ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA
SEDE DI CESENA
SECONDA FACOLTÀ DI INGEGNERIA CON SEDE A CESENA
CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

PROGETTAZIONE E SVILUPPO DI UN' ARCHITETTURA IDE PER IL LINGUAGGIO CUSTOM WDL (WIZARD DEFINITION LANGUAGE) IN AMBIENTE DISTRIBUITO

Tesi in:

Ingegneria dei sistemi software LM

Presentata da
DAVIDE GALEOTTI

Relatore:
Chiar.mo Prof.
ANTONIO NATALI

II SESSIONE
ANNO ACCADEMICO 2011/2012

PAROLE CHIAVE

IDE

Domain specific languages

CodeMirror

Eclipse

Xtext

Indice

Prefazione ... 9

Introduzione ... 9

Scopo ... 10

1 Integrated Development Environment ... 11

1.1 IDE: Cenni storici ... 11

1.2 IDE: Struttura ... 12

1.2.1 Editor ... 12

1.2.2 Compilatori ed interpreti ... 15

1.2.3 Linker ... 16

1.2.4 Debugger ... 16

1.3 Gli IDE piu' diffusi ad oggi ... 18

1.3.1 IDE open source e free ... 18

1.3.2 IDE commerciali ... 21

2 Linguaggi custom:DSL ... 23

2.1 DSL: Classificazione ... 24

2.1.1 DSL interni ... 24

2.1.2 DSL esterni ... 24

2.1.3 Language Workbench ... 25

2.2 DSL: vantaggi e svantaggi ... 26

2.2.1 Semantic model ... 28

3 Il linguaggio WDL ... 30

3.1 Il Formula Language ... 32

3.1.1 Formula Language: tipi di dato, operatori e costrutti ... 32

3.1.2 Dot Notation ... 33

3.2 Le astrazioni ed espressività del linguaggio WDL ... 35

3.2.1 Il blocco STEP ... 36
3.2.2 Il blocco FOREACHSTEP ... 37
3.3 Il parser generator JavaCC. ... 38
3.3.1 Che cos'è un parser generator? ... 38
3.3.2 Caratteristiche di JavaCC ... 38
3.3.3 JavaCC in WDL ... 40
3.4 Un interprete per WDL ... 43
3.5 WDL è un DSL? ... 45
4 Analisi del problema ... 47
4.1 Architettura dell'applicativo ... 47
4.2 Requisiti ... 51
4.2.1 Requisiti dell'IDE ... 51
4.2.2 Requisiti di sistema ... 55
4.3 Analisi dei requisiti ... 55
4.3.1 Architettura logica del sistema ... 55
4.3.2 Interazioni nel sistema ... 56
5 Soluzione basata su JavaScript code editor ... 58
5.1 Architettura e pattern ... 58
5.2 I source code editor ... 59
5.2.1 JavaScript: accenni ... 59
5.2.2 I più diffusi JavaScript-based code editor ... 60
5.3 CodeMirror ... 63
5.3.1 Overview ... 63
5.3.2 CodeMirror: componenti principali ... 63
5.4 Sviluppi ... 65
5.4.1 WizardEditorMB ... 66
5.4.2 La pagina JSP. ... 69

5.4.3 Syntax Highlighting: il modo i4CMixed ... 72

5.4.4 Implementazione dell'autocompletamento ... 76

5.4.5 Implementazione del folding ... 79

5.4.6 Implementazione comment/uncomment ... 79

5.4.7 Implementazione autosave ... 80

5.4.8 Syntax checking ... 81

5.4.9 Implementazione Go To Line, Find, Replace, Find Previous, Find Next, Match Bracket ... 83

5.4.10 Esecuzione e debugging ... 83

6 Soluzione basata su Xtext ... 84

6.1.1 Definizione della sintassi EBNF in Xtext ... 85

6.1.2 Infrastruttura client-server per l'accesso all'applicativo ... 87

7 Conclusioni ... 90

7.1 Sviluppi futuri ... 91

Appendici ... 93

A. BNF ... 93

Come funziona ... 93

Quando usarla ... 100

B. Syntax Directed Translation ... 101

Come funziona ... 101

Quando è utile ... 109

C. Delimiter Directed Translation ... 111

Come funziona ... 111

Quando è utile ... 114

D. Recursive Descent Parser ... 115

Come funziona ... 115

Tabella riassuntiva ... 117

Quando è utile 118
E. Foreign code 119
Come funziona 119
Quando è utile 121
F. Context Variable 122
Come funziona 122
Quando usarla 122
G. Notification 123
Come funziona 123
Quando usarla 123
H. New line separator 125
Come funziona 125
Quando usarlo 127
I. Symbol Table 128
Come funziona 128
Quando è utile 130
J. Tree Construction 131
Come funziona 131
Quando è utile 133
Indice delle figure 135
Bibliografia 137

Prefazione

La presente Tesi è il risultato di un periodo di Ricerca e Sviluppo presso i4C Analytics - Software Vendor che propone Advanced Analytics per il supporto alle decisioni ed alla pianificazione strategica attraverso applicazioni Industry Specific. Il progetto è stato realizzato all'interno del Team di Sviluppo Prodotto, tra Luglio e Settembre, perciò tutti gli sviluppi sono stati inseriti nelle corrispondenti milestone interne 5.5M4 e 5.5M5.

Introduzione

L'esigenza delle Software House (SH) di rendere il processo di sviluppo del software sempre più produttivo, per stare al passo con mercati in continua evoluzione, viene soddisfatta adottando metodologie agili (es: Scrum, Extreme Programming) e strumenti sempre più sofisticati a supporto di tutte le figure coinvolte nel progetto, siano essi Team Leaders, Developers, Project Managers, Business Analysts o Designer (Domain Experts). Tra questi ultimi, in particolare, ricoprono un ruolo fondamentale i cosiddetti Integrated Development Environment (IDE), ambienti nati per aiutare il programmatore nella produzione di codice ma che, in realtà, accompagnano ogni aspetto della produzione del software dalle fasi di analisi a quelle di test. Mentre i primissimi IDE, negli anni settanta, erano costituiti solamente da un editor ed un interprete realizzati per uno specifico linguaggio, ad oggi, tali strumenti sono diventati vere e proprie piattaforme estendibili che, per la loro flessibilità, si adattano a qualsiasi tipo di progetto.

Il continuo aumento di complessità nei sistemi software e la necessità delle SH di dominarla, ha portato alla rinascita dei cosiddetti Domain Specific Language (DSL), cioè linguaggi dedicati alla risoluzione di aspetti particolari del problema. Si è di nuovo presentata la necessità di adeguare gli stessi strumenti utilizzati per linguaggi general purpose o per DSL standardizzati, anche a linguaggi custom proprietari.

Martin Fowler ha coniato il termine Language Workbenches per definire moderni ambienti di meta-modellazione in cui è possibile creare rapidamente linguaggi custom, gli strumenti per supportarlo (es:editor) e generatori di codice, fondamentali in un approccio ai progetti basato sul Model Driven Software Development per realizzare prodotti Platform Independent.

Spesso però si presentano situazioni in cui le SH non hanno necessità, per svariati motivi, di sviluppare software Platform Independent e l'integrazione di sistemi complessi con i Language Workbenches, in questi casi, può diventare difficoltosa, soprattutto se avviene in fase avanzate dello sviluppo.

Scopo

L'obiettivo della Tesi è stato quello di realizzare un'IDE avanzato per il Wizard Definition Language (WDL), linguaggio custom introdotto da i4C Analytics per incrementare drasticamente la verticalità delle applicazioni: tramite questo DSL, il designer è in grado di affiancare l'utente finale nella modellazione di un proprio dominio e di tutti gli oggetti di business coinvolti, trascurando i dettagli tecnologici dell'implementazione. L'estrema versatilità del WDL, unita alla necessità di produrre/modellare i tantissimi processi dei clienti, ha evidenziato sin da subito l'esigenza di sviluppare (i.e. creare/modellare wizard) direttamente all'interno delle Application, integrando di fatto l'ambiente di sviluppo con quello di esecuzione. La creazione di un IDE vero e proprio ha quindi come obiettivo finale l'efficienza del processo di sviluppo in termini di quantità e qualità del codice prodotto, con un focus molto importante sul Rapid Prototyping di funzionalità ad alto valore strategico per i clienti.

Strumenti quali debugger, syntax checker, semantic checker, syntax colouring, autocompletamento, e molti altri, sono indispensabili per realizzare programmi di qualità in tempi brevissimi. Solitamente queste funzionalità si ottengono senza sforzi da parte delle SH perchè implementate in quasi tutti gli IDE moderni, anche open source. Purtroppo però non è sempre così, in certe condizioni infatti, soprattutto nel mondo dei linguaggi custom, è richiesto uno sforzo da parte della SH, che può essere più o meno importante in base alla soluzione con cui le funzionalità si realizzano. La tesi ha perciò anche l'obiettivo di determinare soluzioni alternative che possano ridurre costi di questo tipo massimizzando i profitti, intesi come funzionalità ottenute per il linguaggio WDL.

1 Integrated Development Environment

1.1 IDE: Cenni storici

Una tipica esigenza nell'implementazione del software è potersi muovere in un ambiente prettamente dedicato, che faciliti tutte le più comuni azioni di editing, compilazione, linking e debugging in modo da agevolare il programmatore e consentirgli di ottimizzare i tempi di sviluppo.

Per tale motivo fin dagli anni '70 si è fatta sempre più pressante l'esigenza di avere un ambiente di sviluppo, che trova la sua prima realizzazione nel Cornell Program Synthesizer[1]: un syntax-directed programming environment in grado di creare, editare, effettuare debug e lanciare programmi. Molto importante è che i programmi non sono più visti come semplici testi, ma come strutture sintattiche derivate dalla definizione formale della grammatica di un linguaggio (approfondimenti alle Appendici A e B). In questo modo è possibile fornire una correzione interattiva durante la scrittura del codice in modo da liberare il programmatore da frustranti "dettagli sintattici".

L'aggettivo *integrated* compare più avanti nel tempo e indica la capacità di gestire le interdipendenze fra documenti in modo da mantenere uno sviluppo incrementale dei progetti software: le modifiche ad un file si ripercuotono automaticamente su tutti i documenti collegati.

Verso la fine degli anni ottanta compaiono i primi sistemi che forniscono le linee guida alla base dei moderni IDE, PSG System e CENTAUR. Le caratteristiche fondamentali che possiamo individuare in questi due sistemi sono:

- un'interfaccia grafica unica che permetta di integrare tutti gli strumenti di supporto per l'intera fase del processo di sviluppo software;
- un'immediata risposta a ogni input del programmatore (segnalazione degli errori in primis);
- un modello formale di alto livello in cui rappresentare tutti i documenti di lavoro del programmatore, comunemente sotto forma di abstract syntax tree;
- un sistema uniforme di manipolazione di questo modello utilizzabile da tutti gli strumenti dell'IDE, in grado di massimizzare il riutilizzo del codice.

Un ulteriore importante aspetto di PSG e CENTAUR è la genericità: è possibile creare tutti gli strumenti che compongono un IDE per qualsiasi linguaggio di programmazione in maniera semi-automatica, o meglio assistita,

partendo dalla definizione formale di tale linguaggio. Possiamo quindi vedere i due progetti come IDE-generator. Questo concetto è ripreso venti anni più tardi dal framework Xtext utilizzato nella fase implementativa di questa tesi.

Durante gli anni novanta sono stati sviluppati centinaia di IDE per diversi linguaggi di programmazione e aggiunte nuove funzionalità come navigazione del codice, funzioni di auto completamento, suggerimenti, analisi semantiche più o meno avanzate, sistemi di refactoring e tanto altro ancora, ma le caratteristiche comuni restano quelle delineate dai due progetti citati in precedenza.

Verso la fine degli anni novanta sono nati due fra gli IDE più conosciuti, apprezzati e utilizzati nel mondo opensource: Eclipse e Net-Beans. La popolarità è dovuta al linguaggio Java per cui sono stati creati e alla predisposizione all'estendibilità. Allo stato dell'arte esistono infatti plug-in per ogni aspetto dello sviluppo software.

Un IDE rappresenta quindi un importante strumento per aumentare la produttività di programmatori professionisti, ma è anche un ottimo punto di partenza per coloro che si avvicinano per la prima volta a un particolare linguaggio.

1.2 IDE: Struttura

Ad oggi gli IDE sono normalmente accomunati da una struttura simile che comprende:

- un editor di codice sorgente
- un compilatore (interprete)
- un linker
- un debugger
- strumenti di utilità:
    - editor di risorse
    - class browser
    - code profiler

1.2.1 Editor

L'editor è dove viene fisicamente scritto, ovvero editato, il codice sorgente.

Un editor può essere di diversi livelli di complessità, a partire dalla sua natura più semplice: una pagina di testo, si è sempre più evoluto arrivando a fornire strumenti di grande utilità quali

controlli semantici e sintattici, questi ultimi spesso con l'ausilio di forme grafiche immediate quali *syntax highlighting* che consente di visualizzare il codice con differenti colorazioni in base alle specifiche regole sintattiche. I controlli semantici accertano che ci sia congruenza fra le dichiarazioni delle entità e il loro impiego nelle istruzioni oltre al rispetto delle regole che governano i tipi degli operandi

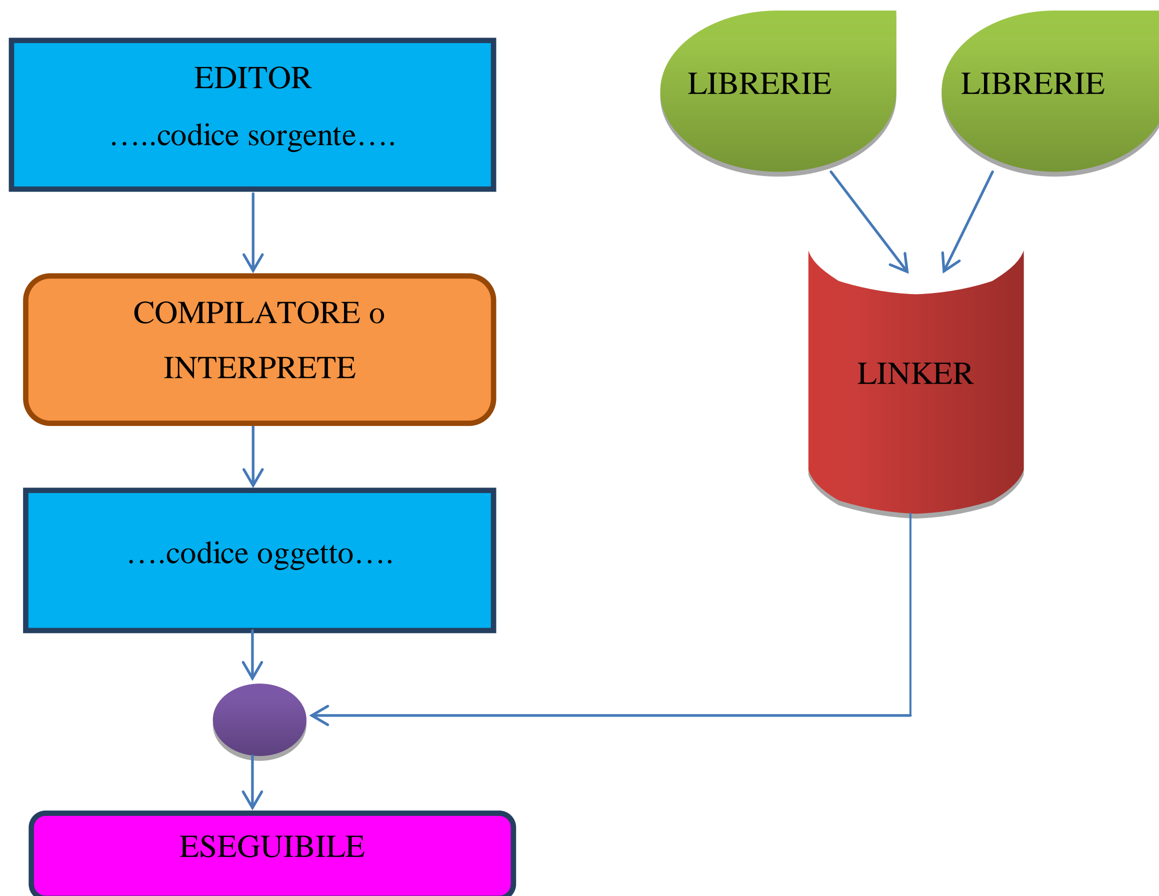


Figura 1: Architettitura generale di un IDE

Il syntax colouring avanzato permette di individuare, grazie a differente colorazione, linguaggi diversi nello stesso codice (es: Html, css , javascript).

Tra le opzioni di visualizzazione specifiche è bene ricordare il go to line che permette di andare automaticamente alla linea desiderata; l'autosave, salvataggio automatico a intervalli di tempo regolari e/o alla chiusura della finestra, fullscreen per una visualizzazione a tutto schermo.

```
31⊖     /*
32       * GETTERS
33       */
34⊖     public String getName() {
35         return name;
36     }
37
38⊖     public String getDescription() {
39         return description;
40     }
41
42⊖     public AuditForCreation getAuditForCreation() {
43         return auditForCreation;
44     }
45
46⊖     public Long getCodEntityType() {
47         return codEntityType;
48     }
49
50⊖     public String getQueryString() {
51         String queryString = webSearchObject.toString();
52         List<WebAggregationCriteria> acList = webSearchObject.getAggregationCriteriasList();
53         if (acList != null && acList.size() > 0) {
54             queryString += " - Aggregated by: ";
55             for (int i = 0; i <acList.size(); i++) {
56                 WebAggregationCriteria wac = acList.get(i);
57                 queryString = queryString + "(" + (i + 1) + ") " + wac.getFieldName() + " - ";
58             }
59             queryString = queryString.substring(0, queryString.length() - 3);
```

Figura 2: Syntax highlighting di un sorgente Java in Eclipse

Per rendere il codice più ordinato e comprensibile è a volte disponibile l'indentazione automatica. Per velocizzare il lavoro del programmatore invece l'editor viene spesso fornito di shortcut ad esempio per commentare o rimuovere i commenti, oltre al find/replace che permette una ricerca con eventuale sostituzione dei termini desiderati.

Il code folding permette di nascondere delle porzioni di un file di codice mentre si lavora ad altre parti dello stesso file.

Un'altra funzione avanzata, solitamente molto gradita ai programmatori, consiste nel completamento automatico del codice, che permette di ottenere il nome completo di un metodo, una variabile o anche un intero costrutto battendo solamente alcuni caratteri.

I suggerimenti, permettono una più agevole composizione dei costrutti; quando si scrive ad esempio un ciclo do o while, una diramazione if o switch, viene automaticamente mostrata la struttura di tali costrutti, comprensiva di parentesi e segnaposti (*placeholders*). Ecco l'autocompletamento relativo al costrutto if:

```
if (condition) {
    statements
}
```

Figura 3: Esempio di autocompletamento

E' poi possibile sovrascrivere il suggerimento scrivendo l'espressione corrispondente.

Lo scopo di questi strumenti, oltre ovviamente a semplificare la stesura di un codice, non è solo quello di risparmiare tempo ma anche quello di evitare errori di ortografia.

1.2.2 Compilatori ed interpreti

Un compilatore è un programma che traduce, ovvero compila, una serie di istruzioni che formano il codice sorgente, in istruzioni di un altro linguaggio formando il codice oggetto. L'interprete permette invece di eseguire il codice sorgente traducendolo di volta in volta in linguaggio macchina senza una compilazione vera a propria, senza quindi salvare il codice oggetto.

Le istruzioni di partenza sono scritte in un linguaggio di programmazione, solitamente ad alto livello, chiamato linguaggio sorgente (C, C++, Pascal, BASIC, Java, ...) mentre il codice oggetto risulta solitamente in un linguaggio di più basso livello (assembly) o linguaggio macchina.

Le fasi della compilazione sono:

- analisi lessicale (lexer o scanner o tokenizer)
- pre-processamento (preprocessor)
- analisi sintattica (parser)
- analisi semantica
- ottimizzazione del codice (ottimizzatore)
- generazione del codice oggetto

Mediante l'**analisi lessicale** un compilatore suddivide il codice sorgente in una sequenza di unità lessicali dette token; un token è un'unità atomica del linguaggio come parole chiave (while, for, ...), identificatori (nomi di variabili, funzioni, classi, …), operatori (+, =, --, ...), valori numerici, ecc.

La fase di **preprocessamento** consiste nella sostituzione di macro e nell'esecuzione di direttive di inclusione e compilazioni condizionali. Tipicamente la fase di preprocessamento precede l'analisi sintattica.

L'**analisi sintattica** esegue il controllo sintattico sulla sequenza di token generata nella fase di analisi lessicale. Il controllo sintattico è effettuato attraverso un grammatica che definisce la sintassi (regole sintattiche) del linguaggio di programmazione. Il risultato di questa fase è un albero di sintassi detto parse-tree.

L'**analisi semantica** controlla il significato (la semantica) delle istruzioni presenti nel codice in ingresso. Controlli tipici di questa fase sono ad esempio il type checking (controllo sui

tipi) e il definite assignment (controllo che tutte le variabili locali siano state inizializzate prima di essere utilizzate).

Se si sta utilizzando un interprete le operazioni terminano qui e viene eseguito direttamente il codice. Il compilatore invece prosegue con i passi successivi.

Mediante la fase di **ottimizzazione** il codice in linguaggio intermedio è trasformato in forme equivalenti ma più veloci (o di dimensioni inferiori). Esempi di ottimizzazioni sono

- espansione di funzioni inline
- eliminazione di codice non raggiungibile
- trasformazione di cicli
- Return Value Optimization (RVO)

Nella fase di **generazione del codice** il codice in linguaggio intermedio viene tradotto nel linguaggio oggetto, solitamente in linguaggio macchina. E' possibile configurare il processo di compilazione (di generazione) mediante l'impostazione delle opzioni di compilazione (di link)

- definizione di macro (per impostare valori e controllare compilazioni condizionali)
- impostazione di directory di inclusione e di librerie (dove ricercare header file e librerie riferite nel programma)
- elenco delle librerie da utilizzare nella generazione del programma

1.2.3 Linker

Un linker è un programma che prende in ingresso uno o più file oggetto (generati dal compilatore) e li collega in un singolo eseguibile e alle librerie. Un file oggetto può, infatti, contenere dei riferimenti a simboli (variabili, funzioni, classi) esterni, ossia non definiti all'interno del file oggetto.

E' compito del linker risolvere i riferimenti ai simboli esterni cercando tali simboli nei file oggetto (librerie) passatigli in input; se un simbolo esterno non viene trovato il linker genera un errore del tipo "simbolo esterno non risolto" (unresolved external symbol).

1.2.4 Debugger

Il *debugger* è un software specificatamente progettato per la ricerca, l'individuazione e l'eliminazione dei bug, ovvero errori che impediscono il corretto funzionamento, presenti in altri programmi.

Il debugger individua la porzione di codice macchina che genera il blocco e lo mostra sul codice sorgente, solitamente utilizzando un disassembler integrato.

Quando il debugger è integrato nell' IDE spesso permette un'esecuzione del programma step-by-step, avendo preventivamente localizzato dei break-point. In questo modo è possibile lavorare sulle singole porzioni di codice risolvendo eventuali errori successivi in modo sequenziale senza dover intervenire sull'interezza del codice che potrebbe essere molto complesso.

1.3 Gli IDE piu' diffusi ad oggi

Come precedentemente accennato, in pochi anni, si sono evoluti centinaia di IDE alcuni specifici per determinati linguaggi e altri a carattere più generico. Alcuni di questi hanno raggiunto un'elevatissima diffusione grazie alla possibilità di essere utilizzati per differenti linguaggi di programmazione e per alcune caratteristiche che li hanno resi vincenti nel panorama degli ambienti di sviluppo integrati. In particolare si può affermare che i più noti e usati sono attualmente Eclipse, IntelliJ IDEA, NetBeans che saranno in seguito trattati uno ad uno, cercando di evidenziarne i lati vincenti che li hanno portati ad avere così grande fama.

Le principale differenza caratterizza la natura di questi strumenti: Eclipse e NetBeans sono open source e free, ciò probabilmente ha contribuito enormemente alla loro diffusione. IntelliJ IDEA e Visual Studio sono software commerciali, la cui fama è dovuta alle ottime qualità dei prodotti, che li rendono competitivi anche in ambienti solitamente ad oggi più ostici come quelli del software a pagamento.

1.3.1 IDE open source e free

I grandi vantaggi di questi IDE sono indubbiamente l'elevata fruibilità e la presenza di comunità di utenti attive e operose, che arricchiscono continuamente i prodotti sviluppando e mettendo a disposizione nuove funzionalità. Ovviamente gli svantaggi sono quelli comuni a tutti i progetti open source ovvero non poter contare su un'assistenza dedicata ma doversi necessariamente appoggiare a forum ed altri utenti oltre al rischio di non avere costanti aggiornamenti se la comunità perde l'interesse verso il progetto.

1.3.1.1 Eclipse

Eclipse è un IDE open source sviluppato inizialmente da IBM. Successivamente il progetto fu reso open source e ceduto ad un consorzio che include tra le varie aziende: *IBM*, *Borland*, *Merant*, *QNX Software*, *Rational*, *Red Hat*, *SuSE*, *TogetherSoft*, *Oracle*, *HP*, *Sybase*.

La sola IBM ha investito in Eclipse circa 40 milioni di dollari prima di renderlo open source anche se attualmente continua lo sviluppo di Eclipse tramite una società sussidiaria chiamata *OTI (Object Technologies International).*

Eclipse si è rivelata un successo fin dalla relase 1.0 per diversi motivi. In primo luogo è l'evoluzione di una IDE della IBM sviluppata da OTI nel 1996 in smalltalk: Visual Age for Java (VA4J) che applicava a Java molte idee innovative che orbitavano intorno a SmallTalk. Per citare due esempi significativi: non esisteva il concetto di file e si effettuava versioning a livello dei metodi.

Eclipse, come già detto, è un progetto open source che viene gestito dal consorzio mediante il "Board of Stewards": ovvero c'è un rappresentante per ogni azienda membro del consorzio e la commissione determina gli obiettivi del progetto sulla base di due principi generali: supportare e favorire una comunità open-source vitale e dinamica e creare opportunità commerciali per i membri del consorzio. Dal punto di vista operazionale il consorzio definisce i "Project Management Committee" (PMC) che gestiscono i vari progetti che compongono Eclipse.

Sostanzialmente i progetti fondamentali sono:

- La piattaforma Eclipse (*Eclipse Platform*)
- Il toolkit di sviluppo java (*Java ToolKitJava Development ToolKit*,JDT) [Eric Gamma]
- Il toolkit di sviluppo C/C++ (*C++ Development ToolKit* , CDT)
- L'ambiente di sviluppo di plug-ins (*Plug-in EnvironmentPlug-in Development Environment*,PDE)

Eclipse viene fornito con diversi modalità di packaging:

- Eclipse SDK che contiene platform, JDT e PDT, docs, binari e sorgenti
- Eclipse Platform Run-time che contiene solo i binari del platform
- Eclipse Platform SDK che contiene binari e sorgenti del platform
- Componenti singoli

E' importante osservare che JDT,CDT e PDT sono sostanzialmente aggiunte al platform. Il cuore di Eclipse è la Eclipse Platform il cui scopo è fornire solo i servizi necessari per integrare gli strumenti software di più alto livello che vengono implementati ad un livello superiore come plug-ins.

L'aspetto più interessante del progetto di Eclipse è proprio questo: eccetto i servizi forniti da un piccolo kernel a run-time, ogni altra caratteristica di Eclipse è fornita da un insieme di plug-in comunicanti. Questo lo rende un'IDE estremamente versatile in quanto modulare ed estensibile, adattabile alle diverse esigenze del singolo programmatore. Solitamente un aspetto cruciale delle architetture a plug-in è quello prestazionale dato che i plug-in possono divenire un numero elevatissimo: Eclipse utilizza una politica di load on demand dei plug-in.

Questo minimizza la quantità di memoria utilizzata ed il tempo di startup nel quale diventa necessario solo rilevare (ma non istanziare) tutti i plug-in.

Altro punto vincente di questo IDE è la neutralità rispetto al linguaggio di programmazione, il risultato di ciò è che molti dettagli di utilizzo pratico non risultano standardizzati e possono adattarsi agilmente ai diversi linguaggi.

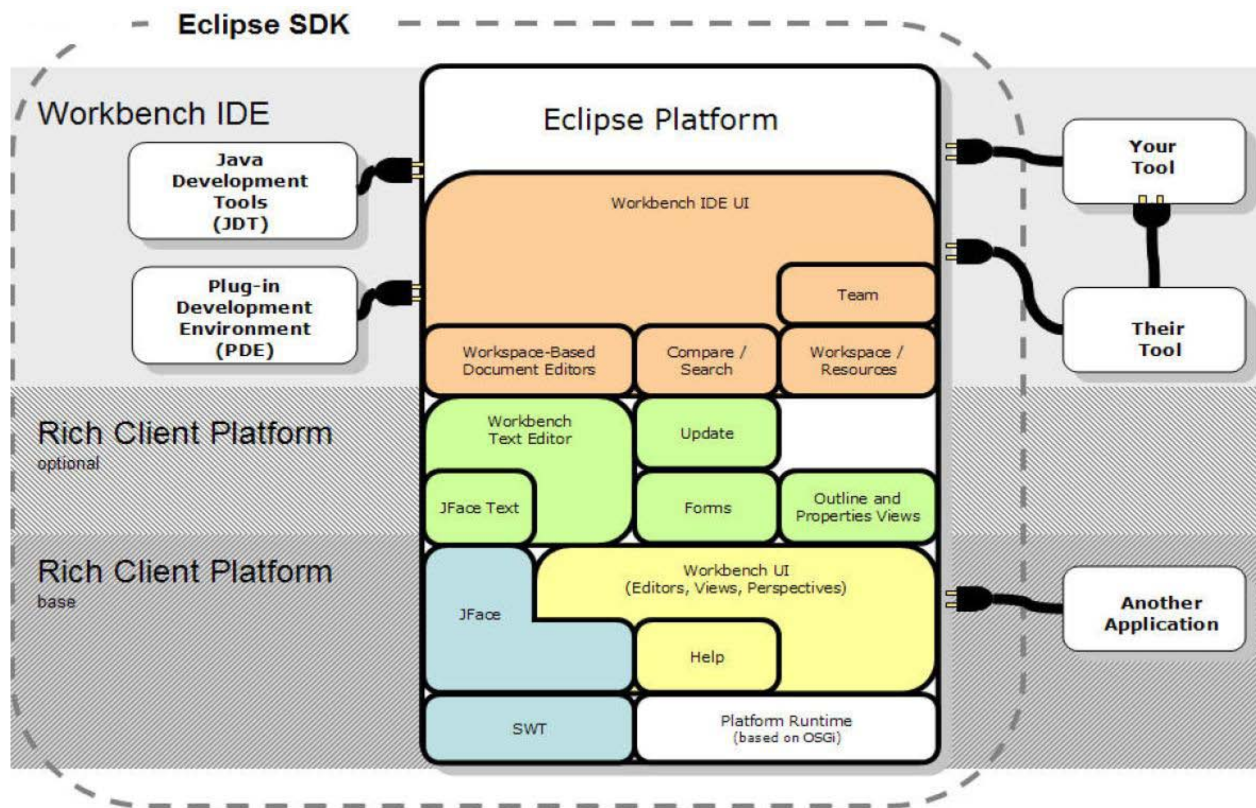


Figura 4: Archittettura generale di Eclipse

1.3.1.2 NetBeans

NetBeans ha avuto le sue origini nel 1996 come progetto studentesco chiamato inizialmente Xelfi in Cecoslovacchia (Repubblica-Ceca). L'obiettivo era quello di scrivere un IDE Java simile a Delphi, Xelfi è stato il primo IDE Java scritto in Java.

Al termine degli studi gli studenti provarono a commercializzare il loro prodotto incontrando l'interesse di un imprenditore e creando JavaBeans e rendendo disponibile in rete lo sviluppo di componenti.

In questo modo veniva mantenuto un approccio a Plug-in simile a quello di Eclipse.

In poco tempo ci fu un'importante diffusione e molti utenti sviluppavano i loro plug-in rendendolo disponibili in rete. Dal 1999 Sun Microsystems era alla ricerca di migliori strumenti di sviluppo Java e guardò con interesse a NetBeans divenenedone lo sponsor e rendendolo l'IDE standard per Java. Quando la Oracle acquisì la Sun, nel 2010, divenne a sua volta lo sponsor e lo

rese il suo IDE ufficiale. Ad oggi NetBeans è distribuito in forma integrata con l'SDK ed è open-source come Eclipse.

E' importante far notare che la disponibilità dei plugin di NetBeans è molto più ridotta rispetto a quella di Eclipse, NetBeans supporta i seguenti linguaggi di programmazione: Java, JavaScript, PHP, (J)Ruby (e la variante "on Rails"), C e C++;

1.3.2 IDE commerciali

Gli IDE commerciali non sono fruibili gratuitamente ma vengono vendute dalle software house produttrici. Negli ultimi anni, però, sta prendendo piede l'abitudine di affiancare al prodotto commerciale anche una versione più limitata, disponibile in forma gratuita o open source.

In generale questo tipo di prodotti fornisce un'attenta assistenza al cliente e, in genere, ampia disponibilità di manuali d'uso oltre a continui aggiornamenti. Ne vengono citati qui due tra i più noti ma se ne contano numerosissimi.

