Università degli Studi di Bologna

Facoltà di Ingegneria

Corso di Laurea in
Ingegneria Informatica

Ingegnerizzazione di uno strumento di supporto alle teorie etichettate per l'interprete tuProlog in ambiente NetBeans

Tesi in
Linguaggi e Traduttori

Candidato:
Luca Coccini Gailli

Relatore:
Chiar.mo Prof. Enrico Denti

Correlatori:
Dott. Ing. Giulio Piancastelli
Dott. Ing. Alessandro Ricci

Anno Accademico 2005/2006
Sessione II

“I computer sono incredibilmente veloci, accurati e stupidi.
Gli uomini sono incredibilmente lenti, inaccurati e intelligenti.
Insieme sono una potenza che supera l'immaginazione.”

Albert Einstein

Indice

Introduzione.....1
Capitolo 1 – tuProlog e la gestione delle teorie etichettate.....3
1.1 Introduzione a tuProlog.....3
1.2 Architettura logica di tuProlog.....4
1.2.1 La base di conoscenza.....6
1.2.2 L'ambiente operativo.....8
1.2.3 Il motore inferenziale.....10
1.2.4 Gestione delle librerie.....13
1.3 Supporto alle teorie etichettate.....14
1.3.1 L'etichettatura delle teorie.....14
1.3.2 Gestione delle teorie etichettate.....15
1.3.3 Funzionalità di gestione delle teorie etichettate.....17
Capitolo 2 – Caratteristiche dell'ambiente NetBeans.....21
2.1 Introduzione all'ambiente NetBeans.....21
2.2 Caratteristiche fondamentali dell'ambiente NetBeans.....23
2.2.1 Moduli NetBeans.....23
2.2.2 System Filesystem e XML Layer.....26
2.2.3 Recupero di un componente registrato nel System Filesystem.....27
2.2.4 Altri meccanismi per la registrazione di componenti nel System Filesystem.....28
2.3 Componenti principali che caratterizzano l'ambiente NetBeans.....29
2.3.1 File object e filesystem.....30
2.3.2 Data object e data loader.....31
2.3.3 Nodi.....34
2.3.4 Explorer view e explorer manager.....35
2.3.5 Azioni.....37

2.3.6 Window component e window manager..... 39
2.3.7 Editor..... 43
2.4 Definizione di nuovi tipi di file..... 45
Capitolo 3 – Analisi del problema.....47
3.1 Gestione della risoluzione di query..... 47
3.2 Gestione delle teorie multiple.....48
3.3 Gestione della persistenza delle teorie..... 56
3.4 Gestione di un generico elenco di teorie..... 59
3.5 Gestione delle librerie.....60
3.6 Adattamento del motore tuProlog al plug-in..... 62
3.7 Architettura logica..... 63
3.8 Piano di lavoro.....66
3.9 Nomi delle versioni dei prototipi.....67
Capitolo 4 – Primo prototipo..... 69
4.1 Progettazione del primo prototipo..... 69
4.1.1 Collaborazione: *esegui query temporanea*..... 70
4.1.2 Collaborazione: *visualizza informazioni provenienti dal motore tuProlog*..... 73
4.1.3 Interazione: *avviamento della risoluzione di una query*.....74
4.1.4 Interazione: *terminazione della risoluzione di una query*..... 76
4.1.5 Interazione: *ricerca di un'ulteriore soluzione*.....77
4.1.6 Interazione: *accettazione di una soluzione*..... 78
4.1.7 Interazione: *notifica di un evento di output*..... 79
4.1.8 Interazione: *notifica di un evento interno al motore tuProlog*..... 79
4.1.9 Progettazione dei partecipanti: *PrologEngine*..... 80
4.1.10 Progettazione dei partecipanti: *QueryControlPanel e QueryActionController*..... 81
4.1.11 Progettazione dei partecipanti: *OutputView* e *SpyView*.....86
4.1.12 Progettazione dei partecipanti: *StartAction*, *StopAction*, *NextAction* e *AcceptAction*..... 87
4.1.13 Inizializzazione dei partecipanti..... 89
4.2 Architettura del primo prototipo.....90
4.3 Implementazione del primo prototipo..... 92
4.3.1 Implementazione del *PrologEngine*..... 92

4.3.2 Implementazione del *QueryControlPanel* e del *QueryActionController*..... 93
4.3.3 Implementazione dell'*OutputView* e dello *SpyView*..... 97
4.3.4 Implementazione delle azioni di gestione della risoluzione di una query..... 99
4.3.5 Integrazione dei partecipanti nell'ambiente NetBeans e loro inizializzazione..... 101
4.4 Collaudo del primo prototipo..... 104
Capitolo 5 – Secondo prototipo..... 105
5.1 Progettazione del secondo prototipo..... 105
5.1.1 Collaborazione: *modifica elenco delle teorie attive*..... 106
5.1.2 Collaborazione: *aggiorna motore tuProlog*..... 108
5.1.3 Collaborazione: *esegui query*..... 109
5.1.4 Collaborazione: *importazione teoria dal filesystem locale*..... 111
5.1.5 Collaborazione: *esportazione teoria verso il filesystem locale*..... 113
5.1.6 Collaborazione: *estrai una teoria attiva*..... 115
5.1.7 Collaborazione: *estrai la teoria attiva globale*..... 116
5.1.8 Interazione: *impostazione di un'unica teoria attiva*..... 118
5.1.9 Interazione: *rimozione di una teoria attiva*..... 118
5.1.10 Interazione: *rimozione di tutte le teorie attive*..... 119
5.1.11 Interazione: *avviamento della risoluzione di una query*..... 119
5.1.12 Interazione: *terminazione della risoluzione di una query*..... 120
5.1.13 Interazione: *ricerca di un'ulteriore soluzione*..... 121
5.1.14 Interazione: *accettazione di una soluzione*..... 121
5.1.15 Interazione: *importazione teoria dal filesystem locale*..... 121
5.1.16 Interazione: *esportazione teoria verso il filesystem locale*..... 122
5.1.17 Interazione: *estrazione di una teoria attiva*..... 123
5.1.18 Interazione: *estrazione della teoria attiva globale*..... 124
5.1.19 Progettazione dei partecipanti: *ActiveTheoryList, TheoryManager e PrologEngine*..... 124
5.1.20 Progettazione dei partecipanti: *ActiveTheoryListView*..... 129
5.1.21 Progettazione dei partecipanti: *azioni di gestione delle teorie attive*..... 131
5.1.22 Progettazione dei partecipanti: *LocalTheoryLoader e LocalTheorySaver*..... 133

5.1.23 Progettazione dei partecipanti: *AddTheoryAction e ExportTheoryAction*.....134
5.1.24 Inizializzazione dei partecipanti..... 134
5.2 Architettura del secondo prototipo..... 135
5.3 Implementazione del secondo prototipo.....137
5.3.1 Strutturazione dei package.....137
5.3.2 Implementazione dell'*ActiveTheoryList*, del *TheoryManager* e del *PrologEngine*.....138
5.3.3 Implementazione dell'*ActiveTheoryListView*.....145
5.3.4 Implementazione delle azioni relative alla gestione delle teorie attive..... 150
5.3.5 Implementazione del *LocalTheoryLoader* e del *LocalTheorySaver*..151
5.3.6 Implementazione dell'*ExportTheoryAction*..... 153
5.3.7 Implementazione dell'*AddTheoryAction*.....154
5.3.8 Implementazione del *LocalTheoryChooser*.....155
5.3.9 Implementazione del *LocalFileSystem*..... 156
5.4 Definizione del tipo di file denominato “Prolog Theory”..... 157
5.5 Collaudo..... 158
5.6 Modifiche al motore tuProlog..... 160
Capitolo 6 – Ulteriori prototipi e sviluppi futuri.....163
6.1 Progettazione del terzo prototipo.....163
6.1.1 Collaborazione: importazione teoria dall'elenco delle teorie disponibili..... 164
6.1.2 Collaborazione: esportazione teoria verso l'elenco delle teorie disponibili..... 166
6.1.3 Collaborazione: estrai una o più teorie attive..... 167
6.1.4 Collaborazione: estrai la teoria attiva globale.....168
6.1.5 Collaborazione: estrai tutte le teorie..... 168
6.1.6 Collaborazione: modifica elenco delle teorie disponibili..... 168
6.1.7 Collaborazione: importazione teoria.....171
6.1.8 Collaborazione: esportazione teoria..... 172
6.1.9 Collaborazione: apri teoria da file..... 173
6.1.10 Collaborazione: salva teoria su file.....173
6.1.11 Collaborazione: aggiorna teoria.....173
6.1.12 Collaborazione: aggiorna tutte le teorie.....174

6.1.13 Collaborazione: crea teoria..... 174
6.1.14 Collaborazione: cancella teoria.....176
6.1.15 Collaborazione: modifica contenuto teoria disponibile..... 178
6.1.16 Interazione: importazione teoria dall'elenco delle teorie disponibili..... 180
6.1.17 Interazione: esportazione teoria verso l'elenco delle teorie disponibili..... 181
6.1.18 Interazione: modifica elenco delle teorie disponibili.....181
6.1.19 Interazioni: importazione teoria e esportazione teoria.....182
6.1.20 Interazione: crea teoria..... 182
6.1.21 Interazione: cancella teoria..... 183
6.1.22 Interazione: modifica contenuto teoria disponibile..... 184
6.1.23 Progettazione dei partecipanti: *AvailableTheoryList* e *ActiveTheoryList*..... 185
6.1.24 Progettazione dei partecipanti: *AvailableTheoryListView* e *ActiveTheoryListView*..... 187
6.1.25 Progettazione dei partecipanti: azioni di gestione delle teorie attive..... 190
6.1.26 Progettazione dei partecipanti: azioni di gestione delle teorie disponibili..... 190
6.1.27 Progettazione dei partecipanti: *TheoryEditor* e *TheoryEditorSupport*..... 193
6.1.28 Progettazione dei partecipanti: altri partecipanti..... 194
6.2 Progettazione del quarto prototipo..... 195
6.2.1 Collaborazione: verifica correttezza query..... 195
6.2.2 Interazione: verifica correttezza query..... 197
6.2.3 Progettazione dei partecipanti: *PrologParser*, *QueryTextField* e *ParseAction*.....198
6.2.4 Reingegnerizzazione: *ActiveTheoryList* e *TheoryManager*.....199
6.2.5 Reingegnerizzazione: *ActiveTheoryListView* e *ActiveTheoryListNode*.....202
6.2.6 Reingegnerizzazione: *QueryControlPanel*.....202
6.3 Progettazione del quinto prototipo..... 205
6.3.1 Collaborazione: carica libreria nel motore tuProlog.....205
6.3.2 Collaborazione: rimuovi libreria dal motore tuProlog.....207

6.3.3 Interazione: carica libreria nel motore tuProlog..... 209
6.3.4 Interazione: rimuovi libreria dal motore tuProlog..... 210
6.3.5 Progettazione dei partecipanti: *LibraryManager* e *LibraryList*.....210
6.3.6 Progettazione dei partecipanti: *LibraryListView*..... 213
6.3.7 Progettazione dei partecipanti: *LoadLibraryAction* e *UnloadLibraryAction*.....215
6.4 Indicazioni sulla realizzazione del sesto prototipo.....216
6.5 Sviluppi futuri.....217
6.5.1 Miglioramento relativo alla integrazione del plug-in con l'ambiente NetBeans.....217
6.5.2 Miglioramento relativo alla organizzazione delle teorie logiche etichettate.....218
6.5.3 Introduzione di un nuovo tipo di progetto per l'ambiente NetBeans. 219
6.5.4 Riusabilità delle classi che costituiscono il nucleo del plug-in..... 220
6.6 Questioni ancora aperte..... 221
Conclusioni.....225
Appendice A – Inversion of Control.....227
A.1 Inversion of Control e il design pattern Dependency Injection.....227
A.2 Design pattern Service Locator..... 230
A.3 Design pattern Cookie..... 232
A.4 Differenze tra i design pattern analizzati.....236
Appendice B – NetBeans API.....237
Bibliografia..... 249
Ringraziamenti..... 251

Introduzione

NetBeans è un ambiente aperto per lo sviluppo di software, basato sulla piattaforma Java e caratterizzato da un'architettura modulare. L'ambiente NetBeans è sostanzialmente una piattaforma che offre funzionalità e componenti di base per la realizzazione di applicazioni complete e di strumenti di supporto a determinate tecnologie. Sono proprio gli strumenti di supporto, comunemente detti plug-in, che conferiscono a NetBeans la sua architettura modulare, dato che possono essere facilmente integrati in questo ambiente estendendolo con nuove caratteristiche.

L'obiettivo di questa tesi è l'analisi, progettazione e prima prototipazione di un plug-in che integri l'interprete tuProlog all'interno dell'ambiente NetBeans, utilizzando gli strumenti di supporto alla programmazione tipici della piattaforma. tuProlog è un interprete Prolog che si contraddistingue per la sua leggerezza, la quale lo rende facilmente integrabile; come NetBeans, è basato sulla piattaforma Java. Inoltre tuProlog è anche estendibile: infatti è possibile aggiungere apposite librerie scritte in Java che gli permettono di allargare il suo campo di azione.

La realizzazione del plug-in sarà focalizzata sulle caratteristiche specifiche di tuProlog, in particolare sulla gestione e sull'utilizzo contemporaneo di teorie logiche etichettate: il motore tuProlog infatti è in grado di utilizzare e gestire teorie logiche multiple, identificate da apposite etichette, le quali sono costituite da semplici stringhe rappresentanti il nome della teoria. Le teorie logiche etichettate conferiscono maggiore flessibilità alla gestione delle teorie caricate nel motore tuProlog, consentendo l'esecuzione di interventi mirati sulla teoria logica complessiva.

Inoltre, lo sviluppo del plug-in dovrà prevedere anche la creazione di apposite funzionalità per la persistenza delle teorie logiche: una teoria logica è una entità dinamica, soggetta a espansioni e contrazioni durante la risoluzione di un goal, pertanto è importante poter memorizzare il suo contenuto al termine

dell'elaborazione di una query; conseguentemente si dovrà indagare sul rapporto tra evoluzione del contenuto delle teorie e i meccanismi di persistenza offerti dall'ambiente NetBeans.

Quindi il plug-in dovrà essere progettato per essere scalabile relativamente alle dinamiche di un sistema Prolog multi-teoria, sia dal punto di vista della interazione con l'utente, sia dal punto di vista relativo alla evoluzione e persistenza delle teorie logiche etichettate.

Per utilizzare al meglio gli strumenti offerti da NetBeans si dovrà effettuare un'esplorazione di questo ambiente, inizialmente cercando di ottenere una visione generale del sistema e successivamente una visione molto più approfondita dei particolari meccanismi che in prima analisi risulteranno essere interessanti per la realizzazione dello strumento di supporto per l'interprete tuProlog. Sarà inoltre molto importante effettuare un'analisi del problema accurata ed esaustiva prima di procedere alla progettazione del sistema, approfondendo gli aspetti relativi alla risoluzione di query, alla gestione delle teorie etichettate, alla persistenza delle teorie logiche e a come adattare il motore tuProlog al plug-in.

Al fine di rendere più mirato lo sviluppo dello strumento di supporto, la creazione dell'intero sistema verrà suddivisa in più fasi, ordinate secondo la priorità degli obiettivi da raggiungere, al termine di ognuna delle quali si produrrà un prototipo coerente dotato di determinate caratteristiche e funzionalità. Ogni fase successiva alla prima si baserà sui risultati ottenuti nella fase precedente, aggiungendo ulteriori caratteristiche e funzionalità, ottenendo al suo termine un nuovo prototipo coerente. In questo modo sarà possibile verificare il corretto comportamento del sistema al termine di ogni fase e mantenere sotto controllo l'impegno necessario alle eventuali correzioni da svolgere. Infine, tale articolazione del processo di sviluppo favorirà l'eliminazione dei difetti tra una fase e l'altra, consentendo di avere un sistema ragionevolmente funzionante al termine di ogni fase dello sviluppo.

Per la realizzazione di questo plug-in verrà dunque impiegato un processo di sviluppo ciclico, dove ad ogni iterazione sarà ottenuto un artefatto funzionante; in tal modo sarà possibile creare una serie di prototipi dosando l'aumento della loro complessità tra una fase e l'altra.

Infine saranno illustrati eventuali sviluppi futuri dello strumento di supporto realizzato.

Capitolo 1

tuProlog e la gestione delle teorie etichettate

In questo capitolo verrà descritto il sistema tuProlog, analizzandone l'architettura [Ben05] e il supporto alla gestione delle teorie etichettate [Nan06].

1.1 Introduzione a tuProlog

tuProlog è un interprete Prolog basato sulla piattaforma Java; esso si contraddistingue per la sua realizzazione minimale, la quale lo rende particolarmente leggero; proprio questa caratteristica di leggerezza consente a tuProlog di poter essere utilizzato all'interno di applet, applicazioni Web e come componente di un framework. Inoltre, tuProlog si distingue anche per la sua estendibilità: infatti è possibile aggiungere apposite librerie di predicati e funtori scritte in Java per allargarne il campo di azione.

Ma ciò che rende tuProlog veramente innovativo è la sua capacità di accedere allo spazio degli oggetti Java. Questo è stato reso possibile da una specifica libreria di predicati: la *JavaLibrary*. Tale libreria costituisce di fatto un ponte tra il paradigma della programmazione logica e il paradigma della programmazione orientata agli oggetti, consentendo ad una teoria logica di interagire con gli oggetti. Inoltre, un'applicazione Java può invocare il motore inferenziale di tuProlog come un oggetto qualunque; in questo modo è possibile risolvere agevolmente problemi non deterministici in ambiente Java. Consegue che l'integrazione tra lo spazio degli oggetti Java e lo spazio delle teorie logiche è

bidirezionale. Infine, si noti che un'applicazione Java può utilizzare contemporaneamente più istanze indipendenti di motori tuProlog, ognuna dotata del proprio stato e della propria base di conoscenza.

1.2 Architettura logica di tuProlog

L'architettura di tuProlog è scomponibile in tre parti principali:

- la base di conoscenza;
- l'ambiente operativo;
- il motore inferenziale.

La base di conoscenza è costituita dalle librerie e da una teoria logica dinamica. Una libreria definisce una collezione di predicati, una teoria logica interna e funtori richiamabili all'interno delle teorie; la teoria logica dinamica definisce una collezione di clausole che il motore inferenziale utilizza nel tentativo di unificare le query di cui si demanda la risoluzione, ma differisce dalle teorie logiche interne alle librerie a causa della sua dinamicità, che consiste nella possibilità di aggiungere nuove clausole in coda a quelle esistenti e nella possibilità di ritrattare una qualunque clausola. Inoltre, a partire dalla versione 1.4 di tuProlog, è anche possibile caricare in un motore tuProlog più teorie logiche contrassegnate da una etichetta; in tal caso la teoria dinamica è costituita dall'insieme delle clausole di tutte le teorie etichettate caricate nel motore inferenziale. La teoria logica dinamica viene ottenuta dopo un processo di compilazione della teoria o delle teorie caricate; le librerie vengono coinvolte nella compilazione delle teorie a causa della loro teoria logica interna.

Il motore inferenziale e l'ambiente operativo sono strettamente correlati tra loro: infatti il primo è l'automa risolutore, mentre il secondo ne rappresenta lo stato. Per eseguire l'elaborazione di una query, il motore inferenziale utilizza la teoria compilata e le librerie caricate oltre alle informazioni date dall'ambiente operativo.

Osservando con maggiore dettaglio l'architettura generale sopra delineata, essa risulta essere maggiormente frastagliata e può essere scomposta in:

- *base di conoscenza*, principalmente gestita dalla classe `alice.tuprolog.TheoryManager`;

- *ambiente operativo*, identificabile con la classe `alice.tuprolog.RTContext`;
- *motore inferenziale*, implementato dalla classe `alice.tuprolog.Prolog`.

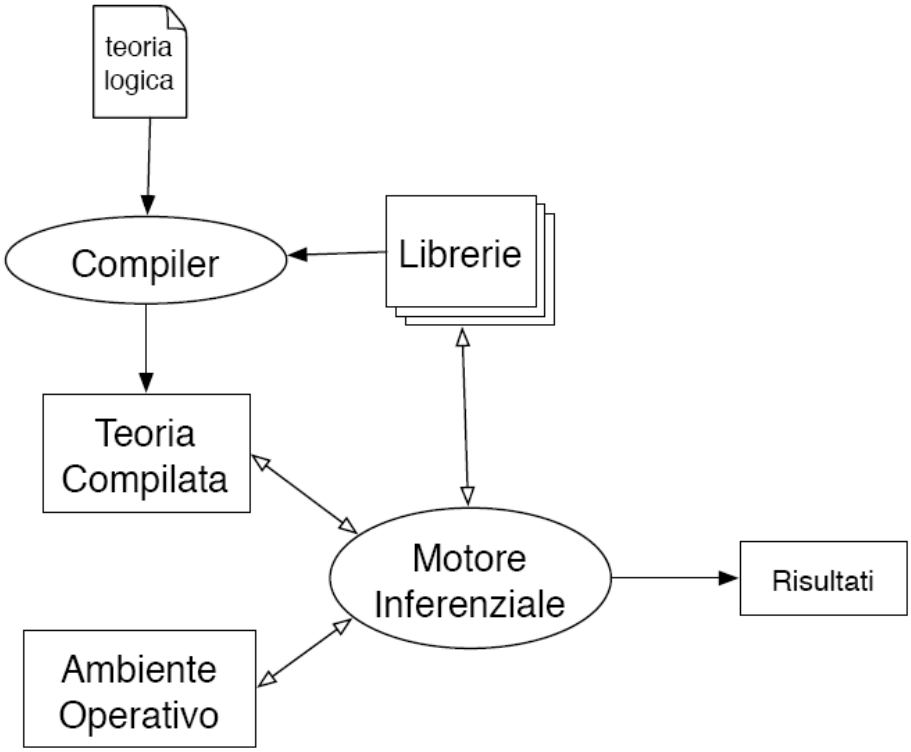


Figura 1.1: Struttura logica di tuProlog

I termini (`alice.tuprolog.Term`) che costituiscono la teoria compilata vengono utilizzati dal motore inferenziale per eseguire l'elaborazione delle query. Un termine può essere una variabile (`alice.tuprolog.Var`), un numero (`alice.tuprolog.Number`) o una struttura (`alice.tuprolog.Struct`). Un numero può essere specializzabile in un intero (`alice.tuprolog.Int`), in un intero lungo (`alice.tuprolog.Long`), in un reale (`alice.tuprolog.Float`) o in un reale a doppia precisione (`alice.tuprolog.Double`); infine, una struttura è composta da uno o più termini, realizzando di fatto delle strutture dati ad albero composte da termini.

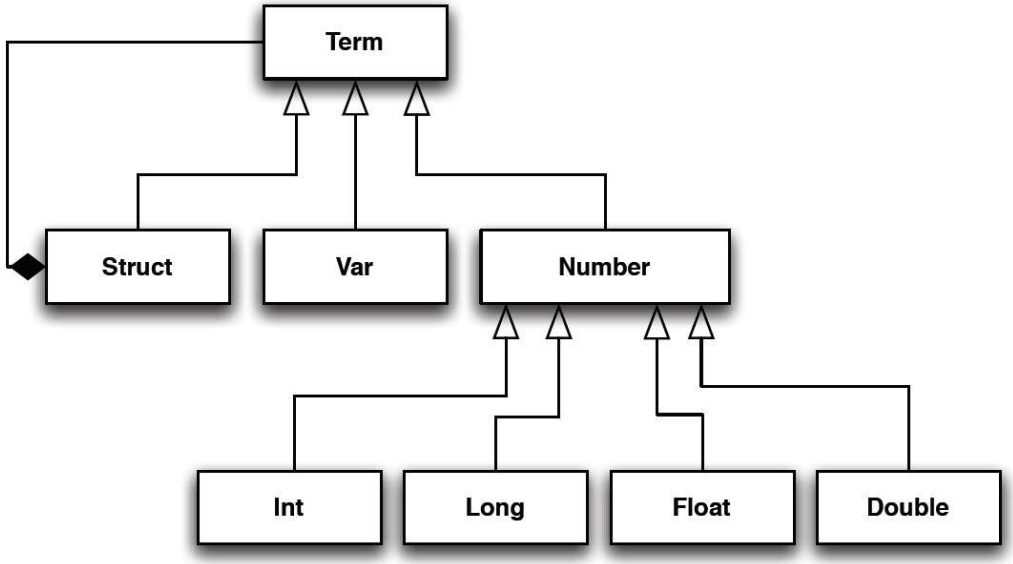


Figura 1.2: Tassonomia dei termini di tuProlog

1.2.1 La base di conoscenza

La base di conoscenza coincide con l'astrazione relativa alla teoria logica compilata, rappresentata nella figura 1.1, ed è suddivisa in diverse classi che contengono e manipolano le informazioni della teoria logica dinamica. L'astrazione di teoria logica compilata è stata introdotta allo scopo di caratterizzare il funzionamento di tuProlog, il quale non contempla il concetto di teoria compilata ma piuttosto quello di gestore della teoria dinamica, dato dal TheoryManager.

In base alla figura 1.1 la teoria logica è ottenuta da un compilatore partendo dalla teoria logica dinamica e dalle librerie; anche il compilatore è un'astrazione: il suo effetto è ottenuto dall'azione combinata del parser, il quale analizza e scompone la teoria logica dinamica, e del gestore delle teorie (TheoryManager), il quale gestisce le nuove strutture dati ottenute dall'azione del parser, rappresentanti clausole e termini.

Il principale gestore delle strutture dati relative alla teoria compilata è il TheoryManager, ma l'entità che ha l'effettivo controllo della base di conoscenza è il motore tuProlog, il quale è in grado di modificare la teoria compilata generando nuove asserzioni, dichiarando nuovi funtori, aggiungendo librerie e modificando dei flags.

La classe che implementa il TheoryManager è dotata di tre campi fondamentali:

- `alice.util.LinkedList transientClauseList`: lista contenente le clausole asserite tramite il predicato *assert* e che devono essere rimosse in caso di backtracking;
- `HashMap clauseDBase`: tabella hash contenente liste ordinate di clausole unificanti allo stesso modo, ovvero dotate dello stesso operatore e della stessa arità.

Le clausole all'interno del TheoryManager sono realizzate tramite istanze della classe `alice.tuprolog.ClauseInfo`, contenenti informazioni utili in fase di elaborazione. Tale classe è dotata di alcuni campi contenenti informazioni di interesse per il motore inferenziale:

- `Struct clause`: struttura che rappresenta la clausola sotto forma di termine;
- `boolean dynamic`: valore booleano che indica se la clausola è statica o dinamica, ovvero se essa è stata dichiarata in una libreria oppure se è stata dichiarata in una teoria logica o asserita; una clausola dinamica, a differenza di una statica, può essere rimossa durante l'elaborazione;
- `String libName`: stringa che indica il nome della libreria di appartenenza della clausola; se la clausola non appartiene ad una libreria, tale campo assume valore `null`;
- `boolean stillValid`: valore booleano che indica se una clausola è stata ritrattata o meno; un valore corrispondente a `false` indica che la clausola è stata ritrattata, quindi non è più valida; le clausole non più valide vengono eliminate al termine delle risoluzioni di goal.

Inoltre, la classe `TheoryManager` fornisce dei metodi per accedere al proprio contenuto informativo, ma la creazione e la modifica della teoria logica o delle teorie logiche rimane sotto il controllo del motore tuProlog, ovvero della classe `Prolog`. Infatti, tale classe per caricare una teoria richiama direttamente il metodo `void consult(Theory,boolean,boolean,String)` della classe `TheoryManager`, il quale svolge le seguenti azioni:

- inizializzazione della lista contenente le clausole asserite dal predicato *assert*;

- analisi di ogni clausola della teoria logica;
- esegue il parsing delle clausole della teoria logica richiamando il metodo `boolean parseClause(Struct,LinkedQueue,boolean,String)` per ognuna di esse.

Se la chiamata del metodo `consult` non va a buon fine, viene sollevata un'eccezione, ma soltanto dopo aver richiamato il metodo `boolean transEnd(false)`, in modo tale da annullare i cambiamenti apportati. Se invece termina con successo, le clausole della teoria caricata diventano parte della base di conoscenza.

1.2.2 L'ambiente operativo

Nell'ambito dell'architettura generale di tuProlog rappresentata in figura 1.1, l'ambiente operativo è un'astrazione che mantiene le strutture dati necessarie al motore inferenziale per la risoluzione del goal richiesto. Le sue principali caratteristiche sono raccolte nella classe `RTContext`, che svolge le sue funzioni avvalendosi anche della collaborazione del motore inferenziale (implementato dalla classe `Prolog`) e utilizzando informazioni contenute nelle variabili.

La classe `RTContext` incapsula una lista in grado di contenere i riferimenti ad altre istanze della classe `RTContext`, in modo da poter rappresentare l'albero decisionale.

In Prolog le variabili hanno lo scope limitato alla clausola, ovvero, in una clausola successiva, lo stesso nome non si riferisce alla stessa variabile. Analogamente, ogni clausola processata dal motore inferenziale mantiene un record di attivazione, implementato da un'istanza della classe `RTContext`; quindi, se una clausola è ricorsiva, la prima istanza della classe `RTContext` contiene le informazioni relative alla prima elaborazione della clausola, mentre le seguenti istanze della classe `RTContext` contengono le informazioni relative alle successive elaborazioni della stessa clausola. In questo modo è possibile unificare le variabili senza interferire tra una chiamata e l'altra della medesima clausola.

Dato che il nome di una variabile è univoco solamente nell'ambito di una clausola, ai fini della computazione è necessario attribuire un identificatore globale ad ognuna di esse. Pertanto, ogni variabile è caratterizzata dal campo `private int time`, il quale le identifica in modo univoco; infatti, creata una variabile e

inizializzato il corrispondete campo `int time`, alla creazione della successiva variabile il valore di tale campo corrisponde a quello della variabile precedentemente creata, incrementato di una unità. Allo stesso modo si distinguono le varie clausole: pertanto è presente un campo `private int time` anche nella classe `RTContext`; in tal caso il valore del campo viene impostato dopo che sono state rinominate tutte le variabili non anonime interne alla clausola.

Al fine di eliminare le unificazioni non più utili dopo un backtracking, ogni variabile è dotata del campo `private int mark`; infatti una operazione di backtracking necessita di conoscere il valore del campo `int mark` a partire dal quale deve deunificare le variabili. Esso viene inizializzato nel momento in cui una variabile viene unificata e il suo meccanismo di attribuzione del valore è del tutto identico a quello del campo `int time`.

La classe `RTContext` è costituita dai seguenti campi fondamentali:

- `int time,mark`: campi di cui si è appena discusso;
- `alice.tuprolog.GoalInfo currentGoal`: sub-goal correntemente in fase di elaborazione;
- `alice.util.LinkedList goalsToEval`: lista di sub-goal non elaborati;
- `alice.util.LinkedList termsExecuted`: lista dei sub-goal già eseguiti e con alternative aperte.

Si noti che nella classe `RTContext` ogni sub-goal è rappresentato da una istanza della classe `alice.tuprolog.GoalInfo`. Inoltre, la classe `RTContext` dispone di metodi per la manipolazione di dati all'interno delle liste:

- `void saveFGoal()`: inserisce il sub-goal correntemente in fase di elaborazione nella lista dei sub-goal non elaborati a cui fa riferimento il campo `alice.util.LinkedList goalsToEval`;
- `void loadFGoal()`: (*fetch*) preleva un sub-goal dalla lista a cui si riferisce `alice.util.LinkedList goalsToEval` e lo fa diventare il sub-goal correntemente in fase di elaborazione;
- `void saveTGoal()`: (*store*) inserisce un goal appena eseguito nella lista dei sub-goal già eseguiti, ovvero in `alice.util.LinkedList termsExecuted`; se nel goal eseguito erano presenti riferimenti ad

istanze della classe `RTContext`, allora esistono punti di scelta alternativa;

- `void loadTGoal()`: in caso di backtracking, estrae un goal dalla lista dei sub-goal già eseguiti, ovvero da `alice.util.LinkedList termsExecuted`; se nel goal estratto sono presenti riferimenti ad istanze della classe `RTContext`, allora era presente un punto di scelta alternativa.

Il motore inferenziale utilizza il metodo `void loadFGoal()` per eseguire una operazione di *fetch* di un sub-goal da elaborare, mentre utilizza il metodo `void saveTGoal()` per eseguire una operazione di *store* di un sub-goal appena elaborato. Se si escludono eventuali operazioni di backtracking, le quali comportano la chiamata del metodo `void loadTGoal()`, il motore inferenziale elabora un goal eseguendo sequenzialmente delle operazioni di *fetch* e *store* di sub-goal presenti in istanze della classe `RTContext`.

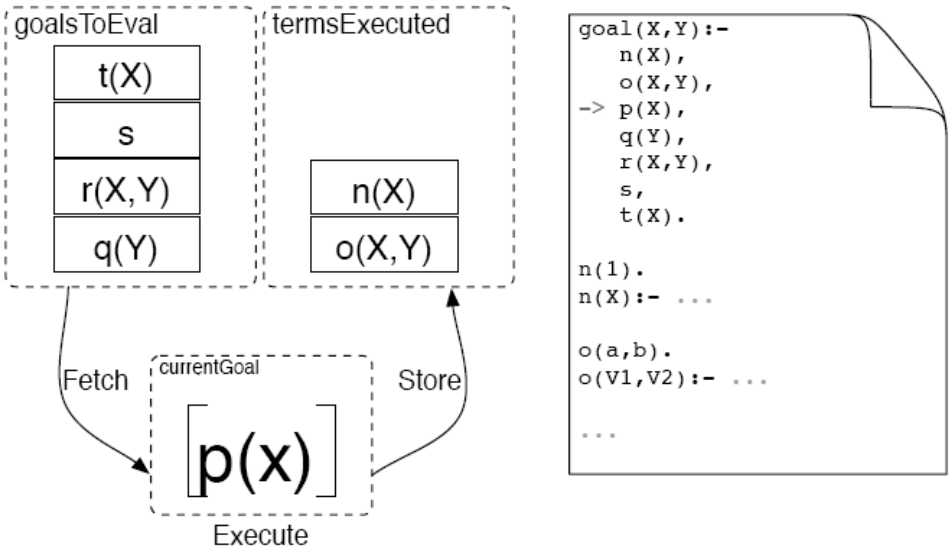


Figura 1.3: Processo di caricamento ed esecuzione utilizzando strutture RTContext

1.2.3 Il motore inferenziale

Il motore inferenziale di tuProlog è implementato dalla classe `Prolog`, la quale è realizzata in modo tale da consentire l'istanziazione di più motori tuProlog indipendenti. Essa offre una collezione di metodi a visibilità *public* per il controllo del motore e per la gestione della teoria dinamica ed è dotata di uno stato interno per la memorizzazione delle informazioni relative alla risoluzione di query. Inoltre, dispone di alcuni metodi a visibilità *private* che svolgono specifiche azioni, tra i quali i più importanti sono:

- `void init(RTContext,Term,int,int,int)`: inizializza il motore inferenziale al fine di eseguire la valutazione del goal specificato dal parametro di tipo `Term`, utilizzando l'ambiente operativo specificato dal parametro di tipo `RTContext`;
- `int eval()`: esegue la valutazione del goal specificato nella precedente chiamata del metodo `void init(RTContext,Term,int,int,int)` e può restituire tre valori distinti: `TRUE` se la valutazione ha avuto successo e non esistono alternative alla soluzione individuata, `TRUE_CP` se la valutazione ha avuto successo ed esistono alternative alla soluzione individuata, infine `FALSE` se la valutazione è fallita;
- `int backTracking()`: esplora il il primo tra i punti di scelta ancora aperti, restituendo gli stessi valori del metodo `int eval()`.

La risoluzione di un goal viene avviata dall'oggetto che detiene il controllo del motore tuProlog, richiamando il metodo `SolveInfo solve(Term)` oppure il metodo `SolveInfo solve(String)`; entrambe questi metodi restituiscono un'istanza della classe `alice.tuprolog.SolveInfo`, la quale rappresenta lo stato della risoluzione del goal. Al suo interno, oltre al valore relativo all'esito della valutazione, contiene anche l'insieme di sostituzioni relativo alle variabili contenute nel goal, ovviamente soltanto se è stata trovata una soluzione.

Al termine dell'elaborazione dei un goal viene richiamato il metodo `void solveEnd()`, il quale, nel caso in cui sia stata individuata una soluzione svuota il campo `transientClauseList` del TheoryManager, mentre nel caso in cui la risoluzione del goal sia fallita esegue un backtracking; prima di terminare, il metodo `void solveEnd()` rimuove le clausole invalidate durante la computazione richiamando il metodo `void optmize()` del TheoryManager.

Nel caso in cui esista la possibilità di una ulteriore soluzione, l'oggetto che detiene il controllo del motore tuProlog può richiamare il metodo `SolveInfo solveNext()`, al fine di eseguire la ricerca di una ulteriore soluzione.

Il processo di risoluzione è principalmente gestito dal metodo `int eval(RTContext)` relativo alla classe `Prolog` (evocato dal metodo `int eval()` relativo alla medesima classe); tale metodo richiama iterativamente il metodo `private int stepE(RTContext)`, finché questo non restituisce il valore `CONT`. Gli altri possibili valori che `int stepE(RTContext)` può

restituire sono `TRUE`, `TRUE_CP` e `FALSE`; nel caso in cui restituisca il valore `FALSE`, sarà richiamato il metodo `private int stepB(RTContext)` relativo alla classe `Prolog`. Tale metodo esegue di fatto il backtracking dello stato del motore inferenziale, ed in grado di restituire gli stessi valori del metodo `int stepE(RTContext)`. Nel caso in cui restituisca il valore `CONT`, la ricerca di una soluzione proseguirà come descritto sopra, richiamando il metodo `int stepE(RTContext)`.

I metodi `int stepE(RTContext)` e `int stepB(RTContext)` esplorano l'albero decisionale in modo opposto: `int stepE(RTContext)` ricerca eventuali soluzioni effettuando un'esplorazione depth-first con derivazione left-most, mentre `int stepB(RTContext)` esegue un backtracking risalendo l'albero decisionale alla ricerca di eventuali punti a scelta multipla.

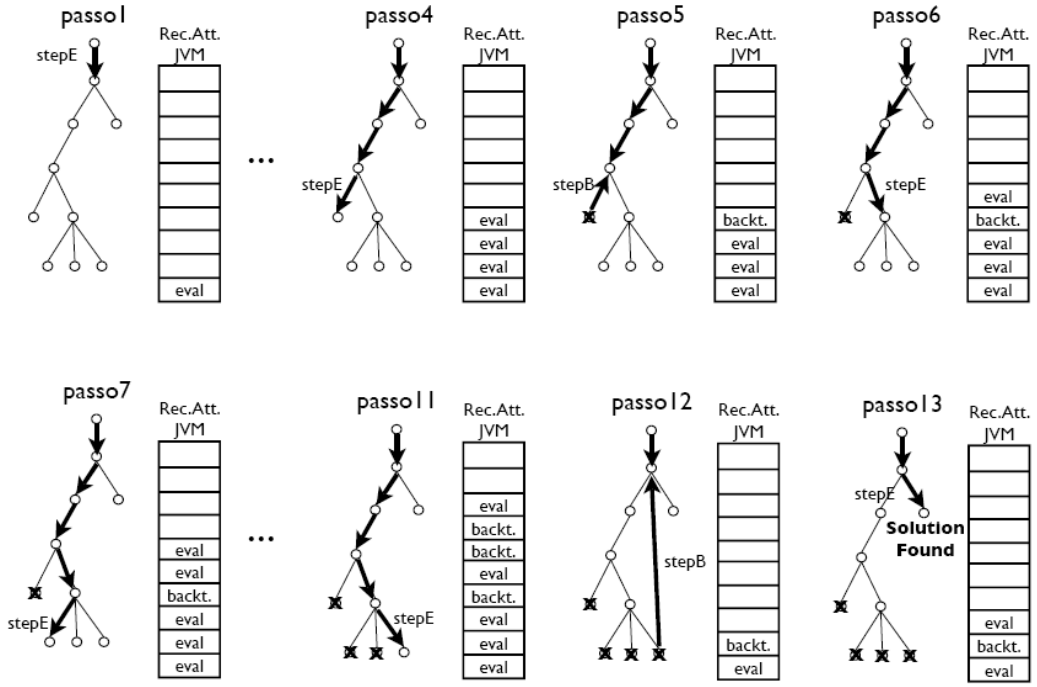


Figura 1.4: Esplorazione dell'albero decisionale durante il processo di risoluzione

In questo modo, nello stack della *Java Virtual Machine* (JVM) sono presenti una serie di riferimenti relativi ai metodi `int eval(RTContext)` e `int stepE(RTContext)`; nei casi in cui il metodo `int stepE(RTContext)` restituisce `FALSE`, dal metodo `int stepE(RTContext)` viene richiamato il metodo `int stepB(RTContext)` il quale, nel caso in cui siano presenti punti a scelta multipla, richiama il metodo `private int backTracking()`; quindi

nello stack della JVM, al termine di un processo di risoluzione, saranno presenti solamente riferimenti ai metodi `int eval(RTContext)`, `int stepE(RTContext)` e `int backTracking()`.

1.2.4 Gestione delle librerie

Al fine di espanderne le funzionalità, tuProlog è stato dotato della caratteristica di poter aggiungere apposite librerie. Una libreria può arricchire la base di conoscenza di un motore tuProlog aggiungendo nuovi predicati, nuovi funtori ed anche le clausole relative ad una propria teoria logica interna; le entità in essa contenute sono statiche, ovvero non possono essere rimosse dalla base di conoscenza, a meno che non venga rimossa l'intera libreria.

Il caricamento di una libreria nel motore tuProlog viene eseguito utilizzando il metodo `alice.tuprolog.Library loadLibrary(String)` relativo alla classe `Prolog`. La teoria logica inclusa in una libreria viene aggiunta alla base di conoscenza richiamando il metodo `void consult(Theory, boolean, boolean, String)` relativo alla classe `TheoryManager`.

Le clausole definite nella teoria logica interna ad una libreria hanno un comportamento statico, ovvero non possono essere ritrattate, mentre le clausole definite da una qualsiasi altra teoria logica sono dinamiche, ovvero possono essere rimosse dalla base di conoscenza. Per distinguere tra una clausola dinamica definita da una teoria logica e una clausola statica definita da una libreria, si utilizza il campo `String libName`, il quale se assume valore `null` indica che la clausola è statica, altrimenti è dinamica (vedi sezione 1.2.1).

Come già detto, le librerie introducono nuovi predicati nella base di conoscenza; per definire un nuovo predicato si specifica un metodo nella classe che rappresenta la libreria, utilizzando le seguenti regole:

- deve avere visibilità *public*;
- deve restituire un valore booleano;
- il nome del metodo deve corrispondere al nome del predicato, seguito dal carattere *underscore* ("_") e da un valore intero che ne indica l'arità;
- il numero di parametri deve corrispondere al valore numerico che indica l'arità del predicato.

Ad esempio, una signature che rispetta tutte queste regole può essere:

```
public boolean java_object_3(Term,Struct,Term).
```

1.3 Supporto alle teorie etichettate

Il corrente sistema di gestione delle teorie etichettate consente di intervenire sulla teoria logica complessiva in modo mirato, garantendo flessibilità alla sua manipolazione. Grazie all'etichette associate ad ogni teoria logica è possibile risalire alle singole clausole di una determinata teoria.

Nella versione di tuProlog priva del supporto per la gestione delle teorie etichettate, viene utilizzata un'unica teoria dinamica, il cui ampliamento è possibile soltanto aggiungendo in fondo ad essa nuove clausole. Tale soluzione si è dimostrata poco flessibile in scenari in cui la presenza di un'unica teoria pone limiti alla necessità di modificare o eliminare parte della teoria logica. In tali scenari, la soluzione migliore era la cancellazione completa della teoria e conseguente inizializzazione con le nuove clausole. Per questo motivo si è giunti all'attuale sistema di gestione della teoria dinamica, basato sulle teorie etichettate.

1.3.1 L'etichettatura delle teorie

L'etichetta di una teoria è costituita da una semplice stringa, la quale rappresenta il nome della teoria. In un insieme di teorie logiche etichettate caricate nel motore tuProlog possono esistere teorie dotate dello stesso nome; ciò consente di aggiungere alla teoria dinamica complessiva parti distinte di una stessa teoria logica in momenti diversi di una elaborazione.

Una etichetta consente di attribuire una determinata semantica alla teoria a cui si riferisce, inserendola in uno specifico contesto. Dato che la semantica di una teoria è unica e particolare, non non ha senso permettere di cambiare il contesto di una teoria logica; pertanto, nel momento della creazione di una teoria è necessario specificare la relativa etichetta, la quale giustamente non potrà essere sostituita. Nel caso in cui il nome di una nuova teoria logica non fosse specificato, verrebbe automaticamente attribuito il nome predefinito "*default*", permettendo di associare comunque un nome ad una teoria e garantendo la compatibilità con le modalità precedenti di creazione delle teorie.

L'etichetta di una teoria è implementata dal campo `private String name`

della classe `Theory`; inoltre, al fine di rendere immutabile il nome di una teoria, la classe `Theory` non è dotata di metodi per la modifica del campo `String name`. Pertanto il nome di una teoria può essere specificato solamente quando viene richiamato un costruttore della classe `Theory`; nel caso in cui esso non venisse specificato, il campo `String name` assumerà il valore "`default`".

1.3.2 Gestione delle teorie etichettate

Al fine di gestire in modo corretto e flessibile l'insieme delle teorie etichettate e le relative clausole, il motore tuProlog deve mantenere un elenco delle teorie caricate e le clausole inserite nella base di conoscenza devono riportare il nome della teoria di cui fanno parte. Ovviamente è necessario anche mantenere una corrispondenza tra teoria logica e relative clausole al fine di poter eseguire le operazioni di gestione delle teorie logiche, soprattutto per poter eseguire la cancellazione di una specifica teoria.

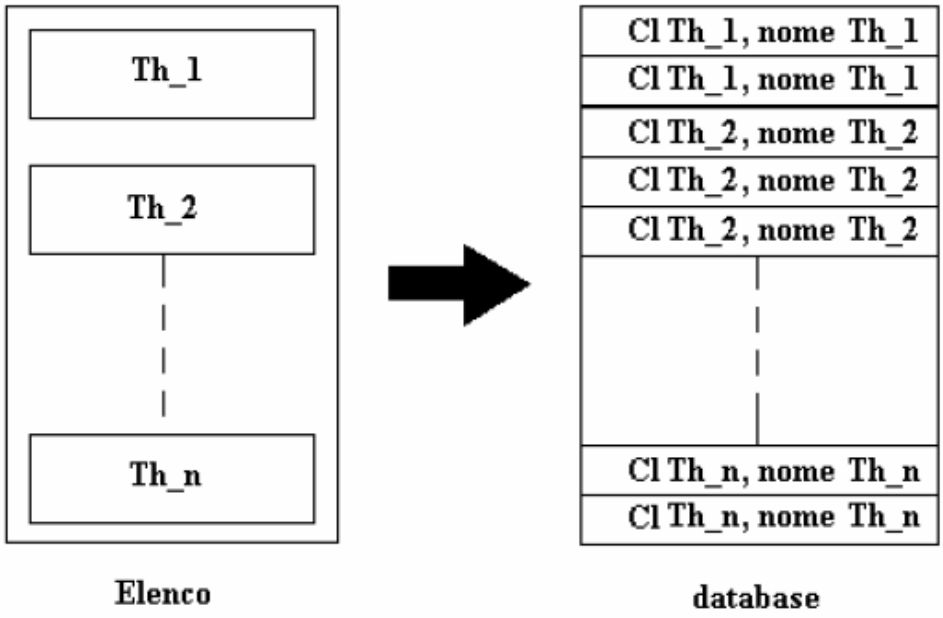


Figura 1.5: Elenco delle teorie e database delle clausole

In questo modo è possibile eseguire le seguenti azioni sull'insieme delle teorie caricate:

- *aggiunta di una teoria logica*: la nuova teoria viene inserita in coda all'elenco delle teorie etichettate e le nuove clausole vengono aggiunte in

fondo alla base di conoscenza;

- *rimozione di una teoria logica*: rimozione della teoria dall'elenco delle teorie etichettate e invalidazione delle corrispondenti clausole;
- *recupero di una teoria logica*: l'elenco delle teorie etichettate fornisce direttamente la teoria cercata.

L'eventuale aggiunta di una nuova clausola tramite il predicato *assert* ha come conseguenza l'inserimento di questa in coda alla base di conoscenza, e dovrebbe anche implicare la sua associazione alla teoria denominata *default*; analogamente il predicato *retract* elimina l'ultima clausola della base di conoscenza corrispondente al termine indicato e dovrebbe anche cancellare il riferimento contenuto nella teoria a cui appartiene la clausola eliminata.

Si noti come l'aggiunta di clausole e teorie logiche avvenga sempre in coda al rispettivo elenco: in questo modo si mantiene l'ordinamento temporale sia delle clausole, sia delle teorie.

L'elenco delle teorie logiche caricate nel motore tuProlog è mantenuto da una lista di istanze della classe `Theory` e corrisponde al campo `private ArryList currentTheories` della classe `TheoryManager`.

Al fine di garantire una corrispondenza biunivoca tra teorie logiche e clausole, una teoria logica deve essere in grado di risalire alle clausole che la compongono, mentre una clausola deve disporre di un identificatore univoco della teoria logica di appartenenza.

Per fare in modo che esista una corrispondenza tra teorie logiche e clausole, distinguendo eventuali teorie logiche omonime, si deve utilizzare un identificatore globale delle teorie logiche. Questo identificatore è costituito da una stringa, formata dal nome della teoria logica concatenato con il valore di un contatore gestito dal TheoryManager. Pertanto la classe `TheoryManager` è stata dotata del campo `static private int ThCount`, il quale viene incrementato ogni volta che viene aggiunta una nuova teoria logica. Per associare una teoria logica al corrispondente identificatore globale viene utilizzata una tabella hash, individuata dal campo `private HashMap clauseNames` della classe `TheoryManager`, alla quale sono utilizzate come chiavi le teorie logiche, come valori le stringhe che formano i corrispondenti identificatori globali.

Per avere la corrispondenza tra clausole e teoria logica, quando il TheoryManager trasforma le singole clausole in istanze della classe `ClauseInfo`, al valore del campo `String libName` (vedi sezione 1.2.4) viene attribuito l'identificatore globale della teoria logica al posto del valore `null`. In questo modo ogni clausola sa a quale specifica teoria logica essa appartiene.

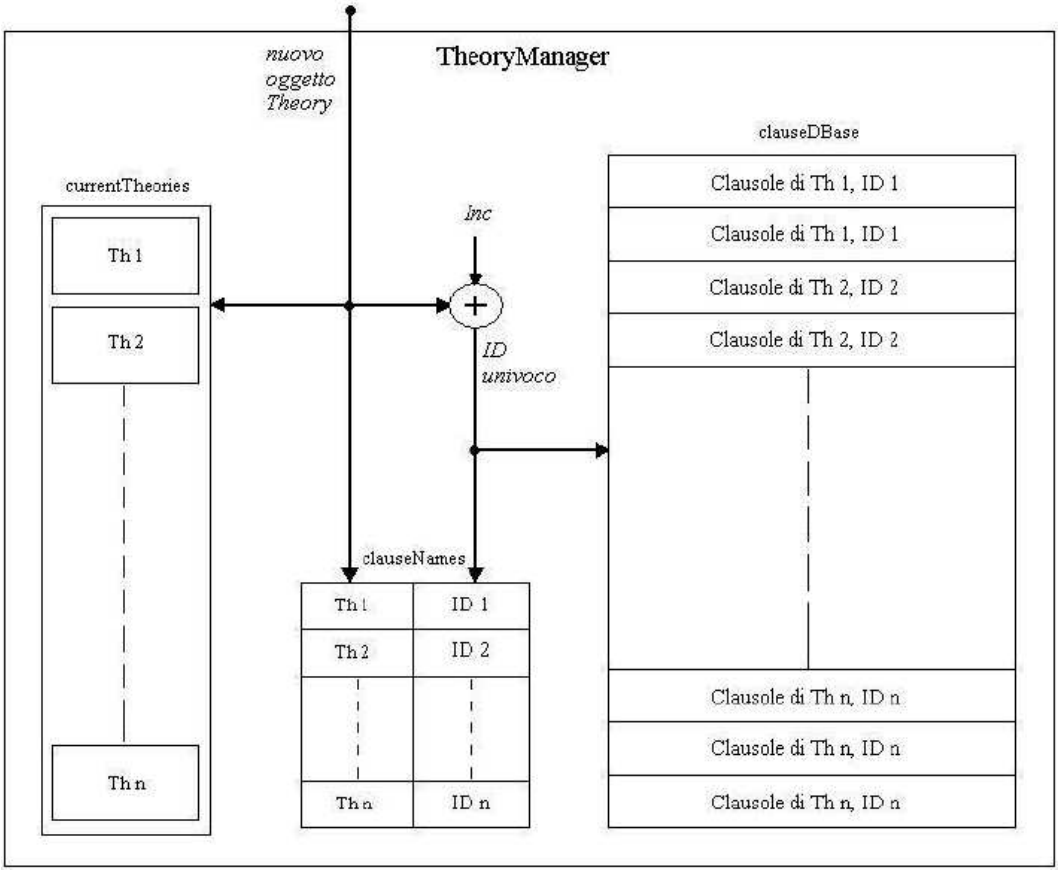


Figura 1.6: Corrispondenza tra clausole e relativa teoria logica

1.3.3 Funzionalità di gestione delle teorie etichettate

Le funzionalità di gestione delle teorie etichettate consentono di manipolare l'elenco delle teorie logiche caricate nel motore tuProlog; dato che tale elenco è incapsulato nel TheoryManager, tali funzionalità sono implementate da specifici metodi della classe `TheoryManager`; esse consistono in:

- *aggiunta di una teoria logica* (`void addTheory(Theory)`): la nuova teoria logica viene inserita in coda all'elenco delle teorie caricate nel motore, al fine di preservarne l'ordine temporale; i passi per l'esecuzione dell'inserimento della nuova teoria logica sono:

  1. aggiunta dell'istanza della classe `Theory`, che implementa la nuova

teoria logica, all'array referenziato dal campo `ArrayList currentTheories` della classe `TheoryManager`, il quale realizza l'elenco delle teorie caricate nel motore tuProlog;

2. incremento del contatore delle teorie logiche, realizzato dal campo `int ThCount` relativo alla classe `TheoryManager`;
3. creazione dell'identificatore univoco della nuova teoria concatenando il nome della teoria al valore del contatore;
4. aggiunta nella tabella hash data dal campo `HashMap clauseNames` della classe `TheoryManager` del nuovo identificatore ottenuto al punto precedente, utilizzando come chiave l'istanza della classe `Theory` che implementa la nuova teoria logica;
5. invocazione del metodo `void consult(Theory,boolean, boolean,String)` relativo alla classe `TheoryManager` al fine di inserire le clausole della nuova teoria logica nella base di conoscenza, specificando l'identificatore globale della nuova teoria;

- *impostazione di un'unica teoria logica* (`void setTheory(Theory)`): la teoria specificata viene impostata come unica teoria logica caricata nel motore; i passi per imporre una teoria logica sono identici a quelli per l'inserimento di una nuova teoria, tranne per il fatto che prima l'elenco delle teorie caricate e la tabella hash relativa agli identificatori globali devono essere svuotate, sostituendo al campo `ArrayList currentTheories` e al campo `HashMap clauseNames` delle nuove istanze, prive di contenuto;
- *recupero di tutte le teorie logiche caricate raggruppandole in un'unica teoria* (`Theory getWholeTheory()`): estrae dalla base di conoscenza tutte le clausole dinamiche, raccogliendole in un'unica teoria dotata di un nome speciale; la teoria restituita corrisponde alla teoria dinamica complessiva del motore tuProlog;
- *rimozione di una teoria logica* (`void removeTheory(String)`): rimuove l'ultima teoria logica dotata del nome specificato; viene rimossa solamente l'ultima teoria in modo tale da avere una gestione LIFO relativamente alle teorie logiche omonime; nel caso in cui il nome specificato non corrisponda ad alcuna teoria, viene sollevata un'eccezione;

i passi per la rimozione di una teoria logica sono:

1. recupero e rimozione dell'ultima teoria logica dotata del nome specificato dall'elenco delle teorie caricate, ovvero dall'array referenziato dal campo `ArrayList currentTheories`;
2. recupero e rimozione dell'identificatore globale della teoria logica da rimossa dall'elenco delle teorie caricate dalla tabella hash relativa al campo `HashMap clauseNames`;
3. invalidazione di tutte le clausole contenute nella base di conoscenza ed appartenenti alla teoria logica da rimuovere, imponendo il valore `false` al campo `boolean stillValid` di tutte le istanze relative alla classe `ClauseInfo` contenute nella tabella hash referenziata dal campo `HashMap clauseDBase` della classe `TheoryManager` e tali che risultino associate all'identificatore globale della teoria logica da rimuovere;
4. eliminazione delle clausole non più valide richiamando il metodo `void optimize()` della classe `TheoryManager`;

- *recupero dei nomi delle teorie logiche* (`String[] getTheoryNames()`): restituisce un array di stringhe contenente l'elenco dei nomi delle teorie logiche che costituiscono l'elenco delle teorie caricate; l'array di stringhe restituito ha le stesse dimensioni dell'array delle teorie caricate incapsulato nel TheoryManager, pertanto può riportare anche nomi ripetuti, relativi a teorie omonime;
- *recupero di tutte le teorie logiche dotate di un particolare nome* (`Theory[] getTheories(String)`): restituisce un array contenente tutte le teorie logiche dotate del nome specificato; per costruire l'array da restituire saranno analizzate tutte le teorie contenute nell'array delle teorie caricate incapsulato nel TheoryManager (campo `ArrayList currentTheories`); nel caso il nome specificato non corrisponda ad alcuna teoria, verrà sollevata un'eccezione;
- *recupero delle posizioni relative alle teorie logiche dotate di un particolare nome nell'elenco delle teorie caricate* (`int[] getTheoryPos(String)`): restituisce un array di numeri interi, i quali indicano le posizioni delle teorie logiche dotate del nome specificato

all'interno dell'elenco delle teorie caricate; come nel caso precedente, per costruire l'array da restituire verrà analizzato l'intero contenuto dell'array relativo alle teorie caricate incapsulato nel TheoryManager; anche in tal caso se il nome specificato non corrisponde ad alcuna teoria, verrà sollevata un'eccezione.

Oltre al TheoryManager, anche il motore tuProlog offre funzionalità di gestione delle teorie logiche, il quale utilizza comunque il TheoryManager per poterle realizzare. Pertanto la classe `Prolog` presenta appositi metodi per poter offrire le seguenti funzionalità:

- *aggiunta di una teoria logica*:

  ```
  void addTheory(Theory);
  ```

- *impostazione di un'unica teoria logica*:

  ```
  void setTheory(Theory);
  ```

- *recupero di tutte le teorie logiche caricate raggruppandole in un'unica teoria*:

  ```
  Theory getWholeTheory();
  ```

- *eliminazione di tutte le clausole contenute nelle teorie logiche caricate*:

  ```
  void clearTheory();
  ```

- *rimozione di una teoria logica*:

  ```
  void removeTheory(String);
  ```

- *recupero dei nomi delle teorie logiche*:

  ```
  String[] getTheoryNames();
  ```

- *recupero di tutte le teorie logiche dotate di un particolare nome*:

  ```
  Theory[] getTheories(String);
  ```

- *recupero delle posizioni relative alle teorie logiche dotate di un particolare nome nell'elenco delle teorie caricate*:

  ```
  int[] getTheoryPos(String).
  ```

Capitolo 2

Caratteristiche dell'ambiente NetBeans

In questo capitolo saranno descritti i principali aspetti che caratterizzano l'ambiente NetBeans; non sarà una trattazione esaustiva, ma finalizzata a conoscere le proprietà relative agli elementi dell'ambiente che verranno utilizzati per la realizzazione del plug-in.

2.1 Introduzione all'ambiente NetBeans

NetBeans è un progetto open-source dedito a fornire prodotti per lo sviluppo di software che vengono incontro alle esigenze degli sviluppatori [NbOrg]. I prodotti principali del progetto NetBeans sono *NetBeans Platform* [NbPlat] e *NetBeans IDE* [NbIDE].

NetBeans Platform è il progetto di base di NetBeans; si tratta di un framework in grado di fornire le funzionalità di base di un'applicazione complessa (comunemente detta *rich-client application*) a livello di interfaccia grafica, gestione dei file e configurazione; non realizza alcunché di specifico.

NetBeans IDE è il progetto principale di NetBeans e si basa sulla NetBeans Platform; si tratta di un ambiente di sviluppo integrato (IDE – Integrated Development Environment) progettato per la realizzazione di applicazioni Java, in grado di supportare anche altri linguaggi. Quindi si tratta di una specifica *rich-client application* pensata esplicitamente per lo sviluppo di software basato sulla piattaforma Java.

Le funzionalità specifiche, che permettono al NetBeans IDE di realizzare una *rich-client application* basata sulla NetBeans Platform, sono implementate da specifici plug-in, detti *moduli*. E' possibile realizzare una qualunque *rich-client application* installando una collezione di specifici moduli nella NetBeans Platform. Quindi, fondamentalmente, l'architettura di una *rich-client application* basata sulla NetBeans Platform è formata da una collezione di plug-in.

Con il termine *ambiente NetBeans* (o più sinteticamente soltanto *NetBeans*) si intende l'infrastruttura che fornisce le funzionalità basilari di un'applicazione complessa, assieme alle ulteriori funzionalità aggiunte da eventuali moduli. La NetBeans Platform è il framework che implementa l'ambiente NetBeans dotato delle sole funzionalità di base, mentre il NetBeans IDE è un'estensione della NetBeans Platform, che aggiunge funzionalità specifiche per la realizzazione di applicazioni in ambito Java.

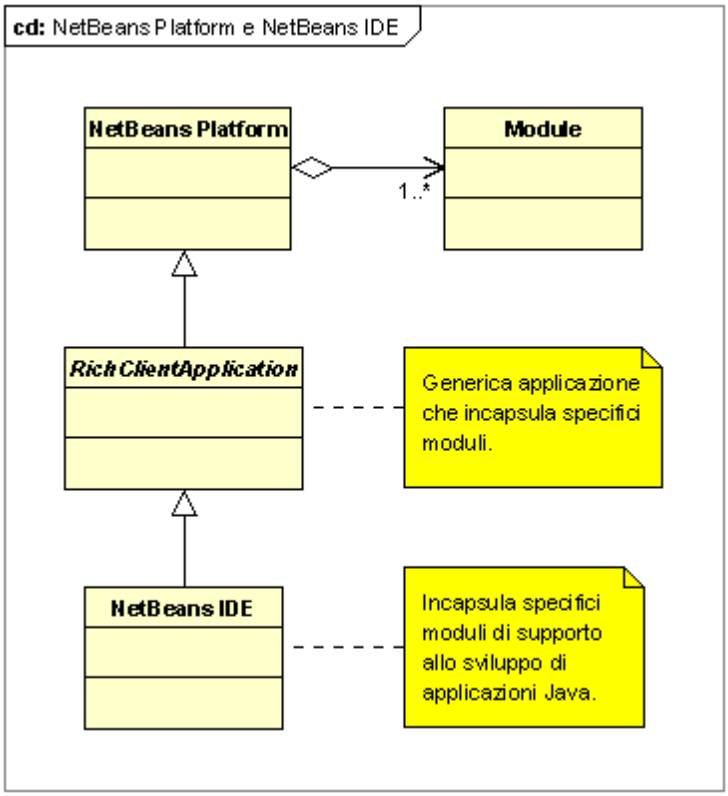


Figura 2.1: Strutturazione della NetBeans Platform e del NetBeans IDE

Riepilogando, la principale caratteristica di NetBeans consiste nel permettere lo sviluppo di intere applicazioni o singole funzionalità implementandone solamente la *business logic* (ovvero la logica specifica che contraddistingue l'applicazione), utilizzando, per il resto, l'infrastruttura offerta dall'ambiente e le sue eventuali

estensioni; la business logic è ovviamente racchiusa in appositi moduli.

2.2 Caratteristiche fondamentali dell'ambiente NetBeans

Le caratteristiche fondamentali dell'ambiente NetBeans riguardano i meccanismi interni che permettono la gestione, installazione e individuazione dei componenti che costituiscono l'ambiente. In particolare, l'elemento fondamentale di NetBeans è il *System Filesystem*, un repository che permette di referenziare i componenti che costituiscono l'ambiente. I componenti dell'ambiente sono contenuti nei già citati moduli, i quali strutturalmente sono dei particolari archivi Java. Ogni modulo contiene anche un documento XML particolare, denominato *XML Layer*, la cui funzione consiste nell'indicare i componenti che devono essere accessibili tramite il *System Filesystem*. Quando viene installato un modulo nel sistema le informazioni contenute nello *XML Layer* vengono inserite nel *System Filesystem*; quindi, da un punto di vista logico, il *System Filesystem* è costituito da tutti gli *XML Layer* relativi ai moduli installati nell'ambiente.

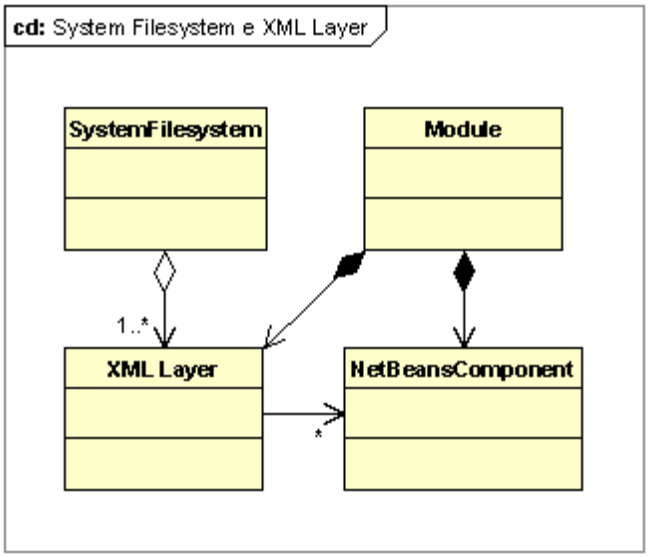


Figura 2.2: Strutturazione degli elementi fondamentali dell'ambiente NetBeans

2.2.1 Moduli NetBeans

In NetBeans il termine modulo (*module*) è comunemente utilizzato per indicare un plug-in per tale piattaforma; un modulo è sostanzialmente un archivio Java (*Java Archive*: file con estensione *.jar*, comunemente detto file JAR) che implementa una determinata caratteristica applicativa o un'estensione dell'ambiente, affiancato da alcuni file di configurazione. Un modulo si identifica come tale attraverso

un'apposita voce del file MANIFEST.MF.

Un modulo contiene un insieme di classi che utilizzano le *NetBeans API*, ovvero l'insieme di interfacce e classi pubbliche che permettono lo sviluppo di moduli per l'ambiente NetBeans o applicazioni basate su di esso [NbAPI]; tali API sono suddivise in gruppi, ciascuno dei quali fornisce il supporto a specifiche categorie di funzionalità.

Dato che un modulo estende l'ambiente NetBeans dotandolo di nuove caratteristiche, queste potrebbero essere necessarie ad altri moduli; conseguentemente, possono esistere relazioni di dipendenza tra moduli. In particolare possiamo avere un insieme di moduli che assieme arricchiscono di una nuova funzionalità l'ambiente; ognuno di questi moduli aggiunge ad esso un componente o un insieme di componenti necessari alla nuova caratteristica. La separazione di una funzionalità in più moduli permette di aggiornare i vari componenti che costituiscono la funzionalità in modo indipendente.

Un modulo può essere sviluppato per raggiungere uno dei seguenti obiettivi:

- *estendere l'ambiente*: il modulo fornisce nuove caratteristiche, tipicamente aggiungendo un adeguato supporto ad una determinata tecnologia;
- *contribuire alla costruzione di una rich-client application*: un modulo può implementare un'intera applicazione complessa oppure solo una parte, basandosi sui moduli contenuti nella NetBeans Platform e su altri eventuali moduli che estendono l'ambiente.

Il deployment di un modulo viene realizzato impacchettando il file JAR contenente il bytecode delle classi all'interno di un *NetBeans Module file* (file con estensione *.nbm*, comunemente detto *NBM file*, strutturalmente realizzato allo stesso modo di un file JAR), assieme ad altri file contenenti informazioni sul modulo, come la licenza d'uso, dipendenze da altri moduli, librerie utilizzate, contenuto dei manifest file, ecc.. In generale, ogni *NBM file* contiene un unico modulo[1].

Le informazioni relative al modulo contenuto in un *NBM file* vengono utilizzate dal *Module Manager* e dall'*Update Center*; il primo gestisce l'installazione/rimozione e l'attivazione/disattivazione dei moduli, mentre il

1 Nella documentazione di NetBeans spesso viene utilizzato ambiguamente il termine *module* al posto di *NBM file*, perché solitamente un *NBM file* contiene uno specifico modulo.

secondo gestisce l'individuazione e l'eventuale download di nuovi moduli, richiedendone l'installazione al *Module Manager*; prima di richiedere l'installazione di un modulo, *Update Center* verifica se sono già stati installati i moduli da cui dipende il modulo da installare. Nel caso di qualche mancanza l'installazione del nuovo modulo non procede. Infine, un *NBM file* può essere firmato digitalmente al fine di avere garanzie sull'origine del file, utilizzando gli stessi meccanismi per l'apposizione di firme digitali dei file JAR.

Affinché sia possibile la contemporanea installazione di più moduli tra loro interdipendenti o comunque strettamente legati, NetBeans prevede la realizzazione di suite di moduli (*module suite*).

Una suite di moduli è una collezione di *NBM file* affiancata da un apposito descrittore, che di fatto raccoglie un insieme di moduli tra loro dipendenti[2]; generalmente vengono create per raccogliere moduli che assieme realizzano una specifica funzionalità. Il descrittore associato alla suite di moduli è un file denominato *update.xml* e raccoglie le informazioni sui singoli moduli tra cui le loro dipendenze; le informazioni contenute in questo file vengono utilizzate per aggiornare i singoli moduli con versioni più recenti in modo indipendente.

Oltre alle suite di moduli esistono anche moduli libreria (*library wrapper module*), ovvero particolari *NBM file* il cui unico scopo è quello di aggiungere al classpath dell'ambiente un determinato archivio Java contenente funzionalità esterne a NetBeans, generalmente necessarie ad altri moduli. Ad esempio, se uno specifico interprete fosse necessario per l'implementazione di un plug-in, il relativo archivio Java dovrebbe essere incluso in un apposito *library wrapper module* e questo inserito assieme al modulo del plug-in in una *module suite*.

2 Esistono sostanzialmente due tipologie di suite di moduli: quelle che raccolgono solamente moduli che realizzano estensioni dell'ambiente NetBeans e quelle che raccolgono moduli che assieme realizzano un'applicazione stand-alone. Alle prime purtroppo non corrisponde una forma di deployment, mentre per le seconde è prevista la compressione in formato *.zip* dei moduli di cui sono composte assieme a quelli di base che realizzano la NetBeans Platform. Per distribuire una suite di moduli che estende l'ambiente NetBeans si deve ricorrere ad un *Update Center server*, ovvero un server che svolge il ruolo di repository di moduli e suite moduli al fine di permettere il download dei vari *NBM file* e dei file *update.xml*; ovviamente dal lato client sarà presente l'*Update Center* relativo all'ambiente NetBeans dell'utente che gestirà l'installazione dei moduli.

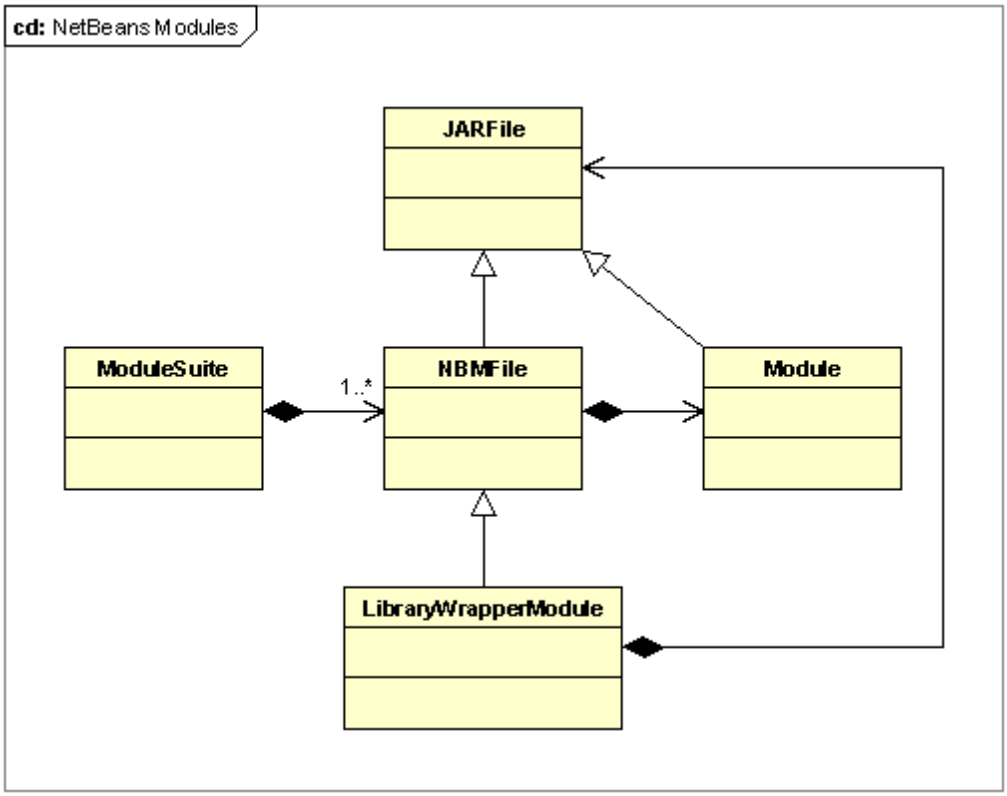


Figura 2.3: Strutturazione dei moduli e contenitori di moduli dell'ambiente NetBeans

2.2.2 System Filesystem e XML Layer

Il *System Filesystem* è sostanzialmente un filesystem virtuale contenente le informazioni relative ai componenti che fanno parte di NetBeans. E' possibile aggiungere al *System Filesystem* qualsiasi oggetto che deve essere reperibile da altri componenti inseriti nell'ambiente.

Ogni modulo è dotato di un documento XML detto *XML Layer* (file denominato *layer.xml* e inserito nel package principale del modulo), il cui scopo è quello di dichiarare i componenti contenuti nel modulo che devono essere registrati nel *System Filesystem* ed eventuali informazioni di configurazione. Al momento del caricamento di un modulo vengono estratte tutte le informazioni contenute nello *XML Layer* e inserite nel *System Filesystem*; quindi, a livello logico il *System Filesystem* è costituito da un insieme di *XML Layer*. In ogni modulo, la locazione dello *XML Layer* all'interno dell'archivio Java è specificata in una apposita voce del file MANIFEST.MF.

Il *System Filesystem* può essere modificato dinamicamente dal *Module Manager*, nel momento in cui viene installato/rimosso o attivato/disattivato un modulo a runtime; come conseguenza dell'installazione/rimozione di un modulo, viene

notificato un opportuno evento al fine di aggiornare quei componenti che devono cambiare il loro stato in base ai componenti installati nel *System Filesystem*. Ad esempio, se viene aggiunto un modulo al *System Filesystem* il cui *XML Layer* contiene solamente un riferimento ad una voce di menù, verrà lanciato un evento il cui effetto sarà quello di far apparire automaticamente nell'interfaccia grafica dell'ambiente la nuova voce di menù.

2.2.3 Recupero di un componente registrato nel System Filesystem

Il *System Filesystem* è di fatto un repository centrale da cui è possibile recuperare i vari componenti contenuti nei moduli. Tali componenti sono sostanzialmente degli oggetti ad unica istanza (*singleton*); possono essere presentati come servizi o risorse, ma si trattano comunque di specifici oggetti.

Al fine di ridurre le dipendenze tra classi, il meccanismo per l'accesso ai componenti dichiarati nel *System Filesystem* utilizza il design pattern *Service Locator* [Fow04], una particolare forma di *Inversion of Control* (vedi Appendice A); questo design pattern permette di accedere all'implementazione di un oggetto specificando l'interfaccia del servizio richiesto o una sua super-classe. La classe astratta che realizza tale design pattern è la `org.openide.util.Lookup`[3], ovviamente appartenente alle NetBeans API. Ad esempio, se fosse necessario il recupero della classe che implementa il servizio di gestione dei messaggi di log o di errore di NetBeans (la cui implementazione base è fornita dalla classe astratta `org.openide.ErrorManager`), si dovrebbe procedere nel seguente modo:

```
ErrorManager err = (ErrorManager)
    Lookup.getDefault().lookup(ErrorManager.class);

err.log("log message");
```

Il metodo di classe `Lookup getDefault()` della classe `Lookup` permette di recuperare l'unica istanza della classe che implementa la classe astratta `Lookup`; il metodo `Object lookup(Class)` permette di recuperare dal *System Filesystem* un'istanza dell'interfaccia o classe specificata dall'istanza della classe `Class` indicata come argomento.

Il comportamento del metodo `Object lookup(Class)` prevede tre possibili

3 Nel tutorial "Introducing to the NetBeans Idioms and Infrastructure" [NbTut1] viene erroneamente indicato che la classe `Lookup` realizza il design pattern *Dependency Injection* (vedi Appendice A).

casi:

- nel *System Filesystem* non sono specificati oggetti riferiti all'interfaccia/classe indicata; in tal caso il metodo restituisce `null`;
- nel *System Filesystem* è specificata un'unica istanza dell'interfaccia/classe indicata; in tal caso il metodo restituisce l'unica istanza individuata;
- nel *System Filesystem* sono specificate istanze multiple riferite all'interfaccia/classe indicata; in tal caso il metodo restituisce arbitrariamente una delle istanze individuate.

2.2.4 Altri meccanismi per la registrazione di componenti nel System Filesystem

Un modulo può dichiarare dei componenti da registrare nel *System Filesystem* non soltanto attraverso la loro dichiarazione nello *XML Layer* ad esso associato, ma anche attraverso il cosiddetto *Java provider extension mechanism*.

Tale meccanismo permette di specificare la classe che realizza una determinata interfaccia o che estende una determinata classe; permette il recupero di tale classe facendo riferimento solamente alla sua interfaccia o ad una sua super-classe, applicando di fatto il design pattern *Dependency Injection*. Esso è utilizzato esclusivamente all'interno di archivi Java.

Quindi, con tale meccanismo è possibile fornire una classe che implementa un servizio conoscendo solamente le specifiche comportamentali di tale servizio, ovvero l'interfaccia o una classe astratta. Conseguentemente dovrà essere presente un'entità che permetta il collegamento tra classe e interfaccia/super-classe; questa entità è costituita di fatto da un file denominato *service provider configuration file*. Il nome di tale file corrisponde al nome dell'interfaccia/super-classe implementata (comprensivo del nome del package) e il suo contenuto consiste in una sola riga di testo contenente il nome della classe implementante. La locazione di tale file corrisponde alla directory `META-INF/services`.

Ad esempio: se fosse necessario che la classe astratta `java.io.spi.CharCodec` sia implementata dalla classe `sun.io.StandardCodec`, allora all'interno del file `META-INF/services/java.io.spi.CharCodec` dovrebbe essere presente la seguente riga di testo:

```
sun.io.StandardCodec      #Eventuale commento.
```

Al momento del caricamento di un modulo, viene esplorato il corrispondente file JAR alla ricerca di eventuali *service provider configuration file*; individuato uno di questi, la classe indicata al suo interno viene automaticamente registrata nel *System Filesystem* riferendola al nome dell'interfaccia/super-classe da essa implementata.

2.3 Componenti principali che caratterizzano l'ambiente NetBeans

Dopo l'analisi delle caratteristiche fondamentali è bene esplorare i componenti che contraddistinguono principalmente l'ambiente NetBeans, limitatamente a quelli che verranno utilizzati per la realizzazione del plug-in. Questi sono incapsulati nei moduli di base che costituiscono la NetBeans Platform.

Possiamo suddividere i componenti principali in cinque gruppi:

- *componenti relativi alla persistenza delle risorse*: gestiscono la persistenza delle singole risorse (*file object*) e l'accesso ad esse (*filesystem*);
- *componenti relativi alla modellazione delle risorse*: definiscono le informazioni contenute nelle risorse e le loro funzionalità (*data object*), e ne gestiscono il caricamento (*data loader*);
- *componenti relativi alla rappresentazione delle risorse*: rappresentano graficamente le singole risorse (*node*) e le viste che raggruppano e gestiscono le rappresentazioni grafiche delle risorse (*explorer view*);
- *componenti relativi alle azioni*: implementano i meccanismi di base per l'esecuzione di azioni globali (*callable system action*) ed azioni relative alle singole risorse (*cookie action*);
- *componenti relativi all'interfaccia grafica dell'ambiente*: gestiscono l'intera interfaccia grafica di NetBeans (window manager), realizzano le finestre agganciate dentro di essa (*window component*) e gli editor delle risorse (*editor*).

Le risorse in NetBeans non sono rappresentate e gestite da un unico oggetto: la gestione della persistenza, la definizione del loro modello e la loro rappresentazione sono affidate ad oggetti distinti, organizzati su più livelli.

Al livello più basso si trova la rappresentazione "fisica" di una risorsa, tipicamente costituita da un file; al livello seguente la risorsa è rappresentata da un

file object, il quale ne gestisce la persistenza; al livello successivo viene definito il modello della risorsa attraverso un *data object*; al livello superiore è presente la rappresentazione grafica della risorsa, che è data da un *nodo*.

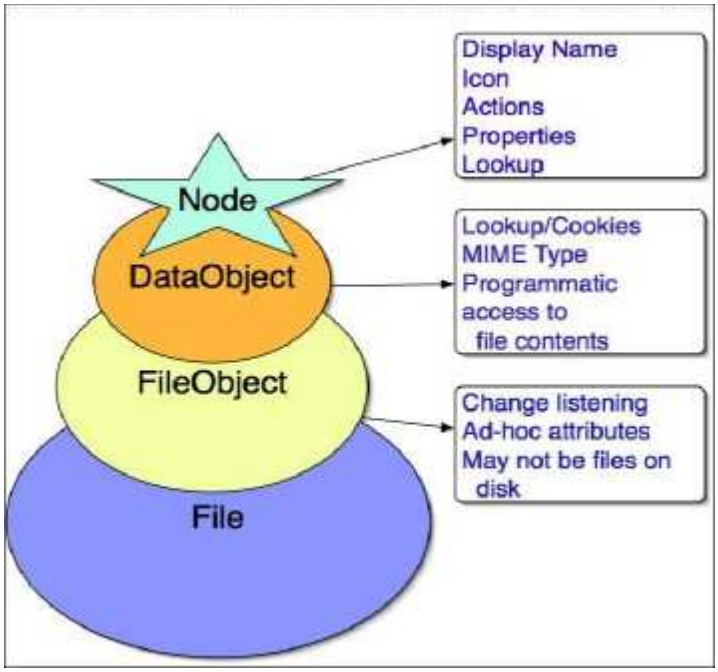


Figura 2.4: Immagine tratta dal tutorial "Recognizing a File Type Tutorial" [NbTut2] presente nel sito di NetBeans.org

Questa è la strutturazione tipica dei livelli di rappresentazione e gestione di una risorsa, ma sono ammesse anche delle varianti: infatti è possibile avere al livello base una entità diversa da un file, oppure avere solamente due livelli, con l'oggetto che implementa una risorsa al livello più basso e il corrispondente nodo al livello superiore.

2.3.1 File object e filesystem

La gestione della persistenza di una singola risorsa è affidata ad un *file object*, il quale è anche in grado di notificare i cambiamenti relativi al contenuto della risorsa. L'accesso ai vari *file object* è gestito da un *filesystem*[4], il quale costituisce uno spazio dei nomi per *file object*. Un *filesystem* dispone di una radice costituita da un *file object*; dato che ad ogni *file object* possono essere associati dei *file object* figli, un *filesystem* di fatto forma una struttura dati ad albero esplorabile costituita da *file object*.

4 Per rendere più semplice la comprensione del testo, i termini "filesystem" che si riferiscono al concetto tipico dell'ambiente NetBeans sono in corsivo.

L'implementazione di base di un *file object* è data dalla classe astratta `org.openide.filesystems.FileObject`. Nel caso in cui la risorsa incapsulata sia un file, l'oggetto incapsulato è un'istanza della classe `java.io.File`. A differenza della classe `File`, un oggetto che implementa la classe `FileObject` non si ottiene creandolo con un costruttore, ma recuperandolo tramite un *filesystem*.

I *filesystem* sono implementati dalla classe astratta `org.openide.filesystems.FileSystem`; un *filesystem* non rappresenta limitatamente un insieme di file presenti su disco, ma una generica forma di memorizzazione gerarchica strutturata ad albero, ovvero un qualunque filesystem, anche virtuale. Quindi, un *filesystem* può rappresentare la struttura del filesystem locale, o di un filesystem remoto, o di un file XML, o del contenuto di un file JAR, o di un insieme di tabelle relative ad un data base, ecc., ma non può rappresentare il contenuto di un file o di una qualunque risorsa. Infatti, per la prospettiva di un *filesystem*, il contenuto di un *file object* consiste in un flusso di byte non strutturato; per effettuare l'elaborazione del contenuto di un *file object* si deve ricorrere ad un *data object* (vedi sezione successiva).

