run()*
- *newPosition(String street, String city)*

Funzione Finder(String s, String c)

Il costruttore della classe accetta due argomenti in input:

- la via in cui si trova l'utente
- la città in cui si trova l'utente

Per effettuare la prima ricerca è necessario possedere già questi dati. Ad ogni spostamento verranno modificati tramite la funzione *newPosition(String street, String city)*.

Funzione run()

La funzione **run()** rappresenta il codice che viene eseguito in parallelo al resto del sistema. Il funzionamento è molto semplice ed è descritto dai seguenti step:

1. Fino a quando non viene generata una nuova chiave di ricerca oppure non viene modificata la posizione dell'utente, il thread è in attesa.
2. Appena viene segnalato un evento sopra citato, viene estratto l'iteratore sulla lista di chiavi di ricerca e viene eseguito, per ciascuno, il metodo *find(String street, String city, String category)* della classe derivata da *GISFinder* di cui parleremo in seguito. Tale funzione si occupa di cercare se esistono dei luoghi corrispondenti alla categoria passata per argomento in un raggio di 10 km dalla posizione corrente dell'utente. Per tutta l'iterazione si mantiene il lock sul *DatabaseRicerche*.

La procedura viene eseguita fino a terminazione del programma.

Funzione newPosition(String street, String city)

La funzione **NewPosition(String street, String city)** si occupa di aggiornare la posizione dell'utente in termini di via e città. Ogni volta che viene richiamata, acquisisce il lock sull'oggetto di tipo *DatabaseRicerche*, segnala la condizione di sveglia da parte del *Thread Finder* e lo sveglia attraverso la chiamata della funzione *notify* sul database su cui è in *wait*.

Questa funzione deve essere chiamata dal *GPS* ogni volta che cambiano le coordinate dell'utente. Il successivo ricalcolo viene effettuato ogni 3 cambi di posizione per evitare calcoli complessi su piccoli spostamenti.

6.3.3 Classe GISFinder

La classe astratta *GISFinder* serve per definire un template a cui le classi derivate devono fare riferimento. Tali classi si occuperanno di effettuare la ricerca su uno specifico *GIS* tenendo conto delle peculiarità che questo possiede rispetto agli altri. L'utilità è data dal fatto di voler rendere il più modulare possibile il sistema e renderlo adattabile a qualsiasi struttura di riferimento. Cambiare *GIS* comporta solamente scrivere un nuovo oggetto derivato da *GISFinder*.

Essendo una classe astratta, non sono previsti dati membro. I metodi non sono implementati per vincolare il programmatore a implementarne il corpo durante la scrittura delle classi da essa derivate. Essi sono:

- **calcolaDistanza(String via, String citta, Location l)**: calcola la distanza tra la Location passata per argomento e la posizione dell'utente passata anch'essa per argomento. Quest'ultima stringa ha la seguente sintassi: *via+città*.
- **find(String via, String citta, String categoria)**: effettua la ricerca dei luoghi appartenenti alla categoria passata per argomento nei pressi della posizione dell'utente ottenuta tramite i primi due argomenti.

L'implementazione di questi metodi dipende dal *GIS* di riferimento e verranno sviluppati negli oggetti da essa derivati.

6.3.4 Classe GoogleMaps

Come caso studio per poter effettuare i test sul prototipo realizzato si è scelto di utilizzare **GoogleMaps** come *GIS* di riferimento. Per adattare il sistema a interrogarlo si utilizza la classe **GoogleMaps**.

Google Maps è un servizio offerto via web e non dispone di API di interrogazione in java. Vengono fornite unicamente API Javascript per visualizzare le mappe ma non è possibile effettuare interrogazioni da applicazioni esterne. E' stato quindi necessario studiare il codice html generato da Google e gli indirizzi di ricerca per poter pilotare le richieste http. Vediamo in seguito i dati raccolti durante l'analisi e come si è implementata la classe.

Ricerca di luoghi

Per capire come poter effettuare correttamente la ricerca dei luoghi su *Google Maps* è stato necessario fare delle prove utilizzando un browser classico. In figura 6.3 è possibile vedere

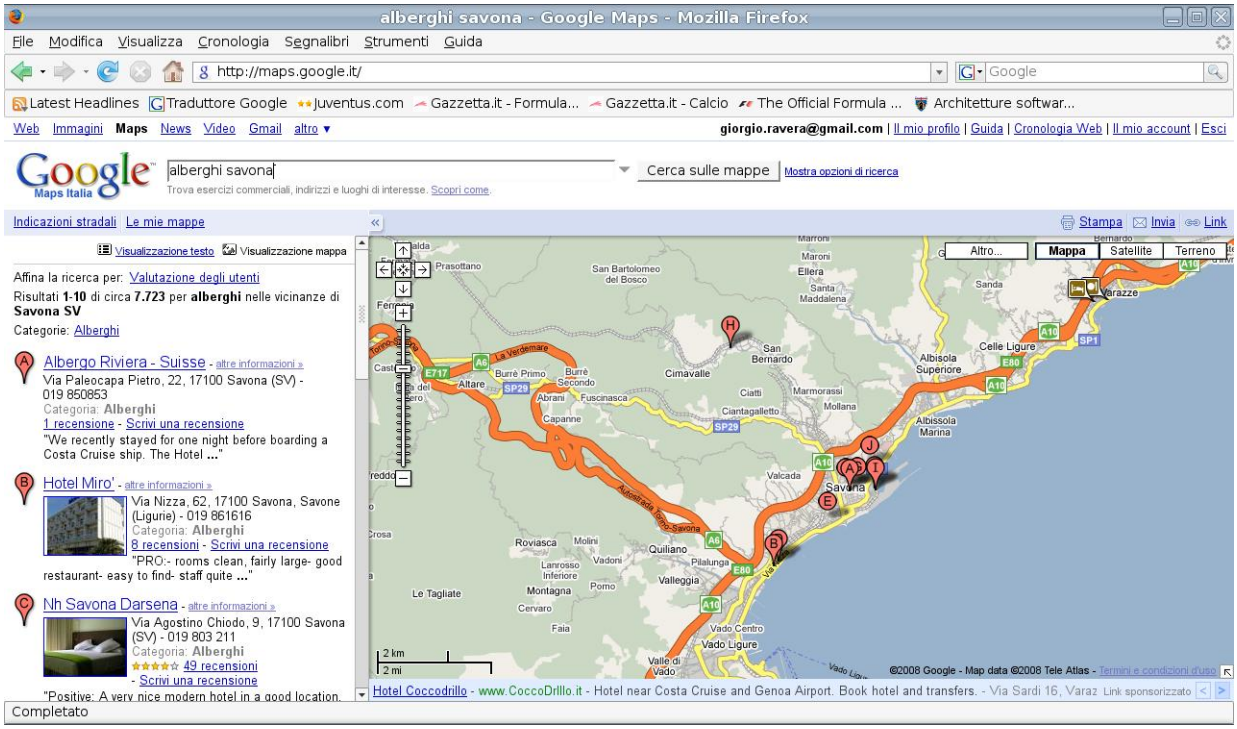


Figura 6.3: Ricerca di luoghi su Google Maps

l'output ottenuto inserendo nel campo di testo la chiave:

alberghi savona

Come si può vedere viene visualizzato un frame diviso in due parti:

- a sinistra sono presenti gli alberghi trovati, ordinati più o meno a caso
- a destra la mappa in cui sono posizionati gli alberghi trovati

La parte di sinistra è divisa in pagine: ogni pagina mostra un massimo di 10 entità. In fondo, non presente nella figura, ci sono i pulsanti che permettono di avanzare alla pagina successiva o tornare a quella precedente. Pian piano che si scorrono le pagine la ricerca si estende, mostrando, non solo gli alberghi di savona, ma anche quelli nei dintorni. Se il numero di pagine continua a salire compariranno anche quelli presenti a Genova e così via.