1.3.2.1 IntelliJ idea

Decisamente recente, la sua prima versione è uscita nel 2001, è prodotto dalla JetBrains ed è un IDE orientato prevalentemente a Java ma che supporta un grandissimo quantitativo di linguaggi. Nella versione base sono infatti compresi: Java, JavaScript, CoffeeScript, HTML/XHTML/CSS, XML/XSL, ActionScript/MXML, Python, Ruby/JRuby, Groovy, SQL, PHP.

Anche questo software dispone di plugins che ne incrementano le potenzialità, in particolar modo alcuni plugin lo rendono in grado di supportare ulteriori linguaggi di programmazione quali: Scala, Clojure, JavaFX 1, Dart, haXe, Kotlin, TypoScript. Da notare che i plugins sono organizzati in un repository disponibile sul sito ufficiale.

La JetBrains offre anche una versione open source dell'IDE, disponibile sul sito ufficiale, chiamata IntelliJ IDEA Community Edition, sicuramente più limitata nei contenuti ma ampiamente personalizzabile grazie alla disponibilità del codice.

IntelliJ IDEA attualmente pubblicizza come punti di forza un completamento del codice veloce e intelligente che sa quando è possibile eseguire il cast ad un determinato tipo, ha un l'editing multilinguaggio riconoscendo le iniezioni di altri linguaggi di programmazione all'interno del principale, come, ad esempio può essere Java HTML all'interno di JavaScript, ecc. Per queste inclusioni l'editor fornisce adeguata assistenza in linea e un apposito editor per la porzione di codice da inserire. Inoltre vi è la Brand New Cross-Platform UI ovvero una

Interfaccia utente completamente ridisegnata che rende IntelliJ IDEA più leggero, flessibile e personalizzabile[2].

1.3.2.2 Visual Studio

Visual studio è un IDE proprietario della Microsoft, di cui esistono diverse versioni, nel tipico stile dell'azienda, a seconda del grado di professionalità che si desidera dal prodotto: ultimate, premium e professional. Anche la Microsoft ha però pensato di rilasciare una versione gratuita Visual Studio Express Edition, dedicata a studenti e principianti ma limitata su certi aspetti funzionali.

Fin dalla sua prima versione, datata 1997, la sua missione era già quella di fornire un ambiente di sviluppo grafico ed integrato che aiutasse lo sviluppatore a gestire i progetti in maniera semplice, ma efficace, aumentandone quindi la produttività. La novità principale fu che, per la prima volta, Microsoft incluse, in un solo prodotto, il supporto a differenti linguaggi (C++, J++).

L'intenzione era quella di ridurre la complessità attuale, in cui ogni linguaggio o tecnologia possedeva uno strumento dedicato ed obbligava lo sviluppatore a dover familiarizzare con molti ambienti differenti.

La versione successiva, Visual Studio 6.0, rimase sul mercato per quattro anni e fu una versione di "transizione" perché nel 2002 uscì la prima versione del **Visual Studio .NET**, il cui nome deriva dal framework **.NET** e di cui la versione attuale è diretta evoluzione[3].

Dalla prima versione è iniziata quindi una serie di upgrade cadenzati con i quali Microsoft rilascia una nuova versione orientativamente ogni due anni ed un service pack negli anni dispari.

Ad oggi Visual studio supporta diversi linguaggi di programmazione quali C, C++, C#, F#, Visual Basic .Net e ASP .Net ma la sua caratteristica originale è sicuramente il designer visuale dei forms, da cui l'aggettivo visual nel nome.

I punti di forza pubblicizzati dall'azienda sono la possibilità di lavorare con lo stesso IDE per creare soluzioni per web, desktop, cloud, server e phone. In particolare per il cloud vi è la possibilità di trasferire le applicazioni a Windows Azure, compresi nuovi modelli e opzioni di pubblicazione oltre al supporto per il caching distribuito.

Peculiarità è inoltre il potersi appoggiare a Windows Store, un canale di distribuzione ampiamente disponibile, che può raggiungere utenti ad esso registrati e consentire di vendere il codice prodotto.

2 Linguaggi custom:DSL

Il termine “custom” sta ad indicare semplicemente che il linguaggio è stato realizzato all’interno di un’azienda e che quindi si adatta alle esigenze specifiche di quest’ultima. I linguaggi di questo tipo, ma, in realtà, anche i linguaggi non-custom, si possono suddividere in due categorie: linguaggi general-purpose e Domain Specific Language. Questi ultimi, in particolare, stanno tornando molto in voga negli ultimi anni e per questo di seguito verranno approfonditi.

"Domain Specific Language" è un insieme di termini che esprime già di per sè un concetto, ma non permette una definizione dai confini ben definiti in quanto alcuni linguaggi sono chiaramente DSL, ma altri possono essere considerati appartenenti o meno a tale definizione.

Dalla traduzione letterale: il “linguaggio di dominio specifico” è un linguaggio di programmazione di espressività limitata focalizzato su un particolare dominio[4].

Ci sono quattro elementi chiave di questa definizione:

- Linguaggio di programmazione del computer: un DSL è utilizzato dagli utenti per ordinare a un computer di fare qualcosa. Come con qualsiasi linguaggio di programmazione moderno, la sua struttura è stata progettata per una facile comprensione da parte degli esseri umani, ma dovrebbe rimanere un qualcosa di eseguibile da parte di un computer.
- Natura del linguaggio: Un DSL è un linguaggio di programmazione, e come tale deve dare un senso di in cui l'espressività non deriva soltanto da espressioni a se stanti ma da come queste vengono composte tra loro.
- Limitata espressività: un linguaggio di programmazione fornisce moltissime funzioni: supporto di vari tipi di dati, controllo e strutture per gestire le astrazioni; tutto ciò rende più difficile imparare ad utilizzarlo. Un DSL supporta solamente le funzionalità necessarie per sostenere il proprio dominio. Questo fa sì che non si possa costruire un intero sistema software in un DSL, ma che si utilizzi un DSL per un aspetto particolare di un sistema.
- Domain focus: un linguaggio limitato è funzionale soltanto se ha un chiaro focus su un piccolo dominio.

I linguaggi Domain Specific Language o DSL sono quindi linguaggi creati appositamente per affrontare un particolare tipo di problematica, piuttosto che i linguaggi di uso generale, diretti a qualsiasi tipo di problema software.

Sono linguaggi molto comuni in informatica, un tipico esempio sono CSS, SQL, HQL, ecc orientati a funzioni specifiche e molto performanti in tale utilizzo.

Secondo Fowler, è doveroso fare una prima grande distinzione tra DSL interni o esterni e language workbench[4].

2.1 DSL: Classificazione

2.1.1 DSL interni

Un DSL interno è un modo particolare di usare un linguaggio general-purpose. Uno script in un DSL interno è un codice valido nel suo linguaggio general-purpose, ma utilizza solo un sottoinsieme delle funzionalità del linguaggio in uno stile specifico per gestire un piccolo aspetto del sistema complessivo. Il risultato deve dare la sensazione di una lingua personalizzata, piuttosto che di essere un linguaggio ospite. L'esempio classico di questo stile è Lisp; i cui programmatori spesso descrivono la programmazione Lisp come la creazione e l'utilizzo di un DSL. Ruby ha anch'esso sviluppato una forte cultura DSL: molte librerie Ruby sono nello stile dei DSL. In particolare, il framework più famoso di Ruby, Rails, è spesso visto come un insieme di DSL.

Un DSL interno è quindi uno stile idiomatico di scrittura del codice in un linguaggio di programmazione general-purpose. I DSL interni non necessitano di alcun riconoscitore specifico, ma vengono riconosciuti proprio come qualsiasi altro programma scritto nel general-purpose, per questo motivo sono più facili da creare, non richiedendo un riconoscitore speciale. Per contro, i vincoli del linguaggio sottostante limitano le modalità con cui esprimere i concetti del dominio.

2.1.2 DSL esterni

Un DSL esterno è un linguaggio separato dal linguaggio principale dell'applicazione, si differenzia dall'interno in quanto è personalizzato con la propria grammatica e il proprio riconoscitore.

Di solito, un DSL esterno ha una sintassi personalizzata, ma a volte capita che utilizzi la sintassi di un altro linguaggio (XML viene scelto di frequente). Uno script in un DSL esterno di solito è analizzato da un codice nell'applicazione host utilizzando tecniche di analisi del testo.

Non ha il vincolo di un DSL interno che è comunque legato al linguaggio sottostante: il DSL esterno può essere progettato come si desidera, purché venga scritto un riconoscitore affidabile. Infatti, lo svantaggio di un DSL esterno è proprio questo: la necessità di creare e usare un riconoscitore personalizzato.

Esempi di DSL esterni possono essere SQL, CSS, e file di configurazione XML per sistemi come Struts e Hibernate.

2.1.3 Language Workbench

Un Languaggio Workbench è un IDE specializzato per la definizione e la costruzione di DSL. In particolare, è usato non solo per determinare la struttura di un DSL, ma anche come ambiente di editing su misura per i programmatori di script DSL. Gli script risultano così strettamente collegati sia all'ambiente di editing che al linguaggio.

La diffusione sempre maggiore che i DSL hanno avuto negli ultimi anni ha portato alla nascita di diversi Language Workbench. Il più completo e diffuso è senza dubbio Xtext.

2.1.3.1 Xtext

Xtext è un framework DSL sviluppato da Itemis, una società specializzata in modelbased programming, in collaborazione con l'Eclipse Foundation e rilasciato in modalità opensource. Il suo scopo è quello di supportare il programmatore nello sviluppo di linguaggi di dominio specifici. E' infatti stato concepito per limitare lo sforzo del progettista alla descrizione di un DSL con una semplice notazione in stile EBNF (Extended Backus Naur Form). A partire dalla grammatica del linguaggio definita dall'utente, Xtext riesce a generare automaticamente un parser, un modello Ecore che ne descrive la sintassi astratta e un editor testuale per Eclipse.

In sintesi Xtext permette di generare un editor per un linguaggio partendo da una grammatica in notazione EBNF. Il parser utilizzato internamente da Xtext è ottenuto dal parser generator ANTLR integrato nel framework.

Le sue caratteristiche hanno fatto in modo che trovasse ampio spazio nel settore della telefonia mobile, automobilistico, sistemi embedded e nell'industria dei videogiochi.

E' infatti uno strumento molto versatile che permette l'implementazione completa di linguaggi, di cui la generazione di un IDE è soltanto un aspetto. E' progettato per i DSL ma può benissimo essere utilizzato anche per linguaggi general purpose.

Una serie di API e DSL interni sono disponibili per descrivere il linguaggio di programmazione che si desidera implementare, permettendo di ottenere un'implementazione completa basata su componenti che si appoggiano semplicemente alla Java VM e non a Eclipse. I componenti prodotti con Xtext come il parser, l'AST, lo scoping framework, il validator e

l'interpeter sono quindi indipendenti da Eclipse e possono essere utilizzati anche in modalità stand-alone.

Gli strumenti generati da Xtext sfruttano EMF e quindi permettono l'integrazione con strumenti collegati a EMF come ad esempio Graphical Modeling Framework (GMF), un framework per l'editing grafico di modelli molto versatile.

Con Xtext è possibile inoltre ottenere un editor Eclipse-based per il linguaggio che offre tutte le funzionalità classiche di un IDE, estendibili in modo semplice tramite le API fornite o inserendo componenti esterni.

Grazie a XText, le entità di un dominio possono essere descritte definendo uno specifico linguaggio testuale capace di esprimere le proprietà ricavate dai requisiti o dal colloquio con il committente[5].

Il funzionamento del framework è facilmente descrivibile in quanto X-Text fornisce un quadro ad alto livello che genera la maggior parte dei manufatti tipici e ricorrenti necessari per un vero e proprio IDE su Eclipse. In XText, la sintassi del linguaggio è definita utilizzando una grammatica EBNF. A partire da questa grammatica, il generatore XText crea un parser, un AST-meta modello, nonché un editor completo basato su Eclipse. I plugin generati da XText implementano già la maggior parte degli artefatti ricorrenti per i liguaggi IDE, e possono essere facilmente personalizzati grazie a "iniezioni" delle specifiche implementazioni dei meccanismi linguistici[6].

Il successo di questo strumento è fondamentalmente dovuto al fatto che sia basato ed integrato in Eclipse. Escluso MS Visual Studio dal principio per la natura non opensource, Eclipse è stato preferito a NetBeans per la sua maggiore diffusione la disponibilità di un maggior numero di plug-in e la community di supporto.

Inoltre dispone di un gran numero di funzionalità offerte: permette di specificare referenze incrociate durante la definizione della grammatica per il linguaggio target, che oltre le funzionalità classiche di un IDE permette di rendere il codice navigabile, collegando ad esempio la chiamata di una funzione alla sua definizione.

2.2 DSL: vantaggi e svantaggi

Se ben progettati, i DSL offrono diversi vantaggi, come ad esempio:

- INCAPSULAMENTO: Un DSL nasconde i dettagli di implementazione ed espone solo quelle astrazioni che sono rilevanti per il dominio.

- EFFICIENZA: Dato che i dettagli di implementazione sono incapsulati, un DSL ottimizza lo sforzo richiesto per creare o modificare le funzionalità di una applicazione.
- COMUNICAZIONE: Un DSL aiuta gli sviluppatori a capire il dominio, e gli esperti di dominio a verificare che l'implementazione soddisfi i requisiti.
- QUALITÀ: Un DSL minimizza il "disadattamento di impedenza" tra i requisiti funzionali, nel modo in cui vengono espressi dagli esperti del dominio, e il codice sorgente dell'implementazione, limitando così i potenziali errori.

Tuttavia, i DSL hanno anche alcuni svantaggi che è bene focalizzare:

- DIFFICOLTÀ DI IMPLEMENTARE UN DSL DI BUONA QUALITÀ: Un buon DSL è più difficile da progettare rispetto alle API (Application Program Interface) tradizionali, anche se le difficoltà nel progettare una API elegante, efficace e facile da usare non mancano. Le API tendono a seguire gli idiomi del linguaggio di programmazione allo scopo di favorire l'uniformità, che per una API è una caratteristica importante. Al contrario, ogni DSL dovrebbe riflettere gli idiomi linguistici specifici del proprio dominio. Il progettista di DSL gode di una maggiore libertà di azione, ma questo significa anche che è molto più difficile determinare le scelte di progettazione "migliori".
- MANUTENZIONE DI LUNGO TERMINE: I DSL possono richiedere una manutenzione più frequente nel lungo termine per tenere conto dei cambiamenti nel dominio. Inoltre gli sviluppatori avranno bisogno di più tempo per imparare a usare e a mantenere un DSL.

2.2.1 Semantic model

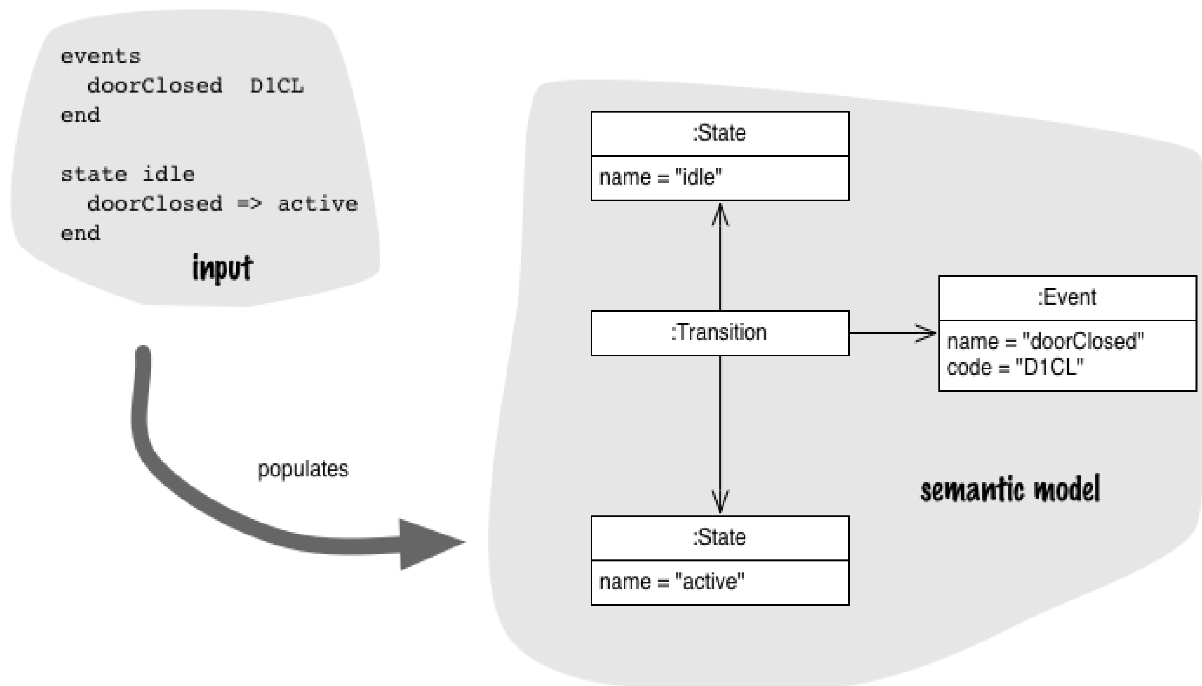


Figura 5: Relazione tra DSL e Semantic model

Nel contesto di una DSL, un modello semantico è una rappresentazione, come un oggetto memorizzato, dello stesso soggetto descritto dalla DSL. Ad esempio, se il DSL descrive una macchina a stati, allora il modello semantico dovrà essere un oggetto modello con classi per stati, eventi, ecc. Uno script DSL descrive particolari stati ed eventi che dovrebbero corrispondere a particolari popolazioni dello schema, nel quale un'istanza di ogni evento, per ogni evento dichiarato nello script DSL. Il semantic model è quindi la libreria o il framework che il DSL popola.

Il modello semantico dovrebbe essere progettato attorno allo scopo della DSL. Per una macchina a stati, lo scopo è quello del controllo del comportamento utilizzando un modello computazionale di una macchina a stati. Infatti, il modello semantico dovrebbe essere utilizzabile senza un che ci sia un DSL; dovrebbe essere possibile compilare un modello semantico attraverso un'interfaccia comand-query.

Questo assicura che il modello semantico catturi completamente la semantica del dominio e permetta di testare in modo indipendente sé stesso e il parser.

La nozione di modello semantico è molto simile a quella di modello del dominio. Si usano termini differenti in quanto, sebbene i modelli semantici sono spesso un sottoinsieme dei modelli del dominio, non è detto che vi appartengano sempre. Con Domain Model, infatti, ci si riferisce ad un modello ad oggetti ricco dal punto di vista comportamentale, mentre un modello

semantico può essere costitutito soltanto da dati. Un modello del dominio cattura il comportamento “core” dell’applicazione, mentre un modello semantico può giocare solo un ruolo di supporto. Un buon esempio per spiegare questi concetti è quello di un object-relational mapper che mappa le coordinate tra un object model e un database relazionale. E’ possibile uitilizzare un DSL per descrivere i mapping, ed il modello semantico sarà costituito dai Data Mappers, non dal modello del dominio che invece è il soggetto del mapping.

3 Il linguaggio WDL

Il Wizard Definition Language è un DSL proprietario di i4C che permette la modellazione di processi utente unendo costrutti aritmetico/logici (sottolinguaggio "Formula Language") a direttive per la gestione della persistenza, della rappresentazione grafica e del workflow decisionale.

Con Wizard si intende un programma che guida l'utente nello svolgere una precisa azione, suddividendola in passi più semplici[7]. Questo concetto è stato sfruttato ottimamente nell'applicativo di i4C Analytics. Ha infatti permesso di rendere fruibili Applicazioni Analitiche Avanzate (AAA) ad utenti business e decision maker in maniera trasparente, senza richiedere alcun tipo di competenze statistiche o metodologiche (Figura 6).

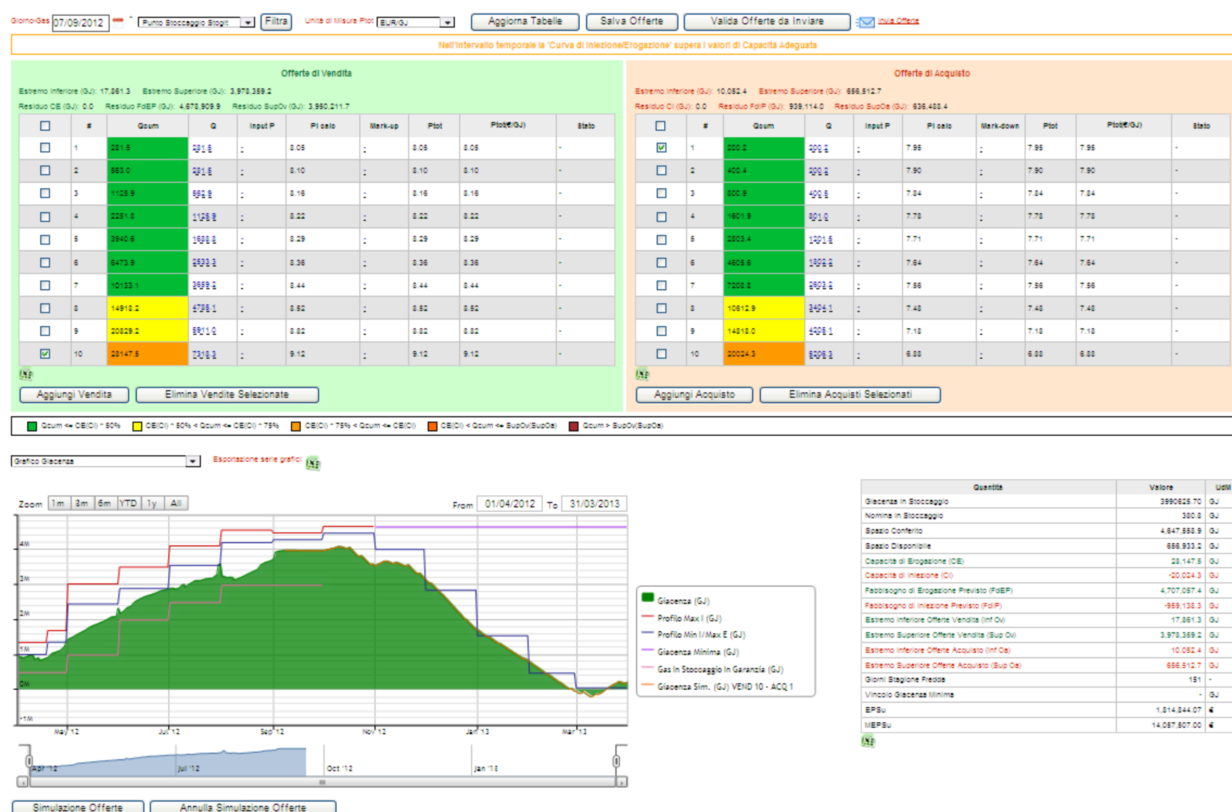


Figura 6: Esempio di un passo in un Wizard generato con WDL

L'approccio di i4C Analytics è "verticale" nel senso che vengono proposte ai propri clienti AAA specifiche per settore industriale e per processo di business. In questa situazione è fondamentale avere un software flessibile in grado di catturare i cosiddetti Business Objects (BO) del dominio specifico del cliente. All'interno dell'applicativo questa flessibilità è garantita dal concetto di Entity Type che rappresenta la classi di BO da cui poter creare delle istanze

concrete. Un programma in WDL, o Wizard, rappresenta quindi un'azione che un utente può intraprendere sull'istanza di un Entity Type ovvero su un BO (Figura 7).

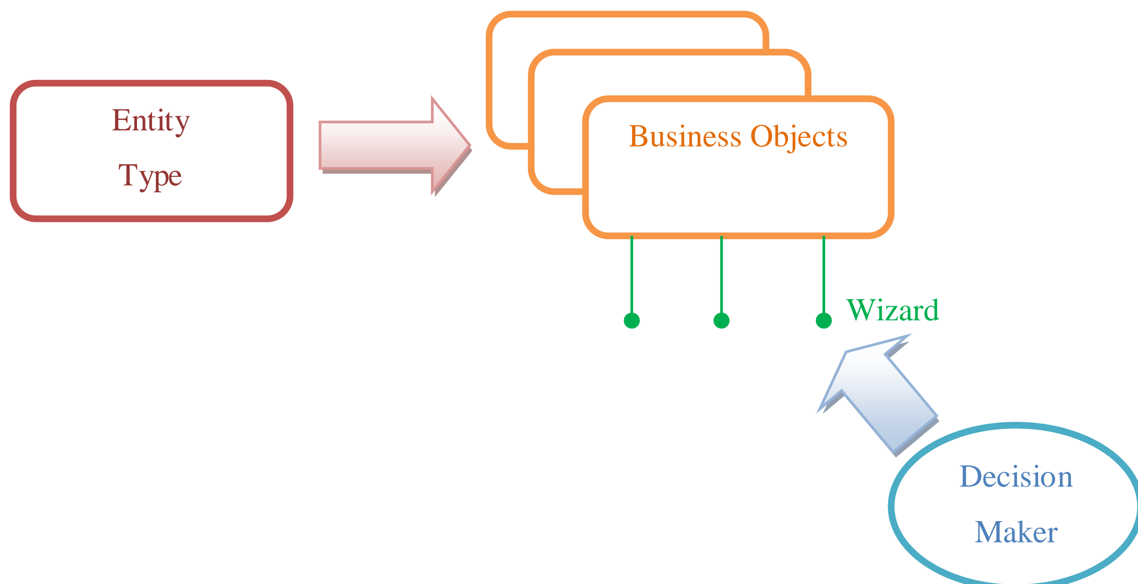


Figura 7: Relazione tra Entity Type, Business Objects e Wizard

L'obiettivo di WDL è quindi quello di definire Wizard su un applicativo. La natura del linguaggio è perciò strettamente legata a quella dell'applicativo che è caratterizzato da un'architettura distribuita. Il sistema, una volta in produzione, viene installato presso le infrastrutture del cliente, solitamente dei server, in modo da rendere fruibili i servizi a tutte le figure aziendali interessate. L'accesso ai servizi viene effettuato tramite browser Web, ed è proprio tramite questo strumento che avviene anche l'interazione con i Wizard. Da ciò si deduce come il linguaggio WDL debba essere in grado di definire un vero e proprio sottosistema distribuito in cui il client gestisce l'interazione con l'utente tramite pagine Web (View) mentre al server è affidato il compito di eseguire la parte di Model e Controller. Tutto ciò è trasparente al programmatore che si deve preoccupare soltanto di definire la Business Logic del Wizard senza che gli sia richiesta alcuna conoscenza di tecnologie e linguaggi legati al Web, come Html, JSP, Javascript, Ajax, JavaEE.

Gli obiettivi di WDL sono in realtà molteplici:

- definire le azioni per le istanze di Entity Type;

- creare un sottosistemi distribuiti in un'architettura client-server: generare le pagine Web per l'interazione con l'utente (lato Client), definire la parte controllo e individuare le parti di modello interessate (lato Server).
- rendere trasparente al programmatore l'uso di varie tecnologie e linguaggi del Web (Html, Ajax, Javascript, JavaEE, ecc).

3.1 Il Formula Language

Il linguaggio WDL nasce come estensione di un altro linguaggio, il Formula Language (FL), e per questo ne eredita tutte le caratteristiche. Questo linguaggio aveva, ed ha tutt'ora, un dominio ben preciso; nasce infatti con l'idea principale di poter gestire operazioni aritmetiche tra tipi di dato semplici e composti, e di poter accedere in lettura e scrittura alle diverse entità del sistema. Viene impiegato infatti per definire, all'interno di attributi particolari degli Entity Types, chiamati appunto attribuiti-formula, delle formule ricalcolate periodicamente. Per capire meglio che cosa effettivamente sia una formula vengono riportati di seguito gli elementi peculiari del FL.

3.1.1 Formula Language: tipi di dato, operatori e costrutti

All'interno del linguaggio FL sono presenti, come accennato, sia tipi di dato semplici che complessi. Si trovano infatti numeri interi, numeri reali, stringhe, valori booleani e date fra i tipi di dato semplici, mentre tra quelli composti si hanno a disposizione serie temporali, file, array e set. Gli operatori messi a disposizione del designer si possono classificare in quattro categorie:

- operatori di assegnamento: =;
- operatori aritmetici: + (addizione), - (sottrazione), * (moltiplicazione), / (divisione), ^ (elevamento a potenza);
- operatori di comparazione: < (minore), > (maggiore), == (eguaglianza), <= (minore uguale), >= (maggiore uguale);
- operatori logici: *AND*, *OR*, *NOT*

La creazione di una variabile viene effettuata usando la parola chiave *VAR*. Da ciò si deduce che il linguaggio non è tipato, è affidata quindi al programmatore l'incombenza di usare appropriatamente gli operatori con valori semanticamente validi, mentre la conversione dei tipi viene gestita automaticamente dall'interprete.

Oltre agli operatori appena descritti, sono a disposizione del programmatore anche dei costrutti per gestire il flusso di esecuzione. Si trovano infatti:

- costrutti condizionali: IF-THEN-ELSE
- costrutti per le iterazioni: FOREACH, FOREACHDATE

Le regole sintattiche per tali costrutti sono definite con la notazione EBNF in questo modo:

- IF-THEN-ELSE:

```
IF (<condition>)
THEN ( <expression>; | <block>)
[ ELSE ( <expression>; | <block>) ]
```

- FOREACH:

```
FOREACH (<array_or_set> AS <var>)
( <expression>; | <block> )
```

- FOREACHDATE:

```
FOREACHDATE (<startDate>, <endDate>, <ts> AS <var>)
( <expression>; | <block> )
```

All'interno delle formule c'è la possibilità, per i tipi di dato interi, reali e serie temporali di associare un'unità di misura (UdM), ai valori contenuti specificandole tra parantesi angolari. Ad esempio:

VAR qty1 = 500<g>;

memorizza nella variabile *qty1* la quantità di cinquecento grammi di peso. Le Udm possono essere semplici, come quella dell'esempio precedente, ma anche composte come nell'esempio seguente in cui si memorizza un il prezzo di cinquecento euro al chilo:

VAR price = 500<€/Kg>;

Tutte le unità di misura vengono definite, assieme ai fattori di conversione, in fase di configurazione dell'applicativo.

3.1.2 Dot Notation

La caratteristica più interessante del linguaggio FL è la parte denominata Dot Notation, introdotta allo scopo di accedere e navigare il diagramma degli Entity Types. Tutte le istanze di Entity Type nell'applicativo vengono, infatti, memorizzate su database relazionali; la Dot Notation permette di accedervi usando una notazione alternativa alle query, e più user-friendly.

Un’espressione in Dot Notation (Figura 8) è composta da un set di token separati dal simbolo “.” (Dot) di cui il primo token viene anche chiamato *root*. Un’espressione siffatta viene ritenuta valida se rispetta si seguenti vincoli:

- il token root deve essere il nome di tipi entità main, relazione, componente o aggregato;
- tutti i token intermendi sono nomi di Entity Type main, relazione, componente;
- ogni token intermedio deve essere un Entity Type direttamente connesso al precedente nello schema E-R
- l’ultimo token deve essere:
  - il nome di un Entity Type direttamete connesso con il penultimo Entity Type oppure
  - il nome di un attributo dell’Entity Type specificato nella penultima posizione

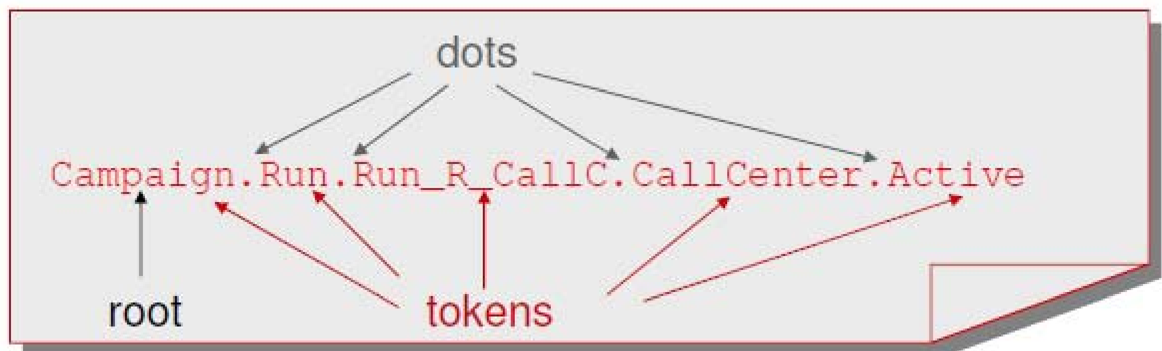


Figura 8: Esempio di espressione con la Dot Notation

La Dot Notation permette anche di filtrare le istanze specificate in un’espressione. Il filtro può essere applicato a qualsiasi Entity Type dell’espressione specificando un condizione tra parentesi quadre. Questa condizione può riguardare :

- attributi pre-configurati
- attributi nativi:
  - ID, NAME, DESCRIPTION;
  - COD_ENTITY_TYPE;
  - CREATION_USER, CREATION_DATE;
  - UPDATE_USER, UPDATE_DATE
- tutti gli Entity Types (e loro attributi) direttamente connessi nel modello E-R con quello filtrato

Nelle condizioni possono essere utilizzati anche:

- variabili, precedute da $;

- gli operatori relazionali per comparare scalari o date;
- gli operatori logici per combinare condizioni diverse;
- l'operatore di uguaglianza e l'operatore *LIKE* utilizzato in combinazione con *%* e _ per comparare le stringhe

Eccone un esempio:

```
Run[NAME LIKE "R%" OR Run_R_CallC.CallCenter.ID="CALL_INT1"]
```

Figura 9: Esempio di espressione in Dot Notation con filtro

3.2 Le astrazioni ed espressività del linguaggio WDL

Dato che, come già accennato, WDL è un linguaggio che estende il FL, tutto ciò che può essere fatto in una formula, può anche essere inserito in un Wizard. Per definizione un Wizard è:

"un insieme ordinato di passi presentati all'utente in sequenza allo scopo di collezionare dati, eseguire calcoli, presentare report, ecc "[8]

perciò, un programma in WDL non sarà altro che una lista di passi all'interno dei quali è possibile inserire formule e nuove funzioni built-in. A tutte le funzionalità appena viste, infatti, ne vanno aggiunte di nuove, in particolare:

- La possibilità, ovviamente, di definire i passi del Wizard utilizzando costrutti specifici;
- La capacità di renderizzare vari controlli grafici grazie a numerose funzioni built-in;
- la disponibilità di funzioni built-in in combinazione con la Dot Notation per la creazione, l'aggiornamento e la cancellazione delle istanze.