Un particolare filesystem presente nell'ambiente NetBeans è il *System Filesystem*, il quale raccoglie tutte le informazioni di configurazione dei componenti caricati nell'ambiente, strutturandole ad albero. Per ottenere l'istanza del *System Filesystem* si deve utilizzare la classe `org.openide.filesystems.Repository`, richiamando il metodo `FileSystem getDefaultFileSystem()`:

```
FileSystem sfs =
   Repository.getDefault().getDefaultFileSystem();
```

Ottenuto il *System Filesystem* è possibile eseguire delle ricerche al suo interno utilizzando il metodo `FileObject findResource(String)`.

Per poter implementare un particolare *filesystem* le NetBeans API offrono la classe astratta `org.openide.filesystems.AbstractFileSystem`, la quale costituisce l'implementazione di base più conveniente per la realizzazione di nuove tipologie di *filesystem*.

2.3.2 Data object e data loader

I *data object* sono entità che rappresentano una risorsa o una macro-risorsa

incapsulando uno o più *file object*; a differenza dei *file object*, i *data object* non rappresentano la risorsa come un generico file, ma sono a conoscenza del tipo della risorsa e pertanto sono in grado di aggiungere responsabilità ad essa, rendendo possibile la definizione di operazioni che possono essere eseguite sulla specifica risorsa rappresentata. Un *data object* è in grado di interpretare il contenuto di una risorsa, quindi di fatto è un modello per l'accesso a tale risorsa.

La classe astratta `org.openide.loaders.DataObject` fornisce l'implementazione di base di tali entità. L'aggiunta di responsabilità non avviene definendo dei nuovi metodi in una sottoclasse della `DataObject`, ma attraverso l'uso del design pattern *Cookie*[5] (vedi Appendice A). Tale design pattern permette di aggiungere responsabilità ad una classe senza dover aggiungere nuovi specifici metodi. Ciò è reso possibile separando l'implementazione del modello dell'oggetto, ovvero il *data object*, dall'implementazione del suo comportamento, ovvero un insieme di *cookie*. Quindi, un cookie è di fatto l'implementazione di un certo comportamento, specifica per una determinata classe di *data object*. Per ottenere un'istanza di uno specifico cookie si ricorre al design pattern *Service Locator*, ovvero un cookie viene recuperato richiamando un apposito metodo della classe `DataObject`, specificando la sua interfaccia. Ad esempio, si ipotizzi di disporre di un oggetto che implementa la classe `DataObject` e che sia necessario eseguire l'operazione di apertura della risorsa da esso rappresentata; il relativo codice è:

```
OpenCookie open =
   someDataObject.getCookie(OpenCookie.class);

open.open();
```

Si noti che il cookie viene ottenuto richiamando un metodo relativo ad un'istanza della classe `DataObject`; quindi il cookie così ottenuto può agire esclusivamente sull'istanza utilizzata per recuperarlo.

Tipicamente i *data object* non sono creati richiamando un costruttore, ma utilizzando un *data loader*. Tale entità è in grado di riconoscere il tipo di un *file object*, quindi di creare un opportuno *data object*; infatti un *data loader* non è altro che una factory per un determinato insieme di tipi di file. Dato un file, un *data loader* ne riconosce il tipo tramite l'estensione oppure, se si tratta di un documento XML, analizzando l'elemento principale.

5 Non ha niente a che fare con i più celebri cookie utilizzati dai web browser.

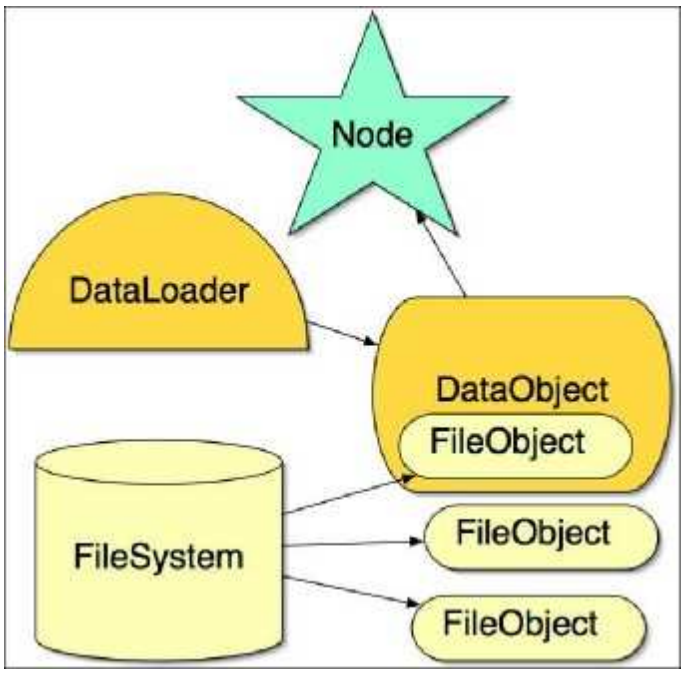


Figura 2.5: Immagine tratta dal tutorial "Recognizing a File Type Tutorial" [NbTut2] presente nel sito di NetBeans.org

La classe astratta `org.openide.loaders.DataLoader` fornisce l'implementazione di base di tali entità. Ogni *data loader* è specificato all'interno di un modulo e, al fine di poter essere utilizzato, registrato nel *System Filesystem*; i *data loader* sono automaticamente utilizzati dall'ambiente NetBeans. Infatti, dato un determinato *file object*, verranno interrogati tutti i *data loader* registrati, fino a che uno di questi non comunicherà che è in grado di trattare il *file object*. A questo punto, il *data loader* che si è fatto avanti verrà utilizzato per creare il *data object* adeguato al *file object* specificato. Quindi, un oggetto *data loader* non viene utilizzato esplicitamente, e nemmeno la classe astratta `DataLoader` viene utilizzata per ottenere un *data object*; infatti, partendo da un *file object* ottenuto tramite un *filesystem*, è possibile ottenere un *data object* utilizzando il metodo di classe `DataObject find(FileObject)` della classe `DataObject`. Vediamo un esempio:

```
FileObject file  =
  someFileSystem.findResource("someFolder/filename.ext");

DataObject dataObj = DataObject.find(file);
```

Infine, un *data object* è anche una factory per un *nodo*; quindi, come vedremo meglio nella sezione successiva, ogni *data object* è in grado di creare il suo corrispondente livello di presentazione.

2.3.3 Nodi

Un nodo (*node*) è il livello di presentazione di una qualunque risorsa, tipicamente implementata da un *data object* o da un bean; un nodo è utilizzato per tre scopi:

- rappresentare graficamente una risorsa, utilizzando un'icona e del testo;
- permettere di elencare insiemi di proprietà relativi alla risorsa rappresentata;
- permettere di eseguire delle azioni sulla risorsa elencandole opportunamente in un menù contestuale; tali azioni corrispondono ai cookie attivi sulla risorsa.

Un nodo può anche formare una gerarchia di nodi, quindi può rappresentare una risorsa costituita da un aggregato di risorse, anch'esse rappresentate graficamente ognuna da un nodo; i cookie di un nodo possono accedere ed apportare modifiche anche ad altri nodi presenti nella gerarchia, grazie al fatto che una gerarchia di nodi è esplorabile.

La classe astratta `org.openide.nodes.Node` fornisce l'implementazione di base di tali entità. Tipicamente un nodo incapsula un *data object*, ma si potrebbe anche realizzare un'estensione della classe `Node` che incapsula un differente modello di risorsa. Concettualmente, un nodo dovrebbe essere creato dal modello di risorsa che esso rappresenta, ma se tale risorsa non proviene dall'ambiente NetBeans, sicuramente non è in grado di creare un nodo; pertanto è anche possibile creare un nodo associando alla risorsa rappresentata. Avendo un *data object*, un nodo deve essere creato in tal modo:

```
someDataObject.getNodeDelegate();
```

Un nodo è in grado di elencare le azioni su di esso eseguibili, richiamando il metodo `Action[] getActions(boolean)`, distinguendo tra azioni relative al contesto del nodo (parametro d'ingresso con valore *true*) e azioni relative al nodo stesso (parametro d'ingresso con valore *false*).

Un nodo può anche incapsulare una serie di cookie che rappresentano il comportamento del nodo; per fare in modo che tali cookie siano accessibili conoscendone solamente l'interfaccia o la classe astratta, si utilizza un'istanza della classe `Lookup` associata al nodo considerato. Infatti, la classe astratta `Node`, realizza l'interfaccia `Lookup.Provider`, la quale definisce un unico metodo che

permette la restituzione dell'istanza della classe `Lookup` associata al nodo.

2.3.4 Explorer view e explorer manager

Gli insiemi di nodi sono tipicamente raccolti in interfacce grafiche, denominate *explorer view*; queste sono viste implementate utilizzando componenti grafici *Swing*. Permettono di visualizzare in modo strutturato le gerarchie di nodi; tipiche strutture di rappresentazione dei nodi sono alberi, liste, combo-box, insiemi di icone, ecc. Due esempi di *explorer view* ben note sono le schede denominate *Projects* e *Files* presenti nell'interfaccia grafica tipica dell'ambiente NetBeans.

Le *explorer view* sono in grado di gestire le operazioni di taglia/copia/incolla, drag & drop, visualizzazione di menù contestuali contenenti le voci relative ai cookie attivi sul nodo a cui si riferiscono. Esse però non hanno una particolare conoscenza di ciò che visualizzano, ed inoltre, la gerarchia di nodi visualizzata non viene direttamente indicata ad una *explorer view*. Infatti, per permettere a più *explorer view* di visualizzare la stessa gerarchia di nodi, si utilizza un *explorer manager*, ovvero un oggetto mediatore in grado di indicare la radice della gerarchia di nodi. In tal modo è possibile avere un componente grafico in grado di visualizzare più *explorer view* che rappresentano la stessa gerarchia di nodi, ma in modo diverso. Una eventuale modifica alla gerarchia di nodi, apportata tramite l'esecuzione di un'azione contestuale da parte dell'utente mentre sta operando in una *explorer view*, si propaga anche alle altre *explorer view* contenute nello stesso componente grafico.

Un *explorer manager* non viene indicato ad una *explorer view* attraverso uno specifico metodo, ma è compito della *explorer view* individuarne uno. Dato che più *explorer view* inserite nello stesso componente grafico tipicamente rappresentano la stessa gerarchia di nodi, i progettisti dell'ambiente NetBeans hanno pensato di rendere disponibile l'*explorer manager* da uno dei componenti grafici che racchiudono l'insieme di *explorer view*; per accedere all'*explorer manager*, le *explorer view* devono risalire la gerarchia di composizione dell'interfaccia grafica di cui fanno parte, alla ricerca di un componente che realizzi l'interfaccia `org.openide.explorer.ExplorerManager.Provider`. Tale interfaccia specifica il metodo `ExplorerManager getExplorerManager()`, il quale restituisce l'*explorer manager* che indica alle *explorer view* qual'è la radice della gerarchia di nodi.

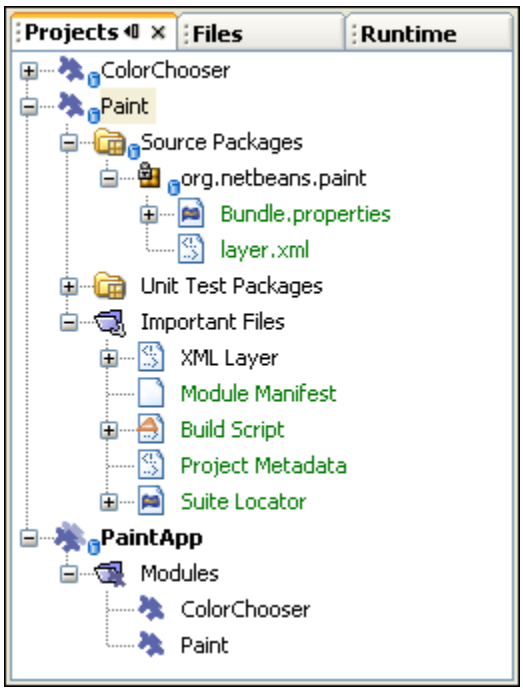


Figura 2.6: Esempio di explorer view; immagine tratta dal tutorial "Introduction to NetBeans Module Development" [NbTut3] presente nel sito di NetBeans.org

Purtroppo, non sempre la gerarchia relativa ad una composizione di interfacce grafiche *Swing* è esplorabile; infatti è possibile inserire dei componenti *Swing* all'interno di particolari *gruppi di layout*[6], e questi componenti non risulteranno essere associati ad un contenitore, ovvero il metodo `Container getParent()`, utilizzato per l'esplorazione della gerarchia, restituisce sempre `null`. In tali casi, la gerarchia di composizione risulta essere interrotta e ad una *explorer view* viene restituita una nuova istanza della classe `org.openide.explorer.ExplorerManager`, la quale non contiene alcuna gerarchia di nodi.

Affinché i cookie relativi ai vari nodi siano disponibili, ad ogni nodo deve essere associato un *lookup*[7]; i nodi correntemente selezionati dall'utente e i relativi *lookup* sono inseriti a loro volta all'interno di un *lookup* radice. Questo *lookup* radice è reperibile allo stesso modo dell'*explorer manager*, ovvero risalendo la gerarchia di composizione dei componenti grafici; quindi lo stesso componente grafico che restituisce l'*explorer manager* relativo alla gerarchia di nodi interessata, deve anche restituire un apposito *lookup* contenente soltanto i nodi selezionati e i *lookup* ad essi associati. Pertanto, oltre a realizzare l'interfaccia

6 Ad esempio, il layout implementato dalla classe `org.jdesktop.layout.GroupLayout` permette di raggruppare dei widget in gruppi di layout inseribili all'interno di altri widget.

7 Per semplificare, chiameremo semplicemente *lookup* una generica istanza della classe `Lookup`.

`ExplorerManager.Provider`, tale componente grafico deve anche realizzare l'interfaccia `org.openide.util.Lookup.Provider`. Questa interfaccia specifica il metodo `Lookup getLookup()`, il quale restituisce il *lookup* radice accennato prima.
Affinché il contenuto di questo *lookup* corrisponda ai nodi effettivamente selezionati dall'utente, l'*explorer manager* deve aggiornare il *lookup* radice relativamente ai cambiamenti dell'insieme dei nodi selezionati. Per evitare che l'implementazione dell'*explorer manager* sia dipendente da una particolare implementazione del *lookup* radice, quest'ultimo dovrebbe essere in grado di ricevere delle notifiche relative ai cambiamenti dell'insieme dei nodi selezionati dall'utente, registrandosi come listener presso l'*explorer manager*. Infatti, la classe `Explorer Manager` permette la registrazione di istanze che realizzano l'interfaccia `java.beans.PropertyChangeListener`, la quale dovrà essere realizzata dall'istanza della classe `Lookup` che implementa il *lookup* radice.
Al fine di creare e inizializzare opportunamente il *lookup* radice, in modo tale che sia registrato presso l'*explorer manager* che gestisce la gerarchia di nodi interessata, è possibile utilizzare il metodo di classe `Lookup createLookup(ExplorerManager,ActionMap)`, relativo alla classe `org.openide.explorer.ExplorerUtils`, utilizzando come parametri di ingresso un riferimento all'*explorer manager* interessato e una mappa contenente un insieme di azioni predefinite, le quali verranno aggiunte al *lookup*.
Se il *lookup* radice non fosse opportunamente inizializzato, esso non riceverebbe le notifiche relativamente all'insieme dei nodi selezionati dall'utente, e pertanto le azioni associate ai vari nodi selezionati non sarebbero reperibili; conseguentemente, richiamando un menù contestuale, le azioni dei nodi selezionati dall'utente risulterebbero sempre non attive.

2.3.5 Azioni

Le principali azioni richiamabili da parte dell'utente nell'ambito dell'ambiente NetBeans sono definite dalle *Actions API*. Esse sono richiamabili in più modi, ovvero possono essere invocate dall'utente tramite interfaccia grafica, combinazioni di tasti, ecc.

Le azioni in NetBeans sono tipicamente degli oggetti singleton. Esse devono essere registrate in un apposito repository; si hanno due casi:

- *repository globale* (il *System Filesystem*): le azioni sono globali e, conseguentemente, possono essere invocate in più modi e da più componenti;
- *repository locale* (*lookup* interno ad un determinato nodo): le azioni sono locali e invocabili se determinate condizioni dipendenti dal contesto sono verificate.

Le *Actions API* forniscono due tipi di azioni:

- *Callable system action*: azione globale, sempre attiva, integrata in NetBeans e potenzialmente richiamabile da qualsiasi componente presente nell'ambiente; deve essere registrata nel *System Filesystem*; implementata dalla classe astratta `org.openide.util.actions.CallableSystemAction`;
- *Cookie action*: azione locale, attiva condizionatamente e sensibile al contesto a cui è applicata; questo tipo di azione è riferita ad un determinato nodo, ed è richiamabile tramite menù contestuale; tipicamente eseguono un'operazione svolta da un determinato cookie. Tali azioni sono implementate dalla classe `org.openide.util.actions.CookieAction`. I meccanismi per il controllo dell'attivazione di un'azione implementata da tale classe, sono due:
  - ridefinizione del metodo `protected boolean enable(Node[])` da parte della sottoclasse che implementa l'azione; tale metodo realizza la logica di verifica dell'attivazione dell'azione, analizzando i nodi selezionati dall'utente forniti in ingresso;
  - utilizzo combinato dei metodi `protected Class[] cookieClasses()` e `protected int mode()`, i quali devono essere definiti da parte della sottoclasse che implementa l'azione. Il primo metodo restituisce l'elenco delle classi/interfacce di cui almeno una deve essere supportata da ciascun nodo selezionato, mentre il secondo metodo specifica la modalità di selezione dei nodi necessaria affinché l'azione sia attiva. Le modalità di selezione dei nodi sono:
    - `MODE_ONE`: uno o più nodi sono selezionati e soltanto uno supporta almeno uno dei cookie indicati;

- `MODE_SOME`: più nodi sono selezionati ed almeno uno supporta come minimo uno dei cookie indicati;
- `MODE_ALL`: uno o più nodi sono selezionati e tutti supportano almeno uno dei cookie indicati;
- `MODE_EXACTLY_ONE`: un solo nodo è selezionato e supporta almeno uno dei cookie indicati;
- `MODE_ANY`: uno o più nodi sono selezionati ed almeno uno supporta come minimo uno dei cookie indicati.

2.3.6 Window component e window manager

Con il termine *window component* si indicano genericamente delle viste perfettamente integrate nell'interfaccia grafica tipica dell'ambiente NetBeans. Tali viste possono essere finestre indipendenti o essere agganciate all'interno di altre finestre.

La classe che implementa tali viste è denominata `org.openide.windows.TopComponent`; nella documentazione delle NetBeans API, generalmente il termine *top component* è utilizzato al posto del termine *window component*. La caratteristica da evidenziare maggiormente consiste nel fatto che tale classe estende la `javax.swing.JComponent`; questo implica che questa classe implementa un componente grafico *Swing*, quindi è possibile inserire all'interno di un *top component* tutti i widget *Swing* e relative estensioni disponibili, tra cui le *explorer view*. Conseguentemente, eventuali interfacce grafiche *Swing* sviluppate per applicazioni stand-alone possono essere facilmente integrate nell'ambiente NetBeans.

Tipicamente viene creata un'unica istanza di una classe che estende la `TopComponent`, applicando il design pattern *Singleton* [GoF95]. Ognuna di queste classi, deve avere un campo di classe non modificabile denominato `PREFERRED_ID` di tipo `String`, il cui valore verrà utilizzato per collegare la vista alla sua classe. Si utilizza un identificatore di classe per referenziare un'istanza proprio perché si dà per scontato che esista un'unica istanza della classe che estende la `TopComponent`.

Inoltre, ogni classe che estende la `TopComponent` usualmente è corredata di un'apposita azione di attivazione e di due documenti XML, allo scopo di ottenere

l'integrazione della relativa istanza in NetBeans; sia l'azione di attivazione che i due documenti XML devono essere dichiarati all'interno del documento *XML Layer*, relativo al modulo che contiene la classe `TopComponent`; *XML Layer*, *top component*, i relativi documenti XML per l'integrazione e l'azione di attivazione devono essere contenuti nel package principale del modulo.

L'azione di attivazione deve estendere la classe astratta `CallableSystemAction`; se il nome della classe relativa alla vista è `xxxTopComponent`, allora il nome della corrispondente classe di attivazione deve essere `xxxAction`.

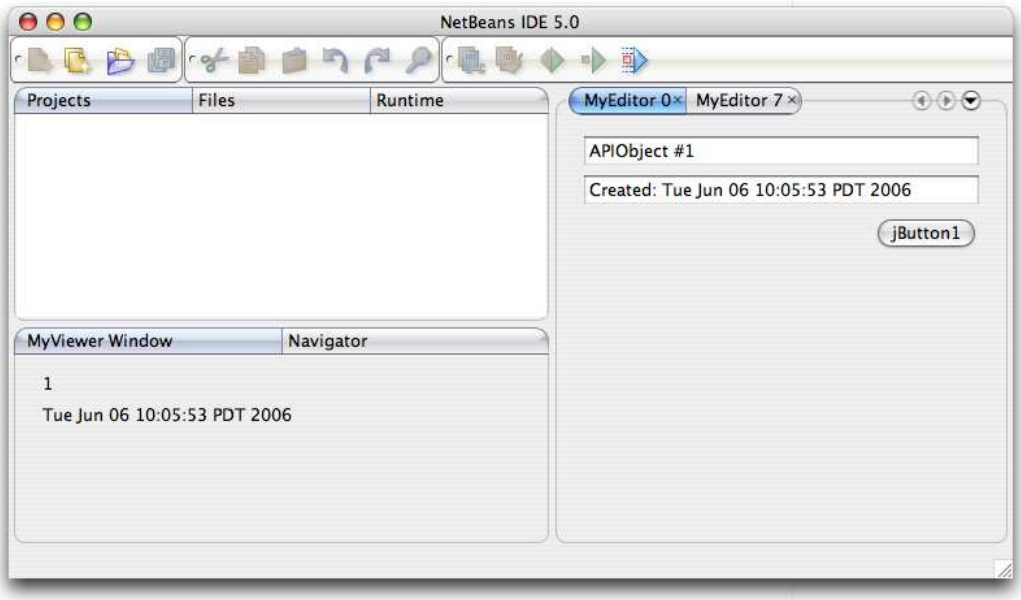


Figura 2.7: Esempio di TopComponent agganciato (MyEditor); immagine tratta dal tutorial "NetBeans Anagram Game Tutorial" [NbTut4] presente nel sito di NetBeans.org

I documenti XML per l'integrazione servono a specificare le caratteristiche della classe relativa alla vista e la configurazione della vista nell'ambiente; se la classe relativa alla vista è `xxxTopComponent`, allora questi due documenti sono:

- *xxxTopComponentSetting.xml*: specifica la configurazione della vista, elencando le classi che implementa e indicando il metodo di classe che permette di ottenere un'istanza della classe `xxxTopComponent`;
- *xxxTopComponentWstcref.xml*: referenzia la vista utilizzando il valore del campo di classe `PREFERRED_ID` della sua classe e ne specifica la configurazione all'interno dell'ambiente; tale tipo di documento indica soltanto se una vista deve essere aperta o chiusa all'avvio di NetBeans.

Una caratteristica importante della classe `TopComponent` è la possibilità di rendere persistenti le proprie istanze, utilizzando la tecnologia di serializzazione degli oggetti Java. Si hanno tre tipi di persistenza, relativamente ad un *top component*:

- `PERSISTENCE_ALWAYS`: il *top component* sarà sempre persistente, anche se è stato chiuso;
- `PERSISTENCE_ONLY_OPENED`: il *top component* non è persistente se è stato chiuso;
- `PERSISTENCE_NEVER`: il *top component* non è mai persistente.

Il tipo di persistenza utilizzata da un'istanza della classe `TopComponent` è indicata dal valore restituito dal metodo `int getPersistenceType()`.

I vari *top component* sono amministrati dal sistema di gestione delle finestre (*window manager*), il quale crea la finestra principale e permette l'apertura, la chiusura e la visualizzazione dei *top component*. L'implementazione di base relativa ad un sistema di gestione delle finestre è data dalla classe astratta `org.openide.windows.WindowManager`. Ovviamente ogni applicazione dispone di un unico *window manager*, opportunamente attivato al momento del suo lancio.

Inoltre il *window manager* gestisce, oltre ai *top component*, anche i così detti *mode*[8], ovvero generiche aree di una finestra, identificate da un nome unico, in cui è possibile agganciare dei *top component*; tipicamente sono visualizzati come schede (*tabbed pane*), ma anche come pannelli a scomparsa richiamabili tramite pulsanti (*sliding pane*). Purtroppo l'attribuzione del termine *mode* a queste aree di aggancio non è tra le più azzeccate; quindi per rendere più chiara l'esposizione dell'argomento, scegliamo di utilizzare il termine più significativo "area di aggancio di una finestra".

Notiamo subito che un'area di aggancio di una finestra non è affatto una interfaccia grafica, anche se si riferisce ad una specifica porzione di una finestra, ma è un contenitore di finestre amministrato dal *window manager*. Ogni finestra principale di un'applicazione basata sulla NetBeans Platform, (quindi anche il

8 Il termine completo sarebbe *docking mode*, ovvero "modalità di aggancio", ma nella documentazione di NetBeans è più utilizzato il termine *mode*, pertanto utilizzeremo quest'ultimo.

NetBeans IDE) è caratterizzata dalle aree di aggancio che dispone. Le tipiche aree di aggancio della finestra principale della NetBeans Platform sono:

- *output*: area in basso a sinistra della finestra che tende ad estendersi lungo tutta la sua larghezza; generalmente ospita il *top component* che visualizza l'output;
- *debugger*: area in basso a destra della finestra che tende ad estendersi lungo tutta la sua larghezza; generalmente ospita i *top component* per il debugging;
- *explorer*: area in alto a sinistra, al di sotto della barra dei menù; generalmente ospita i *top component* che effettuano l'esplorazione di gerarchie di nodi, come il *top component* relativo ai progetti aperti e quello relativo ai file dei progetti aperti;
- *navigator*: area in basso a sinistra, al di sopra dell'area di aggancio denominata *output*; generalmente ospita il *top component* che permette la navigazione del contenuto di un file;
- *commonpalette*: area in alto a destra, al di sotto della barra dei menù; generalmente ospita i *top component* che realizzano delle *palette*;
- *properties*: area in basso a destra, al di sopra dell'area di aggancio denominata *debugger*; generalmente ospita il *top component* che permette di visualizzare e modificare le proprietà del componente selezionato;
- *editor*: area centrale della finestra, al di sotto della barra dei menù, in cui vengono visualizzati i *top component* che permettono di modificare il contenuto di una risorsa;
- *leftSlidingSide*: area a sinistra della finestra che si estende lungo tutta la sua altezza; in questa area i *top component* sono visualizzati come *sliding pane*, pertanto non saranno visibili, ma potranno essere richiamati tramite un apposito pulsante;
- *rightSlidingSide*: area esattamente uguale alla *leftSlidingSide*, ma a destra della finestra;
- *bottomSlidingSide*: area in basso alla finestra, al di sotto dell'area di aggancio denominata *output*, che si estende lungo tutta la sua larghezza; in questa area i *top component* sono visualizzati come *sliding pane*.

2.3.7 Editor

L'ambiente NetBeans prevede un editor testuale di base, dotato di funzionalità fondamentali come undo/redo, ricerca di testo, evidenziazione del testo cercato, pulsanti per gestire l'indentazione, ecc.; tale editor è implementato dalla classe `org.openide.text.CloneableEditor`, la quale estende la classe astratta `org.openide.windows.CloneableTopComponent`, che a sua volta estende la classe `TopComponent`. Conseguentemente, l'editor di base non è altro che un *top component*, il quale verrà aperto proprio nell'area di aggancio denominata *editor*; l'editor di base non si distingue dagli altri *top component* solamente in base all'area in cui viene visualizzato, ma per il fatto che è clonabile, ovvero è possibile creare un altro editor in grado di modificare lo stesso oggetto, in modo tale da poter avere due punti di vista distinti relativamente ad esso.

Per poter creare un editor relativo ad un determinato oggetto si deve specificare un *editor support*, ovvero un'entità che permetta di accedere all'oggetto da editare e di apportare modifiche, oltre a fare in modo che esso riceva eventuali notifiche relative al cambiamento del contenuto dell'oggetto. Le funzionalità principali relative ad un *editor support* sono implementate dalla classe astratta `org.openide.text.CloneableEditorSupport`[9]. Un *editor support* non incapsula l'oggetto che deve essere editato, ma il suo ambiente, ovvero l'entità che permette la comunicazione tra supporto e l'oggetto da editare, in modo tale che questi non siano interdipendenti. Conseguentemente, per poter inizializzare un *editor support* occorre specificare l'ambiente dell'oggetto che verrà editato. Notiamo che il supporto agisce sull'ambiente, mentre l'oggetto notifica eventuali cambiamenti attraverso gli eventi implementati dalla classe `java.beans.PropertyChangeEvent`. Infine, la classe che implementa un *editor support* deve realizzare l'interfaccia `org.openide.cookies.EditorCookie`, al fine di permettere ad un editor di accedere e modificare il contenuto dell'oggetto associato all'*editor support*.

Il comportamento di un'ambiente relativo ad un oggetto editabile è specificato dall'interfaccia `org.openide.text.CloneableEditorSupport.Env`, la quale estende l'interfaccia `org.openide.windows.CloneableOpenSupport.Env`.

9 Notiamo che questa classe implementa i metodi relativi alle interfacce `OpenCookie`, `EditorCookie`, `EditCookie` e `ViewCookie` (tutte contenute nel package `org.openide.cookies`), ma non le realizza; pertanto saranno le sottoclassi a stabilire quali di queste devono essere effettivamente realizzate.

```
31      }
32
33      /** This method is called from within the constructor to
34       * initialize the form.
35       * WARNING: Do NOT modify this code. The content of this method is
36       * always regenerated by the Form Editor.
37       */
38      Generated Code
59
60
61      // Variables declaration - do not modify
62      private javax.swing.JTabbedPane jTabbedPanel;
63      // End of variables declaration
64
65      /**
66       * Gets default instance. Do not use directly: reserved for *.settings files only,
67       * i.e. deserialization routines; otherwise you could get a non-deserialized instance.
68       * To obtain the singleton instance, use {@link findInstance}.
69       */
70      public static synchronized PrologTopComponent getDefault() {
71          if (instance == null) {
72              instance = new PrologTopComponent();
73          }
74          return instance;
75      }
```

Figura 2.8: Esempio di editor testuale

Riepilogando, per editare un oggetto, si inizializza il corrispondente ambiente, dopodiché si crea il relativo supporto, infine si può creare e visualizzare un opportuno editor; un modo molto più pratico è sicuramente quello utilizzato dalla classe `org.openide.text.DataEditorSupport`, la quale implementa il metodo di classe denominato `CloneableEditorSupport create(DataObject, MultiDataObject.Entry, CookieSet)`; questo metodo permette la creazione di un *editor support* senza specificare l'ambiente dell'oggetto. Notiamo che in tal caso l'ambiente dell'oggetto da editare è prestabilito, visto che esso deve essere implementato da una specifica classe, la `DataObject`. Quindi, l'idea di questo esempio consiste nel prestabilire il tipo di oggetto da editare, in modo tale che il corrispondente ambiente sia ben noto e quindi incapsulabile; però si deve pagare un certo prezzo: in tal caso la classe che implementa l'*editor support* è dipendente dalla classe che implementa l'oggetto da editare.

Ovviamente, l'apertura di un editor è eseguita da un'azione lanciata dall'utente; l'azione di apertura di un oggetto in un editor è implementata dalla classe `org.openide.actions.EditAction`, ma può essere eseguita anche dall'azione `org.openide.actions.OpenAction`; la differenza tra queste due

azioni consiste nel fatto che un'istanza della classe `OpenAction` potrebbe aprire un oggetto in un qualunque *top component* o addirittura al di fuori dell'ambiente NetBeans[10], mentre un'istanza della classe `EditAction` dovrebbe aprire un oggetto solamente in un editor. Infine notiamo l'esistenza dell'azione `org.openide.actions.ViewAction`, la quale dovrebbe permettere esclusivamente la visualizzazione di un oggetto, aprendolo in un *top component* che non sia in grado di modificarlo; pertanto, per poter visualizzare un oggetto, potrebbe anche essere utilizzato uno speciale editor le cui funzionalità di modifica siano state preventivamente inibite. Osserviamo che abbiamo utilizzato dei termini condizionali relativamente alle azioni che potrebbero o dovrebbero svolgere le azioni di cui abbiamo appena discusso, questo perché il loro corretto comportamento dipende dalle implementazioni dei cookie che esse andranno ad utilizzare. Infatti, la classe `OpenAction` utilizza istanze della classe `org.openide.cookies.OpenCookie`, la classe `EditAction` utilizza istanze della classe `org.openide.cookies.EditCookie`, infine la classe `ViewAction` utilizza istanze della classe `org.openide.cookies.ViewCookie`.

2.4 Definizione di nuovi tipi di file

Il riconoscimento di un file nell'ambiente NetBeans è basato sull'estensione del file; ad ogni tipo di file sono associate, da parte dell'ambiente, specifiche funzionalità. E' anche possibile distinguere tipi di file distinti tra file XML (riconosciuti tramite l'estensione *.xml*) basandosi sul namespace; pertanto, i file XML possono presentare tra di loro funzionalità distinte, a differenza dei file dotati di altre estensioni per i quali le funzionalità sono le stesse se l'estensione è la stessa. Due o più file XML sono dotati delle stesse funzionalità se e solo se sono hanno elementi principali omonimi e associati allo stesso namespace.

Per default, NetBeans Platform non è predisposto al riconoscimento di alcun tipo di file, mentre NetBeans IDE è predisposto al riconoscimento di molti tipi di file; per riconoscere tipi di file non ancora supportati, è necessario estendere l'ambiente NetBeans, aggiungendo un modulo che contiene i meccanismi per il riconoscimento dei nuovi tipi di file.

10 Ad esempio può aprire un documento HTML nel web browser predefinito della macchina in cui è in esecuzione.

Per il riconoscimento di un nuovo tipo di file, all'interno di un modulo occorre inserire (ove xxx sta ad indicare un generico nome associato al nuovo tipo di file):

- un documento XML denominato *xxxResolver.xml*, contenente la dichiarazione dell'estensione e l'identificatore MIME[11] del tipo di contenuto; i documenti di questo tipo sono comunemente detti *MIMEResolver*;
- una classe `xxxDataObject`, in grado di rappresentare ed elaborare il contenuto del nuovo tipo di file;
- una classe `xxxDataLoader`, in grado di caricare i file relativi al nuovo tipo e di creare, per ognuno di essi, il relativo *data object*;
- una classe `xxxDataBeanInfo`, ovvero un descrittore del *data loader* relativo al nuovo tipo di file, il quale definisce una serie di informazioni utili all'introspezione del *data loader* e l'icona associata al nuovo tipo di file;
- una classe `xxxDataNode`, la quale specifica la presentazione dei file relativi al nuovo tipo.

Notiamo che non è necessario che tutte le classi e il *MIMEResolver* che definiscono il nuovo tipo di file siano contenute nello stesso package; in particolare, il *MIMEResolver* deve essere incluso nel package principale del modulo, dato che lo *XML Layer* deve includere un riferimento ad esso all'interno del direttorio `Services/MIMEResolver`.

Abbiamo precedentemente illustrato che per ottenere un *data object* si utilizza il metodo di classe `DataObject find(FileObject)` relativo alla classe `DataObject`; tale metodo ricerca nel *System Filesystem* un *data loader* in grado di creare un *data object* partendo dal *file object* dato in ingresso. La selezione del *data loader* da utilizzare si basa sull'identificatore MIME relativo al *file object* dato in ingresso; l'identificatore MIME di un *file object* si ottiene richiamando il metodo `String getMIMEType()`.

11 Le specifiche MIME (Multi-Purpose Internet Mail Extensions - [RFC 2045] [RFC 2046] [RFC 2047]) sono un'estensione di RFC 822 e si prefiggono di risolvere alcune problematiche e limitazioni dei protocolli di posta elettronica. In tali specifiche viene definito un formato relativo ad identificatori di contenuto; gli identificatori di contenuto conformi a tale formato sono comunemente detti identificatori MIME.

Capitolo 3

Analisi del problema

In questo capitolo verrà effettuata l'analisi del problema relativamente allo sviluppo del plug-in di tuProlog per l'ambiente NetBeans, il quale sarà caratterizzato principalmente dalla gestione delle teorie multiple e dalla gestione della persistenza delle teorie, ma anche dalla modalità di adattamento del motore tuProlog all'interno del plug-in, dalla gestione della risoluzione di query e, infine, dalla gestione delle librerie tuProlog. Inoltre, saranno molto importanti anche le scelte relative all'integrazione del plug-in con l'ambiente NetBeans, ovvero quali componenti e funzionalità saranno utilizzati per la sua realizzazione.

3.1 Gestione della risoluzione di query

La risoluzione di una query tuProlog non consiste in una semplice elaborazione come quella di una funzione, in cui dati uno o più parametri si ottiene un unico risultato; infatti, può presentare risultati multipli (finiti o infiniti), ma anche alcun risultato, poiché il motore tuProlog può entrare in un ciclo risolutivo senza fine. Pertanto, all'utente non deve essere consentito soltanto di specificare la query che vuole risolvere, ma deve poter interrompere una risoluzione e, nel caso di possibili soluzioni multiple, di accettare una soluzione oppure di proseguire nella ricerca ed ottenere un ulteriore risultato.

Conseguentemente, affinché l'utente possa gestire la risoluzione di una query, il plug-in deve essere dotato di un apposito componente grafico che permetta di specificare la query da risolvere e di visualizzarne i corrispondenti risultati.

Tale componente grafico dovrà essere dotato di:

- un campo di testo per l'inserimento della query;
- un pulsante per iniziare la risoluzione di una query;
- un pulsante per terminare la risoluzione di una query;
- un pulsante per proseguire la risoluzione di una query nel caso di risultati multipli;
- un pulsante per l'accettazione dell'ultima soluzione ottenuta nel caso di risultati multipli;
- un'area di testo per la visualizzazione dei risultati della query.

Durante la risoluzione di una query, il motore tuProlog potrebbe eseguire dei predicati per la visualizzazione di messaggi di output, pertanto il plug-in deve essere dotato anche di un componente grafico in grado di visualizzare l'output del motore tuProlog. Infine, al motore tuProlog potrebbe anche essere richiesto di visualizzare le informazioni interne di elaborazione attraverso il predicato *spy*, pertanto il plug-in deve essere dotato anche di un componente grafico in grado di visualizzare le informazioni interne del motore tuProlog.

Quando l'utente specifica una query, questa può essere inconsistente, pertanto se ne deve verificare la correttezza; si potrebbe procedere in due modi: verificare la query solamente dopo che l'utente ha richiesto la sua risoluzione oppure durante la digitazione; nel primo caso dovrà essere notificata l'inconsistenza della query con un messaggio di errore, mentre nel secondo caso si dovrà impedire all'utente di richiedere la valutazione di una query inconsistente, non attivando l'apposito pulsante per avviare la risoluzione.

3.2 Gestione delle teorie multiple

La versione del motore tuProlog con il supporto per l'utilizzo di più teorie logiche permette di effettuare risoluzioni di query utilizzando una *teoria dinamica*, ovvero una teoria composta dall'insieme delle clausole appartenenti alle teorie caricate nel motore tuProlog. Le clausole che compongono la teoria dinamica saranno ordinate sia in base all'ordine di caricamento delle teorie nel motore tuProlog, sia in base alla loro posizione all'interno di una teoria; quindi le clausole che appartengono ad una teoria caricata precedentemente ad un'altra teoria saranno precedenti alle

clausole di quest'altra teoria. Pertanto l'ordine di caricamento delle teorie è molto importante, perché può influenzare il risultato della risoluzione di una query. Ne consegue che il plug-in deve permettere all'utente di gestire le teorie caricate nel motore tuProlog in modo chiaro e flessibile, al fine di rendere più versatile la risoluzione di query; la chiarezza relativamente alle teorie caricate e la flessibilità di gestione di queste sono gli obiettivi da raggiungere per poter realizzare un valido supporto per la gestione delle teorie multiple nel plug-in.

Iniziamo l'analisi della gestione delle teorie multiple definendo il concetto di *teoria attiva*: una teoria che l'utente considera utilizzabile per la risoluzione di una query. Quindi una teoria attiva non coincide con una teoria caricata nel motore tuProlog: infatti può essere immediatamente caricata all'interno del motore tuProlog oppure in un secondo momento. Inoltre, l'utente dovrebbe avere la possibilità di escludere o includere una teoria attiva nella valutazione di una query; ovviamente, nel momento in cui sarà avviata la risoluzione di una query, tutte le teorie attive che l'utente ha scelto di includere nella valutazione dovranno essere caricate nel motore tuProlog. Dato che l'ordine di caricamento delle teorie nel motore tuProlog può essere fondamentale per la correttezza dei risultati di una query, le teorie attive devono essere elencate nell'ordine corrispondente a quello dell'eventuale caricamento, indipendentemente dal fatto che siano state incluse o escluse dalla valutazione delle query. Pertanto, ciò che l'utente deve gestire prima di eseguire la risoluzione di una query, è un *elenco di teorie attive*; per questo motivo, spesso verrà utilizzata l'espressione *gestione delle teorie attive* al posto di gestione delle teorie multiple.

Riepilogando, l'utente dovrebbe percepire una teoria attiva come se fosse caricata nel motore tuProlog e con la possibilità di essere utilizzata per la valutazione di query o meno; inoltre l'utente dovrebbe avere la possibilità di gestire un elenco delle teorie attive liberamente, senza limitazioni simili a quelle imposte per la gestione delle teorie nel motore tuProlog. Tale elenco deve essere unico, dato che l'utente lo percepisce come elenco delle teorie inserite nel motore tuProlog.

Le teorie attive dovrebbero essere modificabili sia per quanto riguarda il contenuto, sia per quanto riguarda il loro nome; ricordando che si possono avere teorie etichettate con lo stesso nome caricate nel motore tuProlog, allora si dovrebbe dare la possibilità all'utente di rinominare anche tutte le teorie attive etichettate con lo stesso nome contemporaneamente.

L'elenco delle teorie attive potrebbe essere modificato dall'utente attraverso l'aggiunta di nuove teorie etichettate in fondo all'elenco (anche con etichette uguali a quelle di teorie già presenti nell'elenco), rimozione di teorie, rimozione di tutte le teorie attive e impostazione di un'unica teoria; queste funzionalità di modifica dell'elenco delle teorie attive sono supportate anche dal gestore delle teorie del motore tuProlog[1], pertanto verranno chiamate *funzionalità supportate*.

Altre funzionalità di modifica dell'elenco delle teorie attive potrebbero essere il cambiamento di posizione di una teoria, la fusione tra due o più teorie[2], la separazione di una teoria in due o più teorie; queste, però, non sono supportate dal gestore delle teorie del motore tuProlog, pertanto saranno denominate *funzionalità avanzate*[3]. Per realizzarle, occorrerà un uso più articolato delle funzionalità di gestione delle teorie offerte dal motore tuProlog.

Il seguente elenco evidenzia le funzionalità avanzate, specificando le operazioni che dovrebbero essere eseguite dal motore tuProlog al fine di realizzarle:

- *rimozione di una teoria che non sia l'ultima ad essere etichettata con un nome specifico*; i passi da eseguire sono:
    1. estrazione e rimozione dal motore tuProlog di tutte le teorie successive alla teoria da rimuovere;
    2. rimozione della teoria;
    3. reinserimento nell'ordine precedente delle teorie estratte, in ordine inverso rispetto a quello di estrazione e rimozione;
- *fusione di teorie*; i passi da eseguire sono:
    1. estrazione dal motore tuProlog delle teorie da fondere assieme;
    2. esecuzione della fusione, attribuendo alla nuova teoria un nome indicato da una specifica strategia;
    3. rimozione delle teorie tra loro fuse dall'elenco delle teorie attive;

1 Si ricordi che il motore tuProlog permette la rimozione di teorie limitatamente a quelle che, nell'ambito dell'insieme di tutte le teorie dotate di una determinata etichetta, sono state caricate per ultime.

2 Si potrebbe realizzare la fusione tra un insieme qualunque di teorie, anche tra tutte le teorie attive e tra tutte le teorie dotate di una determinata etichetta.

3 Tra le funzionalità avanzate è compresa la possibilità di rimuovere una qualunque teoria dal motore tuProlog, senza la limitazione descritta in una precedente nota.

4. inserimento della nuova teoria ottenuta dalla fusione in fondo all'elenco delle teorie attive.

- *separazione di una teoria*; i passi da seguire sono:
  1. generazione delle nuove teorie separate;
  2. aggiunta delle nuove teorie all'elenco delle teorie attive;
  3. rimozione della teoria generatrice delle teorie separate dall'elenco delle teorie attive.
- *cambiamento di posizione di una teoria*; implica un cambiamento nell'organizzazione dell'elenco delle teorie attive all'interno del motore tuProlog: infatti il motore tuProlog è pensato per poter aggiungere le nuove teorie solamente in fondo all'elenco delle teorie attive; i passi da eseguire sono:
  1. estrazione dal motore tuProlog della teoria da spostare;
  2. rimozione della teoria da spostare;
  3. estrazione dal motore tuProlog di tutte le teorie dalla posizione di inserimento in poi;
  4. rimozione di tutte le teorie dalla posizione di inserimento in poi;
  5. inserimento in fondo alla lista delle teorie attive della teoria da spostare;
  6. reinserimento nell'ordine precedente delle teorie che si trovavano dalla posizione di inserimento in poi, eventualmente escludendo la teoria da spostare;
- *cambiamento del nome di una teoria attiva*; la realizzazione di questa funzionalità presenta più difficoltà rispetto a quello che sembra: infatti non sono presenti funzionalità che permettono il cambiamento del nome ad una teoria, quindi i passi da eseguire sono:
  1. creazione di una nuova teoria con contenuto identico alla teoria da rinominare, ma dotata del nome nuovo;
  2. estrazione di tutte le teorie dalla posizione di inserimento della nuova teoria in poi;

3. rimozione di tutte le teorie dalla posizione di inserimento in poi;
4. inserimento della nuova teoria;
5. reinserimento nell'ordine precedente delle teorie che si trovavano dalla posizione di inserimento in poi;

- *cambiamento del nome di un insieme di teorie omonime*; consisterà nel cambiamento del nome di ogni teoria dotata del nome indicato con il nuovo nome indicato.

Si rammenti che il motore tuProlog può modificare la composizione delle teorie in esso caricate durante la risoluzione di una query, come conseguenza della valutazione dei predicati *set_theory*, *add_theory* e *remove_theory*; pertanto l'elenco delle teorie attive deve essere opportunamente aggiornato dopo l'esecuzione di uno di questi predicati appena elencati, in modo tale che esso permetta all'utente di capire lo stato delle teorie caricate all'interno del motore tuProlog. Pertanto è lecito dire che il motore tuProlog può apportare modifiche all'elenco delle teorie attive, ma con alcuni limiti rispetto all'utente; infatti l'utente può scegliere di includere o escludere nella valutazione una teoria attiva senza rimuoverla dall'elenco, mentre il motore tuProlog non può farlo; inoltre, il motore tuProlog non può rimuovere una teoria attiva esclusa dalla valutazione di una query, dato che tale teoria non è di fatto caricata nel motore tuProlog[4].

Riepilogando, gli attori che possono modificare l'elenco delle teorie sono due: l'utente e il motore tuProlog; entrambe devono essere avvisati delle modifiche che l'altro apporta. E' ovvio che utente e motore tuProlog non devono interferire l'uno con l'altro nel modificare l'elenco delle teorie attive; perciò all'utente dovrà essere impedita la possibilità di apportare modifiche all'elenco delle teorie attive durante la risoluzione di una query.

Attualmente, il motore tuProlog non notifica a runtime le modifiche apportate ad una teoria, ovvero, se durante la risoluzione di una query venisse eseguito un predicato *assert* o *retract*, il conseguente cambiamento della teoria non verrebbe

4 Nel caso in cui si utilizzi un gruppo ristretto di teorie attive per la risoluzione di una query, il motore tuProlog potrebbe eseguire un predicato che rimuova una specifica teoria dall'elenco delle teorie attive. Nel caso in cui la teoria da rimuovere sia presente nell'elenco delle teorie attive, ma sia esclusa dalla valutazione della query, il predicato di rimozione fallirà. Quindi, in una logica che considera l'elenco delle teorie attive come una struttura dati manipolabile dal motore tuProlog, quello appena descritto è un comportamento errato.

notificato attraverso un evento. Pertanto, al termine dell'esecuzione di una query, non è possibile sapere se sono state modificate delle teorie o meno. Bisognerà porre rimedio a tale problema modificando opportunamente il motore tuProlog.

La modifica dell'elenco delle teorie attive o del contenuto di una teoria appartenente ad esso, può non essere desiderata dall'utente, per esempio quando sta sperimentando il comportamento di un insieme di teorie; per questo motivo, l'utente deve poter scegliere di rendere definitivi o meno i cambiamenti che può apportare la valutazione di una query all'elenco delle teorie attive o del contenuto delle teorie. Quindi, deve avere la possibilità di eseguire *query temporanee*, ovvero query in cui l'eventuale esecuzione di predicati che modificano l'elenco delle teorie attive o il contenuto di teorie non abbiano effetti permanenti, ma solo durante la risoluzione della query stessa.

Dato che il motore tuProlog può modificare l'elenco delle teorie attive, si deve dare la possibilità all'utente di esportare le teorie per permetterne la memorizzazione, affidandole ad un apposito gestore della persistenza, denominato *elenco delle teorie disponibili*, che verrà discusso più avanti. Quindi l'utente deve avere la possibilità di estrarre le teorie dall'elenco delle teorie attive al fine di esportarle; per offrire una maggiore versatilità si possono avere cinque modalità differenti di estrazione di teorie:

- estrazione di una specifica teoria attiva;
- estrazione della teoria derivata dalla fusione di tutte le teorie etichettate con uno specifico nome;
- estrazione della teoria derivata dalla fusione di un qualunque insieme di teorie attive;
- estrazione della teoria derivata dalla fusione di tutte le teorie attive, ottenendo un'unica teoria attiva globale;
- estrazione di tutte le teorie attive, ottenendo un insieme di teorie distinte.

L'estrazione di una teoria o di un insieme di teorie dall'elenco delle teorie attive deve essere sempre seguita da una operazione di esportazione verso l'elenco delle teorie disponibili; al suo interno la teoria esportata sarà etichettata allo stesso modo rispetto all'elenco delle teorie attive.

L'importazione di una teoria consiste nell'aggiungere in fondo all'elenco delle

teorie attive una teoria proveniente dall'elenco delle teorie disponibili; la teoria aggiunta sarà etichettata con lo stesso nome che aveva nell'elenco delle teorie disponibili.

Una funzionalità molto utile per aumentare la versatilità di esecuzione delle risoluzioni di query potrebbe essere sicuramente la possibilità di specificare operatori logici per la selezione delle teorie attive da utilizzare per la valutazione di query; in tal caso l'utente percepirebbe che alle teorie attive incluse nella valutazione verrebbe applicato l'operatore che egli ha scelto.

Nel caso predefinito, il motore tuProlog utilizzerà tutte le teorie attive, applicando di fatto l'operatore logico *and*; se l'utente vorrà eseguire una query diversa da quella predefinita, potrà selezionare un operatore logico diverso oppure escludere delle teorie attive.

Per ogni operatore logico si definisce una tipologia di query:

- *and query*: si tratta della tipologia predefinita; tutte le teorie selezionate sono utilizzate per la risoluzione della query;
- *or query*: la risoluzione della query viene tentata per ogni teoria selezionata; quindi non si tratta dell'esecuzione di una sola risoluzione di query, ma di una sequenza di risoluzioni di query, ognuna eseguita sulla query specificata dall'utente;
- *not query*: tutte le teorie selezionate non sono utilizzate per la risoluzione della query; le teorie non selezionate, ma contenute nell'elenco delle teorie attive, saranno quelle che verranno utilizzate.

Infine, si definisce una tipologia di query a parte:

- *custom query*: un'espressione logica complessa permette di stabilire le teorie da utilizzare nella risoluzione della query; si potrebbe fare uso anche di tecniche di pattern matching per specificare le teorie da utilizzare per la risoluzione della query; a tal fine possono essere utilizzare le espressioni regolari.

In base a tutte queste considerazioni, è possibile stabilire che occorre un componente grafico in grado di rappresentare l'elenco delle teorie attive e tale da offrire le funzionalità di gestione delle teorie attive; tale componente sarà denominato *vista delle teorie attive*. Le funzionalità di gestione saranno offerte da

appositi menù contestuali; inoltre, la vista delle teorie attive deve permettere all'utente di indicare se una teoria deve essere inclusa o esclusa nella valutazione di una query.

Ogni volta che l'elenco delle teorie attive viene modificato, da parte dell'utente o del motore tuProlog, la vista delle teorie attive deve essere opportunamente aggiornata.

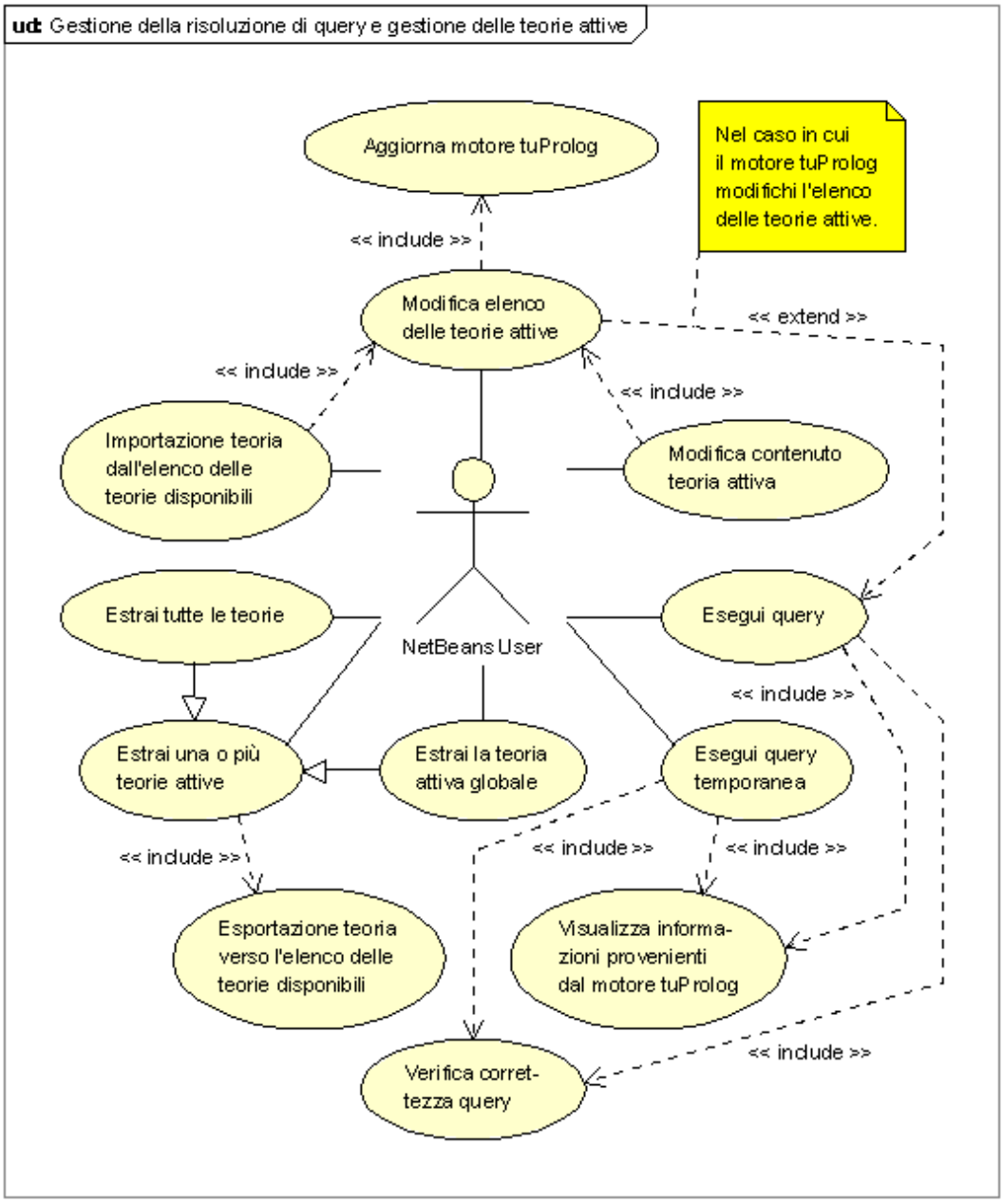


Figura 3.1: Diagramma dei casi d'uso relativo alla gestione della risoluzione di query e alla gestione delle teorie attive

Inoltre, occorrerà anche un pannello che consentirà all'utente di specificare l'operatore logico da utilizzare per la selezione delle teorie attive, e che, in caso di custom query, visualizzi anche un campo di testo che permetta di specificare l'espressione logica per la selezione delle teorie. Infine, occorrerà una casella di selezione per permettere all'utente di specificare se vuole eseguire una query temporanea o meno.

Per permettere all'utente di effettuare la modifica delle teorie logiche, occorre un apposito editor testuale, possibilmente in grado di evidenziare la sintassi attraverso la colorazione del testo.

Sia i componenti grafici relativi alla gestione della risoluzione di query, sia i componenti grafici, appena descritti, per la gestione dell'elenco delle teorie attive, devono essere integrati in un unico pannello che permetta l'utente il controllo completo delle risoluzioni di query.

3.3 Gestione della persistenza delle teorie

Per quanto riguarda la persistenza di teorie, è necessario che il plug-in sia indipendente dai meccanismi di persistenza utilizzati per la memorizzazione delle teorie. Quindi il plug-in potrebbe essere realizzato in modo tale da utilizzare contemporaneamente diversi supporti per la persistenza.

Si parlerà di *teoria persistente* nel caso in cui una teoria sia assegnata ad una precisa locazione relativamente ad un supporto per la persistenza; si parlerà invece di *teoria non persistente* nel caso in cui una teoria non sia assegnata a nessuna locazione di nessun supporto per la persistenza. Per poter specificare una teoria persistente non basta un *nome logico* (quello che costituisce l'etichettatura di una teoria), ma occorre anche un *nome fisico*, il quale permetta di specificare una determinata locazione su di un supporto per la persistenza. Le teorie non persistenti possono essere state generate durante la risoluzione di una query dal motore tuProlog, oppure ottenute eseguendo un'operazione di esportazione di teorie dall'elenco delle teorie attive.

Al fine di poter gestire la persistenza delle varie teorie, dovrà essere presente un unico elenco delle teorie rese disponibili dai vari supporti per la persistenza e delle teorie non persistenti; esso sarà denominato *elenco delle teorie disponibili*. Pertanto, spesso verrà utilizzata l'espressione *gestione delle teorie disponibili* al

posto di "gestione della persistenza di teorie". Come l'elenco delle teorie attive, anche questo è un elenco di teorie, conseguentemente presenta caratteristiche e funzionalità simili all'elenco delle teorie attive che verranno analizzate a fondo più avanti.

L'importazione di una teoria nell'elenco delle teorie disponibili comporta il caricamento di una teoria persistente da una specifica locazione relativa ad un supporto per la persistenza; l'esportazione di una teoria dall'elenco delle teorie disponibili comporta la memorizzazione della teoria in una specifica locazione su di un supporto per la persistenza; per effettuare tali operazioni l'utente deve specificare il nome fisico della teoria da importare/esportare.

Nel caso in cui si effettui l'esportazione di una teoria dall'elenco delle teorie attive verso l'elenco delle teorie disponibili, in questo verrebbe aggiunta una teoria non persistente dotata della stessa etichettatura della teoria esportata. Nel caso in cui si effettui l'importazione di una teoria dall'elenco delle teorie disponibili verso l'elenco delle teorie attive, in fondo a questo verrebbe aggiunta una teoria con la stessa etichettatura della teoria importata. In entrambe le operazioni, il nome della teoria importata/esportata rimane invariato nell'elenco di destinazione; ovviamente questo vale anche per il contenuto della teoria importata/esportata.

Nell'elenco delle teorie disponibili, le operazioni possibili relativamente alla modifica di un elenco di teorie sono le stesse di quelle dell'elenco delle teorie attive, ma le operazioni di importazione e esportazione si differenziano. Infatti, le operazioni di importazione/esportazione dell'elenco delle teorie attive hanno come mittente/destinatario l'elenco delle teorie disponibili; nel caso dell'elenco delle teorie disponibili, ad ogni operazione di importazione/esportazione si deve specificare il supporto per la persistenza da utilizzare e la locazione della teoria su cui si esegue l'operazione. Ovviamente, in questo ambito dovrà essere sfruttato anche il filesystem locale come supporto per la persistenza. Se durante una operazione di esportazione di una teoria disponibile l'utente specifica un nome fisico di un'altra teoria disponibile, allora quest'ultima sarà rimossa dall'elenco delle teorie disponibili perché sostituita dalla teoria esportata.

Una teoria disponibile può essere modificata tramite l'apposito editor di teorie accennato precedentemente; pertanto, il contenuto della teoria modificata deve essere reso persistente eseguendo un'operazione di aggiornamento della teoria. L'utente potrebbe anche eseguire la modifica di diverse teorie disponibili

contemporaneamente, pertanto deve essere possibile anche l'esecuzione dell'aggiornamento di tutte le teorie modificate, rendendo il loro nuovo contenuto persistente.

Per l'utente dovrebbe anche essere possibile creare una nuova teoria non persistente specificando solamente il nome logico; se l'utente vuole rendere subito persistente la nuova teoria, egli dovrà eseguire un'operazione di esportazione. Infine, per l'utente deve essere possibile anche la cancellazione di una teoria disponibile.

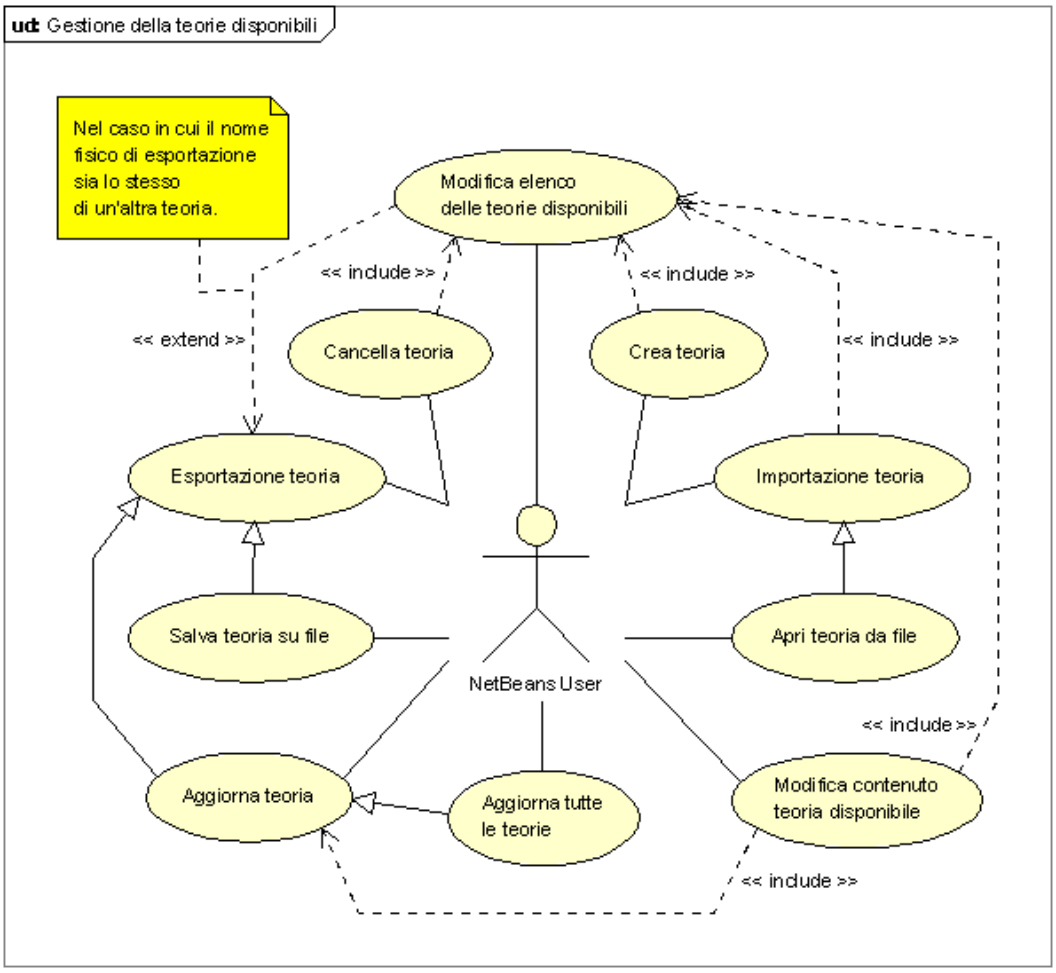


Figura 3.2: Diagramma dei casi d'uso relativo alla gestione delle teorie disponibili

In base a tutte queste considerazioni, è possibile stabilire che occorre un componente grafico in grado di rappresentare l'elenco delle teorie disponibili e tale da offrire le funzionalità di gestione delle teorie disponibili; tale componente sarà denominato *vista delle teorie disponibili*. Le funzionalità di gestione saranno offerte da appositi menù contestuali; inoltre, la vista delle teorie disponibili deve

permettere all'utente di capire se una teoria è aggiornata o non aggiornata, persistente o non persistente, oltre ad indicare sia il nome logico che il nome fisico di una teoria. Perciò, accanto al nome logico di teorie non aggiornate, dovrebbe essere presente un asterisco, mentre per quanto riguarda le teorie non persistenti, oltre a mancare il nome fisico della teoria, il solo nome logico dovrebbe essere scritto in corsivo, in modo da mettere la teoria maggiormente in risalto.

3.4 Gestione di un generico elenco di teorie

Precedentemente è stato osservato che le operazioni di modifica dell'elenco delle teorie disponibili sono le stesse dell'elenco delle teorie attive, ovvero sono:

- aggiunta di una teoria;
- rimozione di una teoria;
- impostazione di un'unica teoria;
- cambiamento del nome di una teoria;
- cambiamento del nome di tutte le teorie dotate di uno specifico nome;
- modifica del contenuto di una teoria;
- cambiamento di posizione di una teoria;
- fusione tra due teorie qualunque;
- fusione tra tutte le teorie dello stesso nome;
- fusione di tutte le teorie in un'unica teoria;
- separazione di una teoria in due o più teorie.

Inoltre, ogni volta che uno di questi elenchi di teorie viene modificato, la relativa vista deve essere opportunamente aggiornata.

Conseguentemente, è possibile definire l'elenco delle teorie attive e l'elenco delle teorie disponibili come due strutture dati con ruoli distinti ma con lo stesso comportamento, con la possibilità di effettuare lo structure sharing di teorie. Così facendo, una teoria disponibile può anche essere una teoria attiva.

A tal fine le teorie dovrebbero essere immutabili; purtroppo così non è, perché alle teorie logiche implementate in tuProlog è possibile aggiungere le clausole di

un'altra teoria. Nonostante ciò, le teorie logiche tuProlog saranno considerate immutabili, non utilizzando la funzionalità di estensione di una teoria descritta prima, se non nei casi in cui una teoria non è certamente contenuta in uno dei due elenchi di teorie. Conseguentemente, nelle eventualità tali che l'utente apporti modifiche al nome o al contenuto di una teoria, dovrà essere generata una nuova teoria.

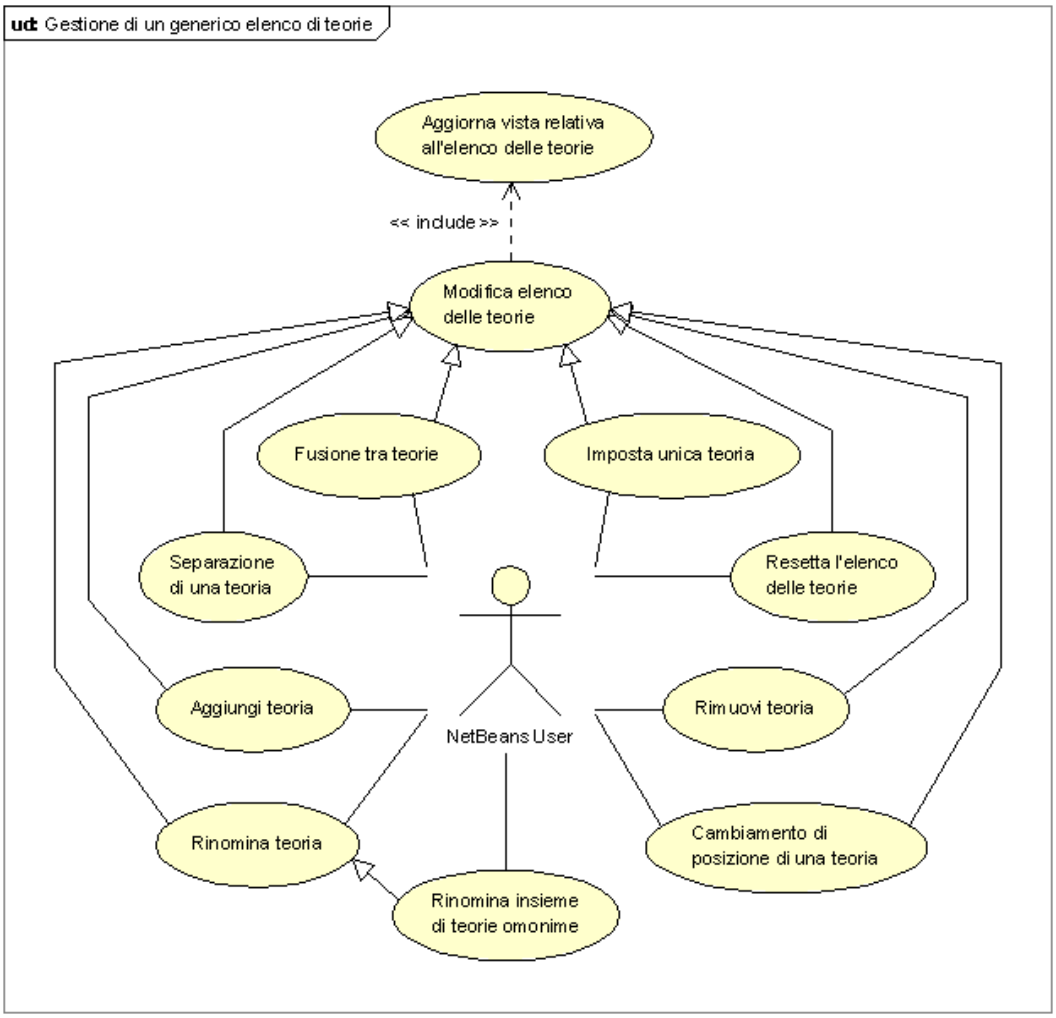


Figura 3.3: Diagramma dei casi d'uso relativo alla gestione di un generico elenco di teorie

3.5 Gestione delle librerie

La gestione delle librerie è molto più semplice rispetto alla gestione delle teorie multiple e della persistenza delle teorie; infatti le librerie non sono modificabili e il loro ordine di caricamento è ininfluente. Le operazioni eseguibili relativamente alla gestione delle librerie sono l'aggiunta e la rimozione di una libreria dal motore tuProlog.

Per aggiungere o rimuovere una libreria al motore tuProlog l'utente deve indicare il nome della classe Java che implementa la libreria interessata, completo del nome del package; a tal fine, devono essere disponibili nel dominio dell'ambiente i file *jar* che contengono il bytecode della libreria, oltre ai file *jar* che contengono il bytecode delle classi di cui la classe che implementa la libreria è dipendente. Ne consegue che bisogna riuscire ad ottenere una valida integrazione delle librerie tuProlog con l'ambiente di appartenenza del plug-in. Tale integrazione potrebbe essere fatta in modo tale che nell'ambiente sia presente un *elenco delle librerie disponibili*; questo elenco non è manipolabile se non integrando nell'ambiente ulteriori librerie tuProlog. Tale elenco deve essere unico.

L'utente potrebbe scegliere le librerie da utilizzare per la risoluzione di una query selezionando le librerie da include o da escludere nella valutazione di una query. Le librerie incluse/escluse nella valutazione saranno immediatamente aggiunte/rimosse dal motore tuProlog.

In base a queste considerazioni, è possibile stabilire che occorre un componente grafico in grado di rappresentare l'elenco delle librerie disponibili nel motore tuProlog, indicando, per ciascuna libreria, se essa è inclusa o esclusa dalla valutazione di query; tale componente sarà denominato *vista delle librerie*. L'indicazione e il controllo relativamente alla inclusione o esclusione dalla valutazione di query di una libreria devono essere affidati ad una casella di selezione. Ad ogni operazione di aggiunta o rimozione di una libreria dal motore tuProlog, la vista delle librerie deve essere aggiornata.

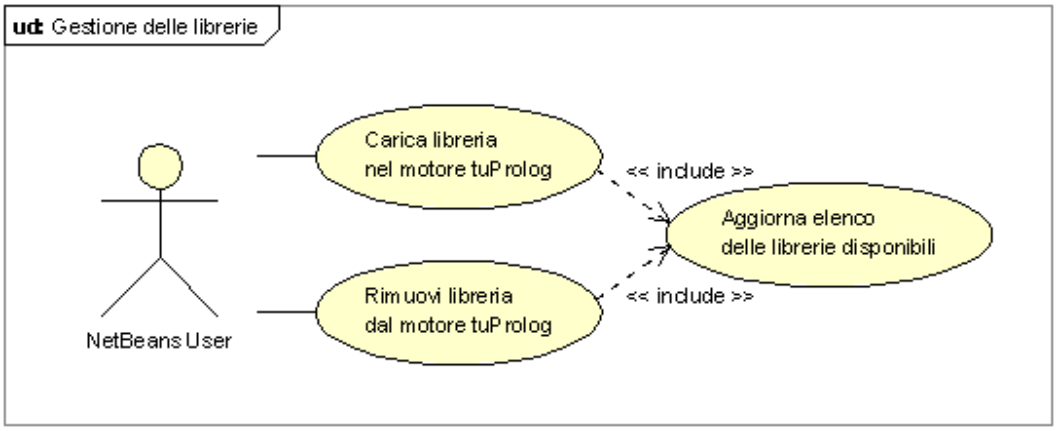


Figura 3.4: Diagramma dei casi d'uso relativo alla gestione delle librerie

3.6 Adattamento del motore tuProlog al plug-in

L'obiettivo principale relativamente all'adattamento del motore tuProlog nel plug-in è quello di avere un unico motore tuProlog disponibile su necessità. Si noti che il plug-in prevede delle funzionalità avanzate non supportate dal motore tuProlog, quindi si potrebbe ipotizzare la realizzazione di un *motore tuProlog esteso*, in grado di fornire anche le funzionalità avanzate.

Per quanto riguarda il rapporto tra vista delle teorie attive e motore tuProlog esteso, è possibile avanzare due proposte:

1. *il contenuto della vista delle teorie attive corrisponde fedelmente alle teorie caricate all'interno del motore tuProlog, escludendo le teorie non selezionate per la valutazione*: ogni modifica all'elenco delle teorie nella vista implica l'immediato aggiornamento delle teorie caricate nel motore tuProlog;
2. *il contenuto della vista delle teorie attive è disaccoppiato rispetto alle teorie caricate all'interno del motore tuProlog*: nel momento in cui si richiede una query, le teorie caricate nel motore tuProlog che non fanno più parte delle teorie attive, o che sono state solamente escluse dalla valutazione della query, devono essere rimosse, dopodiché si devono impostare quelle indicate dall'utente e che non sono caricate all'interno del motore.

La prima proposta è la più semplice; sarebbe adatta per la realizzazione di una versione preliminare del plug-in con le sole funzionalità supportate.

La seconda proposta implica la definizione di un modello dell'elenco delle teorie attive al di fuori del motore tuProlog; tale modello deve essere dotato di un'interfaccia che permetta una gestione più flessibile delle teorie rispetto al motore tuProlog. In questo caso l'esecuzione di una query può essere più onerosa a causa della necessaria inizializzazione delle teorie del motore tuProlog.

Si deve tenere in considerazione che il motore tuProlog può manipolare l'elenco delle teorie attive durante la risoluzione di una query, quindi la vista delle teorie attive deve essere eventualmente aggiornata; sono disponibili tre opzioni:

- aggiornare la vista ogni volta che il motore tuProlog notificherà un evento relativo ad una modifica dell'elenco delle teorie;

- aggiornare la vista al termine dell'esecuzione di ogni query o della notifica di un risultato;
- aggiornare la vista al termine dell'esecuzione di una query se, eventualmente, il motore tuProlog ha notificato un evento relativo ad una modifica dell'elenco delle teorie.

Durante l'esecuzione della risoluzione di una query non dovrebbe essere possibile la modifica del contenuto della vista delle teorie attive, né il recupero di una o più teorie da parte dell'utente; pertanto le funzionalità della vista delle teorie attive devono essere disabilitate durante l'esecuzione della risoluzione di una query.

Nel caso in cui la risoluzione di una query sia caratterizzata dalla possibile esistenza di soluzioni multiple, tra la ricerca di una soluzione e l'altra il contenuto della vista delle teorie attive deve corrispondete esattamente all'elenco delle teorie caricate nel motore tuProlog; pertanto, non deve essere possibile la manipolazione del contenuto delle teorie attive da parte dell'utente fintanto che non viene accettata una soluzione o le possibili soluzioni sono esaurite.

Per quanto riguarda la risoluzione di *query temporanee*, le teorie caricate nel motore tuProlog dovranno essere tutte rimosse in tre casi :

- nel caso in cui l'utente accetti una soluzione;
- nel caso in cui l'utente interrompa la risoluzione di una query;
- nel caso in cui non siano disponibili altre soluzioni;

Ciò comporta che all'atto della richiesta di esecuzione di una nuova query, si debbano ricaricare le teorie attive non escluse dalla valutazione nel motore tuProlog.

Nel caso in cui si eseguissero una serie di query non temporanee, e se tra l'una e l'altra di queste l'utente effettuasse delle modifiche all'elenco delle teorie attive, le teorie oggetto delle modifiche dovranno essere ricaricate nel motore tuProlog; nel caso in cui l'utente non effettui modifiche all'elenco delle teorie attive, non dovranno essere caricate delle teorie nel motore tuProlog.

3.7 Architettura logica

In base a quanto stabilito dall'analisi del problema, è possibile delineare l'architettura logica, rappresentata nella figura 3.5.

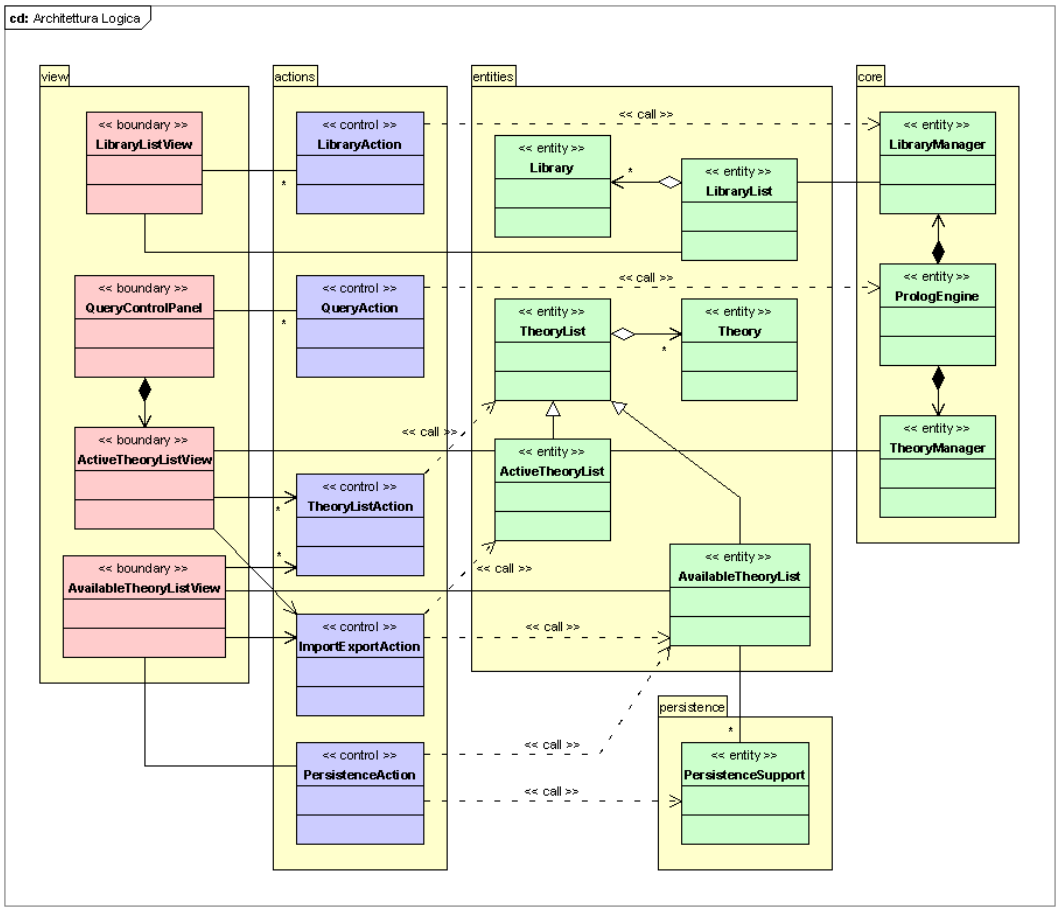


Figura 3.5: Architettura logica

I componenti dell'architettura logica e i relativi ruoli sono:

- *PrologEngine*: il motore tuProlog;
- *TheoryManager*: gestore delle teorie attive del motore tuProlog;
- *LibraryManager*: gestore delle librerie del motore tuProlog;
- *Theory*: modello di una teoria tuProlog;
- *TheoryList*: modello di un generico elenco di teorie;
- *ActiveTheoryList*: modello dell'elenco delle teorie attive;
- *AvailableTheoryList*: modello dell'elenco delle teorie disponibili;
- *Library*: modello di una libreria tuProlog;
- *LibraryList*: modello dell'elenco delle librerie disponibili;
- *PersistenceSupport*: supporto per la persistenza delle teorie;
- *QueryControlPanel*: vista che costituisce il pannello di controllo per la

risoluzione di query;

- *ActiveTheoryListView*: vista relativa all'elenco delle teorie attive;
- *AvailableTheoryListView*: vista relativa all'elenco delle teorie disponibili;
- *LibraryListView*: vista relativa alle librerie disponibili nell'ambiente;
- *QueryAction*: generica azione nell'ambito della risoluzione di query;
- *TheoryListAction*: generica azione nell'ambito della gestione di una generica lista di teorie;
- *PersistenceAction*: generica azione che permette di gestire la persistenza di teorie;
- *ImportExportAction*: generica azione che consente di spostare una teoria da un elenco di teorie verso un altro elenco di teorie;
- *LibraryAction*: generica azione nell'ambito della gestione delle librerie.

Per classificare i ruoli dei vari componenti nel diagramma UML dell'architettura logica, è stato seguito il metodo proposto da Jacobson [JCJO92], il quale suggerisce di suddividere gli oggetti in tre categorie:

- *boundary objects*: oggetti che costituiscono l'interfaccia del sistema;
- *control objects*: oggetti che incapsulano la logica di controllo di insiemi di oggetti; costituiscono di fatto il collante degli oggetti del sistema;
- *entity objects*: oggetti che includono le funzionalità relative alle informazioni del sistema.

Per contraddistinguere meglio il ruolo delle varie classi, nei diagrammi UML di questa tesi verrà utilizzata una specifica colorazione:

- *boundary*: magenta chiaro;
- *control*: azzurro chiaro;
- *entity*: verde chiaro;
- *interfacce relative a listener*: ciano chiaro;
- *package interni al sistema*: giallo;
- *package esterni al sistema*: grigio chiaro.

3.8 Piano di lavoro

Lo sviluppo del plug-in di tuProlog per l'ambiente NetBeans sarà caratterizzato da un processo guidato dai casi d'uso, suddiviso in più fasi; al termine di ciascuna fase sarà ottenuto un prototipo coerente, dotato di determinate caratteristiche; il prototipo successivo sarà ricavato dal precedente aggiungendo funzionalità o, eventualmente, effettuando delle riprogettazioni mirate.