Analizzando la pagina è possibile ottenere il link esatto che consente di visualizzare direttamente la pagina con il risultato della ricerca:

`http://maps.google.it/maps?f=q&hl=it&geocode=&q=`**alberghi**+**savona**`&start=00`

Per visualizzare i 10 elementi successivi il link da usare è il seguente:

`http://maps.google.it/maps?f=q&hl=it&geocode=&q=`**alberghi**+**savona**`&start=10`

Da ciò si può generalizzare la seguente stringa di ricerca:

`http://maps.google.it/maps?f=q&hl=it&geocode=&q=`**<CATEGORIA>**+
<CITTA>`&start=`**<PAGINA-1>**`0`

Una volta che è noto il modello di link da usare per interrogare il *GIS* bisogna capire come identificare le categorie trovate. Di seguito viene riportata una parte significativa del codice html corrispondente alla pagina richiesta attraverso il link visto in precedenza:

```
{id:"A",lat:44.308446000000004,lng:8.4782890000000002,image:
"/intl/it_it/mapfiles/markerA.png",elms:[4,1,6,2,9,1,5],
laddr:"Via Paleocapa Pietro, 22, 17100 Savona (SV) (Albergo Riviera - Suisse)",
sxti:"Albergo Riviera - Suisse",sxst:"Paleocapa Pietro",sxsn:"22",
sxct:"Savona",sxpr:"SV",sxpo:"17100",sxcn:"IT",sxph:"019850853",
.....
cat2:[{name:"Alberghi",lang:"it"}],
latlng:{lat:44.308446000000004,lng:8.4782890000000002}},
.....
```

Come si può notare, esiste un etichetta, **laddr** che identifica l'albergo con la seguente sintassi:

(VIA) (PROVINCIA) (NOME)

Da questa stringa si riesce a identificare correttamente l'albergo e si può procedere per la fase successiva, ovvero ottenere indicazioni sulla distanza tra il luogo trovato e la posizione attuale dell'utente.

Quanto visto fin ora si può realizzare in maniera automatica utilizzando un semplice algoritmo in Java che interroga, tramite protocollo *HTTP*, il server web di *Google Maps* ed effettua un parsing del codice *html* così ottenuto.

Per questo scopo è stata realizzata la funzione **find(String via, String citta, String categoria)** che, data la posizione corrente dell'utente, in termini di via e città, e la categoria da cercare, svolge i seguenti passi:

1. istanzia la connessione con il webserver di *Google Maps* richiedendo la prima pagina
2. legge riga per riga tutto il codice html. Se è presente la stringa *laddr*:
   - isola la stringa identificativa del luogo trovato
   - istanzia un oggetto *Location* corrispondente

- esegue il calcolo della distanza rispetto alla posizione dell'utente:
  - se è minore di 10 km l'oggetto *Location* viene aggiunto al *DatabaseLuoghi*
  - altrimenti l'interrogazione termina e si esce dall'algoritmo

Da questa procedura si può notare che la ricerca termina appena viene identificato un luogo che dista più di 10 km dal punto in cui si trova l'utente. Purtroppo Google Maps non li presenta in ordine e, pertanto, potrebbe capitare che, ad esempio, dopo un ristorante che dista 12 km dalla posizione dell'utente ne venga presentato uno che ne dista solamente 4. Dai test fatti questa eventualità occorre raramente. Solitamente le distanze sono più o meno progressive e, orientativamente, è difficile trovare una differenza di più di 2 km in meno rispetto ad una soluzione precedente, anche con l'avanzare delle pagine di ricerca.

Ottenere Indicazioni Stradali

Come è stato fatto per la fase precedente, anche per interrogare *Google Maps* al fine di ottenere le indicazioni stradali per raggiungere, dalla posizione in cui si trova l'utente, il luogo trovato, è necessario studiare il codice html generato dall'interrogazione tramite browser.
In figura 6.4 è presentata la pagina che viene visualizzata quando si richiedono le indicazioni stradali a *Google Maps*.

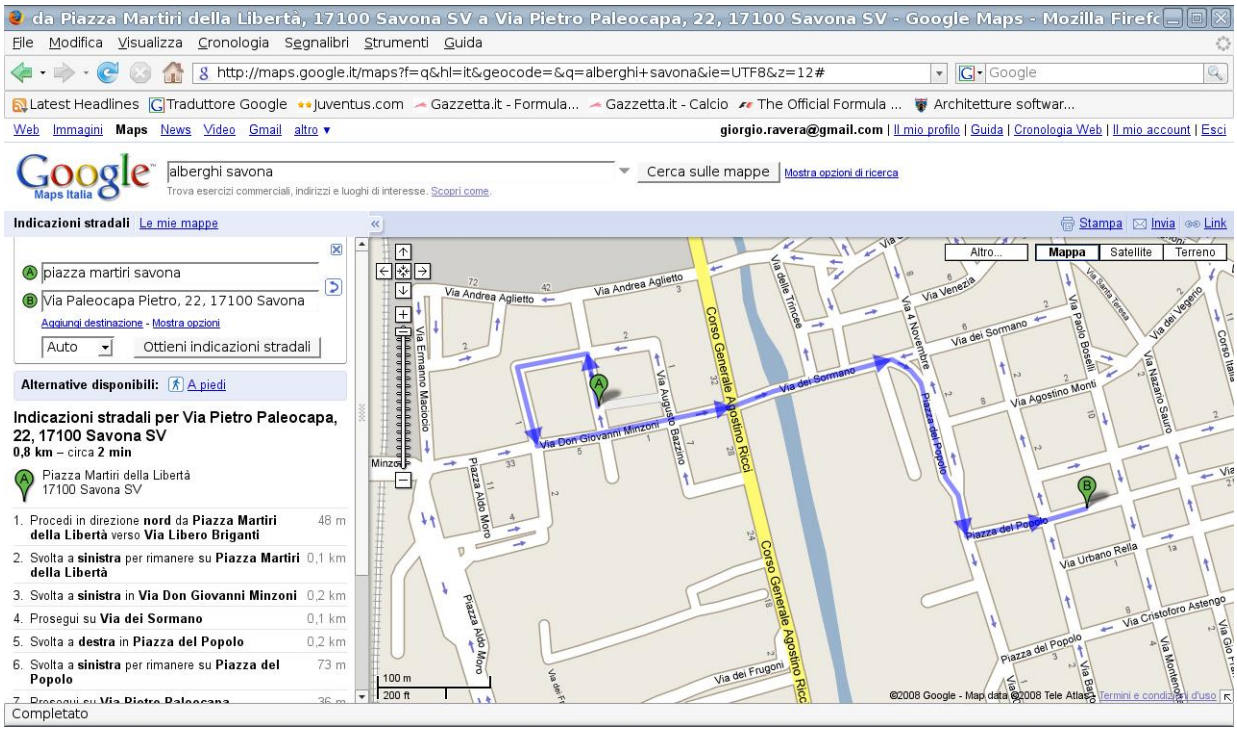


Figura 6.4: Ottenere indicazioni stradali su Google Maps

Dalla figura si possono osservare i seguenti aspetti interessanti:

- L'indirizzo della pagina lo si può ottenere, come in precedenza, da una parte comune e una personalizzabile con indirizzo di partenza e destinazione
- Catturando l'indirizzo del luogo rilevato nella fase precedente e collocandolo nel campo *B*, mentre nel campo *A* si inseriscono i dati rilevati dal *GPS* in termini di via e città, la ricerca viene effettuata correttamente e con precisione.
- Tra i dati presentati come output è indicata anche la distanza in km che è necessario percorrere per muoversi dal punto *A* al punto *B*.