Il linguaggio mette a disposizione del programmatore due nuovi costrutti per la gestione del concetto di passo: STEP e FOREACHSTEP. Il primo è impiegato per la creazione di nuovi passi nel Wizard, mentre il secondo è utile quando si ha bisogno di ripetere un insieme passi per ciascun elemento di un set o array. Analizzare le sintassi dei due costrutti può essere utile a capire che cosa effettivamente è possibile specificare in un costrutto, in modo da comprendere il livello di espressività del linguaggio.

3.2.1 Il blocco STEP

```
STEP(<type>,<title>,[<custom_step_class>],[<custom_step_page>])
        {
                <block>
                [VALIDATION: <block>]
                [STYLE:<block>]
                [LAYOUT:<block>]
                [ON_NEXT: <block>]
                [ON_PREVIOUS: <block>]
                [ON_REFRESH: <block>]
        }
```

Come si può osservare il costrutto è costituito dalla parola chiave *STEP*, da una sezione, indicata tra parentesi tonde, in cui vengono specificati i parametri, e da un'altra sezione, questa volta tra parentesi graffe, che rappresenta il corpo dello step. I parametri da specificare sono quattro :

- *<type>* il tipo dello step. In WDL i tipi possibili sono due input step e custom step quindi i valori possibili per questo parametro sono rispettivamente "*input*" e "*custom*";
- *<title>* il titolo dello step che andrà mostrato all'interno della pagina Web;
- *<custom_step_class>* è il nome della classe che implementa lo step, che deve essere valorizzato specificando anche il package;
- *<custom_step_page>* è la pagina JSP da visualizzare per lo step

ma gli ultimi due sono validi solo quando lo step è di tipo custom.

All'interno del corpo dello step, oltre a formule e built-in, è lecito inserire, opzionalmente, dei blocchi di codice eseguiti per scopi od eventi particolari. Il blocco *VALIDATION*, ad esempio, serve per validare i dati raccolti all'interno dello step; solitamente il codice verifica che i dati soddisfino certi vincoli restituendo un messaggio di errore, mostrato all'utente, nel caso questo non accada. I blocchi STYLE e LAYOUT, aiutano il programmatore nel modificare rispettivamente l'aspetto e la disposizione dei componenti grafici. Nel primo è importante sottolineare come l'aspetto grafico sia modellato a livello di classi CSS. Si può, infatti, inserire del Foreign Code in CSS sotto forma di unica stringa a cui concatenare gli stili di defalut. Ne viene mostrato un esempio in Figura 10.

```
STYLE : {
    ".containerRow {overflow:auto;clear:both;}
    .alignedLabel {float:left; margin-top:5px;}
    .spaced {margin-left:25px;}
    .element {float:left;}
    .fullWidth {width:100%;}
    .left {float:left; width:44%;overflow:auto;padding-left:18px;padding-right:18px;padding-top:20px;}
    .commands {clear:both; padding-top:18px;}"
    + CoreLibrary.getTextboxDefaultStyleClass("dateTextboxClass",70)
    + CoreLibrary.getDefaultCss();
}
```

Css

Figura 10 : Esempio di Foreign Code CSS in WDL

I blocchi *ON_NEXT, ON_PREVIOUS* e *ON_REFRESH* vengono eseguiti a fronte di precisi eventi, rispettivamente alla pressione dei pulsanti "Successivo", "Precedente", per la navigazione tra i vari step del wizard, e *REFRESH_BUTTON,* particolari pulsanti definibili dal programmatore.

3.2.2 Il blocco FOREACHSTEP

```
FOREACHSTEP(<array_or_set> AS <var>){
        [INIT:<block>}]?
        [STEP]+
}
```

Il costrutto FOREACHSTEP consente di iterare su array o un set per mostrare uno o più passi molteplici volte, una per ogni volta che l'elemento si trova nell'array/set. Anche in questo caso si possono distinguere tre sezioni: una tra parentesi tonde, preceduta dalla keyword *FOREACHSTEP,* ed il un corpo del costrutto. Tra parentesi tonde vengono indicati il set o l'array contenenti gli step su cui iterare e la variabile in cui inserire lo step ad una certa iterazione, separati dalla parola chiave *AS*. Tra parentesi graffe si specifica il corpo del costrutto caratterizzato da una lista di dichiarazioni di passi e da un blocco opzionale *INIT* , il cui codice viene eseguito prima di ciascun passo.

3.3 Il parser generator JavaCC.

3.3.1 Che cos'è un parser generator?

Un meta-compilatore o parser generator è un sistema che, a partire dalla specifica di un linguaggio, costruisce un programma (analizzatore) capace di riconoscere sequenze (frasi) del linguaggio. In generale la specifica comprende sia l'aspetto lessicale sia quello sintattico, cosa che permette la costruzione automatica del parser, mentre l'aspetto semantico viene lasciato al lavoro (aggiuntivo) del progettista, anche se sono previsti costrutti che permettono di arricchire il parser con azioni legate all'aspetto semantico e scritte nello stesso linguaggio di programmazione del programma generato.

3.3.2 Caratteristiche di JavaCC

JavaCC ("Java Compiler Compiler") è uno strumento *open-source* per la costruzione automatica di analizzatori lessicali e sintattici in Java, chiamati anche parser generator. A partire da una specifica formale della grammatica del linguaggio, esso produce classi in puro codice Java atte a leggere ed analizzare frasi in tale linguaggio.

JavaCC risulta particolarmente utile allorché i linguaggi da analizzare presentino strutture tanto complesse da sconsigliare un approccio non automatizzato. Questa tecnologia è nata per rendere più semplice l'implementazione dei linguaggi di programmazione - da qui il termine "compilatore di compilatori" - ma trova applicazione anche per linguaggi più semplici e di tipo diverso (*scripting*, *markup*, ecc.). JavaCC non automatizza anche la costruzione esplicita degli *alberi sintattici* ma esistono altri due strumenti a corredo di JavaCC che possono essere utilizzati a tale scopo: JJTree e JTB. Risulta facile, inoltre, estendere le specifiche per automatizzare la costruzione delle *tabelle dei simboli* (Appendice I).

Sia JavaCC sia i parser generati con questo strumento girano su numerose piattaforme. Molti linguaggi hanno già una specifica per JavaCC come quella dello stesso Java, JavaScript, RTF, Visual Basic, PHP, Python, C, C++, ma anche XML, HTML, SQL, e molti altri ancora.

JavaCC possiede diverse caratteristiche rilevanti come ad esempio *l'analisi top-down*: JavaCC genera, infatti, Recursive Descent Parser (Appendice D) differenti dai parser ascendenti generati, ad esempio, da strumenti come YACC e Bison. Ciò consente di trattare un gran numero di grammatiche libere di forma diversa, sebbene la ricorsione sinistra sia vietata. Inoltre i parser discendenti sono più semplici per quel che riguarda il loro debugging e si prestano all'utilizzo a partire da qualunque simbolo non-terminale, consentendo anche il passaggio di valori (attributi) su e giù per l'albero di derivazione durante il processo d'analisi.

Le specifiche lessicali in forma di espressioni regolari, stringhe, ecc. e quelle sintattiche (in EBNF) sono scritte insieme in uno stesso file. Ciò rende la grammatica di facile lettura (si possono usare espressioni regolari - *inline* - nella specifica della grammatica) ed anche più facile di manutenere. Per default, JavaCC genera un parser LL(1). Tuttavia, ci potrebbero essere molte parti non LL(1) nella grammatica. JavaCC offre le capacità di lookahead sintattico e semantico per risolvere le ambiguità *shift-shift*[1] in modo locale nei punti esatti, ad esempio quando il parser è LL(k) solo in certi punti ed LL(1) altrove. I casi di ambiguità *shift-reduce*[2] e reduce-reduce[3], tipici dei parser ascendenti, non costituiscono un problema per parser top-down.

JavaCC consente le specifiche in EBNF - come `(A)*`, `(A)+`, ecc. - nelle regole lessicali e sintattiche. L'uso dell'EBNF solleva, in parte, dalla necessità di usare la ricorsione sinistra. Di fatto, risulta più leggibile rappresentare una regola come `A ::= y(x)*` piuttosto che con `A ::= Ax | y`.

Da notare la possibilità di usare stati ed azioni lessicali come Lex. Altri aspetti specifici di JavaCC che lo rendono superiore ad altri strumenti sono lo status speciale attribuito a costrutti come `TOKEN`, `MORE`, `SKIP`, cambiamenti di stato, ecc. Questo favorisce specifiche più pulite e messaggi d'errore e di avvertimento più precisi con JavaCC. Le specifiche lessicali possono poi definire i token senza tener conto della distinzione minuscole/maiuscole a livello globale o anche locale grazie al fatto che l'analisi lessicale è case-insensitive. I token definiti come speciali nella specifica lessicale sono ignorati in fase di analisi sintattica, ma possono essere elaborati da altri strumenti. Un'applicazione utile potrebbe essere quella relativa all'elaborazione dei commenti ai fini della documentazione.

Attraverso opzioni quali `DEBUG_PARSER`, `DEBUG_LOOKAHEAD`, e `DEBUG_TOKEN_MANAGER`, si può ottenere una analisi dettagliata del processo di parsing e/o di elaborazione dei token.

La gestione degli errori con JavaCC è tra le migliori tra i vari generatori automatici di parser. JavaCC genera parser capaci di indicare il punto di errore e fornire informazioni diagnostiche complete.

Questo parser generator è personalizzabile, offre tante opzioni per adattare il suo comportamento e quello dei parser generati, come ad es. i tipi di caratteri Unicode per il flusso di caratteri in input, il numero di token per il controllo di ambiguità, ecc.

L'internazionalizzazione permette inoltre che uno scanner generato da JavaCC possa manipolare input Unicode e le sue specifiche lessicali possano includere qualunque carattere

Unicode. Ciò facilita la descrizione degli elementi di un linguaggio, come Java, che ammette caratteri Unicode (in numero maggiore rispetto a quelli ASCII).

JavaCC gira su tutte le piattaforme *Java-compliant*. Viene utilizzato su tanti tipi di macchine con nessuno sforzo relativo al porting - a testimoniare la filosofia "*Write Once, Run Everywhere*" di Java; inoltre include lo strumento JJDoc che converte file-grammatica in file di documentazione (anche in HTML).

La release di JavaCC contiene molti esempi di specifiche di grammatiche. Gli esempi, assieme alla loro documentazione, costituiscono un valido ausilio all'apprendimento dell'utilizzo di questo strumento.

Una delle carte vincenti di JavaCC è l'ampia comunità di utenti,infatti, è probabilmente il generatore automatico di parser più usato, in congiunzione con applicazioni Java, con centinaia di migliaia di download ed una platea stimata di circa qualche decina di migliaia di utenti professionali, come testimoniato anche dal numero dei partecipanti alle varie mailing-list e newsgroup [9].

3.3.3 JavaCC in WDL

Il parser del linguaggio WDL è stato realizzato con il tool JavaCC, che oltre a generare il parser produce anche le classi da utilizzare per implementare l'interprete. Come già accennato, JavaCC basa il suo funzionamento sulla specifica BNF del linguaggio per cui deve creare il parser. Per WDL questa specifica è piuttosto lunga e complessa perciò ne verrano riportati soltanto le parti più salienti.

```
Wizard Wizard() #void :
{
  Wizard wizard = new Wizard();
  Step step;
}
{
  ( step = SimpleStep() { wizard.addStep(step); }
  | step = ForEachStep()  { wizard.addStep(step); }
  )+
  {
    return wizard;
  }
}

ForEachStep ForEachStep() #void :
{
  ForEachStep step = new ForEachStep();
  Wizard child;
}
{
  < FOREACHSTEP >
  < LP >
    PrimaryExpression() { step.setForEachArgumentNode(jjtree.popNode()); } < AS > <
```

```
IDENTIFIER > { step.setLocalVarName(token.image); }
  < RP >
  < LB >
    [ < INIT > < COLON >
      < LB >
        InitBody() { step.setInitBody((ASTInitBody) jjtree.popNode()); }
      < RB >
    ]
    child = Wizard() { step.setChild(child); }
  < RB >
  {
    return step;
  }
}

SimpleStep SimpleStep() #void :
{
  SimpleStep step;
}
{
  ( LOOKAHEAD(8)
    step = InputStep()
  | step = CustomStep()
  )
  {
    return step;
  }
}

InputStep InputStep() #void :
{
  InputStep step = new InputStep();
  String literal;
  java.util.List<AbstractDiv> divs;
}
{
  < STEP >
  < LP >
    literal = DotNotationStringLiteral() { step.setType(literal); }
    < COMMA >
    literal = DotNotationStringLiteral() { step.setName(literal); }
    (
            < COMMA > Expression() { step.setEnabledExpression(jjtree.popNode()); }
    )?
  < RP >
  [ <DISABLED> [ < PREVIOUS > { step.setPreviousEnabled(false); } ] [ < CANCEL > {
step.setCancelEnabled(false); } ] ]
  < LB >
    StepDefinition() { step.setStepDefinitionNode(jjtree.popNode()); }
    [ < ON_REFRESH > < COLON > OnRefreshBody()
        {
            BaseNode node = (BaseNode) jjtree.popNode();
            step.setOnRefreshBodyNode(node);
        }
    ]
    [ < VALIDATION > < COLON > Validation() { step.setValidationNode(jjtree.popNode()); }
]
    [ < ON_NEXT >
      ( < COLON > OnNextPreviousBody() | < ASYNCHRONOUS > < COLON > OnAsyncNextBody() )
        {
            BaseNode node = (BaseNode) jjtree.popNode();
            step.setOnNextBodyNode(node);
        }
```

```
    ]
    [ < ON_PREVIOUS > < COLON > OnNextPreviousBody() {
step.setOnPreviousBodyNode(jjtree.popNode()); } ]
    [ < STYLE > < COLON > < LB > Style() < RB >
      {
            BaseNode node = (BaseNode) jjtree.popNode();
            step.setStyleNode(node);
      }
    ]
    [ < LAYOUT > < COLON > Layout() { step.setLayoutNode(jjtree.popNode()); } ]
   < RB >
   {
    return step;
   }
}
```

Figura 11: Grammatica BNF del linguaggio WDL in JavaCC

Con la parte di grammatica mostrata in Figura 11, si intende riconoscere la struttura di un wizard in termini di blocchi *STEP*, di tipo "input" e di tipo "custom", e di blocchi *FOREACHSTEP*. E' riportata anche la struttura dello step "input" in gran parte simile a quello "custom", infatti differiscono sintatticamente soltanto per il numero di parametri specificabili.

Confrontando questa grammatica con quella vista nei paragrafi 3.2.1 e 3.2.2, si può notare come la specifica in JavaCC sia sintatticamente più complessa, rendendola decisamente meno leggibile. Questo problema è legato al fatto che, nelle BNF di un Parser Generator, è necessario specificare anche delle azioni per costruire l'Abstract Syntax Tree (AST) ma mano che viene creato quello sintattico. L'AST, a differenza di quello sintattico, contiene soltanto gli elementi salienti per l'interpretazione eliminando i dettagli sintattici, come i punto e virgola, le parentesi e le parole chiave, che hanno il solo scopo di rendere non ambiguo il linguaggio. Per approfondire la costruzione degli AST si rimanda alla lettura dell'Appendice J.

Spesso però questi dettagli sintattici non sono sufficienti a risolvere tutte le ambiguità del linguaggio, ecco perchè in JavaCC sono presenti i cosiddetti predicati semantici o lookahead semantici, i quali, in aggiunta alle azioni, rappresentano un'altro fattore che influisce sulla scarsa leggibilità delle BNF in JavaCC. Questi predicati sono però fondamentali, ed infatti vi si ricorre molto spesso soprattutto in linguaggi estesi come WDL. In Figura 12 viene riportatto un punto della grammatica particolarmente ambiguo in cui si fa uso dei predicati semantici tramite la funzione *LOOKAHEAD*.

Questa funzione permette di analizzare intere sequenze di token per capire quale regola di produzione utilizzare in punti di ambiguità. In condizioni normali il parser controlla il successivo token e, sulla base del suo tipo, sceglie la regola di produzione appropriata, è come se utilizzasse la funzione *LOOKAHEAD* su un solo token. Come già accennato, in punti di

particolare ambiguità, non è sufficiente analizzare un solo token per scegliere la giusta regola di produzione. Nell'esempio riportato in Figura 12 si può notare come le sintassi per accedere ad array o mappe, per invocare una formula e per invocare una funzione inizino tutte con un token *<IDENTIFIER>*. Il parser, quindi, non ha sufficienti informazioni per procedere. L'uso dei lookahead semantici permette al parser guardare più avanti nella successione dei token recuperando le informazioni necessarie a risolvere l'ambiguità.

```
void PrimaryExpression() #void :
{}
{
  ( Literal()
  | < LP > Expression() < RP >
  | LOOKAHEAD(< IDENTIFIER > < LSB > Expression() < RSB >)
    ( < IDENTIFIER > { jjtThis.setName(token.image); } < LSB > Expression() <
RSB > ) #ArrayOrMapAccessor
  | LOOKAHEAD(< IDENTIFIER > < DOT > < IDENTIFIER > Arguments())
    ( < IDENTIFIER > { jjtThis.setName(token.image); } < DOT > < IDENTIFIER > {
jjtThis.setComponent(token.image); } Arguments() ) #FormulaInvocation
  | LOOKAHEAD(< IDENTIFIER >  Arguments())
    ( < IDENTIFIER > { jjtThis.setName(token.image);
jjtThis.setPosition(token.beginLine, token.beginColumn); } Arguments() )
#FunctionInvocation
  | Identifier()
  )
}
```

Figura 12: Uso dei predicati semantici in WDL

3.4 Un interprete per WDL

Una volta generato il Parser, il lavoro per definire un linguaggio non ancora concluso. Con il Parser si ottiene uno strumento in grado di distinguere le frasi corrette di un linguaggio da quelle non corrette, ma un volta che abbiamo verificato la correttezza delle frasi ci si aspetta che queste frasi inneschino delle azioni, cioè che abbiano una loro semantica. A tale scopo viene è necessario definire l'interprete del linguaggio, il cui unico obiettivo è quello di attribuire una semantica alle frasi riconosciute dal parser.

In WDL l'interprete è stato definito combinando i design pattern Visitor e Double Dispatch di cui JJTree fornisce il supporto settando l'opzione *VISITOR* a *TRUE*. Con tale opzione viene generata automaticamente, oltre alle classi dei nodi dell'AST, anche l'interfaccia DotNotationParseVisitor, che il Visior stesso dovrà implementare per navigare l'AST, e viene aggiunto a ciascun nodo un metodo *jjAccept()* per implementare il pattern Double Dispatch.

Nel grafico in Figura 13 vengono mostrate le classi che intervengono nella fase di interpretazione sia del linguaggio WDL che del FL. In reltà sono presenti anche le interazioni tra le diverse classi ordinate cronologicamente, perciò si può comprendere meglio cosa avviene all'esecuzione di un Wizard o di una Formula. Per semplicità verrà considerata soltanto la parte collegata dalle frecce verdi, cioè quella riguardante WDL, dato che la restante funziona in maniera del tutto simile.

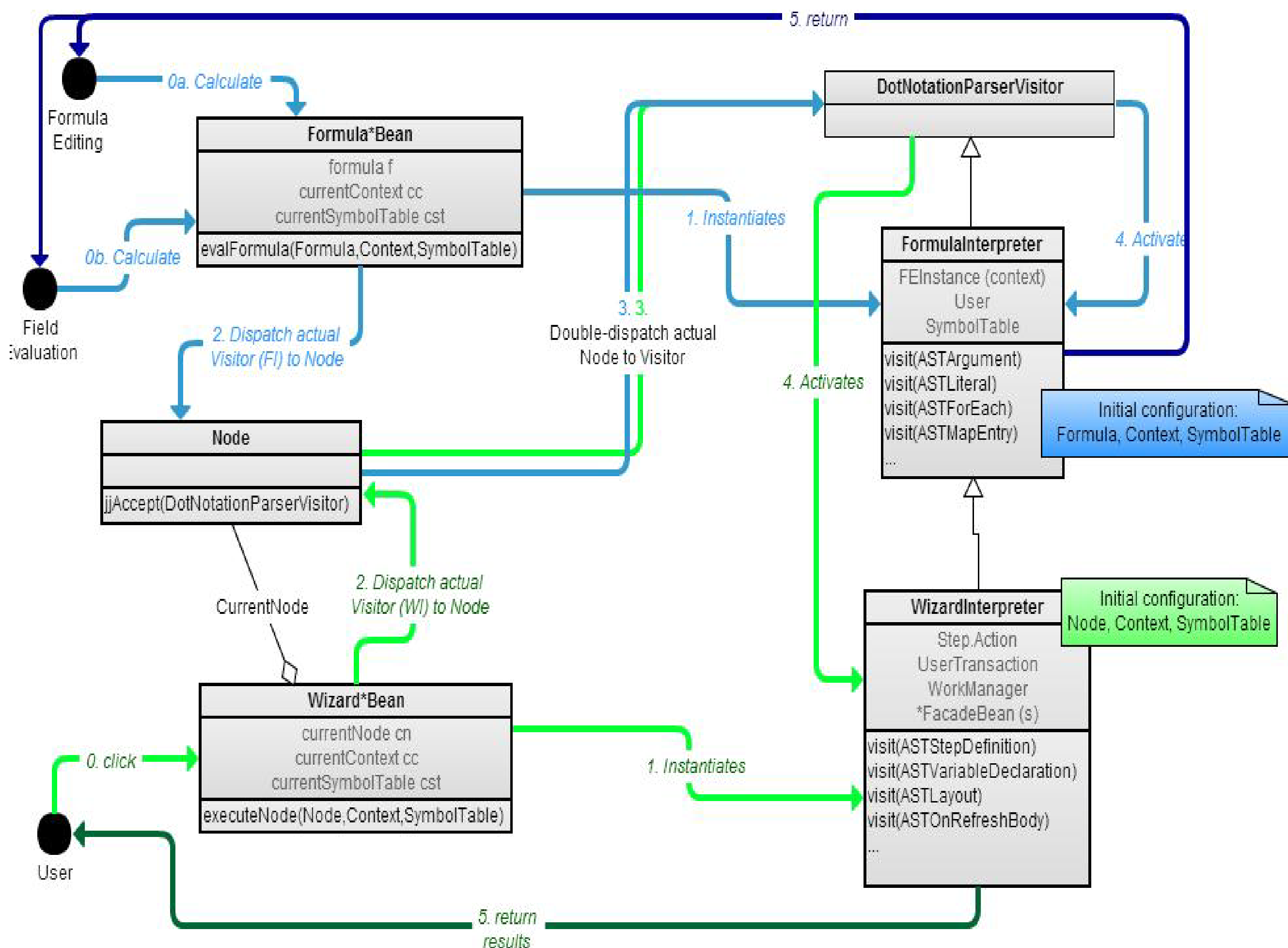


Figura 13: Fase di interpretazione in WDL

Ecco come avviene la fase di interpretazione un Wizard :

0. L'utente avvia un Wizard per eseguire un'azione su un determinato BO, che in questa fase assume anche il significato di contesto d'esecuzione. Una volta che il Wizard ha superato la fase di parsing si procede con l'interpretazione vera e propria.
1. La richiesta di esecuzione viene catturato dall'oggetto *Wizard Bean* che è incaricato di istanziare l'interprete rappresentato dalla classe *WizardInterpeter.* Gli oggetti istanziati da questa classe sono dei Visitors e per questo devono rispettare le

specifiche definite dall'interfaccia *DotNotationParserVisitor*. In questo modo si è certi che il Visitor possa visitare i nodi dell'AST generato dal parser.

2. Il *Wizard Bean* invia il Visitor al nodo radice dell'AST, passandolo come parametro al metodo *jjAccept(DotNotationParserVisitor)*.
3. A questo punto interviene il pattern Double Dispatch. Il nodo, infatti, viene inviato al visitor che in base al tipo specifico effettuerà un differente interpretazione.
4. L'interpretazione avviene a seguito della chiamata del metodo *visit()* che, sfruttando il polimorfismo parametrico, si comporta differentemente in base ai parametri che gli vengono passati, in questo caso in base al tipo di nodo.
5. Il risultato dell'interpretazione verrà poi mostrato all'utente. In WDL ciò significa, ad esempio, mostrare la pagina Web con cui l'utente può interagire ad un certo passo di un Wizard.

3.5 WDL è un DSL?

Allo scopo di capire se il linguaggio WDL sia o meno un Domain Specific Language verrà ora contestualizzata la definizione mostrata nel Paragrafo 2 che indicava un DSL come *"un linguaggio di programmazione di espressività limitata focalizzato su un particolare dominio"*.

Sul fatto che WDL sia un linguaggio di programmazione non vi è dubbio , è stato infatti progettato ed è utilizzato per impartire comandi alle macchine, più precisamente a dei server, come si vedrà più avanti discorrendo dell'architettura dell'applicativo.

Il dominio applicativo di WDL è piuttosto definito: FL si occupa dei calcoli matematici garantendo il supporto delle unità di misura; la DotNotation gestisce l'accesso ai BO tramite la navigazione dei diagrammi Entity-Relationship; i concetti di Step e di Wizard introdotti estendendo FL sono abbastanza specifici. In sostanza in WDL coesistono tre domini ben definiti che presentano alcuni concetti particolari, non identificabili in nessun altro linguaggio.

Ciò che potrebbe far sorgere dei dubbi è il discorso della *limitata espressività*. Infatti, osservando i costrutti presentati nel Paragrafo 3.1, si può notare come molti di essi siano tipici dei linguaggi general-purpose, in cui l'espressività raggiunge il suo massimo. I costrutti in questione sono quelli utilizzati per il controllo del flusso di istruzioni, come l'*IF-THEN-ELSE* oppure il *FOR-EACH*. Tali costrutti aumentano enormemente la capacità espressiva di WDL facendo perdere, in parte, alcuni vantaggi peculiari dei DSL, come la leggibilità del codice volta all'individuazione degli errori e alla fruibilità verso figure aziendali diverse dai programmatori.

Chi produce software spesso utilizza un DSL per configurare le proprie applicazioni, ad esempio sfruttando CSS o XML. WDL non è però un mero linguaggio di configurazione e, come già spiegato in precedenza, ha l'obiettivo di definire, non solo le componenti di view dei Wizard, ma anche il loro comportamento. A questo scopo, i costrutti citati sono fondamentali.

A fronte di tali considerazioni non si può certamente dire che WDL è un linguaggio general-purpose: mancano infatti i concetti di oggetto, di erditarietà, di polimorfismo paramentrico, manca la possibilità di definire funzioni ed il concetto di tipo per le variabili. Tutte features presenti nei principali linguaggi general purpose quali Java o C++.

Concludendo WDL è un linguaggio che ha molto in comune con un DSL, nonostante ne violi in parte i concetti per garantirsi la minima espressività necessaria all'uso per cui è stato destinato. Gode, perciò, in misura minore dei vantaggi principali di un DSL, in particolare per quanto riguarda la migliore leggibilità del codice.

4 Analisi del problema

Il problema affrontato in questa Tesi è quello di rendere disponibile un Integrated Development Environment agli sviluppatori della Software Vendor i4C Analytics, che fornisca gli strumenti adatti per migliorare il processo di sviluppo del software nel proprio applicativo. L'IDE verrà utilizzato esclusivamente per implementare del codice per la logica di Business, che, nella visione di i4C, si traduce nello sviluppo di Wizard per guidare l'interazione dei Decision Maker con gli oggetti di Business.

I Wizard, come spiegato nel Capitolo 3, vengono implementati grazie al linguaggio custom WDL che fornisce tutte le astrazioni necessarie a tale scopo, perciò l'IDE dovrà integrare degli strumenti ritagliati su misura del suddetto linguaggio.

Come descritto nel paragrafo 1.2, tra i componenti di un IDE, sono sempre presenti un editor e un compilatore/interprete. Nel problema specifico il linguaggio WDL è interpretato ed i componenti per effettuare l'analisi sintattica, l'analisi lessicale e l'interpretazione sono inseriti all'interno dell'applicativo, il problema è, perciò, strettamente legato all'architettura di quest'ultimo.

4.1 Architettura dell'applicativo

L'applicativo di i4C Analytics è stato progettato sulla tipica architettura J2EE riportata in Figura 14: Architettura J2EE per l'applicativo . La logica di business in questi sistemi è incapsulata all'interno dei componenti di Enterprise Java Beans (EJB) .

Enterprise JavaBeans è una piattaforma per la creazione di applicazioni business portabili, riusabili e scalabili mediante il linguaggio java. Un'applicazione è costituita da componenti che vivono all'interno di un EJB container che fornisce a tali componenti un certo numero di servizi quali la gestione della sicurezza, delle transazioni, i supporto per i web-services, ecc.

I componenti EJB sono simili a un qualsiasi altro POJO (Plain Old Java Object) e sono suddivisi in tre tipi (Figura 15): session bean, message-driven bean e entity bean. I primi due sono utilizzati per implementare la logica di business mentre gli entity bean sono utilizzati per la persistenza. In particolare, un session bean viene invocato da un client allo scopo di effettuare una specifica elaborazione: il termine "session" fa riferimento al fatto che l'istanza di un bean di questo tipo non sopravvive a crash o shutdown del server. Esistono due tipi di session bean:

- **stateful:** sono quei bean il cui stato viene salvato fra una richiesta e l'altra del client

- **stateless**: sono quei bean che non mantengono alcun stato e vengono utilizzati per modellare quelle elaborazioni che si completano con una sola richiesta

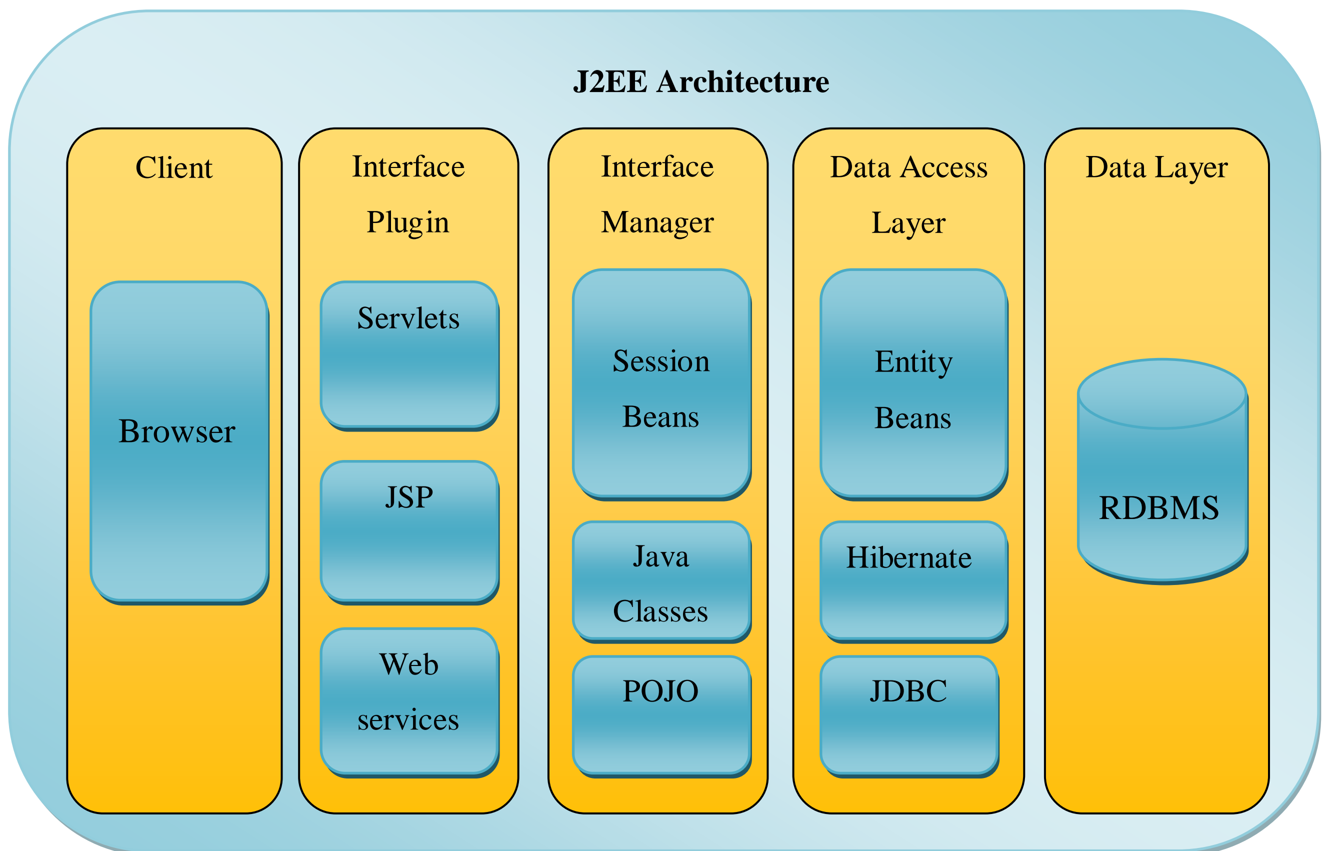


Figura 14: Architettura J2EE per l'applicativo

I session bean possono essere invocati localmente o da remoto mediante Java RMI.