Ecco l'elenco di tali prototipi e delle loro caratteristiche:

- **Primo Prototipo**: integrazione del motore tuProlog in NetBeans, senza gestione delle teorie attive e senza gestione delle teorie disponibili; si deve avere la possibilità di eseguire query, quindi di visualizzare anche il risultato della query, le informazioni interne e l'output del motore tuProlog.
- **Secondo Prototipo**: integrazione della gestione semplificata delle teorie attive nel plug-in; verrà aggiunto al plug-in un componente grafico che rappresenti l'elenco delle teorie attive e ne permetta la manipolazione; le manipolazioni consentite saranno solamente quelle relative alle funzionalità supportate; le teorie attive potranno essere importate/esportate tramite il filesystem locale.
- **Terzo Prototipo**: integrazione della gestione delle teorie disponibili nel plug-in; le teorie attive non saranno importate/esportate tramite il filesystem locale, ma tramite l'elenco delle teorie disponibili; le funzionalità implementate saranno tutte quelle stabilite nell'analisi del problema escludendo la separazione di una teoria; conseguentemente, dovrà essere possibile l'esecuzione di modifiche alle teorie disponibili, e quindi tale prototipo dovrà integrare un editor con funzionalità minime, ma non sarà ancora possibile eseguire modifiche al contenuto delle teorie attive.
- **Quarto Prototipo**: miglioramento del componente grafico per la gestione delle teorie attive tramite l'integrazione delle funzionalità avanzate, escludendo la separazione di una teoria; miglioramento del componente grafico per l'esecuzione di query tramite l'integrazione della funzionalità di indicazione delle teorie da utilizzare, escludendo la possibilità di eseguire le custom query.

- **Quinto Prototipo**: integrazione della gestione delle librerie.
- **Sesto Prototipo**: integrazione di un editor avanzato delle teorie, in grado di evidenziare la sintassi attraverso la colorazione del testo; integrazione della funzionalità di separazione di una teoria sia per l'elenco delle teorie attive, sia per l'elenco delle teorie disponibili; dovrà essere inclusa la possibilità di eseguire le custom query.

3.9 Nomi delle versioni dei prototipi

Al fine di attribuire un nome non ambiguo ad ogni versione dei vari prototipi, è importante indicare alcune specifiche, semplici ed essenziali, per l'attribuzione del nome di ciascuna versione:

- il nome di ogni versione dei vari prototipi deve essere composto da una sequenza di numeri separati da singoli punti (esempio: 0.1.2);
- dato che il lavoro sarà incentrato esclusivamente su dei prototipi, il primo numero del nome di ogni versione sarà lo zero;
- il secondo numero del nome di ogni versione deve corrispondere al numero ordinale del prototipo, specificato nel piano di lavoro;
- il terzo numero del nome di ogni versione deve identificare la revisione del particolare prototipo considerato; dato che le revisioni saranno sequenziali, questo numero deve partire da zero (prima versione di uno specifico prototipo) e aumentare di una unità per ogni revisione.

Capitolo 4

Primo prototipo

In questo capitolo, partendo dalle considerazioni fatte durante nell'analisi del problema e in base a quanto specificato nel piano di lavoro, sarà sviluppato il progetto del primo prototipo del plug-in tuProlog per l'ambiente NetBeans, e successivamente sarà analizzata anche la sua implementazione.

4.1 Progettazione del primo prototipo

Prima di iniziare a progettare il primo prototipo, è bene notare che il progetto relativo allo sviluppo del motore tuProlog è del tutto separato dal progetto relativo allo sviluppo del corrispondente plug-in per l'ambiente NetBeans. Conseguentemente, è necessario fare in modo che il plug-in e il motore tuProlog siano aggiornabili separatamente. Per raggiungere tale scopo verrà realizzato un *library wrapper module* (vedi sezione 2.2.1) distinto dal plug-in, contenente esclusivamente la libreria (sotto forma di file JAR) relativa al motore tuProlog. Segue che, per implementare il plug-in, non verrà realizzato un semplice modulo, ma una suite di moduli composta da un modulo libreria relativo al motore tuProlog e da un modulo che implementa il vero e proprio plug-in.

La progettazione del primo prototipo si baserà su ciò che suggerisce di realizzare il piano di lavoro:

Primo prototipo: *integrazione del motore tuProlog in NetBeans, senza gestione delle teorie attive e senza gestione delle teorie disponibili; si deve avere la possibilità di eseguire query, quindi di visualizzare anche il risultato della query, le informazioni interne e l'output del motore tuProlog.*

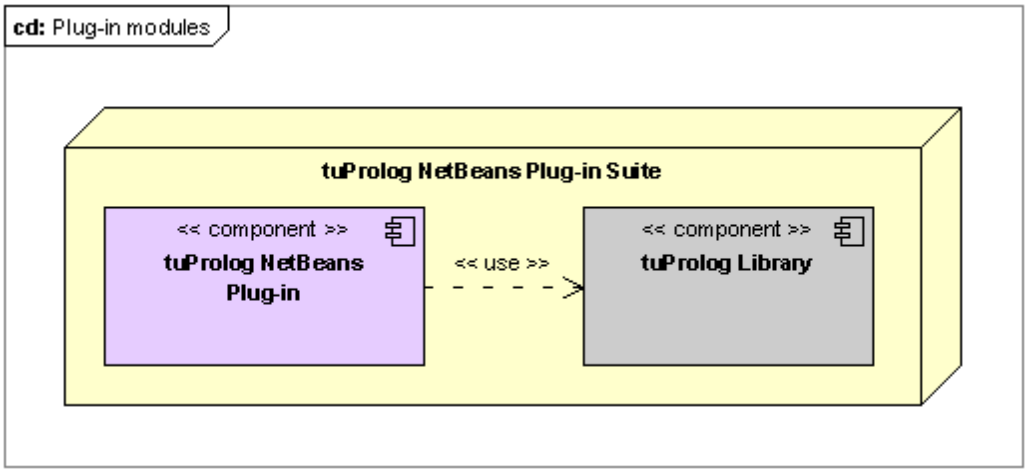


Figura 4.1: Moduli del plug-in

Osservando la figura 3.1, relativa al diagramma dei casi d'uso dal titolo "Gestione della risoluzione di query e gestione delle teorie attive", si deduce che i casi d'uso da realizzare per fare ciò che suggerisce il piano di lavoro sono:

- *esegui query temporanea*;
- *visualizza informazioni provenienti dal motore tuProlog*.

4.1.1 Collaborazione: *esegui query temporanea*

Con riferimento all'analisi del problema, per poter realizzare il caso d'uso "*esegui* query *temporanea*" occorrono:

- un motore tuProlog;
- un componente grafico che sia in grado di gestire l'esecuzione di query, oltre a specificare la query da eseguire e a visualizzarne i corrispondenti risultati.

Inoltre, serviranno anche degli elementi di controllo in grado di realizzare le interazioni tra gli elementi appena elencati. Ogni elemento di controllo implementerà un'azione che può essere richiesta dall'utente tramite interfaccia grafica. Le azioni che può richiedere l'utente sono:

- avviamento della risoluzione di una query;
- terminazione della risoluzione di una query;
- ricerca di un'ulteriore soluzione;
- accettazione di una soluzione.

L'utente gestirà la risoluzione delle query richiedendo l'esecuzione di tali azioni attraverso l'uso di pulsanti, come accennato nell'analisi del problema. Occorrerà anche un ulteriore elemento di controllo in grado di gestire sia l'esecuzione delle varie azioni, sia di ricevere eventuali informazioni in retroazione da parte di esse.

I partecipanti sono:

- *PrologEngine*: il motore tuProlog;
- *QueryControlPanel*: vista che consente di gestire e specificare l'esecuzione di una query e di visualizzarne i corrispondenti risultati;
- *QueryActionController*: controllore relativo all'esecuzione delle azioni;
- *StartAction*: azione che avvia la risoluzione di una query;
- *StopAction*: azione che termina una query;
- *NextAction*: azione che continua la risoluzione di una query cercando una nuova soluzione;
- *AcceptAction*: azione che accetta una soluzione e termina la risoluzione della query.

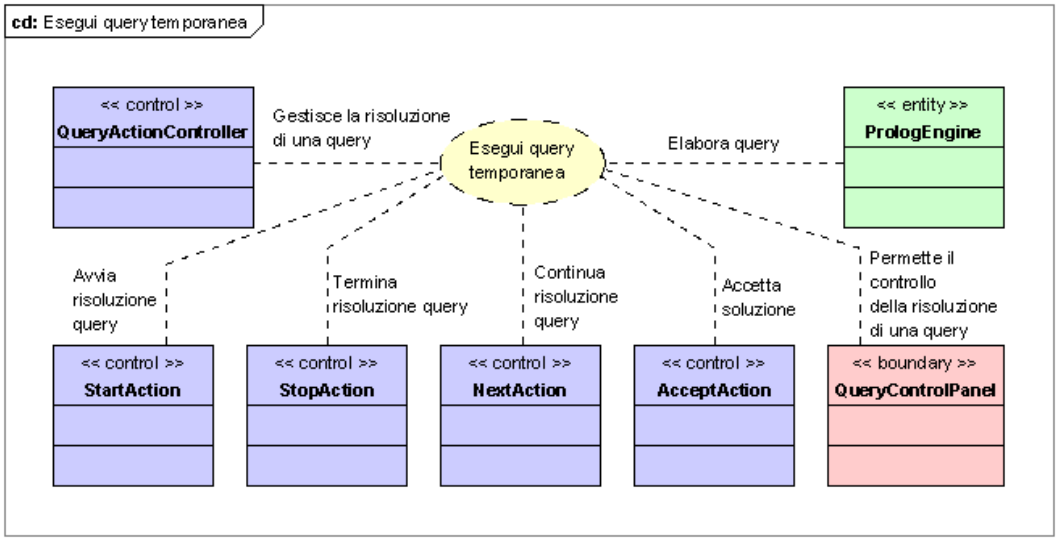


Figura 4.2: Collaborazione relativa al caso d'uso "esegui query temporanea"

Dai ruoli definiti per ogni partecipante, si determinano le relative responsabilità:

- *PrologEngine*:
  - offre il servizio di risoluzione delle query;
  - deve notificare al QueryControlPanel l'individuazione di una soluzione

ad una query;

- *QueryControlPanel*:
  - incapsula i widget grafici che permettono all'utente di specificare e gestire una query, oltre a visualizzarne il risultato;
  - segnala al QueryActionController di richiamare le azioni StartAction, StopAction, NextAction e AcceptAction, su richiesta dell'utente;
- *QueryActionController*:
  - modifica lo stato dei widget grafici incapsulati dal QueryControlPanel;
  - richiede al QueryControlPanel la query da eseguire;
  - richiama le azioni StartAction, StopAction, NextAction e AcceptAction allo scopo di eseguire le corrispondenti operazioni;
  - ogni volta che viene notificata una nuova soluzione deve verificare la possibile presenza di ulteriori soluzioni;
- *StartAction*:
  - richiede al PrologEngine di risolvere una query;
  - può notificare al QueryActionController che una query non è consistente;
  - rimuove tutte le teorie caricate nel PrologEngine nel caso in cui non siano disponibili eventuali ulteriori soluzioni;
- *StopAction*:
  - richiede al PrologEngine di terminare la risoluzione di una query;
  - rimuove tutte le teorie caricate nel PrologEngine;
- *NextAction*: richiede al PrologEngine di cercare una nuova soluzione;
- *AcceptAction*:
  - indica al PrologEngine che la risoluzione di una query è terminata;
  - rimuove tutte le teorie caricate nel PrologEngine.

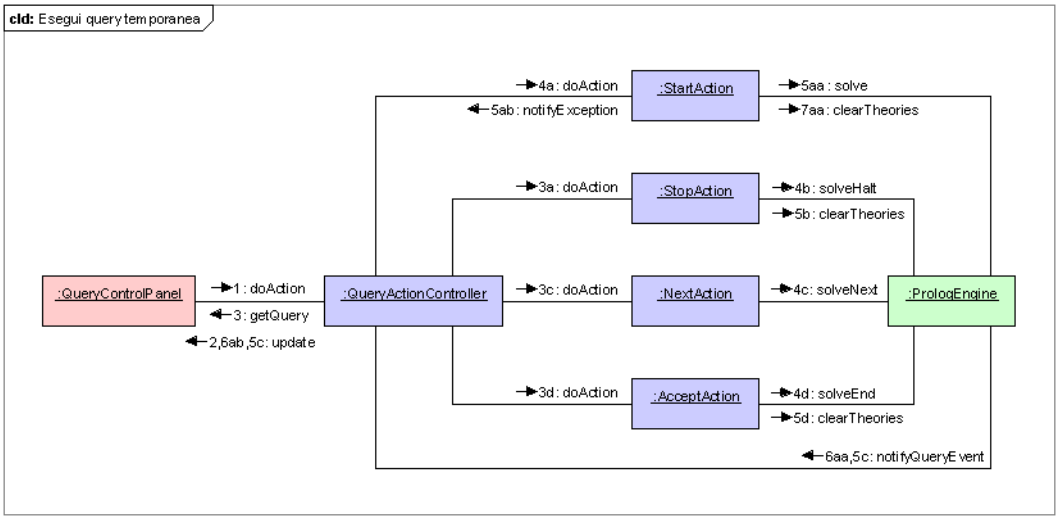


Figura 4.3: Diagramma della collaborazione "esegui query temporanea"

4.1.2 Collaborazione: *visualizza informazioni provenienti dal motore tuProlog*

Con riferimento all'analisi del problema, per poter realizzare il caso d'uso "*visualizza informazioni provenienti dal motore tuProlog*" occorrono:

- un motore tuProlog;
- un componente grafico in grado di visualizzare l'output del motore tuProlog;
- un componente grafico in grado di visualizzare le informazioni interne del motore tuProlog.

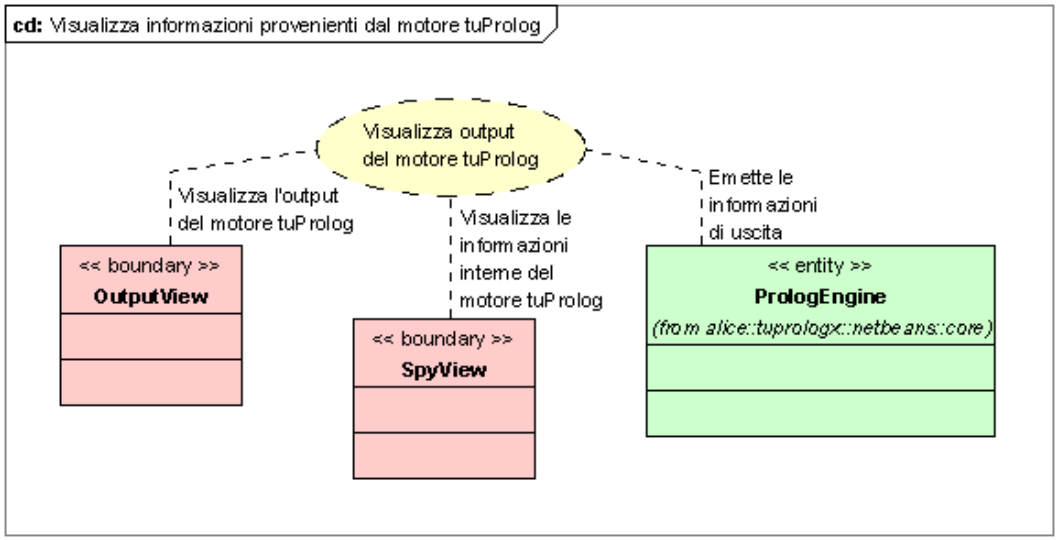


Figura 4.4: Collaborazione relativa al caso d'uso "visualizza informazioni provenienti dal motore tuProlog"

I partecipanti sono:

- *PrologEngine*: il motore tuProlog;
- *OutputView*: vista che visualizza l'uscita del motore tuProlog;
- *SpyView*: vista che visualizza le informazioni interne del motore tuProlog.

Dai ruoli definiti per ogni partecipante, si determinano le relative responsabilità:

- *PrologEngine*:
  - deve notificare ad OutputView l'esistenza di informazioni da inviare in output;
  - deve notificare a SpyView l'esistenza di informazioni interne da inviare in output;
- *OutputView*: riceve gli aggiornamenti relativi all'output dal PrologEngine;
- *SpyView*: riceve gli aggiornamenti relativi agli eventi interni dal PrologEngine.

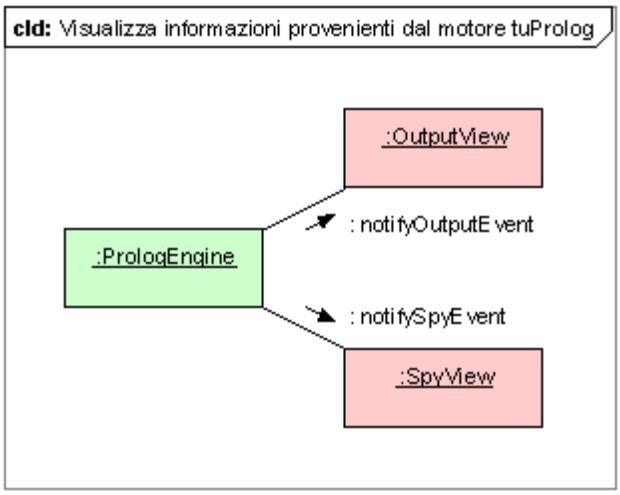


Figura 4.5: Diagramma della collaborazione "visualizza informazioni provenienti dal motore tuProlog"

4.1.3 Interazione: *avviamento della risoluzione di una query*

Tale interazione viene avviata con un comando da parte dell'utente, impartito tramite l'interfaccia grafica implementata dal QueryControlPanel, il quale delegherà al QueryActionController la gestione dell'esecuzione dell'azione. Per prima cosa il QueryActionController deve acquisire il contenuto della query da elaborare, in modo tale da poterlo passare come parametro all'azione StartAction che successivamente sarà lanciata.

La modalità d'esecuzione dell'azione StartAction deve essere asincrona, ovvero la vera e propria azione deve essere eseguita da un thread a parte appositamente creato, questo per evitare che il thread che ha lanciato l'azione rimanga impegnato nella risoluzione della query. Il thread generato, denominato *StartActionThread*, richiederà in modo sincrono la risoluzione della query, e, infine, rimuoverà le teorie caricate se non esisteranno eventuali ulteriori soluzioni. Sarà compito del PrologEngine notificare al QueryActionController il risultato della query, il quale a sua volta provvederà ad aggiornare opportunamente lo stato del QueryControlPanel.

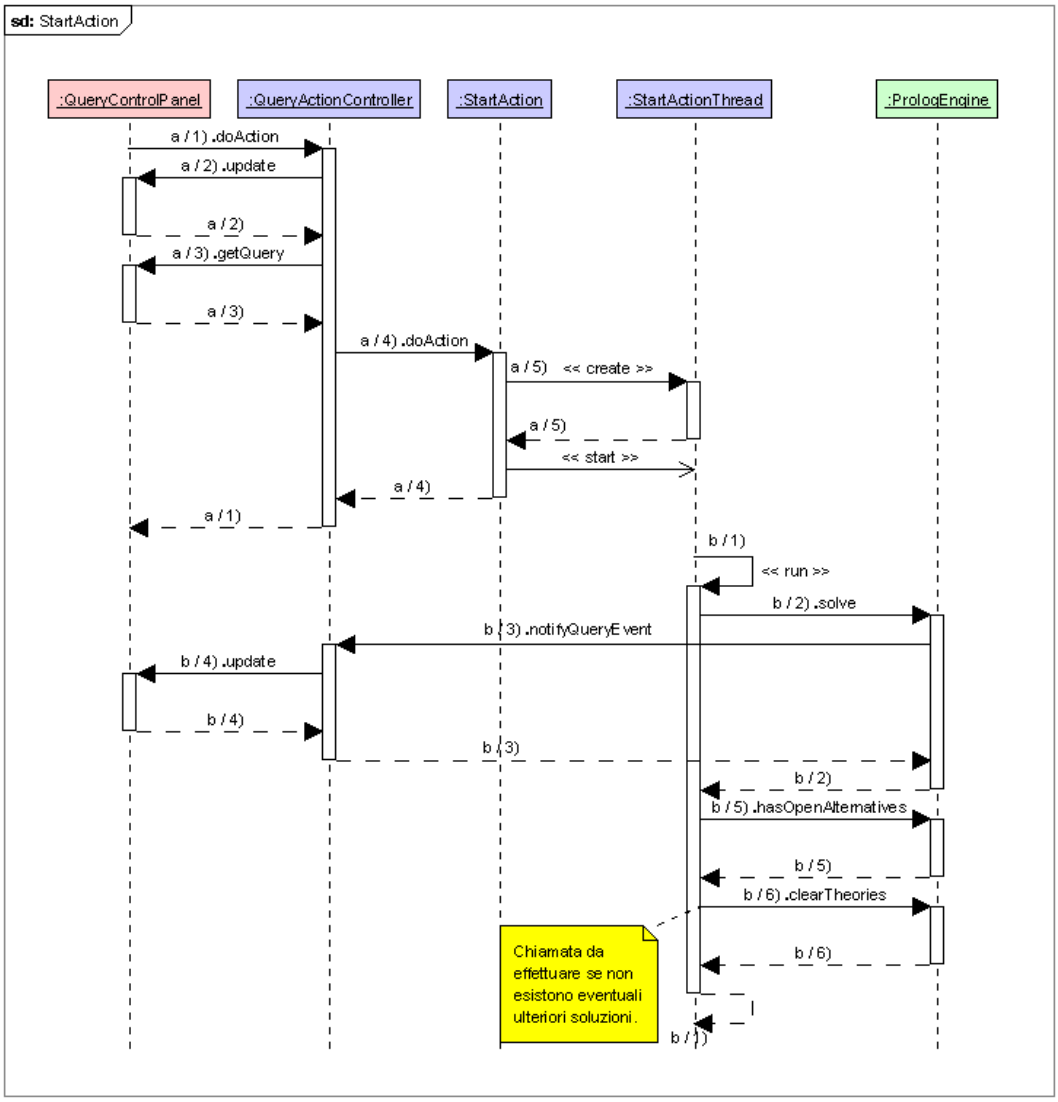


Figura 4.6: Diagramma di sequenza relativo all'interazione "avviamento della risoluzione di una query"

Nel caso in cui il PrologEngine sollevi una eccezione dovuta ad una query inconsistente, questa deve essere notificata al QueryActionController, in modo

tale da poter segnalare l'eventuale eccezione all'utente e modificare opportunamente lo stato del QueryControlPanel.

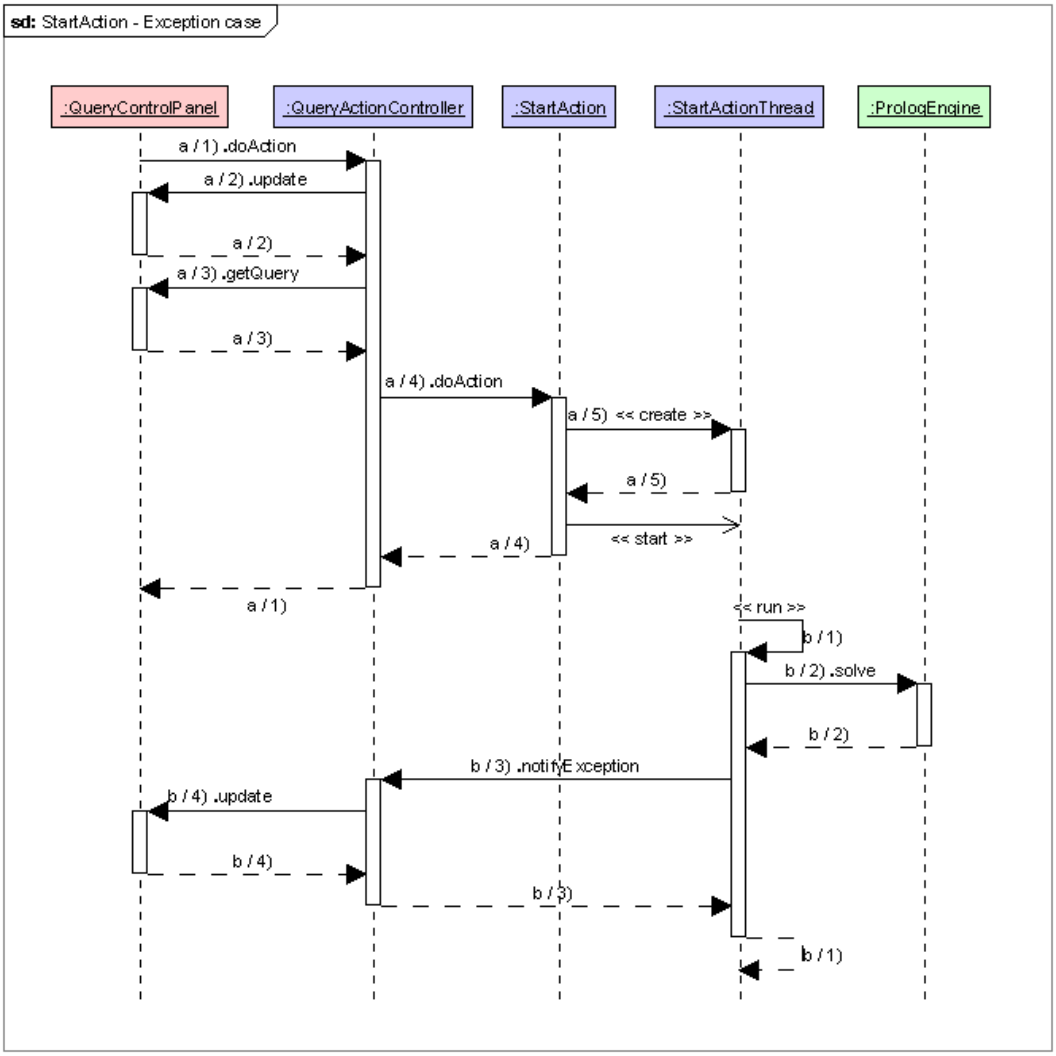


Figura 4.7: Diagramma di sequenza relativo all'interazione "avviamento della risoluzione di una query": caso in cui viene sollevata un'eccezione

4.1.4 Interazione: *terminazione della risoluzione di una query*

Tale interazione viene avviata con un comando da parte dell'utente, impartito tramite l'interfaccia grafica implementata dal QueryControlPanel, il quale delegherà al QueryActionController la gestione dell'esecuzione dell'azione. Il QueryActionController si limiterà a lanciare l'azione StopAction.

Le operazioni eseguite dall'azione StopAction consistono nel richiedere al PrologEngine la terminazione della query e la rimozione delle teorie attive caricate.

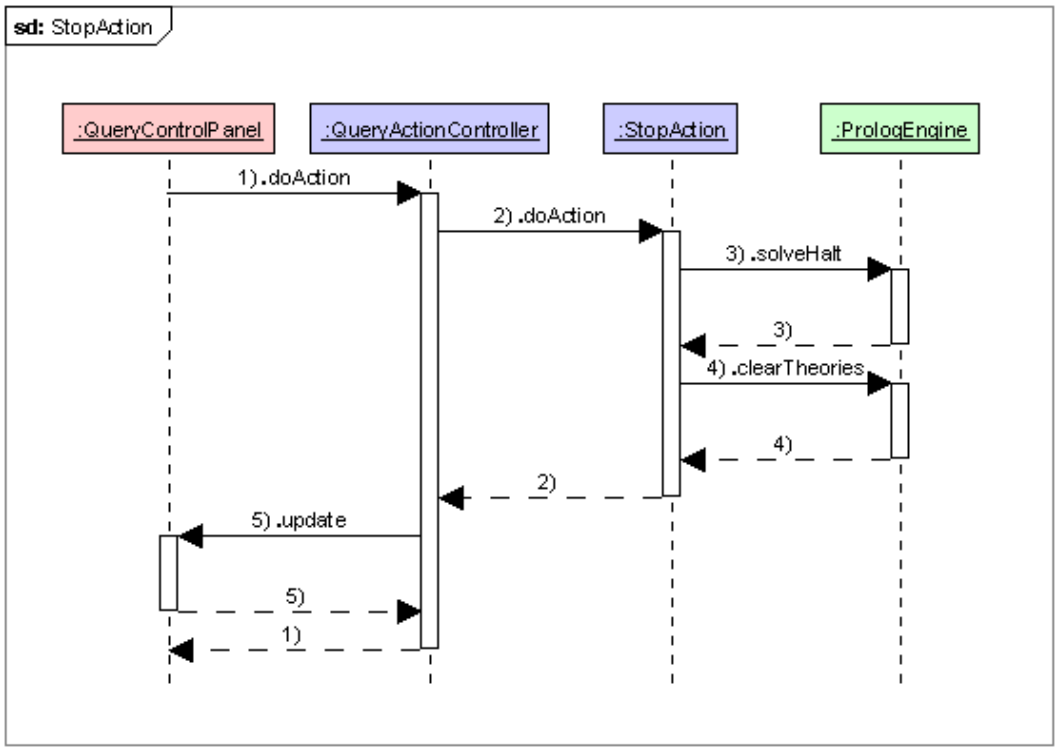


Figura 4.8: Diagramma di sequenza relativo all'interazione "terminazione della risoluzione di una query"

4.1.5 Interazione: *ricerca di un'ulteriore soluzione*

Tale interazione viene avviata con un comando da parte dell'utente, impartito tramite l'interfaccia grafica implementata dal QueryControlPanel, il quale delegherà al QueryActionController la gestione dell'esecuzione dell'azione, il quale lancerà l'azione NextAction.

La modalità d'esecuzione dell'azione NextAction deve essere asincrona, ovvero la vera e propria azione deve essere eseguita da un thread a parte appositamente creato, questo per evitare che il thread che ha lanciato l'azione rimanga impegnato nella risoluzione della query. Il thread generato, denominato *NextActionThread*, si limiterà a richiedere in modo sincrono la ricerca di una ulteriore soluzione. Sarà compito del PrologEngine notificare al QueryActionController il risultato della ricerca di una ulteriore soluzione; questo dovrà verificare la disponibilità di ulteriori soluzioni interpellando il PrologEngine e, infine, aggiornare opportunamente lo stato del QueryControlPanel.

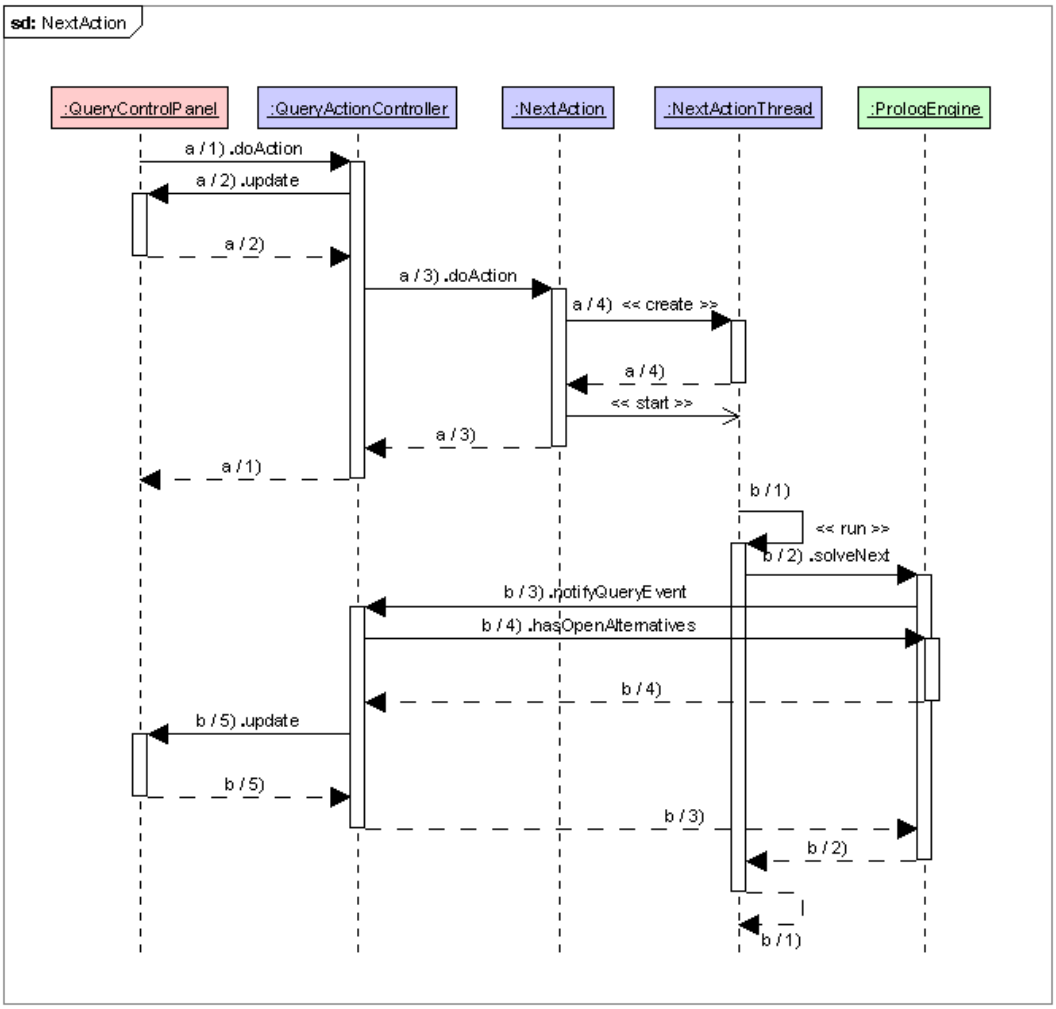


Figura 4.9: Diagramma di sequenza relativo all'interazione: "ricerca di un'ulteriore soluzione"

4.1.6 Interazione: *accettazione di una soluzione*

Tale interazione viene avviata con un comando da parte dell'utente, impartito tramite l'interfaccia grafica implementata dal QueryControlPanel, il quale delegherà al QueryActionController la gestione dell'esecuzione dell'azione. Il QueryActionController si limiterà a lanciare l'azione AcceptAction.

Le operazioni eseguite dall'azione AcceptAction consistono nel richiedere al PrologEngine l'accettazione dell'ultima soluzione trovata e la rimozione delle teorie attive caricate.

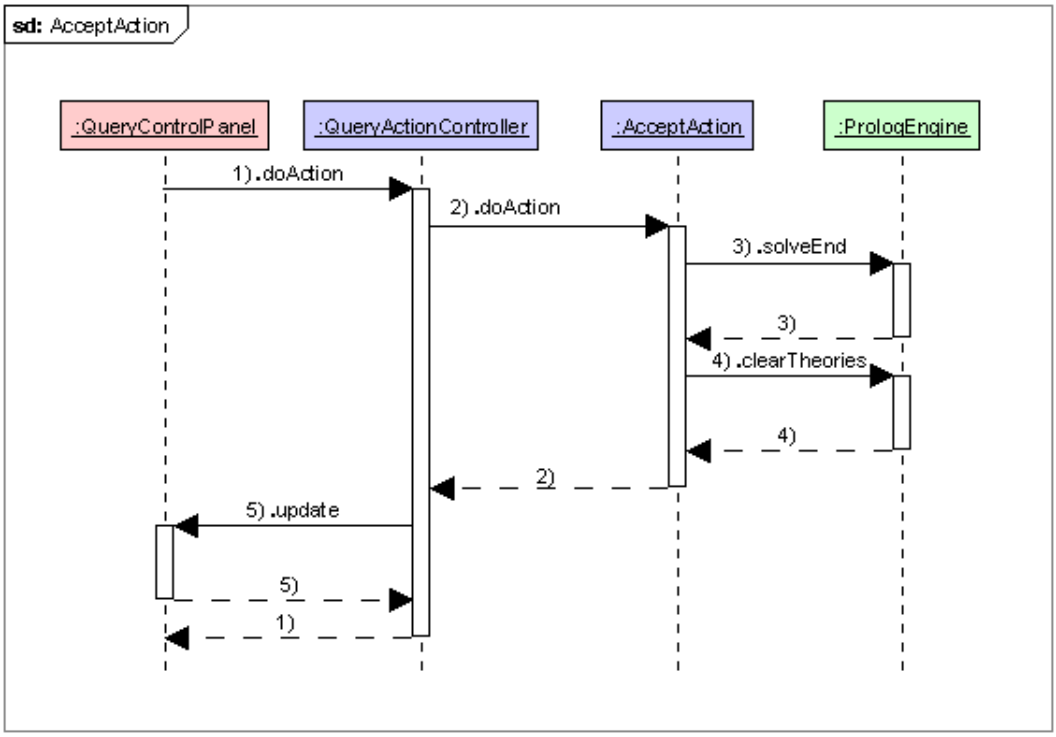


Figura 4.10: Diagramma di sequenza relativo all'interazione "accettazione di una soluzione"

4.1.7 Interazione: *notifica di un evento di output*

Tale interazione consiste semplicemente nella notifica di un evento di output da parte del PrologEngine all'OutputView.

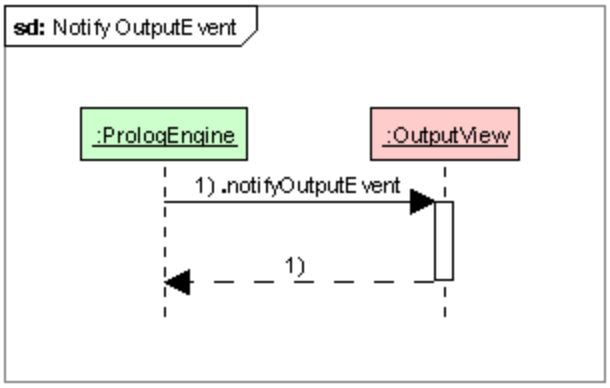


Figura 4.11: Diagramma di sequenza relativo all'interazione "notifica di un evento output"

4.1.8 Interazione: *notifica di un evento interno al motore tuProlog*

Tale interazione consiste semplicemente nella notifica di un evento interno al motore tuProlog da parte del PrologEngine allo SpyView.

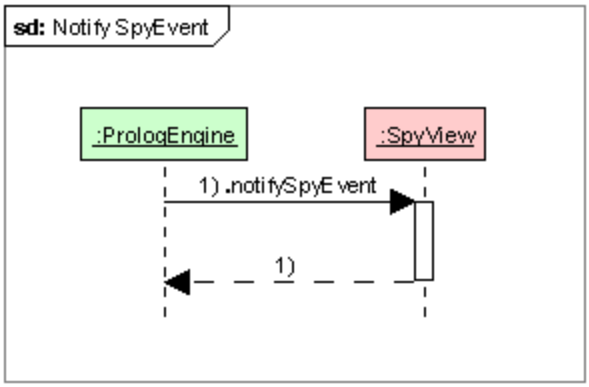


Figura 4.12: Diagramma di sequenza relativo all'interazione "notifica di un evento interno al motore tuProlog"

4.1.9 Progettazione dei partecipanti: *PrologEngine*

Nell'analisi del problema è stato stabilito che il motore tuProlog fosse unico e disponibile su necessità. Quindi è naturale pensare all'uso del design pattern *Singleton* [GoF95] per l'implementazione del motore tuProlog; ma si deve pensare anche a come accedervi.

L'ambiente NetBeans dà la possibilità di registrare nel System Filesystem gli oggetti singleton dichiarandoli all'interno dello *XML Layer* oppure utilizzando il *Java provider extension mechanism*. In entrambi i casi si applicherà il design pattern *Service Locator*[1] (vedi Appendice A), il quale permette di ottenere l'istanza di una classe conoscendo solamente la corrispondente interfaccia o super-classe. L'applicazione di tale design pattern al PrologEngine al momento non sembra necessaria, visto che, per adesso, si prevede di avere un'unica implementazione del motore tuProlog; quindi, non avendo vantaggi ad applicare tale design pattern, si è scelto di non utilizzarlo, in modo tale da non pagare l'overhead conseguente al suo utilizzo. Quindi al PrologEngine verrà applicato il tradizionale design pattern *Singleton*.

Sempre dall'analisi del problema si è stabilito che il plug-in preveda delle funzionalità avanzate non supportate dal motore tuProlog; lo sviluppo del primo prototipo non prevede l'implementazione di funzionalità avanzate, ma è meglio tenere in considerazione fin da subito questo requisito. Quindi, è naturale pensare di incapsulare il motore tuProlog in una classe che aggiunga le funzionalità avanzate al motore, applicando il design pattern *Decorator* [GoF95]. Segue che il

1 Nel tutorial dal titolo "Introduction to the NetBeans Idioms and Infrastructure", presente nel sito Internet dedicato all'ambiente NetBeans, tale design pattern è stato confuso con il design pattern *Dependency Injection*; per chiarimenti vedere [Fow04].

PrologEngine dovrà incapsulare un'istanza della classe `Prolog`, la quale implementa il motore tuProlog vero e proprio. Purtroppo, c'è un problema: il design pattern *Decorator* prevede che l'oggetto incapsulato e l'oggetto decorator incapsulante facciano parte della stessa gerarchia di classi, ovvero dovrebbero realizzare un'interfaccia comune o implementare la stessa classe astratta; purtroppo la classe `Prolog` è una classe concreta che non realizza un'interfaccia che specifica il comportamento di un motore tuProlog. Conseguentemente, quello che verrà realizzato non sarà una vera e propria applicazione del design pattern *Decorator*, anche se ne seguirà le linee guida e gli obiettivi.

Il motore tuProlog ha il compito di notificare vari tipi di eventi ai corrispondenti listener, inviando un oggetto contenente le informazioni relative all'evento; per rispettare le linee guida del design pattern *Decorator*, si deve dotare il PrologEngine delle funzionalità di gestione relative alla aggiunta/rimozione dei listener, ma in realtà i listener saranno registrati nel motore tuProlog vero e proprio, ovvero presso l'istanza incapsulata della classe `Prolog`.

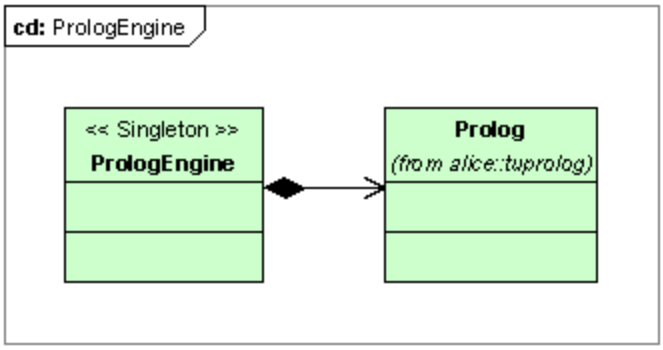


Figura 4.13: Diagramma della classe PrologEngine

4.1.10 Progettazione dei partecipanti: *QueryControlPanel* e *QueryActionController*

Nell'analisi del problema è stato notato che occorre un componente grafico che sia in grado di gestire, specificare e visualizzare i risultati di una query; inoltre, si è stabilito che tale componente deve essere composto dai seguenti widget:

- un pulsante per l'esecuzione di una query;
- un pulsante per la terminazione di una query;
- un pulsante per la ricerca di una ulteriore soluzione;
- un pulsante per l'accettazione della soluzione corrente;
- un campo di testo per l'inserimento della query;

- un'area di testo per la visualizzazione dei risultati della query;
- un radio button per la scelta dell'operatore logico da utilizzare per la selezione di teorie;
- una casella di selezione per scegliere se gli effetti di una query siano temporanei o meno.

Per la realizzazione del primo prototipo non occorrono gli ultimi due componenti; pertanto la loro realizzazione è rimandata ad un successivo prototipo.

Per quanto riguarda tutti gli altri componenti, essi dovranno essere parte integrante del QueryControlPanel, quindi è naturale pensare di applicare il design pattern *Composite* [GoF95], in modo tale che il QueryControlPanel faccia parte della stessa gerarchia di oggetti che implementano le interfacce grafiche tipiche dell'ambiente NetBeans.

Gli oggetti integrati nel QueryControlPanel non sono pochi e lo stato di ciascuno di essi è influenzato dallo stato degli altri, determinando una rete di relazioni complessa; conseguentemente è bene che le classi utilizzanti il QueryControlPanel non facciano delle ipotesi sulla sua composizione e sulle relazioni dei suoi componenti. Per questo motivo verrà applicato anche il design pattern *Façade* [GoF95], in modo tale che il QueryControlPanel disponga di un'interfaccia che renda la vista molto più semplice da utilizzare.

Il QueryControlPanel è in grado di lanciare quattro tipi di *eventi relativi alla gestione della risoluzione di una query*; essi sono:

1. *StartEvent*: pressione del pulsante per l'esecuzione di una query;
2. *StopEvent*: pressione del pulsante per la terminazione di una query;
3. *NextEvent*: pressione del pulsante per la ricerca di una ulteriore soluzione;
4. *AcceptEvent*: pressione del pulsante per l'accettazione della soluzione corrente.

Dato che esso è in grado di lanciare degli eventi, dovrà anche essere in grado di gestire i listener di tali eventi, pertanto la sua classe sarà dotata dei metodi che consentiranno la registrazione/cancellazione e la notifica degli eventi.

I pulsanti che permettono di lanciare gli eventi di gestione della risoluzione di una query devono essere opportunamente attivati, rispettando le seguenti regole:

1. se il pulsante per l'esecuzione di una query è attivo, gli altri pulsanti di gestione della risoluzione di una query devono essere disabilitati;
2. se il pulsante per la terminazione di una query è attivo, gli altri pulsanti di gestione della risoluzione di una query devono essere disabilitati;
3. se il pulsante per la ricerca di una ulteriore soluzione è attivo, l'unico altro pulsante attivo deve essere quello per l'accettazione della soluzione corrente.

Il campo di testo per l'inserimento della query deve essere attivato prima dell'esecuzione di una nuova query, quindi non durante l'esecuzione della risoluzione di una query e non prima di aver accettato una soluzione in caso di possibile presenza di soluzioni multiple. Tale campo di testo è attivo solamente quando è attivo il pulsante per l'esecuzione di una query. Quindi l'azione relativa al pulsante per l'esecuzione della query potrebbe essere associata all'evento relativo alla pressione del tasto *Invio*, durante la digitazione della query nel campo di testo. Infine, si noti che l'area di testo per la visualizzazione dei risultati della query non deve essere editabile da parte dell'utente.

Al fine di gestire opportunamente l'attivazione/disattivazione dei pulsanti per la gestione della risoluzione di una query e del campo di testo per l'inserimento della query, occorre definire gli eventi che producono il loro cambiamento di stato e le corrispondenti reazioni:

1. *evento*: pressione del pulsante per l'esecuzione di una query o pressione del tasto *Invio* mentre il campo di testo della query è selezionato; *reazione*: disattivazione del pulsante per l'esecuzione di una query, attivazione del pulsante per la terminazione di una query, disattivazione del campo di testo per l'inserimento della query;
2. *evento*: pressione del pulsante per la terminazione di una query; *reazione*: attivazione del pulsante per l'esecuzione di una query, disattivazione del pulsante per la terminazione di una query, attivazione del campo di testo per l'inserimento della query;
3. *evento*: pressione del pulsante per la ricerca di una ulteriore soluzione; *reazione*: disattivazione del pulsante per la ricerca di una ulteriore soluzione, attivazione del pulsante per la terminazione di una query;

4. *evento*: pressione del pulsante per l'accettazione della soluzione corrente; *reazione*: attivazione del pulsante per l'esecuzione di una query, disattivazione del pulsante per la terminazione di una query, disattivazione del pulsante per la ricerca di una ulteriore soluzione, attivazione del campo di testo per l'inserimento della query;
5. *evento*: notifica soluzione, nessun altra soluzione disponibile; *reazione*: attivazione del pulsante per l'esecuzione di una query, disattivazione del pulsante per la terminazione di una query, attivazione del campo di testo per l'inserimento della query;
6. *evento*: notifica soluzione, possibile ulteriore soluzione disponibile; *reazione*: attivazione del pulsante di ricerca di una ulteriore soluzione, attivazione del pulsante per l'accettazione della soluzione corrente, disattivazione del pulsante per la terminazione di una query.

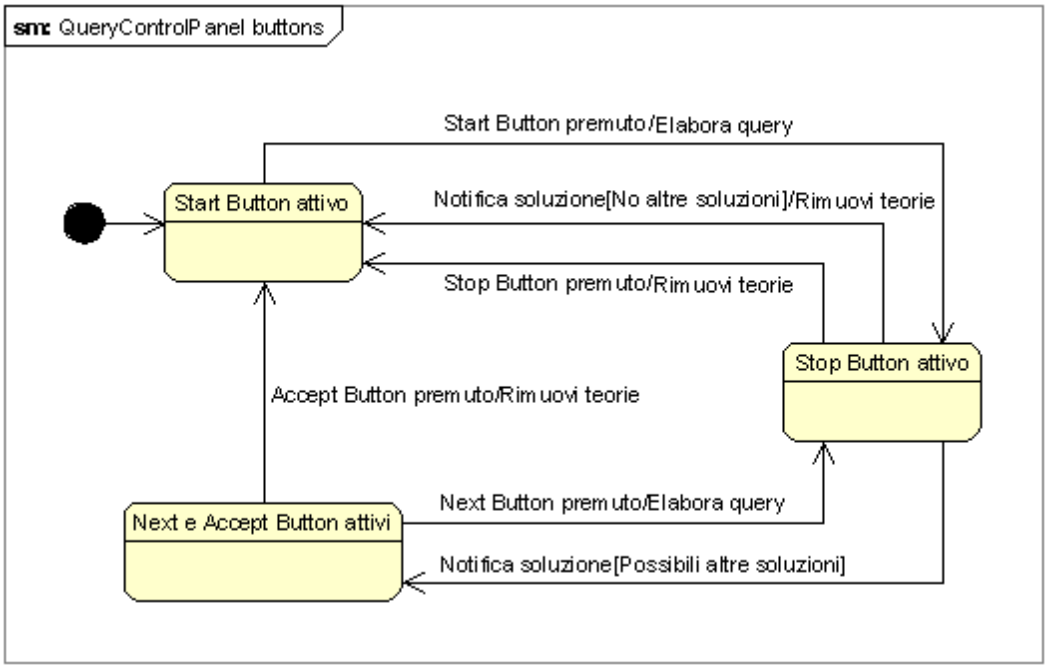


Figura 4.14: Diagramma di stato dei pulsanti per la gestione della risoluzione di una query

Da queste ultime considerazioni si osservi che i pulsanti per la gestione della risoluzione di una query e del campo di testo per l'inserimento della query comunicano tra loro secondo precise regole, ma in modo complesso. Si deve evitare che tali componenti si conoscano tutti tra di loro ed evitare che ognuno di loro contenga la logica di interazione con gli altri componenti. Per far ciò occorre applicare il design pattern *Mediator* [GoF95], in modo tale che ogni componente

conosca e comunichi solamente con un oggetto mediatore e che questo sia l'unico a contenere la logica di interazione tra i componenti. Pertanto sarà l'oggetto mediatore a ricevere gli eventi generati dai vari pulsanti e a predisporre le relative reazioni. Osservando le responsabilità dei vari partecipanti elencate precedentemente, il QueryActionController risulta essere l'entità adatta ad assumere il ruolo di oggetto mediatore.

Per far ciò deve registrarsi come listener degli eventi presso i componenti che producono il cambiamento di stato; quindi in risposta agli eventi generati dai widget, il QueryActionController deve modificare il loro stato e richiamare le azioni StartAction, StopAction, NextAction e AcceptAction, allo scopo di eseguire le corrispondenti operazioni. Una ulteriore responsabilità del QueryActionController riguarda il recupero della query dal relativo campo di testo; tale operazione deve essere eseguita alla ricezione dell'evento *StartEvent* da parte del QueryActionController, in modo tale che la query sia passata come parametro alla StartAction.

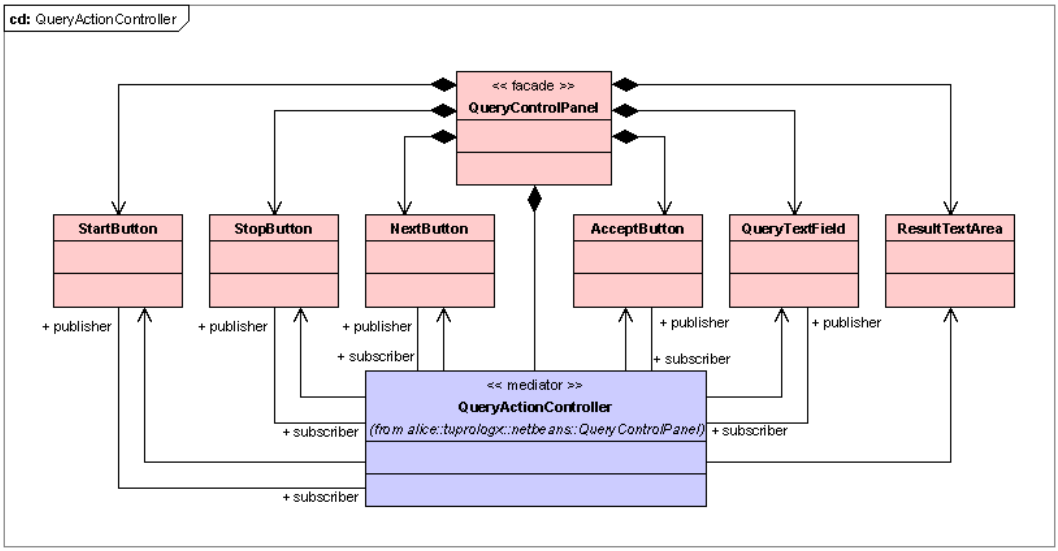


Figura 4.15: Diagramma della classe QueryActionController

In accordo con il design pattern *Façade*, il QueryActionController e tutti i widget che compongono il QueryControlPanel non devono essere visibili esternamente, pertanto saranno creati e associati tra loro nella fase di inizializzazione del QueryControlPanel.

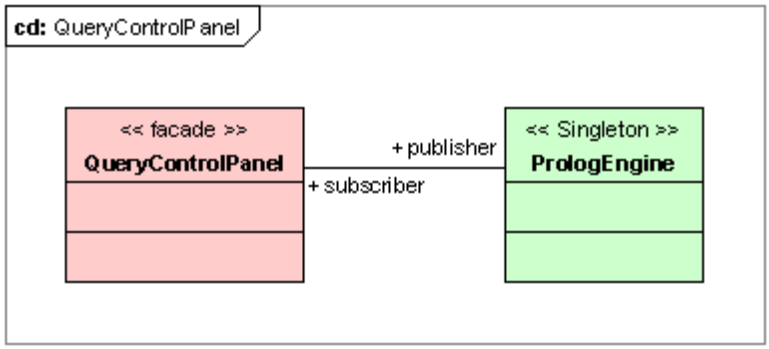


Figura 4.16: Diagramma della classe QueryControlPanel

Al QueryActionController deve essere notificato ogni evento relativo alla risoluzione di una query, in modo tale da poter aggiornare opportunamente i widget del QueryControlPanel; a tal fine dovrebbe essere registrato come listener al motore tuProlog, ma questo violerebbe il design pattern *Façade*. Conseguentemente, sarà il QueryControlPanel ad essere registrato come listener degli eventi relativi alla risoluzione di query, in modo tale che questo inoltri gli eventi ricevuti al QueryActionController. Quindi, al fine di rispettare le direttive del design pattern *Façade*, in fase di inizializzazione il QueryControlPanel deve essere registrato come listener presso il motore tuProlog.

Infine, per poter modificare opportunamente lo stato dei widget, il QueryActionController deve essere a conoscenza della presenza di eventuali ulteriori soluzioni, quindi deve verificare questa eventualità effettuando una chiamata al PrologEngine ogni volta che gli viene notificato un evento relativo alla risoluzione di una query.

Per integrare questa vista nell'ambiente NetBeans, essa dovrà essere inserita come componente all'interno di un'istanza della classe `org.openide.windows.TopComponent`; dato che le istanze della classe `TopComponent` sono degli oggetti singleton, creando una sola istanza della classe che implementa il QueryControlPanel esclusivamente all'interno dell'istanza relativa alla classe `TopComponent`, si avrà la garanzia che esisterà una sola istanza della classe che estende il QueryControlPanel.

4.1.11 Progettazione dei partecipanti: *OutputView* e *SpyView*

Nell'analisi del problema emerse l'esigenza sia di un componente grafico in grado di visualizzare l'output del motore tuProlog, sia di un componente grafico in grado di visualizzarne le informazioni interne. Tali componenti grafici saranno composti

da un'area di testo non editabile per la visualizzazione delle informazioni proveniente dal motore tuProlog, e da un pulsante per la cancellazione del loro contenuto; dato che questi due partecipanti sono costituiti da widget simili, è possibile creare un componente grafico che ne costituisce l'implementazione di base. Tale nuova entità sarà denominata *BlackboardView*.

L'OutputView e lo SpyView si differenziano a causa del fatto che il primo sarà associato al motore tuProlog come listener degli eventi di output, mentre il secondo sarà associato come listener degli eventi interni. Quindi, OutputView e SpyView erediteranno le stesse caratteristiche da BlackboardView, ma si differenzieranno in base agli eventi ascoltati.

Come nel caso del QueryControlPanel, per integrare queste viste nell'ambiente NetBeans, esse dovranno essere inserite come componenti all'interno di un'istanza della classe `org.openide.windows.TopComponent`, garantendo così l'unicità dell'istanze delle classi che implementano l'OutputView e lo SpyView.

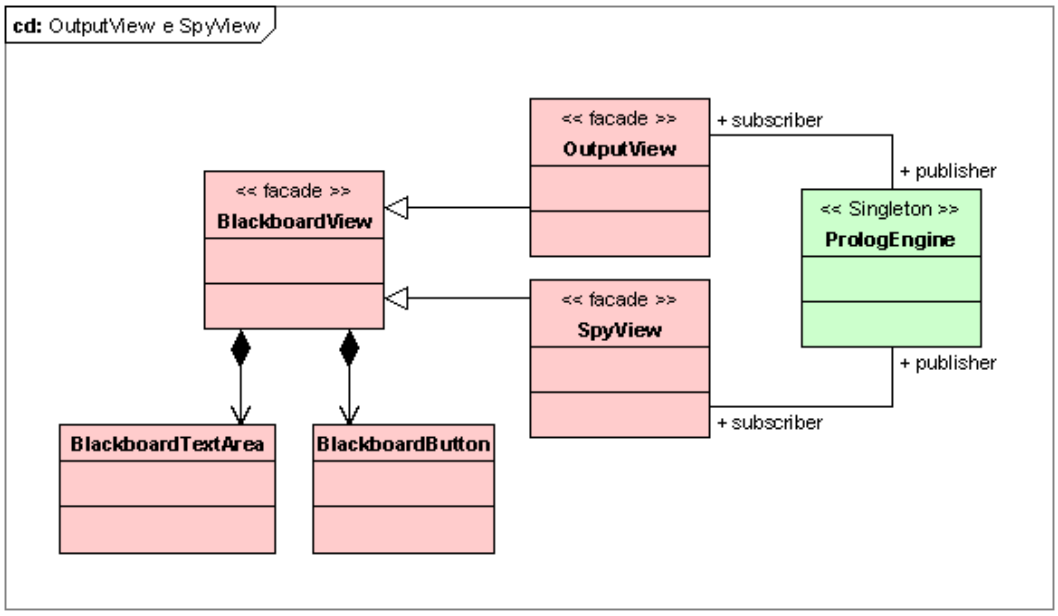


Figura 4.17: Diagramma delle classi OutputView e SpyView

4.1.12 Progettazione dei partecipanti: *StartAction*, *StopAction*, *NextAction* e *AcceptAction*

Si tratta di azioni relative alla gestione della risoluzione di una query, che devono essere eseguite al seguito della pressione di un corrispondente pulsante presente nell'interfaccia grafica del QueryControlPanel. Quindi tali azioni possono essere implementate come listener di eventi e registrate presso il QueryControlPanel.

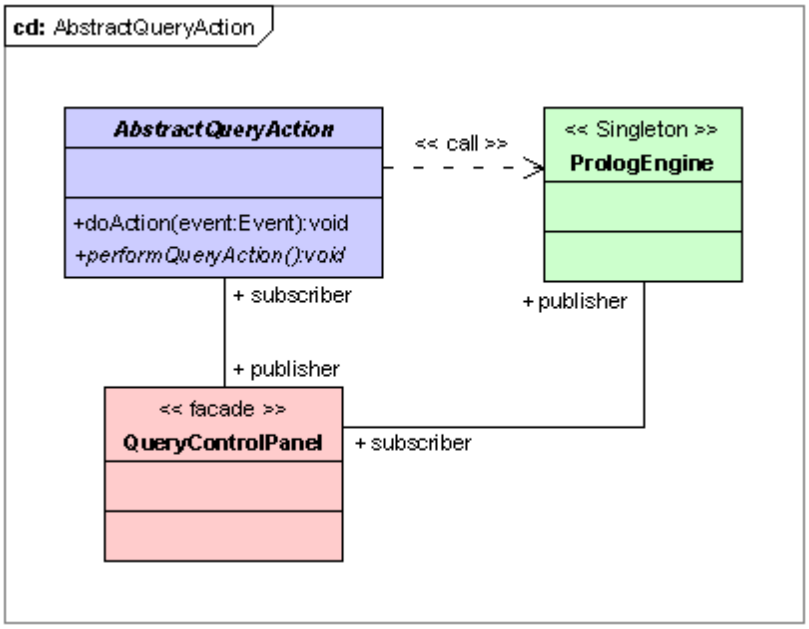


Figura 4.18: Diagramma della classe AbstractQueryAction

Tutte queste azioni collaborano con i soliti partecipanti; l'unica cosa che le differenzia è l'azione propriamente svolta, quindi la loro logica di esecuzione può essere inclusa all'interno di un metodo, applicando il design pattern *Template Method* [GoF95]. L'applicazione di tale design pattern implica la presenza di una gerarchia di classi, le cui sottoclassi si differenziano implementando in modo diverso il metodo che incapsula la logica dell'azione.

L'elemento radice di tale gerarchia di classi sarà la classe astratta `AbstractQueryAction` il cui ruolo è quello di definire il comportamento di base di azioni che influenzano il motore tuProlog.

Come spiegato in precedenza, nei casi della StartAction e della NextAction si dovrà generare un apposito thread per l'esecuzione della vera e propria azione e restituire il controllo all'oggetto chiamante subito dopo il lancio del thread. Quindi StartAction e NextAction possono far parte di una sottogerarchia che a sua volta fa parte della gerarchia delle azioni che influenzano il motore tuProlog; tale sottogerarchia sarà implementata dalla classe astratta `AbstractAsyncQueryAction`.

Inoltre, la StartAction può notificare al QueryActionController se una query non è consistente; tale evenienza può essere trattata sollevando un'eccezione, ma essendo l'azione asincrona (quindi eseguita su di un thread a parte) occorre notificare la query inconsistente attraverso una callback al chiamante dell'azione.

Questo meccanismo potrebbe essere utilizzato anche dalle altre azioni relative alla gestione della risoluzione di una query nei casi in cui si presentino delle evenienze inaspettate.
Le istanze di tali azioni devono essere uniche, quindi ad esse verrà applicato il design pattern *Singleton*.

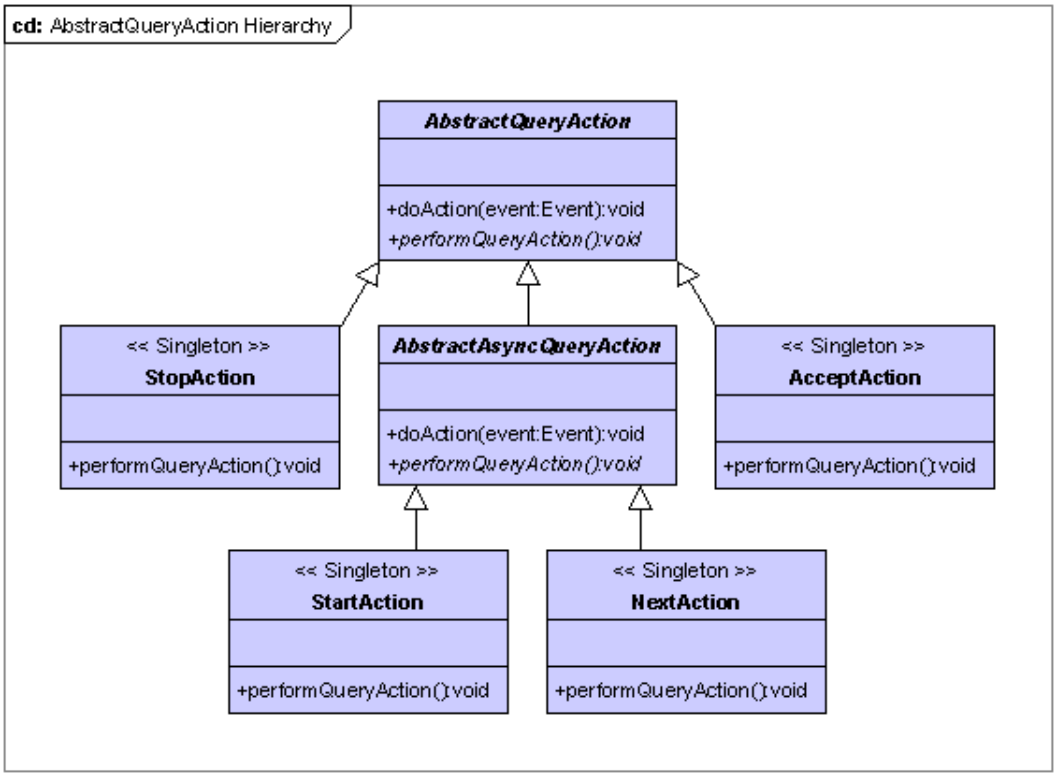


Figura 4.19: Diagramma delle classi relativo alla gerarchia della AbstractQueryAction

4.1.13 Inizializzazione dei partecipanti

Ogni partecipante deve essere opportunamente inizializzato da una determinata classe; per quanto riguarda il PrologEngine, e le azioni relative alla gestione della risoluzione di una query, il design pattern *Singleton* ne predispone l'inizializzazione automatica su necessità.

Gli altri partecipanti devono essere inizializzati in modo tale che non siano presenti reciproche dipendenze; per raggiungere questo obiettivo occorrerà utilizzare una entità a parte che permetta di creare e inizializzare i partecipanti, facendo in modo che solo tale entità abbia una dipendenza con le classi dei partecipanti inizializzati. L'inizializzazione dei partecipanti deve essere fatta in modo tale da permettere la loro interazione, ovvero in questa fase devono essere impostate le loro associazioni, registrando i listener ai corrispondenti publisher.

Come accennato nelle sezioni riguardanti il QueryControlPanel, l'OutputView e lo SpyView, si farà uso della classe `TopComponent` per integrare nell'ambiente NetBeans le viste del plug-in; tale classe potrà essere utilizzata non solo per inizializzare la finestra contenente le interfacce grafiche, ma anche gli altri partecipanti che costituiscono il plug-in, impostando, inoltre, le loro associazioni.

Ogni classe `TopComponent` deve essere affiancata ad un'apposita azione di inizializzazione ad essa associata, richiamabile, ad esempio, tramite una voce di menù; quindi, nel momento in cui l'utente voglia iniziare ad utilizzare il plug-in, non deve far altro che lanciare questa azione richiamando l'apposita voce di menù.

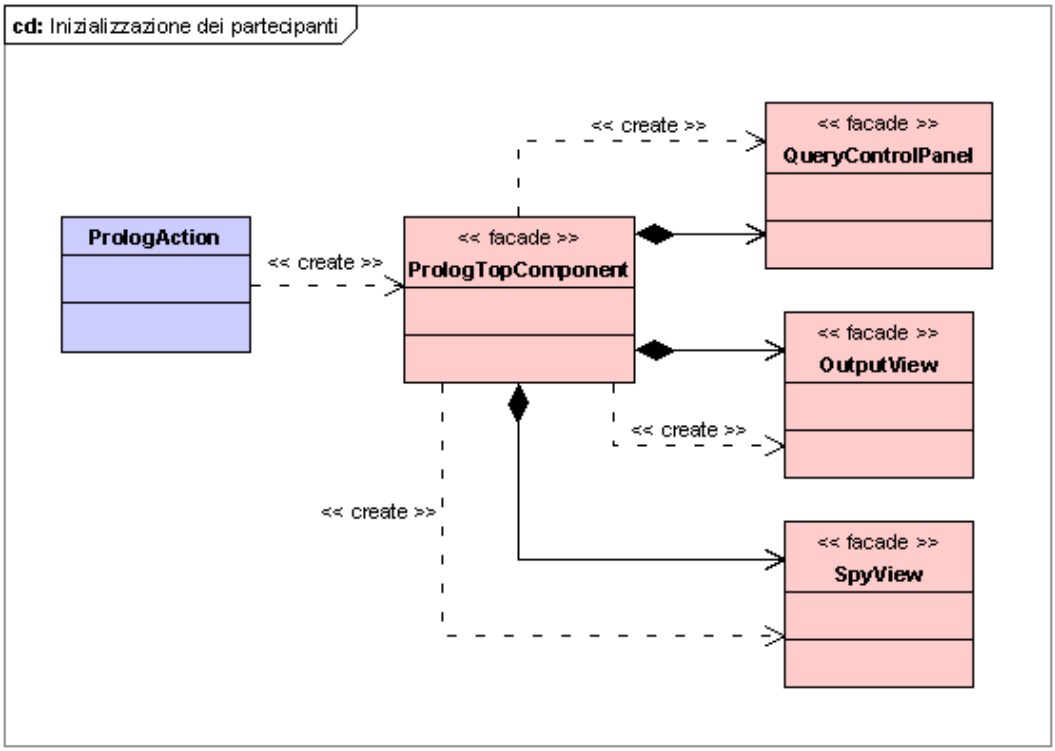


Figura 4.20: Diagramma delle classi relativo all'inizializzazione dei partecipanti

4.2 Architettura del primo prototipo

Alla luce delle precedenti considerazioni, è possibile delineare una prima architettura del sistema, rappresentata nella figura 4.21, a pagina seguente.

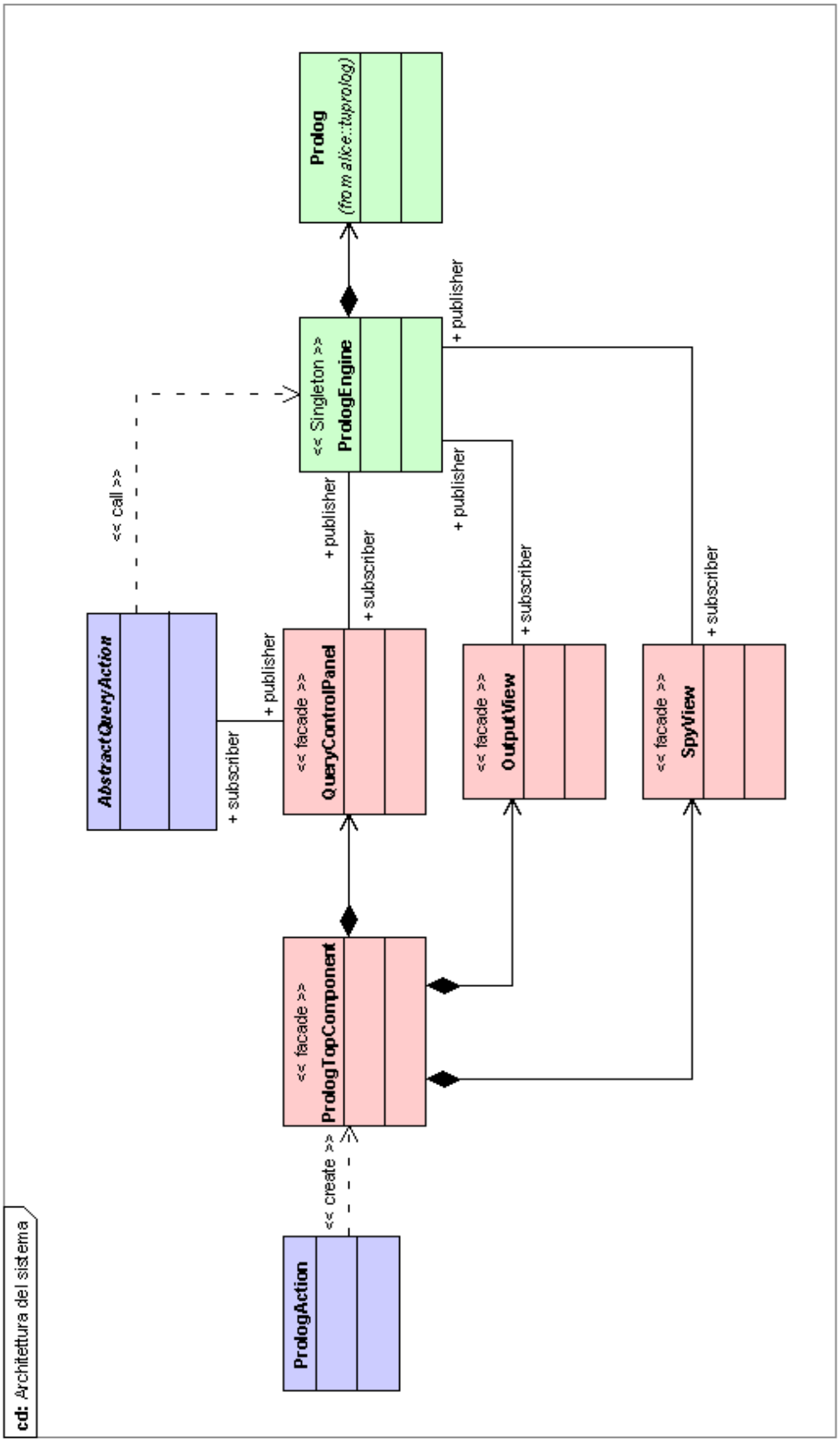


Figura 4.21: Architettura del sistema, relativamente al primo prototipo

4.3 Implementazione del primo prototipo

In base alle indicazioni elaborate nella fase di progettazione, in questa sezione verrà delineata l'implementazione del primo prototipo.

4.3.1 Implementazione del *PrologEngine*

Il nome della classe che implementa l'entità PrologEngine è `alice.tuprologx.netbeans.PrologEngine`. Ad essa è applicato il design pattern *Singleton*, quindi è dotata di:

- un riferimento all'unica istanza della classe, tramite il campo:
  `private static PrologEngine instance;`
- un costruttore a visibilità di tipo *private*:
  `private PrologEngine();`
- un metodo di classe a visibilità *public* che permette il recupero dell'unica istanza della classe; se l'unica istanza della classe non è stata creata, deve provvedere tale metodo:

  `public static synchronized PrologEngine getDefault().`

Inoltre, ad essa è applicato anche il design pattern *Decorator*, quindi incapsulata un'istanza della classe `alice.tuprolog.Prolog`, ampliandone le relative responsabilità; sono stati implementati i metodi caratteristici del motore tuProlog, limitatamente a quelli necessari alla realizzazione del primo prototipo:

- `public SolveInfo solve(String)`
  `throws MalformedGoalException;`
- `public void solveHalt();`
- `public SolveInfo solveNext()`
  `throws NoMoreSolutionException;`
- `public void solveEnd();`
- `public void removeAllTheories().`

La classe `PrologEngine` deve essere dotata dei metodi necessari alla gestione dei listener che realizzano l'interfaccia `alice.tuprolog.QueryListener`, di quelli che realizzano l'interfaccia `alice.tuprolog.OutputListener` e di quelli che realizzano l'interfaccia `alice.tuprolog.SpyListener`.

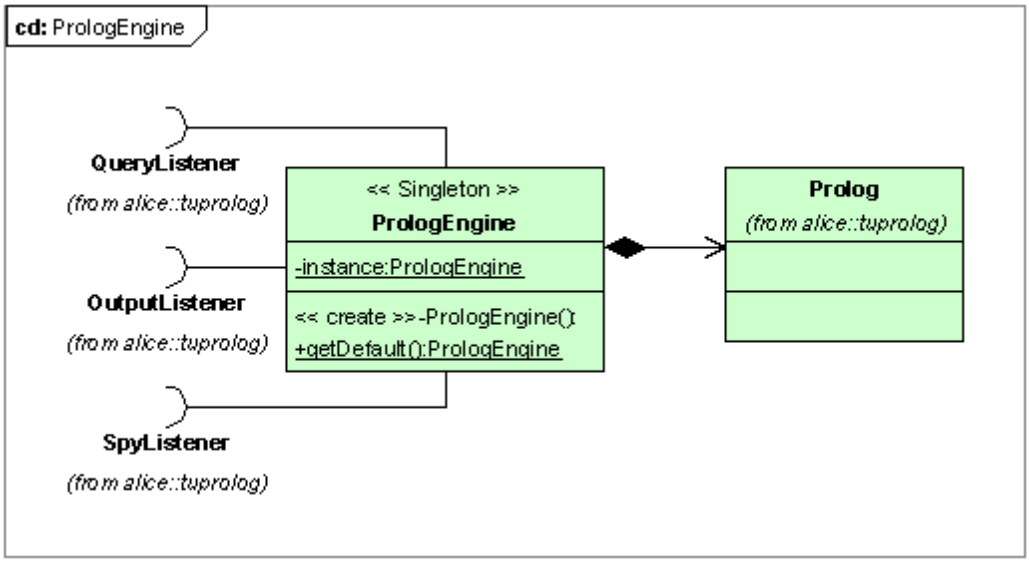


Figura 4.22: Diagramma della classe PrologEngine

4.3.2 Implementazione del *QueryControlPanel* e del *QueryActionController*

Il nome della classe che implementa l'entità QueryControlPanel è `alice.tuprologx.netbeans.QueryControlPanel`. Essa incapsula una serie di widget, i cui corrispondenti campi sono:

- `private javax.swing.JToolBar queryToolBar;`
- `private javax.swing.JButton startButton;`
- `private javax.swing.JButton stopButton;`
- `private javax.swing.JButton nextButton;`
- `private javax.swing.JButton acceptButton;`
- `private javax.swing.JTextField queryTextField;`
- `private javax.swing.JCheckBox tempQueryCheckBox;`
- `private javax.swing.JTextArea resultTextArea.`

Affinché tali widget possano essere visualizzati in modo coerente, essi devono essere inseriti all'interno di un widget che rappresenta un pannello; quindi è stato realizzato il QueryControlPanel in modo tale che sia il suddetto pannello, facendo estende alla classe `QueryControlPanel` la classe `javax.swing.JPanel`. Con questa scelta è stato possibile anche inserire il QueryControlPanel all'interno di un'istanza della classe `org.openide.windows.TopComponent`. Dato che al QueryControlPanel è applicato il design pattern *Façade*, esso non presenta metodi

per l'accesso dei widget che contiene.

L'entità QueryControlPanel è in grado di lanciare eventi relativi alla gestione della risoluzione di una query; tali eventi sono implementati dalla classe `alice.tuprologx.netbeans.QueryActionEvent`, la quale è caratterizzata dal fatto che mantiene un riferimento all'oggetto che ha generato l'evento. Pertanto tale classe implementa la classe `java.awt.event.ActionEvent` che dispone già delle caratteristiche richieste. Inoltre, è stata definita anche un'interfaccia che specifica il comportamento dei listener di questa nuova classe di eventi, denominata `alice.tuprologx.netbeans.IQueryActionListener`; essa è caratterizzata da un unico metodo:

```
public void actionPerformed(QueryActionEvent).
```

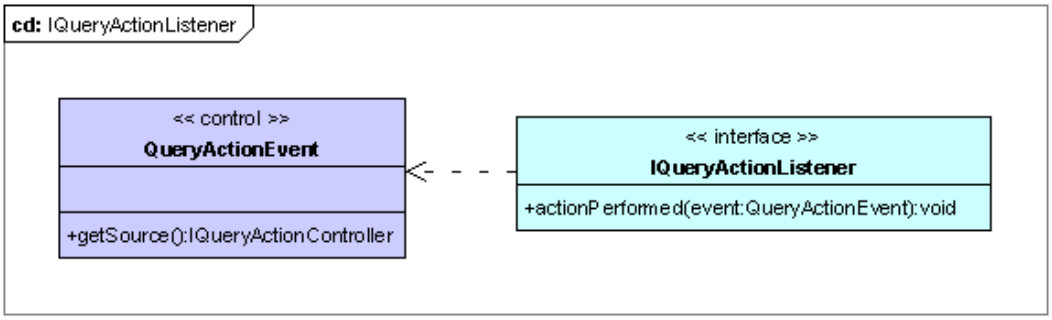


Figura 4.23: Diagramma delle classi relativo all'interfaccia IQueryActionListener

Pertanto la classe `QueryControlPanel` deve presentare gli opportuni metodi per registrare/rimuovere i listener relativi agli eventi per la gestione della risoluzione di una query, in modo separato per ogni tipologia di evento (StartEvent, StopEvent, NextEvent e AcceptEvent).

Tale classe deve essere registrata come listener di eventi relativi alla risoluzione di query presso l'unica istanza della classe `PrologEngine`, pertanto essa deve realizzare l'interfaccia `alice.tuprolog.QueryListener`.

Per poter regolare opportunamente lo stato dei widget incapsulati nel QueryControlPanel (attivandoli e disattivandoli a seconda degli eventi ricevuti), si deve far ricorso ad un oggetto mediatore, implementato da un inner-class della classe `QueryControlPanel`. Il nome di tale classe è `alice.tuprologx.netbeans.QueryControlPanel.QueryActionController`. Si deve ricorrere ad un'inner-class perché i componenti con cui interagire sono tutti interni alla classe `QueryControlPanel` e non visibili esternamente ad essa.

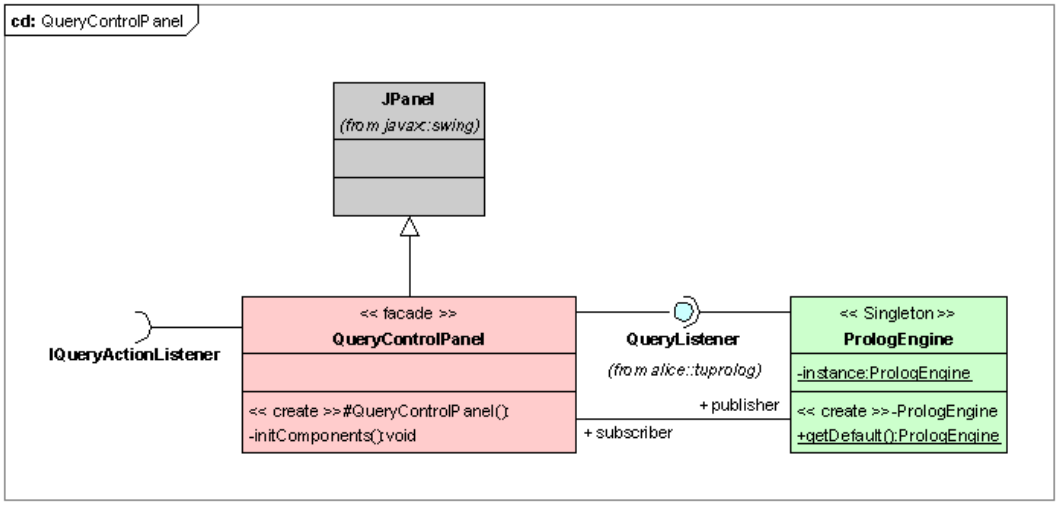


Figura 4.24: Diagramma della classe QueryControlPanel

Dato che tale classe deve essere in grado di ricevere eventi generati da istanze delle classi `JButton` e `JTextField`, essa deve realizzare l'interfaccia `java.awt.event.ActionListener`. Inoltre al QueryActionController devono essere inoltrati gli eventi relativi all'individuazione di nuove soluzioni ricevuti dal QueryControlPanel; pertanto è possibile implementare il QueryActionController in modo tale che anch'esso realizzi l'interfaccia `alice.tuprolog.QueryListener`.

Per quanto riguarda gli eventi relativi alla gestione della risoluzione di una query, sarà il QueryControlPanel a registrare i relativi listener. Gli eventi, però, saranno lanciati dai vari widget contenuti nel QueryControlPanel ed intercettati dal QueryActionController; quest'ultimo modificherà tali eventi e richiamerà i metodi del QueryControlPanel per la notifica degli eventi ai listener registrati; occorre fare questo per soddisfare due necessità:

- la prima necessità consiste nell'aggiungere le informazioni necessarie all'esecuzione dell'azione corrispondente all'interno dell'evento lanciato; infatti per lanciare la StartAction occorre che il relativo evento contenga il contenuto della query, mentre per la StopAction e la AcceptAction occorre che il relativo evento contenga un'indicazione sulla temporaneità degli effetti dell'esecuzione della query;
- la seconda necessità consiste nell'aggiungere un riferimento al QueryActionController all'interno dell'evento lanciato; infatti nel caso in cui durante l'esecuzione di un'azione sia sollevata un'eccezione, l'azione

deve eseguire una callback verso il QueryActionController, in modo tale che l'eccezione sollevata modifichi lo stato dell'interfaccia grafica relativa al QueryControlPanel.

Per aggiungere tale informazioni ad un evento il QueryActionController crea un nuovo evento della classe `QueryActionEvent`, in modo tale che il relativo metodo `String getActionCommand()` restituisca il parametro necessario all'esecuzione della corrispondente azione e che il metodo `getSource()` restituisca il riferimento al QueryActionController. Ma dato che non si vuol evitare la generazione di una dipendenza verso una determinata classe, il metodo `getSource()` non deve restituire un riferimento esplicito alla classe `QueryActionController`, ma ad un'interfaccia che specifica il comportamento di un generico controllore di azioni relative alla risoluzione di una query. Quindi risulta essere conveniente la definizione dell'interfaccia `alice.tuprologx.netbeans.IQueryActionController`, la quale estende l'interfaccia `QueryListener` (si ricordi che il controllore deve ricevere eventi relativi alla risoluzione di query) e definisce il nuovo metodo `public void handleException(PrologException)`, al fine di poter gestire situazioni inaspettate segnalate da eccezioni.