Consideriamo per prima cosa l'indirizzo http da richiedere al server web di *Google Maps*. Come si può vedere dalla figura 6.4 ha la forma seguente:

```
http://maps.google.com/maps?f=d&saddr=ADDRESS_A&daddr=ADDRESS_B
```

ove *ADDRESS_A* costituirebbe la posizione attuale dell'utente ,mentre *ADDRESS_B* il luogo in cui si trova ciò che è stato trovato con la fase precedente. Nel nostro caso, un albergo. Con questo template di link è possibile richiedere le indicazioni stradali per muoversi da un luogo ad un altro.

A questo punto è necessario capire come interpretare la pagina html generata in seguito a questa richiesta. Di seguito viene riportata una porzione significativa del codice html ottenuto in seguito alla richiesta effettuata:

oppure

Come si può vedere esistono due versioni diverse di output:

- la prima codifica i tag html in codice esadecimale
- la seconda li mostra codificati in ascii

Differenziando le procedure di analisi, si può ricavare la distanza tra i due luoghi trovati andando a ricercare la stringa seguente:

```
b\\x3eDISTx26
```

oppure

```
<b>DIST&
```

ove *DIST* è la distanza espressa in codifica decimale tenendo conto della virgola. Una volta estratta tale informazioni è possibile ordinare i luoghi trovati con la procedura precedente ed ordinarli.

Può capitare che la ricerca risulti ambigua. In quel caso Google Maps fornisce una pagina speciale con elencate tutte le possibili ricerche. Questo caso non viene supportato e il luogo trovato viene escluso dalla ricerca. Si ritiene che questo non costituirà più un problema nel momento in cui si potrà effettuare direttamente la ricerca sul database di *Google Maps* tramite API e non tramite parsing di pagine html.

Quanto descritto fin ora viene gestito tramite la funzione **calcolaDistanza(String via, String citta, Location l)**. Il suo funzionamento è descritto in modo molto semplice dai seguenti passi:

1. viene richiesta la pagina il cui indirizzo è calcolato utilizzando il modello di link visto in precedenza completato con la posizione dell'utente e l'indirizzo della *location* rilevata in precedenza. Entrambi questi dati devono essere formattati in modo da sostituire gli spazi con l'operatore *+*. Lo spazio è un elemento che non può fare parte di un indirizzo http.

2. Viene istanziato un oggetto di tipo *Scanner* che restituisce tutte le stringhe che costituiscono la pagina html ottenuta in risposta alla richiesta fatta. I delimitatori che isolano le stringhe sono gli spazi.

3. se la stringa rilevata contiene le chiavi *noprint* e *km* vuol dire che al suo interno è definita la distanza.

4. se la condizione precedente si è verificata allora viene estratto il valore, convertito in float e inserito nell'oggetto Location e la funzione ritorna senza terminare l'analisi dell'intero documento. Esistono due diverse procedure per ottenere la conversione sia in caso di codifica esadecimale, sia in caso di codifica ascii.

5. se la pagina viene scorsa fino alla fine, vuol dire che la distanza non è stata trovata. In questo caso viene settato un valore -1. In presenza di tale risultato la funzione *find(String via, String citta, String categoria)* rimuove la location trovata dalla lista.

Anche in questo caso, come nel precedente, il tempo di esecuzione dell'algoritmo dipende fortemente dalla banda utilizzata per il collegamento internet. Sperimentalmente si è ossrevato che la fase di interrogazione al *GIS* è la fase più lenta di tutto il sistema. Una volta ottenuta la pagina html di risposta il parsing viene svolto velocemente.

Funzione instance(int page, String citta, String categoria)

L'ultima funzione che andremo ad analizzare è **instance(int page, String citta, String categoria)** che si occupa di realizzare la connessione tra il software e il server web di *Google Maps*.

Viene richiamata dalla funzione *find(String via, String citta, String categoria)* durante la ricerca delle *Location* per poter passare da una pagina di ricerca all'altra.

Il primo argomento indica la pagina di ricerca che si vuole richiedere, se la prima, la seconda ecc. Il secondo e il terzo argomento identificano le chiavi di ricerca: *città* e *categoria*. La prima cosa che viene svolta è la composizione del link in forma di stringa utilizzando gli argomenti ricevuti. Una volta fatto ciò viene creato un oggetto **URL** necessario per inizializzare un oggetto di tipo **URLConnection**. Tramite la funzione **openConnection()** eseguita sull'oggetto *URLConnection* appena creato, viene, di fatto, creata la connessione. In altri termini viene istanziato il Socket, inviata la richiesta al server web di *Google Maps* e ricevuta la risposta. Sempre dall'oggetto *URLConnection* viene estratto un *InputStream* tramite la funzione **getInputStream()** che viene usato per istanziare un oggetto di tipo *Scanner*.

Il risultato di questa operazione è avere un oggetto *Scanner* che legga la pagina html ottenuta in risposa alla richiesta del link appena costruito sulla base del template visto in precedenza.

Questo metodo è lo stesso usato dalla funzione *calcolaDistanza(String via, String citta, Location l)* per richiedere le pagine html usate per il calcolo della distanza tra la posizione dell'utente e la *Location* trovata.

Conclusione

Il lavoro svolto per realizzare il prototipo è stato suddiviso in 3 fasi differenti:

- realizzazione del parser
- realizzazione della rete di categorie
- realizzazione del sistema di interrogazione del *GIS*

Durante la prima fase ci si è concentrati sull'analisi delle principali forme grammaticali, senza fare riferimento ad una lingua specifica ma a una serie di legami tra i sintagmi. Questo ha portato a generare un sistema software in grado di adattarsi al linguaggio unicamente definendo delle costanti e potendo accedere ad un database che classifichi alcune parole. E' impensabile che un agente intelligente possa sapere a priori il significato di termini. Si pensi al noto caso studio di imparare il cinese in una stanza chiusa, senza contatti con il mondo esterno, possedendo unicamente il vocabolario cinese-cinese. Proprio come un bambino, il sistema è stato pensato per interrogare un database remoto e apprendere ciò che non sa. Nel momento in cui dovesse *ascoltare* un nuovo termine che non fosse presente nella sua *memoria*, interrogherebbe qualcuno o qualcosa al fine di capire il significato di tale termine. A questo scopo è stata pensata una fase di auto-addestramento sfruttando i servizi informativi messi a disposizione nel web. Il prototipo non è ancora in grado di poter accedere a tali database remoti ma è predisposto un sistema che consentirà, in futuro, di poterlo fare, sia per *imparare* nuovi verbi e loro coniugazioni (specialmente quelli irregolari), sia per distinguere nomi propri da aggettivi.

Il parser vocale, ovvero l'oggetto che ascolta il parlato umano e lo traduce in una serie di stringhe non è stato studiato perché questa tecnologia ha raggiunto un livello di maturità tale da permetterci di assumere di poter utilizzare un componente disponibile commercialmente. Per avere un sistema completamente funzionante si procederà, in seguito, a scegliere un implementazione software compatibile con i parametri richiesti dall'infrastruttura realizzata e inserirla nel sistema globale.

Per quanto riguarda l'analizzatore semantico il lavoro principale è stato quello di predisporre un sistema di gestione delle relazioni e un metodo efficiente per interrogarle. L'idea originaria era quella di utilizzare quanto già sviluppato nel mondo del semantic web limitandosi a definire un ontologia. Per poter comprendere pienamente il funzionamento delle tecnologia già esistenti e per avere una soluzione più performante e specifica a risolvere il problema da noi posto, si è scelto di implementare la struttura presentata. Il

suo funzionamento dipende fortemente da come è stata costruita l'ontologia e dal buon livello di integrazione con il sistema informativo scelto. In futuro si cercherà di uniformare il modello realizzato con quelli già presenti nel mondo del semantic web, predisponendo una ulteriore fase di auto-addestramento e profilazione dell'utente in modo da essere il più preciso possibile nel supportare le persone tenendo conto dei loro comportamenti consueti.