I Message-driven Bean si differenziano dai session bean per il fatto che non vengono mai invocati direttamente dai client ma, piuttosto, sono attivati da messaggi inviati a un messaging server.

Infine gli entity bean sono oggetti java che costituiscono la rappresentazione dei dati dell'applicazione e che vengono memorizzati in maniera persistente in un database. In EJB 3 al persistenza è gestita da Java Persistence API mediante object-relational mapping (ORM): questo termine essenzialmente denota il processo di mapping dei dati fra gli oggetti e le tabelle di un database.

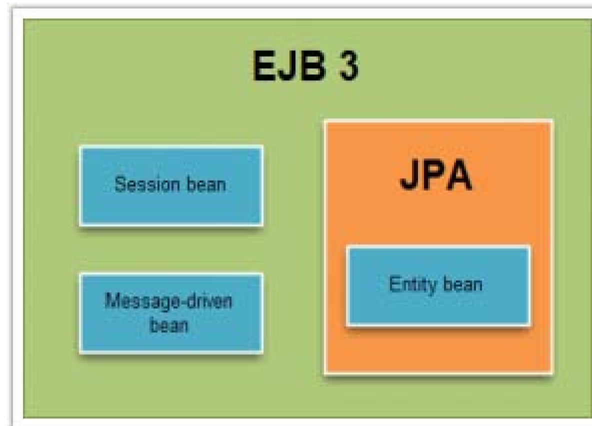


Figura 15: Piattaforma EJB 3

Mentre l'esecuzione di un session bean o di un message-driven bean richiede un EJB container, per assicurare la persistenza degli entity bean è necessario ricorrere ad un persistence provider, come Hibernate. Un persistence provider è essenzialmente un framework ORM che supporta le Java Persistence API [10].

Dal momento che un EJB container offre ai componenti EJB dei servizi, deve essere possibile configurare l'accesso a tali servizi, ciò può essere fatto in due modi:

- mediante annotazioni
- mediante file di configurazione XML (Deployment Descriptor)

Alcuni dei principali servizi offerti da un EJB 3 container sono:

- messaging: consente lo scambio di messaggi fra componenti nell'ottica di una comunicazione asincrona senza conoscere i dettagli implementativi di Java Messaging API
- transazioni: consente di trasformare facilmente i metodi di un componente in metodi transazionali i cui cambiamenti hanno effetto solo se si concludono positivamente (altrimenti si effettua il roll back).
- sicurezza: mediante Java Authentication and Authorization Service (JAAS) è possibile proteggere le risorse dell'applicazione senza modificare il codice ma con semplici opzioni di configurazione.
- accesso remoto: è possibile accedere ad un EJB da remoto senza scrivere alcun codice.
- web services: è possibile esporre i metodi di un EJB all'esterno come web services.

- persistenza: mediante Java Persistence API è possibile rendere persistenti le entità sincronizzandone i cambiamenti con un database mediante object relational mapping (ORM).

Nello strato Interface Manager, assieme agli Entity Bean, l'architettura J2EE offre la possibilità di inserire delle classi Java. Questa chance è stata colta dai progettisti di i4C per inserire nell'applicativo anche i servizi legati al linguaggio WDL. In pratica vi sono state allocate le classi necessarie alla fase di parsing, cioè quelle generate dal tool JavaCC, e alla fase di interpretazione.

In Figura 14 si mostra che gli utenti possono accedere a tutti i servizi dell'applicativo solo ed esclusivamente tramite browser conoscendo l'indirizzo URL a cui questi servizi sono stati resi disponibili. L'utente quindi non conosce dove effettivamente si trovi l'installazione dell'applicativo, potrebbe essere in locale, potrebbe trovarsi su un server remoto o, come accade solitamente, sui server aziendali connessi alla rete locale.

Il browser è in grado di visulizzare pagine Web di qualsiasi tipo, ed è infatti sfuttando delle pagine JSP che si svolge l'interazione con l'utente, la quale è gestita da uno strato di presentazione, rappresentato in Figura 14 dall'Interface Plugin. Questo livello, infatti, si occupa di reperire e di mostrare le pagine richieste, oltre a gestire le informazioni di cui i componenti della pagina hanno bisogno, accedendo se necessario ai Session Bean.

Questa architettura distribuita è state sfruttata da i4C Analytics per rendere fruibili ai Decision Maker delle Advanced Analytics Application (AAA) in modo da permettergli di gestire al meglio il proprio Business. Le AAA, a seconda delle esigenze, possono impiegare le capacità di quattro componenti logici:

- il Profiling Engine: adotta tecniche per raggruppare le istanze in base agli attribuiti. A ciascun gruppo viene applicato un algoritmo di profiling per trasformare le serie temporali da una granularità all'altra.
- il Forecaster Engine: esegue delle previsioni massimizzando l'accuratezza e minimizzando il know-how statistico richiesto per effettuarle.
- il Data Quality Engine: serve a mantenere alta la qualità dei dati sul Database in modo che errori dovuti ai dati non influenzino le analisi. A tale scopo presenta strumenti configurabili per la pulizia e la correzione automatica dei dati.
- il Reporting Engine: fornisce gli strumenti di base per la creazione di report operazionali. E' focalizzato sulla definizione della struttura grafica del report e su come i dati caricati vanno visualizzati sul report.

4.2 Requisiti

Ora che si è compresa la natura distribuita dell'applicativo e le principali entità che intervengono nel sistema, è immediato individuare i principali requisiti di cui una soluzione deve tenere conto.

Considerando che il problema nasce da un'esigenza interna della Software Vendor e non da un confronto con i clienti, non vi è necessità di adottare particolari metodologie di sviluppo che tengano conto della variazione dei requisiti in corso d'opera.

Analizzando il problema emergono due categorie di requisiti: quelli derivanti dai vincoli architetturali, o requisiti di sistema, e quelli strettamente legati all'IDE, le features. In entrambi i casi i requisiti erano ben definiti; i requisiti di sistema sono infatti legati all'architettura di un applicativo di cui la Software Vendor conosceva ogni singola linea di codice mentre, per quanto riguarda le features, il motivo è che riguardano caratteristiche dell'IDE con cui qualsiasi Ingegnere informatico o programmatore ha avuto a che fare nella propria vita lavorativa e di cui conosce chiaramente il significato.

4.2.1 Requisiti dell'IDE

Allo scopo di individuarne le necessità, è stata realizzata un'intervista agli utenti dell'IDE, cioè agli sviluppatori dei Wizard. L'intervista è stata condotta con un approccio volto a catturare non solo i problemi, ma anche possibili soluzioni in grado di soddisfare i bisogni degli utenti, i quali hanno partecipato in gruppo alla discussione, facendo emergere anche informazioni quali la priorità di sviluppo delle features e il maggior apprezzamento di alcune soluzioni rispetto ad altre. Dall'intervista sono emerse, com'era d'aspettarselo, requisiti riguardanti funzionalità tipiche dei moderni IDE ma anche molti requisiti nuovi, legati alle caratteristiche del linguaggio WDL o del framework. Di seguito verranno elencate le funzionalità di rilievo estrapolate dall'intervista.

4.2.1.1 Auto-completamento del codice

La capacità di un editor di suggerire all'utente parole chiave del linguaggio WDL, variabili, nomi delle funzioni built-in e l'insieme dei paramentri che costituisce la loro signature. Durante la digitazione, l'utente attiva, tramite una combinazione di tasti (es: Ctrl-Spazio), la funzione di autocompletamento, una tendina si apre al di sotto del cursore per mostrare i suggerimenti, i quali devono tenere conto anche del contesto in cui l'utente sta digitando. Se, ad esempio, l'utente sta implementando un determinato passo del Wizard, non

devono essere suggerite le variabili definite in passi precedenti perchè non valide in tale contesto. Tecniche avanzate attivano automaticamente il completamento quando incontrano una sequenza di caratteri presente anche nelle possibili parole da suggerire.

4.2.1.2 Syntax highilighting

La capacità di editor di distinguere gli elementi di un linguaggio colorandoli in maniera differente viene definita syntax highlighting, o syntax colouring. Tale funzionalità è immancabile in un IDE, dato che facilita la stesura e l'analisi del codice. Gli elementi del linguaggio che è interessante riconoscere in WDL sono le parole chiave, i commenti, le stringhe e le funzioni built-in. La presenza in WDL di foreign code CSS, richiede, inoltre, che il syntax highlighting sia in grado di riconoscere, e quindi colorare differentemente, le frasi appartenenti ad uno o all'altro linguaggio.

4.2.1.3 Controllo semantico

Il controllo semantico è uno strumento per il riconoscimento di errori semantici nel codice. Tali errori sono legati ad uso errato delle frasi del linguaggio. Sono infatti frasi che non assumono un significato valido pur essendo sintatticamente lecite. In WDL errori di questo tipo si presentano quando, ad esempio, non vi è una congruenza tra la dichiarazioni delle variabili ed il loro impiego nelle istruzioni, oppure quando non viene effettuato un uso corretto delle strutture di controllo; se, ad esempio, si inserisce un'espressione nella condizione di un *IF* che non viene valutata con un valore booleano:

```
IF (5 +3) THEN
        SHOWMESSAGE(“HelloWorld”)
```

4.2.1.4 Controllo sintattico

Il controllo sintattico, a differenza di quello semantico, mira a segnalare errori quando individua frasi che non appartengono al linguaggio, cioè che non ne seguono la sintassi. Questo strumento deve presentare alcune caratteristiche fondamentali allo scopo di risultare efficace nel supporto al programmatore; è infatti importante che sia in grado di effettuare un controllo completo del codice, evitando di interrompersi al primo errore individuato e che l'indicazione dell'errore sia chiara e precisa, cioè che sia segnalata nella riga e nella colonna più vicina possibile a quelle dove è effettivamente presente l'errore.

4.2.1.5 Debugger

Il debugger è uno strumento che assiste il programmatore nell'individuazione e correzione di errori, anche detti bug, in un programma[11]. Solitamente un debugger è in grado di interrompere l'esecuzione di un programma in punti precisi, detti breakpoints, specificati dal programmatore, per poi mostrare il valore di variabili o lo stato degli oggetti. A questo punto il programmatore può continuare l'esecuzione del codice manualmente, cioè eseguendo una ad una le istruzioni per comprendere meglio come lo stato di oggetti e variabili si evolve in seguito.

Dalla discussione con gli utenti dell'IDE è emerso che, in realtà, era sufficiente che questo strumento desse la possibilità di inserire uno o più breakpoint, e che, ovviamente, mostrasse lo stato delle singole variabili una volta che l'esecuzione si fosse interrotta al raggiungimento di uno di essi. Nessuna funzionalità di avanzamento manuale dell'esecuzione è perciò richiesto. E' stata, invece, specificata la possibilità di utilizzare i cosidetti breakpoint condizionali, caratterizzati da una breakpoint condition, cioè un'espressione che il debugger valuta una volta che il breakpoint viene raggiunto, ed in base al valore interrompe o meno l'esecuzione.

4.2.1.6 Auto-formattazione

Lo strumento di auto-formattazione aiuta il programmatore nel mantere uno stile di programmazione che migiliori la leggibilità del codice sorgente e quindi eviti di introdurre errori. Uno strumento del genere inserisce automaticamente elementi sintattici come le parentesi tonde, quadre, graffe, posizionandoli al giusto livello di indentazione.

L'auto-formattazione si deve attivare anche quando il programmatore inserisce un carattere "a capo". In tal caso lo strumento dovrà introdurre un numero corretto di caratteri "tab" per spostare il cursore facendo in modo da indentare opportunamente le istruzioni editate successive.

4.2.1.7 Folding

Il folding è una funzionalità dell'editor che da la possibilità di nascondere temporaneamente parti del codice sorgente, in modo da migliorarne la navigazione. Tale strumento è particolarmente utile quando il codice è costituito da numerose istruzioni. Solitamente è possibile attivare il folding tramite una combinazione di tasti, oppure tramite un'apposita icona presente a fianco della dichiarazione di un blocco di codice, che permette di comprimerlo (operazione di folding) o di mostrarlo (operazione di unfolding).

4.2.1.8 Strumento per la generazione visuale del layout

All'interno di un programma WDL è possibile specificare, per ciascun passo del wizard, una disposizione (layout) dei componenti grafici presenti. Quando, in certo passo, il numero di componenti è elevato oppure si vogliono realizzare disposizioni più complesse, la specifica testuale del layout può diventare complicata. Questo requisito intende individuare uno strumento capace di supportare il programmatore nella generazione del layout o dandogli la possibilità di disporre i componenti in maniera visuale o quantomeno permettendogli di individuare i contenitori già disposti facilitandogli così l'implementazione.

4.2.1.9 Altri requisiti

Oltre ai requisiti finora citati ne sono emersi altri che non richiedono particolari spiegazioni. Tra questi troviamo:

1. Funzionalità per commentare e decommentare interi blocchi di codice tramite apposite shortcut;
2. funzioni di ricerca, anche tramite Regular Expression, e di navigazione delle occorrenze trovate sul singolo sorgente: "Trova", "Trova successivo" e "Trova precedente"
3. funzioni per la sostituzione di parole sul singolo sorgente: "Sostituisci"
4. funzioni per la navigazione del sorgente: "Vai alla riga"
5. funzioni per le ricerche globali, dove per globale si intendono tutti i sorgenti in WDL: "Find Usage", che identifica tutti i sorgenti in cui una determinata funzione viene utilizzata, e "Find All" che invece ricerca una parola in tutti i sorgenti, restituendo una lista dalla quale si può accedere al sorgente in l'occorrenza è stata trovata.
6. segnalazione di utilizzo delle funzioni di libreria deprecate, in stile Eclipse;
7. uno strumento che dia la possibilità di raggiungere immediatamente l'implementazione o la documentazione, presente sul sito aziendale interno, delle funzioni builtin. L'idea è quella di realizzare una funzionalità simile ad Eclipse per cui passando con il mouse su funzioni o metodi, appare una tendina che mostra la documentazione riguardante o quantomeno un link a tale documentazione.
8. ultima, ma non meno importante, è la possibilità di eseguire i Wizard direttamente dall'IDE.

4.2.2 Requisiti di sistema

I requisiti visti finora sono indipendenti gli uni dagli altri, perciò è possibile affrontarli uno per volta individuando soluzioni specifiche per ciascuno. Questa caratteristica, però, non è valida per i requisiti di sistema, i quali emergono da problemi principalmente legati alle caratteristiche dell'applicativo o alla sua interazione con l'IDE.

Tra questi individuiamo, in primis, la necessità di evitare la ricompilazione e l'esecuzione dell'intero applicativo ogni volta che si ha il bisogno di eseguire o implementare dei Wizard. Questa procedura, infatti, richiede parecchi minuti a seconda della macchina su cui viene effettutata, e, dato che è un grosso fattore di rallentamento nello sviluppo del software, è il motivo per cui è stata fatta la scelta di interpretare il linguaggio WDL invece di compilarlo. Uno sviluppatore che vorrà implementare o eseguire del codice in WDL, dovrà interagire "a caldo" con l'applicativo, senza il bisogno di spegnerlo, ricompilarlo e rieseguirlo.

In secondo luogo, la soluzione deve tenere conto del fatto che è richiesta una forte integrazione con l'applicativo, sul quale si trovano il parser e l'interprete WDL, tali componenti interagiscono con gli strati inferiori del software, che adottano complesse strategie il cui scopo è quello di garantire l'accesso alle entità. In una situazione del genere l'acceso agli oggetti di Business è, quindi, strettamente vincolato all'applicativo, ed introdurre modifiche alla sua architettura richiede sforzi troppo elevati in confronto ai benefici che si otterrebbero.

4.3 Analisi dei requisiti

In questo paragrafo non si intende approfondire i requisiti già ampiamente descritti precedentemente, ma piuttosto analizzare le interazioni delle diverse entità logiche per tali requisiti, in modo da individuare problematiche legate al comportamento delle varie componenti.

4.3.1 Architettura logica del sistema

Alla luce dei requisiti presentati è facile capire come, e tra quali entità logiche, avvengono le interazioni nel sistema (Figura 16) .

Dal punto di vista logico, infatti, si possono trovare un certo numero n di programmatori P_1, P2, ..., P_n che vogliono accedere alle funzionalità dell'IDE. Si avranno perciò n istanze dell'IDE, una per ciascun sviluppatore, ognuna delle quali interagirà con l'applicativo per accedere alle istanze degli Entity Type in fase di interpretazione dei Wizard.

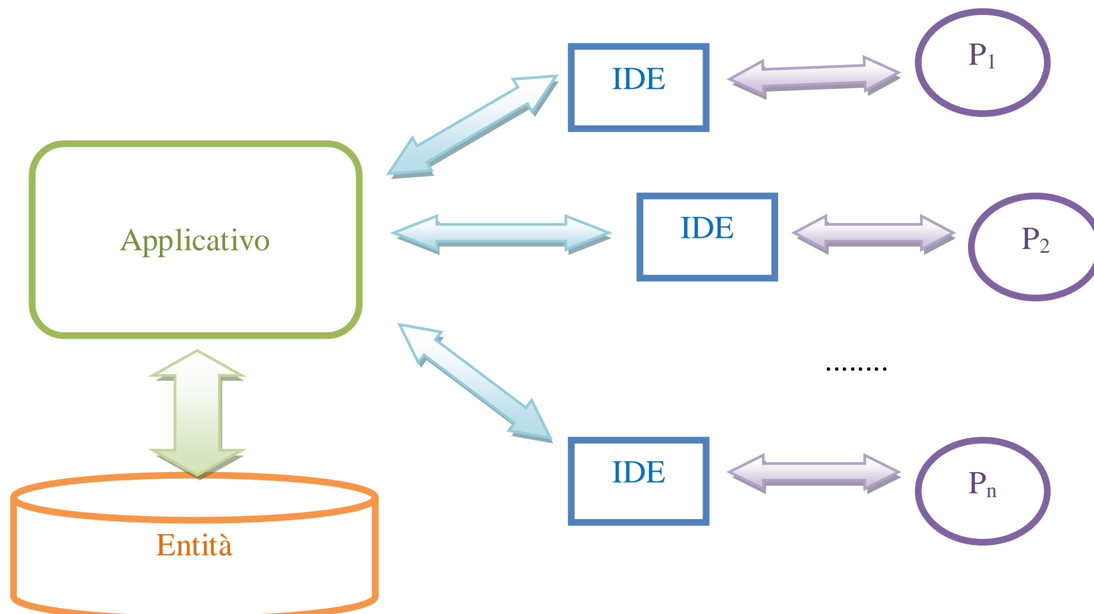


Figura 16 : Architettura logica del sistema

4.3.2 Interazioni nel sistema

A questo livello si può intuire come emergano problemi di concorrenza nell'interazione tra IDE ed applicativo, dato che più programmatori possono implementare contemporaneamente wizard differenti accedendo all'applicativo tramite le funzionalità del proprio IDE.

Gli strumenti dell'IDE che necessitano di questo tipo di interazioni sono sostanzialmente quelli che richiedono di eseguire il wizard, cioè che sfruttano le capacità dell'interprete, oppure quelli che interagiscono solamente con il parser. Tra questi vi sono sicuramente il Debugger, il controllo sintattico, quello semantico e banalmente, la funzionalità di esecuzione classica.

Tutte le rimanenti fuzionalità, come ad esempio il syntax highlighting, l'autocompletamento e le funzioni di ricerca, riguardano quasi esclusivamente l'editor e le interazioni sono per lo più tra il programmatore e questo componente dell'IDE.

Una caratteristica importante che emerge dall'analisi dell'applicativo è l'aspetto distribuito del sistema. La sua architettura, infatti, è stata progettata secondo il paradigma client-server, in cui l'applicativo risiede sul server e gli utenti del sistema vi accedono tramite browser,

i client per l’appunto.   L’IDE stesso infatti presenta alcune componenti sul server e, a seconda delle scelte adottate, potrà o meno essere diventare distribuito anch’esso.

5 Soluzione basata su JavaScript code editor

5.1 Architettura e pattern

La soluzione presentata in questo capitolo ha il principale obiettivo, oltre a quello di soddisfare i requisiti, di realizzare l'integrazione nel modo più naturale possile, senza stravolgere il sistema ma bensì sfruttandone le capacità. Tale presupposto porta immediatamente ad identificare una pagina Web, come l'ambiente di interazione prediletto tra l'IDE e l'utente, esattamente come avviene per l'accesso alle funzionalità di business dell'applicativo per i clienti. Il progetto di questa soluzione è osservabile in Figura 17 ed affronta, in primis, i requisiti di sistema demandando l'implementazione di molte delle features dell'IDE all'estensione di un code editor JavaScript-based integrato nella pagina Web JSP.

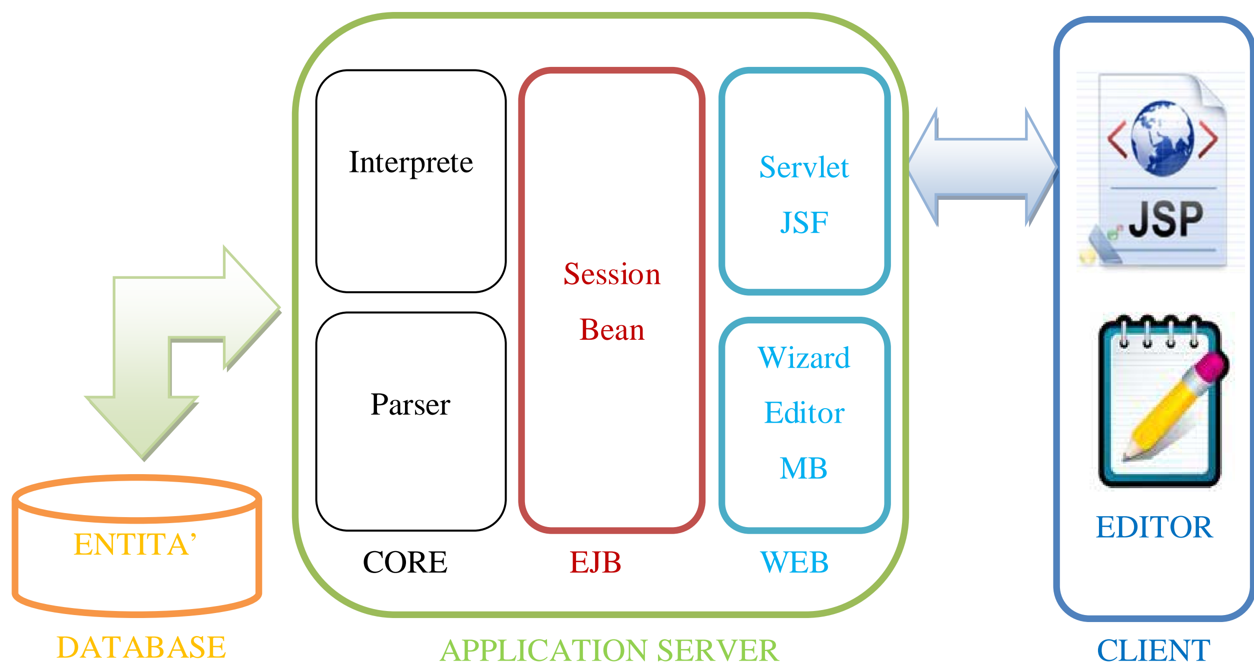


Figura 17: Soluzione basata su JavaScript code editor

Il principale vantaggio di questa soluzione è che risolve tutti i problemi di concorrenza sfruttando il concetto di Session Bean e di staticità dei metodi. Un Session Bean è un componente di logica applicativa usato per modellare le funzioni di business. Il termine Session, sessione, identifica il ciclo di vita del componente. L'application server utilizza tali oggetti durante una sessione utente, creandoli e distruggendoli dinamicamente. Replicare i componenti

grazie al concetto di sessione non risolve, di per sè, i problemi di concorrenza ma li demanda ai componenti del core di cui i programmatori richiedono le funzionalità, cioè il parser e l'interprete. Il comportamento di tali componenti non è legato ad un loro particolare stato ma dipende soltanto dai parametri con cui le funzioni vengono chiamate. I metodi in questione sono stati perciò implementati staticamente ed è proprio grazie a questa proprietà che si evitano effettivamente i problemi legati alla concorrenza.

All'interno di una sessione, l'utente può richiedere al server nuove pagine JSP/JSF oppure, interagendo con la pagina che ha davanti, può aver bisogno di informazioni o funzionalità dal server. Questo tipo di interazioni vengono gestite dalla servlet JSF che si occupa di recuperare le pagine e tradurre le azioni in metodi dei cosidetti Managed Bean (MB). Nell'architettura in Figura 17 è infatti presente il WizardEditorMB, la cui funzione è quella di gestire le azioni dell'utente come il salvataggio, il controllo di sintassi e di rispondere ad una richiesta di informazioni dell'editor.

Quest'approccio permette di sviluppare i Wizard "a caldo" direttamente sull'applicativo evitando quindi i problemi dovuti ai tempi di compilazione.

5.2 I source code editor

Un source code editor è un programma per redigere testo progettato specificatamente per comporre il codice sorgente dei programmi per computer da parte dei programmatori. Un programma del genere può presentarsi come un'applicazione standalone, può essere costruito all'interno di un IDE [12] oppure può essere incapsulato all'interno di pagine Web.

Editor di questo tipo forniscono una serie di funzionalità per rendere più veloce la scrittura di codice sorgente, come il syntax highlighting, l'autocompletamento, la funzionalità di match delle parentesi e molte altre. Solitamente forniscono anche un facile accesso alle funzionalità di compilatori, interpreti, debugger o altri programmi rilevanti nel processo di sviluppo del software.

Un JavaScript-based code editor è un source code editor sviluppato interamente in tecnologia JavaScript e per questo è integrabile in pagine Web.

5.2.1 JavaScript: accenni

JavaScript è un linguaggio di scripting sviluppato per dare interattività alle pagine HTML, il suo uso pervasivo lo ha reso lo standard de facto per l'elaborazione lato client. Al di la del nome, Java e JavaScript sono due cose completamente diverse, l'unica similitudine è legata al fatto di aver entrambi adottato la sintassi del C. Le differenze invece sono molte e profonde:

- JavaScript è un linguaggio interpretato e non compilato, viene infatti eseguito dall'interprete integrato nei browser Web;
- in JavaScript non esiste il concetto di classe ma soltanto quello di oggetto;
- il linguaggio JavaScript è debolmente tipizzato, ciò significa che non è necessario definire il tipo di una variabile. Attenzione però : questo non vuol dire che i dati non abbiano un tipo, sono le variabili a non averlo in modo statico.

Il linguaggio nasce per dare dinamicità alle pagine Web, viene infatti utilizzato per accedere e modificare elementi della pagina HTML, per reagire ad eventi generati dall'interazione fra utente e pagina, per validare i dati inseriti dall'utente e per interagire con il browser, quando, ad esempio, c'è bisogno di determinare il tipodi browser utilizzato e la dimensione della finestra in cui viene mostrata la pagina, lavorare con i cookie ecc.[13].

5.2.2 I più diffusi JavaScript-based code editor

Navigando in rete si trovano moltissimi editor di questo tipo. Nonstante siano molto simili dal punto di vista funzionale, differiscono gli uni dagli altri per numerosi altri aspetti. Scegliere un programma del genere, per inserirlo in un progetto aziendale, non significa soltanto individuare quello con le migliori funzionalità, ma vanno valutati anche altri fattori a volte ritenuti secondari, ma che spesso però si rivelano di gran lunga più importanti. Alcuni di questi fattori sono:

- il tipo di licenza con cui viene rilasciato;
- lo stato del progetto, cioè che sia ancora attivo e non sia stato abbandonato;
- la presenza di una documentazione il più possibile completa, ricca di esempi a cui attingere se si incontrano difficoltà;
- l'open-sourcing, la possibilità di disporre dei sorgenti per estendere, correggere e comprendere meglio il tool;
- il costo, se utilizzabile previo pagamento oppure liberamente scaricabile;
- la compatibilità, il supporto più ampio possibile dei browser Web fin dalle loro prime versioni;
- il supporto di comunità di utenti o degli stessi sviluppatori.

I JavaScript-based code editor migliori e, non a caso, anche i più diffusi, sono Ace e CodeMirror. Tanto per cominciare entrambi sono free, open-source, e sono rilasciati sotto licenze classificate come "Permissive free software licence", che permettono quindi il riuso all'interno di software proprietari a patto che tutte le copie del software includano la propria

licenza. Ace infatti è rilasciato sotto la licenza "new BSD" (Berkeley Software Distribution), mentre CodeMirror sotto quella "MIT", ideata dal Massachussetts Institute of Technology.

Analizzando la compatibilità dei due editor con i più diffusi browser Web, si nota come CodeMirror mostri un più ampio range di versioni supportate. CodeMirror è infatti compatibile con Firefox 2+, Chrome, Safari 3+, Internet Explorer 7+, Opera 9+ mentre per quanto riguarda Ace il supporto è garantito per Firefox 3.5+, Safari 4+, Chrome, IE 9+, Opera 11.5+.

Entrambi gli editor presentano un'ampia documentazione sul proprio sito internet per capire nel dettaglio come inserirli in pagine Web, come personalizzarne lo stile (temi), come configurare ed attivare le funzionalità dell'editor e come utilizzare le API, ove disponibili, per estendere il comportamento di default in modo da adattarli ai propri bisogni. Per quanto riguarda le funzionalità, infatti, entrambi gli editor forniscono un set di strumenti tra i più completi, come mostrato nella Tabella in Figura 18: syntax highlighting, bracket matching, indentazione automatica, folding, ecc. Sono in grado di riconoscere e quindi applicare questi strumenti per più di quaranta linguaggi, ma ovviamente non possono, di default, saper riconoscere il linguaggio WDL in quanto linguaggio custom. Gli aspetti interessanti non sono quindi le funzionalità messe a disposizione per i linguaggi conosciuti, ma piuttosto la possibilità di estenderle a linguaggi custom, perchè è questo che si ha intenzione di fare.

Di seguito viene mostrata una tabella che confronta le features dei due editor.

Feature	Ace	CodeMirror
Implementazione		Parser completi innestabili
Syntax highlight	Sì	Sì
Syntax checking	HTML, CSS, JavaScript	Alcuni
Tab support	Sì	Sì
Indent, new line keeps level	opzionale	Sì
Indent, selected block	Sì	Sì, anche automatico
Bracket matching	Sì	Sì

Code folding	Sì	Sì
Code snippets	No	Da implementare tramite API
Autocompletamento	No	Da implementare tramite API
Toggle syntax highlight on/off	Sì	Sì
Keyboard shortcuts	Tutte le shortcut più comuni	Da implementare tramite API
Numeri di linea	Sì	Sì
Search & replace	Supporto RegExp	Via API (supporto RegExp)
Temi	Sì	Sì
Undo/Redo	Sì	Sì
Multiple cursors / Block selection	Multiple cursors	No
Linee guida per l'indetazione	Sì	No

Figura 18: Tabella per il confronto delle funzionalità tra Ace e CodeMirror

Ciò che dovrebbe saltare all'occhio leggendo la tabella è che, a differenza di Ace, CodeMirror è uno strumento più flessibile perchè presenta delle API che permettono l'estensione dello strumento con nuove funzionalità e quindi anche la sua personalizzazione. Ace è più rigido in tal senso, perchè nonostante presenti numerossime funzionalità, non vi è la possibilità per il programmatore di aggiungerne di nuove. Tale caratteristica si è rivelata fondamentale nella scelta dell' editor da adottare, in aggiunta al fatto che Ace non supporta l'autocompletamento, uno dei requisiti più richiesti dagli utenti dell'IDE.

5.3 CodeMirror

5.3.1 Overview

Il sito ufficiale definisce CodeMirror come “un componente code-editor che può essere incapsulato in pagine Web”. Specifica anche che “la libreria fornisce soltanto il componente editor, nessun pulsante, autocompletamento o altre funzionalità tipiche degli IDE, sono presenti. Fornisce però delle API complete sulle quali tali funzionalità possono essere implementate linearmente”[14].

Il fatto che bisogna implementarsi autonomamente funzionalità, anche complesse, non deve scoraggiare, dato che, sia sul sito che nei sorgenti scaricabili, sono presenti delle demo complete che mostrano come utilizzare le API nel modo migliore.

5.3.2 CodeMirror: componenti principali

Allo scopo di comprendere meglio come CodeMirror è strutturato e quali sono le componenti di cui è costituito, verrà ora analizzati la gerarchia dei sorgenti. In questa discussione e nelle seguenti ci si riferirà sempre alla versione 2.3 di CodeMirror.