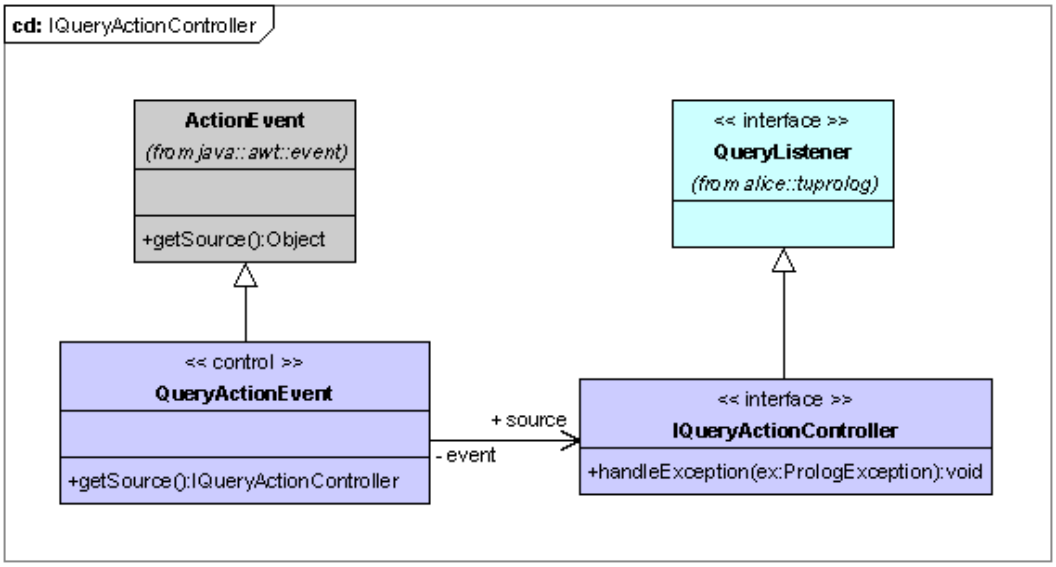


Figura 4.25: Diagramma della classe relativo all'interfaccia IQueryActionController

Riepilogando, affinché il QueryActionController possa essere utilizzato per gestire il sollevamento di eccezioni, un'istanza della classe `QueryActionEvent` deve contenere il riferimento ad esso, sotto forma di istanza relativa all'interfaccia `IQueryActionController`; il design pattern *Façade* non è violato perché

l'inserimento di tale riferimento all'interno degli eventi viene effettuato dal QueryActionController stesso.

Infine, si tenga presente che il QueryActionController deve effettuare la chiamata del metodo `boolean hasOpenAlternatives()` della classe `Prolog` al fine di verificare la presenza di possibili ulteriori soluzioni; ciò comporta che la classe `QueryActionController` sarà dipendente dalla classe `Prolog`.

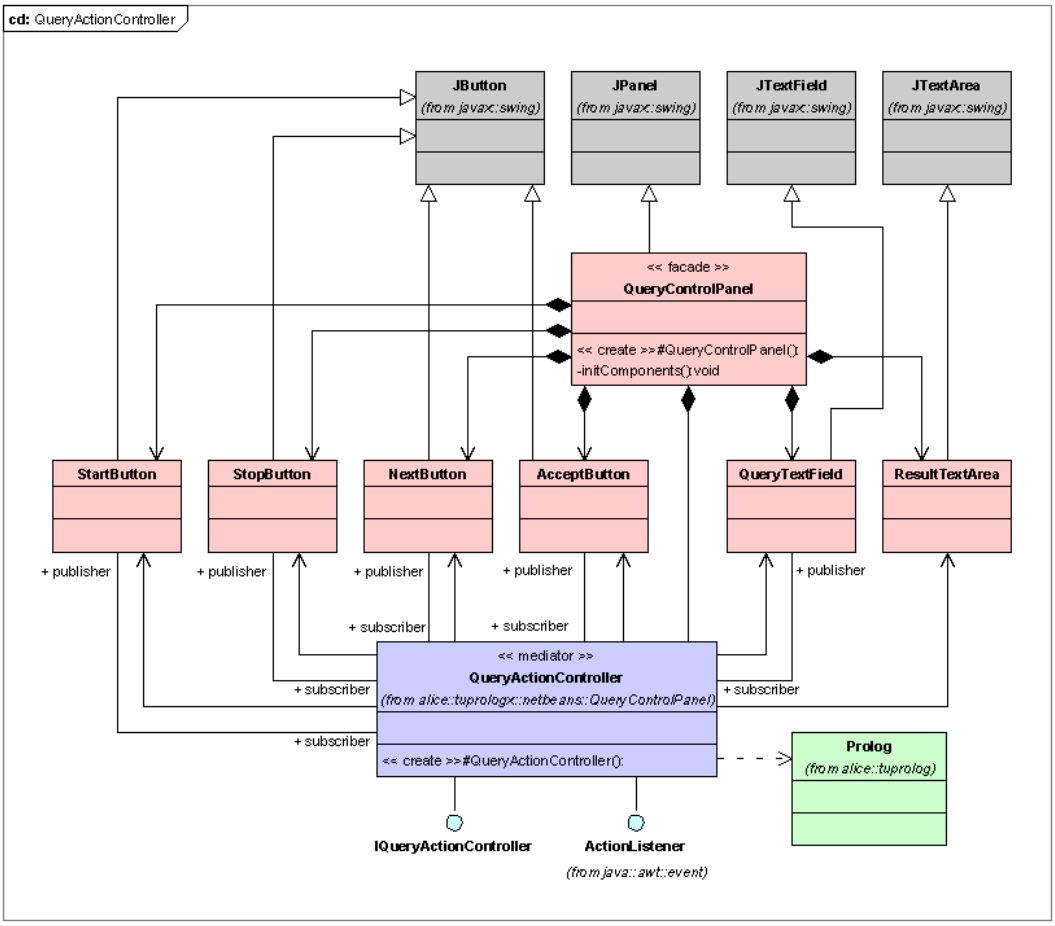


Figura 4.26: Diagramma della classe QueryActionController

4.3.3 Implementazione dell'*OutputView* e dello *SpyView*

Durante la fase di progettazione è stato stabilito che occorre un componente grafico, denominato BlackboardView, tale che implementi l'interfaccia grafica di base dei componenti OutputView e SpyView. Il nome della classe che implementa tale componente grafico è `alice.tuprologx.netbeans.BlackboardView`. Essa deve incapsulare una serie di widget, i cui corrispondenti campi sono:

- `private javax.swing.JTextArea outputTextArea;`

- `private javax.swing.JButton clearButton`.

Affinché tali widget possano essere visualizzati in modo coerente, essi devono essere inseriti all'interno di un widget che rappresenta un pannello; quindi si è stabilito che la BlackboardView sia il suddetto pannello, facendo estendere alla classe `BlackboardView` la classe `javax.swing.JPanel`. Con questa scelta è possibile inserire una qualunque istanza della `BlackboardView` all'interno di un'istanza della classe `org.openide.windows.TopComponent`.

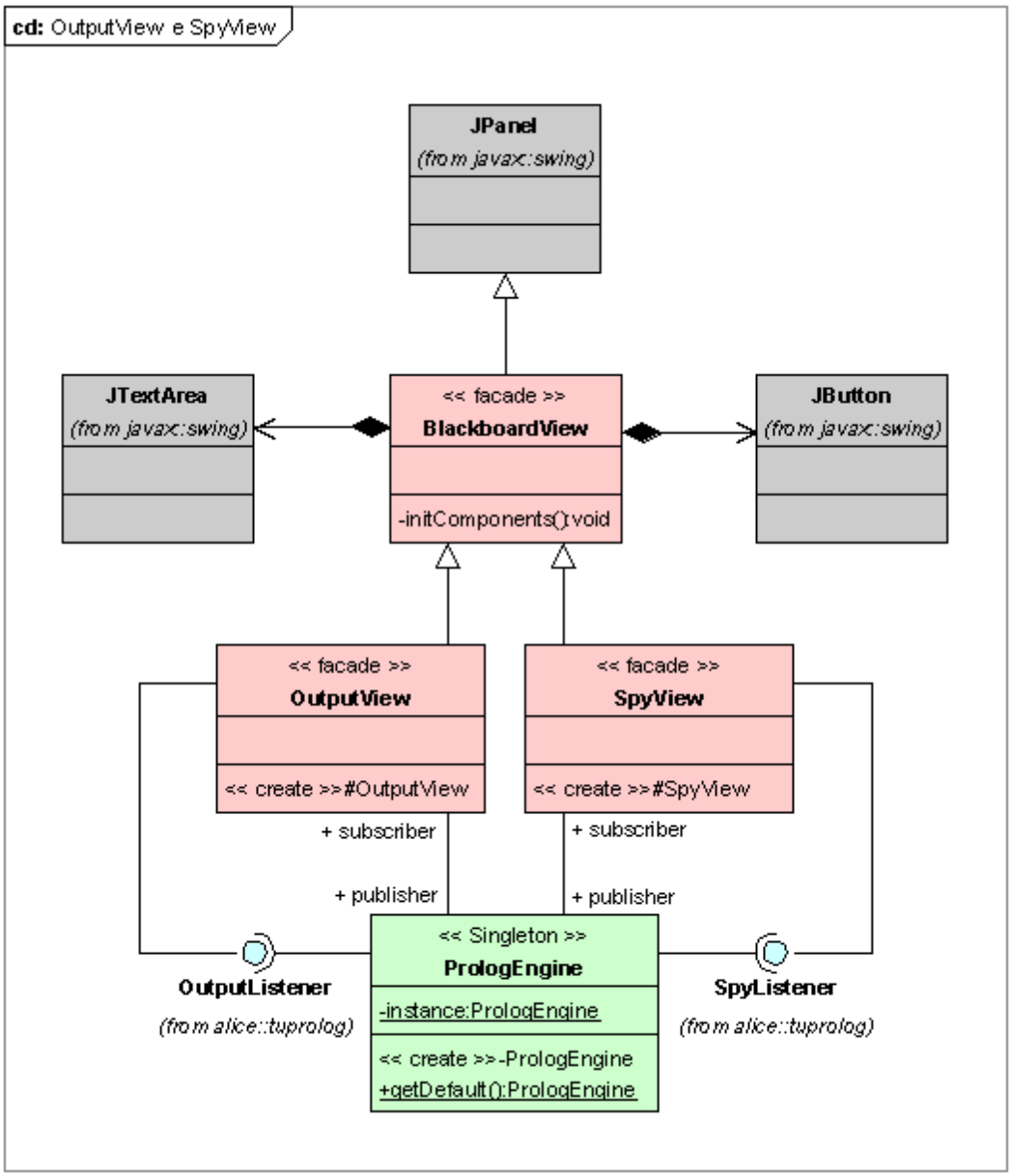


Figura 4.27: Diagramma delle classi OutputView e SpyView

Dato che alla BlackboardView è applicato il design pattern *Façade*, essa non deve presentare metodi per l'accesso dei widget che contiene.

Al fine di ereditare le caratteristiche della BlackboardView, le classi che implementano l'OutputView e la SpyView, rispettivamente la `alice.tuprologx.netbeans.OutputView` e la `alice.tuprologx.netbeans.SpyView`, estendono la classe `BlackboardView`; esse si differenziano in base all'interfaccia realizzata. Infatti, la classe `OutputView` realizza l'interfaccia `OutputListener`, mentre la classe `SpyView` realizza l'interfaccia `SpyListener`.

4.3.4 Implementazione delle azioni di gestione della risoluzione di una query

Queste entità sono azioni da eseguire in risposta ad eventi notificati dal QueryControlPanel; quindi, come già specificato in fase progettuale, gli oggetti che implementano tali azioni devono essere dei listener di eventi registrati presso il QueryControlPanel. Conseguentemente le classi che implementano tali azioni devono realizzare l'interfaccia `alice.tuprologx.netbeans.IQueryActionListener`.

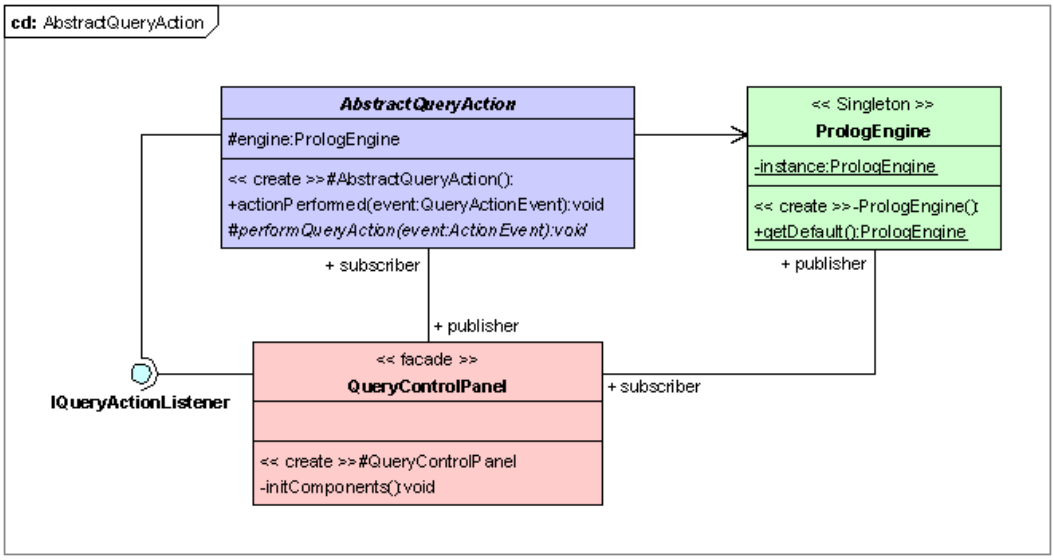


Figura 4.28: Diagramma della classe AbstractQueryAction

Come spiegato in precedenza, le classi che implementano queste azioni devono far parte di una gerarchia di classi, in accordo con il design pattern *Template Method*; la classe radice di tale gerarchia deve essere astratta e il suo nome è `alice.tuprologx.netbeans.AbstractQueryAction`. Tale classe implementa solamente il metodo `public void actionPerformed(QueryActionEvent)`, mentre si limita a definire il metodo astratto `protected void performQueryAction(ActionEvent) throws`

`PrologException`.

Il metodo `void actionPerformed(QueryActionEvent)` è implementato in modo tale che questo richiami il metodo `void performQueryAction(ActionEvent)` e gestisca eventuali eccezioni sollevate eseguendo un'apposita callback alla sorgente dell'evento, il cui riferimento è incapsulato nell'istanza della classe `QueryActionEvent` ricevuta come parametro.

Le sottoclassi concrete della `AbstractQueryAction` sono la `StopAction` e la `AcceptAction`, le quali implementano solamente il metodo `void performQueryAction(ActionEvent)`; queste due azioni verranno richiamate in modo sincrono da appositi widget interni al QueryControlPanel.

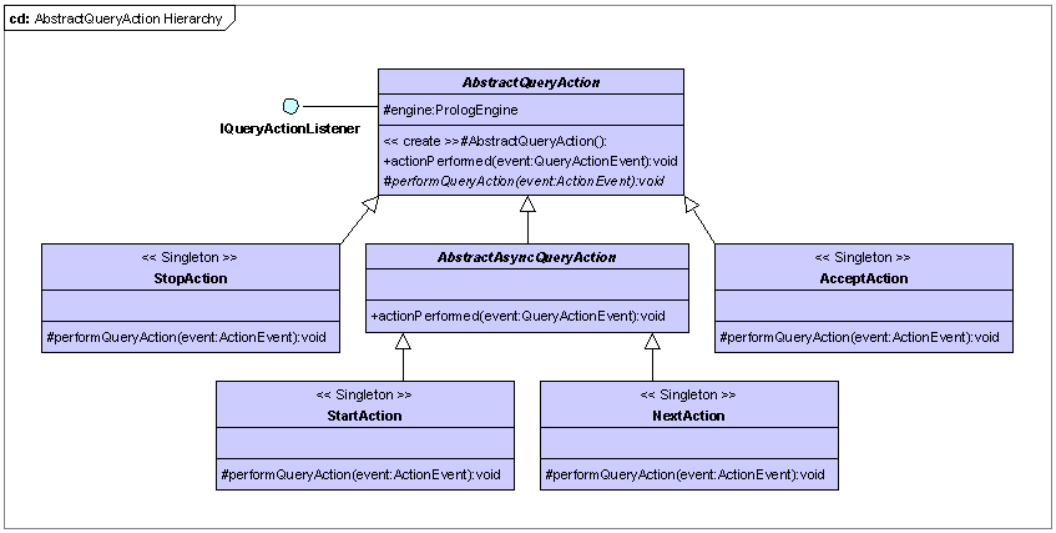


Figura 4.29: Diagramma delle classi relativo alla gerarchia della AbstractQueryAction

Le azioni che devono essere richiamate in modo asincrono sona la StartAction e la NextAction; per riuscire in questo intento si dovrà generare un thread per l'esecuzione vera e propria delle azioni. Questo nuovo thread, una volta messo in esecuzione, si limiterà a richiamare il metodo `void performQueryAction(ActionEvent)`. In tal modo è possibile incapsulare la generazione e l'implementazione del nuovo thread, oltre a poter definire il metodo `void performQueryAction(ActionEvent)` all'interno di sottoclassi, applicando nuovamente il design pattern *Template Method*.

Quindi deve essere implementata una classe astratta, sottoclasse della `AbstractQueryAction`, denominata `alice.tuprologx.netbeans.AbstractAsyncQueryAction`. Questa effettua l'overloading del metodo `void`

`actionPerformed(ActionEvent)`, il quale dovrà generare ed eseguire il thread che permette di eseguire in modo asincrono l'azione. Le relative sottoclassi concrete sono la `StartAction` e la `NextAction`, le quali ridefiniscono solamente il metodo `protected void performQueryAction(ActionEvent)`.

Infine, le classi `StartAction`, `StopAction`, `NextAction` e `AcceptAction` devono realizzare il design pattern *Singleton*, quindi devono essere dotate di un metodo di classe `getDefault()` in grado di restituire l'unica istanza della relativa classe.

4.3.5 Integrazione dei partecipanti nell'ambiente NetBeans e loro inizializzazione

Per integrare nell'ambiente NetBeans i componenti QueryControlPanel e OutputView, questi sono stati inseriti all'interno di un'istanza della classe `org.openide.windows.TopComponent`, la quale implementa una *window component*, ovvero una vista integrabile nell'interfaccia grafica dell'ambiente NetBeans. La classe che implementa tale vista è denominata `alice.tuprologx.netbeans.PrologTopComponent`; affinché la *window component* implementata da tale classe sia correttamente integrata nell'ambiente NetBeans, la classe `PrologTopComponent` deve essere affiancata da:

- una classe che implementa la corrispondente azione di attivazione, denominata `alice.tuprologx.netbeans.PrologAction`; tale classe deve estendere la classe `org.openide.util.actions.CallableSystemAction`; l'azione di attivazione consiste nel recupero di un'istanza della classe `PrologTopComponent` e nella sua visualizzazione;
- un documento XML denominato *PrologTopComponentSetting.xml* relativo alla configurazione della vista; tale documento specifica le classi che implementa la vista e il nome del metodo statico che ne permette il recupero di un'istanza;
- un documento XML denominato *PrologTopComponentWstcref.xml* che permetta all'ambiente di associare la vista alla corrispondente classe e di specificarne lo stato all'avvio.

Si ricordi che ad ogni classe che estende la `TopComponent` deve essere applicato il design pattern *Singleton*, pertanto la classe `PrologTopComponent` deve essere dotata di:

- un campo di classe non modificabile denominato PREFERRED_ID, il quale verrà utilizzato dall'ambiente per associare la vista implementata alla classe `PrologTopComponent`:

  ```
  private static final String PREFERRED_ID =
  "PrologTopComponent";
  ```

- un riferimento all'unica istanza della classe, tramite il campo:

  ```
  private static PrologTopComponent instance;
  ```

- un costruttore a visibilità di tipo *private*:

  ```
  private PrologTopComponent();
  ```

- un metodo di classe a visibilità *public* che permette il recupero dell'unica istanza della classe; se l'unica istanza della classe non è stata costruita, deve provvedere tale metodo:

  ```
  public static synchronized PrologTopComponent
    getDefault();
  ```

- un metodo di classe a visibilità *public* che non si limita a recuperare l'unica istanza della classe, ma verifica se questa è correttamente registrata all'interno dell'ambiente e se il suo identificatore PREFERRED_ID sia effettivamente unico, segnalando eventuali problemi con un messaggio di errore notificato all'ErrorManager; utilizza il metodo descritto al precedente punto per ottenere l'unica istanza della classe `PrologTopComponent`:

  ```
  public static synchronized PrologTopComponent
    findInstance().
  ```

Come già specificato nella fase di progettazione, la classe `PrologTopComponent` incapsula le viste relative al primo prototipo, oltre ai widget necessari alla loro visualizzazione. I componenti e widget da essa incapsulati corrispondono ai campi:

- `private javax.swing.JTabbedPane prologTabbedPane;`
- `private QueryControlPanel queryControlPanel;`

- `private OutputView outputPanel;`
- `private SpyView spyPanel.`

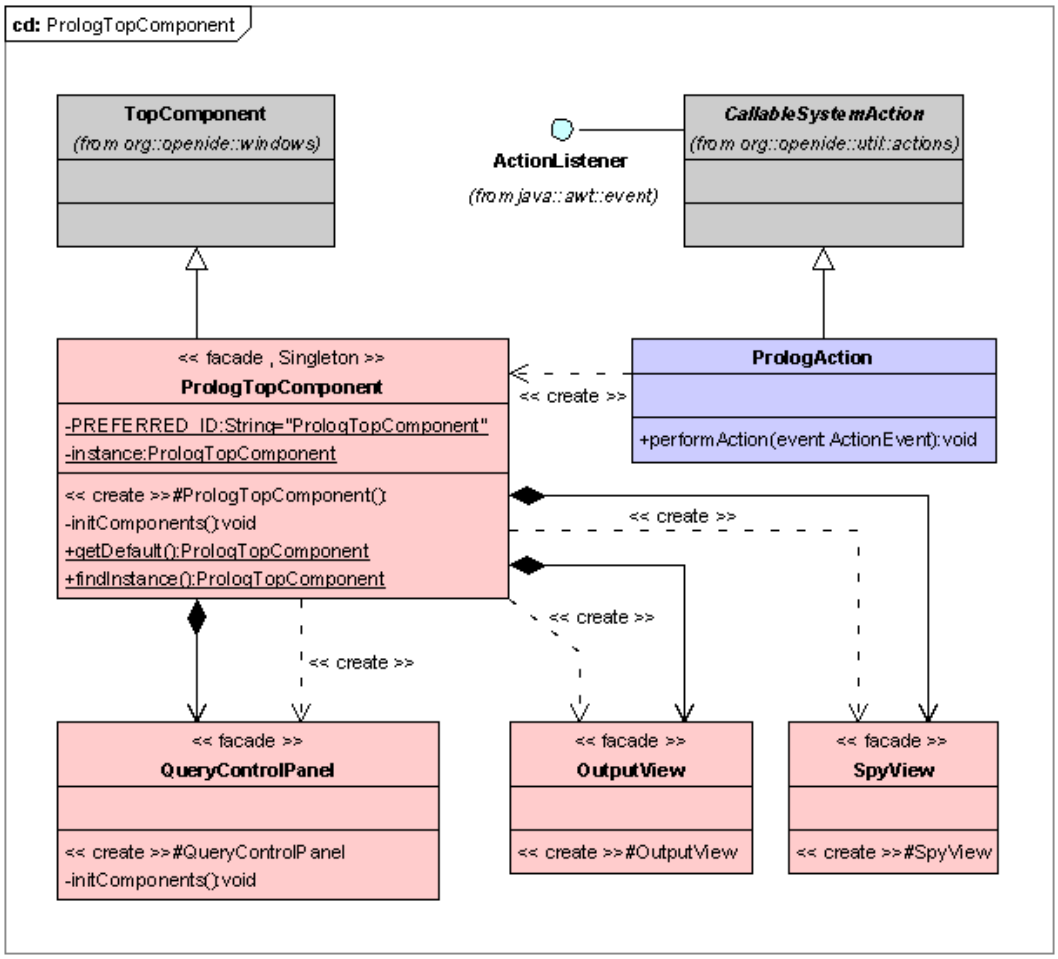


Figura 4.30: Diagramma della classe PrologTopComponent

Non si dimentichi che per inserire una vista che implementa la classe `TopComponent` nell'ambiente NetBeans, occorre dichiarare, all'interno del documento *XML Layer* relativo al modulo di appartenenza, la corrispondente azione di attivazione assieme alla sua locazione all'interno dell'ambiente, oltre ai due documenti XML che specificano le caratteristiche della classe che implementa la vista. Se si vuole che l'azione di attivazione compaia in un menù o una toolbar si dovrà anche indicare nello *XML Layer* la nuova voce di menù o pulsante, associandola all'istanza dell'azione.

4.4 Collaudo del primo prototipo

Dato che il PrologEngine è il componente fondamentale per la correttezza del sistema, saranno verificate le sue funzionalità con dei test unitari; a tale scopo devono essere aggiunti alla classe `PrologEngine` alcuni metodi, necessari all'esecuzione dei test ma non necessari per lo sviluppo del primo prototipo:

- `public boolean hasOpenAlternatives();`
- `public void addTheory(Theory)`
  `throws InvalidTheoryException;`
- `public Theory getWholeTheory().`

Le verifiche effettuate tramite il test unitario della classe `PrologEngine` riguardano:

- il funzionamento del metodo `SolveInfo solve(String);`
- il funzionamento del metodo `void solveHalt();`
- il funzionamento del metodo `SolveInfo solveNext();`
- il funzionamento del metodo `void solveEnd();`
- il funzionamento del metodo `void removeAllTheories();`
- la gestione dei *query listener*;
- la gestione degli *output listener*;
- la gestione degli *spy listener*.

I risultati dei test sono stati tutti positivi, tranne per la gestione degli *spy listener*: il metodo `Collection<SpyListener> getSpyListeners()`, relativo alla classe `PrologEngine`, presenta un comportamento anomalo. Indagando su questo malfunzionamento è stato scoperto un bug nella classe `Prolog` (quindi nel motore tuProlog incapsulato dal PrologEngine, pertanto un elemento esterno a questo plug-in): il metodo `List getSpyListeners()` restituisce la lista degli *output listener* invece di restituire la lista degli *spy listener*.

Dato che questo non è un difetto tale da compromettere il corretto funzionamento del primo prototipo (infatti il metodo che presenta il comportamento anomalo è stato introdotto per completezza e in pratica non viene utilizzato), si è deciso che, per il momento, è meglio non apportare alcuna correzione.

Capitolo 5

Secondo prototipo

In questo capitolo, partendo dal lavoro effettuato per la realizzazione del primo prototipo, verrà sviluppato il progetto del secondo prototipo del plug-in di tuProlog per l'ambiente NetBeans, continuando ad avere come riferimento le considerazioni effettuate durante l'analisi del problema e su quanto specificato nel piano di lavoro; dopodiché si procederà ad analizzare l'implementazione del secondo prototipo.

5.1 Progettazione del secondo prototipo

La progettazione del secondo prototipo si baserà su ciò che suggerisce di realizzare il piano di lavoro:

Secondo prototipo*: integrazione della gestione semplificata delle teorie attive nel plug-in; verrà aggiunto al plug-in un componente grafico che rappresenti l'elenco delle teorie attive e ne permetta la manipolazione; le manipolazioni consentite saranno solamente quelle relative alle funzionalità supportate; le teorie attive potranno essere importate/esportate tramite il filesystem locale.*

Osservando la figura 3.1, relativa al diagramma dei casi d'uso dal titolo "Gestione della risoluzione di query e gestione delle teorie attive", si deduce che i casi d'uso da realizzare per fare ciò che suggerisce il piano di lavoro sono:

- *modifica elenco delle teorie attive*;
- *aggiorna motore tuProlog*;
- *esegui query*;

- *importazione teoria dall'elenco delle teorie disponibili*;
- *esportazione teoria verso l'elenco delle teorie disponibili*;
- *estrai una o più teorie attive*;
- *estrai la teoria attiva globale*.

Dato che l'elenco delle teorie disponibili non sarà ancora presente in questo prototipo, il caso d'uso "*importazione teoria dall'elenco delle teorie disponibili*" sarà realizzato permettendo esclusivamente l'importazione di una teoria dal filesystem locale; pertanto esso sarà rinominato come "*importazione teoria dal filesystem locale*". Analogamente, il caso d'uso "*esportazione teoria verso l'elenco delle teorie disponibili*" sarà realizzato permettendo solamente l'esportazione di una teoria verso il filesystem locale, richiedendo di specificare locazione e nome del file; pertanto esso sarà rinominato come "es*portazione teoria verso il filesystem locale*". Infine, il caso d'uso "*estrai una o più teorie attive*" sarà realizzato permettendo solamente l'estrazione di una sola teoria per volta; pertanto esso sarò rinominato "*estrai una teoria attiva*".

5.1.1 Collaborazione: *modifica elenco delle teorie attive*

Con riferimento all'analisi del problema, per poter realizzare il caso d'uso "*modifica elenco delle teorie attive*" occorrono:

- un modello relativo all'elenco delle teorie attive;
- un componente grafico tale da offrire le funzionalità di gestione delle teorie attive.

Inoltre, serviranno anche degli elementi di controllo in grado eseguire un'azione di modifica dell'elenco delle teorie attive. Essi si limiteranno alle azioni relative alle funzionalità supportate di gestione delle teorie attive, escludendo quella relativa all'aggiunta di una teoria in coda all'elenco, perché farà parte di una successiva collaborazione. Tali azioni sono:

- impostazione di un'unica teoria nell'elenco delle teorie attive;
- rimozione dell'ultima teoria dotata di un determinato nome dall'elenco delle teorie attive;
- rimozione di tutte le teorie dall'elenco delle teorie attive.

I partecipanti sono:

- *ActiveTheoryList*: elenco delle teorie attive;
- *ActiveTheoryListView*: vista che consente di visualizzare e gestire l'elenco delle teorie attive;
- *SetTheoryAction*: azione che impone un'unica teoria all'elenco delle teorie attive;
- *RemoveTheoryAction*: azione che rimuove l'ultima teoria dotata di un determinato nome dall'elenco delle teorie attive;
- *RemoveAllTheoriesAction*: azione che rimuove tutte le teorie dall'elenco delle teorie attive.

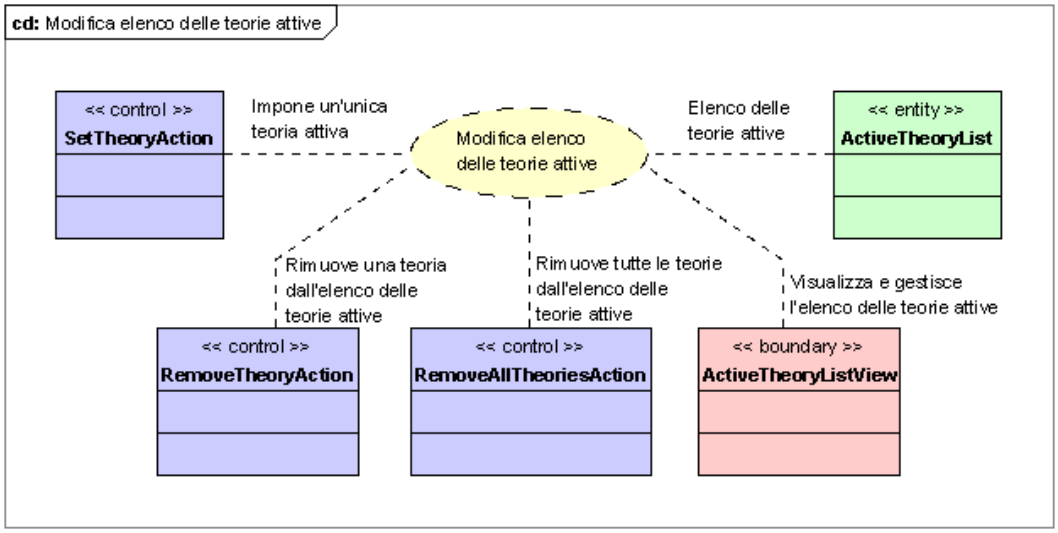


Figura 5.1: Collaborazione relativa al caso d'uso "modifica elenco delle teorie attive"

Dai ruoli definiti per ogni partecipante, si determinano le relative responsabilità:

- *ActiveTheoryList*:
  - offre i servizi di accesso e modifica dell'elenco delle teorie attive;
  - notifica il cambiamento del suo contenuto;
- *ActiveTheoryListView*:
  - richiama le azioni SetTheoryAction, RemoveTheoryAction e RemoveAllTheoriesAction, allo scopo di eseguire le corrispondenti operazioni, su richiesta dell'utente;
  - riceve le notifiche relative al cambiamento del contenuto dell'elenco

delle teorie attive e ne visualizza l'effetto;

- *SetTheoryAction*: impone un'unica teoria all'ActiveTheoryList;
- *RemoveTheoryAction*: rimuove l'ultima teoria dotata di un determinato nome dall'ActiveTheoryList;
- *RemoveAllTheoriesAction*: rimuove tutte le teorie dall'ActiveTheoryList.

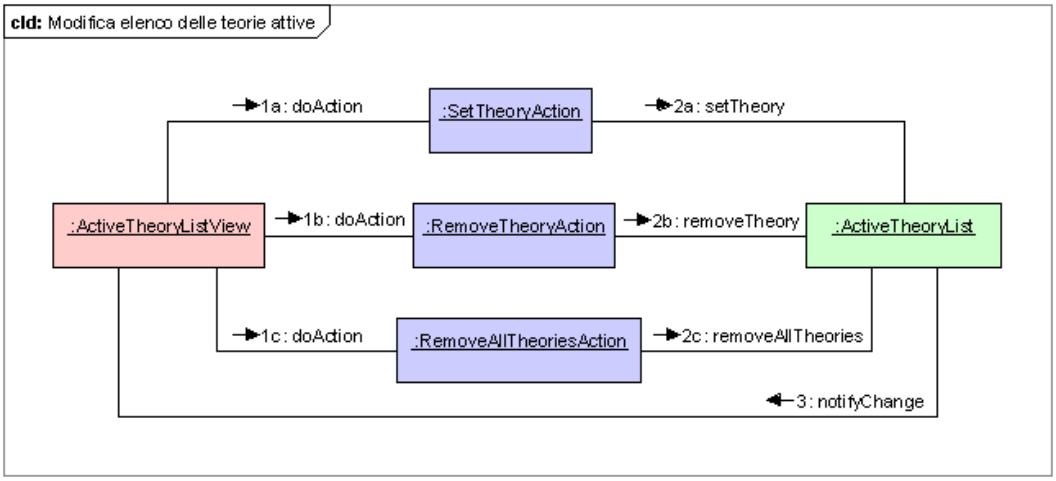


Figura 5.2: Diagramma della collaborazione "modifica elenco delle teorie attive"

5.1.2 Collaborazione: *aggiorna motore tuProlog*

Con riferimento all'analisi del problema, per poter realizzare il caso d'uso "*aggiorna motore tuProlog*" occorrono:

- un modello relativo all'elenco delle teorie attive;
- il motore tuProlog.

Inoltre, servirà anche un elemento di controllo in grado di gestire l'esecuzione dell'aggiornamento del motore tuProlog.

I partecipanti sono:

- *ActiveTheoryList*: elenco delle teorie attive;
- *PrologEngine*: il motore tuProlog;
- *TheoryManager*: gestore delle teorie del motore tuProlog.

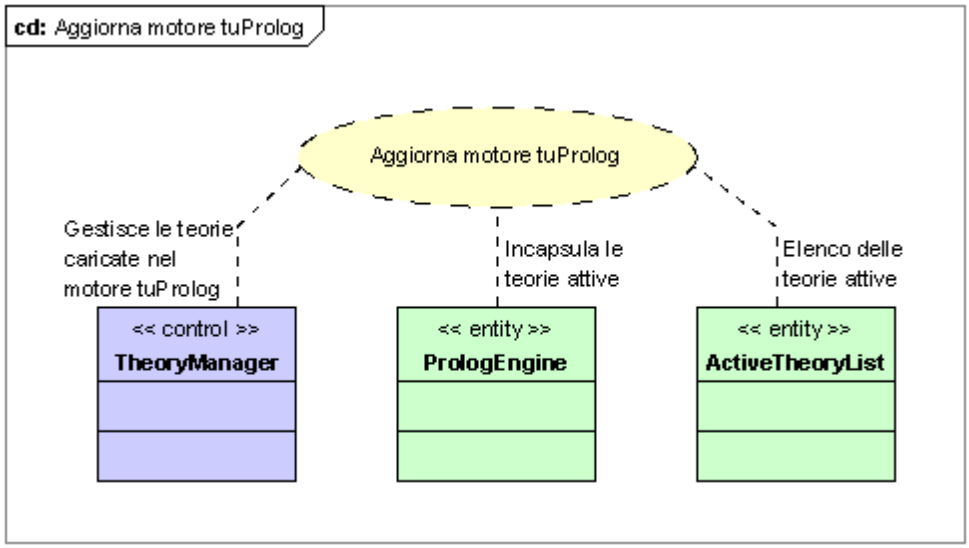


Figura 5.3: Collaborazione relativa al caso d'uso "aggiorna motore tuProlog"

Dai ruoli definiti per ogni partecipante, si determinano le relative responsabilità:

- *ActiveTheoryList*: notifica il cambiamento del suo contenuto;
- *PrologEngine*: memorizza le teorie da utilizzare nella valutazione delle query;
- *TheoryManager*:
  - offre i servizi di accesso e modifica delle teorie attive;
  - aggiorna il motore tuProlog in risposta ad un cambiamento del contenuto dell'elenco delle teorie attive.

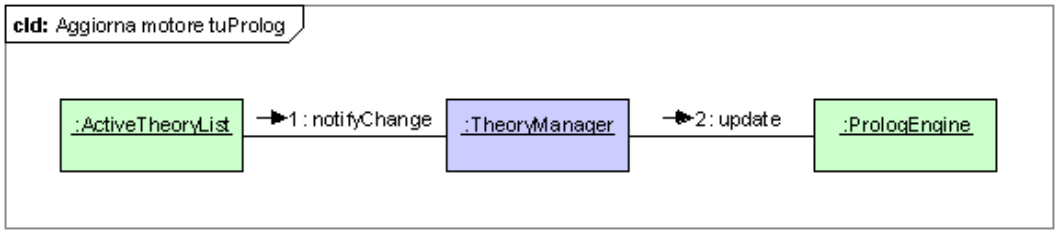


Figura 5.4: Diagramma della collaborazione "aggiorna motore tuProlog"

5.1.3 Collaborazione: *esegui query*

Con riferimento all'analisi del problema, rispetto alla collaborazione "*esegui query temporanea*", per poter realizzare il caso d'uso "*esegui query*" occorrerà aggiungere un modello relativo all'elenco delle teorie attive, mentre gli elementi di controllo in grado di realizzare le interazioni tra i partecipanti della collaborazione saranno gli stessi.

Quindi, l'unico partecipante in più è l'*ActiveTheoryList*, il quale svolge il ruolo di elenco delle teorie attive.

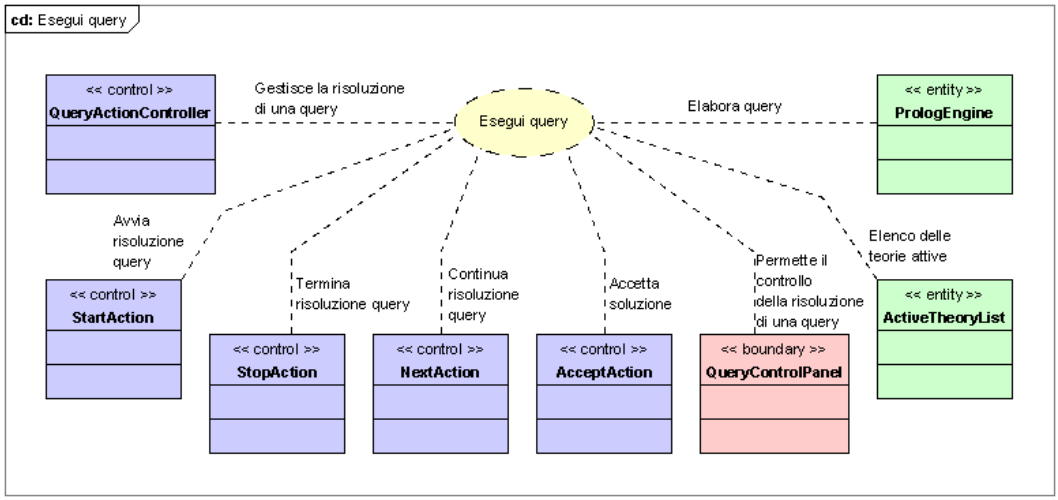


Figura 5.5: Collaborazione relativa al caso d'uso "esegui query"

Rispetto alla collaborazione "*esegui query temporanea*", specificata durante la progettazione del primo prototipo, i partecipanti presentano queste differenze per quanto riguarda le responsabilità:

- *PrologEngine*:
  - modifica l'elenco delle teorie attive;
  - deve notificare all'ActiveTheoryList le modifiche all'elenco delle teorie attive;
- *ActiveTheoryList*: riceve le notifiche relative alle modifiche dell'elenco delle teorie attive;
- *StopAction*: richiede al PrologEngine solamente di terminare la risoluzione di una query, senza rimuovere tutte le teorie caricate nel PrologEngine;
- *AcceptAction*: indica al PrologEngine solamente che la risoluzione di una query è terminata, senza rimuovere tutte le teorie caricate nel PrologEngine.

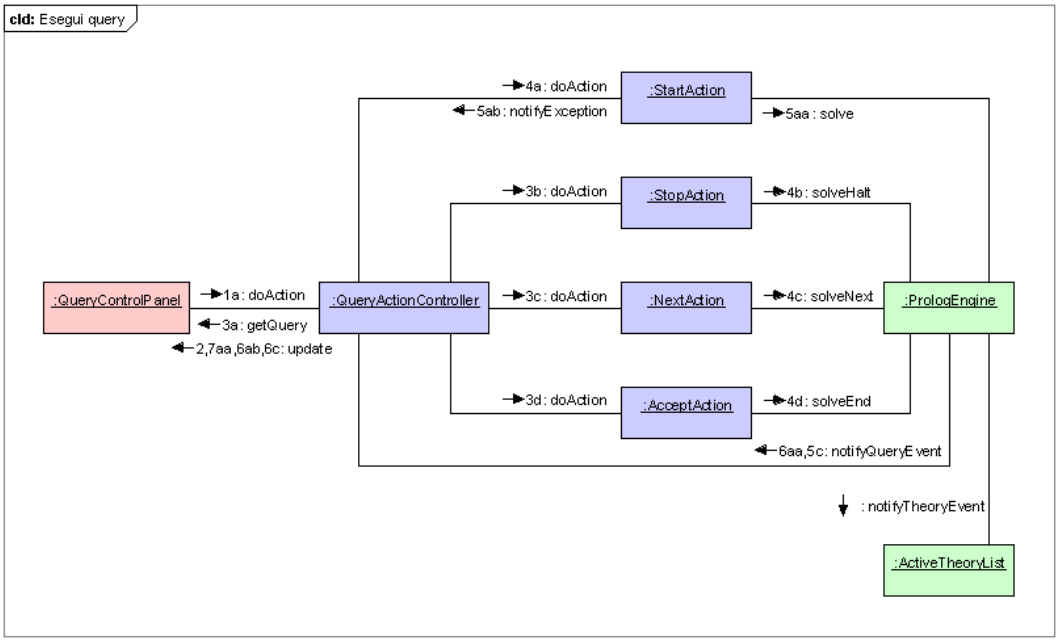


Figura 5.6: Diagramma della collaborazione "esegui query"

5.1.4 Collaborazione: *importazione teoria dal filesystem locale*

Con riferimento all'analisi del problema, per poter realizzare il caso d'uso "*importazione teoria dal filesystem locale*" occorrono:

- un modello relativo all'elenco delle teorie attive;
- un componente grafico tale da offrire le funzionalità di gestione delle teorie attive;
- un oggetto in grado di accedere al filesystem locale;
- un'interfaccia grafica che permetta all'utente di specificare il file contenente la teoria da caricare.

Inoltre, serviranno anche due elementi di controllo: uno in grado di eseguire l'azione di importazione della teoria, l'altro in grado di recuperare la teoria da importare dal filesystem locale.

I partecipanti sono:

- *ActiveTheoryList*: elenco delle teorie attive;
- *ActiveTheoryListView*: vista che consente di gestire l'elenco delle teorie attive;
- *LocalFileSystem*: gestore del filesystem locale;

- *FileChooser*: interfaccia grafica per la scelta del file contenente la teoria da caricare;
- *AddTheoryAction*: azione che aggiunge una teoria in coda all'elenco delle teorie attive;
- *LocalTheoryLoader*: oggetto in grado di gestire il caricamento di una teoria dal filesystem locale.

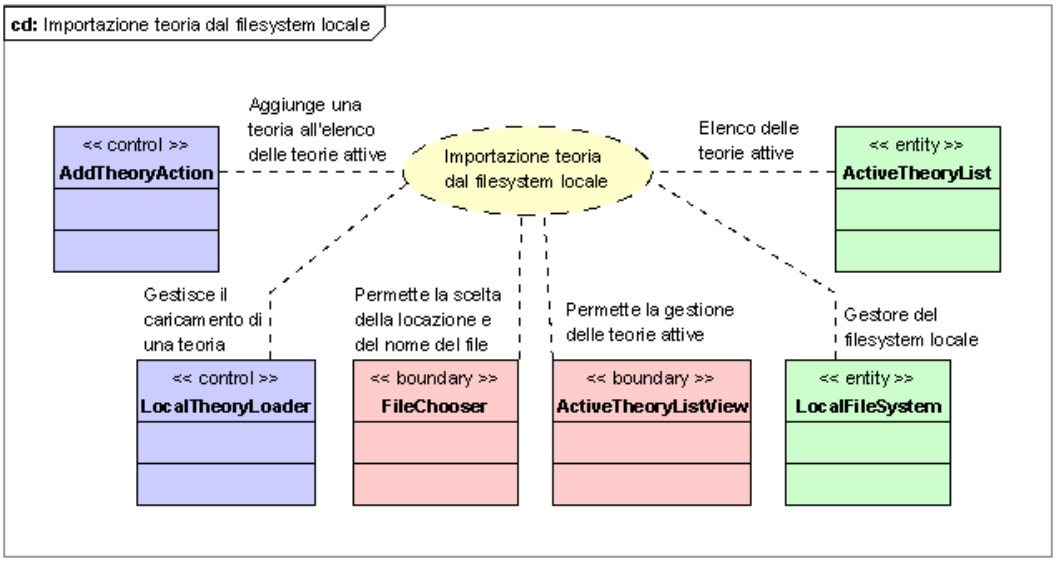


Figura 5.7: Collaborazione relativa al caso d'uso "importazione teoria dal filesystem locale"

Dai ruoli definiti per ogni partecipante, si determinano le relative responsabilità:

- *ActiveTheoryList*: offre il servizio di aggiunta della teoria caricata all'elenco delle teorie attive;
- *ActiveTheoryListView*: richiama l'azione AddTheoryAction allo scopo di aggiungere una teoria all'elenco delle teorie attive, su richiesta dell'utente;
- *LocalFileSystem*: permette di leggere una teoria dal filesystem locale;
- *FileChooser*:
  - richiede all'utente di specificare il file contenente la teoria da caricare;
  - restituisce locazione e nome del file contenente la teoria da caricare;
- *AddTheoryAction*:
  - richiede una teoria al LocalTheoryLoader;
  - aggiunge la teoria caricata in coda all'ActiveTheoryList;

  - deve visualizzare una notifica di errore se ricevere un'eccezione dal LocalTheoryLoader;
- *LocalTheoryLoader*:
  - acquisisce nome e locazione del file contenente la teoria da caricare, utilizzando il FileChooser;
  - carica una teoria tramite il LocalFileSystem;
  - può fallire nel suo compito, notificando un'eccezione.

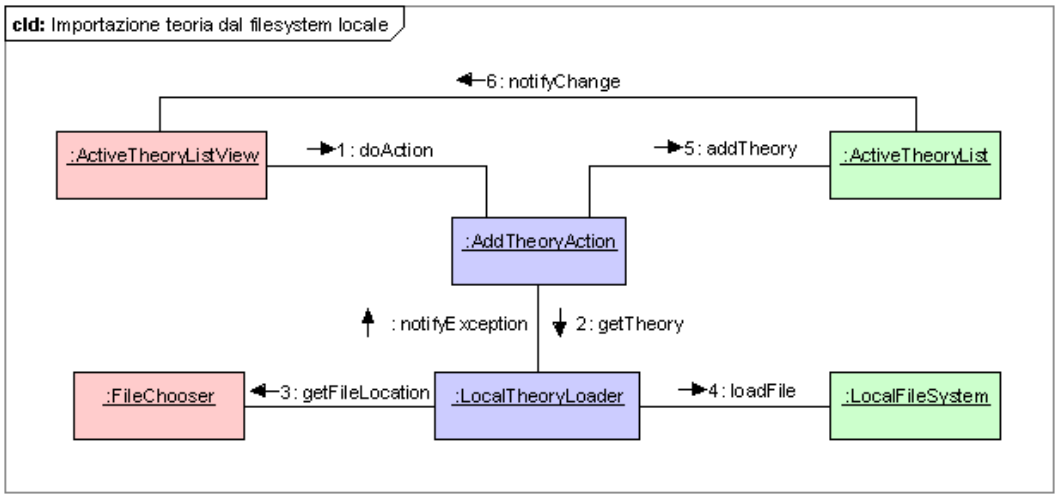


Figura 5.8: Diagramma della collaborazione "importazione teoria dal filesystem locale"

5.1.5 Collaborazione: *esportazione teoria verso il filesystem locale*

Con riferimento all'analisi del problema, per poter realizzare il caso d'uso "*esportazione teoria verso il filesystem locale*" occorrono:

- un oggetto in grado di accedere al filesystem locale;
- un'interfaccia grafica che permetta all'utente di specificare la locazione e il nome del file in cui salvare la teoria.

Inoltre, serviranno anche due elementi di controllo: uno in grado di eseguire l'azione di esportazione della teoria, l'altro in grado di salvare la teoria da esportare nel filesystem locale.

I partecipanti sono:

- *LocalFileSystem*: gestore del filesystem locale;
- *FileChooser*: interfaccia grafica per la scelta del file che dovrà contenere la teoria da salvare;

- *ExportTheoryAction*: azione che esporta una determinata teoria;
- *LocalTheorySaver*: oggetto in grado di gestire il salvataggio di una teoria nel filesystem locale.

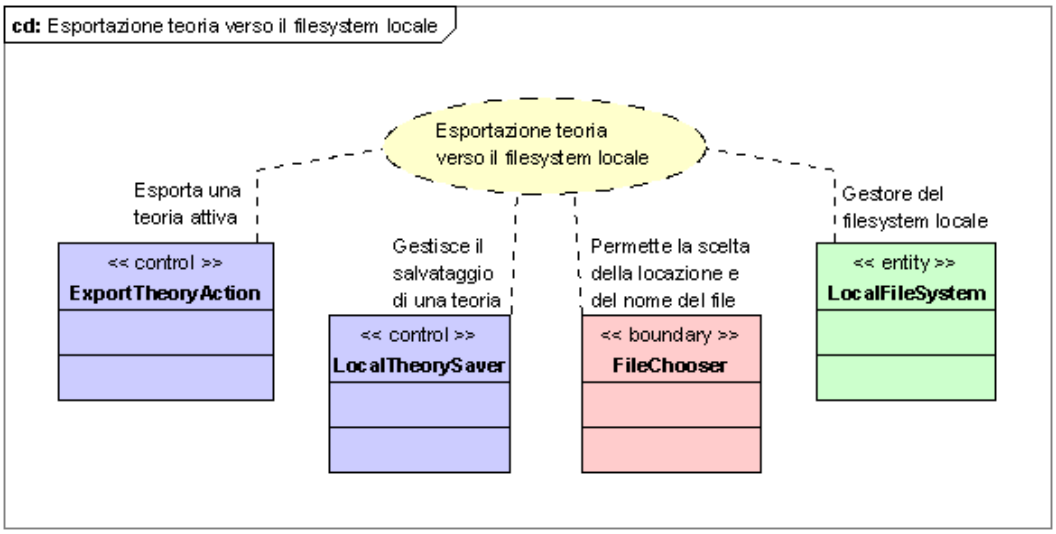


Figura 5.9: Collaborazione relativa al caso d'uso "esportazione teoria verso il filesystem locale"

Dai ruoli definiti per ogni partecipante, si determinano le relative responsabilità:

- *LocalFileSystem*: permette di scrivere una teoria su filesystem locale;
- *FileChooser*:
  - richiede all'utente di specificare il file che dovrà contenere la teoria da salvare;
  - restituisce nome e locazione del file che dovrà contenere la teoria da salvare;
- *ExportTheoryAction*:
  - richiede di salvare una teoria al LocalTheorySaver;
  - deve visualizzare una notifica di errore se riceve un'eccezione dal LocalTheorySaver;
- *LocalTheorySaver*:
  - acquisisce nome e locazione del file che conterrà la teoria da salvare, utilizzando il FileChooser;
  - salva una teoria tramite il LocalFileSystem;
  - può fallire nel suo compito, notificando un'eccezione.

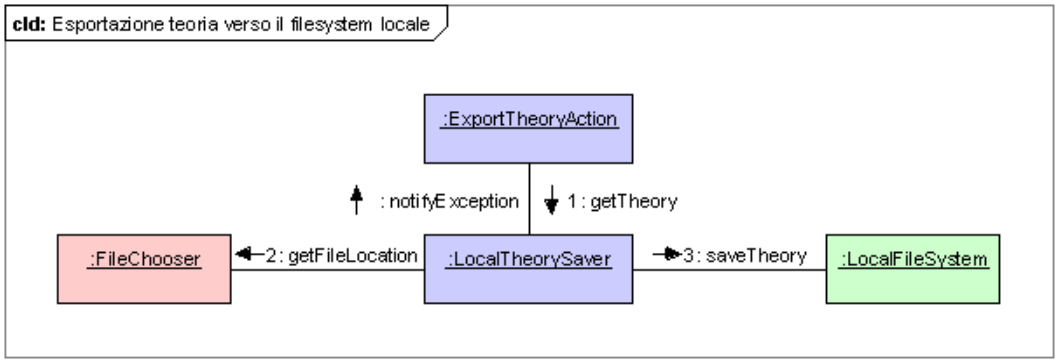


Figura 5.10: Diagramma della collaborazione "esportazione teoria verso il filesystem locale"

5.1.6 Collaborazione: *estrai una teoria attiva*

Con riferimento all'analisi del problema, per poter realizzare il caso d'uso "*estrai una teoria attiva*" occorrono:

- un modello relativo all'elenco delle teorie attive;
- un componente grafico tale da offrire le funzionalità di gestione delle teorie attive.

Inoltre, serviranno anche due elementi di controllo: uno in grado di estrarre una teoria, l'altro in grado di esportare la teoria.

I partecipanti sono:

- *ActiveTheoryList*: elenco delle teorie attive;
- *ActiveTheoryListView*: vista che consente di gestire l'elenco delle teorie attive;

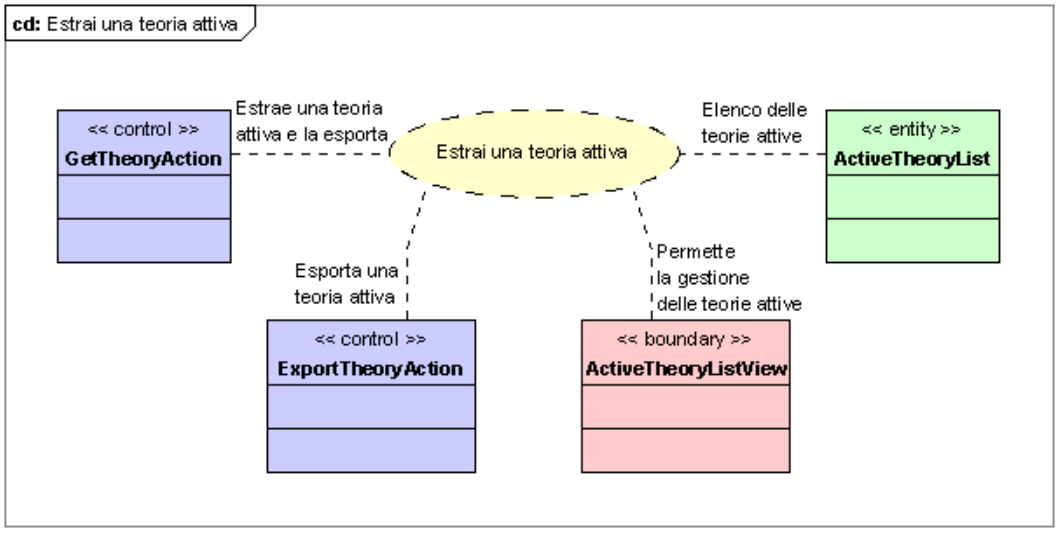


Figura 5.11: Collaborazione relativa al caso d'uso "estrai una teoria attiva"

- *GetTheoryAction*: azione che estrae una determinata teoria e la esporta;
- *ExportTheoryAction*: azione che esporta una determinata teoria.

Dai ruoli definiti per ogni partecipante, si determinano le relative responsabilità:

- *ActiveTheoryList*: fornisce la teoria attiva da salvare;
- *ActiveTheoryListView*: richiama l'azione GetTheoryAction allo scopo di salvare una teoria attiva, su richiesta dell'utente, indicando quale teoria deve essere salvata;
- *GetTheoryAction*:
  - estrae la teoria indicata dall'ActiveTheoryList;
  - esegue la procedura che esporta la teoria indicata;
- *ExportTheoryAction*:
  - esporta la teoria indicata;
  - deve visualizzare una notifica di errore in caso di fallimento.

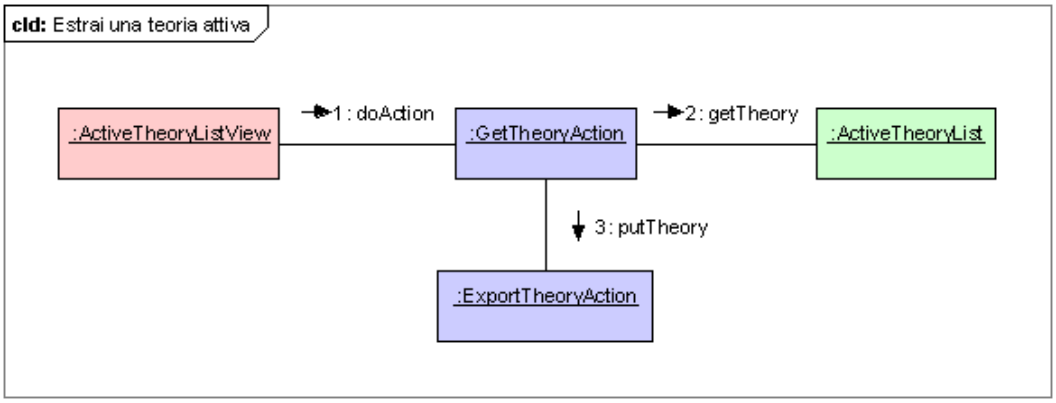


Figura 5.12: Diagramma della collaborazione "estrai una teoria attiva"

5.1.7 Collaborazione: *estrai la teoria attiva globale*

Con riferimento all'analisi del problema, per poter realizzare il caso d'uso *estrai la teoria attiva globale* occorrono:

- un modello relativo all'elenco delle teorie attive;
- un componente grafico tale da offrire le funzionalità di gestione delle teorie attive.

Inoltre, serviranno anche due elementi di controllo: uno in grado di estrarre la teoria attiva globale, l'altro in grado di esportare tale teoria.

I partecipanti sono:

- *ActiveTheoryList*: elenco delle teorie attive;
- *ActiveTheoryListView*: vista che consente di gestire l'elenco delle teorie attive;
- *GetWholeTheoryAction*: azione che estrae la teoria attiva globale e la esporta;
- *ExportTheoryAction*: azione che esporta una determinata teoria.

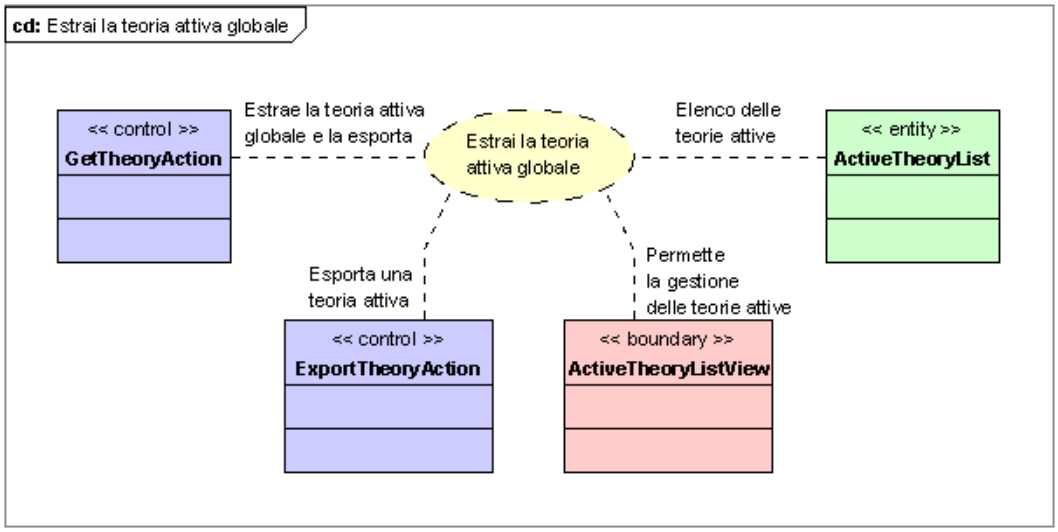


Figura 5.13: Collaborazione relativa al caso d'uso "estrai la teoria globale"

Dai ruoli definiti per ogni partecipante, si determinano le relative responsabilità:

- *ActiveTheoryList*: fornisce la teoria attiva globale da salvare;
- *ActiveTheoryListView*: richiama l'azione GetTheoryAction allo scopo di salvare la teoria attiva globale, su richiesta dell'utente, indicando che la teoria che deve essere salvata è la teoria attiva globale;
- *GetWholeTheoryAction*:
  - estrae la teoria attiva globale dall'ActiveTheoryList;
  - esegue la procedura che esporta la teoria attiva globale;
- *ExportTheoryAction*:
  - esporta la teoria indicata;
  - deve visualizzare una notifica di errore in caso di fallimento.

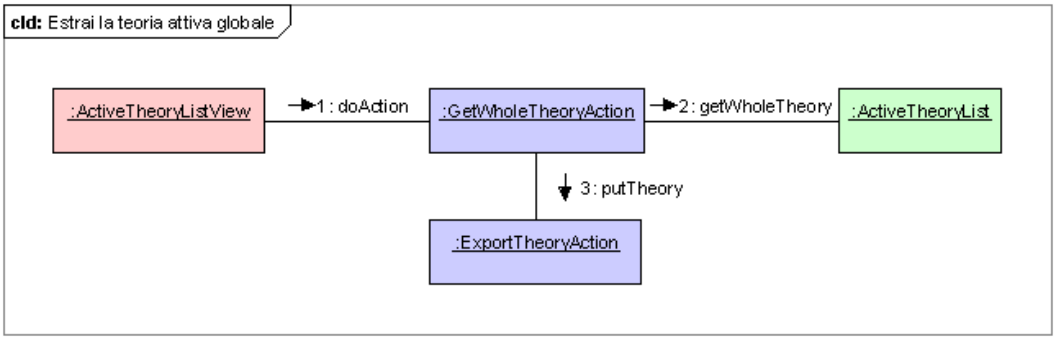


Figura 5.14: Diagramma della collaborazione "estrai la teoria attiva globale"

5.1.8 Interazione: *impostazione di un'unica teoria attiva*

Tale interazione viene avviata con un comando da parte dell'utente, impartito tramite l'interfaccia grafica implementata dall'ActiveTheoryListView, il quale delegherà alla SetTheoryAction la gestione dell'esecuzione dell'azione. Eseguita l'azione, l'eventuale cambiamento del contenuto dell'elenco delle teorie attive deve essere segnalato all'ActiveTheoryListView, affinché si aggiorni.

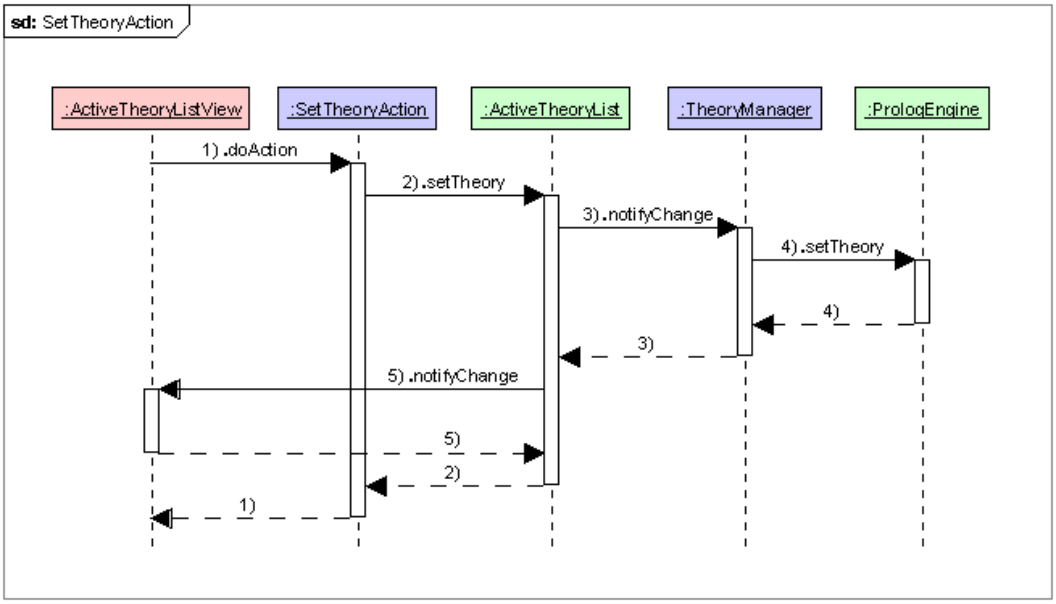


Figura 5.15: Diagramma di sequenza relativo all'interazione "impostazione di un'unica teoria attiva"

5.1.9 Interazione: *rimozione di una teoria attiva*

Tale interazione viene avviata con un comando da parte dell'utente, impartito tramite l'interfaccia grafica implementata dall'ActiveTheoryListView, il quale delegherà alla RemoveTheoryAction la gestione dell'esecuzione dell'azione. Eseguita l'azione, il cambiamento del contenuto dell'elenco delle teorie attive deve essere segnalato all'ActiveTheoryListView, affinché si aggiorni.

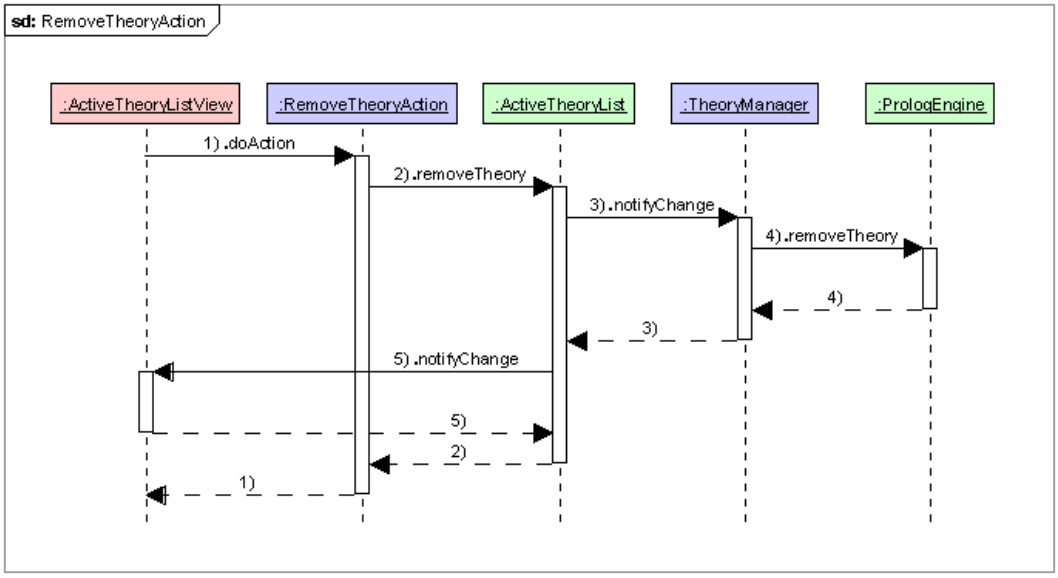


Figura 5.16: Diagramma di sequenza relativo all'interazione "rimozione di una teoria attiva"

5.1.10 Interazione: *rimozione di tutte le teorie attive*

Tale interazione viene avviata con un comando da parte dell'utente, impartito tramite l'interfaccia grafica implementata dall'ActiveTheoryListView, il quale delegherà alla RemoveAllTheoriesAction la gestione dell'esecuzione dell'azione. Eseguita l'azione, il cambiamento del contenuto dell'elenco delle teorie attive deve essere segnalato all'ActiveTheoryListView, affinché si aggiorni.

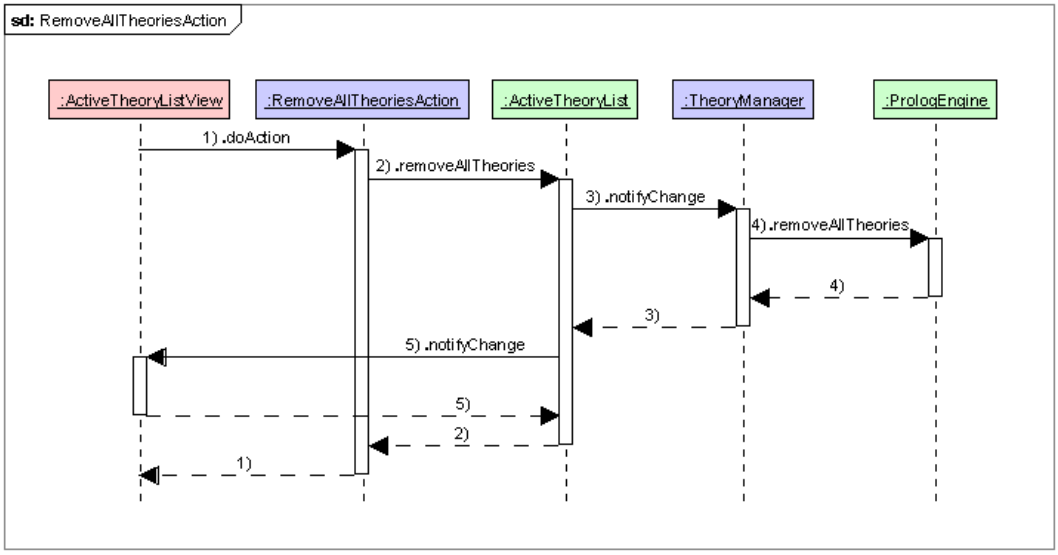


Figura 5.17: Diagramma di sequenza relativo all'interazione "rimozione di tutte le teorie attive"

5.1.11 Interazione: *avviamento della risoluzione di una query*

Tale interazione è identica all'omonima definita nella fase di progettazione del primo prototipo, tranne per il fatto che il PrologEngine può notificare

all'ActiveTheoryList i cambiamenti effettuati nell'elenco delle teorie attive durante la risoluzione di una query. Per poter avere le informazioni necessarie relativamente alle eventuali modifiche dell'elenco delle teorie attive, l'ActiveTheoryList è costretto ad eseguire una callback sul PrologEngine.

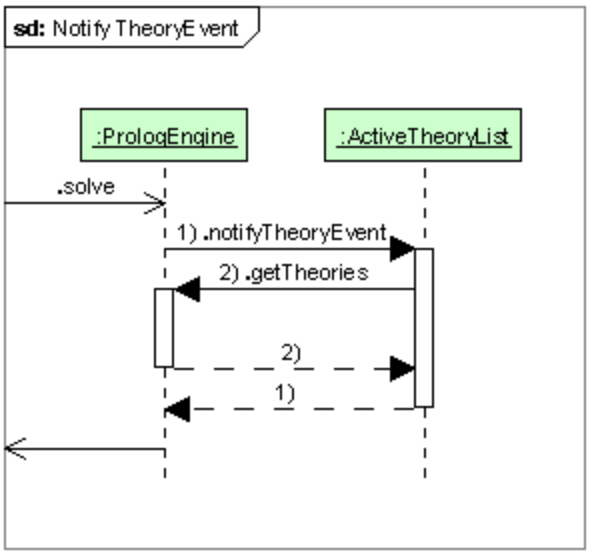


Figura 5.18: Sequenza da aggiungere all'interazione "avviamento della risoluzione di una query"

5.1.12 Interazione: *terminazione della risoluzione di una query*

Tale interazione è identica all'omonima definita nella fase di progettazione del primo prototipo, tranne per il fatto che l'azione StopAction non rimuove tutte le teorie attive dal PrologEngine.

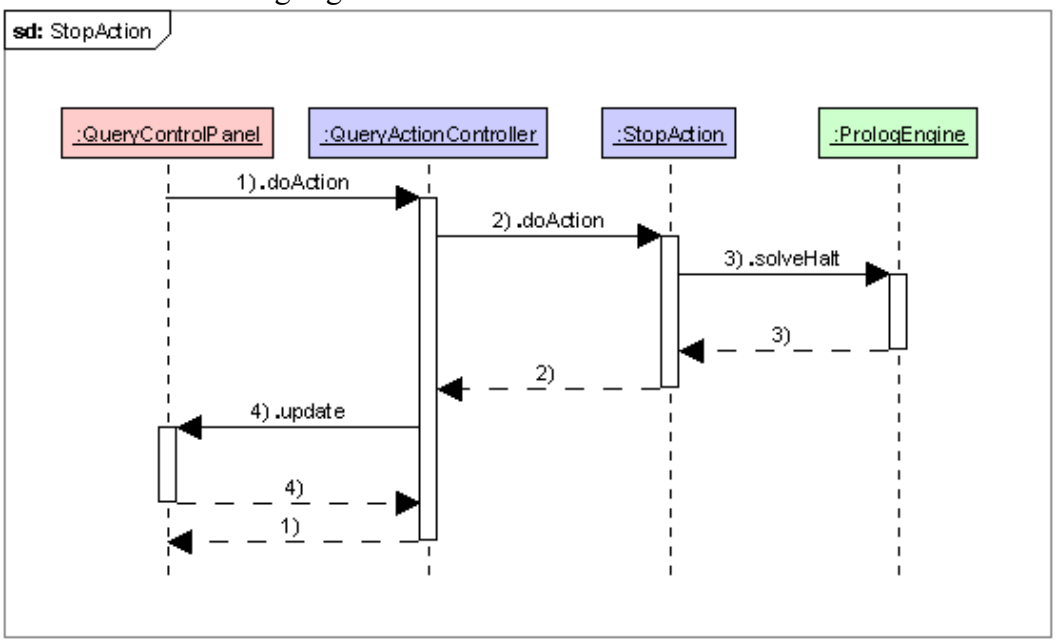


Figura 5.19: Diagramma di sequenza relativo alla nuova interazione "terminazione della risoluzione di una query"

5.1.13 Interazione: *ricerca di un'ulteriore soluzione*

Tale interazione è identica all'omonima definita nella fase di progettazione del primo prototipo, tranne per il fatto che il PrologEngine può notificare all'ActiveTheoryList le variazioni delle teorie attive durante la risoluzione di una query (vedi il diagramma di sequenza in figura 5.18).

5.1.14 Interazione: *accettazione di una soluzione*

Tale interazione è identica all'omonima definita nella fase di progettazione del primo prototipo, tranne per il fatto che l'azione AcceptAction non rimuove tutte le teorie attive dal PrologEngine.

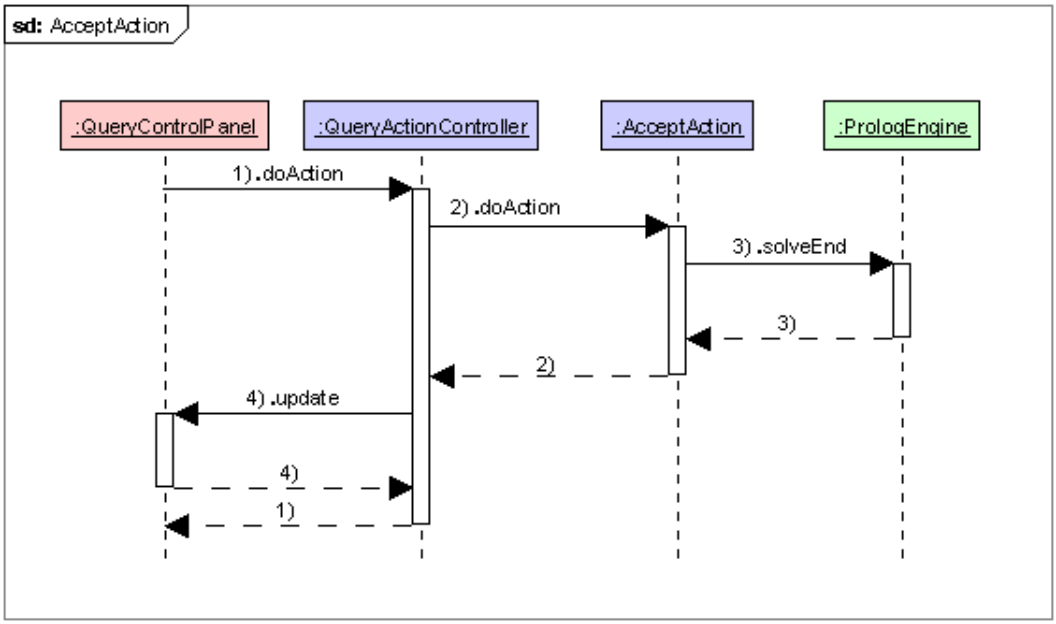


Figura 5.20: Diagramma relativo alla nuova sequenza "accettazione di una soluzione"

5.1.15 Interazione: *importazione teoria dal filesystem locale*

Tale interazione viene avviata con un comando da parte dell'utente, impartito tramite l'interfaccia grafica implementata dall'ActiveTheoryListView, il quale delegherà all'AddTheoryAction la gestione dell'esecuzione dell'azione, la quale richiederà al LocalTheoryLoader di ottenere una teoria.

Il LocalTheoryLoader delega al FileChooser la presentazione della richiesta di scelta del file contenente una teoria da caricare, oltre all'acquisizione di locazione e nome di tale file; dati locazione e nome, il LocalTheoryLoader caricherà il corrispondente file dal LocalFileSystem; una volta caricato il file, verrà estratta la teoria da esso e restituita all'AddTheoryAction.

Soltanto a questo punto l'AddTheoryAction può aggiungere la nuova teoria all'ActiveTheoryList. Infine, l'ActiveTheoryList notificherà i suoi cambiamenti all'ActiveTheoryListView.

Nel caso in cui l'utente non scelga alcun file, il LocalTheoryLoader non deve restituire alcunché, così l'AddTheoryAction capisce che l'azione deve terminare in quell'istante, senza notifiche di errore. Nel caso in cui sia sollevata un'eccezione durante il tentativo di caricamento di una teoria, questa deve essere notificata all'AddTheoryAction, in modo tale da poter essere segnalata all'utente.

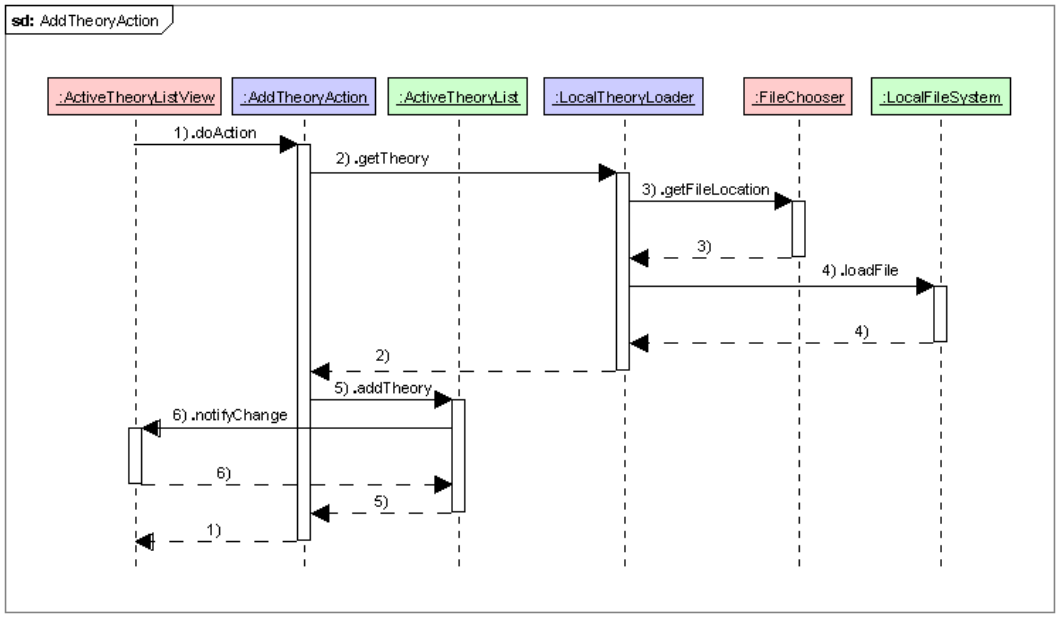


Figura 5.21: Diagramma di sequenza relativo all'interazione "importazione teoria dal filesystem locale"

5.1.16 Interazione: *esportazione teoria verso il filesystem locale*

Tale interazione può essere avviata da una qualsiasi entità, la quale delegherà alla ExportTheoryAction la gestione dell'esecuzione dell'azione, la quale richiederà al LocalTheorySaver di salvare una teoria.

Il LocalTheorySaver delega al FileChooser la presentazione della richiesta di scelta del file che conterrà la teoria da salvare, oltre all'acquisizione di locazione e nome di tale file; dati locazione e nome, il LocalTheorySaver richiederà di salvare il file al LocalFileSystem.

Nel caso in cui l'utente non scelga alcun file, il LocalTheorySaver non deve restituire alcunché, così l'ExportTheoryAction capisce che l'azione deve terminare in quell'istante, senza notifiche di errore. Nel caso in cui sia sollevata un'eccezione

durante il tentativo di salvataggio di una teoria, questa deve essere notificata all'ExportTheoryAction, in modo tale da poter essere segnalata all'utente.

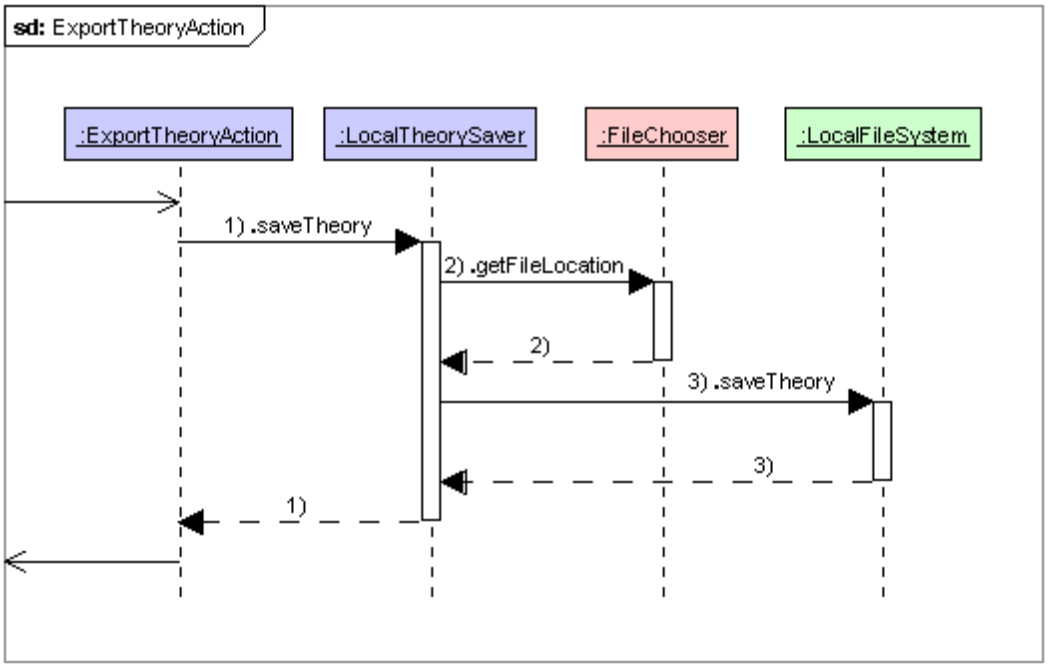


Figura 5.22: Diagramma di sequenza "esportazione teoria verso il filesystem locale"

5.1.17 Interazione: *estrazione di una teoria attiva*

Tale interazione viene avviata con un comando da parte dell'utente, impartito tramite l'interfaccia grafica implementata dall'ActiveTheoryListView, il quale delegherà alla GetTheoryAction la gestione dell'esecuzione dell'azione.