La parte di interrogazione del *GIS* non è stata particolarmente complessa, perché è stato sufficiente applicare tecniche di reverse engineering e di parsing di pagine html. Il problema fondamentale è la frequenza delle interrogazioni e l'eccessivo traffico *tcp* generato. In una versione successiva ci si propone di implementare una cache interna al sistema che permetta di non dover ricalcolare tutte le volte le stesse informazioni tenendo anch'essa in considerazione le preferenze dell'utente. Inserire un sistema prioritario che organizzi la lista di luoghi rilevati consentirebbe di ricercare solo gli elementi preferiti dal cliente. Può capitare, infatti, che il ristorante più vicino non sia di suo gradimento. Per questo motivo, probabilmente, l'utente preferirà sempre altre alternative piuttosto che recarsi in quel luogo e, ad ogni spostamento, ricalcolarne la distanza è un operazione totalmente inutile a meno di essere l'unica soluzione disponibile.

L'idea originale del sistema era creare un software da installare su PDA o cellulari. Oggetti, cioé, con limitata potenza di calcolo e di accesso alla rete (tramite collegamenti wireless quali *WI-FI*, *GPRS*, *UMTS*). E' quindi necessario per lo sviluppo futuro, predisporre delle cache per limitare l'accesso a documenti in rete e sistemi di gestione efficente della ricerca e della memorizzazione di informazioni. Nei test eseguiti il sistema risponde molto bene e la congestione avviene solamente durante la fase di interrogazione remota al *GIS*. Utilizzando una cache locale si aumenterebbe la velocità di risposta ma ridurrebbe lo spazio di memoria utilizzabile. Lo spazio di memoria richiesto per memorizzare le informazioni di un sistema informativo geografico è molto vasto e ciò rende impossibile immagazzinare informazioni di un intera regione, per non parlare di un intero stato. Bisognerà, in futuro, predisporre un meccanismo di configurazione automatica o manuale della cache per velocizzare la ricerca e ottimizzare lo spazio di memoria fisica richiesto.

Appendice - Codice Java

In questa appendice viene presentato il codice che costituisce il sistema, organizzato per pacchetti e classi.

package types

Classe Type

```
package types;

public final class Type
{
    public final static int parola = 1;
    public final static int verbo = 2;
    public final static int pronome = 3;
    public final static int articolo = 4;
    public final static int cifra = 5;
    public final static int aggettivo = 6;
    public final static int preposizione = 7;
    public final static int congiunzione = 8;
    public final static int negazione = 9;
}
```

Class Symbol

```
package types;

public abstract class Symbol
{
    protected String valore;
    protected int tipo;

    public Symbol(String s, int t)
    {
```

```
        tipo = t;
        valore = s;
    }

    public int getType() { return tipo; }
    public String getValore() { return this.valore; }

    public boolean equals(Object o)
    {
        if(o==null || !(o instanceof Symbol))
            return false;
        else if(o == this)
            return true;
        else
        {
            Symbol s = (Symbol)o;
            return tipo==s.tipo && valore==s.valore;
        }
    }

    public String toString() { return new String(this.valore); }
}
```

Class Word

```
package types;

public class Word extends Symbol
{
    private boolean plural = false;

    public Word(String s)
    {
        super(s, Type.parola);
        if(s.endsWith("i"))
            plural=true;
        else
            plural=false;
    }

    public Word(String s, boolean p)
    {
        super(s, Type.parola);
```

```
        plural=p;
    }

    public boolean isPlural() { return plural; }
}
```

Classe Verbo

```
package types;

import databases.DatabaseVerbi;

public class Verbo extends Symbol
{
    protected static DatabaseVerbi db = DatabaseVerbi.getInstance();

    public Verbo(String s)
    {
        super(s,Type.verbo);
        this.addVerboConiugato(s, TempoVerbale.infinito, Persona.nessuna);
        db.addInfinito(this);
    }

    public VerboConiugato addVerboConiugato(String s, int t, int p)
    {
        VerboConiugato v = new VerboConiugato(s,t,p,this);
        db.addConiugato(s, v);
        //verbi.put(s, v);
        return v;
    }

    public static String[] discoverTP(String s)
    {
        String tp[] = new String[3];

        if((s.endsWith("ato") || s.endsWith("uto") || s.endsWith("ito"))  && s.length
        {
            tp[0] = s.substring(0, s.length()-3);
            tp[1]= new String(new Integer(TempoVerbale.passato).toString());
            tp[2] = new String(new Integer(Persona.nessuna).toString());
        }

        else if(s.endsWith("ono") && s.length()>3)
```

```
{
    tp[0] = s.substring(0, s.length()-3);
    tp[1] = new String(new Integer(TempoVerbale.presente).toString());
    tp[2] = new String(new Integer(Persona.noi).toString());
}

else if(s.endsWith("mo") && s.length()>2)
{
    tp[0] = s.substring(0, s.length()-2);
    tp[1] = new String(new Integer(TempoVerbale.presente).toString());
    tp[2] = new String(new Integer(Persona.noi).toString());
}

else if(s.endsWith("te") && s.length()>2)
{
    tp[0] = s.substring(0, s.length()-2);
    tp[1] = new String(new Integer(TempoVerbale.presente).toString());
    tp[2] = new String(new Integer(Persona.voi).toString());
}

else if(s.endsWith("no") && s.length()>2)
{
    tp[0] = s.substring(0, s.length()-2);
    tp[1] = new String(new Integer(TempoVerbale.presente).toString());
    tp[2] = new String(new Integer(Persona.essi).toString());
}

else if(s.endsWith("o") && s.length()>1)
{
    tp[0] = s.substring(0, s.length()-1);
    tp[1] = new String(new Integer(TempoVerbale.presente).toString());
    tp[2] = new String(new Integer(Persona.io).toString());
}

else if(s.endsWith("i") && s.length()>3)
{
    tp[0] = s.substring(0, s.length()-1);
    tp[1] = new String(new Integer(TempoVerbale.presente).toString());
    tp[2] = new String(new Integer(Persona.tu).toString());
}

else if((s.endsWith("a")|| s.endsWith("e")) && s.length()>3)
{
```

```
            tp[0] = s.substring(0, s.length()-1);
            tp[1] = new String(new Integer(TempoVerbale.presente).toString());
            tp[2] = new String(new Integer(Persona.egli).toString());
        }

        else
            return null;

        return tp;
    }

    public static Symbol castVerbo(String s)
    {
        VerboConiugato vc = db.find(s);
        if(vc!= null)
            return vc;

        if(s.endsWith("are")||s.endsWith("ere") || s.endsWith("ire"))
        {
            // Devo istanziare l'infinito e ritornare il coniugato
            new Verbo(s);
            return db.find(s);
        }

        String[] block = discoverTP(s);
        if(block == null)
        {
            return null;
        }

        Verbo v = db.findVerbo(block[0]);
        if(v!=null)
        {
            vc=v.addVerboConiugato(s,Integer.parseInt(block[1]), Integer.parseInt(blo
        }
        return vc;
    }
}
```

Classe VerboConiugato

```
package types;
```

```
public class VerboConiugato extends Symbol
{
    protected Verbo infinito;
    protected int tempo;
    protected int persona;

    public VerboConiugato(String s, int tmp, int p, Verbo inf)
    {
        super(s, Type.verbo);
        tempo = tmp;
        persona = p;
        infinito = inf;
    }

    public boolean equals(Object o)
    {
        if(super.equals(o))
        {
            VerboConiugato s = (VerboConiugato)o;
            return this.tempo==s.tempo;
        }
        else return false;
    }

    public int getTempo()
    {
        return this.tempo;
    }

    public int getPersona()
    {
        return persona;
    }

    public Verbo getVerbo()
    {
        return this.infinito;
    }
}
```