In riferimento alla Figura 19, all’interno dei sorgenti di CodeMirror si trovano cinque cartelle, raffigurate ciascuna all’interno dei riquadri azzurri. La prima, denominata “demo”, contiene dei file Html, uno per ciascuna funzionalità di cui si vuole vedere la demo. Tutte le

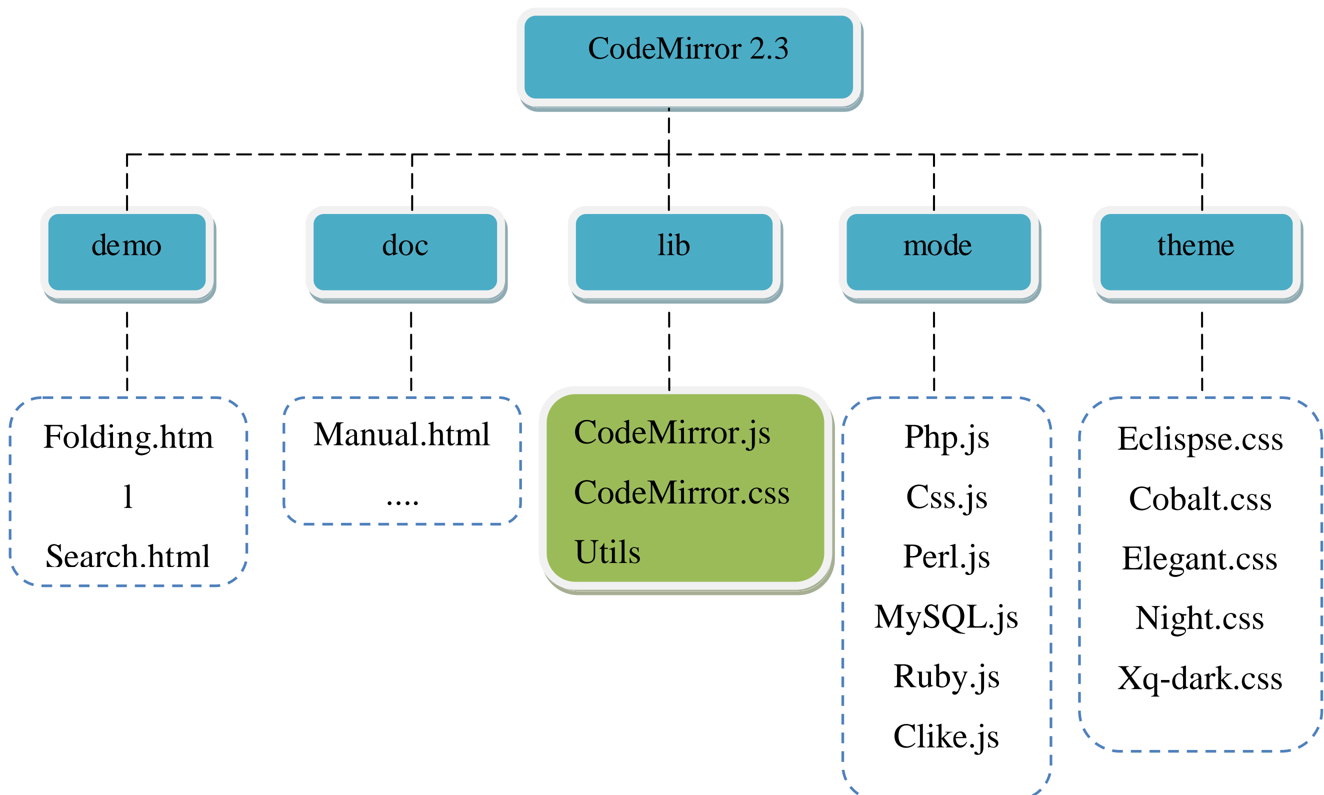


Figura 19: Analisi dei sorgenti di CodeMirror

demo mostrano l'editor CodeMirror con cui poter interagire per capire il funzionamento della features lato utente. L'editor stesso nella demo è stato sfruttato per mostrare il codice in modo da capire come utilizzare al meglio le Api. Una volta aperta è possibile anche ispezionare il codice Html per capire come l'Editor è stato configurato, grazie alle funzionalità del browser, ad esempio, premendo F12 in Chrome.

La cartella "doc", contiene tutta la documentazione. La stessa presente anche sul sito. Il file *Manual.html*, in particolare, elenca e spiega, una ad una, le API di CodeMirror, non prima di mostrare come configurare e personalizzare il tool.

Nella cartella "lib" si trova un unico sorgente: quello dell'editor CodeMirror, accompagnato da un foglio di stile Css che ne definisce l'aspetto di default. E' in questo file JavaScript che sono contenute tutte le implementazioni delle logiche per le API ed è quindi tramite questo file che è anche possibile istanziare l'editor all'interno di una pagina Web. All'interno del foglio di stile omonimo, *CodeMirror.css*, si trovano le classi per gli stili dei componenti grafici dell'editor o per la configurazione di default di funzionalità come Syntax highlighting. Si veda il codce mostrato in Figura 20.

In realtà è presente anche una sottocartella denominata "util" che contiene le implementazioni delle logiche di strumenti di utilità come ad esempio, la funzionalità di ricerca, il folding, il formatting. Tali librerie andranno importate esplicitamente per poterne sfruttare le capacità.

Nella cartella "mode" sono presenti i sorgenti che definiscono i modi. I modi sono programmi JavaScript utilizzati da CodeMirror per colorare ed indentare il testo scritto in un certo linguaggio[14]. In tali oggetti, infatti sono definite le logiche per effettuare il parsing delle linee allo scopo di attribuire alle parole una classe. In base alla classe attribuita ed alla definizione del colore per quella classe nel tema corrente, le parole verrano colorate differentemente. I modi garantiscono un'ottima flessibilità anche nel colorare Foreign Code all'interno dei linguaggi, proprio come ci sarà bisogno di fare per il linguaggio WDL.

```
/* Default theme */

.cm-s-default span.cm-keyword {color: #708;}
.cm-s-default span.cm-atom {color: #219;}
.cm-s-default span.cm-number {color: #164;}
.cm-s-default span.cm-def {color: #00f;}
.cm-s-default span.cm-variable {color: black;}
.cm-s-default span.cm-property {color: black;}
.cm-s-default span.cm-operator {color: black;}
.cm-s-default span.cm-comment {color: #a50;}
.cm-s-default span.cm-string {color: #a11;}
.cm-s-default span.cm-meta {color: #555;}
.cm-s-default span.cm-error {color: #f00;}
.cm-s-default span.cm-qualifier {color: #555;}
.cm-s-default span.cm-builtin {color: #30a;}
.cm-s-default span.cm-bracket {color: #997;}
.cm-s-default span.cm-tag {color: #170;}
.cm-s-default span.cm-attribute {color: #00c;}
.cm-s-default span.cm-header {color: blue;}
.cm-s-default span.cm-quote {color: #090;}
.cm-s-default span.cm-hr {color: #999;}
.cm-s-default span.cm-link {color: #00c;}

div.CodeMirror span.CodeMirror-matchingbracket {color: #0f0;}
div.CodeMirror span.CodeMirror-nonmatchingbracket {color: #f22;}
```

Figura 20: Alcune classi di stile del file *CodeMirror.css*

In ultima analisi, la cartella “theme” contiene alcuni temi preconfigurati per personalizzare moltissime caratteristiche dell’aspetto di default dei componenti grafici di CodeMirror come, ad esempio, il colore dello sfondo dell’Editor, il font e la dimensione del testo, ecc. In questi file sono definite, come già accennato, le associazioni tra colore e classi di token per le funzionalità di syntax highlighting, in maniera molto simile a quella usata in *CodeMirror.css* (Figura 20).

5.4 Sviluppi

In questa sezione vengono mostrate tutte le implementazioni realizzate allo scopo di soddisfare il maggior numero di requisiti. Purtroppo, infatti, non c’è stato spazio nell’ambito di questa Tesi, di implementare tutte le funzionalità richieste. Alcune ritenute meno prioritarie sono state accantonate, altre invece sono giunte alla fase progettuale ma non c’è stato il tempo di completarle, come il Debugger.

5.4.1 WizardEditorMB

Questo componente, assieme alla pagina JSP, è stato uno dei primi ad essere realizzato in quanto rappresenta il punto di accesso tra la pagina JSP stessa e le funzionalità dell'applicativo che più interressano cioè, il parsing, l'interpretazione ed il salvataggio. Oltre a questo, funge anche da contenitore per le informazioni utili a CodeMirror durante una sessione utente, infatti qui si trova un'elenco dei principali elementi WDL come parole chiave, operatori e funzioni builtin. Queste informazioni vengono reperite tramite appositi metodi al caricamento della pagina JSP per poi essere passate all'editor che le utilizza per implementare, come si vedrà in seguito, funzionalità quali syntax highlighting, auto-completamento e controllo sintattico (Figura 21).

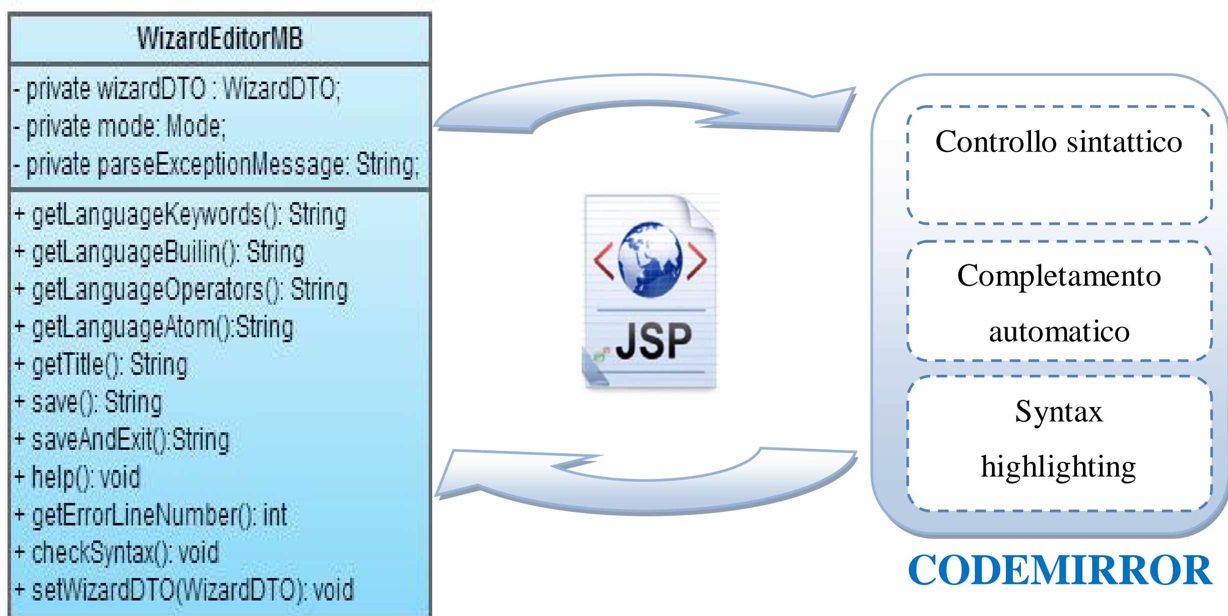


Figura 21: Interazione tra WizardEditorMB e CodeMirror

Questo Bean è stato arrichito in modo incrementale. Man mano che si procedeva con lo sviluppo delle funzionalità dell'IDE e ci si accorgeva di quali informazioni l'editor aveva bisogno, venivano aggiunti i metodi per reperirle. Di seguito ne vengono mostrati alcuni.

Il metodo *getLanguageKeywords()*, mostrato in Figura 22, restituisce l'elenco delle keyword in minuscolo separate dal carattere *","* . Allo stesso modo sono implementati i metodi :

- *getLanguageOperators()* che restituisce gli operatori di WDL, come +, *, -, \, ecc;
- *getLanguageBlockKeyword()* che restituisce le keyword per i blocchi di istruzioni come : *foreach, while, if, step, foreachdate*, utili durante il parsing effettuato per il syntax highlighting per determinare il contesto e per la funzionalità di folding;
- *getLanguageBuiltin()*, questo metodo contiene tutti i nomi delle funzioni builtin finora sviluppate per WDL;
- *getLanguageAtoms(),* restituisce semplicemente gli atomi: *true*, *false*, *null*;

```
public String getLanguageKeywords(){
            return "var,if,then,else,for,foreach,foreachdate," +
                         "while,return,break,step,init,"+
                         "foreachstep,on_refresh,validation," +
                         "on_next,on_previous,style,layout,div,"+
                         "asc,desc,like,exists,new," +
                         "previous,cancel,asynchronous,parallel," +
                         "disabled,return,and,or,not,aggregate on," +
                         "order by";
     }
```

Figura 22: Il metodo *getLanguageKeywords()*

Il metodo *help()* è utilizzato per recuperare le informazioni da mostrare all'utente alla pressione del pulsante contrassegnato dal simbolo "?". L'utilità di questo pulsante è appunto quella di aiutare l'utente nell'uso dell'editor suggerendogli le shortcut per attivare le funzioni a disposizione (Figura 23).

```
public void help() {
        StringBuilder stringBuilder = new StringBuilder(300);
        stringBuilder.append("[Ctrl+Space]  Autocompletion <br/>");
        stringBuilder.append("[F11/Esc]      Toggle Fullscreen <br/>");
        stringBuilder.append("[Curly Brace] " +
                             "click on number to Fold/Unfold <br/>");
        stringBuilder.append("[Ctrl+L]       Go To Line <br/>");
        stringBuilder.append("[Ctrl+F]       Find <br/>");
        stringBuilder.append("[Ctrl+R]       Replace <br/>");
        stringBuilder.append("[Ctrl+G]       Find Next <br/>");
        stringBuilder.append("[Ctrl+Shit+G]  Find Previous <br/>");
        stringBuilder.append("[TAB]          Indent<br/>");
        stringBuilder.append("[Shift+TAB]    Unindent <br/>");
        stringBuilder.append("[Ctrl+D]       Delete Line <br/>");
        stringBuilder.append("[Ctrl+H]       Comment <br/>");
        stringBuilder.append("[Ctrl+Shift+H] Uncomment <br/>");
        stringBuilder.append("[Ctrl+Shift+S] Save <br/>");
        stringBuilder.append("[Ctrl+Shift+A] Toggle Auto-Save<br/>");
        stringBuilder.append("Best experienced " +
                             "with Mozilla Firefox 14.0 or greater.");

        WebUtils.setSimpleModalMessage(stringBuilder.toString());
    }
```

Figura 23: Il metodo *help()*

Il *wizardEditorMB* non contiene soltanto metodi per restituire informazioni, ma anche funzioni che attivano altri componenti nei livelli inferiori dell'applicativo. E' il caso del metodo *checkSyntax()* (Figura 24) che, attivato dall'omonimo pulsante nella pagina JSP, chiama il parser per effettuare il controllo sintattico del codice mostrato sull'editor, poi, in base all'esito,

riporta una notifica di successo oppure, in caso di errore, il messaggio di errore generato dal Parser JavaCC.

```
public void checkSyntax() {
            String sourceCode = wizardDTO.getSourceCode();
            if (sourceCode == null)
                   return;
            try {
                   DotNotationParser.parseWizard(sourceCode);
             String msg = MessagesUtil.getMessageResourceString
                     (WebConstants.BUNDLE, "Wizard_CheckSyntaxOk",
                      FacesContext.getCurrentInstance());
             addInfoMessage(addTimeStamp(msg), saveButtonID);
              parseExceptionMessage = null;
            } catch (ParseException e) {
                   addErrorMessage(
                      addTimeStamp(
                           MessagesUtil.getMessageResourceString(
                                 WebConstants.BUNDLE,
                                "Wizard_CheckSyntaxError",
                                FacesContext.getCurrentInstance()
                           )
                      ) + ": " + e.getMessage(), saveButtonID);
                   parseExceptionMessage = e.getMessage();
            }
     }
```

Figura 24: Il metodo *checkSyntax()*

Un'altro metodo degno di nota è quello che effettua il salvataggio del codice WDL sull'applicativo, il metodo *save()* mostrato in Figura 25.

```
public String save() {
        if (doSave()) {
            String msg = MessagesUtil.getMessageResourceString(
                          Constants.Messages.ANTARES_BUNDLE,
                          Constants.Messages.SAVE_SUCCESS,
                          FacesContext.getCurrentInstance());
            addInfoMessage(addTimeStamp(msg), saveButtonID);
            init(wizardDTO, Mode.EDIT);
        }
         return null;
     }
```

Figura 25: Il metodo *Save()*

5.4.2 La pagina JSP.

All'interno di questa pagina sono stati definiti i principali componenti visuali grazie ai quali il programmatore interagirà con i servizi applicativi. Infatti, importando le opportune librerie all'interno della sezione "head" della pagina, è possibile utilizzare i componenti JSF, AJAX e HTML nel "body" tramite gli appositi tag. Allo stesso modo vengono importate le librerie di CodeMirror per poter istanziare l'editor.

In Figura 26 viene mostrata l'interfaccia dell'IDE, dalla quale si scorge, procedendo dall'alto verso il basso:

- La barra di navigazione dell'applicativo. Utile per muoversi nelle sezioni principali dell'applicativo come il Pannello di Controllo o gli strumenti, presenta anche un menù a tendina per raggiungere le pagine preferite. Ogni pagina dell'applicativo viene creata a partire da un template in cui le parti comuni vengono definite, la barra di navigazione è una di queste;
- I campi informativi. Contengono informazioni riguardo il Wizard attualmente caricato sull'editor o in fase di creazione, come il titolo in italiano, quello in inglese, ed una descrizione del Wizard;
- L'Editor CodeMirror. Qui è possibile editare il codice WDL supportati da funzioni come Syntax Highlighting, autocompletamento, folding, search, ecc;
- Una serie di pulsanti. Il loro scopo è quello di effettuare operazioni come il salvataggio, l'uscita dalla pagina, il controllo sintattico e la visualizzazione dell'help;
- Una zona per le notifiche. Affiancata ai pulsanti, quest'area viene usata per comunicare il successo o il fallimento delle operazioni attivate tramite pulsanti.

i4C fore see

Utente: Administrator Preferiti Pannello di controllo Strumenti Logout

Modifica Wizard

Nome wizard updEmailField * Descrizione

Titolo italiano updEmailField * Titolo inglese

* I campi sono obbligatori

```
1  STEP("input","Modify e-mail "){
2       VAR entity = E4CProperties[ID="DEFAULTPROPERTIES"];
3       SHOWMESSAGE([
4             "key" : "E-mail: " + entity.auth_mail_sender ,
5             "id" : "idEmail"
6       ]);
7
8
9       VAR new_address;
10      TEXTBOX([
11          "primitiveType" : "STRING",
12          "variable": "new_address",
13          "key" : "New E-mail"
14     ]);
15
16     VAR message;
17     SHOWMESSAGE([
18            "key" : ""+message,
19            "id" : "ok",
20            "labelStyleClass" : "successMessageStyle",
21            "messageStyleClass" : "successMessageStyle"
22     ]);
23     REFRESH_BUTTON([
```

Salva Esci Salva ed esci Controllo sintattico ? Controllo sintattico eseguito con successo (23.05.34)

Copyright © 2005-2012, i4C Srl. Tutti i diritti riservati
i4C BAF v5.5.M4.i4C-SNAPSHOT (2012-08-09 15:12:18,REV.17002) Guitar Pro 5

Powered by i4C

Figura 26:Aspetto della pagina JSP

La parte più interessante contenuta nella definizione della pagina JSP è la configurazione dell'editor CodeMirror, di cui viene mostrato il codice in Figura 27. Come si può osservare il codice di configurazione è costituito da uno script JavaScript in cui vengono istanziati gli oggetti *i4Clanguage* e *weCodeMirrorOptions*. Il primo configura il tipo di modo ed i suoi parametri. I modi, come si approfondirà in seguito, sono oggetti che definiscono il comportamento di CodeMirror per funzionalità legate al linguaggio specifico come il syntax highlighting, ma non solo. In questo caso, il modo associato a WDL è denominato *i4Cmixed* i cui parametri sono gli elementi del linguaggio prelevati dal *wizardEditorMB*, splittati dalla funzione *word*, memorizzati in *i4Clanguage* ed in seguito iniettati in CodeMirror tramite l'oggetto *weCodeMirrorOptions*. Quest'ultimo ha lo scopo di memorizzare le opzioni per CodeMirror:

- Attivazione di funzionalità statiche, come la numerazione delle linee o la funzione di match delle parentesi;
- Configurazione di parametri, come il modo, il tema, la dimensione del "tab", la dimensione dell'indentazione, ecc.

- Configurazione delle shortcut. Contenute nell'oggetto *extraKeys*, sono delle associazioni tra combinazione di tasti e funzione da attivare;
- Attivazione di funzionalità dinamiche, cioè attivate a seguito di particolari eventi, come *onBlur*, *onGutterClick*, *onCursorActivity,*ecc.

```
<script>
     var i4Clanguage = {
        name: "i4Cmixed",
       keywords: words("#{wizardEditorMB.languageKeywords}"),
        blockKeywords:
           words("#{wizardEditorMB.languageBlockKeywords}"),
        builtin: words("#{wizardEditorMB.languageBuiltin}"),
        atoms: words("#{wizardEditorMB.languageAtoms}"),
        operators: words("#{wizardEditorMB.languageOperators}")
    };
    var weCodeMirrorOptions = {
           lineNumbers: true, matchBrackets: true,
           mode: i4Clanguage, theme: 'eclipse',
           onGutterClick: CodeMirror.newFoldFunction(
                             CodeMirror.braceRangeFinder),
           indentUnit: 4, tabSize: 4, smartIndent: false,
        indentWithTabs: true, electricChars: false,
         extraKeys: {
                 "F11": toggleFullScreen, "Esc": toggleFullScreen,
                 "Ctrl-R": "replace", "Shift-Tab": "indentLess",
                 "Ctrl-H": commentSelection,
               "Shift-Ctrl-H": unCommentSelection,
                 "Ctrl-Space": "autocomplete",
                 "Shift-Ctrl-S": fastSave,
               "Shift-Ctrl-A": toggleAutoSave,
                 "Ctrl-L": "gotoLine"
           },
           onBlur: function(){ weCodeMirror.save(); },
           onCursorActivity:
               function() {
                  weCodeMirror.setLineClass(hlLine, null, null);
                    hlLine = weCodeMirror.setLineClass(
                             weCodeMirror.getCursor().line,
                             null, "activeline");
                    weCodeMirror.matchHighlight(
                             "CodeMirror-matchhighlight");
                 }
           };
     </script>
```

Figura 27:Configurazione di CodeMirror

Non meno importante è il codice, mostrato in Figura 28 in cui l'editor viene istanziato passandogli i parametri appena discussi ed il codice sorgente del Wizard.

```
<h:inputTextarea
        id="Wizard_SourceCode"
        value="#{wizardEditorMB.wizardDTO.sourceCode}"/>
<!-- TEXT AREA to CODEMIRROR -->
<script>
     CodeMirror.commands.autocomplete = function(cm) {
                                            CodeMirror.simpleHint(
                                                       cm,
                                                       CodeMirror.i4CHint,
                                                       i4Clanguage);
                                       };
     var weCodeMirror = CodeMirror.fromTextArea(
                                  document.getElementById(
                                       'createWizardForm:Wizard_SourceCode'),
                                   weCodeMirrorOptions);
     var hlLine = weCodeMirror.setLineClass(0, "activeline");

     updateErrorSymbol(#{wizardEditorMB.errorLineNumber});
</script>
```

Figura 28: Creazione di un' istanza dell' editor CodeMirror

5.4.3 Syntax Highlighting: il modo i4CMixed

L'implementazione del syntax highlighting in CodeMirror è legata quasi esclusivamente ai modi. Nei modi infatti sono definiti i tokenizer, il cui scopo è quello di individuare parole e caratteri del linguaggio attribuendogli un'etichetta. Sulla base di questa etichetta CodeMirror colorerà il token prelevando il colore dalla classe CSS in cui è definito il tema. Se ad esempio il tokenizer rioconosce un certo token come parola chiave vi attribuirà l'etichetta *keyword*. Il tema configurato è denominato *eclipse* e nel file *eclipse.css,* sotto la cartella *theme,* è definita la classe CSS per colorare le parole chiave in Bourdeaux e settare il font in grassetto:

```
.cm-s-eclipse span.cm-keyword{line-height:1em; font-weight:bold; color:#7F0055;}
```

Allo scopo rendere fruibili a CodeMirror le funzionalità, ogni modo deve implementare un'interfaccia definita dalle funzioni:

- *startState*. Chiamato in fase di inizializzazione questa funzione restituisce lo stato iniziale del modo;
- *token*. E' la funzione usata per spezzare righe in token e attribuirvi le classi CSS ai token. E' utile anche per determinare il contesto, cioè un nuovo blocco di codice delimitato dalle parentesi graffe o un'istruzione, informazioni sfruttate anche dall'autocompletamento.

- *Indent.* Questa funzione viene chiamata da CodeMirror quando è attiva l'indentazione automatica o quella tramite "tab" per ricalcolare la posizione dei caratteri sulla riga.

La definizione di un tokenizer all'interno del modo si ottiene ovviamente implementando la funzione *token*. CodeMirror utilizza le tecniche tipiche della Delimiter Directed Translation (Appendice C) per supportare l'highlight, infatti scompone il codice il linee delimitate dal carattere "a capo". Per ciascuna di esse viene chiamata la funzione *token,* una volta per ogni parola della linea, che esegue le seguenti operazioni:

1. identifica un token ignorando gli spazi bianchi;
2. assegna un'etichetta al token;
3. aggiorna il contesto;

Tutte le operazioni elencate sono state implementate sfruttando semplici funzioni messe a disposizione dalle API di CodeMirror e grazie alle Regular Expression.

Per la fase uno il codice implementato è il seguente:

```
if (stream.eatSpace()) return null;
var style = tokenBase(stream, state);
```

L'oggetto *stream* contiene il flusso di caratteri per una linea; la funzione *eatSpace()* permette di avanzare nello stream ignorando gli spazi bianchi e per tali caratteri non viene assegnata un'etichetta, ecco perchè il valore di ritorno è *null*. Lo stream in questo modo è posizionato all'inizio di un token che verrà elaborato dalla funzione *tokenBase* per la fase due. Di seguito viene mostrato una parte del codice.

```
var ch = stream.next();
if (ch == '"' || ch == "'") {
       ......
      return state.tokenize(stream, state);
}
if (/\d/.test(ch)) {
      stream.eatWhile(/[\w\.]/);
      return "number";
}
if (ch == "#") {
      stream.skipToEnd();
      return "comment";
}
stream.eatWhile(/[\w\$_]/);
var cur = stream.current();
if (keywords.propertyIsEnumerable(cur)) {
      ......
      return "keyword";
```

```
}
if (builtin.propertyIsEnumerable(cur)) {
      return "builtin";
}
```

Le prime istruzioni analizzano soltanto il carattere successivo nello stream per identificare il simbolo del commento, che in WLD è lo sharp ("#"), caratteri sintattici, numeri o i simboli di apertura di una stringa. Nelle differenti situazioni il comportmento del codice è differente:

- commento: il cursore viene spostato alla fine dello stream e quindi della linea. Tutti i caratteri vengono identificati in un unico token a cui viene assegnata l'etichetta *comment*;
- numero: si identificano tutti i digit fino al primo spazio bianco. Raggruppati in un solo token vengono etichettati come *number;*
- caratteri inizio stringa: vengono gestiti dalla funzione *tokenString()* che si occupa di identificare la terminazione della stringa per assegnare a tutto il token l'etichetta *string;*
- caratteri sintattici: si intendono le parentesi o simboli di punteggiatura che solitamente non vengono colorati;

Successivamente lo stream viene avanzato fino al primo spazio bianco e si verifica, usando la funzione JavaScript *propertyIsEnumerable*, se il token identificato appartiene a una delle liste di parole WDL passate come parametro al modo, cioè quella delle parole chiave, delle builtin, degli operatori, ecc. Se c'è una corrispondenza verrà assegnata l'etichetta relativa. Nell'esempio viene mostrata la procedura per le parole chiave e per le builtin.

La logica, in realtà, è più complicata data la presenza di Foreign Code CSS all'interno di WDL di cui si vuole effettuare un syntax highlight differente e che, altrimenti, verrebbe colorato come una unica stringa. Per risolvere il problema si è dovuti ricorrere ad un implementazione del modo su più strati, come mostrato in Figura 29.

Figura 29: Architettura a layer del modo

In questa architettura il ruolo del modo *i4CMixed* è quello di orchestrare i modi *CSS* e *WDL* al livello sottostante. Dovrà quindi analizzare lo stream per riconocere particolari sequenze di caratteri e poi demandare a uno dei due modi l'assegnazione dell'etichetta per lo specifico linguaggio. Il codice CSS, infatti, si può trovare in WDL soltanto all'interno di una stringa nel blocco *Style*:

```
STYLE : {
    ".containerRow {overflow:auto;clear:both;}
    .alignedLabel {float:left; margin-top:5px;}
    .spaced {margin-left:25px;}
    .element {float:left;}
    .fullWidth {width:100%;}
    .left {float:left; width:44%;overflow:auto;
           padding-left:18px;padding-right:18px;padding-top:20px;}
    .commands {clear:both; padding-top:18px;}"
    + CoreLibrary.getTextboxDefaultStyleClass("dateTextboxClass",70)
    + CoreLibrary.getDefaultCss();
}
```

i4Cmixed utilizza di default il modo *WDL,* quando riconosce la sequenza di token *STYLE :{* , memorizza che il contesto è cambiato. In seguito al riconoscimento del caratteri " o ', attiva poi il modo CSS finchè non si presenta il carattere di chiusura della stringa. Questa logica viene implementata in *i4CMixed* con il seguente codice:

```
function i4C(stream, state) {
    var style = i4CMode.token(stream, state.i4CState);
    var cur = stream.current();
    if (style == "keyword" && cur.toUpperCase() === "STYLE") state.ctx="afterStyle";
    if(state.ctx=="afterStyle" && cur==":") state.ctx="after:";
    if(state.ctx=="after:" && cur=="{"){
      state.ctx="after{";
      state.i4CState.onStyleBlock=true;
    }
    if(state.ctx=="after{" && style == "string"  &&  cur == "\""){
            state.ctx="css";
            state.token = css;
            state.localState = cssMode.startState(i4CMode.indent(state.i4CState, ""));
            state.mode = "css";
    }
    if(state.ctx=="after{" && cur=="}") {
      state.ctx="i4Clang";
      state.i4CState.onStyleBlock=false;
    }
    return style;
  }

  //highlights CSS language
  function css(stream, state) {
      var style = cssMode.token(stream, state.localState);
      if(style == "string"  && state.ctx=="css"){
      var cur = stream.current();
         if(cur == "\""){
             state.ctx="after{";
             state.token = i4C;
             state.localState = null;
```

```
            state.mode = "i4C";
        }
    }
    return style;
}
```

Questa soluzione ha l'unico difetto che qualsiasi stringa viene inizializzata all'interno del blocco *Style* viene poi colorata con il modo CSS. La soluzione è comunque funzionale per il linguaggio WDL e il difetto passa praticamente inosservato, soltanto un'occhio attento ne avrebbe notato la presenza nell'esempio mostrato, dato che il codice CSS in questi blocchi è preponderante.

5.4.4 Implementazione dell'autocompletamento

In fase di configurazione dei parametri dell'editor, è stata assegnata alla combinazione di tasti *Ctrl + Space* l'esecuzione del comando *autocomplete* a cui viene associata una specifica funzione grazie al seguente codice.

```
CodeMirror.commands.autocomplete = function(cm) {
                                        CodeMirror.simpleHint(
                                                cm,
                                                CodeMirror.i4CHint,
                                                i4Clanguage
                                        );
                                   };
```

Il corpo di questa funzione richiama un'altra funzione chiamata *SimpleHint*, implementata nella classe di utilità di CodeMirror omonima contenuta nella cartella *Utils*. A *SimpleHint* è affidato il compito di :

1. Effettuare la chiamata alla funzione per l'individuazione dei suggerimenti;
2. Creare la tendina nell'editor in cui mostrare i suggerimenti all'utente;
3. Inserire i suggerimenti nel componente grafico;

I suggerimenti vengono individuati dalla funzione *i4CHint* definita nell'omonima classe e indicata a *SimpleHint* tra i parametri in modo che possa essere chiamata al momento opportuno. Questa classe è quella che ricopre maggior importanza nell'implementazione dell'autocompletamento in quanto determina i suggerimenti non solo per gli elementi statici di WDL come parole chiave, le funzioni builtin e gli atomi, ma anche per quelli definiti dinamicamente come le variabili, legate tra l'altro ad un contesto di validità.

Per gli elementi statici *i4CHint* ricerca le corrispondenze tra i caratteri digitati e l'elenco dei termini di WDL contenuti nell'oggetto *i4CLanguage* e le inserisce nell'elenco dei suggerimenti. Per gli elementi dinamici è stato sfruttato il modo *WDL* ed in particolare le sue

informazioni riguardo al contesto. All'interno dei modi infatti viene utilizzato uno stack in cui inserire il contesto alla creazione di nuovi blocchi di codice, quelli delimitati dalle parentesi graffe. Eventuali variabili dichiarate in questi blocchi, infatti, hanno un visibilità limitata al blocco stesso, per questo bisogna evitare di dare suggerimenti che possona indurre in errore il programmatore. Allo scopo è definito nello stato dei modi un campo *context* in cui è contenuto lo stack, oltre alle operazioni *pushContext* e *popContext*, mostrati di seguito.

```
...
startState: function(basecolumn) {
      return {
        tokenize: null,
        context: new Context((basecolumn || 0) - indentUnit, 0, "top", false,
parserConfig.localVars),
        indented: 0,
        startOfLine: true,
        onStyleBlock: false,
        onVarDeclaration:false
      };
    }
....
function Context(indented, column, type, align, prev, vars) {
    this.indented = indented;
    this.column = column;
    this.type = type;
    this.align = align;
    this.prev = prev;
    this.vars = vars;
  }

  function pushContext(state, col, type) {
    return state.context = new Context(
            state.indented, col, type, null, state.context, state.context.vars);
  }

  function popContext(state, stream, keepVariables) {
     var vars=state.context.vars;
    var t = state.context.type;
    if (t == ")" || t == "]" || t == "}")
      state.indented = state.context.indented;
    state.context = state.context.prev;
    if(keepVariables) state.context.vars=vars;
    return state.context;
  }
```

Gli oggetti memorizzati sullo stack contengono informazioni riguardo l'indentazione, il tipo di contesto, l'allineamento, l'elenco delle variabili e un puntatore al contesto successivo nello stack.