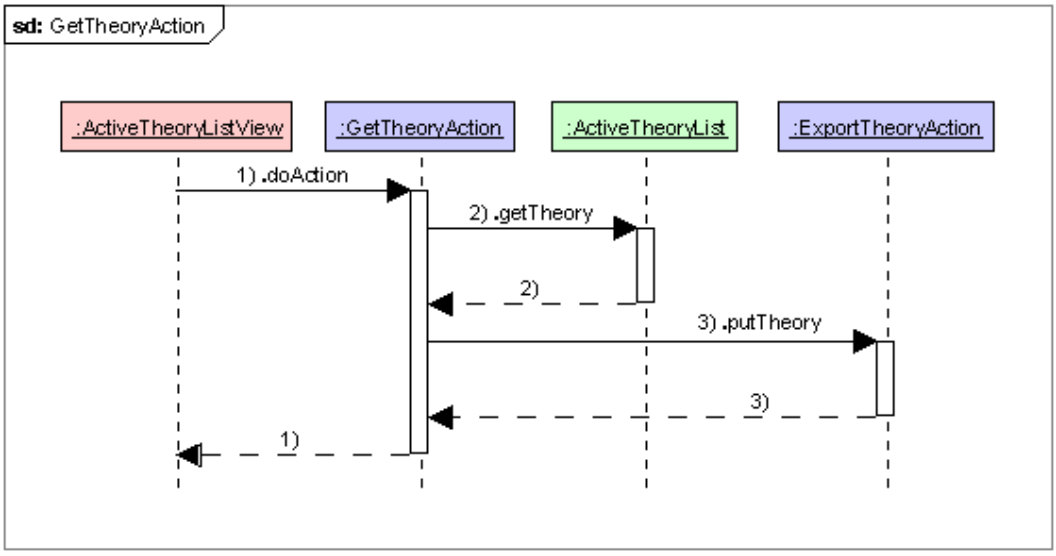


Figura 5.23: Diagramma di sequenza dell'interazione "estrazione di una teoria attiva"

L'azione GetTheoryAction dovrà estrarre la teoria da esportare dall'ActiveTheoryList e infine richiedere all'ExportTheoryAction di eseguire l'esportazione della teoria estratta.

5.1.18 Interazione: *estrazione della teoria attiva globale*

Tale interazione è del tutto identica alla precedente, tranne per il fatto che la teoria estratta dall'ActiveTheoryList è la teoria attiva globale e al posto dell'azione GetTheoryAction è utilizzata l'azione GetWholeTheoryAction.

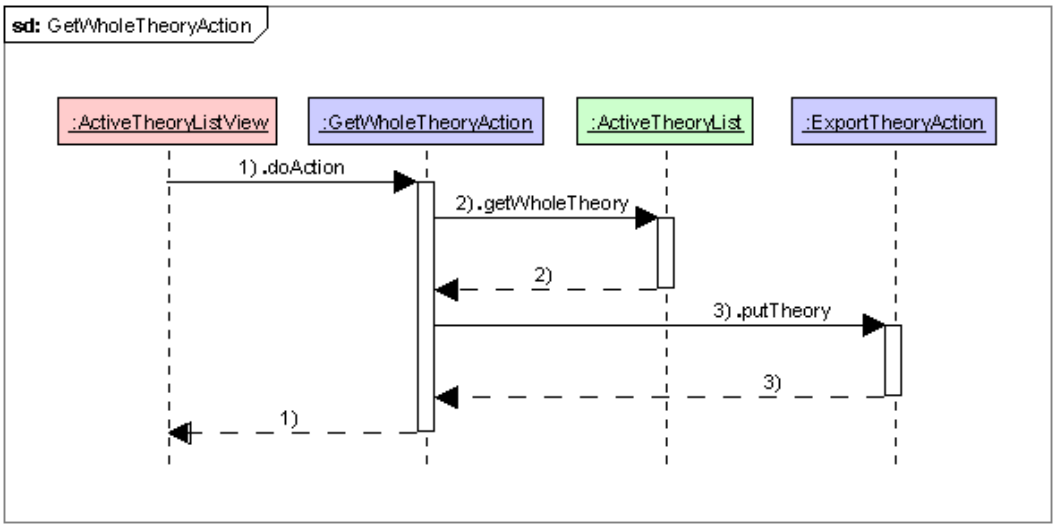


Figura 5.24: Diagramma di sequenza dell'interazione "estrazione della teoria attiva globale"

5.1.19 Progettazione dei partecipanti: *ActiveTheoryList, TheoryManager e PrologEngine*

L'ActiveTheoryList deve fornire le funzionalità di accesso e modifica dell'elenco delle teorie attive, ovvero:

- aggiunta di una teoria in fondo all'elenco delle teorie attive;
- impostazione di un'unica teoria attiva;
- rimozione di tutte le teorie attive;
- rimozione di una qualunque teoria attiva[1];
- recupero della teoria attiva globale;
- recupero di una teoria attiva, specificando la sua posizione nell'elenco;

1 Dato che verranno realizzate solamente le funzionalità supportate, è possibile rimuovere solamente l'ultima teoria dotata di uno specifico nome; pertanto, per rispettare tale restrizione la voce del menù contestuale relativa a tale funzionalità non dovrà essere sempre attiva, ma opportunamente attiva o disattiva.

- recupero di tutte le teorie attive.

Come già stabilito, l'elenco delle teorie attive deve notificare i cambiamenti relativi al suo contenuto inviando eventi ai corrispondenti listener; tali eventi devono contenere le informazioni riguardanti il cambiamento a cui è stato sottoposto l'elenco delle teorie attive. Quindi l'ActiveTheoryList deve essere dotato di opportuni metodi per la registrazione/cancellazione di listener relativi agli eventi di modifica dell'elenco delle teorie attive. Inoltre, la lista delle teorie attive può essere modificata durante la risoluzione di una query, quindi l'entità ActiveTheoryList dovrebbe ricevere le notifiche relative agli eventi di modifica dell'elenco delle teorie attive dal motore tuProlog.

Il TheoryManager è quel componente che costituisce l'interfaccia di gestione delle teorie attive del PrologEngine; il suo ruolo consiste nel servire le richieste di modifica dell'elenco delle teorie attive e gestire il caricamento delle teorie nel motore tuProlog. Dato che in questo prototipo verranno implementate solamente le funzionalità supportate, il TheoryManager attuerà immediatamente le modifiche alle teorie caricate nel motore tuProlog; in un successivo prototipo tale entità supporterà le funzionalità avanzate, caricando le teorie attive nel motore tuProlog nel momento più opportuno. Per evitare che il TheoryManager debba rivolgersi ogni volta al PrologEngine per eseguire il caricamento/rimozione di una teoria attiva, esso dovrebbe accedere direttamente al vero e proprio motore tuProlog incapsulato nel PrologEngine. I componenti del plug-in interessati a ricevere notifiche relative ad eventi di modifica dell'elenco delle teorie attive, devono registrarsi presso il TheoryManager e non presso il PrologEngine, dato che le operazioni di gestione delle teorie sono di competenza esclusiva del TheoryManager.

Il TheoryManager può essere considerato parte del PrologEngine, innanzitutto perché deve condividere con esso l'accesso al motore tuProlog vero e proprio ed anche perché sarà esclusivamente il TheoryManager a gestire le teorie attive che, da un punto di vista logico, sono caricate nel PrologEngine. Così, in tal modo, si suddividono le responsabilità, e si rende più leggera la classe che implementerà il PrologEngine. Quindi, si procederà in modo tale che il TheoryManager venga creato dal PrologEngine, il quale ne permetterà il recupero con un apposito metodo. Il PrologEngine dovrà creare un'unica istanza della classe che implementerà il TheoryManager, pertanto l'unicità del PrologEngine garantirà

l'unicità del TheoryManager, nell'ipotesi che solo il PrologEngine crei il TheoryManager. Infine, si osservi che questa soluzione permette al PrologEngine di passare al TheoryManager, nel momento relativo alla sua creazione, il riferimento del motore tuProlog vero e proprio in esso incapsulato, necessario per poter permettere al TheoryManager il caricamento delle teorie.

Il TheoryManager, essendo l'unica interfaccia del motore tuProlog per la gestione delle teorie attive, verrà utilizzato dall'ActiveTheoryList per modificare l'elenco delle teorie attive; le modifiche apportate causeranno il lancio di eventi relativi al cambiamento delle teorie attive, pertanto tutti i listener interessati a questi eventi saranno notificati. Dato che l'ActiveTheoryList è un listener di questo tipo, esso verrebbe notificato mentre sta eseguendo la modifica; ciò deve essere evitato, perché comporterebbe una doppia notifica dei listener registrati presso l'ActiveTheoryList: una prima notifica dovuta alla segnalazione del TheoryManager e una seconda notifica segnalata dall'ActiveTheoryList al termine dell'esecuzione della modifica all'elenco delle teorie attive. Conseguentemente, l'ActiveTheoryList dovrà ricevere le notifiche relative alle teorie attive solamente durante la risoluzione di query.

Il cambiamento relativo alle teorie attive non verrà notificato dal TheoryManager, semplicemente perché la notifica del TheoryManager avrà origine dal motore tuProlog vero e proprio, il quale è in grado di segnalare solamente le modifiche relative alle teorie effettivamente caricate e non relative alle teorie attive; inoltre, le segnalazioni del motore tuProlog non contengono informazioni sulla particolare modifica effettuata, ma solamente sullo stato della teoria dinamica prima e dopo l'esecuzione della modifica. Per tali motivi si preferisce che l'ActiveTheoryList segnali le modifiche che effettua, in modo tale da fornire ai listener maggiori dettagli relativi al cambiamento eseguito. La mancanza di dettagli relativi ai cambiamenti all'elenco delle teorie attive è inevitabile per quanto riguarda le modifiche effettuate durante la risoluzione di una query; pertanto, al ricevimento delle relative notifiche, i listener registrati presso l'ActiveTheoryList non possono fare un aggiornamento mirato ma devono effettuarne uno completo, quindi richiedono le informazioni relativamente a tutto l'elenco delle teorie attive. Perciò si è scelto che tali listener siano notificati tramite eventi di modifica dell'elenco delle teorie attive provenienti esclusivamente dall'ActiveTheoryList e non dal motore tuProlog; quindi, sarà l'ActiveTheoryList ad essere il listener degli eventi

relativi alle teorie attive presso il TheoryManager e propagherà tali eventi ai propri listener includendo le informazioni sull'elenco delle teorie attive.

Sostanzialmente, l'ActiveTheoryList svolgerà il ruolo di *adapter* del TheoryManager; infatti i componenti del plug-in, interessati a gestire l'elenco delle teorie attive, utilizzeranno solamente l'ActiveTheoryList e non il TheoryManager. Quindi è possibile realizzare l'ActiveTheoryList in modo tale da essere il più adeguato possibile per l'ambiente NetBeans, conseguentemente potrebbe essere benissimo un *JavaBean*.

In questo prototipo, l'unico componente del plug-in che utilizzerà il TheoryManager sarà l'ActiveTheoryList, pertanto, affinché il TheoryManager non possa essere creato o utilizzato da altri componenti del plug-in, il metodo della classe `PrologEngine` che lo restituisce e la classe che lo implementa dovranno avere un livello di visibilità limitato, in modo tale da escludere i componenti al di fuori del loro contenitore logico. Conseguentemente si ha la necessità di strutturare i sorgenti del plug-in raccogliendoli in opportuni package.

Nell'analisi del problema è stato stabilito che l'elenco delle teorie attive fosse unico, quindi è naturale pensare all'uso del design pattern *Singleton* [GoF95] per realizzare l'ActiveTheoryList.

Infine, al PrologEngine dovrà essere aggiunta una nuova funzionalità: la generazione di eventi relativi all'inizio e alla terminazione di risoluzioni di query, i quali saranno denominati *SolveEvent*; questi eventi consentiranno all'ActiveTheoryList di attivare la ricezione delle notifiche relative ai cambiamenti delle teorie attive effettuati dal motore tuProlog durante la risoluzione di query. Questa nuova funzionalità è anche utile alla disattivazione di quei widget che devono essere inattivi durante la risoluzione di query, come l'ActiveTheoryListView. Pertanto, il PrologEngine dovrà fornire dei metodi per la registrazione/cancellazione dei listener relativi ai SolveEvent.

Nei casi in cui devono essere risolte query temporanee, lo stato relativo all'elenco delle teorie attive deve essere memorizzato precedentemente alla risoluzione della query, e, una volta terminata la risoluzione, esso deve essere ripristinato. Tutto ciò deve essere svolto dal TheoryManager, ma questo non ha cognizione di quando inizia o termina una risoluzione; conseguentemente dovrà essere il PrologEngine a dare questa indicazione. Il TheoryManager potrebbe essere registrato come

listener dei SolveEvent, ma questa soluzione consentirebbe ad un terzo oggetto di cancellare il TheoryManager come listener del PrologEngine, rompendo di fatto la nostra architettura; pertanto, il TheoryManager sarà dotato di appositi metodi per poter essere informato dal PrologEngine quando inizia o termina la risoluzione di una query.

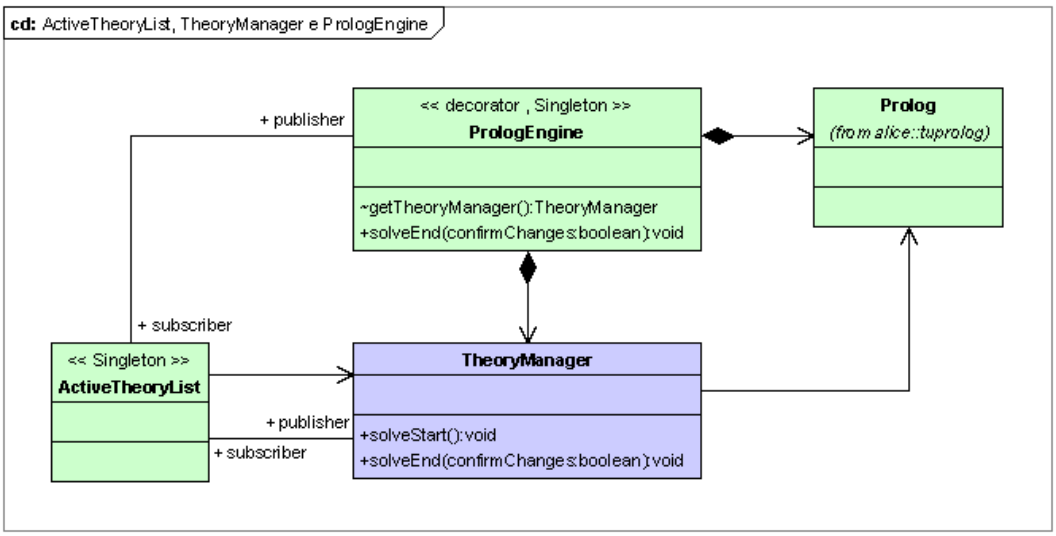


Figura 5.25: Diagramma delle classi ActiveTheoryList, TheoryManager e PrologEngine

Rimane un'ultima questione: in che modo indicare al PrologEngine che la query richiesta deve essere temporanea. Una valida soluzione sarebbe quella di modificare la semantica del metodo `solveEnd` del PrologEngine; infatti, tale metodo è stato inizialmente concepito per permettere di accettare una soluzione in caso di risultati multipli, ma esso dovrà svolgere un compito più ampio: indicare la terminazione della risoluzione di una query, ricevendo l'indicazione se la query appena risolta era temporanea o meno. Pertanto, dovrà essere aggiunto alla signature del metodo `solveEnd` un parametro booleano che indicherà se confermare i cambiamenti effettuati nella teoria dinamica oppure ignorarli. Conseguentemente dovranno essere modificate anche le azioni StartAction, StopAction e AcceptAction, in modo tale che indichino la terminazione della risoluzione delle query richiamando il metodo `solveEnd`; con il parametro booleano della `solveEnd` dovranno indicare se le eventuali modifiche all'elenco delle teorie attive sono da confermare o ignorare.

5.1.20 Progettazione dei partecipanti: *ActiveTheoryListView*

Nell'analisi del problema è stato notato che occorre una vista delle teorie attive, ovvero un componente grafico che sia in grado di rappresentare e gestire l'elenco delle teorie attive; è stato notato anche che le funzionalità di gestione dell'elenco devono essere offerte tramite appositi menù contestuali. Inoltre, è stato stabilito anche che la vista delle teorie attive deve permettere di indicare se includere o meno una determinata teoria nella valutazione di una query; la realizzazione di tale funzionalità è rimandata ad un successivo prototipo.

Quindi, l'ActiveTheoryListView permette all'utente di avere il controllo sulle teorie attive utilizzate per la valutazione di una query; conseguentemente, è naturale pensare di inserire tale vista all'interno del pannello di controllo delle query, ovvero il QueryControlPanel. Ovviamente, affinché ciò sia possibile, l'ActiveTheoryListView deve essere un componente grafico *Swing*, eventualmente dotato di barre di scorrimento al fine di poter rappresentare anche ampi elenchi di teorie attive.

Affinché l'ActiveTheoryListView possa rappresentare correttamente l'elenco delle teorie attive, deve poter visualizzare una serie di widget rappresentanti le teorie attive, posti in un ordine tale da rispecchiare l'elenco delle teorie attive. In base a tali caratteristiche e al fine di ottenere una migliore integrazione nell'ambiente NetBeans, si potrebbe pensare di implementare l'ActiveTheoryListView sotto forma di *explorer view*; esso sarà caratterizzato dal nodo principale ActiveTheoryListNode, costituito da un elenco di nodi foglia TheoryNode. L'ActiveTheoryListNode incapsulerà la lista delle teorie attive, mentre i TheoryNode incapsuleranno le singole teorie.

Per fare in modo che l'ActiveTheoryListView rappresenti sempre l'elenco delle teorie attive corrente, l'ActiveTheoryListNode deve essere aggiornato relativamente agli eventuali cambiamenti dell'elenco delle teorie attive, quindi deve essere un listener degli eventi di modifica dell'elenco delle teorie attive, registrato presso l'ActiveTheoryList.

Dato che il QueryControlPanel realizza il design pattern *Façade* [GoF95], l'ActiveTheoryListNode non deve essere visibile esternamente, ma incapsulato al suo interno; quindi esso sarà registrato come listener degli eventi di modifica dell'elenco delle teorie attive presso l'ActiveTheoryList durante l'inizializzazione

del QueryControlPanel.

Inoltre, l'ActiveTheoryListView deve essere disattivato sia durante la risoluzione di una query, sia quando sono eventualmente disponibili ulteriori risoluzioni, perché non ha senso che l'utente possa cambiare l'elenco delle teorie attive durante una elaborazione di una query, né prima di eseguire la ricerca di ulteriori soluzioni. Pertanto, l'ActiveTheoryListView deve essere notificato dal PrologEngine relativamente agli eventi di inizio e terminazione di risoluzione di query, quindi deve essere un listener di SolveEvent registrato presso il PrologEngine. Nel momento in cui riceverà una notifica di inizio risoluzione query, esso si disattiverà automaticamente, mentre nel momento in cui riceverà una notifica di terminazione, esso si riattiverà automaticamente.

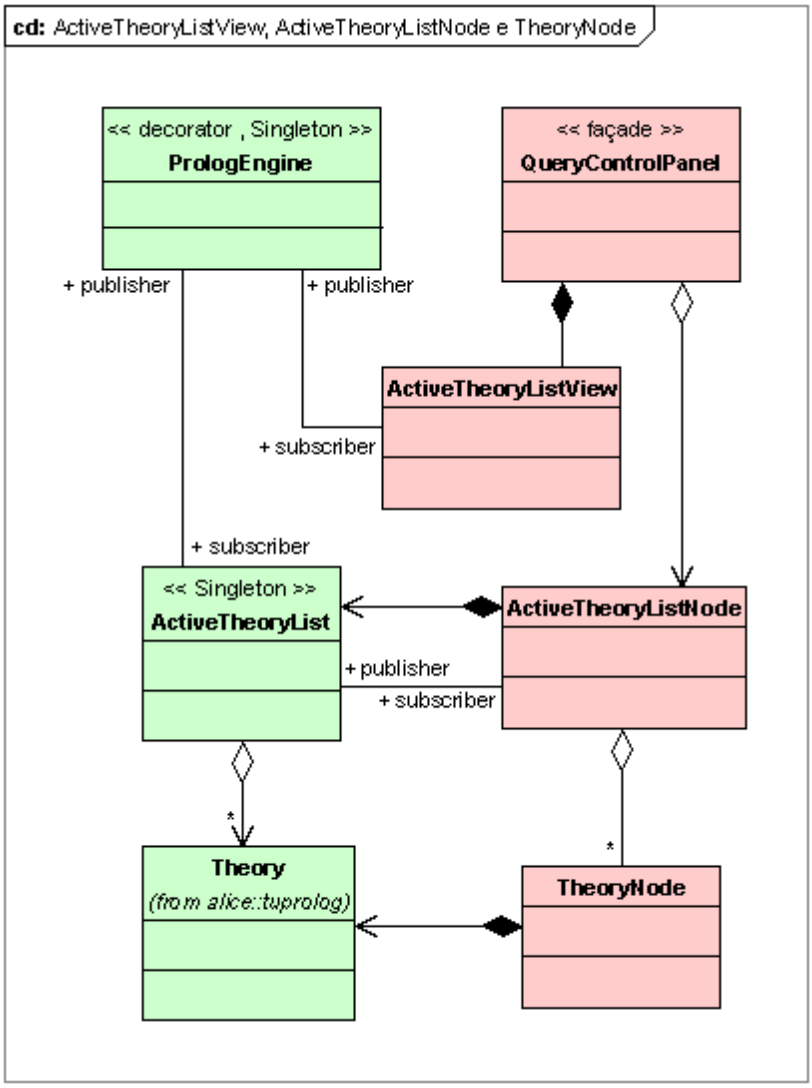


Figura 5.26: Diagramma delle classi ActiveTheoryListView, ActiveTheoryListNode e TheoryNode

5.1.21 Progettazione dei partecipanti: azioni di gestione delle teorie attive

Precedentemente è stato accennato che l'ActiveTheoryListView deve essere in grado di permettere all'utente l'esecuzione di opportune azioni di gestione dell'elenco delle teorie attive; tali azioni si suddividono in due tipologie:

- *azioni che si riferiscono a tutto l'elenco delle teorie attive*:
  - AddTheoryAction: aggiunge una teoria in fondo all'elenco delle teorie attive;
  - GetWholeTheoryAction: estrae la teoria attiva globale dall'elenco delle teorie attive;
  - RemoveAllTheoriesAction: rimuove tutte le teorie dall'elenco delle teorie attive;
- *azioni che si riferiscono ad una specifica teoria attiva*:
  - GetTheoryAction: estrae una specifica teoria dall'elenco delle teorie attive;
  - RemoveTheoryAction: rimuove la teoria indicata dall'elenco delle teorie attive;
  - SetTheoryAction: imposta la teoria indicata come unica teoria attiva.

Le azioni relative ad una specifica teoria attiva devono essere richiamabili esclusivamente attraverso un *menù contestuale relativo ad una specifica teoria*; tale tipo di menù contestuale è richiamabile da parte dell'utente cliccando con il pulsante destro sopra il widget che rappresenta la specifica teoria. Le azioni relative alla struttura dell'elenco delle teorie attive devono essere richiamabili sia attraverso un qualunque menù contestuale relativo ad una specifica teoria, sia attraverso un *menù contestuale relativo alla vista delle teorie attive*; tale tipo di menù contestuale è richiamabile da parte dell'utente cliccando con il pulsante destro in uno spazio vuoto all'interno dell'ActiveTheoryListView.

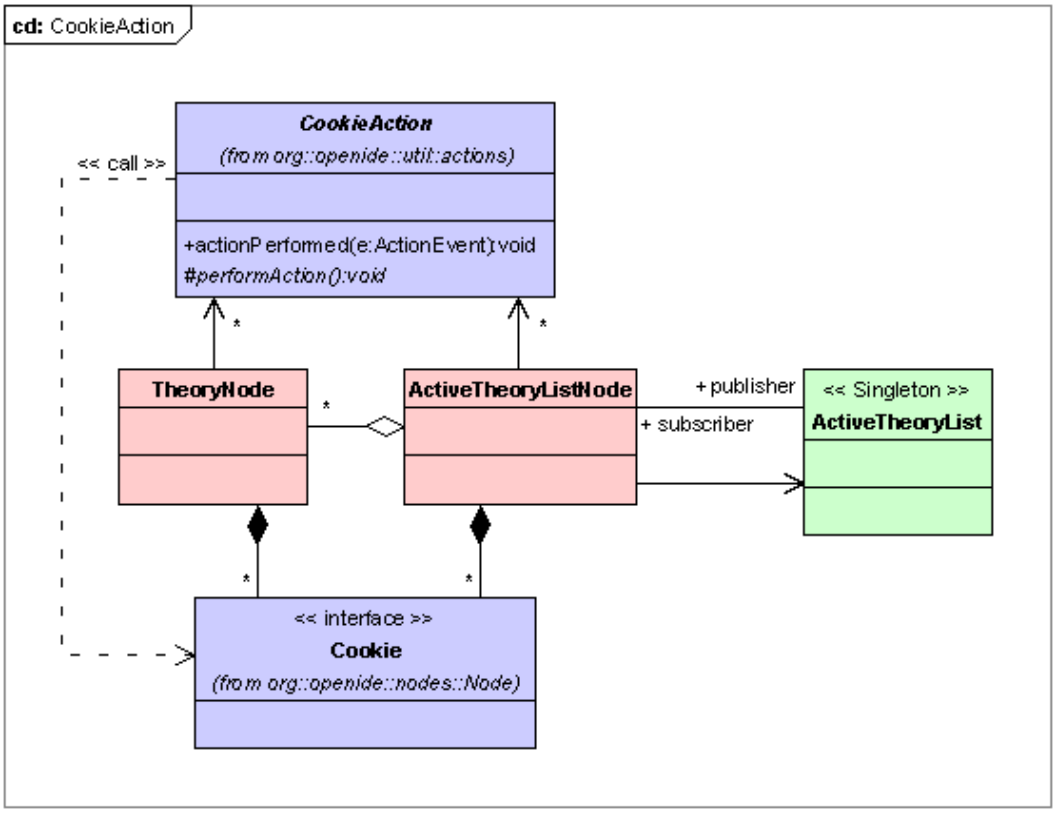


Figura 5.27: Diagramma della classe CookieAction

Le azioni AddTheoryAction, GetTheoryAction, GetWholeTheoryAction e SetTheoryAction sono sempre attive, mentre l'azione RemoveAllTheoriesAction deve essere attiva solamente se l'elenco delle teorie attive contiene almeno una teoria, mentre l'azione RemoveTheoryAction deve essere attiva solamente se si riferisce all'ultima teoria presente nell'elenco delle teorie attive dotata di uno specifico nome. Dato che ognuna di queste azioni opera in uno specifico contesto, esse devono essere richiamate direttamente dal widget specifico a cui si riferiscono, ovvero o dall'ActiveTheoryListNode o da un qualunque TheoryNode; inoltre sono attivabili condizionatamente allo stato del contesto; pertanto tali azioni devono implementare delle *cookie action*, le quali richiamano uno specifico *cookie* incapsulato nel nodo che delinea il contesto di esecuzione dell'azione.

Infine, le istanze di tali azioni devono essere uniche, quindi quindi ad esse verrà applicato il design pattern *Singleton*.

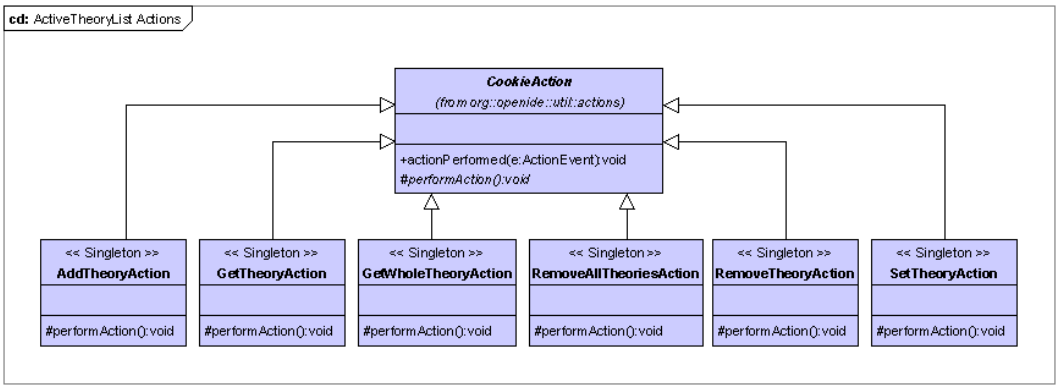


Figura 5.28: Diagramma delle classi AddTheoryAction, GetTheoryAction, GetWholeTheoryAction, RemoveAllTheoriesAction, RemoveTheoryAction, SetTheoryAction

5.1.22 Progettazione dei partecipanti: *LocalTheoryLoader* e *LocalTheorySaver*

Il LocalTheoryLoader e il LocalTheorySaver sono due entità che gestiscono, rispettivamente, il caricamento e il salvataggio di teorie su filesystem locale; sono accomunate dall'utilizzo dei partecipanti FileChooser e LocalFileSystem. Il FileChooser è un widget che permette all'utente di specificare locazione e nome del file da cui caricare una teoria o in cui salvare una teoria; il LocalFileSystem è un'entità che offre servizi per la lettura o la scrittura di file su filesystem locale. Le istanze di tali entità devono essere uniche, quindi ad esse verrà applicato il design pattern *Singleton*.

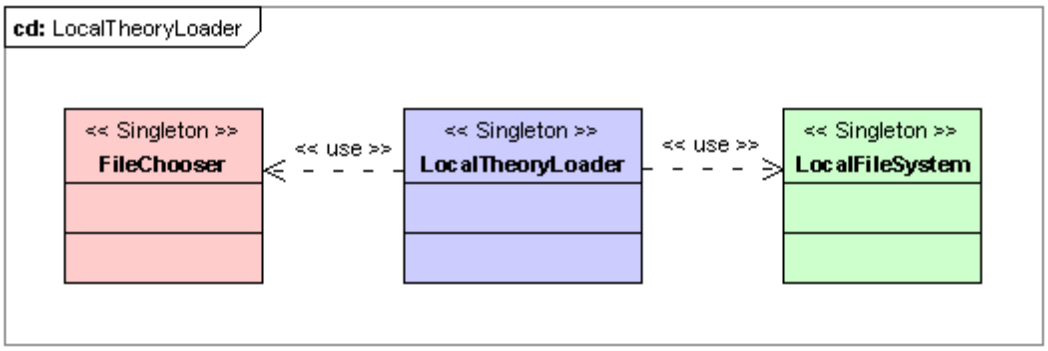


Figura 5.29: Diagramma della classe LocalTheoryLoader

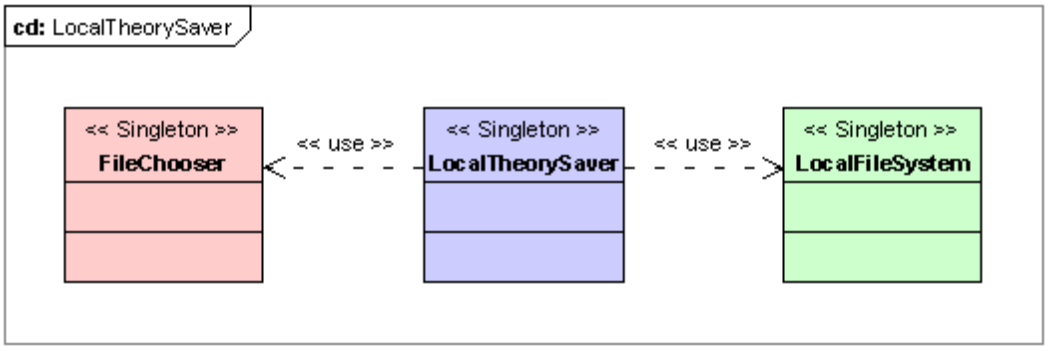


Figura 5.30: Diagramma della classe LocalTheorySaver

5.1.23 Progettazione dei partecipanti: *AddTheoryAction e ExportTheoryAction*

Le azioni AddTheoryAction ed ExportTheoryAction permettono, rispettivamente, di caricare una teoria da filesystem locale e di salvare una teoria su filesystem locale; per svolgere il proprio ruolo, la prima azione utilizza il LocalTheoryLoader, mentre la seconda utilizza il LocalTheorySaver. La ExportTheoryAction è a sua volta utilizzata dalle azioni GetTheoryAction e GetWholeTheoryAction, mentre la AddTheoryAction è richiamata tramite dall'ActiveTheoryListView. Entrambe realizzano il design pattern *Singleton*.

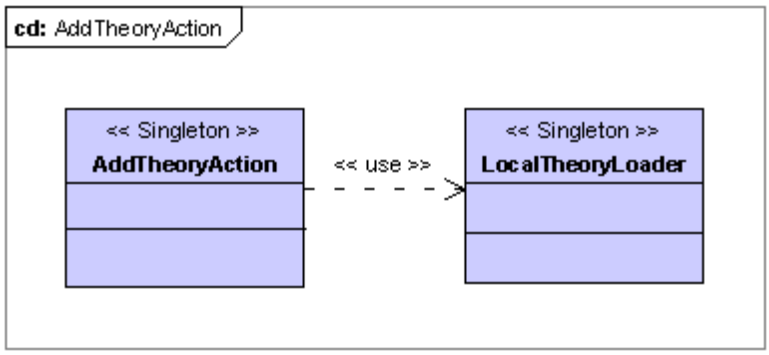


Figura 5.31: Diagramma della classe AddTheoryAction

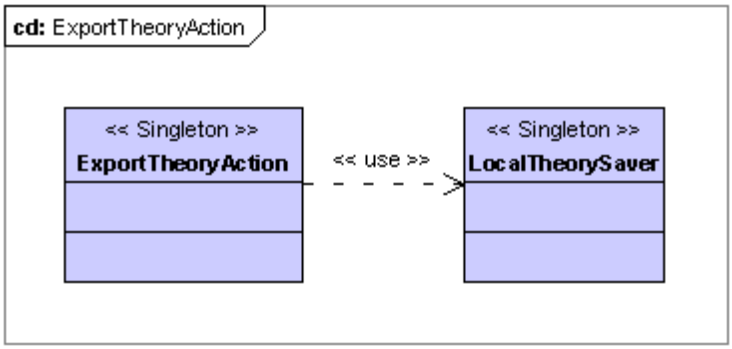


Figura 5.32: Diagramma della classe ExportTheoryAction

5.1.24 Inizializzazione dei partecipanti

Molti dei nuovi partecipanti, introdotti per la realizzazione del secondo prototipo, realizzano il design pattern *Singleton*, il quale ne predispone l'inizializzazione automatica su necessità.

I partecipanti che non utilizzano il design pattern *Singleton* e che, pertanto, devono essere opportunamente inizializzati sono l'ActiveTheoryListView e l'ActiveTheoryListNode; essi verranno inizializzati rispettivamente dal QueryControlPanel e dal PrologTopComponent; inoltre, il PrologTopComponent aggregherà l'ActiveTheoryListNode al QueryControlPanel.

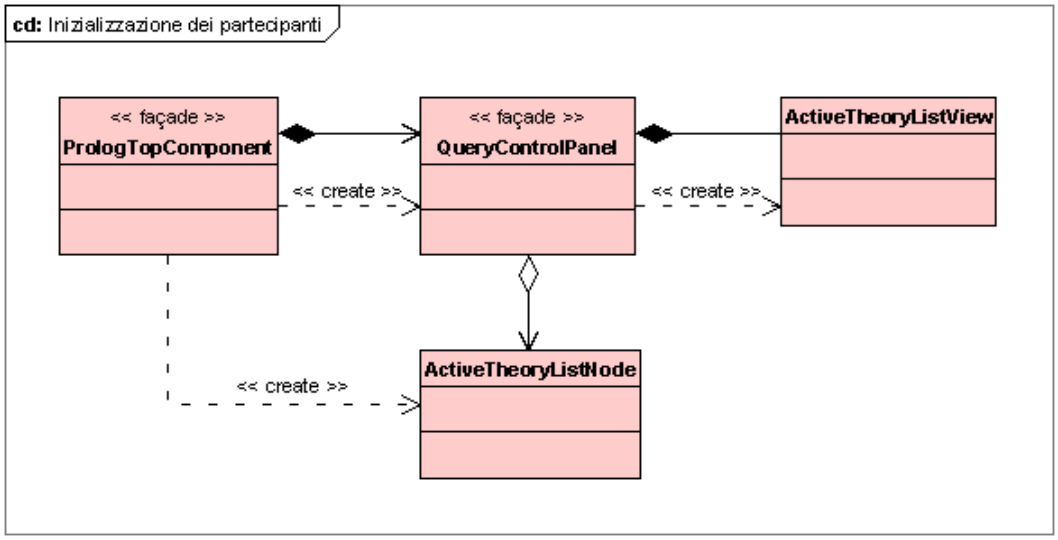


Figura 5.33: Diagramma delle classi relativo all'inizializzazione dei partecipanti

5.2 Architettura del secondo prototipo

Alla luce delle precedenti considerazioni, è possibile delineare la nuova architettura del sistema, rappresentata nella figura 5.34, a pagina seguente.

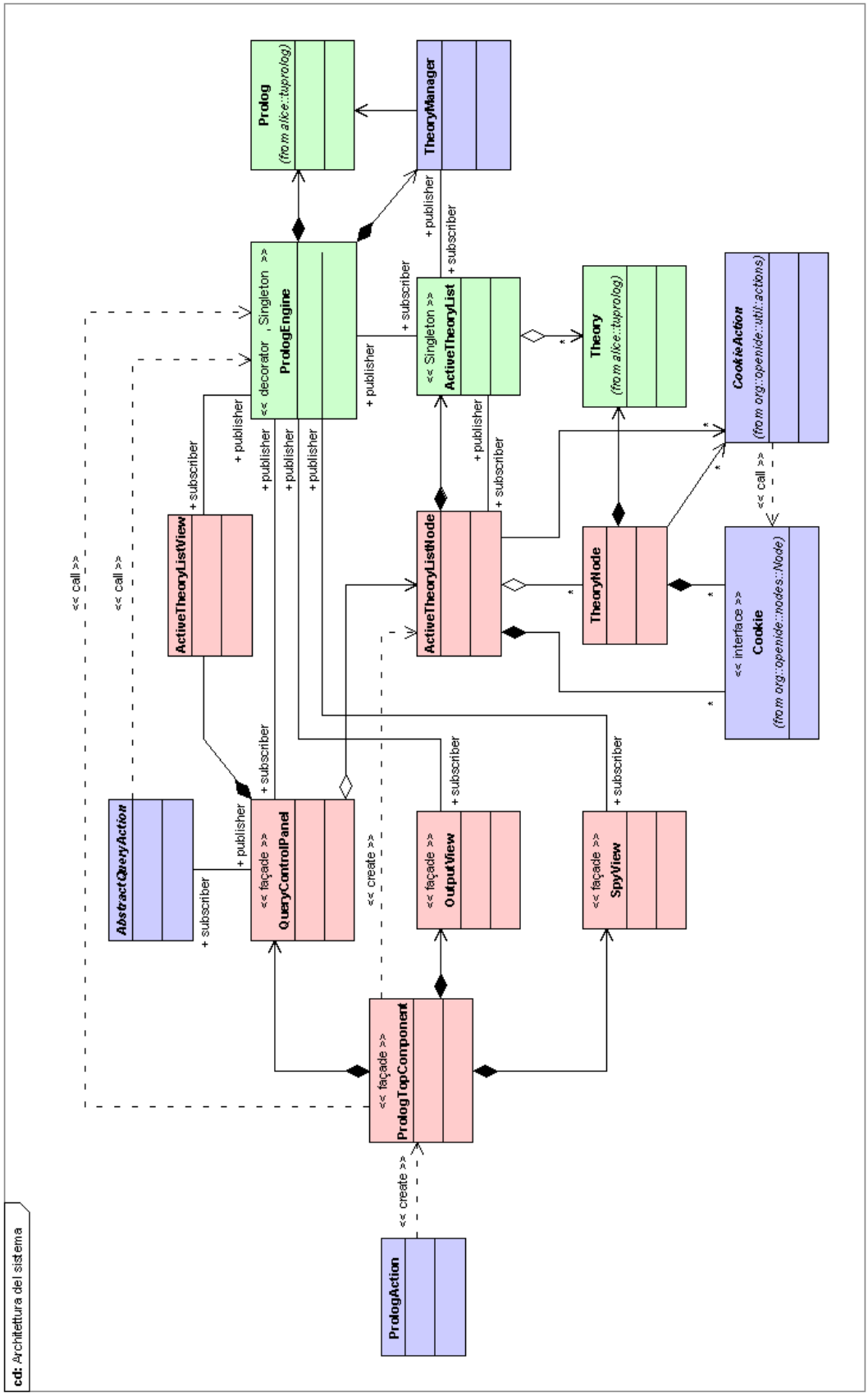


Figura 5.34: Architettura del sistema, relativamente al secondo prototipo

5.3 Implementazione del secondo prototipo

In base alle indicazioni elaborate nella fase di progettazione, in questa sezione verrà delineata l'implementazione del secondo prototipo.

5.3.1 Strutturazione dei package

Il primo prototipo presentava un numero di classi e interfacce relativamente piccolo; con il secondo prototipo si prevede che il numero di queste aumenti significativamente, pertanto sarà necessario raggrupparle in opportuni sotto-package del package `alice.tuprologx.netbeans`. Per una maggiore comprensione della strutturazione del codice sorgente in sotto-package si potrebbe ricalcare la struttura tipica dei principali package delle NetBeans API [NbAPI]; quindi i sotto-package del plug-in potrebbero essere i seguenti:

- `alice.tuprologx.netbeans.actions`: raccoglie classi e interfacce per l'implementazione di azioni;
- `alice.tuprologx.netbeans.icons`: raccoglie le icone necessarie all'implementazione del plug-in;
- `alice.tuprologx.netbeans.filesystems`: raccoglie classi che permettono l'accesso ai supporti per la persistenza;
- `alice.tuprologx.netbeans.loaders`: raccoglie classi per la gestione della persistenza;
- `alice.tuprologx.netbeans.nodes`: raccoglie le classi che implementano il livello di presentazione delle risorse e le interfacce che specificano il comportamento delle operazioni che possono essere svolte sulle risorse;
- `alice.tuprologx.netbeans.view`: raccoglie le classi che implementano la vista del plug-in.

In base a queste scelte, i componenti delle implementazioni di alcuni partecipanti del primo prototipo verranno spostati nell'opportuno package:

- `alice.tuprologx.netbeans.actions`: classi `AbstractQueryAction`, `AbstractAsyncQueryAction`, `StartAction`, `StopAction`, `NextAction`, `AcceptAction` e `QueryActionEvent`, interfacce `IQueryActionController` e `IQueryActionListener`;

- `alice.tuprologx.netbeans.view`: classi `QueryControlPanel`, `BlackboardView`, `OuputView` e `SpyView`.

Le classi `PrologAction` e `PrologTopComponent` non verranno spostate perché referenziate all'interno dello *XML Layer*, il quale impone che si trovino nel suo stesso package. La classe `PrologEngine` rimane dov'è perché costituisce il vero e proprio nucleo del plug-in.

5.3.2 Implementazione dell'*ActiveTheoryList*, del *TheoryManager* e del *PrologEngine*

La classe che implementa l'ActiveTheoryList è denominata `alice.tuprologx.netbeans.ActiveTheoryList` e realizza il design pattern *Singleton*, pertanto è dotata di un costruttore a visibilità *private* e di un metodo per restituire, ed eventualmente istanziare, l'unica istanza della classe `ActiveTheoryList`. Come deciso in fase di progettazione, l'ActiveTheoryList deve essere l'interfaccia per la manipolazione dell'elenco delle teorie attive per gli altri componenti del plug-in; inoltre, in tale fase è stato suggerito che tale entità sia un *JavaBean*. Un JavaBean specifica una serie di proprietà attraverso una collezione di naming pattern relativamente ai metodi implementati. Il bean implementato dalla classe `ActiveTheoryList` specifica una sola proprietà indicizzata composta da istanze della classe `Theory`, denominata *theories*. Quindi secondo le specifiche JavaBeans [JBS96] dovranno essere presenti i metodi:

- `public Theory[] getTheories();`
- `public void setTheories(Theory[]);`
- `public Theory getTheories(int);`
- `public void setTheories(int,Theory).`

Utilizzando il naming pattern delle proprietà indicizzate tipicamente si attribuiscono nomi semanticamente non adeguati ai due metodi indicizzati; infatti, il nome del metodo `Theory getTheories(int)`, il quale restituisce un'unica teoria, e il nome del metodo `void setTheories(int,Theory)`, il quale imposta un'unica teoria, non sono coerenti. Pertanto, si è scelto di denominare diversamente i due metodi indicizzati, in modo tale che essi abbiano un nome coerente alla loro funzione svolta:

- `public Theory getTheory(int);`

- `public void setTheory(int,Theory).`

In questo modo, però, non si rispettando più le specifiche JavaBeans; ciò comporta seri problemi per quanto riguarda l'uso di quei componenti che per interagire con un JavaBean utilizzano l'introspezione. Questo prototipo non utilizza né implementa oggetti in grado di interagire con la classe `ActiveTheoryList` sfruttando l'introspezione, quindi si potrebbe anche non considerare il problema appena sollevato; però questa è una scelta che potrebbe limitare gli sviluppi futuri del plug-in. Pertanto, per risolvere tale problema, si potrebbe affiancare al bean `ActiveTheoryList` un descrittore di bean, ovvero un oggetto che realizzi l'interfaccia `java.beans.BeanInfo` [JBS96]. Tale descrittore deve essere implementato in modo tale che indichi come metodi di accesso e modifica indicizzati della proprietà *theories* rispettivamente i metodi `Theory getTheory(int)` e `void setTheory(int,Theory)`; esso sarà denominato `alice.tuprologx.netbeans.ActiveTheoryListBeanInfo`.

Relativamente all'elenco delle teorie attive, durante la fase di progettazione sono state specificate le funzionalità di accesso e modifica; per ottenere tali funzionalità sono stati introdotti ulteriori metodi alla classe `ActiveTheoryList`. Ecco l'elenco delle funzionalità e i relativi metodi utilizzati per implementarle:

- *aggiunta di una teoria in fondo all'elenco delle teorie attive*; si ottiene utilizzando il metodo:

  ```
  public void addTheory(Theory);
  ```

- *impostazione di un'unica teoria attiva*; si ottiene utilizzando in sequenza i metodi:

  ```
  public void clear();
  public void addTheory(Theory);
  ```

- *rimozione di tutte le teorie attiv*e; si ottiene utilizzando il metodo:

  ```
  public void clear();
  ```

- *rimozione di una qualunque teoria attiva*; si ottiene utilizzando il metodo:

  ```
  public void removeTheory(int);
  ```

- *recupero della teoria attiva globale*; si ottiene utilizzando il metodo:

  ```
  public Theory getWholeTheory();
  ```

- *recupero di una teoria attiva, specificando la sua posizione nell'elenco*; si ottiene utilizzando il metodo:

  `public Theory getTheory(int);`

- *recupero di tutte le teorie attive*; si ottiene utilizzando il metodo:

  `public Theory[] getTheories().`

Quindi alla classe `ActiveTheoryList` si devono aggiungere i metodi `void addTheory(Theory)`, `void clear()`, `void removeTheory(int)` e `Theory getWholeTheory()`. Escludendo il metodo `void removeTheory(int)`, i metodi della classe `ActiveTheoryList` utilizzano l'unica istanza della classe `TheoryManager` richiamando un solo metodo. Il metodo `void removeTheory(int)`, invece, fa un uso più articolato della classe `TheoryManager`: infatti, prima richiama il metodo `Theory getTheory(int)` per ottenere la teoria da rimuovere, dopodiché utilizza il nome di tale teoria per richiamare il metodo `void removeTheory(String)`, al fine di eseguire effettivamente la rimozione della teoria.

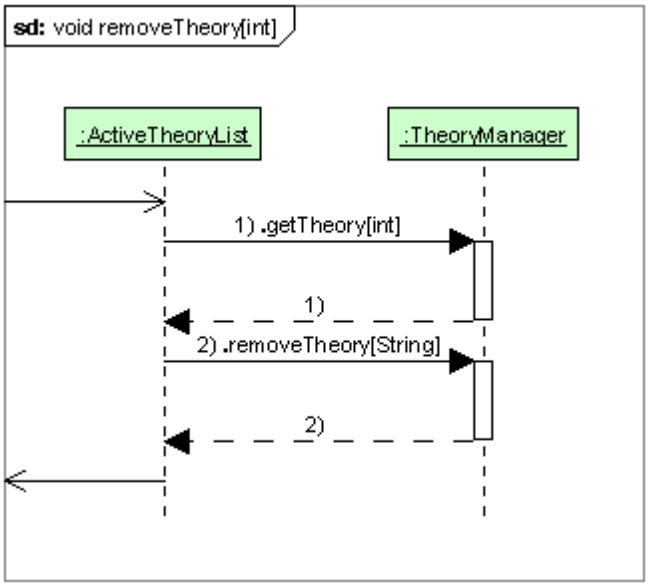


Figura 5.35: Diagramma di sequenza relativo al metodo void removeTheory(int) della classe ActiveTheoryList

Così facendo però si rimuove l'ultima teoria etichettata con uno specifico nome, non una teoria di indice qualunque. A causa del fatto che si stanno realizzando solamente le funzionalità supportate, è possibile rimuovere soltanto l'ultima teoria etichettata con uno specifico nome; tale restrizione deve essere applicata dai menù contestuali relativi alle teorie attive, disattivando le azioni di rimozione applicabili

a teorie che non siano le ultime dell'elenco ad essere dotate di una specifica etichetta. In questo modo il metodo `void removeTheory(int)` verrebbe richiamato soltanto per effettuare la rimozione di teorie che soddisfano tale restrizione.

I cambiamenti apportati dai metodi di modifica dovranno essere opportunamente notificati ai listener tramite eventi in grado di indicare le variazioni all'elenco delle teorie attive; pertanto, per implementare tali eventi si potrebbero utilizzare sia la classe `java.beans.PropertyChangeEvent`, sia la classe `java.beans.IndexedPropertyChangeEvent`. Conseguentemente, la classe `ActiveTheoryList` deve prevedere anche i seguenti metodi per la gestione dei listener:

- `public void addPropertyChangeListener( PropertyChangeListener );`
- `public void removePropertyChangeListener( PropertyChangeListener );`
- `public Collection<PropertyChangeListener> getPropertyChangeListeners().`

Per poter notificare gli eventi relativi alla modifica della proprietà *theories* si deve ricorrere ad un supporto per la notifica dei cambiamenti relativi alle proprietà; pertanto, la classe `ActiveTheoryList` deve incapsulare ed utilizzare un'istanza della classe `java.beans.PropertyChangeSupport`, al fine di registrare i listener e notificare gli eventi relativi ai cambiamenti di proprietà.

Infine si noti che i metodi `void setTheory(int,Theory)` e `void setTheories(Theory[])` non saranno ancora utilizzati per effettuare modifiche: infatti, dato che si stanno implementando solamente le funzionalità supportate, non si possono effettuare né la sostituzione di una teoria specificata da un qualunque indice, né la sostituzione di tutte le teorie dell'elenco delle teorie attive. Saranno introdotti solamente per un loro utilizzo in un successivo prototipo, allo stesso modo della classe `ActiveTheoryListBeanInfo`.

Il TheoryManager implementa l'interfaccia per la gestione delle teorie attive del motore tuProlog, pertanto tutti i metodi per la gestione delle teorie attive introdotti durante l'implementazione del primo prototipo sono stati trasferiti dalla classe `PrologEngine` alla classe che implementa il TheoryManager, denominata `alice.tuprologx.netbeans.TheoryManager`. Come specificato durante la fase di progettazione, il componente del plug-in che può utilizzare il

TheoryManager è uno solo, l'ActiveTheoryList, pertanto la classe `TheoryManager` deve avere una visibilità a livello di *package*, affinché vi abbiano accesso solamente classi appartenenti allo stesso package. Inoltre, il TheoryManager deve accedere direttamente al motore tuProlog vero e proprio, pertanto deve avere un riferimento all'istanza della classe `Prolog` incapsulata nella classe `PrologEngine`. Oltre a ciò, è stato stabilito anche che l'ActiveTheoryList potrà recuperare il TheoryManager solamente richiamando un apposito metodo della classe `PrologEngine`, denominato `TheoryManager getTheoryManager()`; esso è caratterizzato da visibilità a livello di *package*, in modo tale da limitare l'accesso al TheoryManager alle soli classi appartenenti al suo stesso package. Quindi, per soddisfare questi requisiti la classe `PrologEngine` crea e restituisce l'unica istanza della classe `TheoryManager`, passando in ingresso al costruttore un riferimento al motore tuProlog vero e proprio.

I metodi per la gestione delle teorie attive implementati dalla classe `TheoryManager`, in questo prototipo saranno:

- `public void addTheory(Theory)`: aggiunge una teoria in fondo all'elenco delle teorie attive;
- `public Theory[] getTheories()`: restituisce un array contenente tutte le teorie attive;
- `public Theory getTheory(int)`: restituisce la teoria corrispondente all'indice specificato;
- `public Theory getWholeTheory()`: restituisce la teoria dinamica;
- `public void removeAllTheories()`: rimuove tutte le teorie attive;
- `public void removeTheory(String)`: rimuove l'ultima teoria attiva etichettata con il nome specificato.

E' interessante notare che non compaia alcun metodo `void setTheory(Theory)`, evidenziando che l'equivalente metodo presente nella classe `Prolog` sia, limitatamente al nostro caso, diventato inutile.

Affinché l'ActiveTheoryList possa ricevere gli eventi relativi a cambiamenti nell'elenco delle teorie attive, esso dovrebbe registrarsi come listener al motore tuProlog vero e proprio; ma il motore tuProlog vero e proprio non è esternamente

visibile. Pertanto, l'ActiveTheoryList deve essere registrato come listener di tali eventi presso il gestore delle teorie attive, ovvero il TheoryManager, il quale ha provveduto automaticamente a registrare il nuovo listener al motore tuProlog vero e proprio. Quindi, la classe `ActiveTheoryList` deve realizzare l'interfaccia `alice.tuprolog.TheoryListener`, mentre la classe `TheoryManager` deve disporre dei metodi di registrazione/cancellazione dei listener relativi agli eventi di modifica dell'elenco delle teorie attive, ovvero i metodi:

- `public void addTheoryListener(TheoryListener);`
- `public void removeTheoryListener(TheoryListener);`
- `public Collection<TheoryListener> getTheoryListeners().`

Durante la progettazione è stata introdotta una nuova funzionalità del PrologEngine: la generazione di eventi relativi all'inizio e alla terminazione di risoluzioni di query; tali eventi saranno implementati dalla classe `alice.tuprologx.netbeans.SolveEvent`, la quale estende la classe `java.util.EventObject`. Per adesso non si prevedono funzionalità particolari per questa classe, la quale presenterà esclusivamente un costruttore a visibilità di *package*, il cui parametro d'ingresso sarà un riferimento alla sorgente dell'evento, ovvero l'unica istanza della classe `PrologEngine`. L'accorgimento di assegnare al costruttore della classe `SolveEvent` una visibilità a livello di *package* consente di limitare alla sola classe `PrologEngine` la generazione degli eventi SolveEvent. I listener di tali eventi dovranno realizzare l'interfaccia `alice.tuprologx.netbeans.ISolveListener`, la quale prevede i metodi:

- `public void onSolveStart(SolveEvent)`: richiamato all'inizio della risoluzione di una query;
- `public void onSolveEnd(SolveEvent)`: richiamato al termine della risoluzione di una query.

Rammentando quanto stabilito durante la fase di progettazione, l'ActiveTheoryList non deve ricevere gli eventi relativi alla modifica dell'elenco delle teorie attive mentre questo lo sta modificando, ma solamente quando il motore tuProlog lo sta modificando. Pertanto, l'unica istanza della classe `ActiveTheoryList` deve registrarsi come listener presso l'unica istanza della classe `PrologEngine` al fine di sapere quando iniziano o terminano le risoluzioni di query, in modo tale da potersi registrare come listener al TheoryManager nel

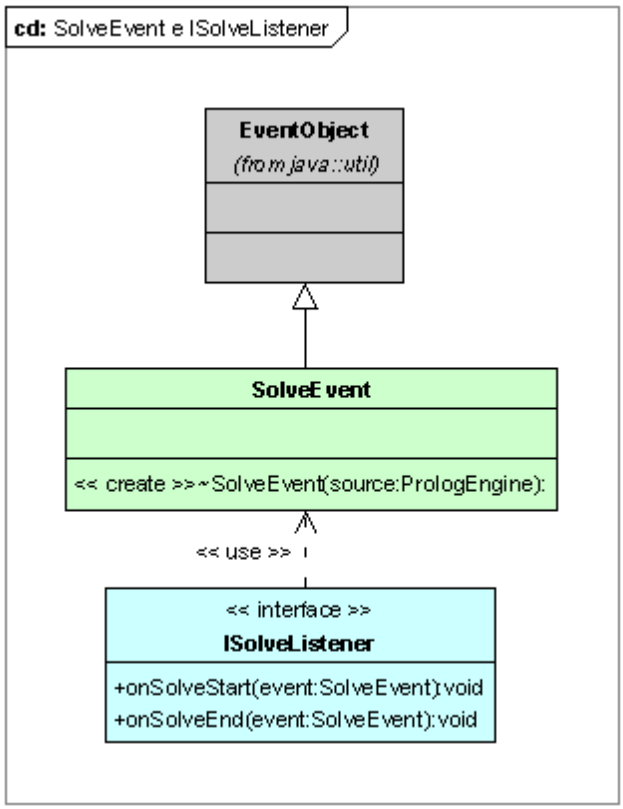


Figura 5.36: Diagramma relativo alla classe SolveEvent e all'interfaccia ISolveListener

momento in cui inizia una risoluzione e cancellarsi nel momento in cui è terminata. Pertanto la classe ActiveTheoryList deve realizzare l'interfaccia `ISolveListener`.

Affinché sia possibile la risoluzione di query temporanee, il TheoryManager deve sapere quando inizia una risoluzione e quando termina, ed in più deve sapere se i cambiamenti effettuati all'elenco delle teorie attive durante una risoluzione appena conclusa sono da ignorare o meno. A tal fine, la classe `TheoryManager` deve essere dotata dei seguenti metodi, richiamati dall'unica istanza della classe `PrologEngine`:

- `public void solveStart()`: richiamato all'inizio della risoluzione di una query;
- `public void solveEnd(boolean confirmChanges)`: richiamato al termine della risoluzione di una query; il parametro booleano deve avere valore *true* quando i cambiamenti all'elenco delle teorie attive devono essere confermati.

Infine, si noti che, rispetto al primo prototipo, la signature del metodo `solveEnd` relativo alla classe `PrologEngine` è stata cambiata; la nuova signature è:

```
public void solveEnd(boolean confirmChanges).
```

Oltre a cambiare la signature, il metodo `solveEnd` è stato cambiato anche a livello di semantica, e pertanto, come spiegato durante la progettazione, anche le classi `StartAction`, `StopAction`, `NextAction` e `AcceptAction` subiscono dei cambiamenti. Infatti, le azioni StartAction e NextAction adesso devono richiamare il metodo `solveEnd` nel caso in cui non siano presenti ulteriori possibili soluzioni, l'azione StopAction adesso lo deve richiamare sistematicamente, mentre l'azione AcceptAction già lo richiamava sistematicamente, ma nel primo prototipo la sua signature era diversa, dato che non specificava se la query era temporanea o meno.

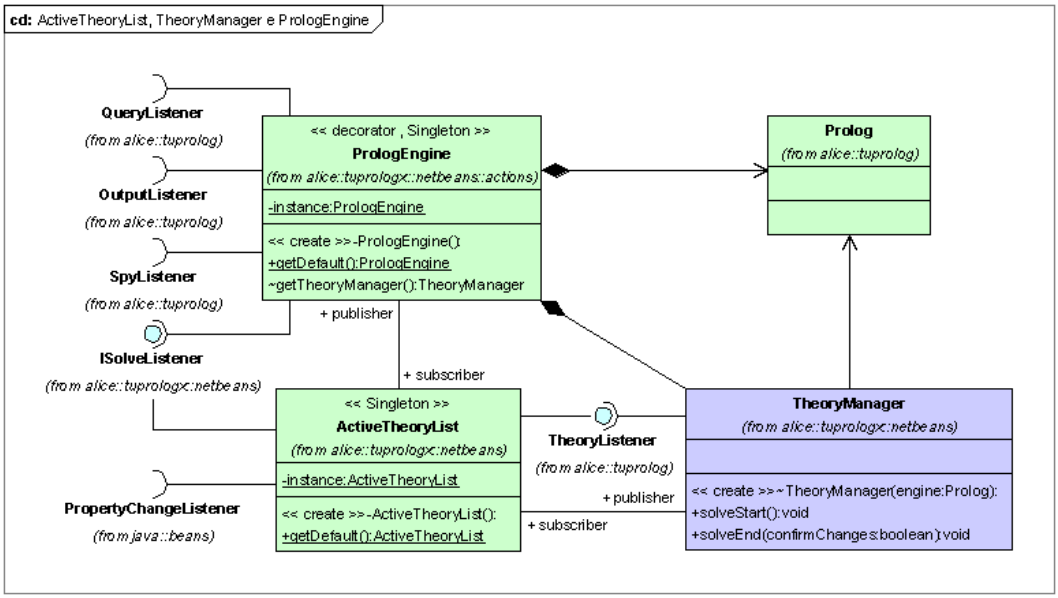


Figura 5.37: Diagramma relativo alle classi ActiveTheoryList, TheoryManager e PrologEngine

Infine, si osservi che il PrologEngine deve essere inizializzato in modo tale che non sia presente la teoria iniziale denominata *default*, dato che il predicato *assert* dovrebbe, una volta richiamato, crearsi automaticamente una teoria *default* in fondo all'elenco delle teorie attive, se questa già non vi è presente.

5.3.3 Implementazione dell'*ActiveTheoryListView*

Il nome della classe che implementa l'entità ActiveTheoryListView è `alice.tuprologx.netbeans.view.ActiveTheoryListView`. In fase di

progettazione è stato stabilito che, al fine di ottenere una migliore integrazione nell'ambiente NetBeans, si deve implementare l'ActiveTheoryListView sotto forma di *Explorer View*; pertanto, i widget che rappresenteranno l'elenco delle teorie attive e le varie teorie in esso contenute, sono dei *nodi*.

Dato che l'elenco delle teorie attive è fondamentalmente una lista, si può pensare di estendere la classe `org.openide.explorer.view.ListView` per implementare la classe `ActiveTheoryListView` sotto forma di *Explorer View*.

Durante la risoluzione di una query non deve essere possibile l'esecuzione di alcuna azione sull'ActiveTheoryListView, conseguentemente, durante tale periodo, i menù contestuali non dovranno comparire. Per raggiungere tale obiettivo, la vista ActiveTheoryListView deve essere registrata presso il PrologEngine in modo tale da ricevere gli eventi di inizio e terminazione della risoluzione di una query, pertanto, la classe `ActiveTheoryListView` deve realizzare l'interfaccia `ISolveListener`. Il metodo `onSolveStart` verrà utilizzato per disattivare la generazione di menù contestuali, mentre il metodo `onSolveEnd` la riattiverà. E' importante tener conto che in un'eventuale futura versione del plug-in in cui l'ActiveTheoryListView supporti il *drag and drop*, anche tale funzionalità dovrà essere disattivata.

L'ActiveTheoryListNode è implementato dalla classe `alice.tuprologx.netbeans.nodes.ActiveTheoryListNode`, ed incapsula l'unica istanza della classe `ActiveTheoryList`. Dato che il modello incapsulato è indicizzato, la classe `ActiveTheoryListNode` estende la classe `org.openide.nodes.IndexedNode`, la quale costituisce l'implementazione di base di un nodo indicizzato. Per quanto riguarda i TheoryNode, essi sono implementati dalla classe `alice.tuprologx.netbeans.nodes.TheoryNode`, ed incapsulano ciascuno un'istanza della classe `Theory`.

Dato che l'elenco delle teorie attive può essere sottoposto a cambiamenti e le teorie invece sono considerate immutabili, allora l'unica istanza della classe `ActiveTheoryList` deve notificare ogni suo cambiamento all'unica istanza della classe incapsulante, ovvero alla `ActiveTheoryListNode`, mentre le istanze della classe `Theory` non presentano mai notifiche all'istanza della classe `TheoryNode` che le incapsula. Quindi la classe `ActiveTheoryListNode` deve realizzare l'interfaccia `PropertyChangeListener`.

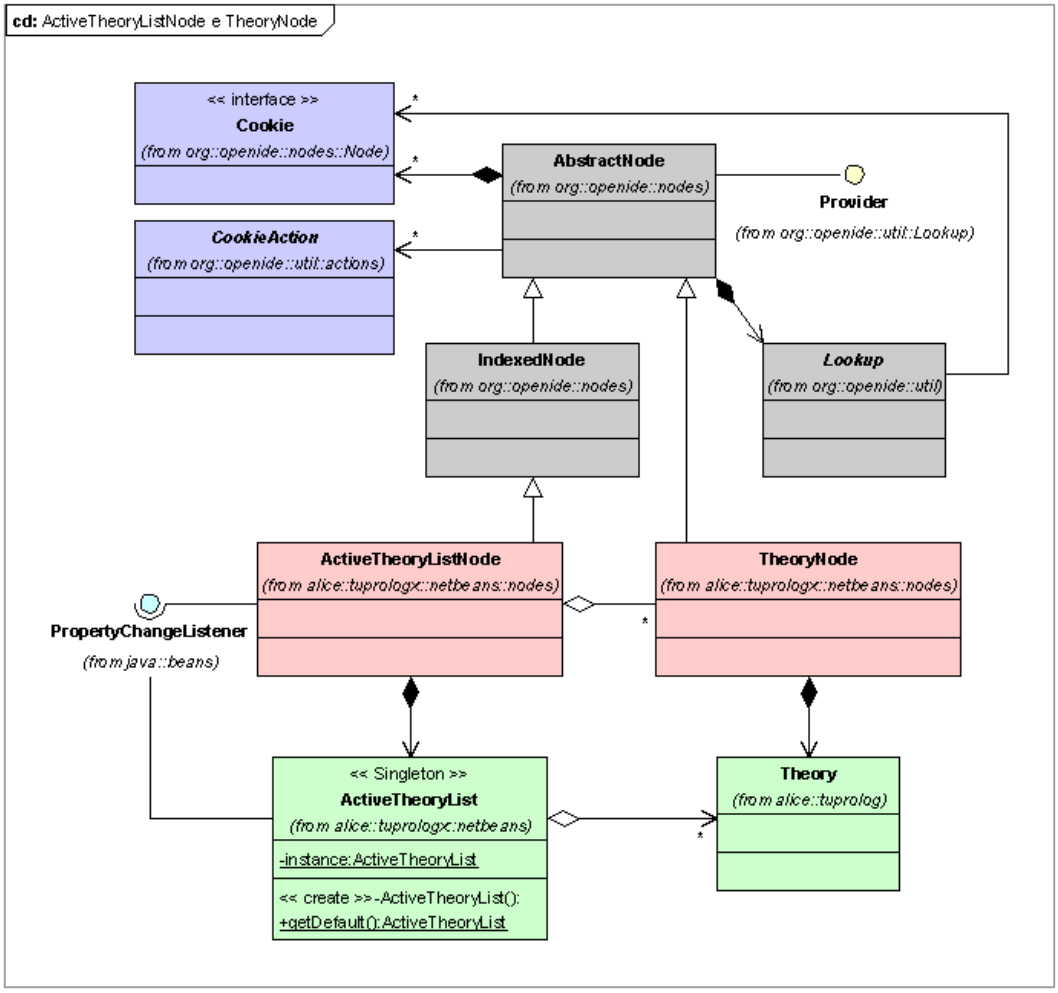


Figura 5.38: Diagramma relativo alle classi ActiveTheoryListNode e TheoryNode

Dato che l'ActiveTheoryListView è di fatto un explorer view, qualcuno dei componenti grafici che lo contengono deve svolgere il ruolo di explorer manager, oppure lui stesso deve svolgere tale ruolo. Il candidato ideale è indubbiamente il QueryControlPanel, dato che rappresenta il pannello che permette il controllo del motore tuProlog, e l'elenco delle teorie attive, rappresentato dall'ActiveTheoryListView, è un'entità strettamente legata con il motore tuProlog. Pertanto, la classe `QueryControlPanel` è stata modificata, facendole realizzare anche le interfacce `org.openide.explorer.ExplorerManager.Provider` e `org.openide.util.Lookup.Provider`, in modo tale che possa restituire sia un'istanza della classe `org.openide.explorer.ExplorerManager`, sia un'istanza della classe `org.openide.util.Lookup`. In questo modo è possibile aggiungere dei Cookie ai vari nodi visualizzati dall'ActiveTheoryListView.

Questo ultimo aspetto, però, viola le direttive del design pattern *Façade*, dato che le istanze delle classi `ExplorerManager` e `Lookup` sono parte del QueryControlPanel; comunque, in linea di massima, la classe nasconde la sua complessità interna, pertanto anche se non realizza il design pattern *Façade*, ne assume comunque il ruolo.

Per evitare i problemi legati all'esplorazione di una gerarchia di componenti grafici (vedi sezione 2.3.4), alla ricerca di un'istanza che realizza l'interfaccia `ExplorerManager.Provider`, anche all'ActiveTheoryListView è stata fatta realizza questa interfaccia, in modo tale che si assuma la responsabilità di restituire un riferimento ad un explorer manager. In realtà non ha a disposizione alcun riferimento ad istanze della classe `ExplorerManager`, ma delega la ricerca di una di queste al QueryControlPanel, richiamando il relativo metodo `ExplorerManager getExplorerManager();` il riferimento al QueryControlPanel viene passato come parametro di ingresso al costruttore della classe `ActiveTheoryList`. Per evitare uno stretto accoppiamento tra la classe `ActiveTheoryListView` e la classe `QueryControlPanel`, la signature del costruttore dell'ActiveTheoryList specifica come parametro un'istanza dell'interfaccia `ExplorerManager.Provider`. Con questo modo di procedere, praticamente si applica il design pattern *Chain of Responsibility* [GoF95], anche se la relativa gerarchia di oggetti è composta soltanto da due elementi.

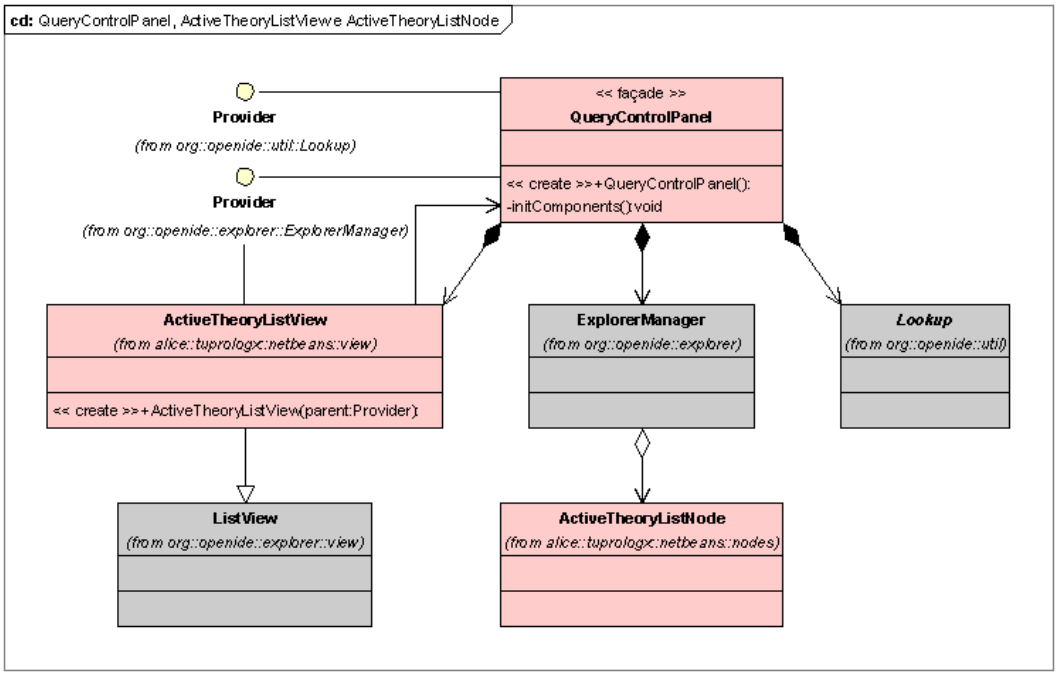


Figura 5.39: Diagramma relativo alle classi QueryControlPanel, ActiveTheoryListView e ActiveTheoryListNode

Nel caso in cui, in un successivo prototipo, si modificasse l'organizzazione dei vari explorer view presenti nell'interfaccia grafica, si dovrà mantenere una gerarchia di istanze che realizzino l'interfaccia `ExplorerManager.Provider`, in modo tale che si possa continuare ad utilizzare il design pattern *Chain of Responsibility* per la ricerca di un'istanza della classe `ExplorerManager`. Pertanto, tutti gli oggetti che delegheranno la ricerca di un explorer manager ad un'istanza di livello superiore, dovranno realizzare l'interfaccia `ExplorerManager.Provider` ed aver un costruttore che accetti un'istanza di tale interfaccia, in modo tale da poter specificare in fase di inizializzazione l'istanza di livello superiore relativamente alla gerarchia di ricerca di un explorer manager.

I vantaggi nell'uso di tale design pattern sono sostanzialmente due: innanzitutto si definisce un percorso sicuro che permette di individuare ciò che si cerca (in contrapposizione con la soluzione proposta dai progettisti dell'ambiente NetBeans, in cui possono essere presenti gerarchie di oggetti interrotte) e, infine, permette di avere classi disaccoppiate, dato che la gerarchia di oggetti che definisce il percorso di ricerca di istanze della classe `ExplorerManager` è logicamente costituita soltanto da istanze che realizzano l'interfaccia `ExplorerManager.Provider`, e pertanto tutti i riferimenti saranno di questo tipo.

Per quanto riguarda la visualizzazione dei *menù contestuali relativi ad una specifica teoria*, essi devono comparire cliccando sul nodo che rappresenta la teoria su cui si vuole eseguire un'azione. Per quanto riguarda, invece, la visualizzazione del *menù contestuale relativo alla vista delle teorie attive*, esso deve comparire cliccando su uno spazio vuoto all'interno dell'ActiveTheoryListView. Nel primo caso, il supporto per la visualizzazione del menù contestuale è offerto dalla classe `ListView`, quindi ereditato dalla classe `ActiveTheoryListView`; pertanto, le azioni eseguibili sui nodi saranno elencabili richiamando il metodo `Action[] getActions(boolean)`. Nel secondo caso, non si ha alcun supporto per la visualizzazione del menù contestuale; pertanto si deve provvedere appositamente aggiungendo un listener relativo agli eventi di interazione con il mouse, il quale, rilevata la posizione di click su di una zona vuota, provvederà a visualizzare il menù contestuale relativo alla vista delle teorie attive. Pertanto, deve essere aggiunta un'apposita inner-class che estenda la classe `java.awt.event.MouseAdapter`, o, per lo meno, che

realizzi l'interfaccia `java.awt.event.MouseListener`.

Purtroppo, però, non sempre sarà disponibile uno spazio vuoto all'interno dell'ActiveTheoryListView che permetta di richiamare il menù contestale relativo all'elenco delle teorie attive; pertanto, i menù contestuali relativi ad una specifica teoria devono contenere anche le voci presenti nel menù contestuale relativo all'elenco delle teorie attive.

5.3.4 Implementazione delle azioni relative alla gestione delle teorie attive

Come accennato durante la progettazione, le azioni relative alla gestione delle teorie attive devono implementare delle CookieAction, pertanto estendono la classe `org.openide.util.actions.CookieAction`; affinché tali azioni possano svolgere il loro compito quando richiamate, i nodi selezionati devono incapsulare appositi *cookie*. Di seguito, sono elencate le classi delle azioni relative alla gestione delle teorie attive e le interfacce dei cookie richiesti:

- azione: `alice.tuprologx.netbeans.actions.AddTheoryAction`;
  cookie: `alice.tuprologx.netbeans.nodes.IAddTheoryCookie`;
- azione: `alice.tuprologx.netbeans.actions.GetTheoryAction`;
  cookie: `alice.tuprologx.netbeans.nodes.IGetTheoryCookie`;
- azione: `alice.tuprologx.netbeans.actions.GetWholeTheoryAction`;
  cookie: `alice.tuprologx.netbeans.nodes.IGetWholeTheoryCookie`;
- azione: `alice.tuprologx.netbeans.actions.RemoveAllTheoryAction`
  cookie: `alice.tuprologx.netbeans.nodes.IRemoveAllCookie`;
- azione: `alice.tuprologx.netbeans.actions.RemoveTheoryAction`;
  cookie: `alice.tuprologx.netbeans.nodes.IRemoveCookie`;
- azione: `alice.tuprologx.netbeans.actions.SetTheoryAction`;
  cookie: `alice.tuprologx.netbeans.nodes.ISetAsUniqueCookie`.

Le interfacce che specificano il comportamento dei *cookie* estendono l'interfaccia `org.openide.nodes.Node.Cookie`; ciascun *cookie* è implementato in base al contesto in cui deve operare, pertanto sono implementati come inner-class dei relativi nodi. Tali classi non sono visibili esternamente, conseguentemente sono incapsulate all'interno delle classi `ActiveTheoryListNode` e `TheoryNode`. Nonostante le classi dei *cookie* non siano esternamente visibili, è possibile

accedere alle loro istanze tramite il metodo `Node.Cookie getCookie(Class)`, ovvero sfruttando il Lookup del nodo considerato.

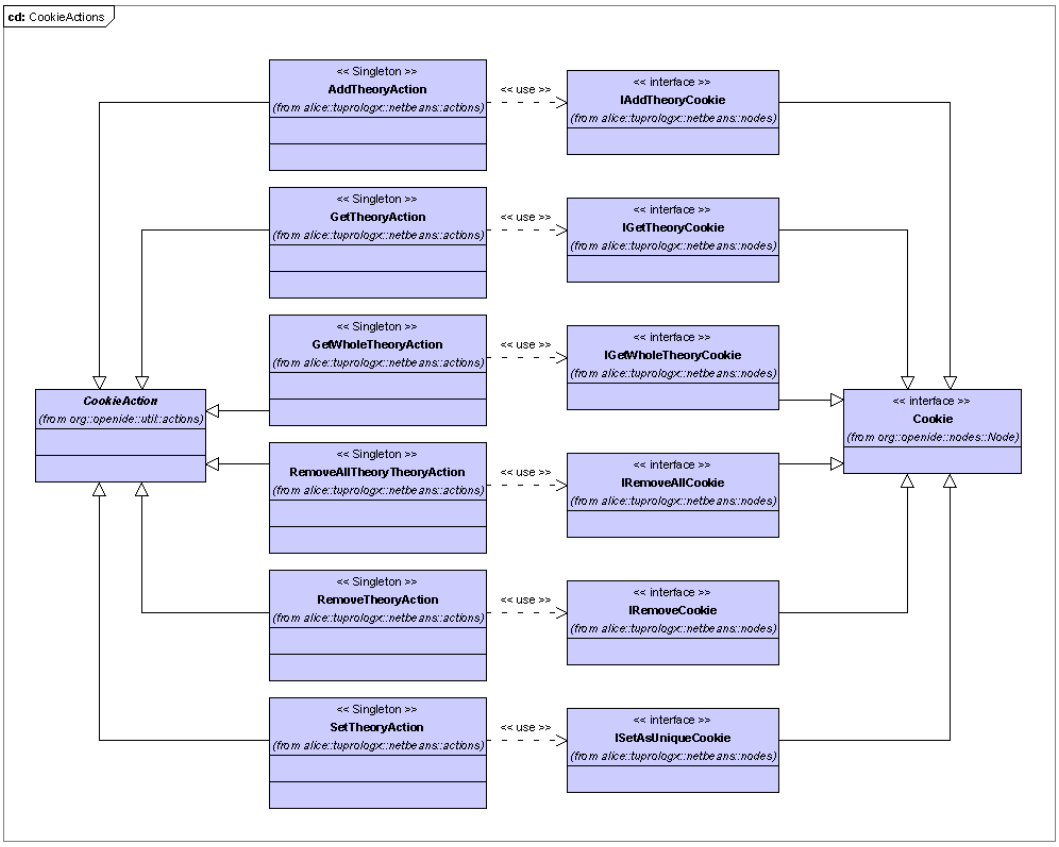


Figura 5.40: Diagramma delle classi relative alle azioni per la gestione delle teorie attive

Ogni azione relativa alla gestione delle teorie attive deve realizzare il design pattern *Singleton*, pertanto tutte le classi che le implementano sono dotate di un costruttore a visibilità `private` e del metodo di classe `getDefault()` per poterne recuperare l'unica istanza.

5.3.5 Implementazione del *LocalTheoryLoader* e del *LocalTheorySaver*

Il LocalTheoryLoader e il LocalTheorySaver sono implementati rispettivamente dalle classi `alice.tuprologx.netbeans.loaders.LocalTheoryLoader` e `alice.tuprologx.netbeans.loaders.LocalTheorySaver`. Queste due classi sono accomunate dall'utilizzo dei partecipanti FileChooser e LocalFileSystem, i quali sono implementati dalle classi `LocalTheoryChooser` e `LocalFileSystem`.

Il LocalTheoryLoader permette di ottenere un data object relativo ad una teoria logica richiamando il metodo `DataObject getDataObject()`. Le operazioni

svolte da tale metodo consistono in:

- visualizzazione della finestra di dialogo per la scelta del file contenente la teoria da caricare, utilizzando il FileChooser;
- una volta che l'utente ha scelto il file relativo alla teoria da caricare, tenterà di ottenere un file object tramite il LocalFileSystem;
- ottenuto il file object, tenterà di ottenere un data object richiamando il metodo di classe `DataObject find(FileObject)` relativo alla classe `DataObject`.

Il metodo restituisce un'istanza della classe `DataObject` se e solo se le operazioni appena elencate saranno tutte svolte con successo e se il data object recuperato sarà un'istanza della classe `TheoryDataObject`, la quale implementa i data object relativi a teorie logiche.

Per poter ottenere uno specifico data object richiamando il metodo di classe `DataObject find(FileObject)` relativo alla classe `DataObject`, deve essere preventivamente inserito nel *System FileSystem* un riferimento ad un apposito data loader; il data loader relativo alle teorie logiche sarà implementato dalla classe `alice.tuprologx.netbeans.loaders.TheoryDataLoader`. Ogni data loader è un *JavaBean*, pertanto per una corretta integrazione nell'ambiente NetBeans la classe `TheoryDataLoader` deve essere corredata di un descrittore di bean. Questo descrittore di bean è implementato dalla classe `alice.tuprologx.netbeans.loaders.TheoryDataLoaderBeanInfo` ed è utilizzato per specificare le informazioni addizionali e l'icona relative al bean a cui è associato.

Il LocalTheorySaver permette di memorizzare nel file system locale una teoria richiamando il metodo `void putTheory(Theory)`.

Sia la classe `LocalTheoryLoader` che la classe `LocalTheorySaver` realizzano il design pattern *Singleton*, pertanto presentano costruttori a visibilità *private* e un metodo di classe per poter ottenere l'unica istanza.

Il LocalTheoryLoader è utilizzato dalla classe `AddTheoryAction`, mentre il LocalTheorySaver è utilizzato dalla classe `ExportTheoryAction`; pertanto queste due classi sono rispettivamente dipendenti dalle classi `LocalTheoryLoader` e `LocalTheorySaver`.

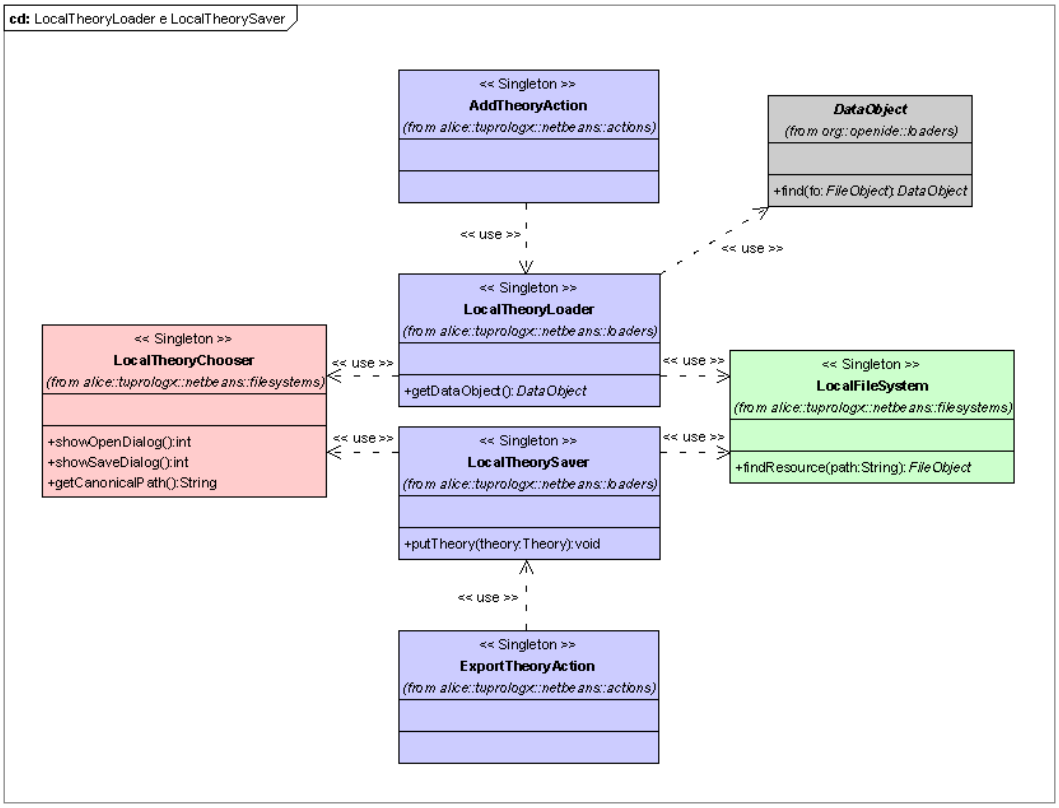


Figura 5.41: Diagramma relativo alle classi LocalTheoryLoader e LocalTheorySaver

5.3.6 Implementazione dell'*ExportTheoryAction*

L'azione ExportTheoryAction è particolare, perché è richiamata dalle azioni GetTheoryAction e GetWholeTheoryAction, le quali forniscono la teoria da esportare prima di eseguire l'azione. La ExportTheoryAction è implementata dalla classe `alice.tuprologx.netbeans.ExportTheoryAction`, pertanto le classi `GetTheoryAction` e `GetWholeTheoryAction` risultano essere dipendenti dalla classe `ExportTheoryAction`. Dato che tale azione è richiamata a sua volta da altre azioni e non è associata a dei *nodi*, essa è implementata estendendo la classe astratta `javax.swing.AbstractAction`. A sua volta, l'azione ExportTheoryAction utilizza il LocalTheorySaver per poter esportare la teoria, richiamando il metodo `void putTheory(Theory)`.