Classe Pronome

```
package types;
```

```
import databases.*;

public class Pronome extends Symbol
{
    protected DatabasePronomi db = DatabasePronomi.getInstance();

    private boolean passivo = false;
    private int persona;

    public Pronome(String s, int ps)
    {
        this(s,ps,false);
    }

    public Pronome(String s, int ps, boolean p)
    {
        super(s,Type.pronome);
        persona = ps;
        passivo = p;
        db.add(this);
    }

    public boolean isPassive() { return passivo; }
    public int getPersona() { return persona; }
}
```

Classe Proposizione

```
package types;

public class Preposizione extends Symbol
{
    public Preposizione(String s)
    {
        super(s, Type.preposizione);
    }
}
```

Classe Numero

```
package types;

public class Numero extends Symbol
```

```
{
        protected float number;

        public Numero(String s)
        {
                super(s, Type.cifra);
                number = Float.parseFloat(s);
        }

        public Numero(float f)
        {
                super((new Float(f)).toString(), Type.cifra);

                number = f;
        }

        public boolean equals(Object o)
        {
                if(super.equals(o))
                {
                        return this.valore == (((Numero)o).valore);
                }
                else return false;
        }

        public static Symbol castNumber(String s)
        {
                float n = 0;
                try
                {
                        n = Float.parseFloat(s);
                }
                catch(NumberFormatException nfe)
                {
                        return null;
                }
                return new Numero(n);
        }

        public float getNumber() { return number; }
}
```

Classe Coniugazione

```
package types;

public class Congiunzione extends Symbol
{
    public Congiunzione(String s)
    {
        super(s, Type.congiunzione);
    }
}
```

Classe Articolo

```
package types;

public class Articolo extends Symbol
{
    private boolean plural=false;

    public Articolo(String s)
    {
        this(s, false);
    }

    public Articolo(String s, boolean p)
    {
        super(s,Type.articolo);
        plural = p;
    }

    public boolean isPlural() { return plural; }
}
```

Classe Negazione

```
package types;

public class Negazione extends Symbol
{
    public Negazione(String s)
    {
        super(s,Type.negazione);
```

```
    }
}
```

Classe Persona

```
package types;

public final class Persona
{
    public final static int nessuna=0;
    public final static int io=1;
    public final static int tu=2;
    public final static int egli=3;
    public final static int noi=4;
    public final static int voi=5;
    public final static int essi=6;
}
```

Classe TempoVerbale

```
public final class TempoVerbale
{
    public final static int infinito=0;
    public final static int presente=1;
    public final static int passato=2;
    public final static int futuro=3;
}
```

Classe Frase

```
package types;

import java.util.List;
import java.util.ListIterator;

public class Frase
{
    private List<Symbol> frase;
    private List<Symbol> soggetto;
    private VerboConiugato verbo;
    private List<Symbol> oggetto;
    private boolean other=false;
    private boolean negative=false;
```

```
public Frase(List<Symbol> f, List<Symbol> s, VerboConiugato v, List<Symbol> o, bo
{
    soggetto=s;
    verbo=v;
    oggetto=o;
    frase = f;
    if(
            frase.size()>3 ||
            (frase.size()>2) && !frase.contains(s) ||
            (frase.size()>2 && oggetto==null) ||
            (frase.size()>1 && oggetto==null && !frase.contains(s))

        )
        other=true;
    else
        other=false;
    this.negative = negative;
}

public List<Symbol> getSoggetto() { return soggetto; }
public Symbol getVerbo() { return verbo; }
public List<Symbol> getOggetto() { return oggetto; }
public List<Symbol> getFrase() { return frase; }

public boolean equals(Object o)
{
    if(o==null || !(o instanceof Symbol))
        return false;
    else if(o == this)
        return true;
    else
    {
        Frase f = (Frase)o;
        return (f.soggetto.equals(this.soggetto) && f.verbo.equals(this.verbo) &&
    }
}

public String toString()
{
    String str1 = new String();
    //String str2 = new String();
    //String str3 = new String();
```

```
        Symbol s = null;

        ListIterator<Symbol> li = (ListIterator<Symbol>) soggetto.iterator();
        str1 = new String("Soggetto: ");
        System.out.print("Soggetto: ");
        while(li.hasNext())
        {
            s = li.next();
            if(s!=null)
            {
                System.out.print(s.getValore() + " ");
                str1.concat(s.getValore() + " ");
            }
        }

        if(verbo!=null)
        {
            System.out.print("Verbo: " + verbo.getValore() + " " + verbo.getPersona()
            //str2 = new String("Verbo: " + verbo.getValore() + " " + verbo.getPerson
        }

        li = (ListIterator<Symbol>)oggetto.iterator();
        str1 = new String("Oggetto: ");

        System.out.print("Oggetto: ");
        while(li.hasNext())
        {
            s = li.next();
            if(s!=null)
            {
                str1.concat(s.getValore() + " ");
                System.out.print(s.getValore() + " ");
            }
        }

        System.out.println("");
        return null;
        //return str1.concat(str2.concat(str3.concat("Altro: " + this.other)));
    }

    public boolean areOthers() { return other; }
    public boolean isNegative() { return negative; }
}
```