Oltre alla gestione del contesto nel modo viene effettuato anche il riconoscimento della dichiarazione di una variabile che viene opportunamente memorizzata nel contesto attuale grazie alla funzione *register* mostrata in Figura 30. Alla creazione di un nuovo blocco di codice, le

variabili nel contesto esterno sono ancora visibili, per questo motivo vengono copiate nel nuovo contesto.

```
function register(varname, state, overwrite) {
        for (var v=state.context.vars; v; v = v.next){
            if (v.name == varname) return;
        }
        if(overwrite){
              state.context.vars = {name: varname, next:
state.context.vars.next};
        }else{
              state.context.vars = {name: varname, next: state.context.vars};
        }
  }
```

Figura 30: Funzione *register* per la memorizzazione di variabili nel contesto

A questo punto, dopo aver visto come le variabili vengono riconosciute e memorizzate correttamente nel contesto del modo, rimane da analizzare come queste informazioni vengono utilizzate dalla funzione *i4CHint* per decidere o meno di inserirle nei suggerimenti. Il problema infatti è che, nel contesto, non è riportato il numero di linea dove la variabile è dichiarata, e, per evitare di suggerire una variabile prima della sua dichiarazione, è stato impiegato un'algoritmo che naviga il sorgente per capire se le variabili memorizzate nel contesto sono state dichiarate prima o dopo la posizione corrente del cursore. Ecco come è stato implementato l'algoritmo:

```
function isVariableDeclaredBeforeCursor(variable,editor){
     var e=editor, cursorLineInfo=
                          e.lineInfo(e.getCursor(true).line),
                          lineCounter=0;
     while(lineCounter<=cursorLineInfo.line){
            var line = e.getLine(lineCounter);
            var keywordPos=line.search(/var\b/i);
            if(keywordPos>0){
                  var varPos=line.indexOf(variable.toString());
                  var varLength=variable.toString().length;
                  var nearChar=line.charAt(varPos+varLength);
                  if(varPos>=0 && nearChar.match(/[\s;=\t]/)){
                        for(var i=varPos-1;i>keywordPos+3;i--){
                              if(cuttedLine.charAt(i)==" ") {
                                    return false;
                              }
                        }
                        return true;
                  }
            }
            lineCounter++;
     }
     return false;
    }
```

5.4.5 Implementazione del folding

Implementare questa funzionalità si è rivelato, in realtà, essere un semplice problema di configurazione dato che CodeMirror, tra le classi di utilità, presenta già tutti gli strumenti per farlo. In particolare ci si sta riferendo al file *foldcode.js* in cui è definita la funzione *newFoldFunction* dichiarata in fase di configurazione (Figura 27) per l'evento *onGutterClick*. Il gutter è una colonna di CodeMirror in cui è possibile inserire dei simboli e che scatena eventi a seguito del click dell'utente. Al verificarsi dell'evento CodeMirror attiva la funzione associata (*newFoldFunction*), che, a sua volta, richiama l'algoritmo per l'individuazione delle parentesi graffe, denominato *braceRangeFinder*. Se l'algoritmo trova una parentesi graffa aperta sulla stessa linea dove è avvenuto il click ed una chiusa in linee successive e allo stesso livello di indentazione, allora il codice viene nascosto ed un simbolo appare sul gutter. Un nuovo click sull'icona del gutter provocherà l'unfolding e la sua scomparsa.

Un problema incontrato nell'implementazione di questa funzionalità era collegato allo sviluppo della funzionalità di salvataggio. La pressione del pulsante infatti provocava l'aggiornamento dell'intera pagina causando l'unfolding dei blocchi. Il problema è stato risolto utilizzando un tag Ajax invece Html, in modo da effettuare il rendering soltanto dell'area di notifica.

5.4.6 Implementazione comment/uncomment

Questa funzionalità, semplice ma di estrema utilità, è stata implementata grazie alle funzioni mostrate in Figura 31.

```
function commentSelection() {
     innerCommentSelection(true);
}
function unCommentSelection() {
     innerCommentSelection(false);
}
function getSelectedRange() {
     return {
            from : weCodeMirror.getCursor(true),
            to : weCodeMirror.getCursor(false)
     };
}

function innerCommentSelection(isComment) {
     var range = getSelectedRange();
     weCodeMirror.commentRangeWF(isComment, range.from, range.to);
}

// Comment/uncomment the specified range with '#' at the beginning of line
CodeMirror.defineExtension("commentRangeWF", function(isComment, from, to) {
     if (isComment) { // Comment range
```

```
            var i;
            for (i = from.line; i <= to.line; i++)
                  this.setLine(i, "#" + this.getLine(i));
      } else { // Uncomment range
            var i;
            for (i = from.line; i <= to.line; i++)
                  if (this.getLine(i)[0] == '#')
                        this.setLine(i, this.getLine(i).substring(1));
      }
});
```

Figura 31: Implementazione Comment/Uncomment

Si tratta di quattro funzioni ed una estensione delle API di CodeMirror:

- *commentSelection*, associata in fase di configurazione alla shortcut *Ctrl+H*, richiama *innerCommentSelection*;
- *unCommentSelection*, associata in fase di configurazione alla shortcut *Ctrl+Shift+H*, richiama *innerCommentSelection*;
- *getSelectedRange*, determina l'inizio e la fine della selezione del testo in CodeMirror;
- *innerCommentSelection,* recupera le informazioni sulla selezione ed attiva l'algoritmo vero e proprio con le informazioni a sua disposizione;
- *commentRangeWF*, un'estensione di CodeMirror che aggiunge, o rimuove il simbolo di commento (#) in base alla shortcut attivata. E' la parte core della funzionalità.

5.4.7 Implementazione autosave

L'implementazione di queste funzionalità, un volta abilitata tramite la shortcut *Ctrl+Shift+A*, si basa sulla funzione JavaScript *setInterval* che permette di eseguire una funzione (primo parametro) ogni periodo di tempo specificato in millisecondi come secondo parametro. All'attivazione e alla disattivazione, inoltre, viene mostrato un messaggio all'utente nell'area di notifica per confermargli il successo dell'operazione. Ecco il codice:

```
function toggleAutoSave() {
    var saveBtn = document.getElementById('createWizardForm:saveButton');
    var off_str = "Autosave: OFF", on_str = "Autosave: ON (every 30s)";

    if(!weCodeMirror.autosaveInterval){
        mySave();
        weCodeMirror.autosaveInterval = setInterval(mySave,30000);
        saveBtn.title = on_str;
        weCodeMirror.autosavePreviousColor = saveBtn.style.color;
```

```
        saveBtn.style.color = "gray";
    } else {
        clearInterval(weCodeMirror.autosaveInterval);
        weCodeMirror.autosaveInterval = null;
        saveBtn.title = off_str;
        saveBtn.style.color = weCodeMirror.autosavePreviousColor;
    }
}
```

Il salvataggio tramite shortcut (*Ctrl-Shift-S*) viene effettuato simulando un click sul bottone di salvataggio. L'evento scatena un chiamata ai metodi del *WizardEditorMB* il quale si occupa dell'esecuzione.

```
function fastSave() {
    weCodeMirror.save();
     document.getElementById('createWizardForm:saveButton').click();
}
```

5.4.8 Syntax checking

Questa funzionalità, in realtà, viene implementata da JavaCC ed, in particolare, dal parser che è stato generato per il linguaggio WDL. Rendere disponibile questa feature all'utente, consite, come visto anche in altri casi, nel collegare la pressione dell'apposito pulsante o shortcut ad un metodo del *WizardEditorMB* incaricato di chiamare i componenti dedicati o di eseguire lui stesso la funzionalità richiesta. Non è nelle intenzioni ripresentare il pattern già visto ma piuttosto si vuole focalizzare l'attenzione sull'efficienza del controllo effettuato. Il parser, infatti, interrompe la sua esecuzione al primo errore rilevato invece di collezionarli tutti, inoltre in molti casi l'errore non viene nemmeno rilevato nella posizione esatta. Effettuando il debug delle chiamate, ci si è accorti che il problema dell'imprecisione non è legato prettamente al parser ma piuttosto alla definizione della specifica BNF del linguaggio. Si è infatti scoperto che la causa è l'uso eccessivo dei lookahead semantici. Quando il parser raggiunge un punto di ambiguità del linguaggio i lookahead venivano effettuati per intere porzioni dell'albero sintattico. Un errore in fase di lookahead non provocava l'avanzamento dello stream di token che rimaneva quindi fermo al nodo in cui il lookahead era stato chiamato. L'errore era quindi segnalato molto prima della locazione effettiva. L'unica soluzione è quella di limitare il più possibile l'uso di questi predicati soltanto alle porzioni di linguaggio ambigue.

Una tra le imprecisioni più importanti riguardava gli errori nelle funzioni builtin che, segnalati sul nome della builtin erano in realtà situati nella dichiarazione dei parametri. Di seguito ne viene riportato un esempio:

```
SHOWMESSAGE(                         ←segnalazione
                    ["key" : "E-mail: ",
                       "id" : "mymap",]);     ← errore
```

Un errore del tutto simile avviene con le chiamate a formule di libreria:

```
libreria.myfunc(                   ←segnalazione
             ["saluto" : "ciao",
                           5]);     ← errore
```

Gli esempi presentati sono molto semplici; utilizzano soltanto pochi parametri nelle mappe. In chiamate con più parametri il gap tra locazione corretta e segnalazione del parser aumenta molto e questo rende il controllo ancora più impreciso.

La causa della maggior parte delle imprecisioni è il nodo *PrimaryExpression* mostrato in precedenza, per la precisione in Figura 12. E' un punto della sintassi delicato perchè particolarmente ambiguo, come già spiegato nel Paragrafo 3.3.3, perciò la correttezza delle modifiche effettuate è stata verificata a fondo. Le modifiche consitono, come già accennato, nella riduzione della porzione di albero affetta dal lookahead e sono mostrate qui di seguito:

```
void PrimaryExpression() #void :
{}
{
  ( Literal()
  | < LP > Expression() < RP >
  | LOOKAHEAD(< IDENTIFIER > < LSB > Expression() < RSB >)
    ( < IDENTIFIER > { jjtThis.setName(token.image); } < LSB > Expression() < RSB > )
                                                                  #ArrayOrMapAccessor
  | LOOKAHEAD(< IDENTIFIER > < DOT > < IDENTIFIER > <LP>)
    ( < IDENTIFIER > { jjtThis.setName(token.image); } < DOT > < IDENTIFIER >
                   { jjtThis.setComponent(token.image); } Arguments() ) #FormulaInvocation
  | LOOKAHEAD(< IDENTIFIER >  <LP>)
    ( < IDENTIFIER > { jjtThis.setName(token.image);
                       jjtThis.setPosition(token.beginLine, token.beginColumn);
                     } Arguments() ) #FunctionInvocation
  | Identifier()
  )
}
```

I lookahead interessati dalle modifiche sono per l'invocazione delle formule di libreria e per l'invocazione delle funzioni. Grazie a queste modifiche gli errori dei precedenti esempi ed altri simili a quelli, vengono rilevati nell'esatta posizione o al massimo al carattere successivo.

5.4.9 Implementazione Go To Line, Find, Replace, Find Previous, Find Next, Match Bracket

Queste funzionalità sono già presenti in CodeMirror, per cui è stato sufficiente configurare le opzioni dell'editor correttamente per ottenerle. Si veda la Figura 27.

5.4.10 Esecuzione e debugging

L'esecuzione di un Wizard è strettamente connesso all'istanza per cui è stato lanciato che ne definisce il contesto applicativo. Questo è stato il principale motivo per cui questa funzionalità non è stata implementata nell'IDE dando la precedenza ad altre features per questioni di tempo. Ciò non significa ovviamente che non sia possibile eseguire Wizard sull'applicativo. Non è possibile farlo dalla pagina dell'editor in quanto è assente il contesto applicativo, ma l'esecuzione è effettuabile dalle pagine relative alle istanze. Questo ha fatto in modo che il problema passasse in secondo piano dato che per il programmatore sarebbe stato sufficiente mantenere aperte due finestre separate del proprio browser.

Implementare un debugger invece comporta la risoluzione di maggiori problemi, non solo quello della mancanza di contesto. Va data anche la possibilità di inserire breakpoints nell'editor, durante l'interpretazione il visitor deve poter controllare se ha raggiunto uno dei breakpoints, e soprattuto bisogna dotare la pagina JSP di componenti grafici adatti a visualizzare lo stato della tabella dei simboli una volta che l'esecuzione si interrompe. Questo come minimo, perchè solitamente un debugger offre anche i controlli per avanzare nel flusso di istruzioni. Questa features rappresenta l'unico punto di una certa priorità rimasto irrisolto nel progetto di questa tesi.

6 Soluzione basata su Xtext

Lo sviluppo delle funzionalità ad un basso livello su CodeMirror, come si sarà notato, è un percorso lungo e difficoltoso che obbliga ad implementare una ad una tutte le funzionalità di cui si ha bisogno. La disponibilità di IDE in rete è ampia, ma sono pochi quelli che in grado di supportare anche linguaggi custom come WDL. La nascita dei cosiddetti language workbench ha semplificato moltissimo la creazione di DSL e di tool ritagliati su misura del linguaggio. Xtext è uno dei più avanzati strumenti di questi tipo; è un vero e proprio framework che integra componenti per la definizione di tutte le principali fasi per la creazione di un linguaggio, dal parsing, al linking, all'interpretazione, e delle funzionalità utili all'utente che andrà ad utilizzare il linguaggio. L'utente finale, il programmatore, avrà infatti a disposizione la piattaforma Eclipse ed un editor guidato dalla sintassi del DSL integrato nell'IDE.

Xtext, inoltre, gioca un ruolo chiave nello sviluppo di applicazioni con un approccio model-driven grazie all'integrazione con la tecnologia Xtend per la generazione di codice Java. E' proprio su questi due componenti che si fonda la soluzione presentata in Figura 32.

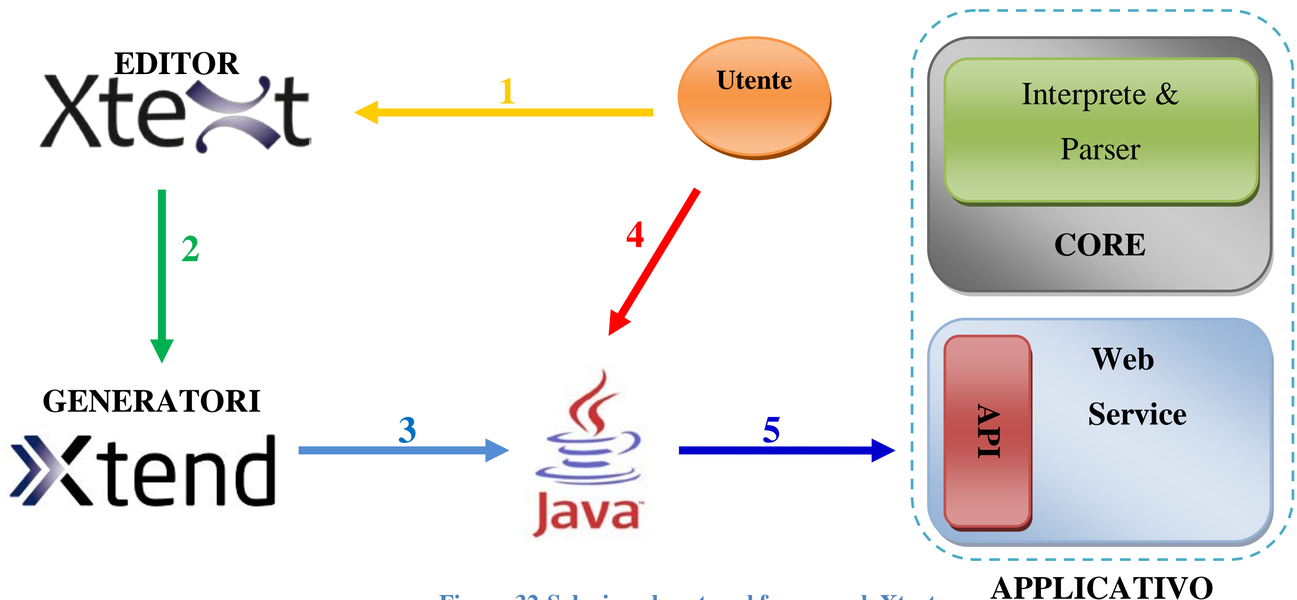


Figura 32:Soluzione basata sul framework Xtext

Nello schema viene infatti mostrato come interagiscono i diversi componenti del sistema per rendere fruibili all'utente le funzionalità core per lo sviluppo del linguaggio, cioè quelle del parser e dell'interprete che sono vincolati all'applicativo.

In quest'ottica una sessione di sviluppo del programmatore si svolge nei passi mostrati nello schema:

1. Il programmatore utilizza l'editor guidato dalla sintassi WDL supportato da tutti gli strumenti e i plugin di Eclipse;
2. Al salvataggio del codice da parte dell'utente, Xtext attiva i generatori Xtend;
3. I generatori producono dei sorgenti Java contenti il codice per effettuare le operazioni remote, come l'installazione sull'applicativo del wizard, il controllo sintattico, ed , in teoria, l'esecuzione;
4. L'utente esegue una delle applicazioni generate in base alla funzionalità di cui ha bisogno;
5. L'applicazione, tramite le API esposte dal Web Service, ha accesso ai servizi offerti dall'interprete e dal parser.

Lo sviluppo di questa soluzione è stato svolto in due fasi indipendenti: la definizione della sintassi EBNF di WDL in Xtext 2.3 e l'implementazione del infrastruttura client-server per l'accesso all'applicativo.

6.1.1 Definizione della sintassi EBNF in Xtext

Il punto di partenza, negli approcci moderni, per lo sviluppo di un linguaggio è la definizione della sua sintassi EBNF, la quale verrà utilizzata da un parser generator che produrra il riconoscitore delle frasi o parser. In Xtext queste operazioni vengono svolte su Eclipse grazie al supporto di un'editor dedicato e al parser generator ANTLR che assieme a JavaCC è uno dei più completi strumenti di questo tipo in circolazione.

Dato che si aveva già a disposizione la sintassi del linguaggio WDL in JavaCC, quello che si è dovuto effettivamente fare in questa fase è stato effettuare la traduzione nella sintassi EBNF di Xtext, che è, in realtà, leggermente differente da quella di ANTLR, ma molto più pulita ripetto a quella di JavaCC. A dimostrazione di questa affermazione si confronti la Figura 11 con quella seguente.

```
Wizard:
	{Wizard}
	(step+=SimpleStep|step+=ForEachStep)+
;

ForEachStep:
	{ForEachStep}
	'foreachstep' '(' arg=PrimaryExpression ')'
	'{'
		('init'':'
			'{'
				(stmt+=Statement)*
			'}')?
			wizard=Wizard
```

```
	'}'
;

SimpleStep:
	{SimpleStep}
	(=> step=InputStep| step=CustomStep)
;

InputStep:
	{InputStep}
	'step' '(' type=STRING ',' args+=STRING (',' exp=Expression)? ')'
	('disabled' 'previous'? 'cancel'? )?
	'{' (input_step_body+=Statement)*
		('on_refresh'':' in_refresh=Statement)?
		('validation'':'in_validation=Statement)?
		('on_next'
			(':' in_onNextPrev+=Statement |'asynchronous'':'
in_onAsync=Statement)
		)?
		('on_previous'':'in_onNextPrev+=Statement)?
		('style'':''{'in_style=Statement?'}')?
		('layout'':' in_layout=Statement)?
	'}'
;
```

Nonostante gli operatori di Kleene (Appendice A) e la capacità espressiva siano gli stessi, la sintassi risulta più concisa e questo fattore è molto importante dato che essa documenta come il linguaggio è costruito. Il motivo fondamentale è la riduzione delle action, cioè di foreign code nella sintassi, e l'adozione del simbolo => per i predicati semantici al posto di chiamate a funzioni per il lookahead.

Molte delle action per la memorizzazione della posizione dei token nel testo in Xtext non sono più necessarie perchè già implementate in background. Inoltre eventuali operazioni connesse con la Tree Construction (Appendice J) vengono effettuate elaborando i nodi assegnati alle variabili tramite Foreign Code. A questo scopo in Xtext esistono tre diversi operatori:

- = usato per assegnare un singolo nodo alla variabile;
- += utilizzato in punti ricorsivi, aggiunge il nodo ad una lista;
- ?= utilizzato per assegnamenti di valori booleani;

L'uso di questi operatori, in realtà, è stato fatto perchè richiesto esplicitamente dal tool e non per implementare effettivamente le tecniche di Tree Construction.

La traduzione, perciò, è stato un processo lineare che non ha richiesto particolari sforzi dato che il mapping, tra una sintassi e l'altra, era uno a uno. La parte che ha richiesto un maggiore impegno è stata quella di testing delle frasi che, data la complessità del linguaggio, ha

richiesto di verificare una regola per volta, controllando che le frasi ammesse e quelle non ammesse siano effettivamente le stesse.

Al termine di questa operazione è stato eseguito un programma MWE2 (Modeling Workflow Engine 2), che genera il parser, le classi per l'interpretazione, quelle per la generazione del codice, quelle che implementano l'editor, insomma tutta l'infrastruttura necessaria allo sviluppo e all'esecuzione del codice WDL. Questo programma è sviluppato anch'esso in un linguaggio DSL, come per l'EBNF e Xtend, ed il suo scopo è quello di definire un flusso di operazioni e di poterle configurare. La configurazione in questo caso è stata lasciata pressochè invariata, se non per la modifica del componente ANTLR che non permetteva l'opzione *ignoreCase*. Di seguito viene mostrata la variazione:

```
fragment = org.eclipse.xtext.generator.parser.antlr.ex.rt.AntlrGeneratorFragment{
                options = {
                    backtrack = false
                    ignoreCase = true
                }
 }
```

Alla fine di questa fase è possibile eseguire il progetto Xtext per toccare con mano i componenti generati. La versione dell'IDE che si è ottenuta è ricca di funzionalità:

- Autocompletamento non contestualizzato per le parole chiave;
- Syntax highlighting per commenti, parole chiave e stringhe;
- Tutte le principali funzionalità di Eclipse come la ricerca, l'annullamento delle modifiche, il salvataggio, il go-to-line, commento/decommento, find usage, find all ecc;
- Folding del codice;
- Indentazione automatica;
- Controllo sintattico per tutti gli errori.

Molte altre funzionalità sarebbero disponibili raffinando ulteriormente la sintassi EBNF: i riferimenti incrociati, la finestra di outline, quick fix e molte altre. Allo stesso modo si possono migliorare quelle già implementate, ad esempio, effettuando l'highlight per il Foreign Code CSS e per le builtin oppure estendendo l'autocompletamento alle variabili e facendo in modo che venga contestualizzato.

6.1.2 Infrastruttura client-server per l'accesso all'applicativo

In questa fase si è individuato un sistema per garantire l'interazione dell'IDE con l'applicativo sfruttando i generatori di codice Xtend integrati con Xtext. In pratica è stato prima

realizzato il codice che gestisce l'interazione, poi questo codice è stato inserito in un generatore Xtend perchè venga fornito all'utente durante lo sviluppo.

La natura di questa applicazione può variare notevolmente in base alla funzionalità che deve implementare e all'interfaccia che il Web Server espone, le API. Allo scopo di verificare l'architettura di questa soluzione è stato implementato un prototipo in Java che realizza l'installazione del Wizard sull'applicativo senza appoggiarsi ad un vero WebService, ma riusando la pagina JSP dell'editor della soluzione precedente. All'esecuzione infatti viene attivato il browser passandogli l'indirizzo Web della pagina in cui si trova CodeMirror. Il codice del Wizard viene passato come parametro all'indirizzo e, grazie ad una piccola modifica nella pagina JSP, viene automaticamente caricato nell'editor. L'utente a questo punto deve soltanto effettuare il salvataggio per completare l'installazione.

Il codice Java dell'applicazione è il seguente:

```
public class Install{

    public static void main(String [] args) {

        if( !Desktop.isDesktopSupported() ) {
            System.err.println( "Desktop is not supported (fatal)" );
            System.exit( 1 );
        }

       Desktop desktop = Desktop.getDesktop();
       if( !desktop.isSupported(Desktop.Action.BROWSE ) ) {
            System.err.println( "Desktop doesn't support the browse action (fatal)" );
            System.exit( 1 );
        }
              try {
                      URI uri = new
URI("http://localhost:18080/i4C/wizardeditor/createWizard.jsf?code=%22My%20wizard%20code%22");
                      desktop.browse(uri);
              } catch (Exception e) {

                      System.err.println(e.getMessage());
              }
      }

}
```

Dal codice si può notare come il sorgente del wizard viene passato nei parametri dell'indirizzo URL assegnando il testo alla variabile code. Nell'esempio è stata utilizzata la stringa di test "My Wizard Code".

La modifica effettuata nella pagina JSP è stata veramente minimale:

```
var code=#{param.code};
     document.getElementById('createWizardForm:Wizard_SourceCode').setValue(code
);
```

Poco prima di istanziare l'editor viene memorizzato il contenuto del parametro code in una variabile omonima, che poi viene usata per settare il contenuto del form. In questo modo il l'editor andrà a caricare il codice del Wizard.

Al fine di poter generare automaticamente il codice al salvataggio dell'utente, il sorgente è stato inserito all'interno del file WDLGenerator, il punto di accesso per la generazione del codice, l'equivalente del main in Java. Questo è il contenuto della classe:

```
override void doGenerate(Resource resource, IFileSystemAccess fsa) {
           generate(resource,  fsa);
     }
     def void generate(Resource resource, IFileSystemAccess fsa) {
                 fsa.generateFile(JavaFileName, JavaFileContent)
     }
     def String JavaFileName() {
                 return "i4C\\wizardInstallation\\Install.java"
     }
     def String JavaFileContent()'''
         package i4C.wizardInstallation;

         import java.net.URI;
         import java.awt.Desktop;

         public class Install{

         public static void main(String [] args) {
              ...
       '''
```

L'usabilità di questa soluzione, allo stato attuale, è scarsa, l'utente dovrebbe effettuare il minor numero possibile di operazioni superflue ed interagire con il browser è proprio una di queste. Una usabilità migliore si otterrebbe implementando un Web Service che esponga delle API per accedere all'applicativo e realizzando un'applicazione che chiami queste API eliminando così il bisogno di usare il browser. La difficoltà principale è la definizione dell'interfaccia: si potrebbero infatti adottare le stesse API del WizardEditorMB, lasciando il parser e l'interprete all'interno dell'applicativo ma i benefici di Xtext non sarebbero massimizzati, in quanto questo escluderebbe a priori la realizzazione di un debugger. Oppure si potrebbe implementare l'interprete su Xtext, e sfruttando il parser generato con ANTLR, si avrebbe anche la possibilità di eseguire il Wizard da Eclipse, ma purtroppo non è stato approfondito come questo possa essere correttamente visualizzato.

7 Conclusioni

La disponibilità di strumenti di sviluppo potenti ed integrati è da sempre un aspetto fondamentale di un processo di sviluppo del software agile ed efficace. Il bisogno di questo tipo di soluzioni è in crescita nel panorama IT, di pari passo con l'aumento nella diffusione dei Domain Specific Language.

Il lavoro di questa tesi, incentrato sul Wizard Definition Language di i4C Analytics, ha voluto creare un solido background teorico su questo DSL ed di pari passo sviluppare strumenti avanzati e specializzati che facilitassero la produzione di codice di qualità. A questo riguardo, le soluzioni esplorate hanno volutamente adottato approcci radicalmente differenti, in modo da far emergere chiaramente vantaggi e svantaggi di ciascuno.

La prima soluzione è stata quella di integrare un editor Codemirror nel framework i4C, semplificandone l'adozione da parte degli utenti business ma complicando lo sviluppo delle singole feature dell'IDE, che sono state implementate una per volta, dopo accurata raccolta requisiti e design - per fare qualche esempio, syntax highlighting, code assist e folding hanno richiesto una discreta quantità di lavoro. Ovviamente, la complessità di implementare ulteriori funzionalità avanzate come debugging o compilazione rendono oneroso il proseguimento su questa strada.

La seconda soluzione, basata sul framework Xtext, presenta un editor di primissimo livello come Eclipse ma rende più difficile l'adozione dello strumento ai Domain Expert e più complesso il deploy istantaneo in ambienti cliente, mancando completamente l'integrazione con il framework i4C installato presso i clienti. Un'interazione tramite WebService con l'applicativo (per il deploy rapido dei Wizard) completerebbe questa soluzione permettendo l'aggiunta di ulteriori funzionalità avanzatissime a costo/complessità molto inferiori rispetto alla prima soluzione.

Confrontando le due soluzioni, si intuisce come la bilancia penda indubbiamente dalla parte di Xtext, non solo perchè le funzionalità dell’editor sono già implementate ma piuttosto perchè mette a disposizione strumenti che vanno oltre la semplice programmazione, come, ad esempio, il versionamento dei sorgenti e tutto ciò che è implementato come plugin Eclipse.

Questa soluzione, che, ripeto, è ancora ad uno stadio prototipale, già dimostra come l’uso dei generatori di codice possa aggirare abilmente le problematiche di integrazione con l’architettura distribuita.

Quello che si vuole sottolineare e che, se si fosse optato fin dall'inizio per Xtext, i benefici sarebbero stati maggiori e sarebbero stati raggiunti in tempi più brevi. Quest'affermazione trova maggiori conferme quando il linguaggio viene sviluppato fin da subito con questi tool. L'azienda, però, aveva già realizzato i due componenti chiave per il linguaggio WDL: il parser e l'interprete. In fasi così avanzate, adottare la soluzione Xtext, per sfruttarne le piene potenzialità, avrebbe comportato la sostituzione del parser e l'adattamento dell'interprete, buttando così parte degli sforzi fatti precendentemente.

Questi motivi portano a pensare che, nello sviluppo di un nuovo linguaggio, sia sempre meglio considerare, fin dalle prime fasi di analisi e progettazione, anche la realizzazione di strumenti a supporto del programmatore. In tal senso l'approccio di Xtext è proprio quello di unire le due cose.

Uno dei vantaggi della soluzione CodeMirror-based, invece, è che il progetto è potuto procedere in maniera incrementale. Fin da subito è stato dato un'IDE funzionale in mano ai programmatori e, man mano che nuove funzionalità venivano sviluppate, questi ultimi ne potevano disporre immediatamente. Questo fattore non è da sottovalutare in progetti sviluppati all'interno di aziende. Inoltre, la possibilità di eseguire senza troppe complicazioni i Wizard è grosso punto a favore rispetto alla soluzione Xtext in cui questa features non è ancora ben chiaro come realizzarla.

7.1 Sviluppi futuri

Entrambe le soluzioni presentano ampi margini di miglioramento.

Per quanto riguarda la prima soluzione, si può sia migliorare il livello di integrazione dell'IDE permettendo l'esecuzione dalla pagina dell'editor, che aggiungere i requisiti mancanti per le features. Tra queste ultime sono presenti, in particolare, la possibilità di effettuare il debugging, la disponibilità rapida della documentazione e delle implementazioni per le builtin e le formule di libreria, la find usage e la find all, un tool grafico per la creazione del layout nei Wizard ed il controllo semantico, che si potrebbe realizzare semplicemente affiancando al parser una symbol table (Appendice I). Le features già presenti, inoltre, sono ancora migliorabili; si potrebbe utilizzare l'oggetto Notification (Appendice G) per raccogliere tutti gli errori individuati durante il syntax checking, si potrebbe estendere il syntax highlighting anche alle formule ed effettuare il discovery automatico delle parole del linguaggio dalla BNF, oppure si potrebbe ottimizzare l'algoritmo per i suggerimenti delle variabili nell'autocompletamento.

La soluzione basata su Xtext, essendo ancora in fase prototipale, richiede ancora molti sviluppi ed un'analisi più approfondita su come affrontare l'esecuzione di un Wizard da Eclipse.

L'introduzione di un WebService che permette l'installazione del Wizard sull'applicativo renderebbe sicuramente questa soluzione più usabile portandola ad uno stato che nulla ha da invidiare a quella con CodeMirror.

Uno studio approfondito delle ultime versioni di Xtext, inoltre, potrebbe aiutare a capire se esistono meccanismi che aiutino ad integrare anche l'esecuzione dei Wizard e di conseguenza il debugger, che sono i punti chiave rimasti irrisolti.

Anche per questa soluzione si possono anche intraprendere sviluppi per il miglioramento delle funzionalità esistenti come il syntax highlighting, abilitandolo a riconoscere Foreign Code CSS, e l'autocompletamento, estendendolo a più parole possibili.

Tante sono le funzionalità ancora attivabili: i riferimenti incrociati, la finestra di outline, quick fix, e molte altre che saranno disponibili nelle nuove versioni di Xtext. Non tutti i requisiti però potrebbero comunque essere soddisfatti, ci si sta riferendo, in particolare, al tool per la generazione del layout.

Uno sviluppo particolarmente interessante, ma che non riguarda direttamente l'IDE, è la reingegnerizzazione del linguaggio per la realizzazione di una versione 2.0 di WDL. Questo sviluppo dovrebbe individuare gli elementi del linguaggio che lo allontanano dalla definizione di DSL suggerita da Fowler, per fare in modo di godere a pieno dei benefici di tali linguaggi.