Dato che ogni azione è unica, anche la classe `ExportTheoryAction` realizza il design pattern *Singleton*, pertanto presenta un costruttore a visibilità *private* e un metodo di classe per poter ottenere l'unica istanza.

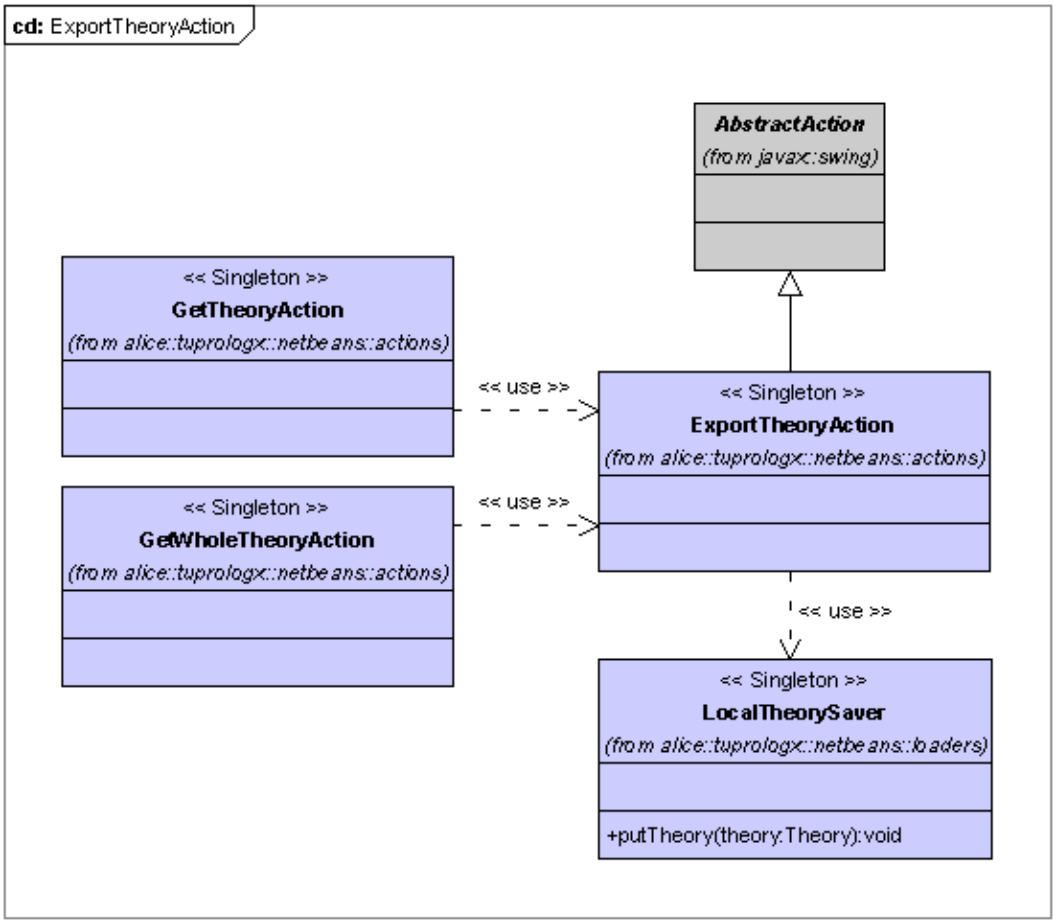


Figura 5.42: Diagramma relativo alla classe ExportTheoryAction

5.3.7 Implementazione dell'*AddTheoryAction*

Un approfondimento lo merita anche l'azione AddTheoryAction, la quale utilizza il LocalTheoryLoader al fine di acquisire un data object relativo ad una teoria. Lo scopo dell'AddTheoryAction è quello di aggiungere una nuova teoria all'elenco delle teorie attive, pertanto dal data object deve ottenere la teoria in esso incapsulata. Affinché ciò sia possibile, ad ogni TheoryDataObject deve essere associato un apposito cookie che consenta di ottenere la teoria incapsulata; tale cookie dovrà realizzare l'interfaccia `IGetTheoryCookie`. Questo verrà utilizzato dalla classe `AddTheoryAction` per ottenere la teoria da aggiungere all'elenco delle teorie attive.

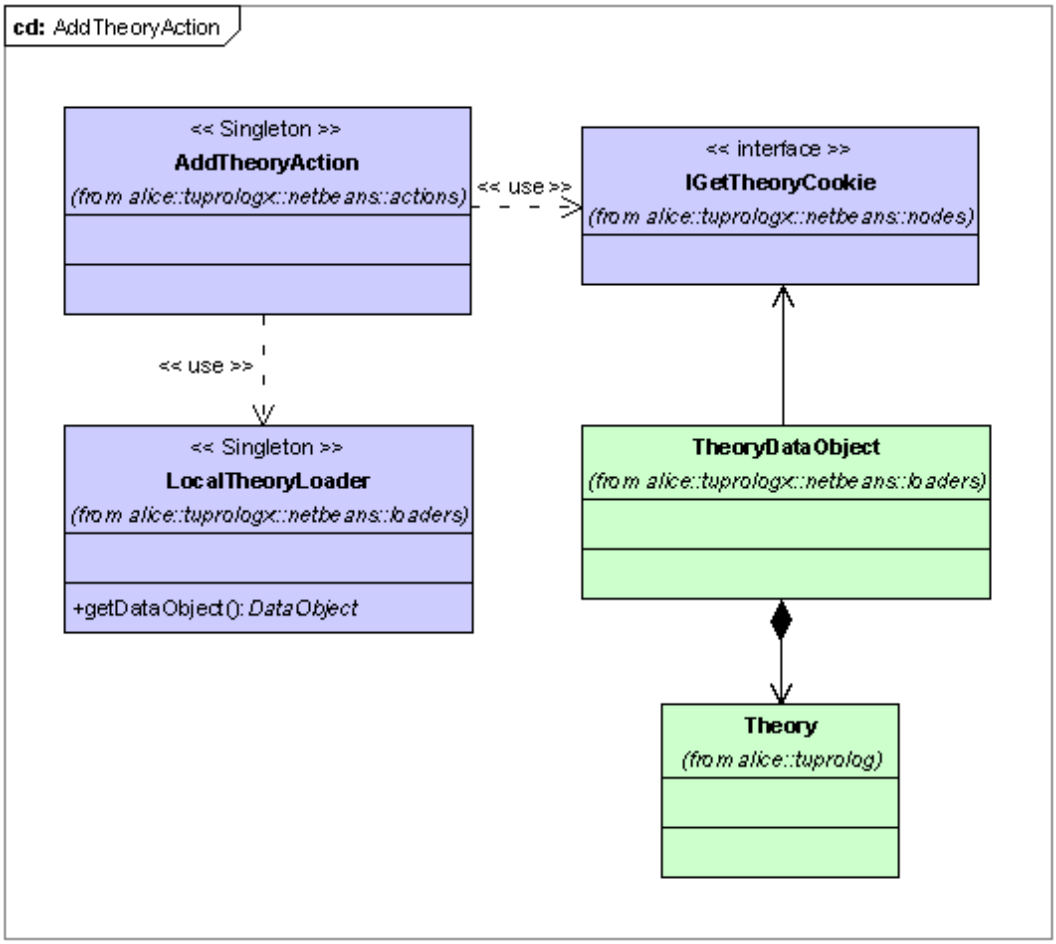


Figura 5.43: Diagramma relativo alla classe AddTheoryAction

5.3.8 Implementazione del *LocalTheoryChooser*

Per effettuare la scelta dei file da utilizzare per caricare o salvare delle teorie, si è deciso di utilizzare una apposita classe, anziché utilizzare la classe predefinita `javax.swing.JFileChooser`. Comunque, la classe `alice.tuprologx.netbeans.filesystems.LocalTheoryChooser` estende la classe `JFileChooser`, in modo tale da avere una finestra di dialogo per la scelta dei file maggiormente integrata con l'ambiente, che, inoltre, permetta la visualizzazione esclusiva dei file dotati delle estensioni predefinite per teorie logiche Prolog, ovvero *.pl* e *.pro*; per la visualizzazione esclusiva di questi file è stato aggiunto un filtro al FileChooser, implementato dalla classe `alice.tuprologx.netbeans.filesystems.TheoryFileFilter`.

La classe `LocalTheoryChooser` realizza il design pattern *Singleton*, pertanto presenta un costruttore a visibilità *private* e un metodo di classe per poter ottenere l'unica istanza.

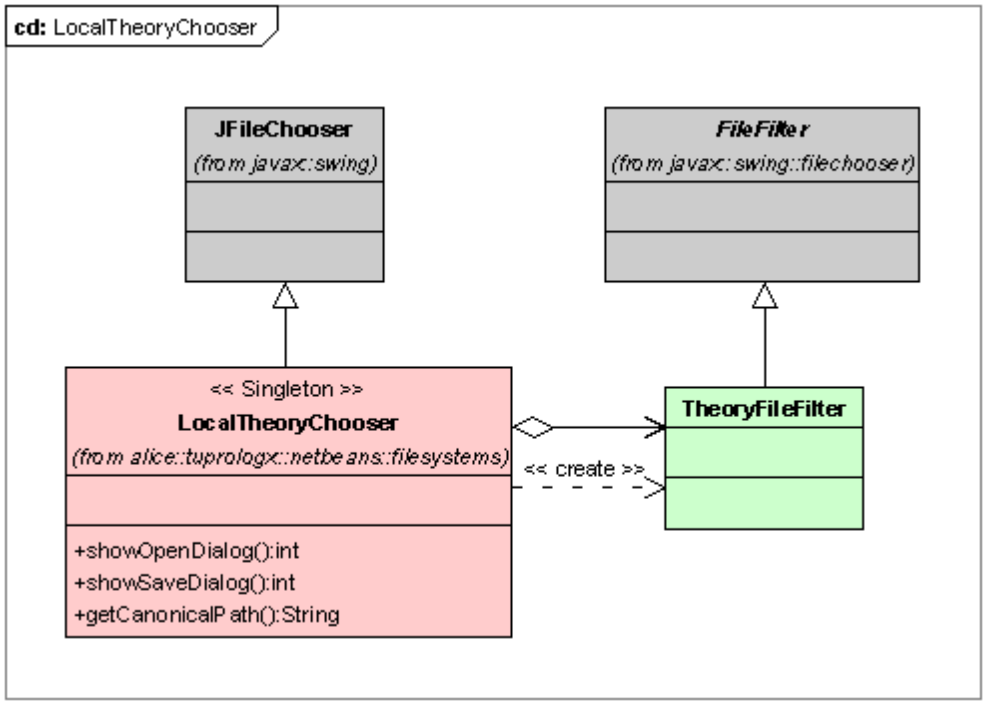


Figura 5.44: Diagramma relativo alla classe LocalTheoryChooser

5.3.9 Implementazione del *LocalFileSystem*

Per poter accedere al file system locale si potrebbe utilizzare la classe `org.openide.filesystems.LocalFileSystem`, ma questa presenta seri problemi quando vengono specificati percorsi canonici[2] relativi a file in ambiente Microsoft Windows: infatti la ricerca di di file, tramite il metodo `FileObject findResource(String)`, è sempre infruttuosa quando sono specificati percorsi ove sono presenti identificatori di drive in stile Windows. Per risolvere tale incompatibilità, la classe `alice.tuprologx.netbeans.filesystems.LocalFileSystem` estende la classe `org.openide.filesystems.LocalFileSystem` in modo tale da aggiungere un metodo per la ricerca di file all'interno del file system locale che supporti correttamente anche l'ambiente Windows.

La classe `LocalFileSystem` realizza il design pattern *Singleton*, pertanto presenta un costruttore a visibilità *private* e un metodo di classe per poter ottenere l'unica istanza.

2 Con percorso canonico si intende un percorso assoluto e unico, quindi privo dei nomi "." e "..".

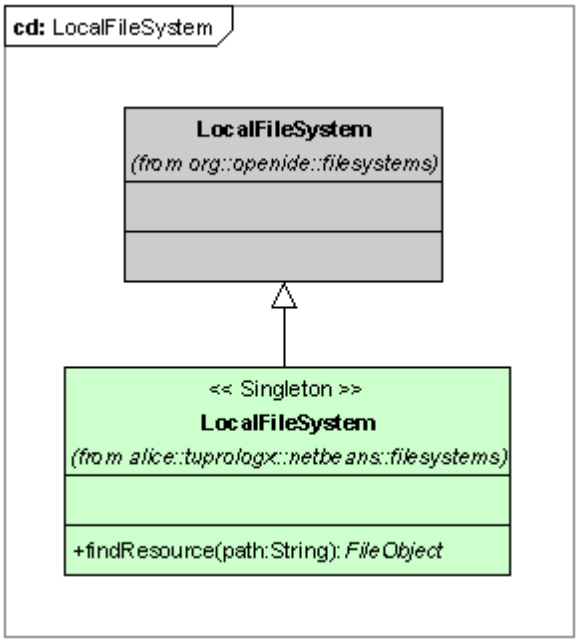


Figura 5.45: Diagramma relativo alla classe LocalFileSystem

5.4 Definizione del tipo di file denominato "Prolog Theory"

Per definire nell'ambiente NetBeans il tipo di file relativo alle teorie logiche Prolog, al quale è associato il nome "*Prolog Theory*" e le estensioni *.pl* e *.pro*, sono stati creati:

- il documento *TheoryResolver.xml*, contenente la dichiarazione dell'estensione e l'identificatore MIME del tipo di contenuto, che è "*text/x-prolog*"[3]; tale documento è referenziato all'interno dello *XML Layer*, pertanto deve essere incluso all'interno del package `alice.tuprologx.netbeans`;
- la classe `TheoryDataObject`, in grado di rappresentare ed elaborare il contenuto di un FileObject contenente una teoria logica Prolog;
- la classe `TheoryDataLoader`, in grado di caricare i file relativi ad una teoria logica Prolog e di creare la relativa istanza della classe `TheoryDataObject`;

3 Un identificatore MIME del tipo di contenuto [RFC2045] è composto da due parti separate dal carattere '/': l'identificatore del tipo principale di contenuto, il quale specifica il tipo generale dei dati, e l'identificatore del sottotipo di contenuto, il quale specifica il particolare formato dei dati. In questo caso si ha a che fare con teorie logiche Prolog, ovvero documenti testuali, pertanto l'identificatore del tipo principale di contenuto deve essere "text", mentre per l'identificatore del sottotipo di contenuto è stato scelto "x-prolog", dove il prefisso "x-" sta ad indicare che non si tratta di un tipo di contenuto definito in qualche RFC.

- la classe `TheoryDataBeanInfo`, ovvero il descrittore del TheoryDataLoader; definisce le informazioni utili all'introspezione del TheoryDataLoader e l'icona associata ai file contenenti teorie logiche Prolog;
- la classe `TheoryDataNode`, la quale specifica la presentazione dei file relativi alle teorie logiche Prolog.

Il vantaggio principale relativo alla definizione di questo nuovo tipo di file all'interno dell'ambiente riguarda la disponibilità delle funzionalità di base per la gestione dei file, come il taglia/copia/incolla, la possibilità di rinominare un file, ecc., ma non l'apertura dei file relativi a questo tipo, dato che non è stato associato alcun editor al nuovo tipo di file; tali funzionalità di base devono essere indicate nello *XML Layer*, associandole all'identificatore MIME del tipo di contenuto relativo al tipo di file introdotto.

5.5 Collaudo

Durante lo sviluppo di questo prototipo sono state spostate le funzionalità di gestione delle teorie attive introdotte nel primo prototipo dal PrologEngine al TheoryManager, oltre a modificare il comportamento e la signature dei metodi relativi alla gestione della risoluzione di una query; pertanto si deve adeguare il test unitario relativo alla classe `PrologEngine`, sviluppato per testare il primo prototipo. Inoltre, si deve aggiungere altri due test unitari, relativi al TheoryManager e all'ActiveTheoryList, dato che anche questi sono componenti fondamentali del plug-in, alla pari del PrologEngine.

Le modifiche apportate al test unitario del PrologEngine riguardano:

- adeguamento dei test relativi ai metodi `SolveInfo solve(String)`, `void solveHalt()`, `SolveInfo solveNext()` e `void solveEnd(boolean)`;
- rimozione del test relativo al metodo `void removeAllTheories()`;
- aggiunta di un test relativo al predicato *assert*;
- aggiunta di un test relativo al predicato *retract*.

Le verifiche effettuate tramite il test unitario della classe `TheoryManager` riguardano:

- la gestione delle teorie attive;
- il funzionamento dei metodi `void solveStart()` e `void solveEnd(boolean);`
- la gestione dei theory listener.

Le verifiche effettuate tramite il test unitario della classe `ActiveTheoryList` riguardano:

- il funzionamento dei metodi `Theory getTheory(int)` e `void setTheory(int,Theory);`
- il funzionamento dei metodi `Theory[] getTheories()` e `void setTheories(Theory[]);`
- il funzionamento dei metodi `void addTheory(Theory)` e `void removeTheory(int);`
- il funzionamento del metodo `void clear();`
- la gestione dei property change listener;
- il meccanismo di notifica dei property change event.

I risultati dei test sono stati generalmente positivi, tranne per i seguenti casi:

- *gestione degli spy listener*: il metodo `Collection<SpyListener> getSpyListeners()` della classe `PrologEngine` presenta ancora il comportamento anomalo riscontrato durante il collaudo del primo prototipo, dato che la risoluzione di tale problema non fu eseguita;
- *rimozione di una teoria*: verificando il meccanismo di notifica dei property change event relativo alla classe `ActiveTheoryList`, si è notato che il predicato *remove_theory* non notifica il cambiamento che applica;
- *predicato assert*: partendo dal test unitario fallito e indagando ulteriormente, si è notato che la *assert* non inserisce la nuova clausola nell'ultima teoria denominata default o in una nuova teoria, sempre con nome *default*, ma in una specie di "limbo", in modo tale da essere recuperabile solamente tramite il metodo `Theory getWholeTheory();`
- *predicato retract*: partendo dal test unitario fallito e indagando ulteriormente, si è notato che la *retract* non elimina la clausola indicata

correttamente; infatti, analogamente al predicato *assert*, elimina la clausola solo nel "limbo" e non all'interno delle teorie, anche se alla fine il suo effetto è proprio quello di impedire ulteriori valutazioni della clausola che specifica; inoltre, non notifica il cambiamento che apporta.

I problemi riscontrati sono riconducibili a difetti insiti nel motore tuProlog incapsulato dal PrologEngine, quindi ad un elemento esterno a questo plug-in; pertanto, per poterli risolvere, si dovrà procedere a correggere i difetti presentati dal motore tuProlog, modificando la classe `Prolog`.

Infine, è stato ritenuto opportuno testare anche la classe `alice.tuprologx.netbeans.filesystems.LocalFileSystem`, la quale dovrebbe risolvere i problemi della classe `org.openide.filesystems.LocalFileSystem` riscontrati con i percorsi assoluti relativi a file in ambiente Microsoft Windows. L'unico test a cui è stata sottoposta è risultato positivo.

5.6 Modifiche al motore tuProlog

Visto che il numero di test falliti non è trascurabile e vista l'entità dei test falliti, si è deciso di apportare delle correzioni al motore tuProlog; tali correzioni non devono apportare modifiche architetturali all'attuale motore tuProlog, né modificare le signature dei metodi, in modo tale da garantire la compatibilità della versione che verrà generata verso i sistemi software preesistenti basati su tuProlog.

Riepilogando, l'attuale versione di tuProlog dotata della gestione delle teorie multiple è la 1.4; la versione che verrà generata sarà denominata 1.4.1 e non 1.5, dato che, come già detto, l'architettura del motore non verrà modificata, né estesa.

Le correzioni che sono state apportate consistono in:

- *aggiornamento della versione del motore tuProlog*: modifica del campo di classe `private static final String VERSION` della classe `alice.tuprolog.Prolog`, cambiando il suo valore da 1.4 a 1.4.1;
- *correzione del bug relativo alla gestione degli spy listener*: il metodo `List getSpyListenerList()` della classe `alice.tuprolog.Prolog`, restituiva la lista dei listener relativi agli eventi di output e non quella dei listener relativi agli eventi interni del motore tuProlog; effettuata la correzione, restituisce correttamente una lista contenente gli spy listener

registrati presso il motore tuProlog; conseguentemente il metodo `Collection<SpyListener> getSpyListeners()` della classe `PrologEngine` non presenta più un comportamento anomalo;

- *trasformazione dei metodi non thread-safe in metodi thread-safe*: alcuni metodi della classe `alice.tuprolog.Prolog`, relativi alla gestione delle teorie multiple, non erano thread-safe; alle signature di tali metodi è stata semplicemente aggiunta la clausola `synchronized`, mantenendo la compatibilità di tali metodi verso i sistemi software preesistenti basati su tuProlog;
- *aggiunta della notifica relativa alla rimozione di una teoria*: il metodo `void removeTheory(String)` della classe `alice.tuprolog.Prolog` non notificava il cambiamento che apporta; effettuata la correzione, ogni volta che tale metodo viene richiamato si ha una opportuna notifica; conseguentemente, quando viene utilizzato il predicato *remove_theory*, l'elenco delle teorie attive viene opportunamente aggiornato;
- *aggiunta della notifica relativa alla valutazione del predicato assert*: il predicato *assert* non notificava alcun evento relativo al cambiamento della teoria dinamica caricata nel motore tuProlog; effettuata la correzione, ogni volta che tale predicato viene invocato, viene lanciato un evento che induce ad aggiornare l'elenco delle teorie attive;
- *aggiunta della notifica relativa alla valutazione del predicato retract*: il predicato *retract* non notificava alcun evento relativo al cambiamento della teoria dinamica caricata nel motore tuProlog; effettuata la correzione, ogni volta che tale predicato viene invocato, viene lanciato un evento che induce ad aggiornare l'elenco delle teorie attive, inoltre nel caso in cui l'invocazione di tale predicato venisse ritrattata, eseguendo un'operazione di backtracking, viene lanciato un ulteriore evento che induce ad aggiornare nuovamente l'elenco delle teorie attive.

Purtroppo non si è potuto procedere alla correzione del comportamento anomalo dei predicati *assert* e *retract*, i quali continuano ad operare all'interno di un "limbo" di teorie e non nell'insieme delle teorie caricate del motore tuProlog; infatti, una eventuale correzione di tale comportamento anomalo avrebbe

comportato un consistente cambiamento di alcuni meccanismi interni del motore tuProlog e, dato che è stato precedentemente deciso di non apportare cambiamenti drastici al fine di salvaguardare la compatibilità di tuProlog verso i sistemi software preesistenti, si è rinunciato a procedere.

Capitolo 6

Ulteriori prototipi e sviluppi futuri

In questo capitolo, partendo dal lavoro effettuato per la realizzazione del secondo prototipo, verranno sviluppati i progetti relativi agli ulteriori prototipi del plug-in di tuProlog per l'ambiente NetBeans, continuando ad avere come riferimento le considerazioni effettuate durante l'analisi del problema e su quanto specificato nel piano di lavoro; dopodiché si procederà ad analizzare gli eventuali sviluppi futuri e le questioni rimaste ancora aperte.

6.1 Progettazione del terzo prototipo

La progettazione del terzo prototipo si baserà su ciò che suggerisce di realizzare il piano di lavoro:

Terzo Prototipo: *integrazione della gestione delle teorie disponibili nel plug-in; le teorie attive non saranno importate/esportate tramite il filesystem locale, ma tramite l'elenco delle teorie disponibili; le funzionalità implementate saranno tutte quelle stabilite nell'analisi del problema escludendo la separazione di una teoria; conseguentemente, dovrà essere possibile l'esecuzione di modifiche alle teorie disponibili, e quindi tale prototipo dovrà integrare un editor con funzionalità minime, ma non sarà ancora possibile eseguire modifiche al contenuto delle teorie attive*.

Osservando la figura 3.1, relativa al diagramma dei casi d'uso dal titolo "Gestione della risoluzione di query e gestione delle teorie attive", si deduce che alcuni dei casi d'uso da realizzare per fare ciò che suggerisce il piano di lavoro sono:

- *importazione teoria dall'elenco delle teorie disponibili*;

- *esportazione teoria verso l'elenco delle teorie disponibili*;
- *estrai una o più teorie attive*;
- *estrai la teoria attiva globale*;
- *estrai tutte le teorie*.

Nel secondo prototipo tutti questi casi d'uso sono già stati realizzati, tranne quello denominato "*estrai tutte le teorie*", ma senza assumere una forma definitiva: infatti nel precedente prototipo essi sono stati sviluppati giusto per ottenere una versione del plug-in in grado di caricare/salvare le teorie logiche. Pertanto tali realizzazioni saranno sostituite da quelle che verranno definite di seguito.

Osservando la figura 3.2, relativa al diagramma dei casi d'uso dal titolo "Gestione delle teorie disponibili", si deduce che i rimanenti casi d'uso da realizzare per fare ciò che suggerisce il piano di lavoro sono:

- *modifica elenco delle teorie disponibili*;
- *importazione teoria*;
- *esportazione teoria*;
- *apri teoria da file*;
- *salva teoria su file*;
- *aggiorna teoria*;
- *aggiorna tutte le teorie*;
- *crea teoria*;
- *cancella teoria*;
- *modifica contenuto teoria disponibile*.

6.1.1 Collaborazione: *importazione teoria dall'elenco delle teorie disponibili*

Con riferimento all'analisi del problema, per poter realizzare il caso d'uso "*importazione teoria dall'elenco delle teorie disponibili*" occorrono:

- un modello relativo all'elenco delle teorie attive;
- un modello relativo all'elenco delle teorie disponibili;
- un componente grafico in grado di rappresentare l'elenco delle teorie

attive;

- un componente grafico tale da offrire le funzionalità di gestione delle teorie disponibili.

Inoltre, servirà anche un elemento di controllo in grado di eseguire l'azione di importazione della teoria specificata dall'utente nell'elenco delle teorie attive.

I partecipanti sono:

- *ActiveTheoryList*: elenco delle teorie attive;
- *AvailableTheoryList*: elenco delle teorie disponibili;
- *ActiveTheoryListView*: vista che rappresenta l'elenco delle teorie attive;
- *AvailableTheoryListView*: vista che consente di gestire l'elenco delle teorie disponibili;
- *ImportActiveTheoryAction*: azione che preleva una teoria dall'elenco delle teorie disponibili e la aggiunge all'elenco delle teorie attive.

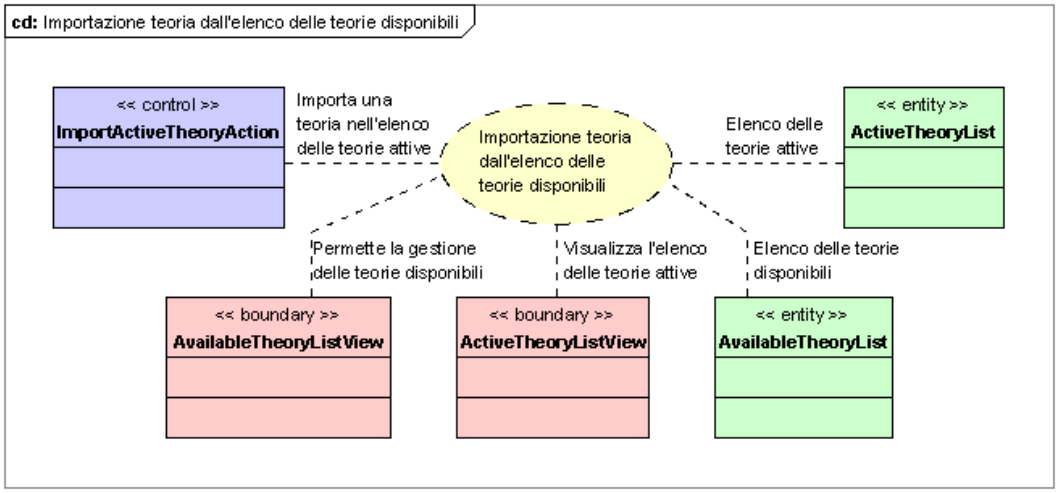


Figura 6.1: Collaborazione relativa al caso d'uso "importazione teoria dall'elenco delle teorie disponibili"

Dai ruoli definiti per ogni partecipante, si determinano le relative responsabilità:

- *ActiveTheoryList*: offre il servizio di aggiunta della teoria da importare all'elenco delle teorie attive;
- *AvailableTheoryList*: fornisce la teoria da importare nell'elenco delle teorie attive;
- *ActiveTheoryListView*: riceve le notifiche relative al cambiamento del

contenuto dell'elenco delle teorie attive e ne visualizza l'effetto;

- *AvailableTheoryListView*: richiama l'azione ImportActiveTheoryAction allo scopo di importare una teoria nell'elenco delle teorie attive, su richiesta dell'utente;
- *ImportActiveTheoryAction*:
  - estrae la teoria indicata dall'AvailableTheoryListView;
  - richiama l'azione per aggiunge la teoria estratta all'ActiveTheoryList;
  - deve visualizzare una notifica di errore se viene sollevata un'eccezione durante la fase di estrazione della teoria.

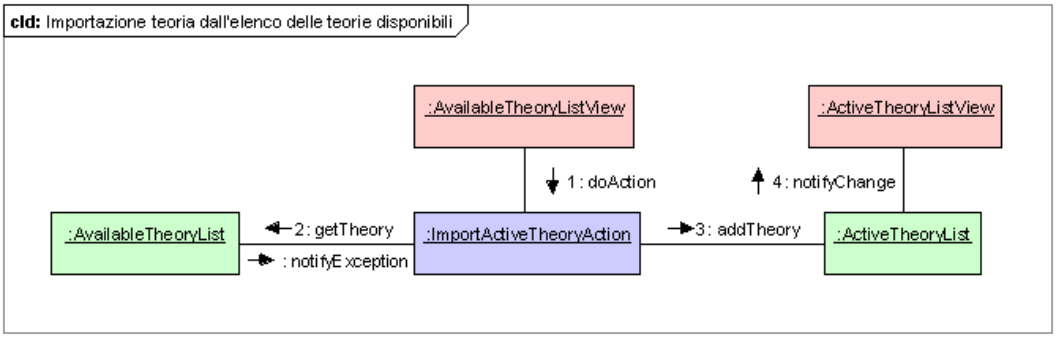


Figura 6.2: Diagramma della collaborazione "importazione teoria dall'elenco delle teorie disponibili"

6.1.2 Collaborazione: *esportazione teoria verso l'elenco delle teorie disponibili*

Con riferimento all'analisi del problema, per poter realizzare il caso d'uso "*esportazione teoria verso l'elenco delle teorie disponibili*" occorre:

- un modello relativo all'elenco delle teorie disponibili;
- un componente grafico in grado di rappresentare l'elenco delle teorie disponibili.

Inoltre, servirà anche un elemento di controllo in grado eseguire l'azione di esportazione della teoria specificata dall'utente.

I partecipanti sono:

- *AvailableTheoryList*: elenco delle teorie disponibili;
- *AvailableTheoryListView*: vista che rappresenta l'elenco delle teorie disponibili;

- *ExportActiveTheoryAction*: azione che, data una teoria attiva, la aggiunge all'elenco delle teorie disponibili.

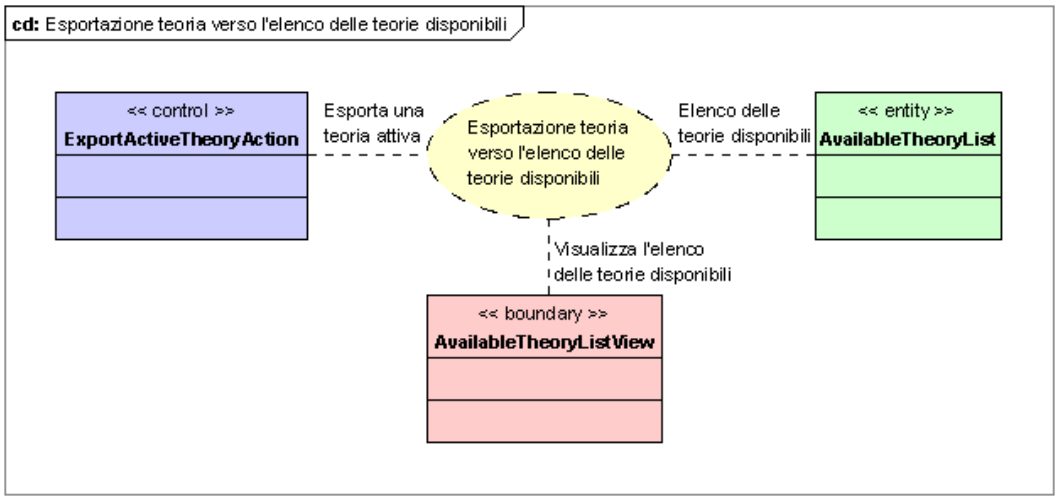


Figura 6.3: Collaborazione relativa al caso d'uso "esportazione teoria verso l'elenco delle teorie disponibili"

Dai ruoli definiti per ogni partecipante, si determinano le relative responsabilità:

- *AvailableTheoryList*: offre il servizio di aggiunta della teoria da esportare all'elenco delle teorie disponibili;
- *AvailableTheoryListView*: riceve le notifiche relative al cambiamento del contenuto dell'elenco delle teorie disponibili e ne visualizza l'effetto;
- *ExportActiveTheoryAction*: aggiunge la teoria da esportare in coda all'AvailableTheoryList.

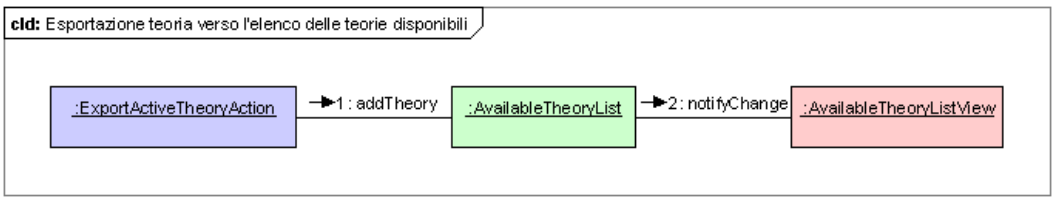


Figura 6.4: Diagramma della collaborazione "esportazione teoria verso l'elenco delle teorie disponibili"

6.1.3 Collaborazione: *estrai una o più teorie attive*

Questa collaborazione è del tutto analoga alla collaborazione "*estrai una teoria attiva*" definita nel secondo prototipo, ma è caratterizzata da due differenze:

- l'entità *ExportTheoryAction* è sostituita dalla nuova entità *ExportActiveTheoryAction*, la quale svolge lo stesso ruolo della precedente, ma in tal caso la teoria da esportare viene inviata verso

l'elenco delle teorie disponibili e non verso il filesystem locale;

- l'entità *GetTheoryAction* può estrarre anche più di una teoria dall'*ActiveTheoryList*, richiedendone l'esportazione all'*ExportActiveTheoryAction* una per volta.

6.1.4 Collaborazione: *estrai la teoria attiva globale*

Questa collaborazione è del tutto analoga alla collaborazione "*estrai la teoria attiva globale*" definita nel secondo prototipo, tranne per un particolare: utilizza l'entità *ExportActiveTheoryAction* al posto della *ExportTheoryAction*.

6.1.5 Collaborazione: *estrai tutte le teorie*

Questa collaborazione è identica alla collaborazione "*estrai una o più teorie attive*", tranne per il fatto che l'entità *GetTheoryAction* estrae tutte le teorie attive una separata dall'altra e ne richiede l'esportazione all'*ExportActiveTheoryAction*, nell'ordine in cui sono disposte nella lista delle teorie attive.

6.1.6 Collaborazione: *modifica elenco delle teorie disponibili*

Con riferimento all'analisi del problema, per poter realizzare il caso d'uso "*modifica elenco delle teorie disponibili*" occorrono:

- un modello relativo all'elenco delle teorie disponibili;
- un componente grafico tale da offrire le funzionalità di gestione delle teorie disponibili.

Inoltre, serviranno anche degli elementi di controllo in grado eseguire un'azione di gestione di un generico elenco di teorie, escluse quella relativa alla separazione di una teoria (perché il piano di lavoro specifica di realizzare tale caratteristica in un prototipo successivo) e quella relativa all'aggiunta di una teoria (perché fa parte di altre collaborazioni). Tali azioni sono:

- impostazione di un'unica teoria ad un elenco di teorie;
- rimozione di una qualunque teoria da un elenco di teorie;
- rimozione di tutte le teorie da un elenco di teorie;
- cambiamento di posizione di una teoria contenuta in un elenco di teorie;

- cambiamento del nome di una teoria contenuta in un elenco di teorie;
- cambiamento del nome di tutte le teorie dotate dello stesso nome contenute in un elenco di teorie;
- fusione in un'unica teoria di un insieme di teorie contenute in un elenco di teorie.

I partecipanti sono:

- *AvailableTheoryList*: elenco di teorie disponibili;
- *AvailableTheoryListView*: vista che consente di visualizzare e gestire l'elenco delle teorie disponibili;
- *SetTheoryAction*: azione che impone un'unica teoria all'elenco delle teorie disponibili;
- *RemoveTheoryAction*: azione che rimuove una qualunque teoria dall'elenco delle teorie disponibili;
- *RemoveAllTheoriesAction*: azione che rimuove tutte le teorie dall'elenco delle teorie disponibili;
- *SetTheoryPositionAction*: azione che cambia la posizione di una teoria contenuta nell'elenco delle teorie disponibili;
- *SetTheoryNameAction*: azione che cambia il nome di una teoria contenuta nell'elenco delle teorie disponibili;
- *SetTheoriesNamesAction*: azione che cambia i nomi di un insieme di teorie omonime contenute nell'elenco delle teorie disponibili;
- *MergeTheoriesAction*: azione che fonde in un'unica teoria un insieme di teorie contenute nell'elenco delle teorie disponibili.

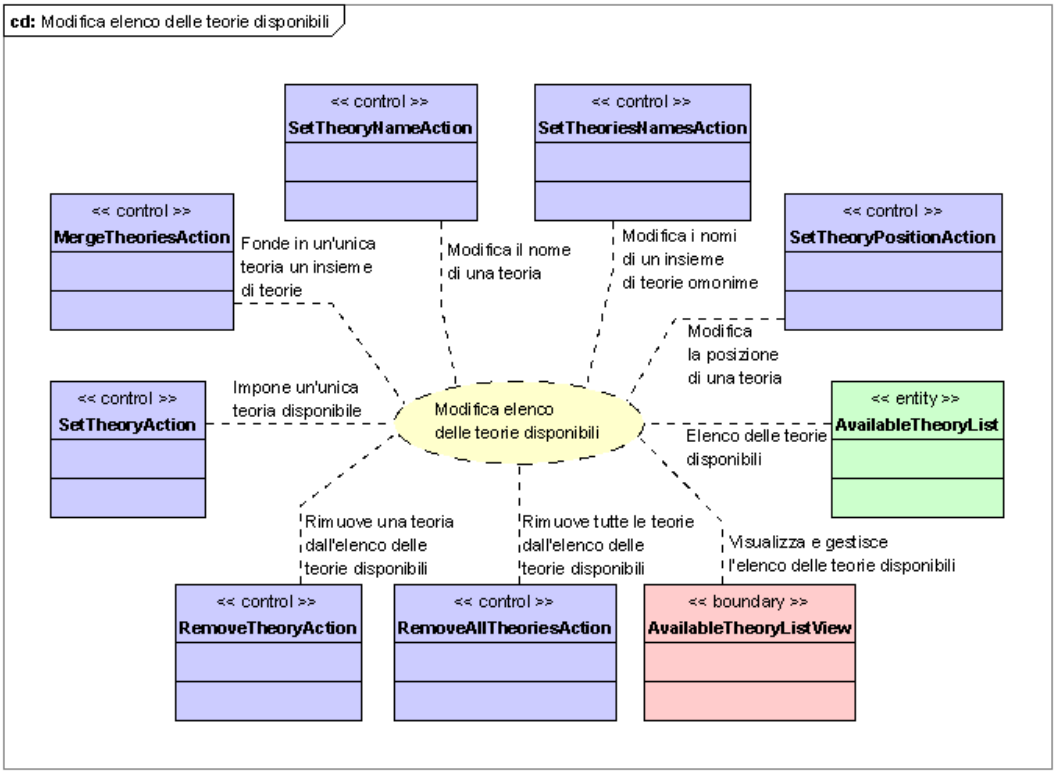


Figura 6.5: Collaborazione relativa al caso d'uso "modifica elenco delle teorie disponibili"

Dai ruoli definiti per ogni partecipante, si determinano le relative responsabilità:

- *AvailableTheoryList*:
  - offre i servizi di accesso e modifica dell'elenco di teorie;
  - notifica il cambiamento del suo contenuto;
- *AvailableTheoryListView*:
  - richiama le azioni di modifica dell'elenco di teorie, su richiesta dell'utente;
  - riceve le notifiche relative al cambiamento del contenuto dell'elenco delle teorie disponibili e ne visualizza l'effetto;
- *SetTheoryAction*: impone un'unica teoria all'AvailableTheoryList;
- *RemoveTheoryAction*: rimuove una qualunque teoria dall'AvailableTheoryList;
- *RemoveAllTheoriesAction*: rimuove tutte le teorie dall'AvailableTheoryList;

- *SetTheoryPositionAction*: cambia la posizione di una teoria contenuta nell'AvailableTheoryList;
- *SetTheoryNameAction*: cambia il nome di una teoria contenuta nell'AvailableTheoryList;
- *SetTheoriesNamesAction*: cambia i nomi di un insieme di teorie omonime contenute nell'AvailableTheoryList;
- *MergeTheoriesAction*: fonde in un'unica teoria un insieme di teorie contenute nell'AvailableTheoryList.

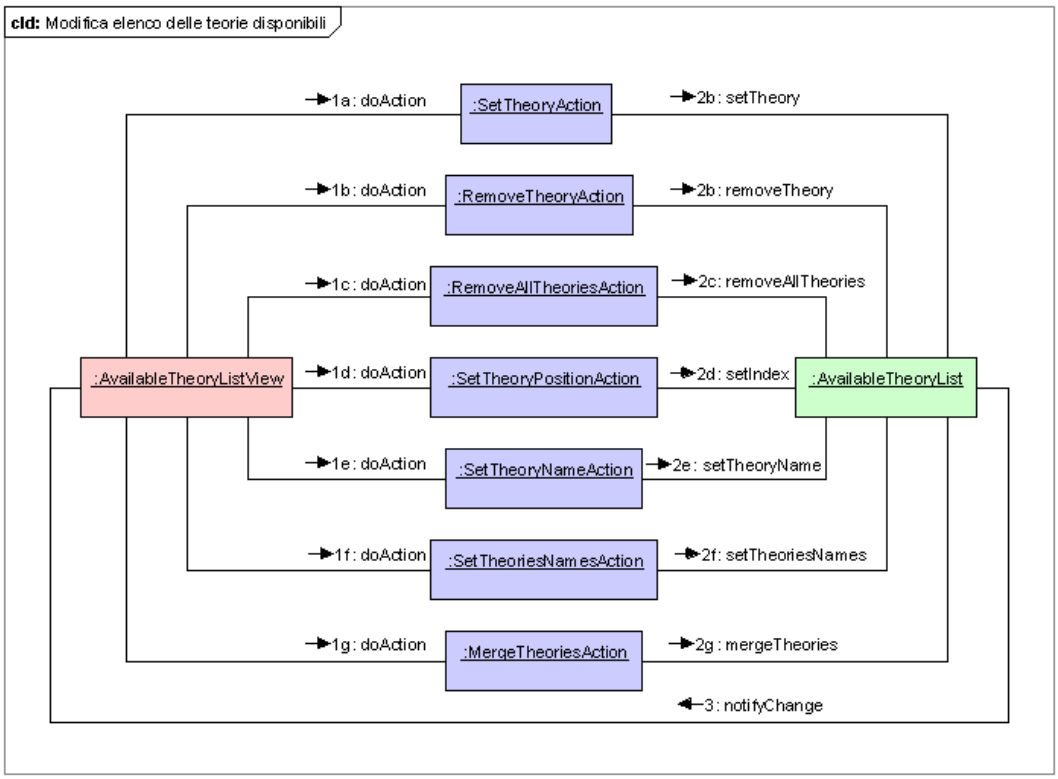


Figura 6.6: Diagramma della collaborazione "modifica elenco delle teorie disponibili"

6.1.7 Collaborazione: *importazione teoria*

Con riferimento all'analisi del problema, per poter realizzare il caso d'uso "*importazione teoria*" occorrono:

- un modello relativo all'elenco delle teorie disponibili;
- un componente grafico tale da offrire le funzionalità di gestione delle teorie disponibili;
- un oggetto in grado di accedere ad un supporto per la persistenza;

- un'interfaccia grafica che permetta all'utente di specificare la locazione della teoria da importare.

Inoltre, serviranno anche due elementi di controllo: uno in grado di recuperare la teoria dal supporto per la persistenza, e l'altro in grado di eseguire l'azione di importazione della teoria.

I partecipanti sono:

- *AvailableTheoryList*: elenco delle teorie disponibili;
- *AvailableTheoryListView*: vista che consente di gestire l'elenco delle teorie disponibili;
- *PersistenceSupport*: gestore del supporto per la persistenza;
- *LocationChooser*: interfaccia grafica per la scelta della locazione della teoria da importare;
- *TheoryLoader*: oggetto in grado di gestire il caricamento di una teoria dal supporto per la persistenza;
- *ImportAvailableTheoryAction*: azione che preleva una teoria dal supporto per la persistenza e la aggiunge all'elenco delle teorie disponibili.

Questa collaborazione è del tutto analoga alla collaborazione del secondo prototipo "*importazione teoria dal filesystem locale*", tranne per il fatto che il *PersistenceSupport* sostituisce il *LocalFileSystem*, il *LocationChooser* sostituisce il *FileChooser*, il *TheoryLoader* sostituisce il *LocalTheoryLoader* e infine l'*ImportAvailableTheoryAction* sostituisce l'*AddTheoryAction*.

6.1.8 Collaborazione: *esportazione teoria*

Con riferimento all'analisi del problema, per poter realizzare il caso d'uso "es*portazione teoria*" occorrono:

- un oggetto in grado di accedere ad un supporto per la persistenza;
- un'interfaccia grafica che permetta all'utente di specificare la locazione della teoria da esportare.

Inoltre, serviranno anche due elementi di controllo: uno in grado di memorizzare la teoria nel supporto per la persistenza e l'altro in grado di eseguire l'azione di esportazione della teoria.

I partecipanti sono:

- *PersistenceSupport*: gestore del supporto per la persistenza;
- *LocationChooser*: interfaccia grafica per la scelta della locazione della teoria da esportare;
- *TheorySaver*: oggetto in grado di gestire il salvataggio di una teoria nel supporto per la persistenza;
- *ExportAvailableTheoryAction*: azione che memorizza una teoria in un supporto per la persistenza.

Questa collaborazione è del tutto analoga alla collaborazione del secondo prototipo "*esportazione teoria verso il filesystem locale*", tranne per il fatto che il *PersistenceSupport* sostituisce il *LocalFileSystem*, il *LocationChooser* sostituisce il *FileChooser*, il *TheorySaver* sostituisce il *LocalTheorySaver* e infine l'*ExportAvailableTheoryAction* sostituisce l'*ExportTheoryAction*.

6.1.9 Collaborazione: *apri teoria da file*

Questa collaborazione è un caso particolare della collaborazione "*importa teoria*": infatti vengono utilizzate le entità *LocalFileSystem*, *FileChooser* e *LocalTheoryLoader* rispettivamente al posto delle entità *PersistenceSupport*, *TheoryChooser* e *TheoryLoader*.

6.1.10 Collaborazione: *salva teoria su file*

Questa collaborazione è un caso particolare della collaborazione "*esporta teoria*": infatti vengono utilizzate le entità *LocalFileSystem*, *FileChooser* e *LocalTheorySaver* rispettivamente al posto delle entità *PersistenceSupport*, *TheoryChooser* e *TheorySaver*.

6.1.11 Collaborazione: *aggiorna teoria*

Questa collaborazione è un caso particolare della collaborazione "*esporta teoria*": infatti si dispone già della locazione della teoria nel supporto per la persistenza, pertanto l'entità *TheorySaver* non richiederà tale informazione al *TheoryChooser*.

6.1.12 Collaborazione: *aggiorna tutte le teorie*

Questa collaborazione è identica alla collaborazione "*aggiorna teoria*", tranne per il fatto che non si limita ad aggiornare una teoria specifica, ma tutte le teorie contenute nella lista delle teorie disponibili a cui è associata una locazione di un qualunque supporto per la persistenza.

6.1.13 Collaborazione: *crea teoria*

Con riferimento all'analisi del problema, per poter realizzare il caso d'uso "*crea teoria*" occorrono:

- un modello relativo all'elenco delle teorie disponibili;
- un componente grafico tale da offrire le funzionalità di gestione delle teorie disponibili;
- un componente grafico che permetta all'utente di specificare il nome logico della teoria da creare.

Inoltre, servirà anche un elemento di controllo in grado di eseguire l'azione di creazione della teoria.

I partecipanti sono:

- *AvailableTheoryList*: elenco delle teorie disponibili;
- *AvailableTheoryListView*: vista che consente di gestire l'elenco delle teorie disponibili;
- *TheoryNameChooser*: interfaccia grafica per la scelta del nome logico della teoria da creare;
- *CreateTheoryAction*: azione che crea una nuova teoria non persistente e la aggiunge all'elenco delle teorie disponibili.

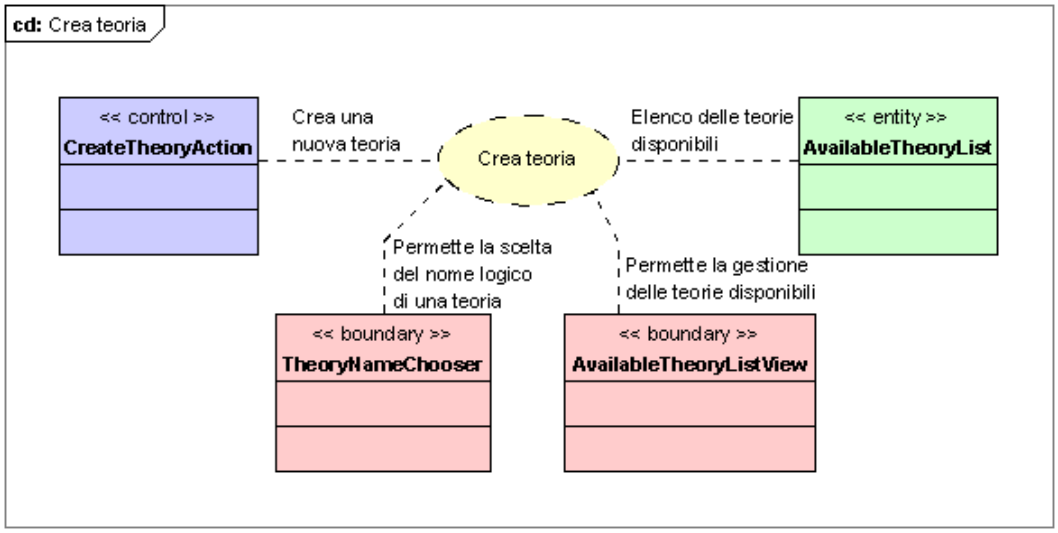


Figura 6.7: Collaborazione relativa al caso d'uso "crea teoria"

Dai ruoli definiti per ogni partecipante, si determinano le relative responsabilità:

- *AvailableTheoryList*:
  - offre i servizi di modifica dell'elenco delle teorie disponibili;
  - notifica il cambiamento del suo contenuto;
- *AvailableTheoryListView*:
  - richiama l'azione CreateTheoryAction allo scopo di creare una nuova teoria e di aggiungerla all'elenco delle teorie disponibili, su richiesta dell'utente;
  - riceve le notifiche relative al cambiamento del contenuto dell'elenco delle teorie disponibili e ne visualizza l'effetto;
- *TheoryNameChooser*:
  - richiede all'utente di specificare il nome logico della teoria da creare;
  - restituisce il nome logico della teoria da creare;
- *CreateTheoryAction*:
  - richiede all'utente di specificare il nome logico della nuova teoria tramite il TheoryNameChooser;
  - crea una nuova teoria dotata del nome logico specificato;
  - aggiunge la nuova teoria in coda all'AvailableTheoryList.

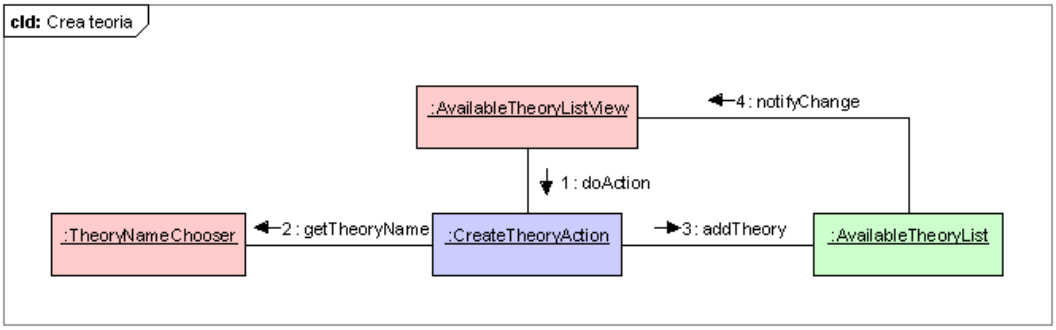


Figura 6.8: Diagramma della collaborazione "crea teoria"

6.1.14 Collaborazione: *cancella teoria*

Con riferimento all'analisi del problema, per poter realizzare il caso d'uso "*cancella teoria*" occorrono:

- un modello relativo all'elenco delle teorie disponibili;
- un componente grafico tale da offrire le funzionalità di gestione delle teorie disponibili;
- un oggetto in grado di accedere ad un supporto per la persistenza.

Inoltre, serviranno anche un elemento di controllo in grado di eseguire l'azione di cancellazione della teoria.

I partecipanti sono:

- *AvailableTheoryList*: elenco delle teorie disponibili;
- *AvailableTheoryListView*: vista che consente di gestire l'elenco delle teorie disponibili;
- *PersistenceSupport*: gestore del supporto per la persistenza;
- *DeleteTheoryAction*: azione che rimuove una teoria dall'elenco delle teorie disponibili e se questa è persistente la cancella dal supporto per la persistenza.

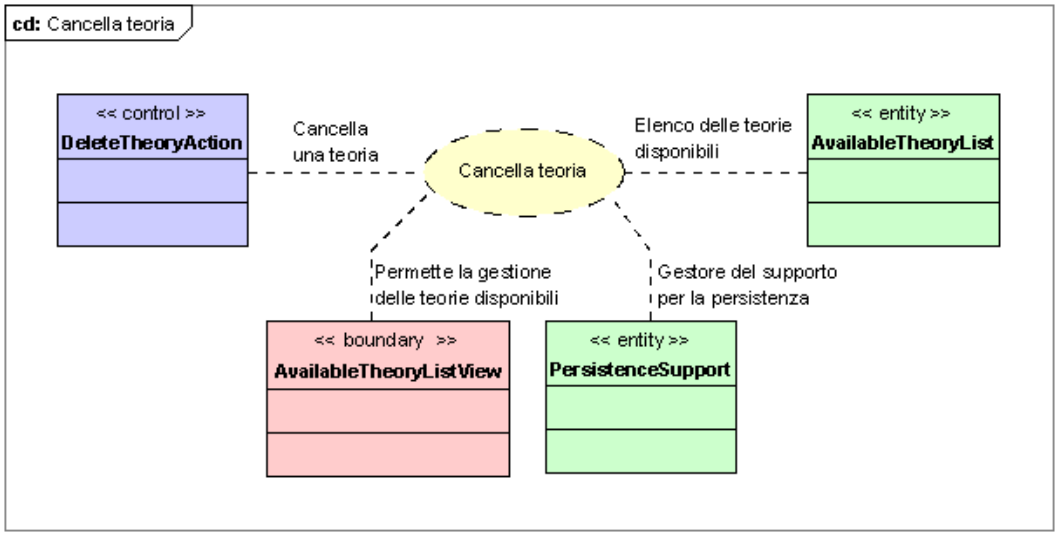


Figura 6.9: Collaborazione relativa al caso d'uso "cancella teoria"

Dai ruoli definiti per ogni partecipante, si determinano le relative responsabilità:

- *AvailableTheoryList*:
  - offre il servizio di cancellazione di una teoria dall'elenco delle teorie disponibili;
  - notifica il cambiamento del suo contenuto;

- *AvailableTheoryListView*:
  - richiama l'azione DeleteTheoryAction allo scopo di cancellare una teoria, su richiesta dell'utente;
  - riceve le notifiche relative al cambiamento del contenuto dell'elenco delle teorie disponibili e ne visualizza l'effetto;
- *PersistenceSupport*:
  - permette di eliminare una teoria dal supporto per la persistenza;
  - solleva un'eccezione in caso di fallimento dell'azione di eliminazione di una teoria;
- *DeleteTheoryAction*:
  - se la teoria specificata dall'utente è persistente richiede al PersistenceSupport la sua cancellazione;

- se la cancellazione della teoria specificata va a buon fine o se essa non è persistente, rimuovere la teoria dall'AvailableTheoryList.

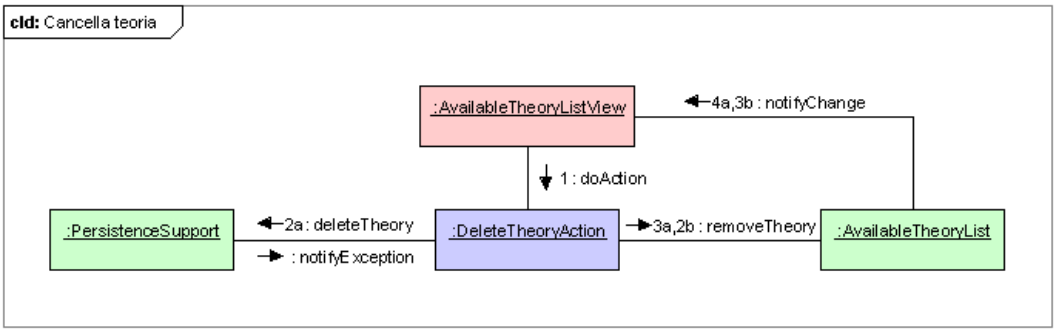


Figura 6.10: Diagramma della collaborazione "cancella teoria"

6.1.15 Collaborazione: *modifica contenuto teoria disponibile*

Con riferimento all'analisi del problema, per poter realizzare il caso d'uso "*modifica contenuto teoria disponibile*" occorrono:

- un modello relativo all'elenco delle teorie disponibili;
- un componente grafico tale da offrire le funzionalità di gestione delle teorie disponibili;
- un editor che permetta all'utente di visualizzare e modificare il contenuto di una teoria.

Inoltre, serviranno anche due elementi di controllo: uno in grado di aprire l'editor di una teoria e l'altro in grado di eseguire l'azione di sostituzione della teoria da modificare con quella modificata.

I partecipanti sono:

- *AvailableTheoryList*: elenco delle teorie disponibili;
- *AvailableTheoryListView*: vista che consente di gestire l'elenco delle teorie disponibili;
- *TheoryEditor*: editor di teorie logiche;
- *OpenTheoryAction*: azione che apre una teoria visualizzandola in un apposito editor;
- *TheoryEditorSupport*: supporto per l'editor che consente di modificare il contenuto di una teoria persistente oppure di sostituire una teoria non persistente con una nuova teoria dotata del contenuto aggiornato.

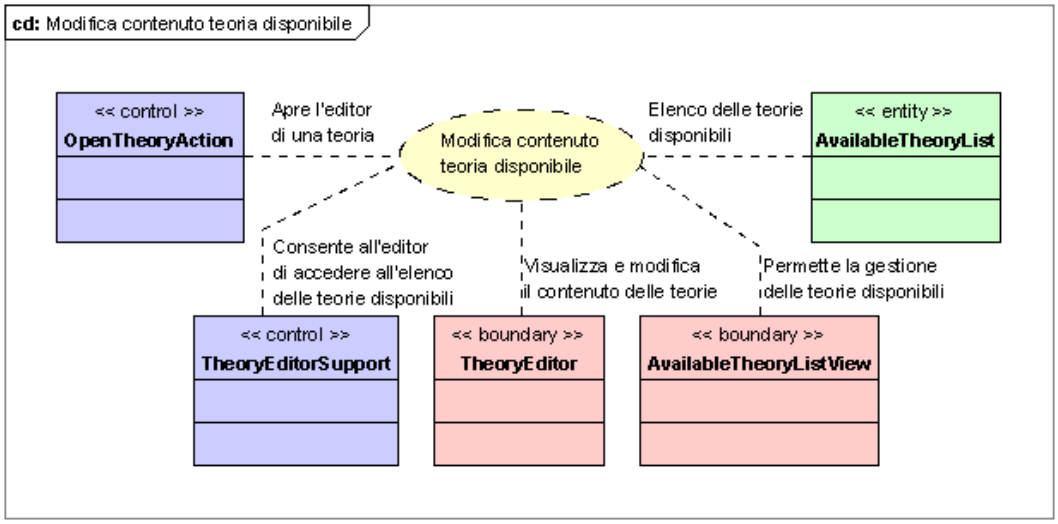


Figura 6.11: Collaborazione relativa al caso d'uso "modifica contenuto teoria disponibile"

Dai ruoli definiti per ogni partecipante, si determinano le relative responsabilità:

- *AvailableTheoryList*:
  - offre i servizi di modifica dell'elenco delle teorie disponibili;
  - notifica il cambiamento del suo contenuto;
- *AvailableTheoryListView*:
  - richiama l'azione OpenTheoryAction allo scopo di aprire una teoria visualizzandola tramite un TheoryEditor, su richiesta dell'utente;
  - riceve le notifiche relative al cambiamento del contenuto dell'elenco delle teorie disponibili e ne visualizza l'effetto;
- *TheoryEditor*:
  - visualizza il contenuto di una teoria logica;
  - permette all'utente di modificare il contenuto di una teoria logica;
  - nel momento in cui il contenuto modificato di una teoria viene salvato, viene delegato al TheoryEditorSupport di sostituire la teoria originale con la teoria modificata;
- *OpenTheoryAction*: apre un TheoryEditor per una specifica teoria;
- *TheoryEditorSupport*:
  - nel caso in cui la teoria da modificare sia persistente, memorizza il contenuto aggiornato in corrispondenza della sua locazione di

persistenza;

- nel caso in cui la teoria da modificare non sia persistente, la sostituisce con una nuova teoria dotata del contenuto aggiornato.

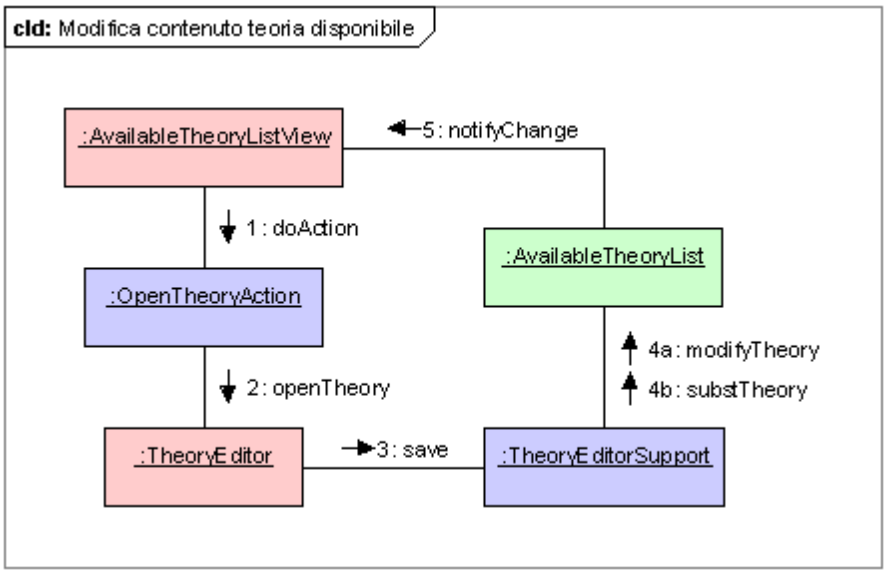


Figura 6.12: Diagramma della collaborazione "modifica contenuto teoria disponibile"

6.1.16 Interazione: *importazione teoria dall'elenco delle teorie disponibili*

Tale interazione viene avviata con un comando da parte dell'utente, impartito tramite l'interfaccia grafica implementata dall'AvailableTheoryListView, il quale delegherà all'ImportActiveTheoryAction la gestione dell'esecuzione dell'azione.

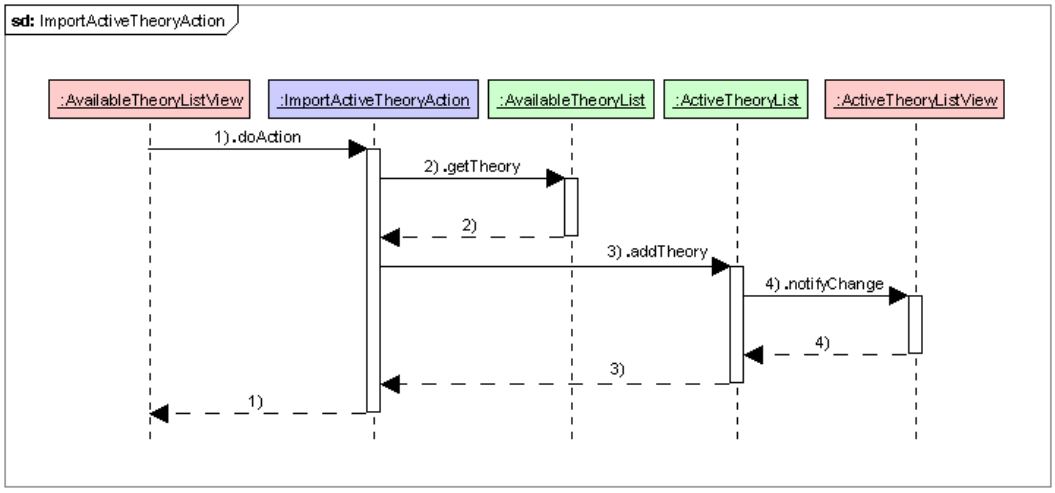


Figura 6.13: Diagramma di sequenza relativo all'interazione "importazione teoria dall'elenco delle teorie disponibili"

Innanzitutto, l'ImportActiveTheoryAction recupererà dall'AvailableTheoryList la teoria da importare, dopodiché aggiungerà la teoria recuperata

all'ActiveTheoryList, il quale provvederà a notificare il cambiamento del proprio contenuto all'ActiveTheoryListView. Nel caso sia sollevata un'eccezione durante il recupero della teoria da importare dall'AvailableTheoryList, l'operazione dovrà fallire e inoltre dovrà essere visualizzato un messaggio di errore.

6.1.17 Interazione: *esportazione teoria verso l'elenco delle teorie disponibili*

Tale interazione può essere avviata da una qualsiasi entità, la quale richiederà alla ExportActiveTheoryAction l'esecuzione dell'azione di esportazione, che consisterà semplicemente nell'aggiungere una teoria all'AvailableTheoryList.

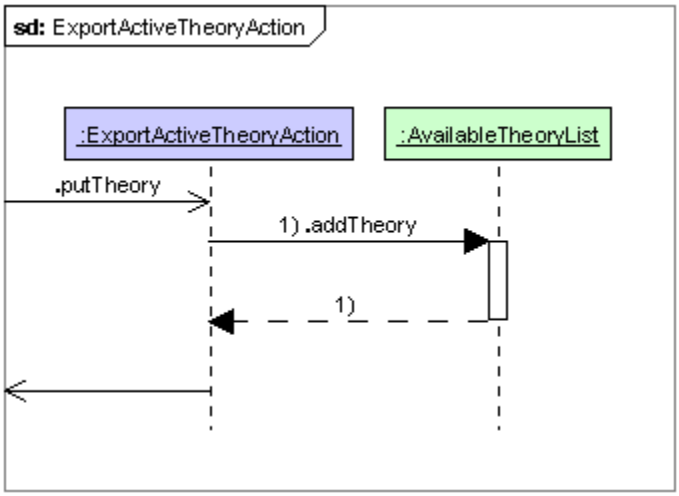


Figura 6.14: Diagramma di sequenza relativo all'interazione "esportazione teoria verso l'elenco delle teorie disponibili"

6.1.18 Interazione: *modifica elenco delle teorie disponibili*

Una interazione di modifica all'elenco delle teorie disponibili viene avviata con un comando da parte dell'utente, impartito tramite l'interfaccia grafica implementata dall'AvailableTheoryListView, il quale delegherà ad una opportuna azione l'esecuzione della modifica.

Le azioni che apportano una modifica all'elenco delle teorie disponibili sono: SetTheoryAction, RemoveTheoryAction, RemoveAllTheoriesAction, SetTheoryPositionAction, SetTheoryNameAction, SetTheoriesNamesAction e MergeTheoriesAction. Le interazioni di cui ognuna di queste azioni è protagonista sono del tutto analoghe tra di loro, ovvero sono caratterizzate dalla stessa sequenza di chiamate:

- chiamata da parte dell'AvailableTheoryListView dell'azione;
- chiamata da parte dell'azione dei metodi dell'AvailableTheoryList

necessari all'effettiva esecuzione dell'azione;

- invio all'AvailableTheoryListView delle notifiche relative ai cambiamenti del contenuto dell'AvailableTheoryList.

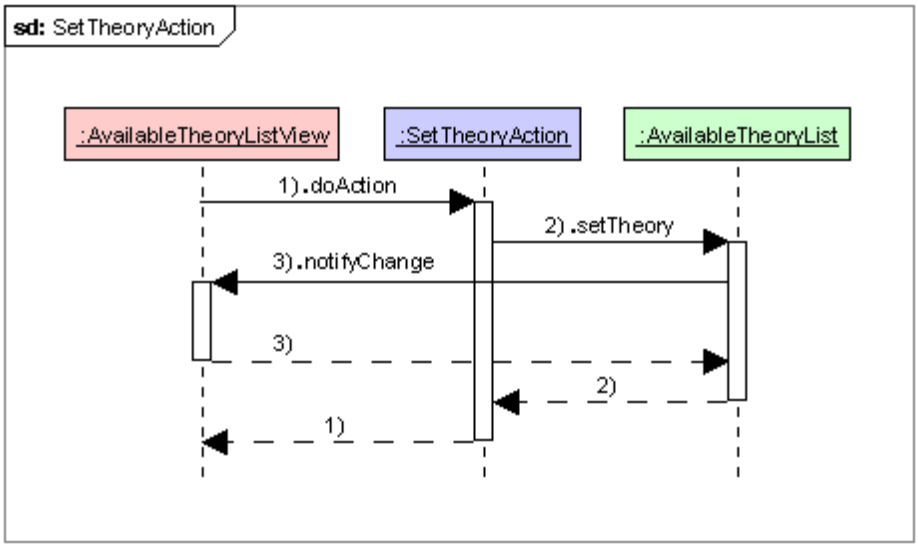


Figura 6.15: Diagramma di sequenza relativo all'interazione "modifica elenco delle teorie disponibili"

6.1.19 Interazioni: *importazione teoria e esportazione teoria*

Le interazioni relative alle collaborazioni denominate "*importazione teoria*" e "*esportazione teoria*" sono analoghe rispettivamente alle interazioni "*importazione teoria dal filesystem locale*" e "*esportazione teoria verso il filesystem locale*" definite nel secondo prototipo, sostituendo opportunamente i partecipanti come indicato nelle corrispondenti collaborazioni.

6.1.20 Interazione: *crea teoria*

Tale interazione viene avviata con un comando da parte dell'utente, impartito tramite l'interfaccia grafica implementata dall'AvailableTheoryListView, il quale delegherà al CreateTheoryAction la gestione dell'esecuzione dell'azione.

Innanzitutto, il CreateTheoryAction delegherà al TheoryNameChooser la presentazione della richiesta di scelta del nome della nuova teoria, oltre ad eseguirne l'acquisizione; data il nome, il CreateTheoryAction creerà una nuova teoria che aggiungerà all'AvailableTheoryList. Infine questo notificherà il cambiamento relativo al suo contenuto all'AvailableTheoryListView.

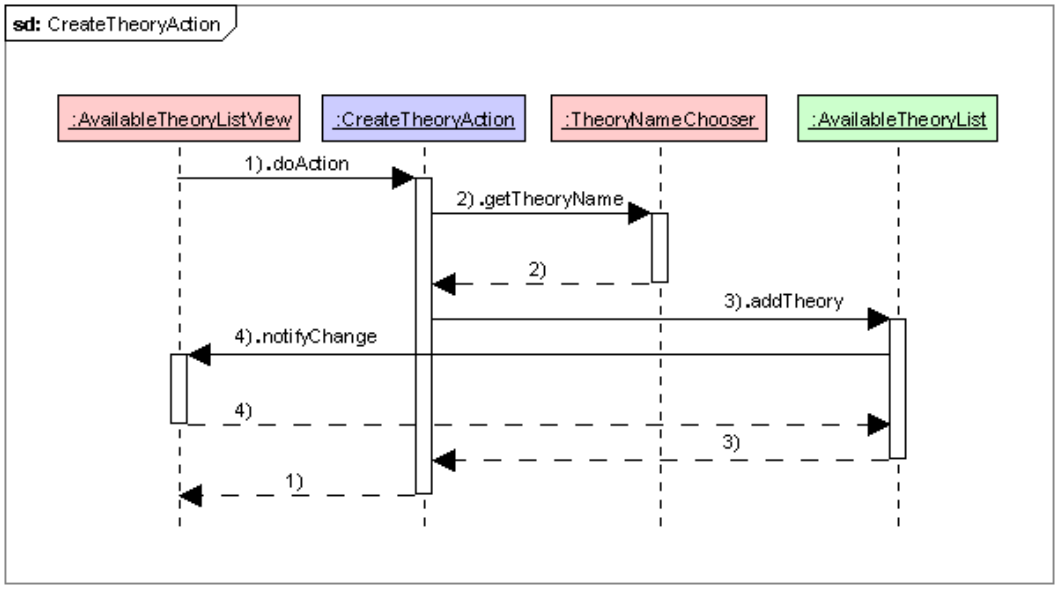


Figura 6.16: Diagramma di sequenza relativo all'interazione "crea teoria"

6.1.21 Interazione: *cancella teoria*

Tale interazione viene avviata con un comando da parte dell'utente, impartito tramite l'interfaccia grafica implementata dall'AvailableTheoryListView, il quale delegherà al DeleteTheoryAction la gestione dell'esecuzione dell'azione.

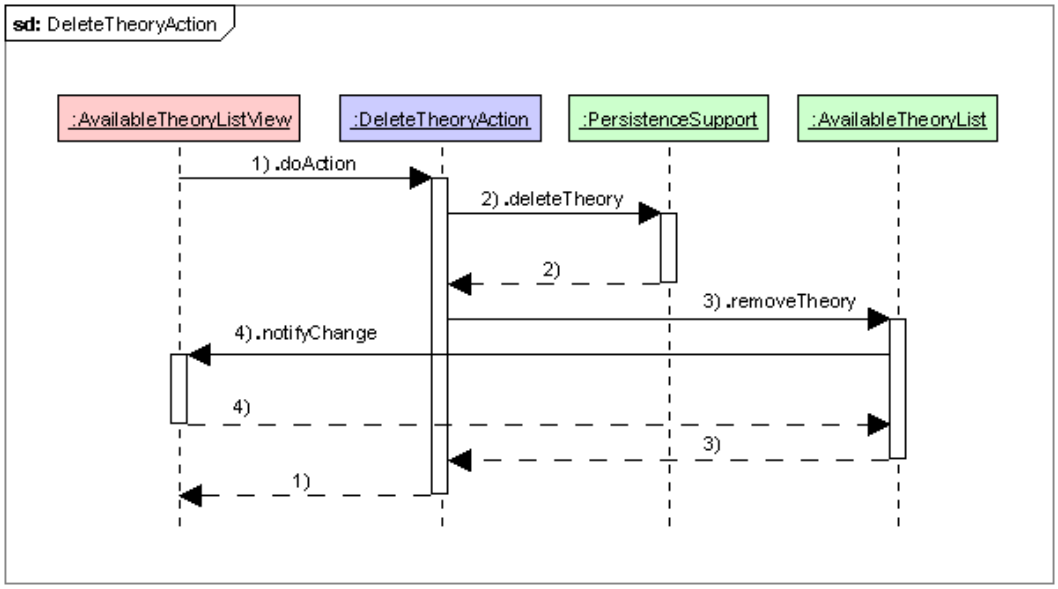


Figura 6.17: Diagramma di sequenza relativo all'interazione "cancella teoria"

Se la teoria specificata dall'utente è persistente, il DeleteTheoryAction tenterà di cancellarla dal supporto per la persistenza tramite il PersistenceSupport, dopodiché procederà alla rimozione della teoria dall'AvailableTheoryList; se invece la teoria specificata dall'utente non è persistente, il DeleteTheoryAction

procederà direttamente alla rimozione di questa dall'AvailableTheoryList. Infine questo notificherà il cambiamento relativo al suo contenuto all'AvailableTheoryListView.

Nel caso in cui sia sollevata un'eccezione durante il tentativo di cancellazione della teoria dal supporto per la persistenza, questa deve essere notificata al DeleteTheoryAction, in modo tale da poter essere segnalata all'utente.

6.1.22 Interazione: *modifica contenuto teoria disponibile*

Tale interazione viene avviata con un comando da parte dell'utente, impartito tramite l'interfaccia grafica implementata dall'AvailableTheoryListView, il quale delegherà al OpenTheoryAction l'apertura della teoria specificata dall'utente con il TheoryEditor.

Il TheoryEditor consentirà all'utente di salvare la teoria eventualmente modificata in qualunque istante; la richiesta di salvataggio dell'utente è di fatto un comando per l'esecuzione di una azione asincrona, la quale consiste nel delegare al TheoryEditorSupport l'aggiornamento del contenuto di una teorie persistente o la sostituzione di una teoria non persistente con una nuova teoria dotata del contenuto aggiornato.

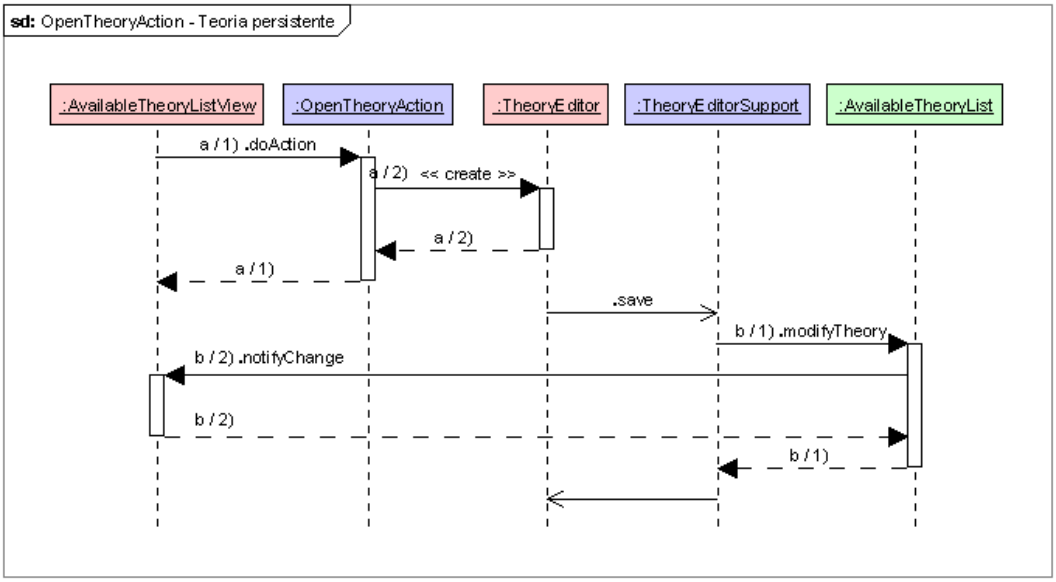


Figura 6.18: Diagramma di sequenza relativo all'interazione "modifica contenuto teoria disponibile" – Caso relativo ad una teoria persistente

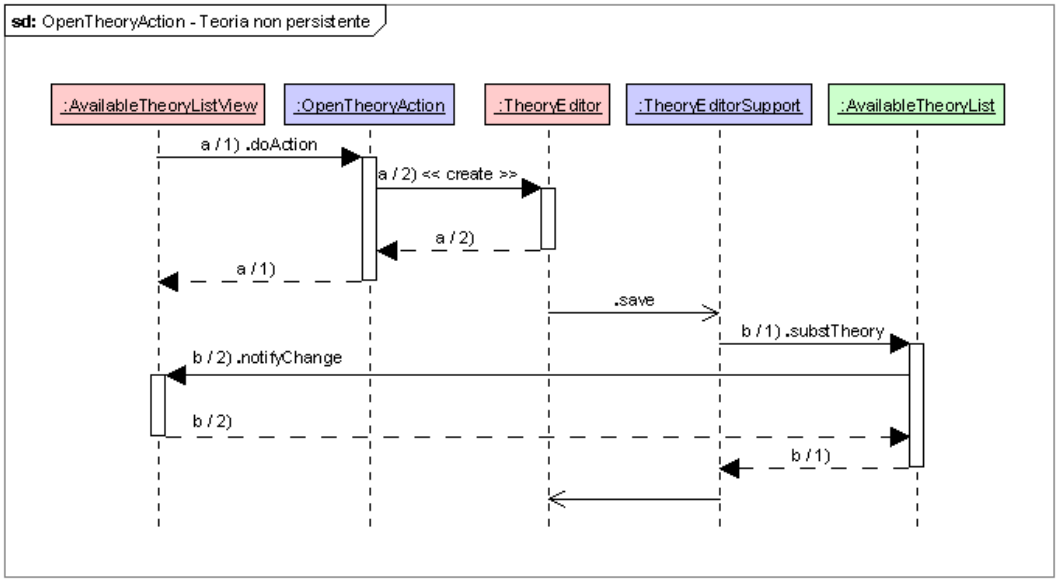


Figura 6.19: Diagramma di sequenza relativo all'interazione "modifica contenuto teoria disponibile" – Caso relativo ad una teoria non persistente

6.1.23 Progettazione dei partecipanti: *AvailableTheoryList e ActiveTheoryList*

L'AvailableTheoryList implementa l'elenco delle teorie disponibili, quindi come l'ActiveTheoryList è un elenco di teorie; queste entità si differenziano soprattutto per le funzionalità offerte: la prima deve essere in grado di offrire tutte quelle relative alla gestione di un elenco di teorie (escludendo quella relativa alla separazione di una teoria, come stabilito dal piano di lavoro), mentre la seconda deve essere in grado di fornire solamente le cosiddette funzionalità supportate.

Un altro aspetto che differenzia l'AvailableTheoryList e l'ActiveTheoryList è il loro contenuto: entrambe possono contenere generiche teorie rappresentate dall'entità Theory, ma solamente l'AvailableTheoryList può contenere teorie persistenti rappresentate da TheoryDataObject[1]. Conseguentemente l'AvailableTheoryList potrebbe essere realizzato utilizzando due liste distinte, una relativa alle teorie non persistenti (oggetti Theory) e una relativa alle teorie persistenti (oggetti TheoryDataObject), mantenendo un'interfaccia esterna come quella di un'unica lista di teorie.

In base all'architettura logica, l'ActiveTheoryList e l'AvailableTheoryList devono condividere interfaccia e caratteristiche comuni e interfaccia, estendendo la medesima entità: la AbstractTheoryList; tale entità sarà realizzata da una classe astratta, la quale sarà estesa dalle classi concrete che realizzeranno

1 I TheoryDataObject erano stati introdotti durante la realizzazione del secondo prototipo al fine di definire il tipo di file "*Prolog Theory*" e incapsulano *file object* relativi a teorie logiche.

l'ActiveTheoryList e l'AvailableTheoryList; le differenze tra queste due entità che dovranno essere implementate distintamente dalle rispettive classi concrete sono:

- l'ActiveTheoryList non supporterà le funzionalità avanzate, anche se implementerà i relativi metodi come l'AvailableTheoryList, ma questi però non faranno altro che lanciare un'eccezione al fine di segnalare che la relativa funzionalità non è supportata;
- l'ActiveTheoryList consente di rimuovere solamente l'ultima teoria dotata di un certo nome, mentre l'AvailableTheoryList consente di rimuovere una qualunque teoria, pertanto tale funzionalità sarà implementata in modo diverso;
- l'AvailableTheoryList dovrà essere in grado di incapsulare anche teorie persistenti realizzate da oggetti TheoryDataObject.