package parser

Classe Parser

```
package parser;

import java.util.List;
import java.util.LinkedList;
import java.util.LinkedHashMap;
import java.util.ListIterator;
import java.util.Scanner;

import databases.DatabasePronomi;
import databases.DatabaseVerbi;

import types.*;

public class Parser
{
    private Scanner read = new Scanner(System.in);
    private LinkedHashMap<String, Symbol> dictionary = new LinkedHashMap<String,Symbo

    private DatabasePronomi pronomi = DatabasePronomi.getInstance();
    private DatabaseVerbi dbverbi = DatabaseVerbi.getInstance();

    // Verbi Ausiliari
    private Verbo avere = new Verbo("avere");
    private Verbo essere = new Verbo("essere");

    private List<Symbol> frase = new LinkedList<Symbol>();

    public Parser()
    {
        // Procedura di lettura da file di testo

        // Inserimento Negazioni
        dictionary.put("non", new Negazione("non"));
        ...

        //Inserimento Preposizioni
        dictionary.put("di", new Preposizione("di"));
        ...
```

```
        // Inserimento Congiunzione
        dictionary.put("e", new Congiunzione("e"));
        ...

        // Inserimento Pronomi
        dictionary.put("io", new Pronome("io", Persona.io));
        ...

        // Inserimento Articoli
        dictionary.put("il", new Articolo("il"));
        ...

        // Verbo Avere
        dictionary.put("avere", avere);
        avere.addVerboConiugato("ho", TempoVerbale.presente, Persona.io);
        ...

        // Verbo Essere
        dictionary.put("essere", essere);
        essere.addVerboConiugato("sono", TempoVerbale.presente, Persona.io);
        ...

        // Altri elementi utili per evitare ricerche su web
        ...
    }

    public Frase nextFrase()
    {
        Symbol s=null;
        boolean plural = false;
        while(read.hasNext())
        {
            s = nextSymbol(plural);
            plural=false;
            if(s.getValore().equals("#"))
            {
                return generateFrase();
            }
            else if(s.getValore().equals("exit"))
            {
                return null;
            }
            else
```

```
            {
                if(s.getType()==Type.articolo)
                {
                    System.out.println("Articolo, salto");
                    Articolo a = (Articolo)s;
                    if(a.isPlural())
                        plural = true;
                }
                else
                {
                    if(s.getType()==Type.cifra)
                    {
                        Numero n = (Numero)s;
                        if(n.getNumber()>1)
                            plural=true;
                    }
                    frase.add(s);
                }
            }
        }
        return null;
    }

    private Symbol nextSymbol(boolean plural)
    {
        String s = null;
        s = read.next();

        VerboConiugato Temp = null;

        Symbol sy = dictionary.get(s);
        if(sy!=null)
        {
            if(sy instanceof Verbo)
                sy = dbverbi.find(sy.getValore());

        }

        if(sy==null)
        {
            sy=Verbo.castVerbo(s);
            if(sy!=null)
            {
```

```
        if(sy.getClass().equals(VerboConiugato.class))
        {
            if(((VerboConiugato)sy).getVerbo().equals(essere) || ((VerboConiu
            {
                Temp = (VerboConiugato) sy;
                s = read.next();
                sy = dictionary.get(s);
                if(sy==null)
                {
                    sy=Verbo.castVerbo(s);
                    if(sy!=null && ((VerboConiugato)sy).getPersona()==Persona
                    {
                        VerboConiugato vc = (VerboConiugato)sy;
                        sy = vc.getVerbo().addVerboConiugato(Temp.getValore()
                        vc = (VerboConiugato)sy;
                    }
                    else
                    {
                        frase.add(Temp);
                    }
                }
                else
                {
                    frase.add(Temp);
                    if(sy instanceof Verbo)
                        sy = dbverbi.find(sy.getValore());
                }
            }
        }
    }
    if(sy==null)
    {
        sy=Numero.castNumber(s);
        if(sy==null)
            sy=find(s, plural);
    }
    if(sy.getClass()==VerboConiugato.class)
    {
        VerboConiugato v = (VerboConiugato)sy;
        Verbo verbo = v.getVerbo();

        s = verbo.getValore();
        if(!dictionary.containsKey(s))
```

```
                {
                    dictionary.put(s, verbo);
                }
            }
            else
            {
                dictionary.put(s, sy);
            }
        }
        return sy;
    }

    private Symbol find(String s, boolean plural)
    {
        int id=Type.parola;
        Symbol simbolo = null;
        switch(id)
        {
            case Type.parola:
            {
                simbolo = new Word(s, plural);
                break;
            }
             case 6:
            {
                simbolo = new Numero(s);
                break;
            }
        }
        return simbolo;
    }

    private Frase generateFrase()
    {
        boolean plural = false;
        boolean negative = false;

        List<Symbol> frase_completa = new LinkedList<Symbol>(frase);

        List<Symbol> soggetto = new LinkedList<Symbol>();
        VerboConiugato verbo=null;
        List<Symbol> oggetto = new LinkedList<Symbol>();
```

```
Symbol s = null;
List<Symbol> pre_verbo = new LinkedList<Symbol>();
List<Symbol> post_verbo = new LinkedList<Symbol>();

ListIterator<Symbol> li = (ListIterator<Symbol>) frase.iterator();
while(li.hasNext())
{
    s = li.next();
    if(s.getType()==Type.negazione)
    {
        negative=true;
        continue;
    }
    if(s.getType()==Type.verbo && verbo == null)
    {
        li.remove();
        verbo = (VerboConiugato)s;
        if(verbo.getPersona()>3)
            plural=true;
        continue;
    }
    if(verbo==null)
        pre_verbo.add(s);
    else
        post_verbo.add(s);
}

if(verbo==null)
{
    frase.clear();
    return new Frase(frase_completa,soggetto,verbo,oggetto, negative);
}

li = (ListIterator<Symbol>) frase.iterator();
boolean passive=false;
while(li.hasNext())
{
    s = li.next();
    if(s.getType()==Type.pronome)
    {
        Pronome p = (Pronome)s;
        if(p.getPersona()==verbo.getPersona() && !p.isPassive())
        {
```

```
            soggetto.add(p);
            li.remove();
            break;
        }
        else if(plural && !p.isPassive())
        {
            boolean fault = true;
            if(li.hasNext())
            {
                soggetto.add(p);
                li.remove();
                while(li.hasNext())
                {
                    s = li.next();
                    if(s.getType()==Type.congiunzione)
                    {
                        fault=false;
                        if(!li.hasNext())
                            break;
                        li.remove();
                        s = li.next();
                        soggetto.add(s);
                        li.remove();
                    }
                    else
                        break;
                }
                if(fault)
                    soggetto.clear();
            }
            break;
        }
        else if(p.isPassive())
        {
            oggetto.add(p);
            passive=true;
            li.remove();
            while(li.hasNext())
            {
                s=li.next();
                if(s.getType()==Type.congiunzione)
                {
                    li.remove();
```

```
                    if(li.hasNext())
                    {
                        s=li.next();
                        oggetto.add(s);
                        li.remove();
                    }
                    else break;
                }
                else break;
            }
            break;
        }
    }
}



if(frase.size()!=0)
{
    if(soggetto.size()==0)
    {
        if(passive)
            li = (ListIterator<Symbol>) post_verbo.iterator();
        else
            li = (ListIterator<Symbol>) pre_verbo.iterator();
        while(li.hasNext())
        {
            s=li.next();
            if(s.getType()==Type.parola)
            {
                soggetto.add(s);
                li.remove();

                while(li.hasNext())
                {
                    s = li.next();
                    if(s.getType()==Type.congiunzione)
                    {
                        li.remove();
                        if(li.hasNext())
                        {
                            s = li.next();
                            soggetto.add(s);
```

```
                                li.remove();
                            }
                        }
                        else
                            // Gestione aggettivi o doppi nomi
                    }
                    break;
                }
            }
        }

        if(oggetto.size()==0)
        {
            if(!passive)
                li = (ListIterator<Symbol>) post_verbo.iterator();
            else
                li = (ListIterator<Symbol>) pre_verbo.iterator();
            boolean lastcong = true;
            while(li.hasNext())
            {
                s=li.next();
                if(s.getType()==Type.parola)
                {
                    if(lastcong)
                    {
                        oggetto.add(s);
                    }
                    else
                    li.remove();
                    lastcong=false;
                }
                else if(s.getType()==Type.congiunzione)
                {
                    li.remove();
                    lastcong=true;
                }
            }
        }
    }

    if(soggetto.size()==0)
    {
        soggetto.add(pronomi.getAttivi(verbo.getPersona()));
```

```
        }

        frase.clear();

        return new Frase(frase_completa, soggetto, verbo, oggetto, negative);
    }
}
```

knowledge_members

Classe KnowledgeElement

```
package knowledge_member;

import java.util.Iterator;
import java.util.List;

import databases.DatabaseRelazioni;

public class KnowledgeElement
{
    protected String element;
    protected DatabaseRelazioni db = DatabaseRelazioni.getInstance();
    private static boolean findable = false;

    public KnowledgeElement(String s)
    {
        element = s;
    }

    public void addRelazion(Class c)
    {
        db.addRelation(this, c);
    }

    public void addRelations(List<Class> c)
    {
        db.addRelations(this, c);
    }

    public void addRelations(Class[] c)
    {
```

```
        db.addRelations(this, c);
    }

    public Iterator<Class> getRelations()
    {
        return db.getRelations(this);
    }

    public Iterator<Class> getClassRelations()
    {
        return db.getRelations(this.getClass());
    }

    public String getValue()
    {
        return element;
    }

    public static boolean findable()
    {
        return findable;
    }

    public String toString()
    {
        return new String(element);
    }
}
```

Classe Alimenti

```
package knowledge_member;

public class Alimenti extends Oggetti
{
    public Alimenti(String s)
    {
        super(s);
    }
}
```