Appendici

A. BNF

Definisce formalmente la sintassi di un linguaggio di programmazione.

```
grammarDef : rule+ ;
rule : id ':' altList ';' ;
altList : element+ ( '|' element+ )* ;
element : id ebnfSuffix?  | '(' altList ')' ;
ebnfSuffix : '?' | '*' | '+' ;
id : 'a'..'z' ('a'..'z'|'A'..'Z'|'_'|'0'..'9')* ;
```

Come funziona

La BNF (ed EBNF) è una modalità per scrivere le grammatiche che definiscono la sintassi di un linguaggio. BNF (Backus-Naur Form) fu inventata per descrivere il linguaggio Algol negli anni 60. Da allora le grammatiche BNF sono state utilizzate sia per guidare la Syntax Directed Translation che per documentare chiaramente le frasi lecite del linguaggio.

Le BNF, nonostante siano un linguaggio per definire le sintassi, non hanno una sintassi standard. Ogni volta che si ha a che fare con una nuovo grammatica BNF si potrebbero incontrare sottili differenze da quelle già conosciute. Di conseguenza non è corretto affermare che le BNF sono un linguaggio ma piuttosto è meglio pensarle come una famiglia di linguaggi. In molti le associano più a dei pattern ed in effetti vi assomigliano molto.

Nonostante il fatto che sia la sintassi che la semantica delle BNF può variare molto di volta in volta, esistono numerosi elementi in comune. Il principale è la nozione per dare una descrizione del linguaggio come successione di regole di produzione. Si consideri, ad esempio, il seguente linguaggio:

```
contact mfowler {
        email: fowler@acm.org
}
```

Una grammatica per questo linguaggio potrebbe assomigliare a qualcosa del genere:

```
contact : 'contact' Identifier '{' 'email:' emailAddress '}';
emailAddress : localPart '@' domain ;
```

Ci sono due regole fondamentali di produzione, ciascuna regola presenta un nome ed un corpo. Il corpo descrive come si può decomporre la regola in una sequenza di elementi che possono essere altre regole oppure simboli terminali. Un simbolo terminale è qualcosa di

diverso da una regola, come letterali *contact* e *}*. Se si sta utilizzando una grammatica BNF con una Syntax Directed Translation, i simboli terminali saranno i tipi dei token generati dal lexer.

Come già discusso le BNF appaiono sotto diverse forme sintattiche. Quella dell'esempio è la stessa utilizzata dal Parser Generator ANTLR mentre qui sotto ne è stata riportata una versione più vicina a quella originale usata per Algol:

```
<contact> ::= contact <Identifier> { email: <emailAddress> }
<emailAddress> ::= <localPart> @ <domain>
```

In questo caso le regole sono definite tra parentesi angolari, mentre i letterali non lo sono, e poi finiscono con un terminatore di linea invece del punto e virgola. Il simbolo "::=", invece, viene impiegato come separatore tra il corpo e il nome delle regole. Insomma molti di questi elementi possono variare in BNF diverse, quindi è meglio non dilungarsi su questi aspetti. Per gli esempi successivi verrà sempre impiegato ANTLR in quanto i Parser Generator moderni utilizzano uno stile più simile ad esso.

Ora si estenderà il problema considerando che i contatti possono avere o un indirizzo e-mail oppure un numero di telefono. Quindi oltre all'esempio precedente, si potrebbe avere anche:

```
contact rparsons {
    tel: 312-373-1000
}
```

E' possibile estendere la grammatica per riconoscere queste nuove frasi utilizzando l'operatore di alternativa.

```
contact : 'contact' Identifier '{' line '}' ;
line : email | tel ;
email : 'email:' emailAddress ;
tel : 'tel:' TelephoneNumber ;
```

L' operatore di alternativa è il simbolo " | " all'interno delle regole. Indica che la regola si può decomporre o come *email* oppure come *tel.* Un'altra cosa utile è estrarre l'identificatore in un regola *username*.

```
contact : 'contact' username '{' line '}' ;
username : Identifier;
line : email | tel ;
email : 'email:' emailAddress ;
tel : 'tel:' TelephoneNumber ;
```

La regola *username* si risolve in un singolo identificatore, ma è utile mostrare il più chiaramente possibile le intenzioni nella grammatica, come se si stesse estraendo un semplice metodo in un codice imperativo.

L'uso delle alternative è abbastanza limitato in questo contesto; permette di avere una email o un numero di telefono. Come si vedrà, le alternative in realtà possiedono un' enorme potenza espressiva.

Multiplicity Symbols (Kleene Operators)

Un'applicazione seria per la gestione dei contatti non dovrebbe permettere solo un'email o un numero di telefono come punto di contatto. Non si vuole certo ottenere un gestore di contatti realistico ma il prossimo passo si muove in quella direzione. Un contatto deve quindi avere uno username, può avere un nome completo, deve avere almeno un indirizzo email e può avere qualche numero di telefono. Ecco una possibile grammatica:

```
contact : 'contact' username '{' fullname? email+ tel* '}';
username : Identifier;
fullname : QuotedString;
email : 'email:' emailAddress ;
tel : 'tel:' TelephoneNumber ;
```

E' facile riconoscere i simboli di molteplicità dato che sono gli stessi utilizzati nelle regular expression. L'uso dei simboli di molteplicità rende più semplice comprendere le grammatiche.

Quando si presentano i simboli di molteplicità, spesso sono presenti anche i costrutti per il raggruppamento che permettono di combinare diversi elementi a cui applicare una regola di molteplicità. E' possibile ridefinire la grammatica precendete, definendo le regole inline:

```
contact : 'contact' Identifier '{'
    QuotedString?
    ('email:' emailAddress)+
    ('tel:' TelephoneNumber)*
    '}'
 ;
```

In generale è meglio non definire le regole inline in modo da catturare meglio le intenzioni e rendere la grammatica molto più leggibile. Ma ci sono situazioni in cui definire nuove regole può creare confusione e l'operatore di raggruppamento funziona meglio.

Questo esempio mostra anche come sono formattate solitamente regole BNF più lunghe. La maggior parte delle BNF ignorano il terminatore di linea, quindi è possibile dividere la regola in parti logiche ed inserirle ciascuna in una propria linea, questo renderà una regola

complicata più chiara. In questo caso è solitamente più facile inserire il punto e virgola su una propria linea in modo da segnalare la fine della regola.

I simboli di molteplicità visti fin qui sono quelli che più comunemente si possono trovare, certamente per quanto riguarda i parser generator. Esiste però un'altra forma che utilizza le parentesi:

```
contact : 'contact' username '{' [fullname] email {email} {tel} '}';
username : Identifier;
fullname : QuotedString;
email : 'email:' emailAddress ;
tel : 'tel:' TelephoneNumber ;
```

Le parentesi sostituiscono " ? " con " [..] " e " * " con " {..} ". Non c'è una sostituzione per " + ", ma comunque si può sostituire *foo+* con *foo {foo}*. Questo stile con le parentesi è abbastanza comune in grammatiche volte all'uso da parte delle persone, ed è lo stile utilizzato nello standard ISO per le EBNF (ISO/IEC 14977); comunque la maggior parte dei Parser Generator preferiscono la forma delle regular expression ed quest'ultima che verrà adottata negli esempi.

Altri operatori utili

In ANTLR è possibile utilizzare l'operatore " ~ ". Questo operatore trova tutti i token fino all'elemento che segue il simbolo " ~ ". Quindi se si vogliono individuare tutti i caratteri fino alla parentesi graffa " } ", ma senza includerla, si può usare il pattern ~ ' } '. Se non si ha a disposizione questo operatore, l'equivalente nelle regular expression è [^ }] *.

La maggior parte degli approcci alla Syntax Directed Translation separano l'analisi lessicale da quella sintattica. Si può definire anche l'analisi lessicale nello stile delle regole di produzione, ma ci sono di solito sottili ma importanti differenze nel tipo di operatori e nelle combinazioni ammesse. Le regole lessicali tendono ad essere più simili ad una regular expression, perchè queste ultime usano una macchina a stati finiti, ciò di cui si ha bisogno in fase di analisi lessicale, al contrario della fase di parsing è rappresentata meglio da una macchina push-down.

Un importante operatore nell'analisi lessicale è l'operatore intervallo " .." utilizzato per identificare un intervallo di caratteri, come lettere minuscole 'a' .. 'z'. Una regola frequente per gli identificatori è:

```
Identifier:
   ('a'..'z' | 'A'..'Z')
   ('a'..'z' | 'A'..'Z' | '0'..'9' | '_')*
```

;

Ciò permette di iniziare un identificatore con una lettera maiuscola o minuscola, seguita da un lettera un numero oppure un underscore. Gli intervalli hanno senso soltanto nelle regole lessicali, non nelle regole sintattiche.

Tabella riassuntiva dei simboli

Simbolo	Significato	Esempio
|	alternativa	email | tel
*	nessuno o più (Kleene star)	tel*
+	uno o più (Kleene plus)	email+
?	opzionale	fullname?
~	fino a	~ ' } '
..	intervallo	'0' .. '9'
/	alternative ordinate	us_tel / raw_tel

Code actions

Le BNF forniscono un modo per definire la struttura sintattica di un linguaggio, e i Parser generator tipicamente usano le BNF per guidare le operazioni di un parser. Le BNF, però, non sono sufficienti. Forniscono abbastanza informazioni per generare l'albero di parsing, ma non sono abbastanza per generare i più utili abstract syntax tree, e neanche per realizzare l'Embedded Translation o l'Embedded Interpretation. L'approccio comune è quello di inserire delle azioni nelle BNF per fare in modo che il codice reagisca.

Non tutti i Parser Generator utilizzano le Code Actions. Un'altro approccio è fornire un separato DSL per realizzare la Tree Construction.

L'idea principale che sta dietro alle Code Action è inserire pezzi di Foreign Code in alcuni punti della grammatica. Queste parti sono eseguite quando quella parte della grammatica viene riconosciuta dal parser. Eccone un esempio:

```
contact : 'contact' username '{' email? tel? '}';
username: ID;
email : 'email:' EmailAddress {log("got email");};
tel : 'tel:' TelephoneNumber;
```

In questo caso, il messaggio viene registrato una volta che la clausola *email* viene riconosciuta nel parsing. E' utile registrare il fatto che un'e-mail è stata trovata, ma si vorrebbe fare la stessa cosa anche per l'indirizzo e-mail. Per fare questo, si ha bisogno del riferimento al token dell'indirizzo e-mail quando viene riconosciuto. I vari Parser Generator hanno modi diversi per farlo. Yacc si riferisce ai token con una variabile speciale che indicizza la posizione dell'elemento. Quindi ci si può riferire all'indirizzo e-mail con "$2" ("$1" si riferisce al token *email:*). I riferimenti posizionali sono fragili ai cambiamenti nella grammatica, quindi un approccio più comune nei moderni Parser Generator è quella di etichettare gli elementi. In ANTLR viene fatto in questo modo:

```
contact : 'contact' username '{' email? tel? '}';
username: ID;
email : 'email:' e=EmailAddress {log("got email " + $e.text);};
tel : 'tel:' TelephoneNumber;
```

In ANTLR, un riferimento a $e punta all'elemento etichettato con " e= " nella grammatica. Dato che l'elemento è un token, si usa l'attributo *text* per ottenere il testo individuato. (Si possono ottenere informazioni anche sul tipo di token, numero di linea, ecc.)

Allo scopo di risolvere questi riferimenti, i Parser Generator eseguono le Code Action sfruttando un templating system che rimpiazza le espressioni come " $e " con i valori corretti. ANTLR, di fatto, va oltre. Attributi come *text* non si riferiscono a campi o metodi direttamente; ANTLR effettua ulteriori sostituzioni per ottenere le giuste informazioni.

Esattamente come ci si riferisce ad un token ci si può riferire ad una regola.

```
contact :        'contact' username '{' e=email? tel? '}'
                 {log("email " + $e.text);}
                 ;
username : ID;
email : 'email:' EmailAddress ;
tel : 'tel:' TelephoneNumber;
```

Il log registrerà il testo completo riconosciuto dalla regola *email* ("*email: fowler@acm.org*"). Spesso, restituire degli oggetti del genere non è molto utile, particolarmente se si stanno riconoscendo regole più complesse. Di conseguenza, i Parser Generator di solito danno la possibilità di definire che cosa dev'essere ritornato dalle regole una volta individuate. In ANTLR, si può fare definendo un tipo ed una variabile per l'oggetto ritornato dalla regola.

```
contact :        'contact' username '{' e=email? tel? '}'
                        {log("email " + $e.result);}
                        ;
username : ID;
email returns [EmailAddress result]
                        :        'email:' e=EmailAddress
                                 {$result = new EmailAddress($e.text);}
                        ;
tel : 'tel:' TelephoneNumber;
```

Qualsiasi cosa può essere restituito da una regola per poi rifersi ad esso nella regola padre. (ANTLR permette di definire valori di ritorno multipli). Questo strumento, combinato con le Code Actions, è estremamente importante. Spesso, una regola che restituisce le migliori informazioni sul un valore non è la regola migliore per decidere che cosa fare con quel dato. Trasferire le informazioni nelle regole "padre", permette di catturare informazioni negli strati inferiori del parser e gestirle negli strati più alti. Se tutto ciò non ci fosse sarebbe necessario usare molte Context Variables, facendo diventare il tutto molto caotico.

Le Code Action possono essere utilizzate nelle tecniche Embedded Interpretation, Embedded Translation e Tree Construction. Lo stile particolare del codice nella Tree Construction, comunque, si presta ad un approccio diverso nel quale si usa un altro DSL per descrivere come costruire l'albero risultante.

La posizione di un'azione nella grammatica determina quando questa verrà eseguita. Quindi *parent : first { log ("hello");} second* implica che il metodo *log* verrà chiamato dopo la prima sotto regola ma prima della seconda. La maggior parte delle volte basterà semplicemente inserire le azioni alla fine di una regola, ma occasionalmente si può avere il bisogno di inserirle nel mezzo. Di volta in volta, la sequenza d'esecuzione delle azioni può essere difficile da capire o meno, dipende dall'algoritmo del parser. I Recursive Descent Parser sono solitamente abbastanza semplici da capire, mentre i parser bottom-up spesso creano confusione. Si può avere il bisogno di analizzare nel dettaglio il proprio sistema di parsing per capire esattamente quando le action vengono eseguite.

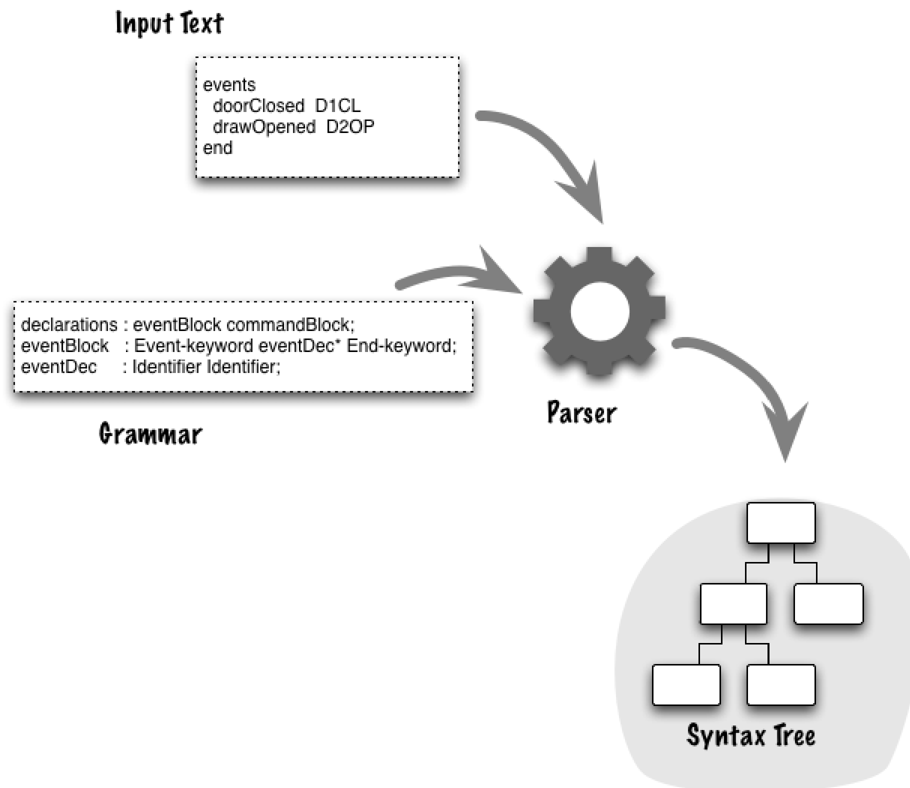
Il pericolo principale nell'usare le Code Action è che si può finire ad inserire molto codice in esse. Se accade, la grammatica diventa difficile da vedere, e si perdono i vantaggi che essa apporta, in particolare alla documentazione . Quando si utilizzano le code action è meglio adottare anche l'Embedment Helper.

Quando usarla

Ogni volta in cui si lavora con un Parser Generator si avrà bisogno di utilizzare una grammatica BNF, dato che questi tool sfruttano proprio la grammatica per generare il parser. E' anche molto utile come strumento per ragionare sulla struttura del proprio DSL o per comunicare le regole sintattiche del linguaggio ad altre persone.

B. Syntax Directed Translation

Traduce i sorgenti definendo una grammatica e utilizzandola per strutturare la traduzione.



I linguaggi computazionali tendono, per loro natura, a seguire una struttura gerarchica con livelli contestuali multipli. E' possibile definire una precisa sintassi di questi linguaggi scrivendo una grammatica che descrive come gli elementi del linguaggio si possono scomporre in sotto-elementi.

La Syntax Directed Translation utlizza qiuesta grammatica per creare il parser in modo che possa tradurre il testo in input in un parse tree che imita la struttura delle regole nella grammatica.

Come funziona

Molti libri sui linguaggi di programmazione affrontano il concetto di grammatica. Una grammatica è un modo per definire la sintassi corretta di un di un linguaggio di programmazione. Si consideri questo esempio:

```
events
  doorClosed D1CL
```

```
  drawerOpened D2OP
# ...
end
commands
  unlockPanel PNUL
  lockPanel PNLK
# ...
end
```

Queste dichiarazioni hanno una forma sintattica definita dalla seguente grammatica:

```
declarations : eventBlock commandBlock;
eventBlock : Event-keyword eventDec* End-keyword;
eventDec : Identifier Identifier;
commandBlock : Command-keyword commandDec* End-keyword;
commandDec : Identifier Identifier;
```

Una grammatica come questa fornisce una definizione del linguaggio leggibile dagli esseri umani. Le grammatiche sono, di solito, scritte in BNF. Una grammatica rende facile per le persone capire qual'è la sintassi accettata da un linguaggio. Con la Syntax Directed Translation è possibile portare oltre la grammatica per utilizzarla come base della progettazione del programma che elabora questo linguaggio.

Questa elaborazione si può intuire dalla grammatica in un paio di modi. Un primo approccio è quello di utilizzare la grammatica come sia come specifica che come guida per l'implementazione di parser fatti manualmente. Le tecniche Recursive Descent Parsing e Parser Combinator sono due approcci per implementare un parser. Una alternativa è quella di utilizzare la grammatica come DSL e sfruttare un Parser Generator per generare automaticamente il parser dalla grammatica stessa. In tal caso, non c'è bisogno di implementare da soli il core del parser visto che tutto viene generato dalla grammatica.

Nonostante sia utilissima, la grammatica risolve soltanto una parte del problema: può servire a trasformare il testo in input in una struttura dati come il parse tree ma, quasi sempre, si avrà bisogno di fare più di così. Per questo i Parser Generator forniscono anche strumenti per modificare il comportamento del parser in modo da poter, ad esempio, popolare il modello semantico. Perciò nonostante il Parser Generator svolge molto del lavoro, bisogna comunque programmare un po' per realizzare qualcosa di veramente utile. In tal senso, un Parser Generator è un eccellente esempio pratico dell'uso di un DSL. Non risolve l'intero problema ma rende molto semplice una buona parte di esso.

Il lexer

Quasi sempre, quando si usa la Syntax Directed Translation, si noterà che lexer e parser sono separati. Il lexer, anche chiamato tokenizer o scanner, è il primo stadio nel processamento del testo in input. Il lexer spezza i caratteri in input in token, che rappresentano una forma più ragionevole dell'input.

I token sono generalmente definiti utilizzando le espressioni regolari; ecco un esempio delle regole di lexing per l'esempio visto sopra:

```
event-keyword: 'events';
command-keyword: 'commands';
end-keyword: 'end';
identifier: [a-z A-Z 0-9]*;
```

Si consideri una parte più ridotta dell'input:

```
events
  doorClosed D1CL
  drawOpened D2OP
end
```

Le regole di lexing traduce quindi l'input in una serie di token:

```
[Event-keyword: "events"]
[Identifier: "doorClosed"]
[Identifier: "D1CL"]
[Identifier: "drawOpened"]
[Identifier:"D2OP"]
[End-keyword: "end"]
```

Ciascun token è essenzialmente un oggetto con due proprietà: il tipo ed il payload. Il tipo rappresenta il genere di token che si ha, ad esempio *Event-keyword* o *Identifier*. Il payload è il testo che è stato individuato dal lexer: *events* o *doorClosed*. Per le keyword il payload è irrilevante, tutto ciò che serve è il tipo, mentre per gli identificatori, il payload è ciò che conta, rappresenta il dato che sarà utile in fase di parsing.

La fase di lexing è separata per semplici ragioni. La prima è che rende il parser più semplice, in quanto può essere scritto ragionando in termini di token invece di meri caratteri. La seconda è l'efficienza: l'implementazione di cui si ha bisogno per raggruppare caratteri in token è differente da quella che si utilizzerebbe nel parser.(Nella teoria degli automi, il lexer è solitamente una macchina a stati mentre il parser è un macchina con stack push-down.) Questa

suddivisione è quindi un classico approccio che però viene messo alla prova da alcune moderne implementazioni. (ANTLR utilizza un macchina push-down per il lexer, mentre alcuni moderni parser combinano lexing e parsing nel parser senza utilizzare un lexer).

Le regole del lexer vengono testate una per volta, in ordine. Non si può utilizzare la stringa *events* come identifier perchè questa verrà sempre riconosciuta come come parola chiave. Generalmente viene vista come una buon modo per ridurre la confusione. Esistono, però, situazioni in cui si ha bisogno di evitarlo e perciò si adottano alcune forme di Alternative Tokenization.

Se si è stati opportunamente attenti confrontando i token con il testo in input, si sarà notato che manca qualcosa nella lista dei token, ciò che manca sono gli spazi bianchi: spazi, tab e terminatori di linea. In molti linguaggi il lexer rimuove gli spazi bianchi in modo che il parser non ne abbia a che fare. In questo c'è una grossa differenza con la Delimiter Directed Translation in cui gli spazi bianchi giocano un ruolo chiave.

Se gli spazi bianchi sono sintatticamente rilevanti il lexer non può semplicemente ignorarli. Dovrà invece generare un qualche tipo di token per indicare cosa sta succedendo, come il token newline per i terminatori di linea. Spesso, comunque, nei linguaggi in cui si intende utilizzare la Syntax Directed Translation si cerca di ignorare gli spazi bianchi. In molti DSL, infatti, si cerca di fare in questo modo.

Un'altra cosa che il lexer solitamente scarta sono i commenti. E' utile avere i commenti anche nel più ridotto dei DSL ed il lexer può liberarsene facilmente. Si potrebbe non voler scartare i commenti; potrebbero essere utili per fini di debug, particolarmente nel codice generato. In questo caso, bisogna pensare se si ha intenzione di allegarli agli elementi del modello semantico.

Si è detto che i token hanno una proprietà per il tipo e una per il payload. In pratica possono anche trasportare altre informazioni. Spesso queste informazioni sono utili per la diagnostica degli errori, come il numero della linea e la posizione del carattere.

Quando si decidono i token, c'è spesso la tentazione di raffinare il processo di riconoscimento. Nell' esempio, si può notare come i codici siano sequenze di quattro caratteri o di caratteri maiuscoli o di numeri. Si potrebbe quindi pensare di scrivere una specifica del genere:

```
code: [A-Z 0-9]{4}
```

Il problema è che così il tokenizer produrrà token sbagliati in casi del genere:

```
events
  FAIL FZ17
end
```

Con quell'input, *FAIL* verrà tokenizzato come codice invece che come identificatore, perchè il lexer controlla soltanto i caratteri, non il contesto generale dell'espressione. Questo tipo distinzione è meglio lasciarla al parser, visto che ha sufficienti informazioni per distinguere un nome da un codice. Ciò significa che la regola per controllo dei quattro caratteri andrà inserita successivamente nel parser. In generale è meglio lasciare il lexer il più semplice possibile.

Il più delle volte conviene pensare che il lexer ha a che fare con tre diversi tipi di caratteri:

- punteggiatura: parole chiave, operatori, o altri costrutti atti all'organizzazione (parentesi, separatori di istruzioni). Con la punteggiatura, il tipo di token è importante, ma il payload non lo è.
- testo del dominio: nomi di oggetti, valori letterali. Per questi, il tipo del token è solitamente di molto generico come ad esempio "number" o "identifier".
- ignorabili: cose che solitamente vengono scartate dal tokenizer, come ad esempio gli spazi bianchi e i commenti.

La maggior parte dei Parser Generator forniscono generatori per il lexer, utilizzando espressioni regolari come quelle viste sopra. In molti però preferiscono scrivere un proprio lexer in quanto abbastanza lineare da implementare utilizzando una Regex Table Lexer. Con i lexer scritti a mano si ha una maggiore flessibilità per interazioni complesse tra parser e lexer, che spesso può essere utile.

Una particolare interazione parser-lexer che può essere utile si verifica nel supportare modalità multiple nel lexer consentendo al parser di decidere quando cambiare modalità. Questo permette al parser di alterare il comportamento della tokenizzazione in certi punti del linguaggio, il che può essere d'aiuto con l'Alternative Tokenization.

Analizzatore sintattico

Una volta che si ha un flusso di token, la parte successiva della Syntax-Directed Translation si effettua il parser di se stessa. Il parser di

comportamento può essere diviso in due sezioni principali, dette analisi sintattica e azioni. L'analisi sintattica prende il flusso di token e lo dispone in un albero sintattico. Questo lavoro può essere derivato interamente dalla stessa grammatica e, in un generatore di parser, il codice viene generato automaticamente dallo strumento. Le azioni

prendono quell'albero di sintassi e con esso fanno qualcosa di più come, ad esempio, la compilazione di un modello semantico.

Le azioni non possono essere generate da una grammatica, e vengono solitamente eseguite mentre l'albero di analisi viene costruito. Di solito un generatore di parser combina la definizione della grammatica con un codice aggiuntivo per specificare le azioni. Spesso, queste azioni sono in un linguaggio di programmazione general-purpose, anche se alcune azioni possono essere espresse in DSLs aggiuntivi.

Se costruiamo un parser utilizzando solo la grammatica e quindi l'analisi sintattica, il risultato del parse da potrà essere un'esecuzione con successo o un fallimento. Questo ci dice se il testo di input corrisponde alla grammatica o no. Capita spesso che spesso ciò venga descritto come se il parser riconoscesse l'input.

Definendo questa grammatica:

```
declarations : eventBlock commandBlock;
eventBlock : Event-keyword eventDec* End-keyword;
eventDec : Identifier Identifier;
commandBlock : Command-keyword commandDec* End-keyword;
commandDec : Identifier Identifier;
```

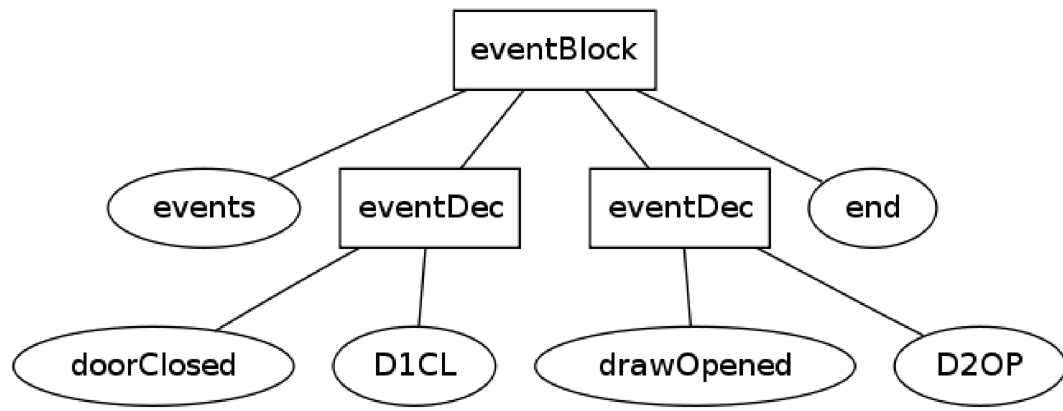
e questi input:

```
events
doorClosed D1CL
drawOpened D2OP
end
```

Si mostra come il tokenizzatore divida l'input in questo flusso di token:

```
[Event-keyword: "events"]
[Identifier: "doorClosed"]
[Identifier: "D1CL"]
[Identifier: "drawOpened"]
[Identifier:"D2OP"]
[End-keyword: "end"]
```

L'analisi sintattica poi prende questi token e la grammatica, e le dispone nella struttura ad albero:

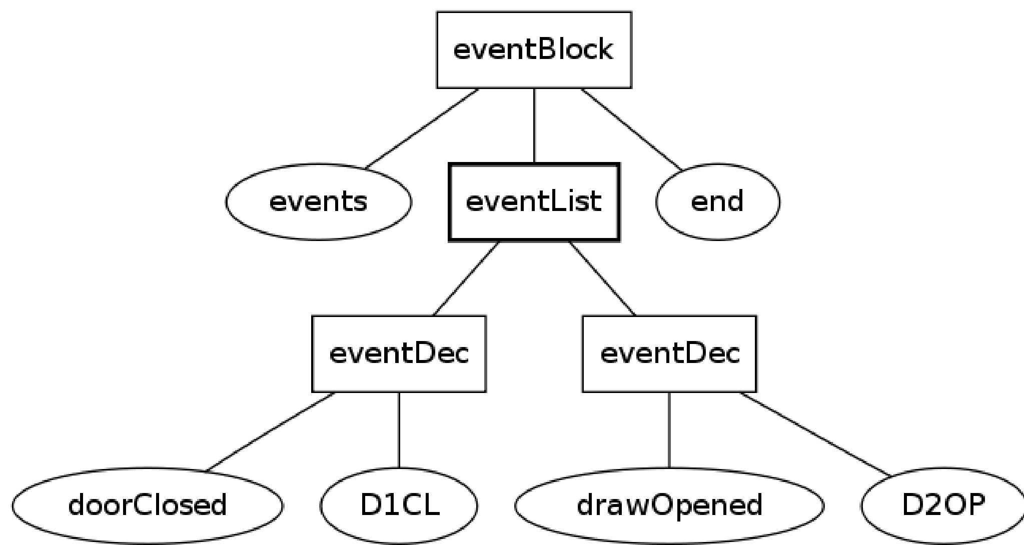


Come si può vedere, l'analisi sintattica presenta nodi extra (mostrati come rettangoli) per formare l'albero sintattico. Questi nodi sono definiti dalla grammatica.

E 'importante rendersi conto che ogni linguaggio può essere accompagnato da molte grammatiche. Così per questo caso, si potrebbe anche utilizzare la seguente grammatica:

eventBlock : Event-keyword eventList End-keyword;
*eventList : eventDec**
eventDec : Identifier Identifier;

Ciò corrisponde a tutti gli ingressi che la precedente formulazione avrebbe corrisposto, tuttavia, produce un diverso parse tree,



Così, in Sintassi-Directed Translation una grammatica definisce come un testo di input viene trasformato in un albero sintattico, e spesso si possono scegliere grammatiche differenti a

seconda di come si desidera controllare l'analisi. Grammatiche differenti appaiono anche a causa delle differenze negli strumenti di generatore di parser.

Finora si è parlato di albero di analisi come se fosse qualcosa che è esplicitamente prodotta dal parser come output dell'analisi, tuttavia, questo non è solitamente il caso.

Nella maggior parte dei casi, non è mai possibile accedere direttamente all'albero sintattico.

Il parser accumula pezzi dell'albero di analisi ed esegue azioni nel mezzo del parsing. Una volta creato un po 'di albero di analisi, e completate le azioni l'albero verrà eliminato (ciò è importante per ridurre il consumo di memoria).

Se si sta facendo una costruzione dell'albero, allora si produrrà un albero pieno di sintassi. Tuttavia, in questo caso, di solito non si produce l'albero completo di analisi, ma una semplificazione di esso chiamato albero di sintassi astratta.

Si può incorrere in una confusione terminologica particolare intorno a questo punto. Libri accademici in questo settore spesso usano "parse" come sinonimo solo per l'analisi sintattica, riferendosi al processo globale di qualcosa come la traduzione, interpretazione o compilazione. Altri, come Fowler, tendono ad usare "parse" molto più ampiamente. I generatori di parser tendono a fare riferimento al parser come l'attività che consuma token in modo da riferirsi al lexer e al parser come a strumenti separati. In questo paragrafo viene usato questo significato anche se spesso nella presente Tesi ci si esprime come se l'analisi dovesse comprendere anche il lexing.