Anche l'AvailableTheoryList, come l'ActiveTheoryList, deve essere in grado di notificare i cambiamenti relativi al suo contenuto inviando eventi ai corrispondenti listener e quindi deve permettere la registrazione/cancellazione di listener che realizzano l'interfaccia `java.beans.PropertyChangeListener`.

Infine, dato che l'elenco delle teorie disponibili è unico, l'AvailableTheoryList dovrà realizzare il design pattern *Singleton*.

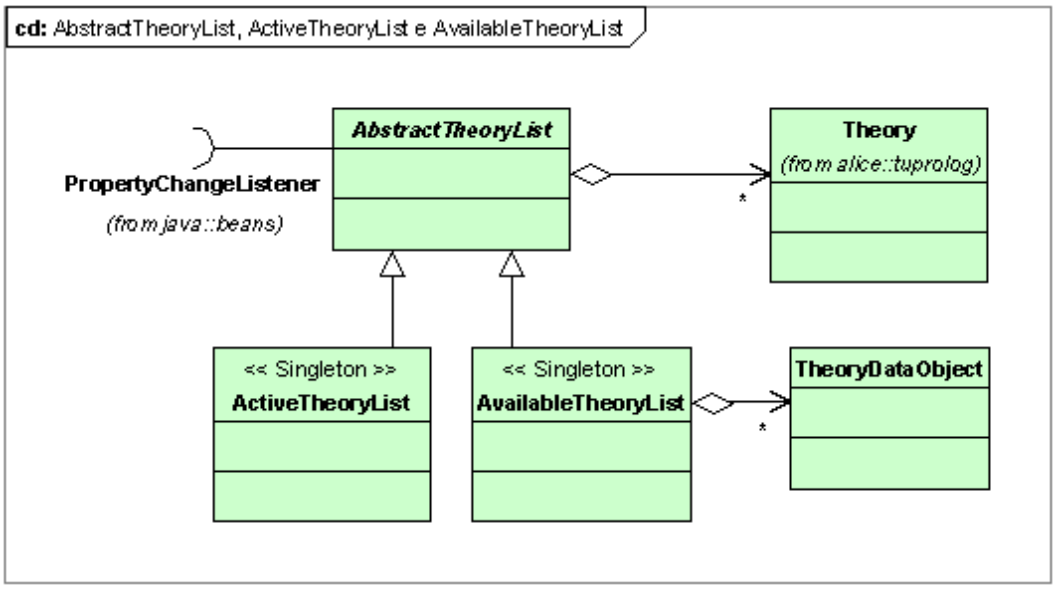


Figura 6.20: Diagramma relativo alle classi AbstractTheoryList, ActiveTheoryList e AvailableTheoryList

6.1.24 Progettazione dei partecipanti: *AvailableTheoryListView e ActiveTheoryListView*

Nell'analisi del problema è stato notato che occorre una vista delle teorie disponibili, ovvero un componente grafico che sia in grado di rappresentare e gestire l'elenco delle teorie disponibili; è stato notato anche che le funzionalità di gestione dell'elenco devono essere offerte tramite appositi menù contestuali.

L'AvailableTheoryListView rappresenterà un elenco di teorie analogamente all'ActiveTheoryListView, pertanto realizzerà una *explorer view*; entrambe incapsuleranno un elenco di teorie, offrendone i relativi servizi di gestione verso l'utente. Quindi queste due entità condivideranno molte caratteristiche, ma si differenzieranno per i seguenti aspetti:

- l'ActiveTheoryListView è un listener del PrologEngine, mentre all'AvailableTheoryListView non è registrato come listener di qualche entità;
- l'ActiveTheoryListView può essere disattivato, mentre l'AvailableTheoryListView no;
- l'ActiveTheoryListView è integrato nel QueryControlPanel, mentre l'ActiveTheoryListView sarà integrato in un apposito *top component*.

In base all'analisi delle differenze tra ActiveTheoryListView e AvailableTheoryListView si può dedurre che l'ActiveTheoryListView è dotato di alcune caratteristiche in più rispetto all'AvailableTheoryList; infatti l'AvailableTheoryListView realizza una vista che svolgerà un'unica funzione: rappresentare un elenco di teorie. Pertanto, è conveniente introdurre una entità che realizzi una *explorer view* in grado di rappresentare un elenco di teorie, che sarà implementata da una classe concreta: la TheoryListView. Tale entità sarà estesa sia dall'AvailableTheoryListView, la quale non aggiungerà nuove importanti caratteristiche, sia dall'ActiveTheoryListView, la quale invece aggiungerà nuove responsabilità.

L'apposito *top component* che conterrà l'AvailableTheoryListView sarà denominato AvailableTheoriesTopComponent; esso potrà anche contenere una toolbar dotata di una serie di pulsanti per eseguire l'importazione o l'esportazione di una teoria disponibile da o verso uno specifico supporto per la persistenza o anche verso l'elenco delle teorie attive. Per nascondere la complessità di questo *top component* sarà opportuno applicare il design pattern *Façade* [GoF95].

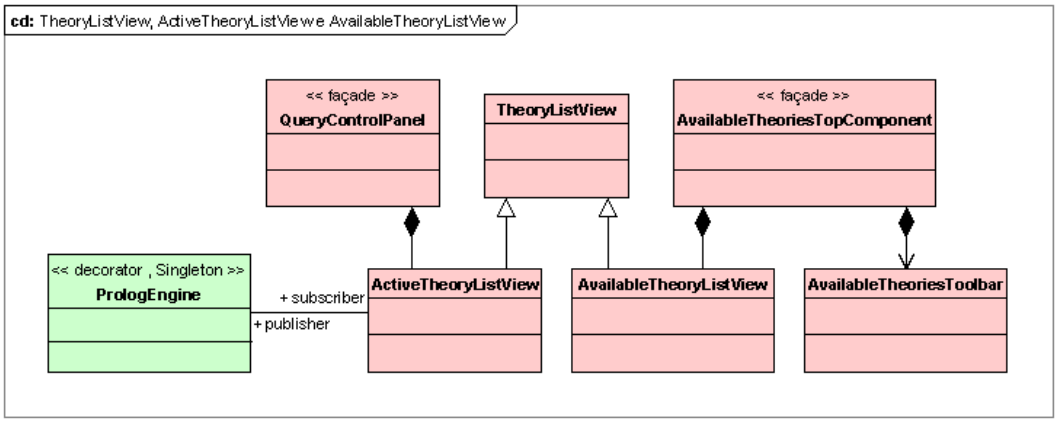


Figura 6.21: Diagramma relativo alle classi TheoryListView, ActiveTheoryListView e AvailableTheoryListView

Il nodo associato all'AvailableTheoriesTopComponent sarà denominato AvailableTheoryListNode e incapsulerà l'AvailableTheoryList; inoltre ad esso saranno aggregati una serie di nodi foglia, i quali incapsuleranno le singole teorie che formano l'elenco delle teorie disponibili. Tali nodi foglia devono rappresentare distintamente teorie persistenti e teorie non persistenti; questi nodi sono realizzati rispettivamente dalle entità TheoryDataNode[2] e TheoryNode. A differenza di un oggetto TheoryNode, un oggetto TheoryDataNode può ricevere notifiche sul cambiamento del contenuto di una teoria dal *data object* che esso incapsula, tra cui la notifica dell'avvenuta cancellazione di una teoria da parte di una entità esterna al plug-in.

Come l'ActiveTheoryListNode, anche l'AvailableTheoryListNode dovrà essere aggiornato dal suo oggetto incapsulato relativamente ai cambiamenti del suo contenuto, pertanto sarà un listener dell'AvailableTheoryList.

Anche in questo caso si può pensare di raccogliere tutte le caratteristiche comuni tra ActiveTheoryListNode e AvailableTheoryListNode in un'unica entità: la AbstractTheoryListNode; tale entità sarà realizzata da una classe astratta, e sarà estesa dalle classi concrete che realizzeranno le entità ActiveTheoryListNode e AvailableTheoryListNode, le quali implementeranno solamente le caratteristiche che le differenziano.

2 I TheoryDataNode erano stati introdotti durante la realizzazione del secondo prototipo al fine di definire il tipo di file "*Prolog Theory*" e incapsulano TheoryDataObject.

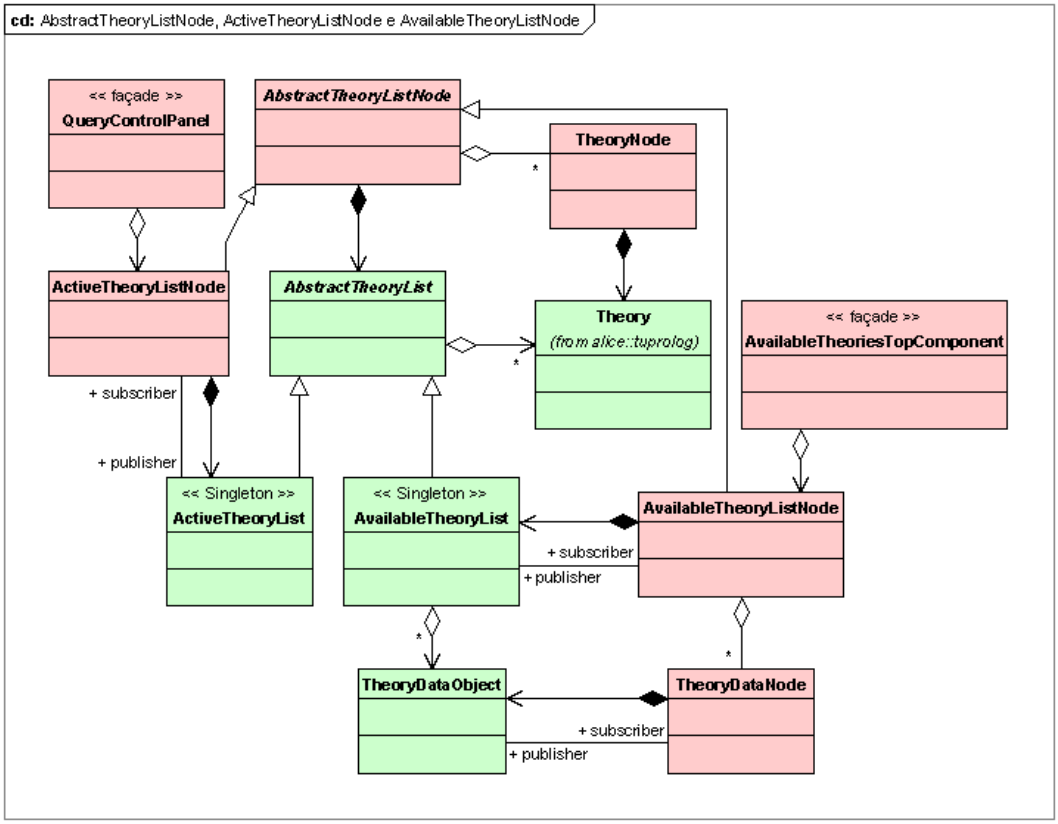


Figura 6.22: Diagramma relativo alle classi AbstractTheoryListNode, ActiveTheoryListNode e AvailableTheoryListNode

Le differenze tra ActiveTheoryListNode e AvailableTheoryListNode consistono in:

- al primo saranno aggregati solamente TheoryNode, mentre al secondo potranno essere aggregati anche TheoryDataNode;
- il primo sarà un listener dell'ActiveTheoryList, mentre il secondo sarà un listener dell'AvailableTheoryList;
- soltanto il secondo permetterà l'esecuzione delle azioni legate alle funzionalità avanzate, escludendo la funzionalità di separazione di una teoria;
- le azioni di importazione/esportazione di teorie che possono essere eseguite sono totalmente differenti tra le due entità.

6.1.25 Progettazione dei partecipanti: azioni di gestione delle teorie attive

Le azioni di gestione relative alle teorie attive saranno le stesse di quelle del secondo prototipo, escludendo la ExportTheoryAction, che sarà sostituita dalla ExportActiveTheoryAction, e la AddTheoryAction, che sarà sostituita dalla ImportActiveTheoryAction.

6.1.26 Progettazione dei partecipanti: azioni di gestione delle teorie disponibili

Le azioni di gestione delle teorie disponibili, analogamente a quelle relative alla gestione delle teorie attive, si suddividono in tre tipologie:

- *azioni che si riferiscono a tutto l'elenco delle teorie disponibile*:
  - CreateTheoryAction: azione che crea una nuova teoria non persistente e la aggiunge all'elenco delle teorie disponibili;
  - ImportAvailableTheoryAction: importa una teoria aggiungendola all'elenco delle teorie disponibili;
  - RemoveAllTheoriesAction: rimuove tutte le teorie dall'elenco delle teorie disponibili;
- *azioni che si riferiscono a più di una teoria disponibile*:
  - MergeTheoriesAction: azione che fonde in un'unica teoria un insieme di teorie contenute nell'elenco delle teorie disponibili;
- *azioni che si riferiscono ad una specifica teoria disponibile*:
  - OpenTheoryAction: azione che apre una teoria disponibile visualizzandola in un apposito editor;
  - ExportAvailableTheoryAction: azione che memorizza una teoria in un supporto per la persistenza;
  - ImportActiveTheoryAction: azione che importa una teoria appartenente all'elenco delle teorie disponibili nell'elenco delle teorie attive;
  - RemoveTheoryAction: rimuove la teoria indicata dall'elenco delle teorie disponibili;

- DeleteTheoryAction: azione che rimuove una teoria dall'elenco delle teorie disponibili e se questa è persistente la cancella dal supporto per la persistenza;
- SetTheoryNameAction: azione che cambia il nome di una teoria contenuta nell'elenco delle teorie disponibili;
- SetTheoriesNamesAction: azione che cambia i nomi di un insieme di teorie omonime alla teoria selezionata e contenute nell'elenco delle teorie disponibili;
- SetTheoryAction: imposta la teoria indicata come unica teoria disponibile;
- SetTheoryPositionAction: azione che cambia la posizione di una teoria contenuta nell'elenco delle teorie disponibili.

Le azioni relative ad una specifica teoria disponibile devono essere richiamabili esclusivamente attraverso un menù contestuale relativo ad una specifica teoria[3] (vedi 5.1.21). L'unica azione relativa a più di una teoria disponibile deve essere richiamabile esclusivamente attraverso un menù contestuale relativo ad una specifica teoria. Le azioni relative all'intero elenco delle teorie disponibili devono essere richiamabili sia attraverso un qualunque menù contestuale relativo ad una specifica teoria, sia attraverso un menù contestuale relativo alla vista delle teorie disponibili (analogo a quello relativo alla vista delle teorie attive, vedi 5.1.21). L'azione SetTheoryNameAction sarà richiamabile anche attraverso l'esecuzione di un'operazione di nuova denominazione di un nodo eseguita dall'utente, mentre le azioni ImportAvailableTheoryAction e ExportAvailableTheoryAction sono richiamabili tramite la pressione di appositi pulsanti presenti nella toolbar dell'AvailableTheoriesTopComponent.

Le azioni relative all'intero elenco delle teorie disponibili sono sempre attive, mentre quella relativa a più di una teoria disponibile è attiva soltanto se sono selezionate due o più teorie disponibili. Infine, le azioni relative ad una specifica teoria sono tutte sempre attive tranne la DeleteTheoryAction, la quale deve essere attiva solamente se la teoria specificata dall'utente è persistente.

3 Ad eccezione dell'azione SetTheoryPositionAction, la quale sarà richiamata a seguito di un drag & drop eseguito dall'utente relativamente ad un nodo di una teoria contenuta nell'elenco delle teorie disponibili.

Come le azioni di gestione delle teorie attive, le azioni relative alle teorie disponibili devono essere richiamate direttamente dal widget specifico a cui si riferiscono, ovvero o dall'AvailableTheoryListNode o da un qualunque TheoryNode o da un qualunque TheoryDataNode; inoltre tali azioni devono implementare delle *cookie action*, le quali richiamano uno specifico *cookie* incapsulato nel nodo che delinea il contesto di esecuzione dell'azione.

Dato che tutte le azioni di gestione di un generico elenco di teorie sono state progettate (escludendo quella relativa alla separazione di una teoria), si potrebbe realizzare un'interfaccia che specifica tutte le funzionalità di un nodo appartenente ad un elenco di teorie; tale interfaccia sarà denominata ITheoryCookie e sarà realizzata dagli specifici cookie realizzati come inner-class delle classi che implementeranno nodi relativi a teorie logiche.

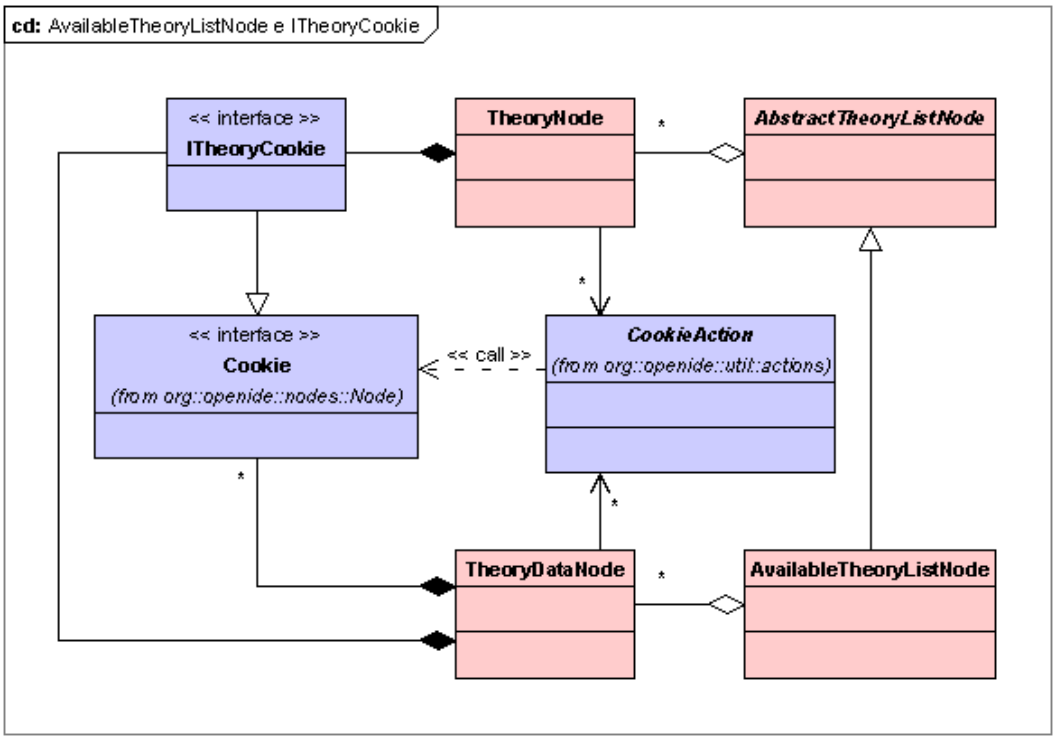


Figura 6.23: Diagramma relativo alle classi AvailableTheoryListNode e ITheoryCookie

Infine, le istanze di tali azioni devono essere uniche, quindi ad esse verrà applicato il design pattern *Singleton*.

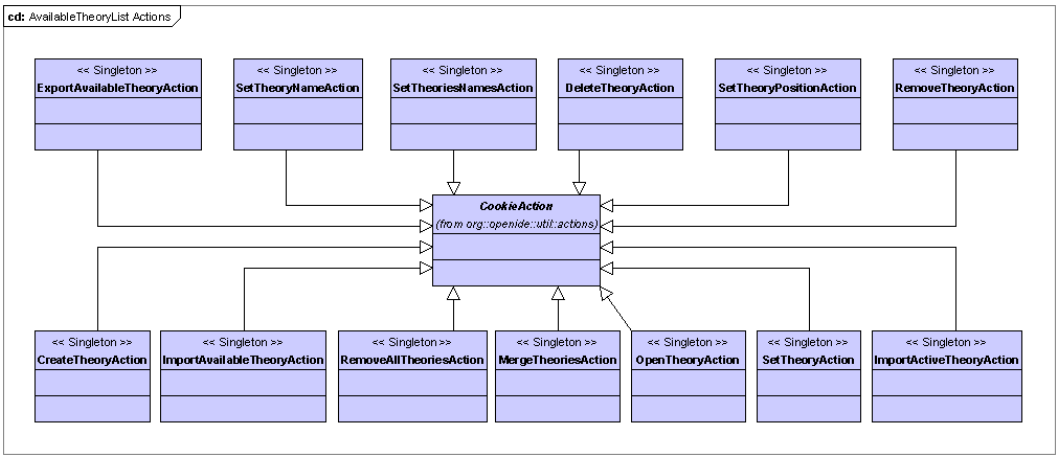


Figura 6.24: Diagramma delle classi relative alle azioni per la gestione delle teorie disponibili

6.1.27 Progettazione dei partecipanti: *TheoryEditor* e *TheoryEditorSupport*

Il partecipante TheoryEditor sarà realizzato dall'editor standard dell'ambiente NetBeans, pertanto dovrà essere affiancato da un opportuno *editor support*, il quale sarà implementato dal TheoryEditorSupport. Oltre a questo è anche richiesto un ambiente relativo all'oggetto editabile, responsabile della comunicazione tra editor support e oggetto editabile; tale ambiente sarà denominato TheoryEditorEnvironment.

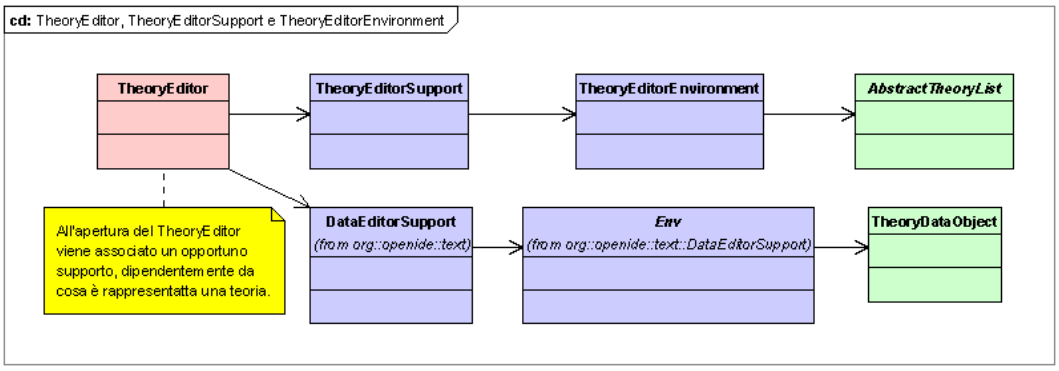


Figura 6.25: Diagramma relativo alle classi TheoryEditor, TheoryEditorSupport e TheoryEditorEnvironment

Gli oggetti che il TheoryEditor deve essere in grado di editare sono di due tipologie: teorie persistenti, rappresentate da oggetti TheoryDataObject, e teorie non persistenti, rappresentate da oggetti Theory. Nel primo caso, invece di realizzare un apposito *editor support* ed un apposito ambiente si potrebbero utilizzare l'*editor support* standard e il relativo ambiente per *data object*, definiti nelle NetBeans API, mentre nel secondo caso non c'è altra soluzione che la realizzazione di apposite entità, corrispondenti al TheoryEditorSupport e al TheoryEditorEnvironment. In particolare il TheoryEditorEnvironment dovrà

effettuare la sostituzione delle teorie non persistenti con le nuove teorie dotate del contenuto aggiornato.

6.1.28 Progettazione dei partecipanti: altri partecipanti

I partecipanti che non sono ancora stati descritti e che consentono la realizzazione di vari casi d'uso sono:

- PersistenceSupport: rappresenta un generico supporto per la persistenza; nell'ambiente NetBeans un supporto per la persistenza generalmente è basato sul concetto di *filesystem*. Tale partecipante è stato introdotto per indicare la necessità di avere un'interfaccia comune per le entità che realizzeranno supporti per la persistenza per questo plug-in;
- TheoryLoader: rappresenta un generico oggetto in grado di caricare una teoria logica persistente da uno specifico supporto per la persistenza; anche questo partecipante è stato introdotto per indicare la necessità di avere un'interfaccia comune per le entità che dovranno effettuare il caricamento di una teoria persistente;
- TheorySaver: rappresenta un generico oggetto in grado di salvare una teoria logica su di uno specifico supporto per la persistenza; anche questo partecipante è stato introdotto per indicare la necessità di avere un'interfaccia comune per le entità che dovranno effettuare la memorizzazione di una teoria persistente;
- LocationChooser: rappresenta un generico oggetto che consente di scegliere la locazione di una teoria logica su di uno specifico supporto per la persistenza; anche questo partecipante è stato introdotto per indicare la necessità di avere un'interfaccia comune per le entità che dovranno consentire all'utente di scegliere la locazione di una teoria persistente;
- TheoryNameChooser: interfaccia grafica che consente all'utente di specificare il nome logico di una nuova teoria generata dall'azione CreateTheoryAction.

6.2 Progettazione del quarto prototipo

La progettazione del quarto prototipo si baserà su ciò che suggerisce di realizzare il piano di lavoro:

Quarto Prototipo: *miglioramento del componente grafico per la gestione delle teorie attive tramite l'integrazione delle funzionalità avanzate, escludendo la separazione di una teoria; miglioramento del componente grafico per l'esecuzione di query tramite l'integrazione della funzionalità di indicazione delle teorie da utilizzare, escludendo la possibilità di eseguire le custom query*.

Lo sviluppo di questo prototipo sarà caratterizzato dalla realizzazione del caso d'uso "*verifica correttezza query*", ma soprattutto dalla reingegnerizzazione di alcuni componenti del sistema al fine di aggiungere ulteriori funzionalità.

6.2.1 Collaborazione: *verifica correttezza query*

Con riferimento all'analisi del problema, per poter realizzare il caso d'uso "*verifica correttezza query*" occorrono:

- un parser Prolog;
- un campo di testo per l'immissione di query;
- un vista che consente di gestire la risoluzione di query.

Inoltre, servirà anche un elemento di controllo in grado di eseguire l'azione di verifica della correttezza di una query.

I partecipanti sono:

- *PrologParser*: parser Prolog;
- *QueryTextField*: campo di testo per l'immissione di query;
- *QueryControlPanel*: vista per la gestione della risoluzione di query;
- *ParseAction*: azione che verifica la correttezza della query inserita.

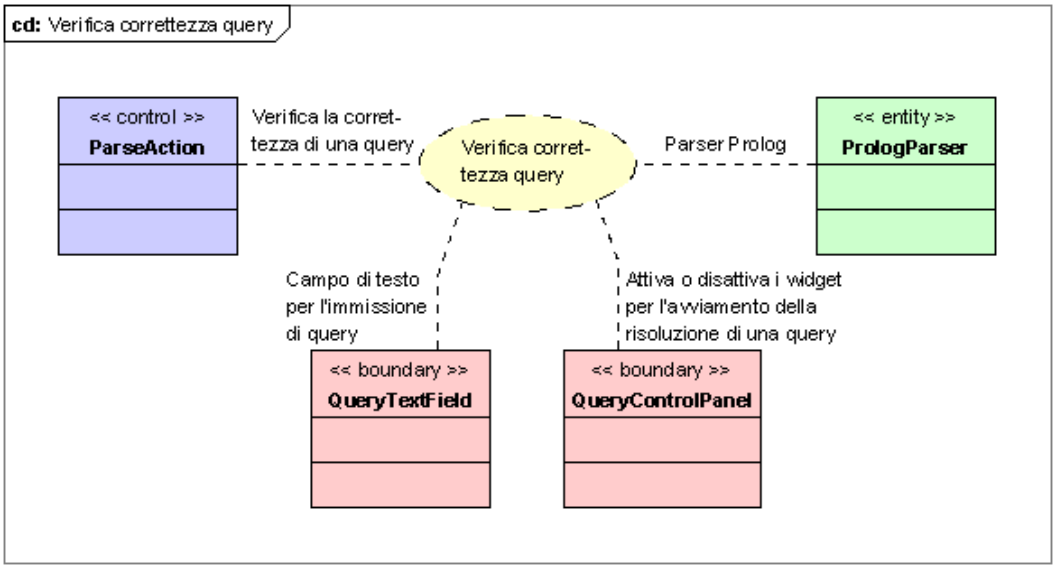


Figura 6.26: Collaborazione relativa al caso d'uso "verifica correttezza query"

Dai ruoli definiti per ogni partecipante, si determinano le relative responsabilità:

- *PrologParser*: offre il servizio di riconoscimento di un termine Prolog;
- *QueryTextField*:
  - permette all'utente di immettere delle query;
  - richiama la ParseAction ad ogni inserimento di carattere, passando come parametro la stringa fino a quel momento immessa dall'utente;
- *QueryControlPanel*: offre il servizio di attivazione/disattivazione dei widget per l'avviamento della risoluzione di una query;
- *ParseAction*:
  - richiede al PrologParser di eseguire il riconoscimento della stringa passata come parametro dal QueryTextField;
  - se il riconoscimento è andato a buon fine segnala al QueryControlPanel di attivare i widget per l'avviamento della risoluzione di una query;
  - se il riconoscimento non è andato a buon fine segnala al QueryControlPanel di disattivare i widget per l'avviamento della risoluzione di una query.

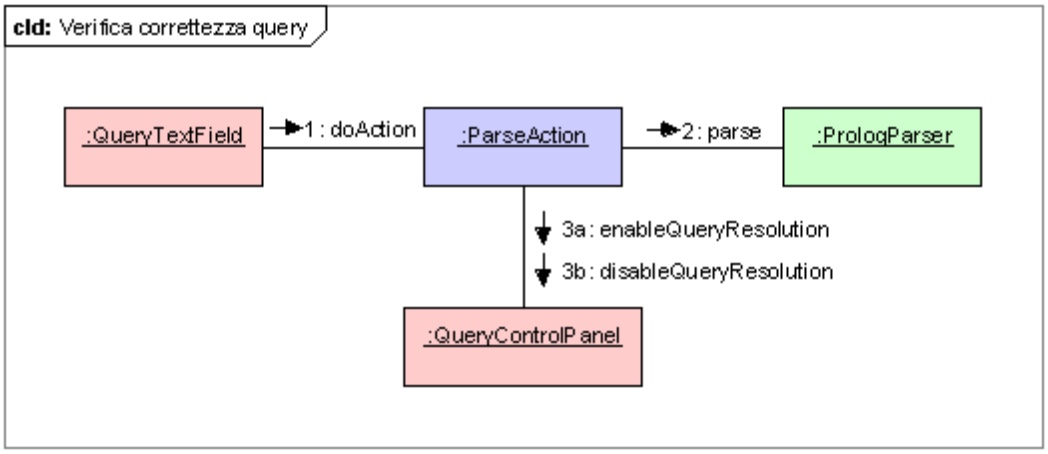


Figura 6.27: Diagramma relativo alla collaborazione "verifica correttezza query"

6.2.2 Interazione: *verifica correttezza query*

Tale interazione viene eseguita ogni volta che l'utente inserisce un carattere nel QueryTextField, il quale delegherà alla ParseAction la gestione dell'esecuzione dell'azione.

La ParseAction richiederà di eseguire il riconoscimento al PrologParser, il quale darà o una risposta positiva o una risposta negativa; nel primo caso la ParseAction indicherà al QueryControlPanel che i widget in grado di avviare la risoluzione di una query devono essere attivi, mentre nel secondo caso indicherà che non devono essere attivi.

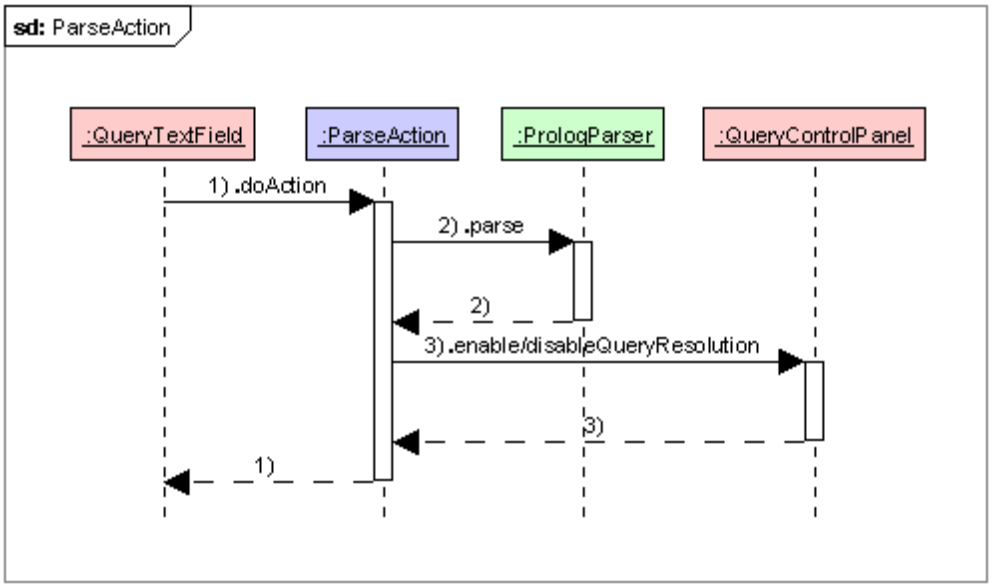


Figura 6.28: Diagramma di sequenza relativo all'interazione "verifica correttezza query"

6.2.3 Progettazione dei partecipanti: *PrologParser, QueryTextField e ParseAction*

Il PrologParser sarà un semplice servizio per il riconoscimento di termini Prolog, il quale dovrà essere in grado di restituire un termine nel caso in cui la stringa analizzata sia una query consistente o un'eccezione nel caso in cui la query non sia consistente; incapsulerà un'istanza della classe `alice.tuprolog.Parser` che utilizzerà per eseguire il suo servizio.

Il QueryTextField realizzerà il campo di testo per l'immissione di query integrato nel QueryControlPanel; esso richiederà la verifica della correttezza della stringa immessa tramite l'azione ParseAction ogni volta che l'utente inserirà un carattere.

La ParseAction sarà un'azione che richiederà il riconoscimento della stringa indicata dal QueryTextField al PrologParser; se l'ultimo carattere diverso dallo spazio di tale stringa non è un punto, il QueryTextField aggiungerà automaticamente un punto finale ad essa prima di richiederne la verifica. A seconda del risultato ottenuto richiamerà un apposito metodo del QueryControlPanel al fine di abilitare o disabilitare tutti i widget per l'avviamento della risoluzione di una query in esso contenuti.

La ParseAction sarà registrata come listener del QueryTextField durante la creazione del QueryTextField stesso; sia la ParseAction, sia il PrologParser realizzeranno il design pattern *Singleton* [GoF95].

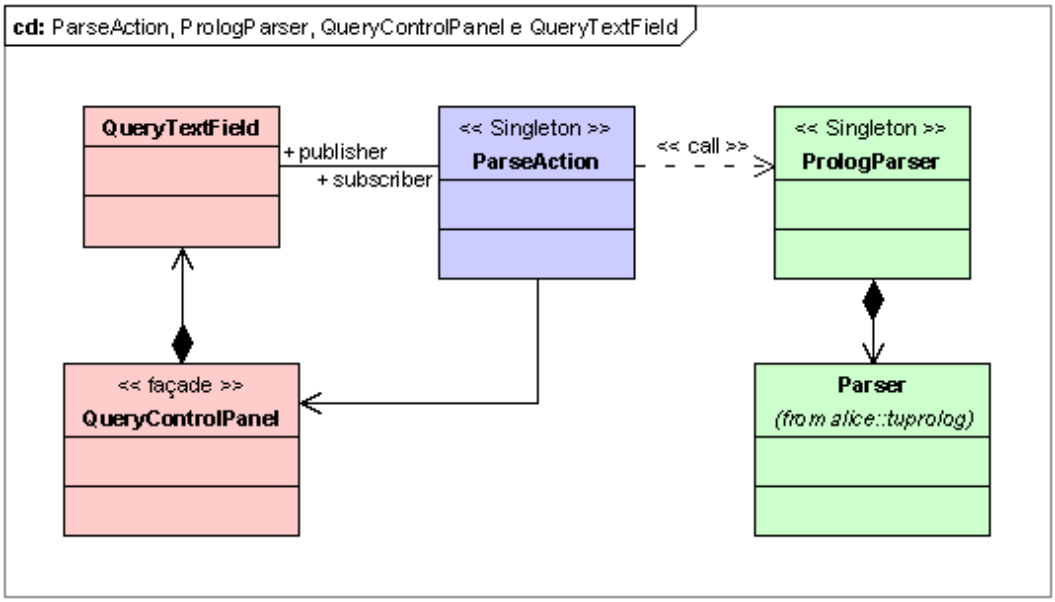


Figura 6.29: Diagramma delle classi ParseAction, PrologParser, QueryControlPanel e QueryTextField

6.2.4 Reingegnerizzazione: *ActiveTheoryList e TheoryManager*

Affinché le funzionalità avanzate siano supportate anche dall'elenco delle teorie attive e non soltanto dall'elenco delle teorie disponibili, si deve effettuare il reingegnerizzazione dei partecipanti ActiveTheoryList e TheoryManager. In particolare, il TheoryManager dovrà essere modificato in modo tale che l'elenco delle teorie attive sia disaccoppiato rispetto all'insieme delle teorie caricate, mentre all'ActiveTheoryList dovranno essere aggiunti dei metodi che agevoleranno l'utilizzo delle funzionalità avanzate.

Come evidenziato durante la fase di analisi del problema, l'esecuzione di una operazione associata ad una funzionalità avanzata può richiedere la rimozione e il successivo reinserimento di una o più teorie nel motore tuProlog, anche se queste non sono direttamente coinvolte nella esecuzione dell'operazione[4]. Per limitare il più possibile l'onere di queste serie di rimozioni e inserimenti, la modalità di caricamento delle teorie nel motore tuProlog deve essere rivista: si dovrà avere una gestione delle teorie più flessibile, ove sia possibile differire il caricamento di una teoria.

Una valida soluzione potrebbe essere la seguente: inizialmente il TheoryManager caricherà direttamente le teorie logiche nel motore tuProlog, fino a quando non riceverà una richiesta di rimozione di una teoria logica dal motore; da quel momento in poi memorizzerà al suo interno le teorie logiche di cui verrà richiesto il caricamento, fino a che non verrà richiesta la risoluzione di una query o fino a che non avverrà la rimozione di tutte le teorie dall'elenco delle teorie attive. Infine, il TheoryManager dovrà occuparsi del caricamento relativo alle eventuali teorie memorizzate al suo interno, nel momento in cui verrà richiesta la risoluzione di una query. Così, l'utente potrà modificare l'elenco delle teorie attive a suo piacimento utilizzando le funzionalità avanzate senza appesantire la piattaforma con continue operazioni di inserimento e rimozione di teorie.

Perciò le responsabilità del TheoryManager saranno ampliate: infatti non dovrà soltanto effettuare il caricamento delle teorie logiche all'interno del motore tuProlog, ma anche tenere traccia delle teorie caricate e delle teorie ancora da

4 Ciò è dovuto alla limitazione del motore tuProlog relativa alla rimozione di teorie: una teoria può essere rimossa solamente se è l'ultima ad essere stata caricata tra quelle dotate di uno specifica etichetta; pertanto, se per eseguire una determinata operazione si deve rimuovere una qualunque teoria, allora può essere necessario rimuovere e successivamente reinserire delle ulteriori teorie.

caricare, mantenendo uno stato che stabilisca se una teoria logica per la quale viene richiesto il caricamento nel motore tuProlog debba essere caricata subito oppure in un secondo momento.

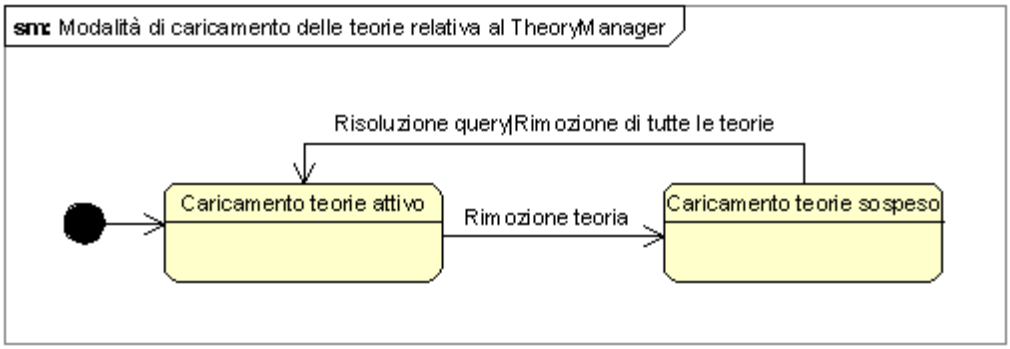


Figura 6.30: Diagramma di stato del TheoryManager relativamente alla modalità di caricamento delle teorie

Conseguentemente, internamente al TheoryManager devono essere aggiunti:

- una variabile booleana che rappresenti il suo stato;
- una lista contenente i nomi delle teorie caricate, necessaria al fine di permettere al TheoryManager la rimozione di teorie caricate ed evitare che esso debba mantenere una lista delle teorie caricate;
- una lista delle teorie da caricare, la quale verrà utilizzata se il TheoryManager avrà sospeso il caricamento diretto delle teorie logiche nel motore tuProlog.

Per rendere più semplice l'ActiveTheoryList, il TheoryManager dovrebbe essere dotato di appositi metodi che consentano di aggiungere o rimuovere teorie specificando un indice; quindi l'ActiveTheoryList sarà modificato in modo tale che i metodi esistenti utilizzino i nuovi metodi indicizzati del TheoryManager.

Le funzionalità da aggiungere all'ActiveTheoryList sono:

- rimozione di una qualunque teoria[5];
- sostituzione di una teoria con una nuova[6];

5 Il metodo relativo a questa funzionalità era già stato implementato nel secondo prototipo, ma in realtà eseguiva la cancellazione dell'ultima teoria contenuta nell'elenco delle teorie attive dotata di una specifica etichetta.

6 Il metodo relativo a questa funzionalità era già stato implementato nel secondo prototipo, ma la sua implementazione si limita a lanciare un'eccezione al fine di segnalare che la relativa funzionalità non è supportata.

- aggiunta di una teoria in una posizione qualunque dell'elenco delle teorie attive[7].

Inoltre, all'ActiveTheoryList devono essere aggiunte due importanti caratteristiche: la possibilità di indicare lo stato di attivazione di ogni teoria e l'indicazione relativa alla tipologia di query (vedi sezione 3.2). Dato che l'ActiveTheoryList è un *JavaBean*, per dotarlo di queste funzionalità verranno aggiunte due nuove proprietà, denominate *activations* e *mode*. La prima sarà una proprietà indicizzata costituita da un array di valori booleani, ai quali corrisponderà lo stato di attivazione di ciascuna teoria, mentre la seconda sarà una proprietà semplice costituita da una enumerazione appositamente definita in grado di identificare la tipologia di query considerata dall'utente. Ovviamente l'ActiveTheoryList dovrà essere in grado di emettere appositi eventi di segnalazione relativi ai cambiamenti di tali proprietà e quindi anche di permettere la registrazione/cancellazione dei relativi listener.

A seconda della tipologia di query indicata e dello stato di attivazione di ciascuna teoria, l'ActiveTheoryList dovrà caricare o meno le varie teorie attive nel motore tuProlog tramite il TheoryManager. Quindi i possibili casi sono:

- *and query*: lo stato di attivazione indica le query effettivamente da caricare; l'ActiveTheoryList indica al TheoryManager tutte le teorie da utilizzare;
- *or query*: lo stato di attivazione indica le query da utilizzare in successione; l'ActiveTheoryList indica al TheoryManager soltanto la prima teoria da utilizzare;
- *not query*: lo stato di attivazione indica le query che non devono essere caricate; l'ActiveTheoryList indica al TheoryManager tutte le teorie da utilizzare.

La tipologia *and query* è quella predefinita che fino ad adesso è stata considerata per default, mentre la tipologia *not query* semplicemente ribalta la logica della proprietà *activations*. La tipologia *or query* è un caso a parte: l'ActiveTheoryList carica attraverso il TheoryManager soltanto la prima tra le teorie il cui corrispondente valore booleano nella proprietà activations è *true*; al termine della

7 Se la posizione di inserimento della nuova teoria è maggiore della lunghezza dell'elenco delle teorie deve essere sollevata un'eccezione.

risoluzione di una query tale valore diventa false, in modo tale che nella successiva esecuzione la teoria caricata sarà quella successiva. Tale accorgimento consentirà di effettuare la risoluzione di una query utilizzando tutte le teorie attive indicate dall'utente in successione.

Infine, il cambiamento di tipologia di query nell'ActiveTheoryList comporta la rimozione di tutte le teorie logiche dal motore tuProlog e il ricaricamento delle teorie attive, dipendentemente dalla nuova tipologie di query scelta e dallo stato di attivazione di ciascuna di esse.

6.2.5 Reingegnerizzazione: *ActiveTheoryListView e ActiveTheoryListNode*

L'ActiveTheoryListView sarà modificata in modo tale che le icone associate a ciascuna rappresentazione relativa ad una teoria attiva siano in grado si rappresentare il valore booleano della proprietà *activations*[8] con indice corrispondente a quello della teoria. Inoltre, verrà modificata la relativa inner-class che realizza il listener degli eventi del mouse, al fine di rilevare anche i click sulle icone delle rappresentazioni delle singole teorie logiche, consentendo all'utente di indicare se includere o escludere una teoria logica dalla valutazione di una query. Quando l'utente cliccherà su una di queste icone, questa dovrà cambiare in modo tale da rappresentare chiaramente il nuovo stato di attivazione.

L'ActiveTheoryListNode dovrà essere modificato adattandolo ai cambiamenti effettuati all'ActiveTheoryList, aggiungendo appositi metodi per la gestione delle nuove proprietà e permettendo all'utente di richiamare le azioni relative alle funzionalità avanzate.

6.2.6 Reingegnerizzazione: *QueryControlPanel*

Il QueryControlPanel attualmente integra tutti i componenti che realizzano la sua interfaccia grafica. Per agevolare l'aggiunta della funzionalità di indicazione delle teorie da utilizzare, il QueryControlPanel dovrà essere scomposto in quattro componenti:

- una toolbar contenente i pulsanti per la gestione della risoluzione di una query, la quale sarà denominata QueryToolbar;

8 Si potrebbe rappresentare una teoria attiva inclusa nella valutazione di una query con un'icona verde, mentre una teoria attiva esclusa dalla valutazione potrebbe essere rappresentata da un'icona rossa.

- un pannello contenente il campo di testo per l'inserimento delle query e la casella di selezione relativa alla temporaneità degli effetti di una query, il quale sarà denominato QueryInputPanel;
- un pannello contenente l'area di testo per la visualizzazione dei risultati di una query, che sarà denominato QueryOutputPanel;
- un pannello contenente sia l'ActiveTheoryListView, sia un radio button per la scelta dell'operatore logico da utilizzare per la selezione delle teorie, che sarà denominato ActiveTheoriesControlPanel.

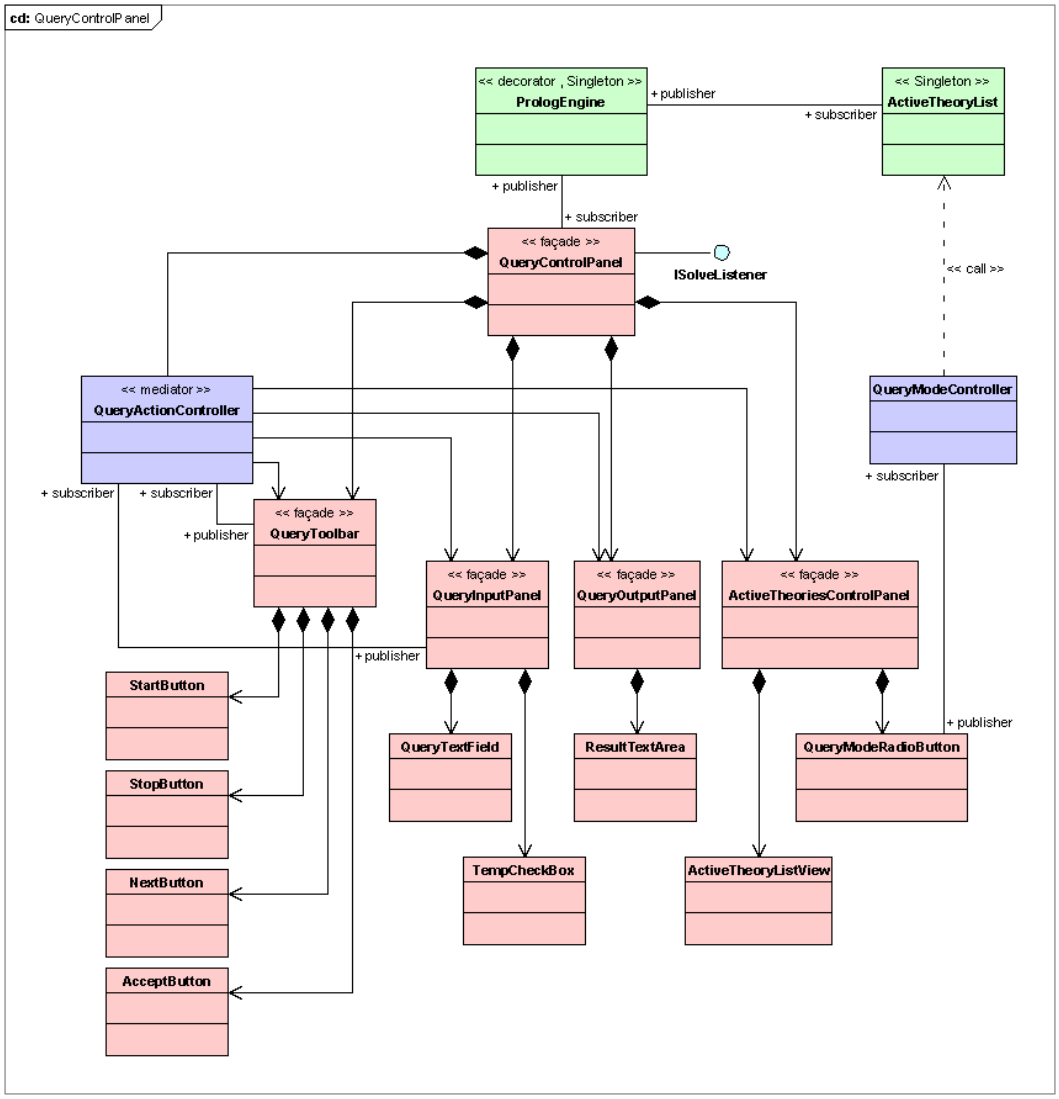
Cambiando la struttura del QueryControlPanel, si dovrà riprogettare il QueryActionController: infatti tale componente interagiva direttamente con i componenti grafici contenuti nel QueryControlPanel, ma questi verranno spostati all'interno delle nuove classi che implementeranno la QueryToolbar, il QueryInputPanel, il QueryOutputPanel e l'ActiveTheoriesControlPanel, pertanto dovrà interagire con queste nuove entità.

Affinché il QueryControlPanel sia maggiormente scalabile, i suoi quattro componenti potrebbero realizzare il design pattern *Façade* [GoF95]; in questo modo il QueryActionController ignorerà la loro composizione interna e potrà modificare lo stato dei loro widget grafici solamente tramite appositi metodi. Così i componenti del QueryControlPanel potranno evolversi separatamente dal QueryActionController.

Inoltre, il QueryActionController dovrà registrarsi come listener della QueryToolbar e del QueryInputPanel, in modo tale da poter ricevere eventi relativi alla pressione di pulsanti o alla immissione di una query e richiedere la notifica delle corrispondenti azioni agli opportuni listener registrati presso il QueryControlPanel.

Una volta ricevuto un evento di richiesta per l'esecuzione di una query, il QueryActionController deve prima verificare qual è la tipologia della query richiesta, recuperando tale informazione dall'ActiveTheoriesControlPanel. Nel caso in cui la tipologia della query richiesta sia una *and query* o una *not query*, il QueryActionController richiederà la risoluzione di un'unica query, assumendo il suo consueto comportamento, mentre nel caso di *or query* dovrà gestire una sequenza di risoluzioni relative alla stessa query, una per ogni teoria attiva selezionata dall'utente.

Per semplificare la logica interna del QueryActionController, il QueryControlPanel sarà registrato presso il PrologEngine in modo tale da poter ricevere eventi relativi all'inizio o alla terminazione della risoluzione di una query e inoltrarli al QueryActionController. Quest'ultimo utilizzerà gli eventi ad esso inoltrati per attivare o disattivare il QueryInputPanel e l'ActiveTheoriesControlPanel, impedendo all'utente di modificare la query inserita e la configurazione delle teorie attive durante una risoluzione.



Disegno 1: Figura 6.31: Diagramma delle classi che compongono il QueryControlPanel

Infine, lo stato del radio button contenuto nell'ActiveTheoriesControlPanel deve corrispondere al valore della proprietà *mode* relativa all'ActiveTheoryList, pertanto dovrà essere presente un listener registrato presso il radio button il cui compito sarà quello di aggiornare il valore relativo alla proprietà *mode* ogni volta

che l'utente cambierà l'impostazione del radio button; tale listener sarà denominato QueryModeController.

6.3 Progettazione del quinto prototipo

La progettazione del quinto prototipo si baserà su ciò che suggerisce di realizzare il piano di lavoro:

Quinto Prototipo: *integrazione della gestione delle librerie*.

Osservando la figura 3.4, relativa al diagramma dei casi d'uso dal titolo "Gestione delle librerie", si deduce che i casi d'uso da realizzare per fare ciò che suggerisce il piano di lavoro sono:

- *carica libreria nel motore tuProlog*;
- *rimuovi libreria dal motore tuProlog*;
- *aggiorna elenco delle librerie disponibili*.

Al fine di rendere maggiormente sintetica la progettazione delle collaborazioni, considereremo il caso d'uso denominato "*aggiorna elenco delle librerie disponibili*" come parte integrante sia del caso d'uso "*carica libreria nel motore tuProlog*", sia del caso d'uso "*rimuovi libreria dal motore tuProlog*".

6.3.1 Collaborazione: *carica libreria nel motore tuProlog*

Con riferimento all'analisi del problema, per poter realizzare il caso d'uso "*carica libreria nel motore tuProlog*" occorrono:

- un gestore delle librerie;
- un modello relativo all'elenco delle librerie disponibili;
- un componente grafico in grado di rappresentare l'elenco delle librerie disponibili e consentirne la gestione.

Inoltre, servirà anche un elemento di controllo in grado eseguire il caricamento di una libreria nel motore tuProlog.

I partecipanti sono:

- *LibraryManager*: il gestore delle librerie;
- *LibraryList*: elenco delle librerie disponibili;

- *LibraryListView*: vista che consente di visualizzare e gestire l'elenco delle librerie disponibili;
- *LoadLibraryAction*: azione che carica una libreria nel motore tuProlog.

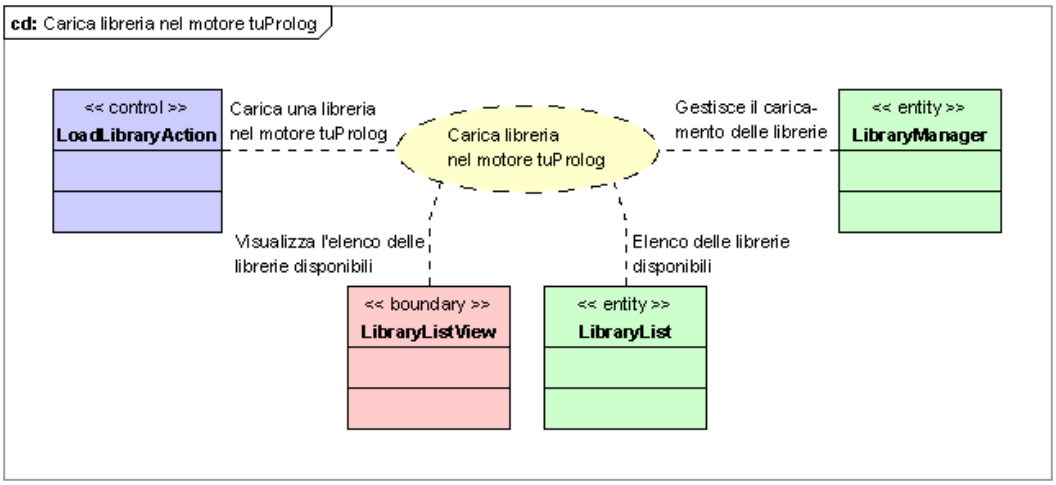


Figura 6.32: Collaborazione relativa al caso d'uso "carica libreria nel motore tuProlog"

Dai ruoli definiti per ogni partecipante, si determinano le relative responsabilità:

- *LibraryManager*: offre il servizio per il caricamento di una libreria nel motore tuProlog;
- *LibraryList*:
  - richiede al LibraryManager il caricamento di una libreria nel motore tuProlog;
  - notifica alla LibraryListView il caricamento di una libreria nel motore tuProlog;
- *LibraryListView*:
  - richiama l'azione per il caricamento di una libreria nel motore tuProlog;
  - riceve la notifica relativa all'avvenuto caricamento di una libreria;
- *LoadLibraryAction*: richiede alla *LibraryList* il caricamento di una libreria nel motore tuProlog.

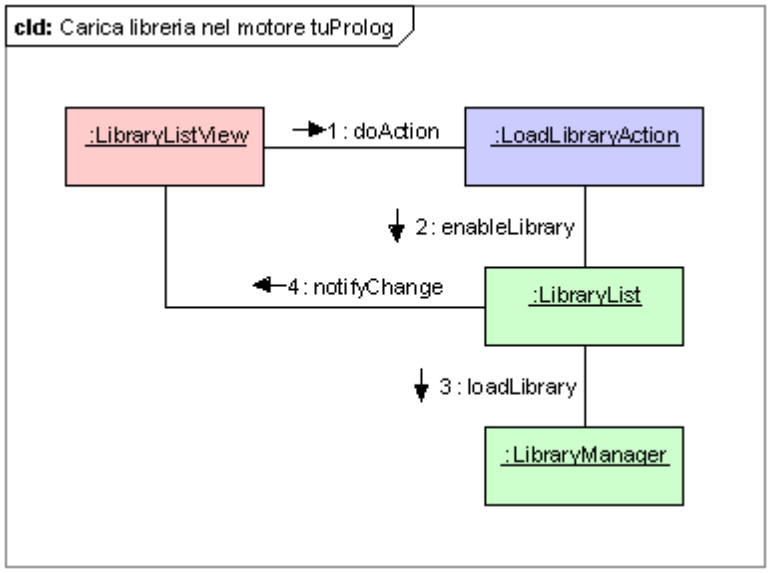


Figura 6.33: Diagramma relativo alla collaborazione "carica libreria nel motore tuProlog"

6.3.2 Collaborazione: *rimuovi libreria dal motore tuProlog*

Con riferimento all'analisi del problema, per poter realizzare il caso d'uso "*rimuovi libreria dal motore tuProlog*" occorrono:

- un gestore delle librerie;
- un modello relativo all'elenco delle librerie disponibili;
- un componente grafico in grado di rappresentare l'elenco delle librerie disponibili e consentirne la gestione.

Inoltre, servirà anche un elemento di controllo in grado eseguire la rimozione di una libreria dal motore tuProlog.

I partecipanti sono:

- *LibraryManager*: il gestore delle librerie;
- *LibraryList*: elenco delle librerie disponibili;
- *LibraryListView*: vista che consente di visualizzare e gestire l'elenco delle librerie disponibili;
- *UnloadLibraryAction*: azione che rimuove una libreria dal motore tuProlog.

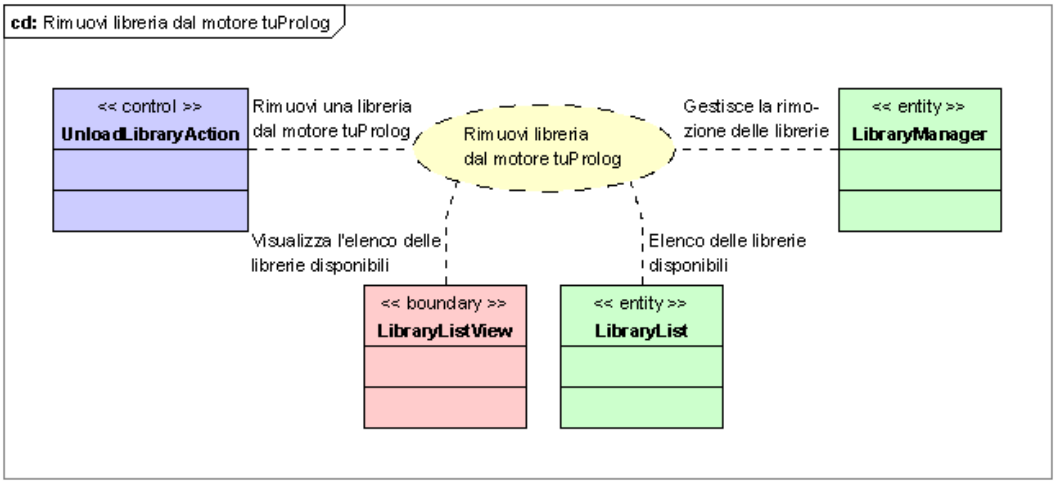


Figura 6.34: Collaborazione relativa al caso d'uso "rimuovi libreria dal motore tuProlog"

Dai ruoli definiti per ogni partecipante, si determinano le relative responsabilità:

- *LibraryManager*: offre il servizio per la rimozione di una libreria dal motore tuProlog;
- *LibraryList*:
  - richiede al LibraryManager la rimozione di una libreria dal motore tuProlog;
  - notifica alla LibraryListView la rimozione di una libreria dal motore tuProlog;

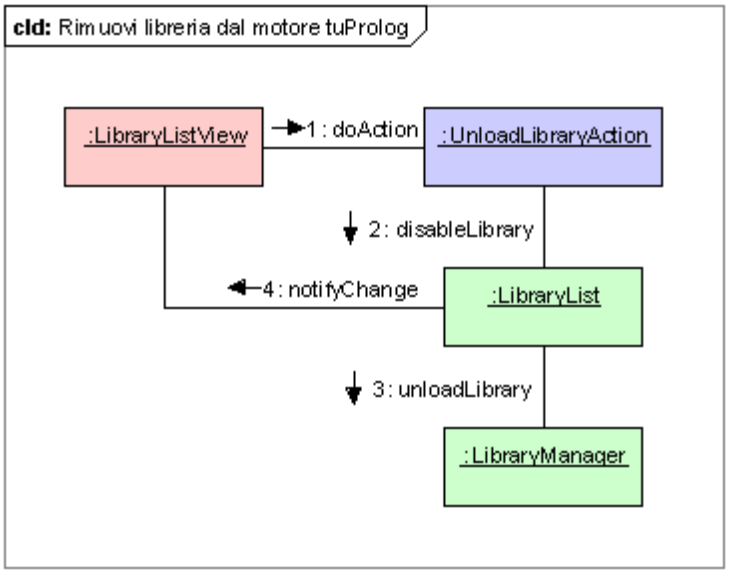


Figura 6.35: Diagramma relativo alla collaborazione "rimuovi libreria dal motore tuProlog"

- *LibraryListView*:
  - richiama l'azione per la rimozione di una libreria dal motore tuProlog;
  - riceve la notifica relativa all'avvenuta rimozione di una libreria;
- *UnloadLibraryAction*: richiede alla *LibraryList* la rimozione di una libreria dal motore tuProlog.

6.3.3 Interazione: *carica libreria nel motore tuProlog*

Tale interazione viene avviata con un comando da parte dell'utente, impartito tramite l'interfaccia grafica implementata dalla LibraryListView, la quale delegherà alla LoadLibraryAction la gestione dell'esecuzione dell'azione.

La LoadLibraryAction richiederà di effettuare il caricamento di una libreria nel motore tuProlog alla LibraryList, la quale richiederà la stessa azione al LibraryManager; soltanto in caso di successo la LibraryList notificherà l'avvenuta operazione al LibraryListView.

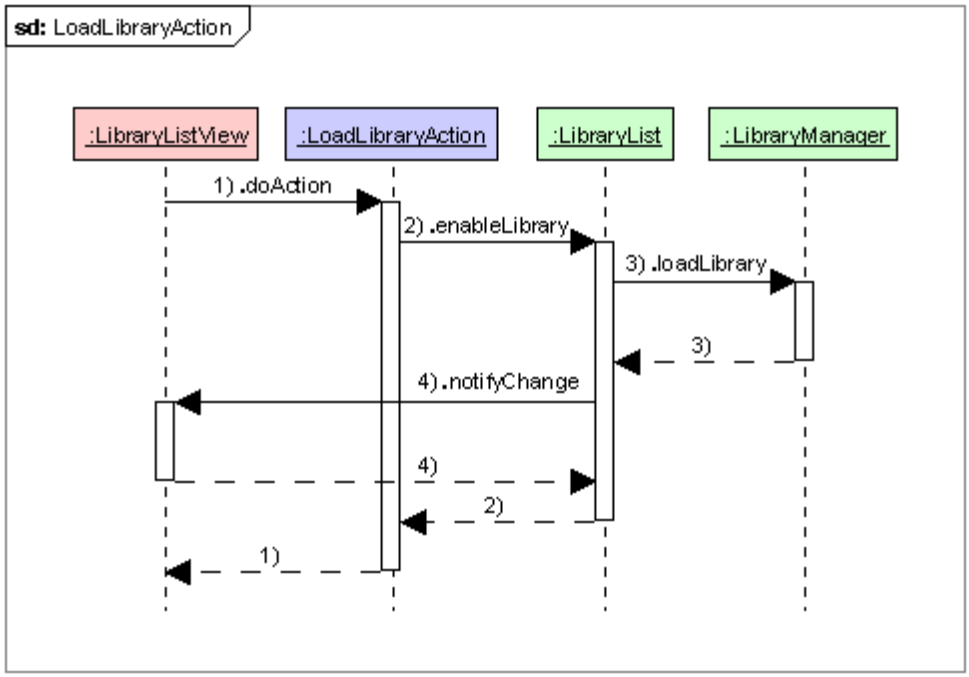


Figura 6.36: Diagramma di sequenza relativo all'interazione "carica libreria nel motore tuProlog"

6.3.4 Interazione: *rimuovi libreria dal motore tuProlog*

Tale interazione viene avviata con un comando da parte dell'utente, impartito tramite l'interfaccia grafica implementata dalla LibraryListView, la quale delegherà alla UnloadLibraryAction la gestione dell'esecuzione dell'azione.

La UnloadLibraryAction richiederà di effettuare la rimozione di una libreria dal motore tuProlog alla LibraryList, la quale richiederà la stessa azione al LibraryManager; soltanto in caso di successo la LibraryList notificherà l'avvenuta operazione al LibraryListView.

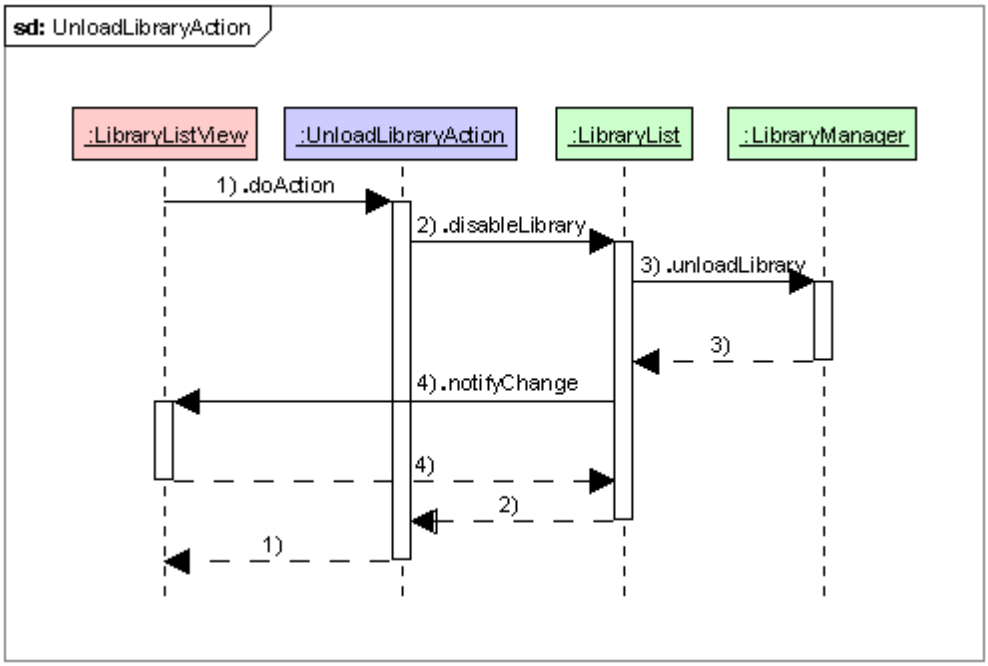


Figura 6.37: Diagramma di sequenza relativo all'interazione "rimuovi libreria dal motore tuProlog"

6.3.5 Progettazione dei partecipanti: *LibraryManager e LibraryList*

Il LibraryManager è uno dei tre componenti che costituiscono il nucleo dello strumento di supporto; esso sarà l'unico componente ad offrire i servizi di caricamento e rimozione di una libreria dal motore tuProlog. Ovviamente per svolgere la sua attività dovrà essere in grado di accedere al motore tuProlog vero e proprio, quindi si ripropone nuovamente la situazione riscontrata con il TheoryManager durante la progettazione del secondo prototipo (vedi sezione 5.1.19). Conseguentemente si adotterà una soluzione analoga: il LibraryManager sarà considerato parte del PrologEngine, e a quest'ultimo sarà affidata la responsabilità di creare il LibraryManager, permetterne il recupero con un

apposito metodo e passare al LibraryManager il riferimento del motore tuProlog vero e proprio. Infine, il LibraryManager dovrà avere un'interfaccia tale da consentire la registrazione/cancellazione presso il motore tuProlog dei listener di eventi relativi alla gestione delle librerie.

La LibraryList è il componente che realizza l'elenco delle librerie tuProlog disponibili all'interno dell'ambiente; i servizi offerti dalla LibraryList agli altri componenti sono due: attivazione e disattivazione di una libreria nella valutazione di una query. Ovviamente l'attivazione di una libreria comporterà il caricamento di questa nel motore tuProlog tramite il LibraryManager, mentre la disattivazione ne comporterà la rimozione. Attivando o disattivando una libreria, la LibraryList cambierà il suo stato interno; in caso di eccezione durante l'attivazione o la disattivazione di una libreria, la LibraryList non cambierà il suo stato interno e notificherà il problema tramite l'ErrorManager.

Quindi la LibraryList svolgerà il ruolo di *adapter* del LibraryManager: infatti i componenti del plug-in, interessati a gestire le librerie caricate, utilizzeranno solamente la LibraryList e non il LibraryManager. Dunque è possibile realizzare la LibraryList in modo tale che sia più adeguata all'ambiente NetBeans; pertanto, analogamente all'ActiveTheoryList e all'AvailableTheoryList, potrà essere realizzata sotto forma di *JavaBean*.

Le proprietà di cui sarà dotata la LibraryList dovranno descrivere le librerie disponibili e il loro stato di attivazione, quindi saranno delle proprietà indicizzate rappresentate rispettivamente da un array di librerie tuProlog e un array di valori booleani; queste proprietà saranno denominate *libraries* e *activations*. Le operazioni di caricamento e rimozione di una libreria dal motore tuProlog comporteranno soltanto la variazione del corrispondente valore booleano della proprietà indicizzata *activations*. L'aggiunta di una libreria alla LibraryList dovrà avvenire solamente nel momento in cui verrà installato un modulo contenente una nuova libreria tuProlog nell'ambiente NetBeans; analogamente, all'atto della disattivazione di un modulo contenente una libreria tuProlog dovrà seguire la rimozione di questa dalla LibraryList[9]. Le librerie tuProlog aggiunte

9 Visto che l'obiettivo di questa tesi è la realizzazione di un primo prototipo, non si è proceduto ad esplorare i meccanismi dell'ambiente NetBeans che consentono di eseguire delle azioni all'atto della installazione o disattivazioni di moduli, dato che si prevedeva di utilizzare tali funzionalità all'interno di un prototipo piuttosto avanzato; si veda la classe `org.openide.modules.ModuleInstall` relativa alle NetBeans API [NbAPI].

dinamicamente dovranno essere inizialmente non attive, ovvero non dovranno essere caricate nel motore tuProlog nel momento in cui saranno inserite nell'ambiente, al fine di evitare l'accidentale caricamento di una libreria installata proprio durante la risoluzione di una query; le uniche librerie attive non appena inserite nella LibraryList saranno quelle predefinite, ovvero la *BasicLibrary*, la *IOLibrary*, la *ISOLibrary* e la *JavaLibrary*. Infine la LibraryList deve essere in grado di notificare i cambiamenti relativi alle sue proprietà ad eventuali listener registrati.

Le librerie caricate nel motore tuProlog potranno essere aggiunte o rimosse dinamicamente, sia dalla LibraryList tramite gli appositi metodi, sia dal motore tuProlog eseguendo appositi predicati durante la risoluzione di una goal. Al fine di poter aggiornare lo stato di attivazione delle librerie disponibili durante la risoluzione di una query, si dovrà registrare la LibraryList come listener degli eventi di gestione delle librerie presso il LibraryManager, ma soltanto durante la risoluzione di una query, altrimenti riceverebbe anche le notifiche delle operazioni che essa effettua tramite il LibraryManager. Per raggiungere questo ultimo obiettivo, la LibraryList dovrà registrarsi come listener di SolveEvent presso il PrologEngine; quando riceverà un evento che segnalerà l'inizio di una risoluzione dovrà registrarsi come listener presso il LibraryManager, quando riceverà un evento che segnalerà il termine di una risoluzione dovrà cancellarsi come listener dal LibraryManager. Inoltre, per ripristinare gli stati di attivazione delle librerie precedenti alla risoluzione di una query temporanea, il LibraryManager dovrà memorizzare i nomi delle librerie caricate prima dell'inizio di una risoluzione; nel caso in cui non verranno confermati i cambiamenti apportati dall'esecuzione di una query, il LibraryManager dovrà confrontarli con quelli delle librerie effettivamente caricate al termine della risoluzione al fine di individuare le singole librerie che necessiteranno di una operazione di ripristino, che consisterà nel caricamento o nella rimozione della libreria considerata. Per riuscire in tale intento, il LibraryManager dovrà essere dotato di due metodi a visibilità limitata, i quali verranno utilizzati dal PrologEngine per segnalare ad esso l'inizio e il termine della dimostrazione di un goal, indicando la conferma dei cambiamenti o meno. Per garantire che la LibraryList riceva le notifiche relative ad eventuali operazioni di ripristino di librerie, il PrologEngine dovrà prima effettuare la segnalazione di risoluzione conclusa al LibraryManager e soltanto dopo notificare i listener relativi agli eventi SolveEvent.

L'unico componente del plug-in che utilizzerà il LibraryManager sarà la LibraryList, pertanto, affinché il LibraryManager non possa essere creato o utilizzato da altri componenti del plug-in, verrà adottata una soluzione analoga a quella del caso riguardante il TheoryManager (vedi sezione 5.1.19): il metodo del PrologEngine che restituisce il LibraryManager e la classe che lo implementa dovranno avere un livello di visibilità limitato. Inoltre, la LibraryList realizzerà il design pattern *Singleton* [GoF95].

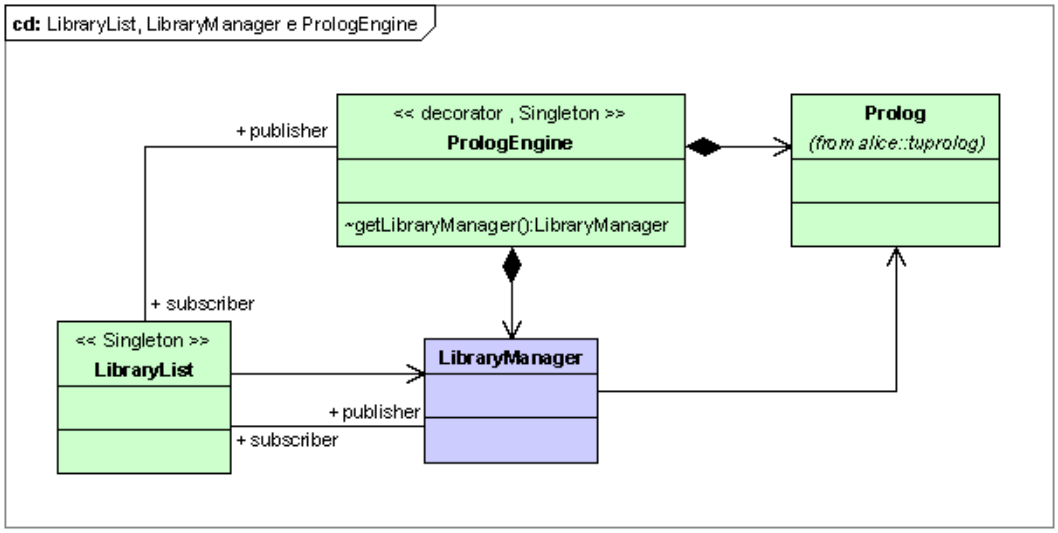


Figura 6.38: Diagramma relativo alle classi LibraryList, LibraryManager e PrologEngine

6.3.6 Progettazione dei partecipanti: *LibraryListView*

La LibraryListView realizzerà la vista relativa all'elenco delle librerie disponibili, visualizzando una serie di widget rappresentanti le singole librerie. Quindi si potrebbe implementare la LibraryListView sotto forma di *explorer view*; essa sarà caratterizzata dal nodo principale LibraryListNode, costituito da un elenco di nodi foglia LibraryNode.

Il LibraryListNode incapsulerà l'elenco delle librerie disponibili, mentre i vari LibraryNode incapsuleranno le singole librerie tuProlog; dato che il contenuto dell'elenco delle librerie disponibili può variare, il LibraryListNode deve essere registrato come listener della LibraryList. Relativamente al cambiamento dello stato di attivazione di una libreria, sarà la LibraryList ad avvisare il LibraryListNode a proposito di tale cambiamento, il quale provvederà ad aggiornare il nodo corrispondente alla libreria protagonista del cambiamento.

La LibraryListView dovrà essere inserita all'interno di una scheda del PrologTopComponent, pertanto sarà inserita in un apposito pannello: il

LibraryControlPanel; questo non soltanto conterrà la LibraryListView, ma incapsulerà anche il nodo radice della LibraryListView. Analogamente al QueryControlPanel, il LibraryControlPanel dovrà nascondere la sua complessità interna, pertanto realizzerà il design pattern *Façade* [GoF95].

Infine, la LibraryListView dovrà essere disattivata durante la risoluzione di una query, perché non ha senso che l'utente possa cambiare lo stato di attivazione di una libreria durante la valutazione di un goal. Pertanto, la LibraryListView dovrà essere registrata come listener presso il PrologEngine al fine di ricevere gli eventi SolveEvent, in modo tale da disattivarsi automaticamente quando riceverà una notifica di inizio risoluzione e riattivarsi quando riceverà una notifica di terminazione della risoluzione di una query.

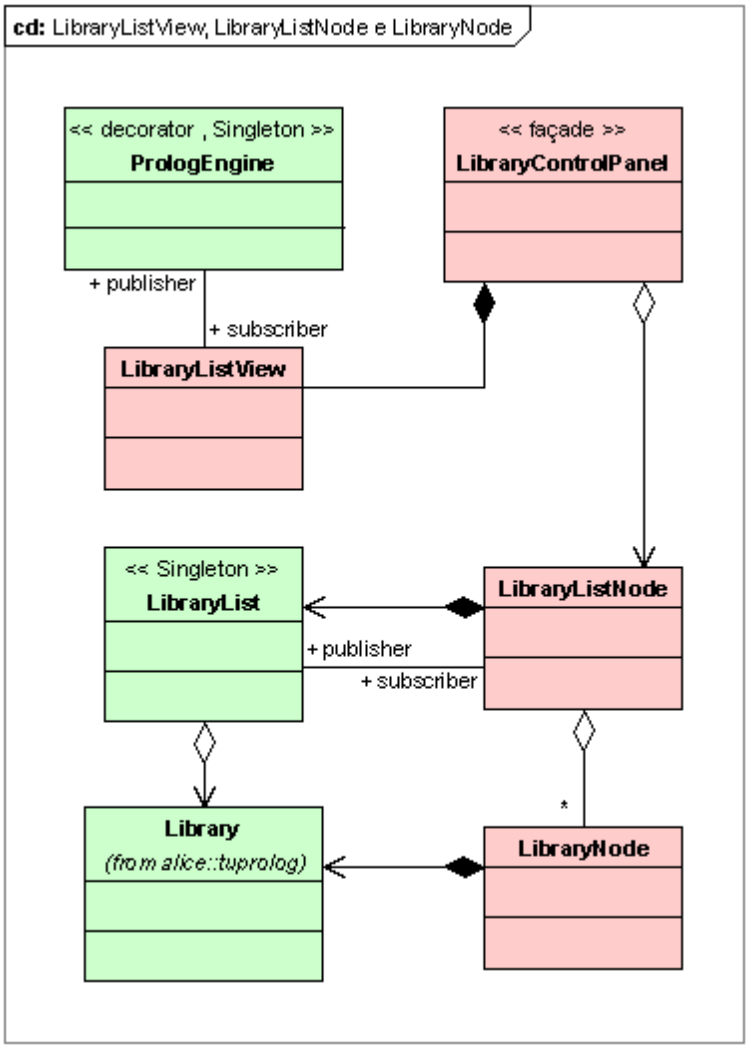


Figura 6.39: Diagramma relativo alle classi LibraryListView, LibraryListNode e LibraryNode

6.3.7 Progettazione dei partecipanti: *LoadLibraryAction* e *UnloadLibraryAction*

Le azioni di gestione delle librerie tuProlog sono soltanto due e si riferiscono alle singole librerie:

- *azione di attivazione di una libreria*: include una specifica libreria nella valutazione di una query, caricandola nel motore tuProlog;
- *azione di disattivazione di una libreria*: esclude una specifica libreria dalla valutazione di una query, rimuovendola dal motore tuProlog;

Esse devono essere mutuamente esclusive, ovvero se una libreria è attiva non dovrebbe essere possibile eseguire l'azione di attivazione su di essa, ma solamente quella si disattivazione, viceversa se una libreria non è attiva. Quindi per innescare tale azioni, invece di ricorrere al consueto menù contestuale si potrebbe più semplicemente utilizzare l'icona associata ad ogni nodo relativo ad una libreria, analogamente al caso delle teorie attive (vedi sezione 6.2.5): cliccando sull'icona si cambia lo stato di attivazione di una libreria. Per poter riuscire in tale intento si dovrà aggiungere alla LibraryListView un apposito listener relativo agli eventi del mouse alla LibraryListView, in grado di intercettare i click del mouse al di sopra delle icone e conseguentemente segnalare questo evento al corrispondente nodo.

In questo modo le funzioni relative ai due partecipanti LoadLibraryAction e UnloadLibraryAction saranno implementate da un singolo cookie associato ad un LibraryNode, denominato LibraryCookie, i cui metodi verranno utilizzati dal listener del mouse incapsulato nella LibraryListView.

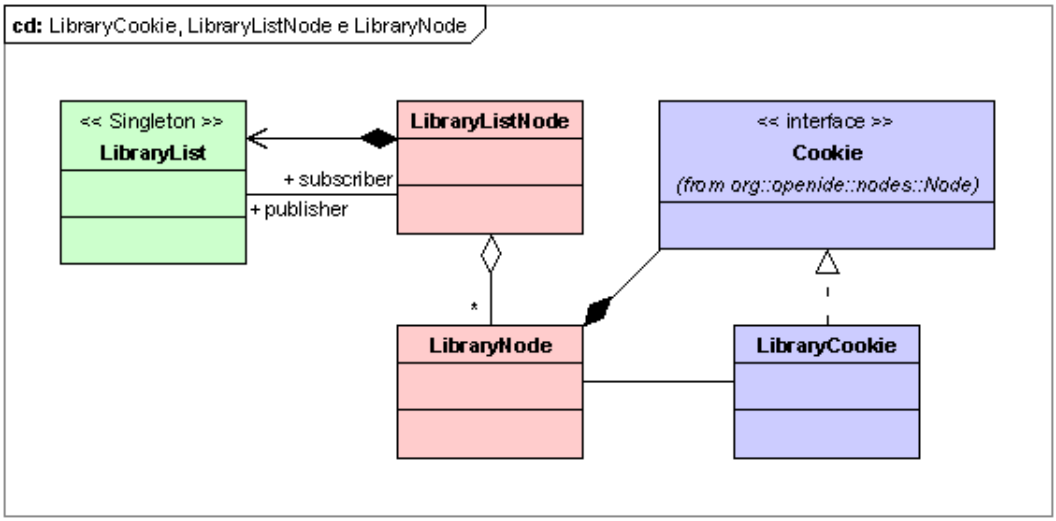


Figura 6.40: Diagramma relativo alle classi LibraryCookie, LibraryListNode e LibraryNode

6.4 Indicazioni sulla realizzazione del sesto prototipo

Come per tutti gli altri prototipi, la progettazione del sesto prototipo si baserà su ciò che suggerisce di realizzare il piano di lavoro:

Sesto Prototipo: *integrazione di un editor avanzato delle teorie, in grado di evidenziare la sintassi attraverso la colorazione del testo; integrazione della funzionalità di separazione di una teoria sia per l'elenco delle teorie attive, sia per l'elenco delle teorie disponibili; dovrà essere inclusa la possibilità di eseguire le custom query*.