Classe Alimentari

```
package knowledge_member;

public class Alimentari extends Supermercati
{
    private static boolean findable = true;

    public Alimentari(String s)
    {
        super(s);
    }

    public static boolean findable()
    {
        return findable;
    }
}
```

package semantica

Classe AnalizzatoreSemantico

```
package semantica;

import java.lang.reflect.Method;
import java.util.Iterator;
import java.util.LinkedHashMap;
import java.util.LinkedList;
import java.util.List;

import databases.DatabaseRelazioni;
import databases.DatabaseRicerche;
import parser.Parser;
import types.*;
import knowledge_member.*;

public class AnalizzatoreSemantico extends Thread
{
    private Parser parser = new Parser();
    private LinkedHashMap<String, KnowledgeElement> bidone = new LinkedHashMap<String

    private DatabaseRelazioni db = DatabaseRelazioni.getInstance();
```

```
private DatabaseRicerche ricerca = DatabaseRicerche.getInstance();

public AnalizzatoreSemantico()
{
    // Lettura dal file di testo dei Knowledge_Element e delle relazioni
    KnowledgeElement k = new Condizioni("fame");
    k.addRelazion(Alimenti.class);
    bidone.put(k.getValue(), k);
    ...

    db.addRelation(Alimenti.class, Alimentari.class);
    ...
}

public void run()
{
    while(true)
    {
        Frase f = parser.nextFrase();
        if(f==null)
        {
            return;
        }
        else
        {
            elaboraFrase(f);
        }
    }
}

private void elaboraFrase(Frase f)
{
    List<Symbol> ric=null;

    if(f.getOggetto().size()==0)
    {
        return;
    }

    if(f.getOggetto().get(0).getType()==Type.pronome)
    {
        Pronome p = (Pronome)f.getOggetto().get(0);
```

```
            if(p.isPassive())
            {
                ric = f.getSoggetto();
            }
            else
                ric=f.getOggetto();
        }
        else
            ric=f.getOggetto();

        Iterator<Symbol> iter = ric.iterator();
        while(iter.hasNext())
        {
            String s = iter.next().getValore();
            KnowledgeElement ke = bidone.get(s);
            if(ke==null)
                continue;

            Iterator<Class> rel = ke.getRelations();
            find(rel);
        }
    }

    public void find(Iterator<Class> rel)
    {
        boolean found = false;
        while(rel.hasNext())
        {
            try
            {
                Class<?> c = rel.next();

                if(!c.getPackage().equals(KnowledgeElement.class.getPackage()))
                    continue;

                Method m = c.getMethod(new String("findable"), null);
                if((Boolean)m.invoke(null, null))
                {
                    String nomeClasse = c.getName();
                    String keyword = nomeClasse.substring(new String("knowledge_membe
                    synchronized(ricerca)
                    {
                        ricerca.add(keyword);
```

```
                        found=true;
                    }
                }
                Iterator<Class> rel2 = db.getRelations(c);
                find(rel2);
            }
            catch (Exception e)
            {
                e.printStackTrace();
            }
        }
        if(found)
        {
            synchronized(ricerca)
            {
                ricerca.signal();
                ricerca.notify();
            }
        }
    }
}
```

package maps

Classe Location

```
package maps;

import java.util.StringTokenizer;

public class Location implements Comparable<Location>
{
    private String name;
    private String address;
    private float distance;

    public Location(String gis_query_string)
    {
        StringTokenizer st = new StringTokenizer(gis_query_string);
        int i=0;

        while(st.hasMoreElements())
        {
```

```
            String str = st.nextToken("(");
            if(i==0)
            {
                address = str.substring(0, str.length()-1);
            }
            else if (i==2)
            {
                name = str.substring(0, str.length()-1);
            }
            i++;
        }
    }

    public String getName() { return name; }

    public String getAddress() { return address; }

    public float getDistance() { return distance; }

    public void setDistance(float d)
    {
        distance = d;
    }

    public int compareTo(Location arg)
    {
        return (int) (distance*10 - ((Location)arg).distance*10);
    }

    public String toString()
    {
        return new String("Nome: " + name + " - Indirizzo: " + address + " - Distanza
    }
}
```

Classe Finder

```
package maps;

import java.util.Iterator;
import databases.DatabaseRicerche;

public class Finder extends Thread
```

```
{
    private String street;
    private String city;

    private int changes=0;

    private DatabaseRicerche db = DatabaseRicerche.getInstance();
    private GISFinder finder = new GoogleMaps();

    public Finder(String s, String c)
    {
        street=s;
        city=c;
    }

    public void newPosition(String street, String city)
    {
        synchronized(db)
        {
            this.street=street.replace(' ', '+');
            this.city=city.replace(' ', '+');
            if(changes==0)
            {
                db.notify();
                db.signal();
            }
            changes = (changes+1) % 3;
        }
    }

    public void run()
    {
        synchronized(db)
        {
            while(!db.signaled())
            {
                try
                {
                    db.wait();
                }
                catch (InterruptedException e)
                {
                    e.printStackTrace();
```

```
                }
            }
            Iterator<String> i = db.iterator();
            while(i.hasNext())
            {
                String s = i.next();
                finder.find(street, city, s);
            }
        }
    }
}
```

Classe GISFinder

```
package maps;

public abstract class GISFinder
{

    public abstract void calcolaDistanza(String via, String citta, Location l);
    public abstract void find(String via, String citta, String categoria);

}
```

Classe GoogleMaps

```
package maps;

import java.io.IOException;
import java.net.*;
import java.util.*;

import databases.DatabaseLuoghi;

public class GoogleMaps extends GISFinder
{
    private URL googlemaps;
    private URLConnection gm;
    private int maxdistance = 10;

    private Scanner sc;
    private DatabaseLuoghi map = DatabaseLuoghi.getInstance();
```

```
public GoogleMaps()
{
}

private void instance(int page, String citta, String categoria)
{
    try
    {
        googlemaps = new URL("http://maps.google.it/maps?f=q&hl=it&geocode=&q=" +
        gm = googlemaps.openConnection();
        sc = new Scanner(gm.getInputStream());
    }
    catch (MalformedURLException e)
    {
        e.printStackTrace();
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }
}

public void calcolaDistanza(String via, String citta, Location l)
{
    String result = null;
    String daddr = l.getAddress().replace(' ', '+');
    String saddr = new String(via.replace(' ', '+') + "+" + citta.replace(' ', '+
    String url = new String("http://maps.google.com/maps?f=d&saddr=" + saddr + "&
    try
    {
        URL u = new URL(url);
        URLConnection urlc = u.openConnection();
        Scanner sc = new Scanner(urlc.getInputStream());
        while(sc.hasNext())
        {
            String st = sc.next();
            if(st.contains("noprint") && st.contains("km"))
            {
                try
                {
                    result = st.substring(st.indexOf("b\\x3e")+5, st.indexOf("x26
                }
                catch(StringIndexOutOfBoundsException sioobe)
```

```
                {
                    result = st.substring(st.indexOf("b>")+2, st.indexOf("&"));
                }
                l.setDistance(Float.parseFloat(result));
                return;
            }
        }
        l.setDistance(-1);
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }
}

public void find(String via, String citta, String categoria)
{
    List<Location> distanze = map.get(categoria);
    if(distanze==null)
    {
        distanze = new LinkedList<Location>();
        map.put(categoria, distanze);
    }
    for(int i=0; ; i++)
    {
        instance(i, citta, categoria);
        String input;
        StringTokenizer st;
        String id;
        Location l;
        while(sc.hasNext())
        {
            input = sc.nextLine();
            if(input.contains("<title>403 Vietato</title>"))
                return;

            if(input.contains("laddr"))
            {
                st = new StringTokenizer(input, "\"");
                boolean found=false;
                while(st.hasMoreTokens())
                {
                    id = st.nextToken();
```

```
                        if(found)
                        {
                            l = new Location(id);
                            calcolaDistanza(via, citta, l);
                            if(l.getDistance()>maxdistance)
                                return;
                            else if(l.getDistance()<0)
                                continue;
                            else
                                distanze.add(l);
                            found=false;
                        }
                        if(id.contains("laddr"))
                            found=true;
                    }
                }
            }
        }
    }
}
```

package databases

Classe DatabaseVerbi

```
package databases;

import java.util.LinkedHashMap;
import java.util.Map;
import types.Verbo;
import types.VerboConiugato;

public class DatabaseVerbi
{
    protected static DatabaseVerbi theInstance;

    private Map<String, Verbo> elenco = new LinkedHashMap<String, Verbo>();
    private Map<String, VerboConiugato> verbi = new LinkedHashMap<String, VerboConiug

    private DatabaseVerbi()    { super(); }

    public static DatabaseVerbi getInstance()
    {
```

```
        if (theInstance == null)
        {
            theInstance = new DatabaseVerbi();
        }
        return (DatabaseVerbi) theInstance;
    }

    public void addInfinito(Verbo v)
    {
        String s = v.getValore();
        elenco.put(s.substring(0, s.length()-3), v);
    }

    public void addConiugato(String s, VerboConiugato v)
    {
        verbi.put(s, v);
    }

    public VerboConiugato find(String s)
    {
        return verbi.get(s);
    }

    public Verbo findVerbo(String s)
    {
        return elenco.get(s);
    }
}
```