Quindi la realizzazione di questo prototipo aggiungerà al plug-in soltanto due nuove caratteristiche: un editor più avanzato e l'ultima funzionalità avanzata che ancora non è stata implementata.

Per quanto riguarda l'editor avanzato, per poterlo implementare dovranno essere ben note le funzionalità tipiche dell'ambiente NetBeans relativamente alla colorazione e al riconoscimento del testo contenuto in un editor; non si è proceduto alla esplorazione di queste perché era evidente che sarebbero state applicate solamente in un prototipo avanzato, quindi oltre gli obiettivi di sviluppo relativi a questa tesi.

Relativamente alla funzionalità di separazione di una teoria, si dovrà sviluppare un'apposita interfaccia grafica che consenta di evidenziare gli effetti relativi al particolare modo prescelto per separare una teoria, ad esempio visualizzando in un'apposita area di testo il contenuto della teoria, in modo tale che le clausole che apparterranno alla stessa sotto-teoria siano colorate allo stesso modo e che colori distinti identifichino sotto-teorie distinte. Infatti si potrebbe separare una teoria manualmente scegliendo le singole clausole, oppure automaticamente secondo una specifica logica; ad esempio, una logica per la separazione automatica potrebbe generare una nuova teoria per ogni distinta intestazione di clausola. Inoltre, in caso di separazione manuale, l'utente deve essere in grado di assegnare liberamente i nomi delle teorie logiche che verranno generate dalla operazione di separazione. Quindi, la realizzazione dell'interfaccia grafica di supporto alla separazione delle teorie logiche deve essere attentamente progettata, al fine di rendere semplice tale operazione ed anche di renderne chiari gli effetti.

Infine, per poter eseguire le *custom query* dovrà essere realizzato un supporto composto sostanzialmente da un campo di testo che consentirà all'utente di

specificare l'espressione per la selezione delle teorie attive da utilizzare nella risoluzione di una query, e da un interprete in grado di riconoscere tale espressione e di indicare quali teorie attive includere nella valutazione e quali escludere, restituendo il nuovo valore della proprietà *activations* relativa all'ActiveTheoryList. Dovrà essere modificato l'ActiveTheoriesControlPanel, al fine di integrare nel radio button per la selezione del tipo di query anche la voce relativa alle *custom query*, ed aggiungere il campo di testo per l'inserimento dell'espressione per la selezione delle teorie attive; inoltre si dovrà trasformare il QueryModeController da semplice listener in un componente più complesso in grado di svolgere il ruolo di mediatore tra il radio button, il campo di testo relativo all'espressione per la selezione delle teorie attive, l'interprete in grado di indicare le teorie attive da utilizzare per la risoluzione di una query e l'ActiveTheoryList. Pertanto sarà opportuno che realizzi il design pattern *Mediator* [GoF95].

6.5 Sviluppi futuri

Durante la progettazione e realizzazione dello strumento di supporto per l'interprete tuProlog in ambiente NetBeans, sono state notate alcune direzioni preferenziali verso cui il sistema potrà evolvere. In particolare, è emerso che si potranno effettuare miglioramenti dal punto di vista dell'integrazione del plug-in nel suo ambiente; inoltre, parte delle classi che lo costituiscono potranno essere rese maggiormente riusabili.

6.5.1 Miglioramento relativo alla integrazione del plug-in con l'ambiente NetBeans

Durante le verifiche relative al corretto funzionamento del plug-in è stato riscontrato come il motore tuProlog non abbia la completa visibilità delle entità presenti a run-time nell'ambiente NetBeans. Infatti, per testare le capacità dello strumento di supporto si è provato a realizzare una teoria in grado di interagire con le entità presenti nell'ambiente, sfruttando i predicati e i funtori offerti dalla *JavaLibrary*: si è riusciti a interagire, peraltro limitatamente, soltanto con il componente *ErrorManager*[10].

Questo è dovuto al fatto che il *library wrapper module* relativo al motore tuProlog

10 Di fatto, attraverso il componente ErrorManager non si è stati in grado di visualizzare la consueta finestra in primo piano contenente il messaggio, ma si è riusciti soltanto a far comparire una icona lampeggiante all'interno della barra di stato relativa alla finestra principale dell'ambiente, la quale mostrava il messaggio soltanto cliccandoci sopra.

vero e proprio è in grado di accedere solamente alle classi relative ai moduli da cui dipende; la soluzione banale consisterebbe nel tentare di accedere a tutte le entità dell'ambiente, rendendo il modulo relativo al motore tuProlog dipendente da più moduli che sia possibile. Questa è improponibile perché sarebbe troppo a basso livello e non del tutto efficace; sarebbe meglio ricorrere ai meccanismi già presenti e appositamente progettati per estendere da una parte l'ambiente NetBeans e dall'altra le librerie tuProlog. Quindi, la soluzione maggiormente scalabile consiste nel creare un modulo in grado di aggiungere al plug-in una libreria tuProlog che aggiunga la funzionalità di accesso diretto al *System FileSystem* e ad altri componenti fondamentali di NetBeans.

In particolare, tale libreria dovrebbe aggiungere al motore tuProlog dei predicati in grado di:

- ricercare una entità nel *System FileSystem*;
- esplorare il *System FileSystem*;
- aggiungere e rimuovere entità al *System FileSystem*;
- visualizzare messaggi di errore tramite l'*ErrorManager*.

In questo modo sarà possibile creare delle teorie in grado di amministrare il NetBeans IDE oppure di coordinare i componenti di una applicazione basata sulla NetBeans Platform.

6.5.2 Miglioramento relativo alla organizzazione delle teorie logiche etichettate

La versione finale dello strumento di supporto per l'interprete tuProlog sarà in grado di memorizzare le singole teorie su diversi tipi di supporti per la persistenza, ma non sarà in grado di organizzare la persistenza di insiemi di teorie logiche. In pratica, un utente che avrà bisogno di memorizzare un set di teorie attive disposte secondo un preciso ordine dovrà prendersi la responsabilità di annotare la struttura dell'insieme di teorie attive utilizzato.

Per risolvere tale questione si potrebbe introdurre un nuovo tipo di file tale che permetta di definire un *insieme di teorie logiche ordinate*. Il contenuto di un file appartenente a questo nuovo tipo potrà essere strutturato in diversi modi:

- elenco delle coppie nome logico-nome fisico delle singole teorie nel giusto ordine;

- elenco strutturato delle teorie logiche, associando ad ognuna di esse nome logico e contenuto;
- elenco strutturato delle teorie logiche, associando ad ognuna di esse il nome logico e il contenuto oppure la coppia nome logico-nome fisico.

Nel primo caso un file rappresentante un insieme di teorie potrà essere realizzato da un *property file*, ovvero da un file testuale in cui ogni riga è formata da una coppia proprietà-valore; in tal caso ogni riga rappresenterà una teoria, ciascuna proprietà rappresenterà il nome logico e ciascun valore rappresenterà il nome fisico di una teoria. Nel secondo e terzo caso un file rappresentante un insieme di teorie sarà più complesso, pertanto è conveniente strutturarlo utilizzando XML, definendo anche uno schema di validazione.

Al fine di aggiungere al plug-in il supporto agli insiemi di teorie logiche ordinati, la vista relativa all'elenco delle teorie disponibili dovrà supportare anche gli insiemi di teorie ordinate e il pannello di controllo per la gestione della risoluzione di query dovrà essere dotato di due ulteriori funzionalità: l'inizializzazione dell'elenco delle teorie attive utilizzando un insieme di teorie ordinate e la creazione di un insieme di teorie ordinate basato sul contenuto dell'elenco delle teorie attive.

6.5.3 Introduzione di un nuovo tipo di progetto per l'ambiente NetBeans

Allo scopo di migliorare l'integrazione di tuProlog, si potrebbe corredare l'ambiente NetBeans di un nuovo tipo di progetto[11], denominato "*tuProlog Project*", le cui istanze consentirebbero di memorizzare in modo strutturato, all'interno di una serie di directory, sia insiemi di teorie logiche ordinate, sia singole teorie logiche memorizzate all'interno del supporto per la persistenza del progetto, sia riferimenti a teorie logiche memorizzate su supporti per la persistenza differenti da quello relativo al progetto, evidenziando i nomi logici e non i nomi fisici. Non appena un progetto di questo tipo verrà aperto dovrà importare automaticamente le singole teorie e gli insiemi di teorie ordinate che contiene all'interno dell'elenco delle teorie disponibili.

11 Un progetto in ambito NetBeans è sostanzialmente formato da un *file object* che rappresenta una directory memorizzata in uno specifico supporto per la persistenza, contenente l'insieme di risorse che lo costituiscono.

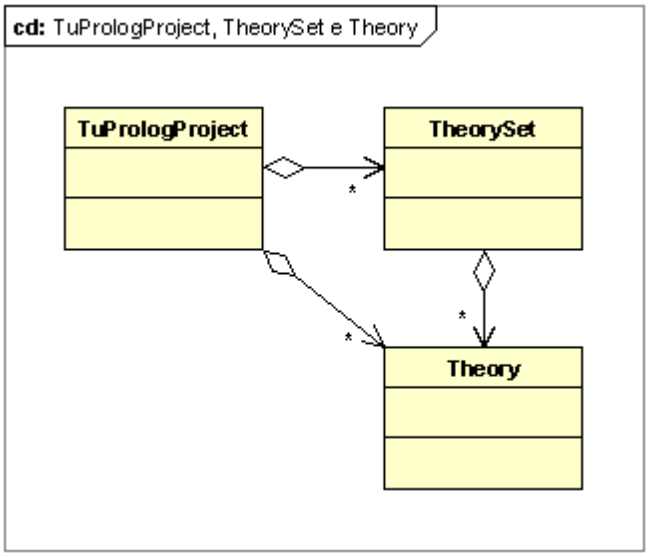


Figura 6.41: Architettura logica di un "tuProlog Project"

6.5.4 Riusabilità delle classi che costituiscono il nucleo del plug-in

Il nucleo del plug-in è costituito dalle classi che realizzano il PrologEngine, il TheoryManager e il LibraryManager, le quali consentono di gestire rispettivamente la risoluzione di query, le teorie caricate nel motore e le librerie utilizzate dal motore. Queste classi sono state progettate e implementate in modo tale da non utilizzare funzionalità o componenti tipici dell'ambiente NetBeans: in pratica si è tenuto conto della possibilità di renderle riusabili nell'ambito di progetti per l'integrazione del motore tuProlog in altri ambienti o framework. A seguito di una verifica relativa alla generalità delle loro caratteristiche, potrebbe essere opportuno spostare tali classi dal package `alice.tuprologx.netbeans` al package `alice.tuprologx`, dato che non sono state realizzate specificatamente per l'ambiente NetBeans.

Per aumentare le possibilità di riuso si potrebbero definire una serie di interfacce al fine di specificare il comportamento di ciascuna di queste classi. Queste saranno utili affinché le tre entità elencate precedentemente siano integrabili nell'ambito di framework o di ambienti basati sul principio dell'*Inversion of Control* (vedi Appendice A).

La definizione di tali interfacce potrà avvenire solamente dopo che lo sviluppo dello strumento di supporto per l'interprete tuProlog in ambiente NetBeans uscirà dalla fase prototipale, ovvero quando l'insieme dei metodi delle classi rammentate precedentemente sarà stabile.

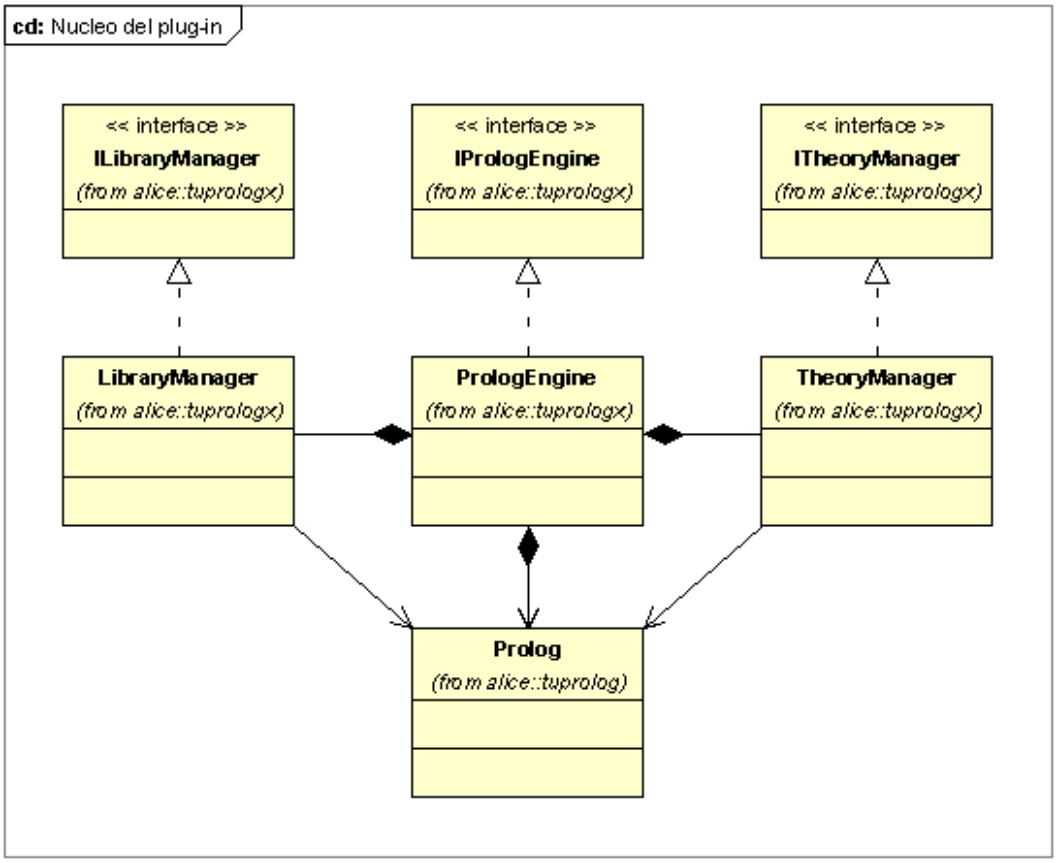


Figura 6.42: Diagramma relativo alle classi che costituiscono il nucleo del sistema, una volta applicato il miglioramento proposto

6.6 Questioni ancora aperte

Lo sviluppo del sistema è consistito essenzialmente nella realizzazione del plug-in attorno all'interprete tuProlog; per risolvere quei problemi dell'interprete in grado di compromettere gravemente le funzionalità del plug-in, si è dovuto procedere alla correzione del codice sorgente di tuProlog. Per altre problematiche, si è riusciti ad evitare di modificare il motore tuProlog, adottando soluzioni alternative piuttosto semplici che hanno consentito di aggirare il problema presente nell'interprete, ma hanno lasciato una questione aperta relativamente ad esso. Inoltre, sono emerse due anomalie che purtroppo non è stato possibile correggere, al fine di evitare modifiche tali da rischiare di compromettere la compatibilità di tuProlog con il software preesistente. Quindi la maggior parte delle questioni che sono rimaste ancora aperte riguardano l'interprete tuProlog, mentre soltanto una riguarda il plug-in relativamente alla percezione che l'utente potrebbe avere nell'uso delle teorie attive.

All'inizio della fase implementativa del primo prototipo è emersa una mancanza in

una particolare classe di tuProlog: la `alice.tuprolog.SolveInfo`. Essa rappresenta il risultato di una richiesta relativa alla risoluzione di un goal, ma è priva di indicazioni relativamente alla presenza di ulteriori soluzioni; per aggirare tale problema si è scelto di eseguire una callback sul PrologEngine al fine di verificare la presenza di eventuali ulteriori soluzioni. Per risolvere tale problema in una futura versione di tuProlog dovrebbe essere modificata la classe `SolveInfo`.

Inoltre, durante il collaudo del primo prototipo è stato notato che la funzionalità di svuotamento delle teorie caricate nel motore, implementata dal metodo `void clearTheory()` relativo alla classe `Prolog`, assume un comportamento anomalo: infatti, invece di eliminare tutte le teorie di cui è composta la teoria logica dinamica, semplicemente ne svuota le clausole, lasciando all'interno del motore le teorie logiche caricate completamente vuote. Un comportamento corretto consisterebbe nella completa rimozione di tutte le teorie logiche caricate nel motore e di tutte le relative clausole. Per non rischiare di effettuare modifiche tali da non rendere più compatibile il motore tuProlog con sistemi software già esistenti si è scelto di aggirare il problema, eseguendo la rimozione di tutte le teorie logiche caricate usando una sequenza di due comandi: impostazione di una teoria vuota e successiva rimozione di questa. In una futura versione di tuProlog tale comportamento anomalo dovrebbe essere corretto.

Nella fase di progettazione del secondo prototipo si è osservato che la classe `alice.tuprolog.TheoryEvent`, la quale rappresenta un evento relativo ad un cambiamento nella teoria logica complessiva, non dà indicazioni sulla particolare teoria logica protagonista del cambiamento, né sulla natura di questo; praticamente non si sa se una teoria è stata aggiunta o rimossa, se è stato eseguito un predicato *assert* o *retract* e in quale teoria logica è stato applicato. Questa mancanza di informazioni costringe l'ActiveTheoryList a richiedere al TheoryManager l'elenco delle teorie caricate, ogni volta che riceve un evento di questo tipo. Per risolvere tale problema in una futura versione di tuProlog dovrebbe essere reingegnerizzata la classe `TheoryEvent`, assieme a tutte quelle classi di cui ne fanno uso.

Durante il collaudo del secondo prototipo è emerso che i predicati *assert* e *retract* si comportano in modo anomalo: non operano sulle teorie etichettate ma su di una specie di "limbo" di teorie. Purtroppo la loro eventuale correzione avrebbe

comportato un cambiamento tale da non garantire la compatibilità del motore tuProlog con i sistemi software preesistenti, pertanto non si è proceduto alla risoluzione del problema. I predicati *assert* e *retract*, oltre a non funzionare correttamente, non consentono di specificare in quale teoria logica dovrebbero operare; se i due predicati assumessero un comportamento corretto, la teoria in cui dovrebbero operare sarebbe solamente quella denominata "*default*". Per risolvere questo problema dovrebbero essere introdotti i seguenti nuovi predicati:

- `assert/2(@clause,+string)`
  `assert(Clause,TheoryName)` è vera, con l'effetto collaterale di aggiungere `Clause` in fondo all'ultima teoria etichettata con `TheoryName`; nel caso in cui non esistano teorie etichettate con `TheoryName`, l'effetto collaterale consiste nella creazione di una nuova teoria logica con nome `TheoryName` dotata soltanto della clausola `Clause`;
- `retract/2(@clause,+string)`
  `retract(Clause,TheoryName)` è vera se esiste una teoria etichettata con `TheoryName` e che contiene almeno una clausola che unifica con `Clause`, con l'effetto collaterale di rimuovere l'ultima delle clausole che unificano con `Clause`; è in grado di restituire soluzioni multiple come il predicato `retract/1`, ed allo stesso modo è *backtrackable*.

Inoltre, al fine di ampliare le possibilità di intervento sulle teoria logica dinamica, si potrebbero aggiungere i seguenti ulteriori predicati:

- `retract_all/2(@clause,+string)`
  `retract_all(Clause,TheoryName)` è vera se esiste una teoria etichettata con `TheoryName` e che contiene almeno una clausola che unifica con `Clause`, con l'effetto collaterale di rimuovere tutte le clausole che unificano con `Clause` dall'ultima teoria etichettata con `TheoryName`;
- `retract_all/1(@clause)`
  `retract_all(Clause)` assume lo stesso comportamento del predicato `retract_all/2(@clause,+string)`, ma considerando "*default*" come etichetta della teoria su cui agire;
- `retract_all_db/1(@clause)`
  `retract_all_db(Clause)` è vera se esiste nella teoria logica dinamica

una clausola che unifica con `Clause`, con l'effetto collaterale di rimuovere tutte le clausole che unificano con `Clause` dalla teoria logica dinamica.

Questi nuovi predicati consentirebbero di eseguire azioni mirate all'interno delle singole teorie logiche, aumentando il livello di flessibilità introdotto dalle teorie etichettate. Un tale aggiornamento però è solo possibile rivedendo in profondità alcuni meccanismi interni del motore tuProlog.

Vi è infine, la questione aperta relativa al plug-in. Durante l'analisi del problema è stato definito il concetto di teoria attiva, ovvero una teoria che l'utente considera utilizzabile per la risoluzione di una query, per la quale può deciderne l'inclusione o l'esclusione dalla valutazione (vedi sezione 3.2). Nel quarto prototipo sono stati progettati il meccanismo e l'interfaccia grafica che consentono di evidenziare e di far scegliere all'utente le teorie attive incluse o escluse dalla valutazione. Utilizzando il quarto prototipo del plug-in o una realizzazione successiva, un utente il quale non abbia ben presente che una teoria esclusa dalla valutazione non è caricata nel motore tuProlog, si aspetterebbe che sia possibile la rimozione, tramite l'apposito predicato *remove_theory*, di una teoria attiva esclusa dalla valutazione. L'utente otterrebbe invece come risultato un fallimento o la rimozione dell'ultima teoria attiva omonima inclusa nella valutazione (che potrebbe non essere l'ultima dell'elenco delle teorie attive a riportare il nome specificato), interpretandolo come un comportamento anomalo.

La questione appena sollevata può essere considerata un problema o meno, a seconda dell'idea di considerare l'elenco delle teorie attive una struttura manipolabile da tuProlog o meno; pertanto si potrebbe prospettare una soluzione flessibile che consenta di attivare o disattivare la possibilità di manipolare l'elenco delle teorie attive da parte del motore tuProlog. Per riuscire in tale intento si potrebbe estendere il motore tuProlog con una apposita libreria che consenta la gestione dell'elenco delle teorie attive tramite appositi predicati: installando tale libreria l'utente potrà anche rimuovere una teoria attiva esclusa dalla valutazione; non installandola non lo potrà fare, ottenendo il comportamento standard.

Conclusioni

L'obiettivo principale di questa tesi era la realizzazione di un prototipo in grado di implementare un valido sistema di supporto alle teorie etichettate per l'interprete tuProlog in ambiente NetBeans. Prima di procedere allo sviluppo del sistema, è stata effettuata un'ampia analisi del problema, in grado di delineare gli obiettivi dell'intera fase realizzativa e non soltanto quelli relativi ad un primo prototipo. Il piano di lavoro conseguente all'analisi prevedeva la scomposizione del processo di sviluppo del plug-in in sei prototipi; il processo di sviluppo si è fermato al secondo prototipo, perché questo è stato considerato dotato di un insieme sufficientemente ampio di funzionalità tale da costituire una prima valida realizzazione del plug-in.

La realizzazione del primo prototipo ha consentito di delineare molte caratteristiche dell'architettura del plug-in, sia per quanto riguarda i componenti che costituiscono il nucleo del sistema, sia per i componenti che costituiscono l'interfaccia grafica e sia per i componenti che realizzano le azioni di controllo che consentono alle varie entità del sistema di interagire.

La realizzazione del secondo prototipo ha aggiunto ulteriori caratteristiche all'architettura del plug-in, consentendo di sfruttare l'innovativa gestione delle teorie etichettate del motore tuProlog, la quale permette una maggiore flessibilità nella manipolazione della teoria dinamica complessiva.

E' interessante notare che la struttura del nucleo del plug-in ottenuta con lo sviluppo del secondo prototipo sia simile a quella di tuProlog: infatti entrambi i sistemi sono dotati di un componente che gestisce la risoluzione di goal, ovvero il motore, e di un componente che gestisce le teorie logiche utilizzate dal motore. Nell'architettura logica dello strumento di supporto, tra le entità che formano il nucleo del sistema è presente un ulteriore componente oltre ai due precedentemente citati: il gestore delle librerie. Effettuando un confronto con tuProlog si nota che in esso tale elemento è mancante: infatti, il caricamento e la rimozione di una libreria avviene tramite il componente che implementa il motore

inferenziale. Questo potrebbe indicare l'opportunità di una futura evoluzione di tuProlog nella direzione in cui la struttura dei suoi componenti fondamentali comprenda anche un gestore delle librerie a se stante.

La principale difficoltà che si è dovuto affrontare inizialmente è consistita nel cercare di ottenere una visione d'insieme dell'ambiente NetBeans: infatti, la documentazione di questo sistema aperto ha un taglio eccessivamente "tutorialistico", oltre ad essere molto frammentata e a contenere alcune lacune ed errori. Un'ulteriore difficoltà è consistita nel riuscire a creare dei collegamenti tra le caratteristiche dei vari componenti del sistema; a volte non è stato chiaro neanche il comportamento di alcuni di essi, soprattutto quelli fondamentali per l'ambiente. Addirittura per capire certe anomalie si è dovuto analizzare il codice sorgente di una classe relativa ad un componente di NetBeans.

Durante la realizzazione dello strumento di supporto per l'interprete tuProlog, sono emerse una serie di problematiche impreviste. I principali problemi emersi riguardavano il motore tuProlog, quindi sono state valutate una serie di sue modifiche al fine di eliminare i problemi riscontrati; alcune modifiche non sono state effettuate perché avrebbero cambiato l'architettura del motore tuProlog o la signature di alcuni metodi, andando a compromettere la compatibilità dell'interprete verso i sistemi software preesistenti. Questa è la motivazione che ha impedito la risoluzione di alcune questioni che sono rimaste ancora aperte.

Le difficoltà riscontrate durante lo sviluppo dello strumento di supporto hanno riguardato soltanto l'esplorazione dell'ambiente NetBeans e la risoluzione dei problemi relativi all'interprete tuProlog. Grazie al particolare processo di sviluppo scelto e all'utilizzo dei test unitari, non sono emersi particolari problemi nella costruzione del plug-in, permettendo la realizzazione di un artefatto di buona qualità.

Per ottenere un plug-in completo dell'interprete tuProlog si deve ancora realizzare ed aggiungere un supporto alle teorie logiche disponibili, un supporto per la gestione delle librerie per il motore tuProlog, un editor di teorie logiche e una gestione avanzata delle teorie attive. Queste caratteristiche mancanti sono state delineate nella progettazione degli ulteriori prototipi, mentre negli sviluppi futuri sono state proposte modifiche ed estensioni tali da rendere maggiormente riusabili i componenti che costituiscono il nucleo del sistema e in grado di aumentare l'integrazione del motore tuProlog con l'ambiente NetBeans.

Appendice A

Inversion of Control

In questa appendice si parlerà del principio dell'*Inversion of Control* e dei design pattern basati su di esso; in particolare si parlerà dei design pattern che caratterizzano l'ambiente NetBeans, proprio perché fondati su tale principio.

A.1 Inversion of Control e il design pattern Dependency Injection

Il design pattern che caratterizza maggiormente i componenti dell'ambiente NetBeans è il *Service Locator* [Fow04]; come già accennato nel Capitolo 2, tale design pattern rappresenta un'applicazione del principio denominato *Inversion of Control*, noto anche come *Hollywood Principle* ("Don't call us, we'll call you.").

L'intento dell'*Inversion of Control* è quello di isolare il controllo, non soltanto di invertirlo. Secondo tale principio, gli oggetti non devono essere realizzati in modo tale che detengano il controllo, ma in modo tale che si aspettino di essere controllati; per riuscire in tale scopo, generalmente si utilizzano appositi *container*, ovvero contenitori di oggetti disponibili ad essere controllati. Il vero e proprio controllo generalmente è detenuto dal framework[1] o dall'ambiente utilizzato; l'entità che detiene il controllo utilizza uno o più container al fine di recuperare gli oggetti necessari all'esecuzione dell'elaborazione.

Per poter applicare con successo l'*Inversion of Control*, si deve tener conto di un altro principio: il cosiddetto *Separation of Concerns*. Secondo questo, un

1 Tutti i framework in un modo o nell'altro utilizzato il principio dell'*Inversion of Control*, ma un esempio di framework progettato per sfruttarne pienamente i vantaggi è Apache Excalibur (ex Apache Avalon) [FrAE], il quale realizza un container di componenti leggero e predisposto all'integrazione in una qualunque applicazione.

problema deve essere scomposto in problemi di complessità minore, relativi ad uno specifico aspetto del problema complessivo. L'intento del *Separation of Concerns* è quello di separare le responsabilità tra gli oggetti, in modo tale che ciascun oggetto sia specializzato, implementando solamente le funzionalità di sua competenza. Gli oggetti inseriti nei container devono applicare tale principio, in modo tale che l'entità che detiene il controllo possa richiamare lo specifico oggetto che offre una determinata funzionalità.

Un oggetto contenuto all'interno di un container deve essere opportunamente recuperato; per poter eseguire tale operazione, il container deve offrire un meccanismo di recupero che consenta all'entità che detiene il controllo di poter ottenere l'oggetto dotato delle funzionalità necessarie. Il meccanismo di recupero dovrebbe consentire il disaccoppiamento tra l'implementazione dell'oggetto che offre la funzionalità necessaria e l'implementazione dell'entità che detiene il controllo. Per attuare tale disaccoppiamento, la classe che implementa l'entità di controllo non deve conoscere ed utilizzare la specifica classe che implementa l'oggetto controllato, altrimenti ne sarebbe dipendente, ma dovrebbe utilizzare l'entità che ne rappresenta la specifica comportamentale, ovvero una sua interfaccia o super-classe. Rimane il problema di stabilire quale entità sceglierà le specifiche classi che implementeranno gli oggetti da controllare; denominiamo come *assembler* la generica entità che svolge questo ruolo.

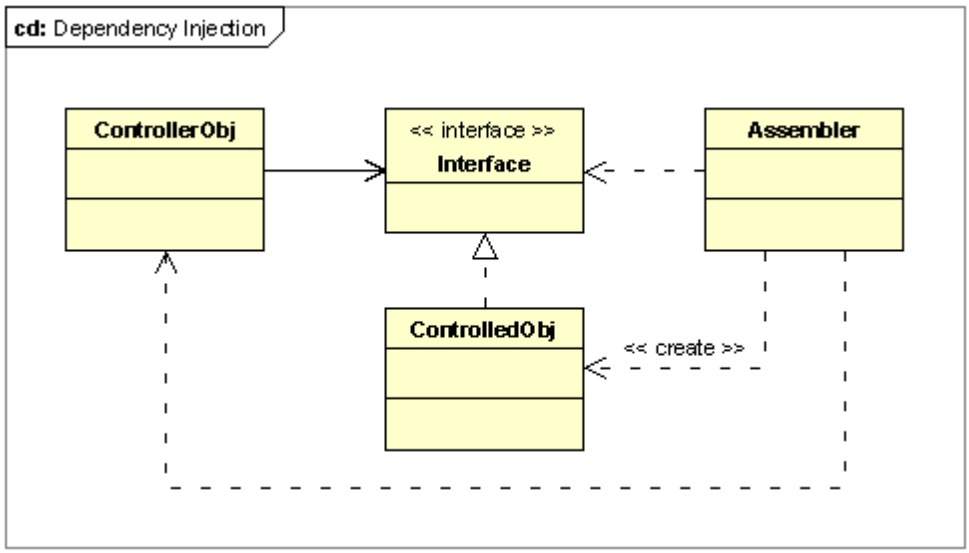


Figura A.1: Diagramma delle classi relativo al design pattern Dependency Injection

Una tecnica che permette il disaccoppiamento tra classe utilizzante e classe da utilizzare consiste nell'iniettare, nell'istanza della classe utilizzante, un riferimento

all'implementazione della classe da utilizzare. Tale tecnica caratterizza il design pattern *Dependency Injection* [Fow04], nel quale è l'assembler ad eseguire l'iniezione del riferimento relativo all'istanza della classe da utilizzare.

Esistono sostanzialmente tre tecniche di iniezione della dipendenza:

- *Constructor Injection*: il riferimento relativo all'istanza della classe da utilizzare viene passato al momento della creazione dell'istanza utilizzante; questa tecnica è usata sia in PicoContainer[2], sia in Spring Framework[3];
- *Setter Injection*: il riferimento relativo all'istanza della classe da utilizzare viene passato attraverso un apposito metodo della classe utilizzante; questa tecnica è usata sia in PicoContainer, sia in Spring Framework;
- *Interface Injection*: il riferimento relativo all'istanza della classe da utilizzare viene passato richiamando un metodo relativo ad un'interfaccia realizzata dalla classe utilizzante; questa tecnica è usata in Apache Excalibur.

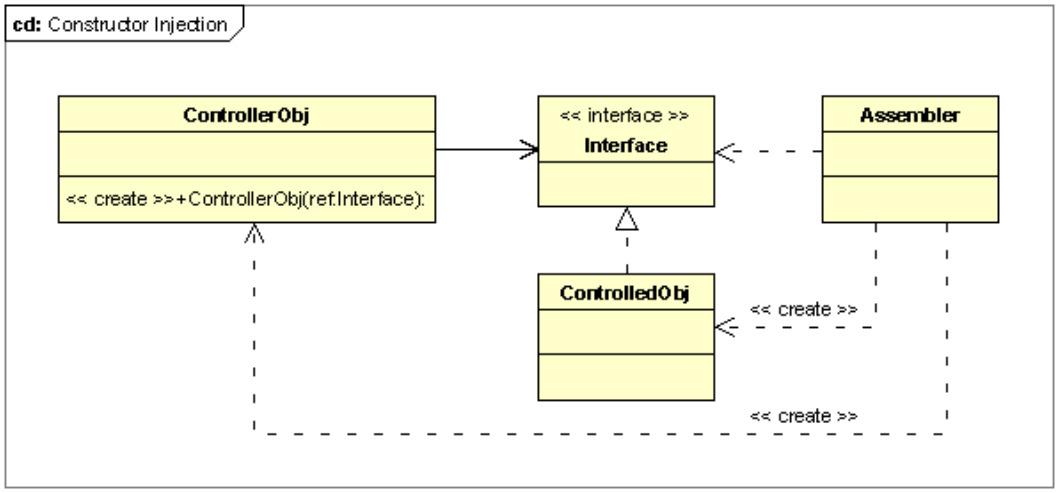


Figura A.2: Diagramma delle classi relativo al design pattern Constructor Injection

2 PicoContainer [FrPC], come Apache Excalibur, realizza un container di componenti leggero e predisposto all'integrazione in una qualunque applicazione; si differenzia da Apache Excalibur perché è completamente basato sul design pattern *Dependency Injection*, mentre l'altro utilizza sia il *Dependency Injection* che il *Service Locator*.

3 Spring Framework [FrSF] è un framework per la realizzazione di applicazioni basate sulla piattaforma Java/J2EE, in particolare applicazioni web; anch'esso è dotato di un container di componenti leggero e predisposto all'integrazione come Apache Excalibur e PicoContainer.

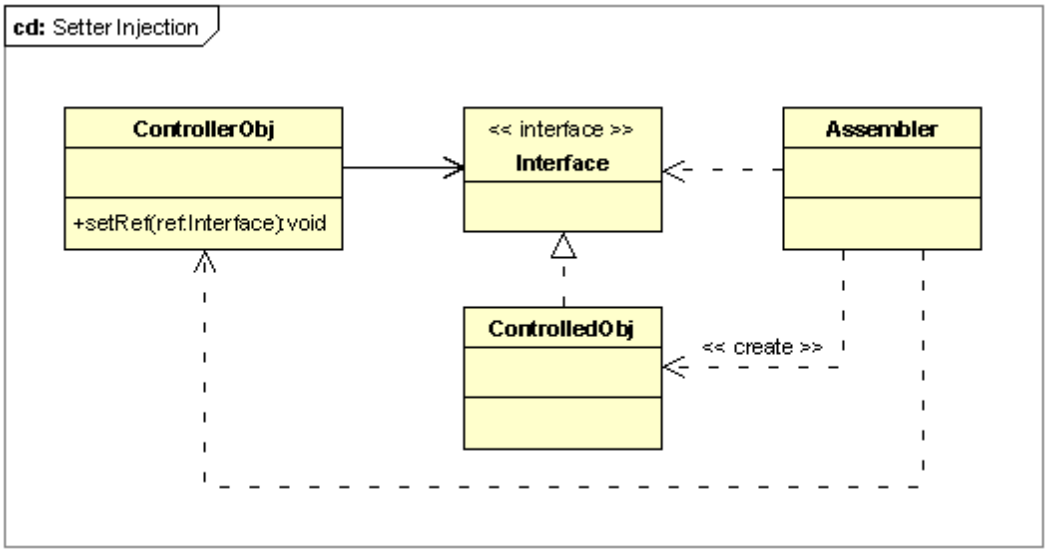


Figura A.3: Diagramma delle classi del design pattern Setter Injection

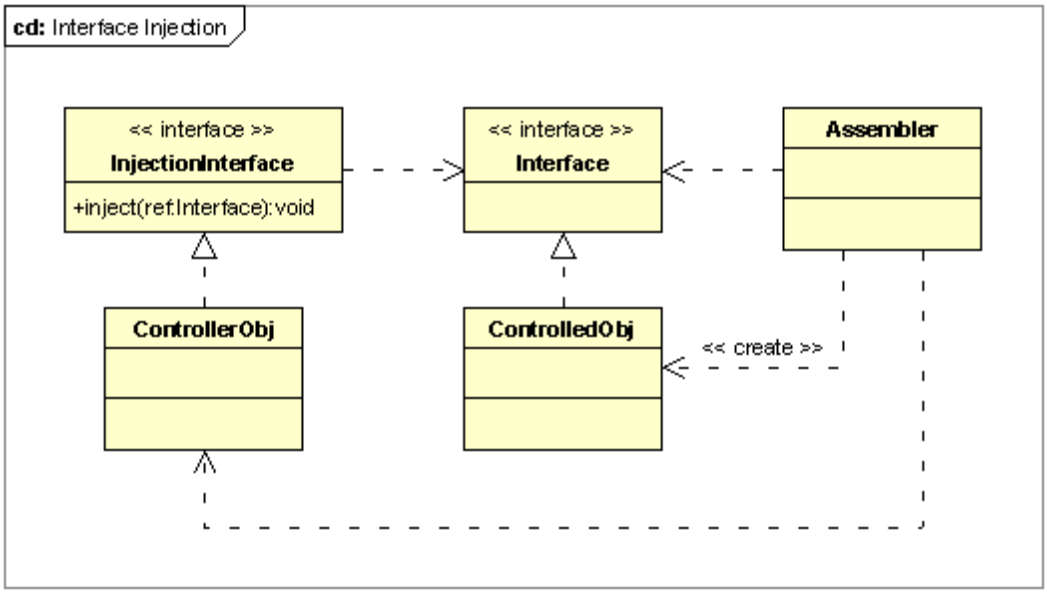


Figura A.4: Diagramma delle classi del design pattern Interface Injection

Notiamo che, per evitare la dipendenza verso un numero imprecisato di classi che implementano le istanze controllate e le istanze che detengono il controllo, le entità assembler vengono opportunamente configurate utilizzando l'introspezione o un file di configurazione.

A.2 Design pattern Service Locator

Una ulteriore tecnica per il disaccoppiamento tra classi prevede che sia l'istanza della classe utilizzante a richiedere esplicitamente ad un container l'istanza di una qualunque classe che realizza una determinata interfaccia o implementa una

specifica classe. Stiamo parlando di un design pattern diverso dal *Dependency Injection*, dato che non esegue l'iniezione della dipendenza, ma si tratta comunque di una forma di *Inversion of Control*. Questo nuovo design pattern è denominato *Service Locator* ed è utilizzato sia in Apache Excalibur che in Jini[4].

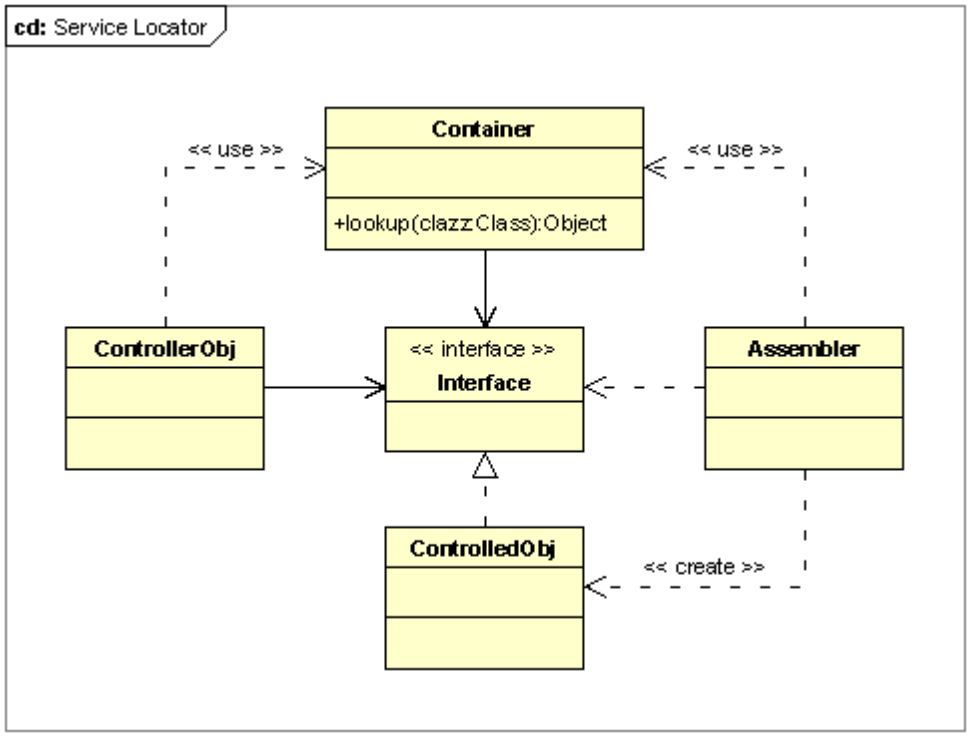


Figura A.5: Diagramma delle classi relativo al design pattern Service Locator

Si noti che nelle tre tecniche di *Dependency Injection* l'oggetto utilizzante si ritrova il riferimento relativo all'oggetto utilizzato senza dover fare richieste esplicite; conseguentemente, usando la tecnica *Service Locator* tutte le classi che utilizzano gli oggetti registrati nel container[5] sono dipendenti da questo. Inoltre, nelle tecniche precedenti il ruolo dell'assembler è svolto dal container, il quale solitamente è il componente principale di un framework, mentre per il design pattern *Service Locator* assembler e container sono due entità nettamente separate,

4 Jini [FrJini] è un architettura per la realizzazione di sistemi distribuiti scalabili e predisposti al cambiamento; ciò che contraddistingue Jini è il suo meccanismo di individuazione dei servizi.

5 Nell'ambito del design pattern che si sta descrivendo, l'entità che svolge il ruolo di container è generalmente denominata *service locator*, da qui il nome del design pattern; per evitare di generare confusione non utilizzeremo questo termine per i container, ma lo useremo per indicare il design pattern. Notiamo, però, che sarebbe più corretto l'uso del termine service locator: infatti, nell'ambito delle reti di calcolatori, l'entità che implementa il service locator non svolge solamente il ruolo di contenitore, ma ha anche la responsabilità di individuare i servizi richiesti e di inizializzare i necessari proxy per il loro utilizzo, oltre a gestire eventuali necessità di caching.

quindi che svolgono ruoli chiaramente distinti. Infatti, in tal caso, l'assembler può essere una qualunque entità che, oltre a inizializzare l'istanza della classe disponibile ad essere controllata, la inserisce anche nel container. Notiamo anche che la separazione tra il concetto di container e assembler implica che l'assembler può anche non avere alcun riferimento alle classi utilizzanti, pertanto risulta essere molto più leggero; conseguentemente l'assembler non fa parte dell'eventuale framework a cui appartiene il container, quindi non sono previsti particolari meccanismi per la sua configurazione.

Generalmente le realizzazioni del design pattern *Service Locator* implementano i container in modo tale che restituiscano una sola istanza per ogni richiesta di acquisizione relativa ad un oggetto che realizza una specifica interfaccia o implementa una specifica classe; quindi se vengono aggiunte al container classi distinte che realizzano la stessa interfaccia, a seguito di una richiesta relativa a tale interfaccia, il container restituisce arbitrariamente la prima classe individuata.

L'ambiente NetBeans utilizza una variante del *Service Locator*, la cui caratteristica distintiva è data dalla capacità di poter definire dei *template* di ricerca, in modo tale da ottenere, tramite richiesta ad un container, istanze multiple che realizzano l'interfaccia o implementano la classe necessaria. L'implementazione di base di un container nell'ambiente NetBeans è data dalla classe astratta `org.openide.util.Lookup`[6].

A.3 Design pattern Cookie

L'altro design pattern che caratterizza i componenti dell'ambiente NetBeans è denominato *Cookie*; tale design pattern è stato concepito dai progettisti dell'ambiente NetBeans e non risulta essere utilizzato in altri progetti pubblicati in Internet, dato che non siamo riusciti a trovare alcun riferimento ad esso oltre alla documentazione dell'ambiente NetBeans.

L'intento del design pattern *Cookie* è quello di dotare un oggetto di funzionalità in modo trasparente rispetto alla classe implementata, ovvero aggiungere funzionalità senza che la classe implementante sia dotata di specifici metodi. Ciò comporta che l'insieme delle funzionalità offerte dall'oggetto considerato sarà dinamico, ovvero gli oggetti che realizzano tale design pattern possono acquisire o

6 Nel tutorial "Introducing to the NetBeans Idioms and Infrastructure" [NbTut4] viene erroneamente indicato che tale classe realizza il design pattern *Dependency Injection*.

perdere funzionalità durante l'elaborazione.

Per rendere possibile tutto ciò occorre separare l'implementazione dell'oggetto, la quale realizza il modello dell'oggetto, dalle implementazioni di ciascuna delle sue funzionalità, le quali realizzano i comportamenti dell'oggetto; pertanto, tali funzionalità, devono essere realizzate da classi a se stanti; si osservi che così facendo stiamo applicando il principio denominato *Separation of Concerns*.

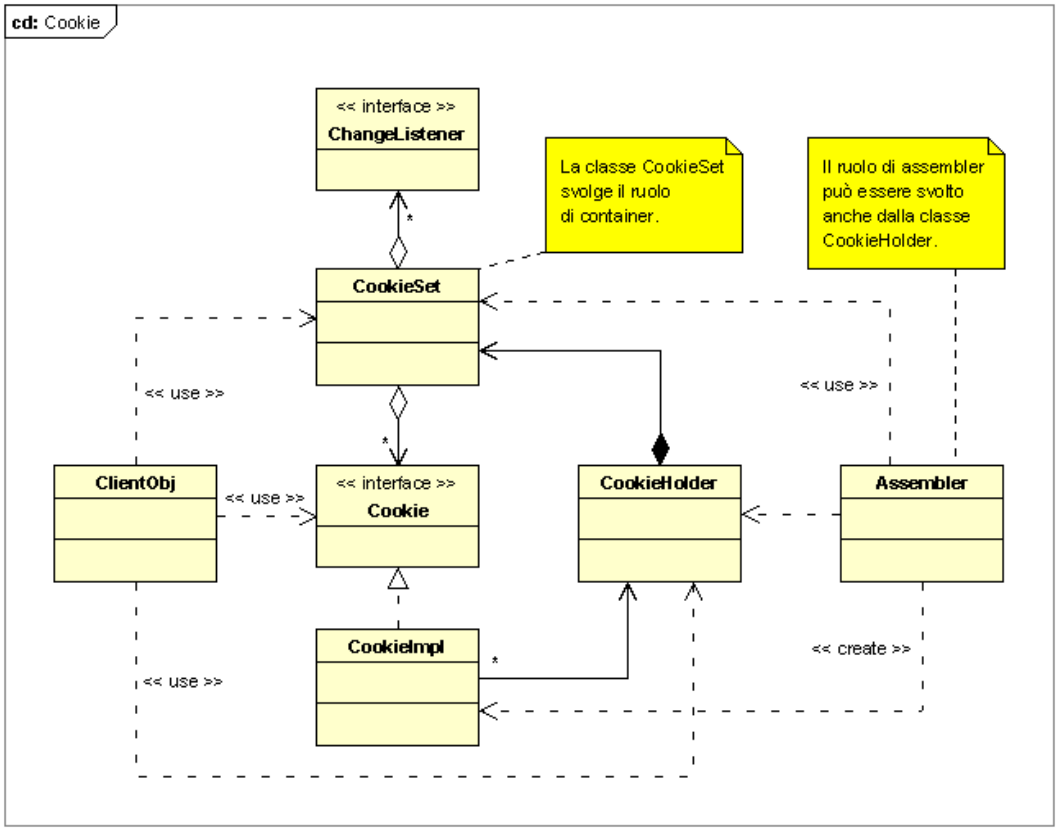


Figura A.6: Diagramma delle classi relativo al design pattern Cookie

Gli oggetti che implementano le funzionalità sono detti comunemente *cookie*, mentre gli oggetti a cui sono associati i cookie sono detti *cookie holder*. Notiamo che un cookie può anche implementare un aggregato di funzionalità, le quali potranno essere aggiunte o tolte ad un determinato oggetto soltanto tutte assieme, seguendo il destino del cookie che le implementa.

Il comportamento implementato da un cookie può essere specifico per la determinata classe che implementa il cookie holder, a causa della necessità di conoscere lo stato interno dell'oggetto a cui un cookie è stato associato; quindi la classe che implementa un cookie potrebbe essere strettamente legata alla classe del cookie holder. Infatti, spesso i cookie sono implementati come inner-class

delle classi che implementano i cookie holder; un'altra soluzione potrebbe consistere nell'applicare il design pattern *Visitor* [GoF95] per l'implementazione dei cookie, in modo tale che la classe che implementa il cookie holder non presenti inner-class, ma soprattutto in modo tale da poter creare nuovi cookie senza dover modificare il codice della classe relativa al cookie holder. Inoltre, tale soluzione è molto conveniente se il cookie holder fa parte di una struttura gerarchica e se le operazioni svolte dai cookie effettuano delle esplorazioni di tale struttura.

I cookie associati ad uno specifico cookie holder sono raccolti in un apposito container, a sua volta associato al cookie holder a cui fanno riferimento i cookie; tali tipi di container sono denominati *cookie set*. Per utilizzare una specifica funzionalità, l'oggetto utilizzante deve, prima di tutto, recuperare il cookie set associato all'oggetto che vuole utilizzare, dopodiché acquisirà il cookie relativo alla funzionalità necessaria, per poi usarlo al fine di eseguire l'elaborazione che deve svolgere.

Per richiedere un cookie, l'oggetto utilizzante deve specificare l'interfaccia realizzata dal cookie oppure una classe da esso implementata; notiamo che in tal modo stiamo di fatto utilizzando il design pattern *Service Locator*, conseguentemente l'applicazione del design pattern *Cookie* implica l'applicazione di quest'altro design pattern.

Per aggiungere un cookie ad un determinato cookie set, basta indicare l'istanza del cookie da aggiungere, ma è anche possibile registrare presso il cookie set una factory di cookie, ovvero un oggetto in grado di creare un determinato tipo di cookie nel momento in cui verrà richiesto. Aggiungendo una factory di cookie si deve specificare la classe o le classi che tale factory è in grado di creare. Analogamente, è possibile rimuovere un cookie specificando l'istanza del cookie da rimuovere, ma è anche possibile rimuovere una factory di cookie, ottenendo come ulteriore effetto la rimozione del corrispondente cookie dal cookie holder. Notiamo che l'aggiunta o la rimozione di un cookie o di una factory di cookie causa la notifica di un evento verso eventuali listener.

L'utilizzo di una factory di cookie è molto utile nei casi in cui il cookie holder svolge il ruolo di assembler, perché rende il cookie holder indipendente dalle implementazioni dei suoi cookie; il comportamento di una factory di cookie può essere specificato da un'interfaccia o da una classe astratta, in modo tale da non

avere uno stretto accoppiamento tra cookie holder e la particolare implementazione della factory di cookie ad esso associata. Se ciò non fosse vero, il cookie holder sarebbe transitivamente dipendente con le implementazioni dei cookie creati dalla factory.

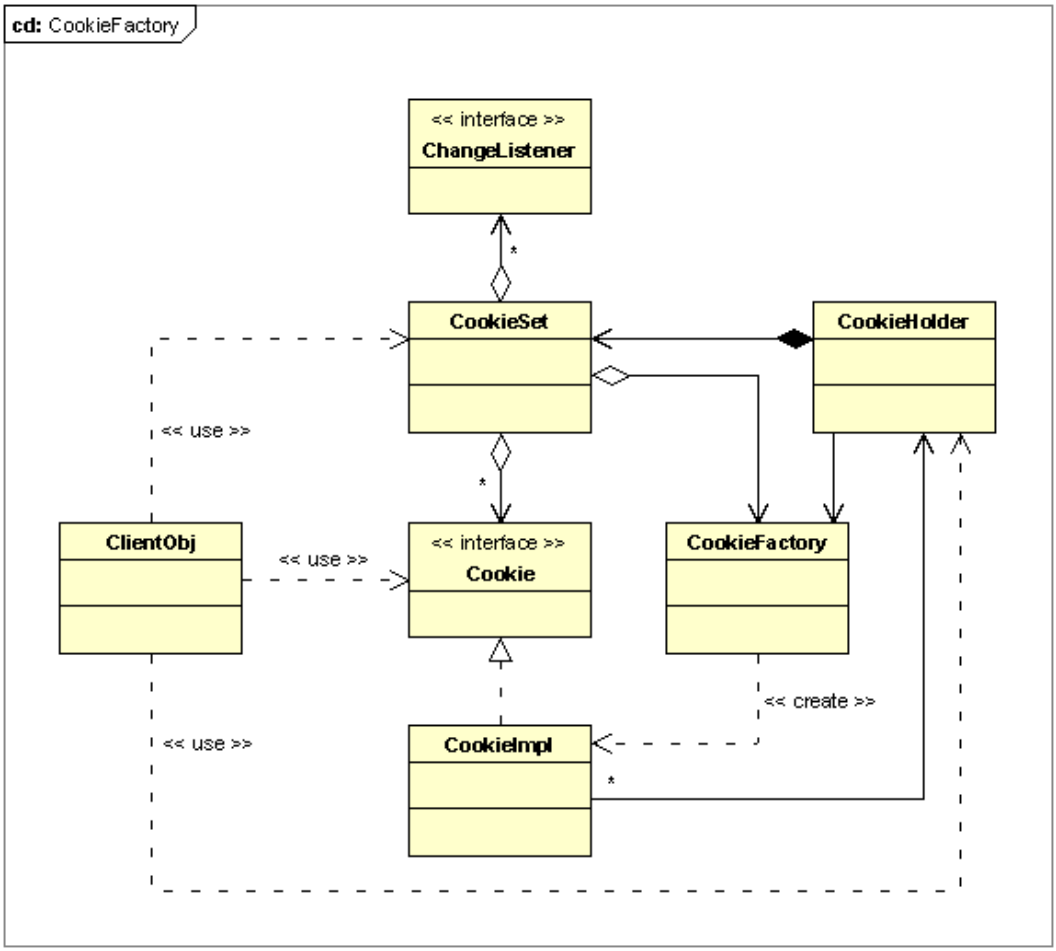


Figura A.7: Diagramma delle classi relativo al design pattern Cookie utilizzante un factory di cookie

Infine, osserviamo che non è ragionevole che, per una richiesta di acquisizione di cookie, il container ne restituisca due o più, dato che si deve svolgere un preciso compito utilizzando uno specifico servizio e non si devono recuperare insiemi di oggetti al fine di manipolarli; pertanto la realizzazione del design pattern *Service Locator* utilizzata dal design pattern *Cookie* sarà quella tradizionale e non la variante realizzata dalla classe astratta `org.openide.util.Lookup` presente nell'ambiente NetBeans.

A.4 Differenze tra i design pattern analizzati

Tutti i design pattern che sono stati analizzati in questa sezione si basano sul principio dell'*Inversion of Control*, ma operano diversamente per evitare che si presentino dipendenze tra classi utilizzanti e classi utilizzate. Infatti, il design pattern *Dependency Injection*, assieme alle sue varianti, utilizza la tecnica dell'iniezione della dipendenza, mentre il design pattern *Service Locator* utilizza la tecnica del recupero di una qualunque classe che implementi una specifica interfaccia o classe astratta. Infine, il design pattern *Cookie* utilizza il design pattern *Service Locator* al fine di evitare la dipendenza tra le classi richiedenti e le classi utilizzate.

L'uso del principio dell'*Inversion of Control* include anche l'uso del principio denominato *Separation of Concerns*, ma il design pattern *Cookie*, a differenza degli altri, si basa esplicitamente su di esso nonostante utilizzi il design pattern *Service Locator*.

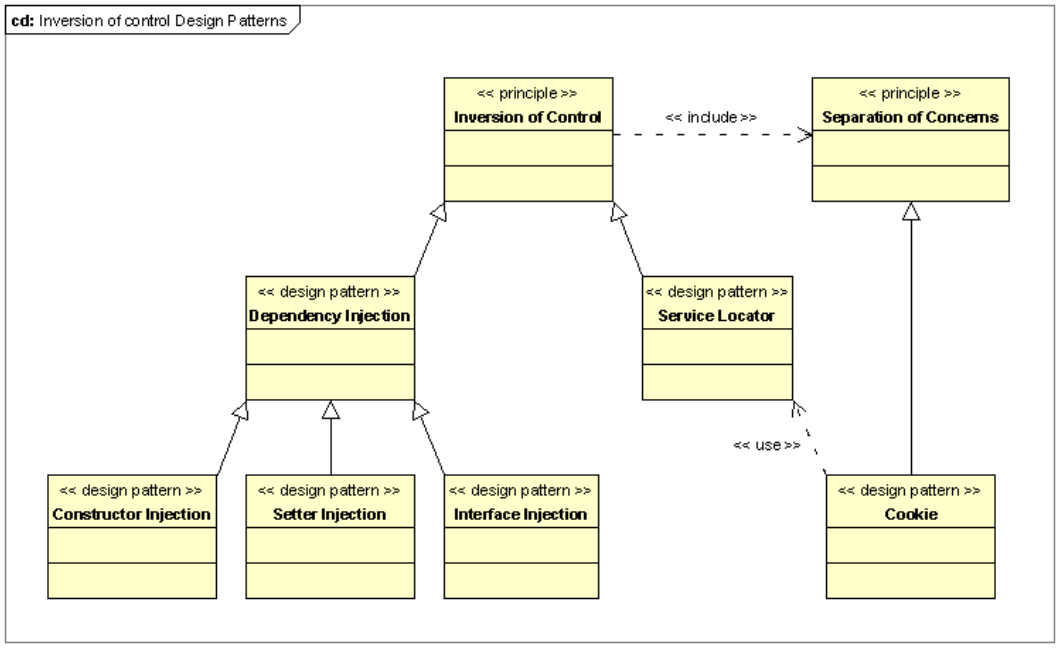


Figura A.8: Diagramma delle classi relativo ai design pattern basati sul principio dell'Inversion of control

Appendice B

NetBeans API

In questa appendice sono raccolti i risultati riguardanti l'esplorazione delle API relative all'ambiente NetBeans, sottoforma di brevi commenti alle classi e interfacce che hanno destato interesse per quanto riguarda la realizzazione del plug-in, suddividendole per package.

org.openide

- `ErrorManager`: classe astratta che implementa un gestore di messaggi di errore o di log. Il metodo di classe `ErrorManager getDefault()` restituisce il gestore principale dei messaggi dell'ambiente NetBeans.
- `NotifyDescriptor`: classe che implementa un descrittore relativo ad un messaggio di notifica; è possibile specificare il tipo di messaggio da visualizzare.
- `NotifyDescriptor.Message`: classe che implementa un descrittore relativo ad un messaggio di notifica per il quale l'utente non deve effettuare una scelta, ma può solamente confermare la lettura del messaggio; tipicamente tali messaggi notificano il risultato di una operazione o il verificarsi di un qualunque evento che non sia un errore dell'applicazione. Estende la classe `NotifyDescriptor`.
- `DialogDisplayer`: classe astratta che implementa un visualizzatore di finestre di dialogo. Utilizzando il metodo `Object notify(NotifyDescriptor)` è possibile visualizzare una finestra di dialogo con il contenuto descritto dall'istanza della classe

`NotifyDescriptor` passata come parametro di ingresso. Per ottenere il visualizzatore di finestre di dialogo predefinito della NetBeans Platform, si deve utilizzare il metodo di classe `DialogDisplayer getDefault()`.

org.openide.actions

- `OpenAction`: classe che implementa un'azione che apre un oggetto. Estende la classe `org.openide.util.actions.CookieAction`, pertanto tale azione è attiva solamente se il nodo selezionato incapsula un cookie che realizza l'interfaccia `org.openide.cookies.OpenCookie`.
- `CloseViewAction`: classe che implementa un'azione che chiude un *top component*. Estende la classe `CallbackSystemAction`.
- `CloneViewAction`: classe che implementa un'azione che clona un *top component*. Estende la classe `CallbackSystemAction`.
- `CopyAction`: classe che implementa un'azione che copia un oggetto nella *clipboard*.
- `CutAction`: classe che implementa un'azione che rimuove un oggetto dopo averlo copiato nella *clipboard*.
- `PasteAction`: classe che implementa un'azione che incolla un oggetto dalla *clipboard*.
- `DeleteAction`: classe che implementa un'azione che elimina un oggetto.
- `EditAction`: classe che implementa un'azione che apre un oggetto in un editor. Estende la classe `org.openide.util.actions.CookieAction`, pertanto tale azione è attiva solamente se il nodo selezionato incapsula un cookie che realizza l'interfaccia `org.openide.cookies.EditCookie`.
- `ViewAction`: classe che implementa un'azione che visualizza un oggetto. Estende la classe `org.openide.util.actions.CookieAction`, pertanto tale azione è attiva solamente se il nodo selezionato incapsula un cookie che realizza l'interfaccia `org.openide.cookies.ViewCookie`.

org.openide.cookies

- `OpenCookie`: interfaccia che specifica il comportamento di un cookie in grado di aprire un oggetto.
- `CloseCookie`: interfaccia che specifica il comportamento di un cookie in grado di chiudere un oggetto.
- `EditCookie`: interfaccia che specifica il comportamento di un cookie in grado di aprire un oggetto in un editor.
- `EditorCookie`: interfaccia che specifica il comportamento di un cookie in grado di editare il contenuto di un oggetto.
- `ViewCookie`: interfaccia che specifica il comportamento di un cookie in grado di visualizzare un oggetto.

org.openide.explorer

- `ExplorerManager`: classe che implementa un gestore della selezione di una gerarchia di nodi; è in grado di indicare la radice della gerarchia di nodi gestita attraverso il metodo `Node getRootContext()` e di indicare i nodi selezionati dall'utente tramite il metodo `Node[] getSelectedNodes()`.
- `ExplorerManager.Provider`: interfaccia che specifica il comportamento di un oggetto in grado di fornire un'istanza della classe `ExplorerManager`.
- `ExplorerUtils`: classe che raccoglie un insieme di metodi di classe utili per l'integrazione di istanze della classe `ExplorerManager` all'interno dei componenti grafici Swing.

org.openide.explorer.view

- `ListView`: classe che implementa una explorer view in grado di visualizzare i figli di un nodo radice come gli elementi di una lista; estende la classe `javax.swing.JScrollPane`, pertanto implementa un componente grafico Swing.
- `IconView`: che implementa una explorer view in grado di visualizzare i figli di un nodo radice come un'insieme di icone; estende la classe

`ListView`.

- `TreeView`: classe astratta che implementa le funzionalità di base di una explorer view che rappresenta una gerarchia di nodi come un albero; estende la classe `javax.swing. JScrollPane`, pertanto implementa un componente grafico Swing.
- `BeanTreeView`: classe che implementa una explorer view che rappresenta una gerarchia di nodi come un albero, estendendo la classe `TreeView`.
- `TreeTableView`: classe che implementa una explorer view composta da due viste: una vista per visualizzare una gerarchia di nodi sotto forma di albero e un'altra vista per visualizzare la tabella delle proprietà relativa alla risorsa rappresentata dal nodo selezionato; estende la classe `BeanTreeView`.
- `ChoiceView`: classe che implementa una explorer view in grado di visualizzare i figli di un nodo radice come una serie di caselle di selezione; estende la classe `javax.swing. JComboBox`, pertanto implementa un componente grafico Swing.

org.openide.filesystems

- `FileObject`: classe astratta che costituisce l'implementazione di base di un file object, definendo i metodi per l'accesso alle informazioni di base e per l'esecuzione delle operazioni fondamentali; permette di registrare dei listener relativi ad eventi di modifica del file object; tali listener devono realizzare l'interfaccia `FileChangeListener`.
- `FileChangeListener`: interfaccia che specifica il comportamento di un listener relativo ad eventi di modifica di un file object; gli eventi notificati tramite questa interfaccia implementano la classe `FileEvent`.
- `FileEvent`: classe che implementa un evento relativo alla modifica di un file object.
- `FileLock`: classe che implementa un lock relativo ad uno specifico file object, il quale permette l'accesso esclusivo al file object a cui si riferisce. Un lock viene ottenuto chiamando il metodo `FileLock lock()`

dell'istanza del file object interessato, mentre viene rilasciato chiamando il metodo `void releaseLock()` del lock ottenuto.

- `FileUtil`: classe che raccoglie una collezione di metodi di classe per la gestione di file.
- `FileSystem`: classe astratta che costituisce una implementazione di base relativamente ad un filesystem virtuale. Definisce una collezione di metodi astratti per l'accesso alle informazioni che caratterizzano il filesystem virtuale. In particolare definisce il metodo astratto `FileObject findResource(String)`, il quale, una volta implementato in una sottoclasse, dovrà permette di recuperare il file object, contenuto nel filesystem, corrispondente al percorso assoluto indicato in ingresso.
- `AbstractFileSystem`: classe astratta che costituisce l'implementazione di base per la definizione di nuovi filesystem virtuali. Estende la classe `FileSystem`, implementando molti dei suoi metodi astratti.
- `LocalFileSystem`: classe che permette l'accesso e la gestione al filesystem locale; estende la classe astratta `AbstractFileSystem`. Osserviamo che tale classe presenta seri problemi quando vengono specificati percorsi assoluti relativi a file in ambiente Microsoft Windows: infatti la ricerca di file, tramite il metodo `FileObject findResource(String)`, è sempre infruttuosa quando sono specificati percorsi ove sono presenti identificatori di drive in stile Windows.
- `JarFileSystem`: classe che implementa un filesystem virtuale basato sul contenuto di un file JAR.
- `XMLFileSystem`: classe che implementa un filesystem virtuale basato sul contenuto di un documento XML.
- `MultiFileSystem`: classe che implementa un proxy in grado di gestire un insieme di filesystem virtuali come se fosse un unico filesystem.
- `MIMEResolver`: classe astratta che definisce il comportamento di un risolutore di identificatori MIME; in particolare definisce il metodo astratto `String findMIMEType(FileObject)`, il quale restituisce l'identificatore MIME del tipo di contenuto relativo al file object passato in ingresso.

- `Repository`: classe che permette di accedere al System Filesystem. Per poter accedere ad esso si deve precedentemente recuperare l'unica istanza di questa classe tramite il metodo di classe `Repository getDefault()`, dopodiché si dovrà richiamare il metodo `FileSystem getDefaultFileSystem()` per poter ottenere l'istanza della classe `FileSystem` che implementa il System Filesystem. Notiamo che tale classe non è utile per svolgere altre funzioni al di fuori di quella appena analizzata. Infatti, storicamente tale classe gestiva i filesystem virtuali montati nell'ambiente NetBeans, ma dato che tale funzionalità non è più utilizzata dalla versione 4.0 dell'ambiente, tutti i metodi di questa classe sono stati deprecati, eccezion fatta per i due metodi citati prima e l'unico costruttore. Adesso l'unico filesystem che gestisce è proprio il System Filesystem.

org.openide.loaders

- `DataObject`: classe astratta che implementa le funzionalità di base di un data object, ovvero un modello relativo ad una risorsa che incapsula uno o più file object, al quale è possibile associare dei cookie. Per creare un data object si può utilizzare il metodo di classe `DataObject find(FileObject)`, il quale restituisce una opportuna istanza della classe `DataObject` relativa al file object fornito in ingresso. Per poter ottenere il file object principale, incapsulato in un'istanza della classe `DataObject`, si utilizza il metodo `FileObject getPrimaryFile()`. Per ottenere il livello di presentazione della risorsa incapsulata da un'istanza di questa classe, si richiama il metodo `Node getNodeDelegate()`, invece per ottenere un cookie si utilizza il metodo `Node.Cookie getCookie(Class)`, specificando l'interfaccia o la classe implementata dal cookie desiderato. Permette la registrazione di listener che realizzano sia l'interfaccia `java.beans.PropertyChangeListener`, sia l'interfaccia `java.beans. VetoableChangeListener`. Infine osserviamo che la classe `DataObject` realizza l'interfaccia `Node.Cookie`, pertanto le sue istanze, oltre ad avere dei cookie associati, sono a loro volta dei cookie.
- `MultiDataObject`: classe che implementa un data object in grado di

gestire più di un file object. Un'istanza di tale classe incapsula una collezione di entry relative ai file object in essa contenuti, implementate dalla classe astratta `MultiDataObject.Entry`, pur mantenendo separata l'entry primaria, relativa al file object principale, dalle entry secondarie, relative agli altri file object. Il metodo `MultiDataObject.Entry getPrimaryEntry()` e il metodo `Set secondaryEntries()` permettono di recuperare, rispettivamente, l'entry relativa al file object primario e una collezione di entry relative ai file object secondari. E' possibile aggiungere ulteriori entry relative a file object secondari, utilizzando il metodo `void addSecondaryEntry(MultiDataObject.Entry)`. Infine, con il metodo `CookieSet getCookieSet()` è possibile recuperare l'insieme di cookie associati ad un'istanza di questa classe; tale metodo ha visibilità a livello protected, quindi è richiamabile solamente dalle sottoclassi e dalle classi appartenenti allo stesso package.

- `DataLoader`: classe astratta in grado di riconoscere il tipo di un file object tramite l'estensione oppure, se si tratta di un documento XML, analizzando l'elemento principale; la sua funzionalità principale consiste nella possibilità di creare un opportuno data object a partire da un file object, attraverso il metodo di classe `DataObject find(FileObject)`.
- `MultiFileLoader`: classe astratta che estende la classe `DataLoader`, ampliando le funzionalità offerte al caso dei data object in grado di gestire più di un file object; è in grado di riconoscere più di un tipo di file object e di effettuare anche il riconoscimento di insiemi di file object.
- `UniFileLoader`: classe astratta che estende la classe `MultiFileLoader`, offrendo le stesse funzionalità della classe estesa, ma limitando le operazioni di riconoscimento a singoli file object, pur essendo in grado di riconoscere diversi tipi di file object.
- `DataNode`: classe che implementa un nodo che rappresenta un data object; estende la classe astratta `AbstractNode`.

org.openide.nodes

- `Node.Cookie`: interfaccia che non specifica alcun metodo, ma la cui

funzione è quella di indicare che la classe che la sta realizzando è un cookie.

- `Node`: classe astratta che implementa le funzionalità di base di un nodo, ovvero il livello di presentazione di una qualunque risorsa, tipicamente rappresentata da un data object o da un bean. Ad un'istanza di questa classe possono essere associate delle azioni eseguibili su di essa, recuperabili tramite il metodo `Action[] getActions(boolean)`, distinguendo tra azioni relative al contesto del nodo (parametro d'ingresso con valore true) e azioni relative al nodo stesso (parametro d'ingresso con valore false). Inoltre, un'istanza di questa classe può anche incapsulare una serie di cookie che rappresentano il comportamento del nodo; per poter recuperare tali cookie, ad ogni nodo è associata un'istanza della classe `Lookup`. Per poter accedere a tale lookup, la classe Node realizza l'interfaccia `Lookup.Provider`. Infine, il metodo `Node.Cookie getCookie(Class)` permette di recuperare un cookie che realizza/implementa l'interfaccia/classe specificata.
- `AbstractNode`: classe astratta che costituisce la più conveniente implementazione di base per la creazione di una nuova classe di nodi; estende la classe Node. Tale classe aggiunge il metodo `CookieSet getCookieSet()`, in grado di restituire un insieme di cookie.
- `CookieSet`: classe che implementa un insieme di cookie; un'istanza di tale classe non è altro che un contenitore che permette di memorizzare e recuperare dei cookie. Solitamente un'istanza di questa classe è associata ad un'istanza della classe `AbstractNode`.
- `Children`: classe astratta che implementa le funzionalità di base di un insieme di nodi, associabile ad un nodo in modo tale da poter costituire una gerarchia di nodi esplorabile.
- `Children.Array`: classe che implementa un array di nodi; estende la classe `Children`.
- `Index.ArrayChildren`: classe che implementa un array di nodi riordinabile; estende la classe `Children.Array`.
- `Index`: interfaccia che specifica il comportamento di un nodo in grado di riordinare i suoi nodi figli.

- `IndexedNode`: classe che realizza l'interfaccia `Index` ed estende la classe `AbstractNode`; pertanto implementa un nodo dotato di figli ed in grado di riordinarli.

org.openide.text

- `CloneableEditor`: classe che implementa l'editor predefinito dell'ambiente NetBeans. Estende la classe `CloneableTopComponent`, pertanto è un *top component* clonabile.
- `CloneableEditorSupport`: classe astratta che associa un editor e un'istanza della classe `javax.swing.text.Document` ad un oggetto, implementando un supporto di base per la modifica di oggetti editabili. Relativamente alle interfacce `OpenCookie`, `CloseCookie` e `ViewCookie`, `EditCookie` e `EditorCookie` (appartenenti al package `org.openide.cookies`), ne implementa i metodi ma non le realizza; saranno le sottoclassi a decidere quali di queste interfacce realizzare. Estende la classe astratta `org.openide.window.CloneableOpenSupport`.
- `CloneableEditorSupport.Env`: interfaccia che specifica il comportamento di un ambiente relativo ad un oggetto editabile; tale interfaccia è utilizzata dalla classe `CloneableEditorSupport`.
- `DataEditorSupport`: classe che associa un editor e un'istanza della classe `javax.swing.text.Document` ad un *data object*, implementando un supporto per la modifica di *data object*. Estende la classe astratta `CloneableEditorSupport`.

org.openide.util

- `Lookup`: classe astratta che implementa un registro in grado di individuare e recuperare generiche istanze, specificando un'interfaccia o una classe; si basa sul design pattern *Service Locator*. Con il metodo `Object lookup(Class)` si ottiene un'istanza che realizza l'interfaccia o implementa la classe specificata come parametro di ingresso; nel caso in cui siano disponibili più istanze relative all'interfaccia o classe specificata, viene restituita la prima individuata. Per poter individuare tutte le istanze

che realizzano una determinata interfaccia o implementano una specifica classe, si deve ricorrere al metodo `Lookup.Result lookup(Lookup.Template)`; tale metodo necessita in ingresso di un'istanza della classe `Lookup.Template`, la quale rappresenta un pattern per la ricerca di istanze, mentre restituisce un'istanza della classe `Lookup.Result`, la quale rappresenta una collezione di istanze. Le istanze di questa classe sono comunemente dette lookup.

- `AbstractLookup`: classe astratta che costituisce l'implementazione di base per poter realizzare un service locator specifico per un determinato ambito. Estende la classe `Lookup`.
- `Lookup.Provider`: interfaccia che specifica il comportamento di un oggetto in grado di restituire l'istanza della classe `Lookup` ad esso associata; specifica solamente il metodo `Lookup getLookup()`.
- `Lookups`: classe che colleziona una serie di metodi di classe per la creazione di implementazioni comuni della classe `Lookup`, per esempio contenenti un'unica istanza, oppure contenenti un prefissato insieme di istanze, ecc..
- `NbBunble`: classe che implementa una collezione di metodi statici utili alla localizzazione di risorse di vario di tipo; estende le funzionalità offerte dal *Java resource support*. Utilizzando il metodo `String getMessage(Class,String)` è possibile ottenere il messaggio localizzato identificato dalla stringa passata in ingresso e contenuto nel documento *Bundle.properties*, a sua volta contenuto nel package di appartenenza della classe passata in ingresso durante la chiamata del metodo.
- `Utilities`: classe che raccoglie un insieme di metodi di classe non appartenenti ad una specifica categoria e di varia utilità.

org.openide.util.actions

- `CallableSystemAction`: classe astratta che implementa un'azione globale, sempre attiva, integrata nell'ambiente NetBeans e potenzialmente richiamabile da qualsiasi componente; normalmente le istanze di questa classe sono registrate nel System Filesystem. L'oggetto invocante

un'azione implementata da questa classe normalmente richiama il metodo `void performAction()`, ma è anche possibile che richiami il metodo `void actionPerformed(ActionEvent)`, relativo all'interfaccia `java.awt.event. ActionListener`, la quale è realizzata da questa classe. Inoltre, notiamo che la classe in questione realizza anche l'interfaccia `javax.swing.Action`.

- `CallbackSystemAction`: classe astratta che implementa un'azione globale, integrata nell'ambiente NetBeans, associata ad un *performer*, ovvero un oggetto in grado di gestirne l'esecuzione; se nessun *performer* è associato ad essa, l'azione non sarà attiva. Estende la classe `CallableSystemAction`.
- `NodeAction`: classe astratta che implementa le funzionalità di base di un'azione locale, attiva condizionatamente e sensibile al contesto a cui è applicata; estende la classe `CallableSystemAction`. Questo tipo di azione è riferita ad un determinato nodo, ed è richiamabile tramite menù contestuale; tipicamente eseguono un'operazione svolta da un determinato cookie. Le sottoclassi che estendendo tale classe devono implementare il metodo `boolean enable(Node[])`, il quale verifica se l'azione deve essere attiva o meno in base ai nodi selezionati dall'utente.
- `CookieAction`: classe astratta che implementa le funzionalità di base di un'azione locale, attiva condizionatamente e sensibile al contesto a cui è applicata; estende la classe `NodeAction`. Questo tipo di azione è riferita ad un determinato nodo, ed è richiamabile tramite menù contestuale; tipicamente eseguono un'operazione svolta da un determinato cookie. Le sottoclassi che estendendo tale classe devono implementare il metodo `boolean enable(Node[])`, il quale verifica se l'azione deve essere attiva o meno in base ai nodi selezionati dall'utente, il metodo `Class[] cookieClasses()`, il quale restituisce l'elenco delle classi/interfacce di cui almeno una deve essere supportata da ciascun nodo selezionato, e il metodo int mode(), il quale specifica la modalità di selezione dei nodi affinché l'azione sia attiva.

org.openide.windows

- `TopComponent`: classe che implementa una vista perfettamente integrata

nell'interfaccia grafica tipica dell'ambiente NetBeans. Tale vista può essere una finestra indipendente o essere agganciata all'interno di un'altra finestra. Notiamo che questa classe estende la classe `javax.swing.JComponent` (quindi ogni sua istanza è un widget Swing) e realizza l'interfaccia `Lookup.Provider` (quindi ad ogni sua istanza è associato un lookup). Infine segnaliamo che richiamando il metodo `int getPersistenceType()` è possibile verificare il tipo di persistenza supportato da questa vista.

- `TopComponent.Cloneable`: interfaccia che specifica il comportamento di una vista clonabile. Tale interfaccia deve essere realizzata da una vista affinché l'azione implementata dalla classe `CloneViewAction` possa eseguirne la clonazione.
- `CloneableTopComponent`: classe astratta che implementa un *top component* clonabile; estende la classe `TopComponent`.
- `CloneableOpenSupport`: classe astratta che implementa un semplice supporto per l'apertura di oggetti. Relativamente alle interfacce `OpenCookie`, `CloseCookie` e `ViewCookie` (appartenenti al package `org.openide.cookies`), ne implementa i metodi ma non le realizza; saranno le sottoclassi a decidere quali di queste interfacce realizzare.
- `CloneableOpenSupport.Env`: interfaccia che specifica il comportamento di un ambiente relativo ad un oggetto apribile; tale interfaccia è utilizzata dalla classe `CloneableOpenSupport`.
- `Mode`: interfaccia che specifica il comportamento di un'area di aggancio di viste implementate dalla classe `TopComponent`.
- `WindowManager`: classe astratta che implementa il gestore delle finestre dell'ambiente NetBeans. In particolare, permette di ottenere la finestra principale dell'IDE/platform con il metodo `Frame getMainWindow()` e di ottenere tutti i contenitori di finestre attivi entro di essa con il metodo di classe `Set getModes()`. Inoltre, con i metodi `Mode findMode(String)` e `Mode findMode(TopComponent)` è possibile ottenere, rispettivamente, l'area di aggancio corrispondente al nome indicato e l'area di aggancio in cui risiede l'istanza della classe `TopComponent` indicata.

Bibliografia

[Ben05] Alex Benini
Estensione di un motore Prolog per la programmazione a vincoli
Università degli Studi di Bologna – Seconda Facoltà di Ingegneria
Tesi in Sistemi Intelligenti Distribuiti LS, A.A. 2004/05, Sessione II
http://www.alice.unibo.it:8080/aliCE/theses.jspx?id=7

[Fow04] Martin Fowler
Inversion of Control Containers and Dependency Injection pattern
http://martinfowler.com/articles/injection.html

[FrAE] *Apache Excalibur*
http://excalibur.apache.org/

[FrJini] *Jini Network Technology*
http://www.sun.com/software/jini/

[FrPC] *Codehaus PicoContainer*
http://www.picocontainer.org/

[FrSF] *Spring Framework*
http://www.springframework.org/

[GoF95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
Design Patterns: Elements of Reusable Object-Oriented Software
Addison-Wesley, 1995

[JBS96] *JavaBeans API specification*
Sun Microsystems, 1996

[JCJO92] Ivar Jacobson, Magnus Christerson, Patrick Jonsson, Gunnar Övergaard
Object-Oriented Software Engineering – A Use Case Driven Approach
Addison-Wesley, 1992

[Nan06] Emanuele Nanetti
Progetto e realizzazione del supporto a teorie logiche multiple nell'interprete tuProlog
Università degli Studi di Bologna – Facoltà di Ingegneria
Tesi in Linguaggi e Traduttori, A.A. 2004/05, Sessione III
http://www.alice.unibo.it:8080/aliCE/theses.jspx?id=29

[NbAPI] *NetBeans API*
http://www.netbeans.org/download/dev/javadoc/

[NbIDE] *NetBeans IDE*
http://www.netbeans.org/products/ide/

[NbOrg] *NetBeans.org*
http://www.netbeans.org/

[NbPlat] *NetBeans Platform*
http://www.netbeans.org/products/platform/

[NbTut1] *Introducing to the NetBeans Idioms and Infrastructure*
http://platform.netbeans.org/tutorials/nbm-idioms.html

[NbTut2] *Recognizing a File Type Tutorial*
http://platform.netbeans.org/tutorials/nbm-filetype.html

[NbTut3] *Introduction to NetBeans Module Development*
http://platform.netbeans.org/tutorials/quickstart-nbm.html

[NbTut4] *NetBeans Anagram Game Tutorial*
http://platform.netbeans.org/tutorials/nbm-windowsapi.html

[RFC2045] *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*
http://tools.ietf.org/html/rfc2045

[RFC2046] *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*
http://tools.ietf.org/html/rfc2046

[RFC2047] *Multipurpose Internet Mail Extensions (MIME) Part Three: Message Header Extensions for Non-ASCII Text*
http://tools.ietf.org/html/rfc2047

Ringraziamenti

Finalmente sono giunto al termine dei miei studi universitari. Non è stato un percorso facile, per questo motivo l'aiuto e il supporto delle persone che durante gli studi mi sono state vicine si è rivelato molto importante.

Desidero ringraziare in modo particolare i miei genitori, per il loro sostegno che non è mai mancato e per gli innumerevoli sacrifici che hanno fatto per consentirmi di studiare. Un altro ringraziamento particolare spetta ad Elena, che grazie al suo supporto affettivo mi ha consentito di superare i momenti più difficili e ha reso i momenti di gioia più belli. Sono molto grato anche a Paolo Caiani, che in questi anni passati ad ingegneria mi è stato vicino come un fratello.

Ringrazio anche tutti i miei compagni di facoltà con cui ho condiviso le sessioni di studio per la preparazione agli esami e le pause caffè; in particolare Luca, Simone, Tatiana, Damiano, Sandro e Pierpaolo.

Devo dire grazie anche tutti quelli che sono riusciti a farmi distrarre dalla mia attività di studio, in particolare ai miei amici Jonathan, David, Giacomo e Valentina sempre disponibili quando avevo bisogno di uscire a Pistoia, ad Alex che mi ha dato l'opportunità di svagarmi a Bologna e a Carmela per le piacevoli pause pranzo.

I miei ringraziamenti vanno anche a chi ha contribuito alla realizzazione di questa tesi, ovvero al Prof. Enrico Denti, per i tanti consigli e per essersi prodigato anche nel miglioramento di questo elaborato, e all'Ing. Giulio Piancastelli, per il suo straordinario appoggio, per la sua infinita pazienza e per il suo meticoloso lavoro di revisione, al quale spetta il merito di avermi consentito di realizzare una tesi di eccellente qualità.

Luca Coccini Gailli