Classe DatabasePronomi

```
package databases;

import java.util.LinkedHashMap;
import java.util.Map;

import types.Pronome;

public class DatabasePronomi
{
    protected static DatabasePronomi theInstance;

    protected Map<Integer, Pronome> attivi = new LinkedHashMap<Integer, Pronome>();
```

```
    protected Map<Integer, Pronome> passivi = new LinkedHashMap<Integer, Pronome>();

    private DatabasePronomi() { super(); }

    public static DatabasePronomi getInstance()
    {
        if (theInstance == null)
        {
            theInstance = new DatabasePronomi();
        }
        return (DatabasePronomi) theInstance;
    }

    public void add(Pronome p)
    {
        if(!p.isPassive())
            attivi.put(new Integer(p.getPersona()), p);
        else
            passivi.put(new Integer(p.getPersona()), p);
    }

    public Pronome getAttivi(int i)
    {
        return attivi.get(new Integer(i));
    }

    public Pronome getPassivi(int i)
    {
        return passivi.get(new Integer(i));
    }
}
```

Classe DatabaseRelazioni

```
package databases;

import java.util.Iterator;
import java.util.LinkedHashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import knowledge_member.KnowledgeElement;
```

```
public class DatabaseRelazioni
{
protected static DatabaseRelazioni theInstance;

private Map<Class, List<Class>> relazioni = new LinkedHashMap<Class, List<Class>>();
private Map<KnowledgeElement, List<Class>> relazioni_istanza = new LinkedHashMap<Know

private DatabaseRelazioni() { super(); }

private List<Class> getList(Class c)
{
List<Class> lista = relazioni.get(c);
if(lista==null)
{
lista=new LinkedList<Class>();
relazioni.put(c, lista);
}
return lista;
}

private List<Class> getList(KnowledgeElement c)
{
List<Class> lista = relazioni_istanza.get(c);
if(lista==null)
{
lista=new LinkedList<Class>();
relazioni_istanza.put(c, lista);
}
return lista;
}

public static DatabaseRelazioni getInstance()
{
if (theInstance == null)
{
theInstance = new DatabaseRelazioni();
}
return (DatabaseRelazioni) theInstance;
}

public void addRelation(Class c, Class r)
{
List<Class> lista = getList(c);
```

```
lista.add(r);
}

public void addRelations(Class c, List<Class>l)
{
List<Class> lista = getList(c);
lista.addAll(l);
}

public void addRelations(Class c, Class[] l)
{
List<Class> lista = getList(c);
for(int i=0;i<l.length;i++)
lista.add(l[i]);
}

public Iterator<Class> getRelations(Class c)
{
List<Class> lista = new LinkedList<Class>(getList(c));
return lista.iterator();
}

public void addRelation(KnowledgeElement c, Class r)
{
List<Class> lista = getList(c);
lista.add(r);
}

public void addRelations(KnowledgeElement c, List<Class>l)
{
List<Class> lista = getList(c);
lista.addAll(l);
}

public void addRelations(KnowledgeElement c, Class[] l)
{
List<Class> lista = getList(c);
for(int i=0;i<l.length;i++)
lista.add(l[i]);
}

public Iterator<Class> getRelations(KnowledgeElement c)
{
```

```
List<Class> lista = new LinkedList<Class>(getList(c));
lista.addAll(getList(c.getClass()));
return lista.iterator();
}
}
```

Classe DatabaseRicerche

```
package databases;

import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

public class DatabaseRicerche
{
    protected static DatabaseRicerche theInstance;

    protected List<String> keywords = new LinkedList<String>();
    private boolean signaled=false;

    private DatabaseRicerche() { super(); }

    public static DatabaseRicerche getInstance()
    {
        if (theInstance == null)
        {
            theInstance = new DatabaseRicerche();
        }
        return (DatabaseRicerche) theInstance;
    }

    public void add(String s)
    {
        if(!keywords.contains(s))
        {
            keywords.add(s);
            signaled = true;
        }
    }

    public Iterator<String> iterator()
    {
```

```
        return keywords.iterator();
    }

    public void remove(String s)
    {
        if(keywords.contains(s))
            keywords.remove(s);
    }

    public void clear()
    {
        keywords.clear();
    }

    public void signal()
    {
        signaled = true;
    }

    public boolean signaled()
    {
        if(signaled==true)
        {
            signaled=false;
            return true;
        }
        else return false;
    }
}
```

Classe DatabaseLuoghi

```
package databases;

import java.util.Collections;
import java.util.Iterator;
import java.util.LinkedHashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;

import maps.Location;
```

```
public class DatabaseLuoghi
{
    protected static DatabaseLuoghi theInstance;

    private Map<String, List<Location>> map = new LinkedHashMap<String, List<Location

    private DatabaseLuoghi() { super(); }

    public static DatabaseLuoghi getInstance()
    {
        if (theInstance == null)
        {
            theInstance = new DatabaseLuoghi();
        }
        return theInstance;
    }

    public void put(String s, List<Location> l)
    {
        map.put(s, l);
    }

    public List<Location> get(String s)
    {
        List<Location> l = map.get(s);
        if(l==null)
        {
            l = new LinkedList<Location>();
            map.put(s, l);
            return l;
        }
        Collections.sort(l);
        return l;
    }

    public void printList(String c)
    {
        List<Location> distanze = this.get(c);
        Iterator<Location> i = distanze.iterator();
        while(i.hasNext())
            System.out.println(i.next());
    }
```

```
    public Iterator<Location> iterator(String c)
    {
        List<Location> distanze = this.get(c);
        return distanze.iterator();
    }
}
```

Bibliografia

[1] *Google*. Website. `http://www.google.com`.

[2] *GoogleMaps*. Website. `http://maps.google.com`.

[3] *ibiblio.org: the public's library and digital archive*. Website. `http://www.ibiblio.org`.

[4] *Verbnet: a lexicon that groups verbs based on their semantic/syntactic linking behavior*. Website. `http://verbs.colorado.edu/~mpalmer/projects/verbnet.html`.

[5] *Wikipedia*. Website. `http://www.wikipedia.org`.

[6] *Java Speech API Programmer's Guide*. Website, 1998. `http://java.sun.com/products/java-media/speech/forDevelopers/jsapi-guide/`.

[7] Ugo Chirico. *Schemi di Rappresentazione della Conoscenza*. Website. `http://www.ugosweb.com/Documents/Articles/Rap-Con.html`.

[8] Prof. Mauro Di Manzo. *Appunti del corso di Informatica Teorica*, 2006.

[9] Prof. Matteo Matteucci. *Appunti del corso di Ingegneria della Conoscenza*, 2007. `http://home.dei.polimi.it/bonarini/Didattica/IngegneriaConoscenza/`.

[10] Prof. George A. Miller. *WordNet: a lexical database for the English language*. Website. `http://wordnet.princeton.edu/`.

[11] Christos H. Papadimitriou. *Computational Complexity*. Addison Wesley, 1994.

[12] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prantice Hall, 2003.

[13] Prof. Armando Tacchella. *Appunti del Corso di Linguaggi e Traduttori*, 2007.

[14] Charles E. Leiserson Thomas H. Cormen, Dartmouth College and Ronald L. Rivest. *Introduction to Algorithms (2nd Edition)*. McGraw-Hill, 2002.

[15] Alan M. Turing. Computing machinery and intelligence. *Mind*, 1950.

Indice analitico