Un'altra confusione terminologica è relativa ai termini "parse tree," ("albero di analisi"), "syntax tree" ("albero di sintassi") e "Abstract syntax tree" ("albero di sintassi astratta.") In generale si è usato "parse tree," per fare riferimento a un albero che rispetti fedelmente il parse con la grammatica che si ha con tutti i token presenti: essenzialmente l'albero grezzo. Albero di sintassi astratta (AST) è invece usato per fare riferimento a un albero semplificato, eliminando i token inutili e riorganizzando la struttura per facilitare il processamento successivo. Si usa albero di sintassi come il supertipo di albero AST e parse tree quando si ha bisogno di un termine per un albero che può essere entambi. Queste definizioni sono sostanzialmente quelle che si trovano in letteratura. Come sempre, la terminologia software varia un po' di più di quanto vorremmo.

Produzioni in output

Mentre la grammatica è adeguata a descrivere l'analisi sintattica, è appena sufficiente per un parser per riconoscere alcuni ingressi. Di solito non si è soddisfatti con il riconoscimento, si

vuole anche la produzione di un output. Si possono classificare i tre metodi generali per produrre output: Embedded Translation, Tree Construction, and Embedded Interpretation. Tutti questi richiedono qualcosa di diverso dalla grammatica per specificare come funzionano, per cui di solito si scrive codice aggiuntivo per ottenere la produzione in uscita.

Come una trama il codice all'interno del parser dipende da come è stato scritto il parser.

Con il Recursive Descent Parser, si aggiungono le azioni nel codice scritto a mano, con Parser Combinator, si passano oggetti azione nei combinatori utilizzando le strutture del linguaggio, con il generatore di parser, si utilizza il Foreign Code per aggiungere azioni dei codici nel testo del file di grammatica.

Predicati semantici

Gli analizzatori sintattici, sia scritti a mano che generati, seguiono un algoritmo di base che permette loro di riconoscere input sulla base di una grammatica. Tuttavia, a volte ci sono casi in cui le regole di riconoscimento non possono essere del tutto espresse nella grammatica e ciò risulta molto importante in un generatore di parser.

Al fine di far fronte a questo, alcuni generatori di parser supportano predicati semantici. Un predicato semantico è un pezzo di codice general-purpose che offre in risposta un booleano per indicare se una produzione grammaticale deve essere accettato o meno, passando oltre a ciò che è espresso dalla regola. In questo modo il parser può fare cose al di là di ciò che la grammatica può esprimere.

Un classico esempio della necessità di un predicato semantico si ha quando si analizza C++ e capita di imbattersi in T (6).

A seconda del contesto, questo potrebbe essere sia una funzione o un costruttore stile typecast. Per distinguerli, è necessario sapere come T è stata definita. Non è possibile specificare ciò in una grammatica context-free, quindi un predicato semantico è necessario per risolvere l'ambiguità.

Si consiglia di evitare di utilizzare predicati semantici per un DSL, dal momento che si dovrebbe essere in grado di definire il linguaggio in modo da evitare questa esigenza.

Quando è utile

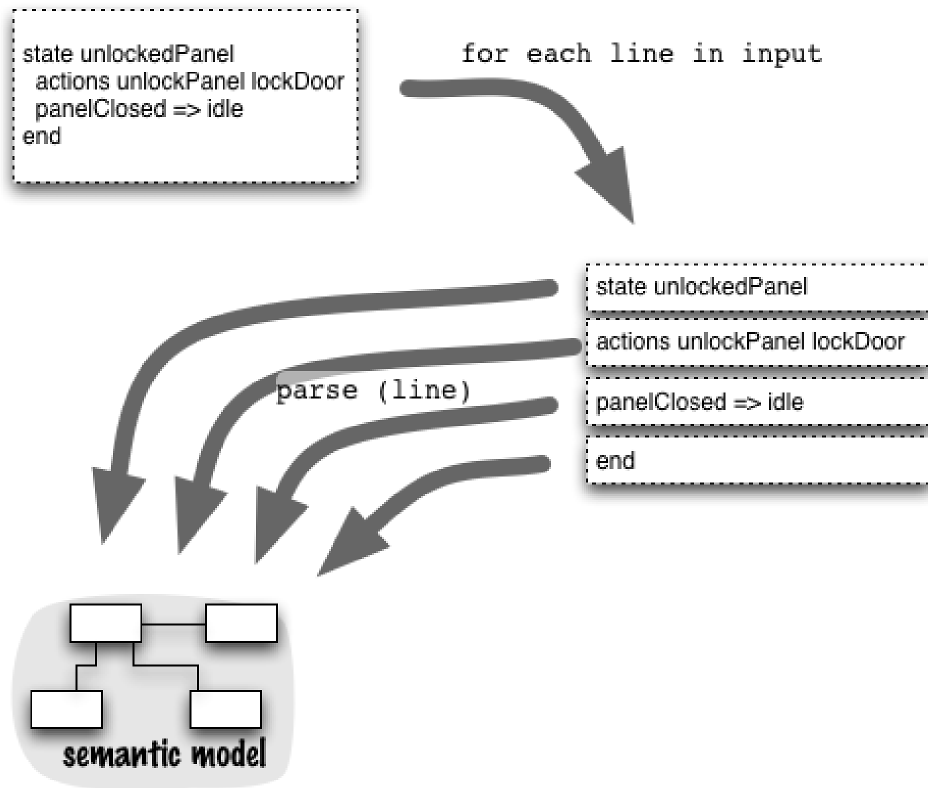
La Syntax Directed Translation è un approccio alternativo alla Delimiter Directed Translation. Il principale svantaggio della Syntax Directed Translation è la necessità di abituarsi a guidare il parser tramite una grammatica, mentre spezzettare utilizzando dei separatori è

solitamente più intuitivo. Non ci vuole molto ad abituarsi alle grammatiche e, una volta fatto, forniscono una tecnica più semplice man mano che il DSL diventa più complesso.

In particolare il file della grammatica, fornisce un chiara documentazione della struttura sintattica che il DSL accetta. Questo rende più semplice evolvere la sintassi del DSL nel tempo.

C. Delimiter Directed Translation

Traduce il testo dei sorgenti scomponendolo in pezzi (di solito linee) e poi effettuando il parsing di ciascun pezzo.



Come funziona

La traduzione Delimiter Directed funziona prendendo l'input e spezzandolo in parti più piccole delimitate da un qualche tipo di carattere. E' possibile utilizzare il carattere delimitatore che si preferisce, il più comune è il terminatore di linea.

Spezzare lo script in linee è solitamente abbastanza semplice dato che, la maggior parte degli ambienti di programmazione, hanno funzioni di libreria che leggono l'input una linea alla volta. Una complicazione può essere data da linee molto lunghe che vuoi fisicamente dividere all'interno dell'editor. In molti ambienti, la maniera più semplice per farlo è quella di "quotare" il terminatore di fine linea.

Quotare il terminatore di linea risulta poco elegante e, comunque, non funziona se c'è uno spazio bianco tra l'apice e la fine della linea. Spesso è meglio utilizzare un carattere di

continuazione della linea. Per fare ciò , scegli un qualche carattere, se è l'ultimo carattere diverso da uno spazio bianco su una linea allora la prossima linea è effettivamente la stessa linea. Quando leggi l'input devi controllare i caratteri di continuazione e se ne incontri appendi la prossima linea a quella che hai appena letto. E' bene ricordarsi che si possono incontrare più di un carattere di continuazione.

Come avviene l'elaborazione dipende dalla natura del linguaggio con cui si ha a che fare. Il caso più semplice è quando ogni linea è autonoma e con la stessa forma. Considera, ad esempio, un semplice lista di regole per la ripartizione di punteggi agli hotel in base alle notti pernottate:

```
score 300 for 3 nights at Bree
score 200 for 2 nights at Dol Amroth
score 150 for 2 nights at Orthanc
```

Con autonome si intende che nessuna linea ha a che fare con le altre. Posso riordinarle e rimuoverle in sicurezza senza cambiare l'interpretazione delle altre. Hanno la stessa forma perchè ciascuna regola codifica lo stesso tipo di informazioni. Elaborare le linee è perciò abbastanza semplice; eseguo la stessa funzione per il processamento di ciascuna di esse, questa funzione preleva le informazioni di cui ho bisogno le traduce nella rappresentazione desiderata. Se si utilizza traduzione embedded, ciò significa inserirle nel modello semantico mentre se si sta utilizzando la Tree Construction, significa creare un abstract syntax tree. E' raro utilizzare la tree construction con la traduzione delimiter directed.

Come prelevare le informazioni che servono dipende dalle capacità di elaborazione sulle stringhe che si hanno a disposizione nel proprio linguaggio e la complessità della linea che si ha. Se possibile, il modo più facile per decomporre l'input è quello di utilizzare una funzione per lo split delle stringhe. La maggior parte delle librerie per stringhe hanno una funzione che divide una stringa in elementi separati da un'altra stringa. In tal caso è possibile separare le parole utilizzando un carattere di spazio bianco ed estrarre ad esempio il punteggio (score) come secondo elemento.

A volte non è possibile separare una stringa in una maniera così pulita. Spesso l'approccio migliore è quello di utilizzare un'espressione regolare che offre una maggiore potenza espressiva rispetto allo split delle stringhe, è anche un buon modo per verificare che la linea sia sintatticamente corretta. Le espressioni regolari sono però più complicate e molti le trovano scomode da seguire. Spesso può aiutare scomporre un'espressione regolare complessa in sottoespressioni, definendo ogni sottoespressione separatamente e poi combinandole tra loro.

Ora si considerino linee di forme differenti. Questo potrebbe essere un DSL che descrive i contenuti della prima pagina per il quotidiano locale:

```
border grey
headline "Musical Cambridge"
filter by date in this week
show concerts in Cambridge
```

In questo caso ogni linea è autonoma ma ha bisogno di essere elaborata in maniera diversa. E' possibile gestire il tutto con un'espressione condizionale che controlla il tipo delle varie linee e chiama la routine appropriata.

```
if (isBorder()) parseBorder();
else if (isHeadline()) parseHeadline();
else if (isFilter()) parseFilter();
else if (isShow()) parseShow();
else throw new RecognitionException(input);
```

Il controllo nella condizione può essere fatto utilizzando le espressioni regolari o altri operatori per stringhe. Si potrebbe discutere se è corretto mostrare le espressioni regolari direttamente nella condizione ma l'uso di funzioni rende il codice più leggibile se vengono attribuiti dei nomi esplicativi.

Oltre alle linee puramente isomorfiche e puramente polimorfiche si potrebbero volere linee ibride in cui ogni linea ha lo stessa struttura divisa in clausole, ma ogni clausola ha forme differenti. Ecco un'altra versione per il punteggio degli hotel:

```
300 for stay 3 nights at Bree
150 per day for stay 2 nights at Bree
50 for spa treatment at Dol Amroth
60 for stay 1 night at Orthanc or Helm's Deep or Dunharrow
1 per dollar for dinner at Bree
```

In questo caso si ha una ampia struttura isomorfica. E' sempre presente una clausola di ricompensa, seguita da "for", seguita da una clausola per l'attività, seguita da "at", seguita da una clausola per la locazione. Posso risolvere questa situazione definendo una unica routine di alto livello che identifica le tre clausole e chiama un precisa routine per ciascuna clausola. Quest'ultima routine segue il pattern polimorfico che utilizza delle condizioni di test e differenti routine per l'elaborazione.

E' possibile paragonarle alle grammatiche utilizzate nella traduzione Syntax Directed. Linee polimorfiche e clausole sono gestite dalle grammatiche come alternative, mentre le linee isomorfiche sono gestite dalle regole di produzione senza l'uso di alternative. Utilizzare i metodi per spezzare le linee in clausole è come utilizzare sottoregole.

La gestione di istruzioni non autonome con la traduzione Delimiter Directed introduce ulteriori complicazioni, in quanto ora bisogna tenere traccia di informazioni riguardanti lo stato del parser. Un buon modo per implementarlo è quello di definire una famiglia di parser. Uno per ciascuno stato del parsing. Quindi il parser di alto livello sarà una macchina a stati che ad un cambiamento di stato cambia il comportamento attivando il parser specifico per quella linea.

Un' area comune in cui risulta scomodo utilizzare la traduzione Delimiter Directed è nella gestione degli spazi bianchi, in particolare attorno agli operatori. Se si ha una linea nella forma *property = value*, bisogna decidere se lo spazio attorno a "=" è opzionale o meno. Rendendolo opzionale si può complicare l'elaborazione delle linee mentre rendendolo obbligatorio il DSL diventa più difficile da usare. Gli spazi bianchi possono creare ulteriori problemi se si effettuare una distinzione tra uno o più spazi bianchi o tra diversi tipi di essi come i tab.

Quando è utile

La grande forza della traduzione Delimiter Directed è che è una tecnica molto semplice da utilizzare. La principale alternativa, la traduzione Syntax Directed, richiede di risalire una curva di apprendimento per capire come lavorare con le grammatiche. La traduzione Delimiter Directed fa affidamento solo su tecniche con cui la maggior parte dei programmatori hanno familiarità e perciò facili da approcciare.

L'altra faccia della medaglia è che spesso è difficile gestire linguaggi più complessi. La traduzione Delimiter Directed funziona molto bene con linguaggi semplici, in particolare quelli che non utilizzano contesti innestati. All'aumentare della complessità diventa caotico molto in fretta, in particolare se l'obiettivo è quello di mantenere il progetto del parser più pulito possibile.

Si preferisce quindi utilizzare la traduzione Delimiter Directed quando si hanno semplici istruzioni autonome o, al massimo, con un singolo contesto innestato.

D. Recursive Descent Parser

Crea un parser top-down sfruttando il controllo di flusso per gli operatori della grammatica e le funzioni ricorsive per i riconoscitori dei simboli non terminali.

```
boolean eventBlock() {
        boolean parseSuccess = false;
        Token t = tokenBuffer.nextToken();
        if (t.isTokenType(ScannerPatterns.TokenTypes.TT_EVENT)) {
                tokenBuffer.popToken();
                parseSuccess = eventDecList();
        }
        if (parseSuccess) {
                t = tokenBuffer.nextToken();
                if (t.isTokenType(ScannerPatterns.TokenTypes.TT_END)) {
                        tokenBuffer.popToken();
                }
                else {
                        parseSuccess = false;
                }
        }
        return parseSuccess;
}
```

Molti DSL sono linguaggi abbastanza semplici. La flessibilità di un linguaggio esterno è allettante ma utilizzare un Parser Generator per creare un parser introduce nuovi strumenti e linguaggi in un progetto, complicando il processo di compilazione.

Un Recursive Descent Parser garantisce la flessibilità di un DSL esterno senza il bisogno di un Parser Generator. Il Recursive Descent Parser può essere implementato in qualsiasi linguaggio general purpose si voglia. Utilizza operatori per il controllo di flusso per implementare diversi operatori della grammatica. Singoli metodi o funzioni implementano le regole di parsing per i simboli non terminali della grammatica.

Come funziona

Come in altre implementazioni, si separano la fase di analisi lessicale dal parsing. Un Recursive Descent Parser riceve uno stream di token da un analizzatore lessicale come un Lexer basato su una tabella di Regular expression.

La struttura base di un Recursive Descent Parser è abbastanza semplice. C'è un metodo per ogni simbolo non terminale della grammatica. Il metodo implementa le varie regole di produzione associate al simbolo non terminale. Il metodo restituisce un valore booleano che

rappresenta il risultato del match. Un fallimento a qualsiasi livello viene propagato in verticale nello stack delle chiamate. Ciascun metodo opera sul buffer dei token, avanzando il puntatore quando una porzione di frase viene riconosciuta.

Dato che gli operatori nelle grammatiche sono relativamente pochi (sequenze, alternative e ripetizioni), le implementazioni dei metodi utilizzano un numero ridotto di pattern. Si consideri inizialmente l'operatore per le alternative, che utilizza un' istruzione condizionale. Ecco un frammento di grammatica:

```
grammar file...
C : A | B
```

la funzione corrispondente sarà semplicemente così:

```
boolean C ()
        if (A())
                then true
                else if (B())
                        then true
                        else false
```

Questa implementazione evidentemente controlla una alternativa alla volta, comportandosi più come alternative ordinate. Se si vuole permettere l'ambiguità introdotta dalle alternative non ordinate, può essere il caso di utilizzare un Parser Generator.

Se la chiamata ad A() ha successo, il buffer dei token viene avanzato fin dal primo token incontrato da A. Se la chiamata ad A() fallisce, il buffer non deve essere modificato.

L'operatore per le sequenze viene implementato con istruzioni *if* innestate, visto che non si vuole continuare il processamento se uno dei metodi fallisce. Quindi l'implementazione di :

```
grammar file...
C : A B
```

sarà banalmente:

```
boolean C ()
        if (A())
        then if (B())
                then true
                else false
        else false
```

L'operatore opzione è leggermente diverso.

```
grammar file...
C: A?
```

Bisogna tentare di riconoscere il token individuato dal simbolo non terminale A, ma non c'è modo di fallire. Se si trova A, si ritorna *true*. Se non si trova A si ritorna comunque *true* dato che A è opzionale. L'implementazione è così:

```
boolean C ()
A()
true
```

Se il riconoscimento di A fallisce, il buffer dei token rimane dov'era quando C era stato chiamato. La chiamata a C ha successo in entrambi i casi.

L'operatore di ripetizione ha due forme: zero o più istanze (" * "),ed uno o più istanze (" + "). Per implementare il secondo:

```
grammar file...
C: A+
```

si potrebbe usare il seguente pattern:

```
boolean C ()
    if (A())
        then while (A())
            {}
        true
    else
        false
```

Questo codice controlla che almeno una A sia presente. Se quello è il caso, la funziona continua a cercare il maggior numero di A che riesce, ma ritornerà sempre *true*, perchè almeno una A è stata trovata. Nel codice per una lista che permette zero o più istanze semplicemente non è presente l'istruzione *if* più esterna e ritorna sempre *true*.

Tabella riassuntiva

Regola della grammatica	Implementazione
A | B	if (A()) then true else if (B()) then true else false
A B	if (A()) then if (B()) then true else false else false
A ?	A(); true

A *	while A(); true

A +	if (A()) then
	while (A());
	else false

Si utilizza lo stile delle helper function per mantenere le azioni distinte dal parsing. Sia la Tree Construction che l'Embedment Translation sono entrambi possibili con un Recursive Descent Parser.

Per rendere questo approccio pulito come effettivamente è, i metodi che implementano le regole di produzione devono comportarsi in modo consistente. Le regole più importanti riguardano la gestione del buffer dei token in input. Se il metodo incontra ciò che sta cercando, la posizione corrente del buffer dei token viene avanzata subito dopo il token riconosciuto. Se, ad esempio, la parola chiave *event* viene riconosciuta, il buffer avanza di una posizione. Se l'individuazione fallisce, la posizione del buffer deve rimanere la stessa di quando il metodo era stato chiamato.

Quando è utile

Il maggior punto di forza del Recursive Descent Parser è la sua semplicità. Una volta capito l'algoritmo di base e come gestire vari operatori grammaticali, la scrittura di un Recursive Descent Parser è un compito di semplice programmazione. Si ha un parser in una classe ordinaria nel progetto. Il testing di sistema si esegue in modo classico, in particolare, un test di unità ha più senso quando l'unità è un metodo, come qualsiasi altro. Infine, dato che il parser è semplicemente un programma, è facile ragionare sul suo comportamento e quindi sul debug.

Il Recursive Descent Parser è una diretta applicazione di un algoritmo di analisi, che rende il tracciamento del flusso attraverso l'analisi molto più facile da discernere.

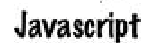
Il difetto più grave del Recursive Descent Parser è che non vi è alcuna rappresentazione esplicita della grammatica. Con la codifica la grammatica verso l'algoritmo Recursive Descent, si perde il quadro chiaro della grammatica, che può esistere solo nella documentazione o nei commenti. Entrambi, parser combinator e parser generator, hanno una dichiarazione esplicita della grammatica, il che li rende più facile da capire ed evolvere.

E. Foreign code

L’inserimento di foreign code in un external DSL serve per renedere più elaborato il comportamento specificabile nel DSL.

DSL

```
scott handles floor_wax in MA RI CT when {/^Baker/.test(lead.name)};
```



Per definizione, un DSL è un linguaggio limitato che può esprimere poche cose. A volte però si può avere la necessità di descrivere qualcosa nello script DSL che va al di là delle capacità di quest'ultimo. Una soluzione può essere quella di estendere il DSL per aggiungere queste capacità, ma seguire questa strada può complicare significativamente il DSL, eliminando molta della semplicità che lo rende interessante.

Il foreign code incapsula un linguaggio differente, spesso general-purpose, in certi punti del DSL.

Come funziona

Inserire parti di un altro linguaggio in un DSL implica due questioni:

- come si possono riconoscere questi pezzi di Foreign Code, modificando opportunamente la grammatica?
- come si può eseguire il codice?

Il foreign code compare soltanto in certi punti di un DSL, quindi sarà la grammatica che marcherà i punti in cui questo codice può apparire. Un problema nella gestione del foreign code è che la grammatica non sarà in grado di riconoscere la struttura interna del foreign code. Di conseguenza, ci sarà bisogno di utilizzare l' alternative tokenization con il foreign code, leggendo quest'ultimo come un'unica stringa.

E' possibile anche inserire questa stringa nel modello semantico così com'è oppure passarlo ad un parser ad-hoc in modo da ottenere un inserimento più elegante nel modello semantico. Quest'ultima opzione è più complicata, ma si potrebbe perseguire se il Foreign Code

è un altro DSL. Spesso, il foreign code è un linguaggio general purpose, nel qual caso l'uso della stringa è più che sufficiente.

Una volta che il foreign code è nel modello semantico, bisogna decidere che cosa farne. Il problema principale dipende dal fatto che il linguaggio vada interpretato oppure compilato.

Il foreign code interpretato è solitamente il caso più semplice, perchè fornisce un meccanismo per interoperare con il linguaggio host. Se il linguaggio host del sistema è anch'esso interpretato, è facile utilizzare il linguaggio host stesso come foreign code. Se invece il linguaggio host è compilato allora bisognerà usare un linguaggio interpretato che verrà chiamato dal linguaggio host. Nel tempo molti linguaggi statici hanno acquisito la capacità di interagire con i linguaggi interpretati. Nel caso ci sia bisogno di trasferire dati può invece risultare troppo problematico, in quanto potrebbe anche introdurre un nuovo linguaggio al progetto, il che crea un nuovo problema.

L'alternativa è quella incapsulare il linguaggio anche se è un linguaggio compilato. La complessità, in tal caso, è che si introduce una fase extra nel processo di compilazione, proprio come se si stesse utilizzando la generazione di codice. Certamente, anche se si sta già utilizzando la generazione di codice, questo passo extra è da realizzare in ogni caso, quindi aggiungendo foreign code compilato non complica le cose ulteriormente. La complessità è da considerare se si sta compilando del codice quando il modello semantico è interpretato.

Tutte le volte che si usa un foreign code general purpose, va seriamente considerata la possibilità di utilizzare un Embedment Helper. In questo modo, il foreign code nel proprio script DSL dovrebbe essere il minimo necessario nel contesto del DSL, richiamando l'Embedement Helper per tutti i compiti più generali. Uno dei problemi più grossi adottando un foreign code è che introducendone molto può sopraffare il DSL, perdendo così tutti i vantaggi derivanti dalla leggibilità di un DSL. L' Embedement Helper è una tecnica semplice ed è utile nei casi meno complicati.

A volte, nel foreign code si fa uso di simboli definiti nello script DSL; questo accade quando lo script DSL include variabili o altri modi per creare costrutti indirettamente. Mentre questi sono onnipresenti nei linguaggi general purpose, non è facile trovarli nei DSL visto che spesso un DSL non ha bisogno di questo tipo di espressività. Di conseguenza, può sembrare che si verifichi raramente nella pratica, mentre, in realtà, è una situazione comune che si verifica nelle grammatiche. Eccone un esempio:

```
allocationRule
: salesman=ID pc=productClause lc=locationClause
```

```
('when' predicate=ACTION)? SEP
{helper.recognizedAllocationRule(salesman, pc, lc, predicate);}
;
```

In questo caso il foreign code è Java. Il codice Java include riferimenti a *salesman*, *pc*, *lc*, e *predicate*, ognuno dei quali sono siboli definiti nella grammatica. Quando viene eseguito il foreign code, il Parser Generator ha bisogno di risolvere questi riferimenti.

Quando è utile

Quando si pensa di utilizzare il foreign code, l'alternativa solitamente è quella di estendere il DSL per fare ciò che dovrebbe fare il foreign code. Introdurre foreign code ha certamente degli aspetti negativi. Utilizzandolo infatti si compromettono le astrazioni fornite dal DSL. Chiunque legge il DSL ha bisogno di capire il foreign code allo stesso modo del DSL, come se fosse un'estensione. Inoltre, utilizzare foreign code complica il processo di parsing e il modello semantico.

Queste complessità aggiuntive vanno messe a confronto con quelle che si intendono aggiungere al codice DSL per supportare le nuove capacità. Aumentando la potenza del DSL però, diminuisce la facilità d'uso e di comprensione.

Quindi quali sono i casi in cui conviene usare foreign code? Una situazione naturale è quando si ha bisogno di un linguaggio general purpose ma non si vorrà trasformare il DSL in un linguaggio general purpose, perciò questo spingerà ad utilizzare foreign code.

Un altro caso è quando si ha bisogno di una certa capacità di rado nel DSL. In tal caso potrebbe essere scomodo estendere il DSL.

Un fattore importante nella decisione è il tipo di utente che utilizza il DSL: se il DSL è utilizzato solo da programmatori, allora aggiungere foreign code non è un problema; saranno in grado di capire il foreign code così come il DSL. Se altre persone, diverse dai programmatori, dovranno leggere il DSL non ha molto senso inserire foreign code perchè non saranno in grado di capirlo. Se il foreign code viene usato per gestire casi poco frequenti, potrebbe non essere un grosso problema.

F. Context Variable

Si tratta di una variabile che durante il parsing memorizza il contesto. Si immagini di dover scorrere una lista di oggetti, catturando i dati di ciascuno di essi. Ogni bit di informazione di un oggetto può essere catturato indipendentemente ma hai bisogno di sapere da quale oggetto stai ottenendo le informazioni. Una context variable permette questo mantendo l'oggetto in una variabile e riassegnandola quando ci si sposta su di un nuovo oggetto.

Come funziona

Stai utilizzando una context variable tutte le volte che chiami una variabile *currentItem* aggiornandola periodicamente durante il parsing quando ti sposti da un oggetto ad un'altro nello script di input.

Una context variable può essere un oggetto del modello semantico oppure un builder. Il modello semantico è più lineare in superficie, soltanto se tutte le propietà sono mutabili quando il parser ha bisogno di modificarle. Se non è questo il caso, è solitamente meglio utilizzare un qualche tipo di builder per raccogliere le informazioni e poi creare un oggetto del modello semantico.

Quando usarla

Esistono molte situazioni in cui hai bisogno di memorizzare il contesto durante il parsing e una context variable è una scelta ovvia.

Le context variable sono però problematiche particolarmente se se ne usano molte. Per loro natura, hanno uno stato mutabile di cui bisogna tenere traccia e i bug adorano questo tipo di stato mutabile. E' facile dimenticarsi di aggiornare la context variable al momento giusto e il debugging, in questi casi, può essere abbastanza difficile. Esistono di solito modi alternativi di organizzare il parsing che possono ridurre il bisogno di context variables.

G. Notification

Colleziona errori ed altri messaggi da riportare al chiamante.

Ho effettutato alcune operazioni che hanno cambiato significativamente un'oggetto del modello. Ora che ho termitato, voglio controllare che il modello risultante sia valido. Posso perciò lanciare un comando valido e voglio sapere la risposta come un semplice booleano, ma se ci sono errori voglio maggiori informazioni. In particolare, voglio conoscere tutti gli errori anzichè fermarmi al primo errore.

Una notification è un oggetto che colleziona errori. Quando una validazione fallisce, viene aggiunto un errore alla notification. Una volta che la validazione termina, viene ritornata una notification. Posso quindi chiedere alla notification se è tutto a posto e se non lo è vorrei approfondire gli errori.

Come funziona

La versione base della notification è una collezione di errori. Svolgendo il task da notificare, devo avere la possibilità di aggiungere un errore alla notification. Questo può avere la forma di una semplice stringa oppure può essere un oggetto più complesso. Quando il task è terminato la notification ritorna al chiamante. Il chiamante invoca un semplice metodo che ritorna un booleano per capire se tutto è andato bene. Se ci sono stati errori il chiamante può interrogare la notification ulteriormente per visualizzarli.

La notification solitamente deve essere disponibile a molti metodi nel modello. Può essere altrimenti essere passato in un argomento come parametro oppure può essere nascosto in un campo se c'è un oggetto che corrisponde al task a portata di mano, come ad esempio un validator object.

Lo scopo principale di una notification è quello di collezionare gli errori, ma a volte è utile catturare anche warning e messaggi informativi. Un errore indica che il comando richiesto è fallito; un warning si verifica quando qualcosa che non è fallito va comunque portato all'attenzione del chiamante. Un messaggio informativo comunica solamente qualche informazione potenzialmente utile.

Per certi versi, una notification è un oggetto che agisce come un file di log, quindi molte funzionalità che troviamo comunemente in fase di logging possono anche essere utili qui.

Quando usarla

Una notification è utile tutte le volte in cui bisogna eseguire un' operazione complicata che può scatenare errori multipli e non vuoi fallire al primo errore. Se vuoi evitare il fallimento

al primo errore puoi semplicemente scatenare un'eccezione. Una notification ti permette di memorizzare più eccezioni per restituire al chiamante un'immagine più completa di ciò che la richiesta ha provocato.

Le notification sono particolarmente utili quando dall'interfaccia utente viene avviata un'operazione nel livello inferiore. Il livello inferiore non deve cercare di interagire con l'interfaccia utente direttamente, quindi una notification diventa il giusto veicolo di informazioni.

H. New line separator

Utilizzare il terminatore di linea come separatore di istruzioni.

```
first statement
second statement
third statement
```

Come funziona

Utilizzare il terminatore di linea per contrassegnare la fine di un'istruzione è una usanza comune nei linguaggi di programmazione. Questa tecnica si adatta molto bene alla traduzione Delimiter Directed dato che questo carattere viene utilizzato solitamente come separatore per spezzare il sorgente in input.

Con la traduzione Syntax Directed, però, l'uso di questi separatori diventa piuttosto complicato, introducono infatti numerose trappole in cui si rischia di cadere. In questa parte verranno evidenziate alcune di esse.

Il motivo per cui il terminatore di linea e la traduzione Syntax Directed non vanno d'accordo è che questo carattere gioca un duplice ruolo quando lo si usa come separatore. Oltre al suo ruolo sintattico, ha anche un ruolo nella formattazione aggiungendo spazi in verticale. Di conseguenza, possono apparire spazi dove non ci si aspettano separatori di istruzioni.

Di seguito è possibile osservare una grammatica ovvia per l'uso di terminatori di linea come separatori:

```
catalog : statement*;
statement : 'item' ID EOL;
EOL : '\r'? '\n';
ID : ('a'..'z' | 'A'..'Z' | '0'..'9' | '_' )+;
WS : (' ' |'\t' )+ {$channel = HIDDEN;} ;
```

Questa grammatica cattura un semplice lista di oggetti in cui ogni linea c'è una parola chiave item seguita da un identificatore dell' oggetto. E' buona abitudine utilizzare questa grammatica come esempio in stile "Hello World" per il parsing, è semplice da seguire (parole chiave, identificatori, terminatori di linea) ma introduce alcuni casi che creano problemi:

- linee bianche tra le istruzioni
- linee bianche prima della prima istruzione
- linee bianche dopo l'ultima istruzione
- l'ultima istruzione sull'ultima linea non ha il terminatore di linea

I primi tre casi riguardano tutti le linee bianche, ma c'è bisogno di modi diversi per gestirli nella grammatica, quindi andrebbero tutti testati. Assicurarsi di avere dei test per questi casi è probabilmente la cosa più importante da fare. In seguito verranno mostrate alcune soluzioni, ma dei buoni test sono la chiave per garantire che queste situazioni sono gestite correttamente.

Un modo per gestire le linee bianche correttamente è quello di usare un terminatore di istruzione che combacia con terminatori di linea multipli. Il punto più logico in cui inserire questa regola è nel lexer. Il tutto è però complicato dall'ultimo test. Per gestire quel caso c'è bisogno di gestire il carattere di fine file nel lexer, il che è possibile in base al Parser Generator che si sta utilizzando. In ANTLR per fare ciò basta aggiungere una regola alla grammatica:

```
catalog : verticalSpace statement*;
statement : 'item' ID eos;
verticalSpace : EOL*;
eos : EOL+ | EOF;
```

Un terminatore mancante nell'ultima linea è spesso un caso difficile. Quanto difficile dipende da come il Parser Generator tratta il carattere di fine file. ANTLR lo presenta al parser come token, questo è il motivo per cui è possibile individuarlo con una regola nel parser (e non nel lexer). Altri rendono questo riconoscimento molto difficile se non impossibile. Un'opzione da considerare è forzare un carattere di fine linea alla fine o tramite il lexer oppure in fase di pre-lexing. Questo può aiutare nell'evitare alcuni casi complicati.

Un altro approccio per trattare i separatori di istruzioni è considerarli come separatori invece di terminatori. Ciò porta ad avere la seguente grammatica:

```
catalog : verticalSpace statement (separator statement)* verticalSpace;
statement : 'item' ID;
separator : EOL+;
verticalSpace : EOL*;
```

Una terza alternativa è pensare al corpo dell'istruzione come un elemento opzionale per ogni linea nel catalogo.

```
catalog : line* ;
line : EOL | statement EOF | statement EOL;
statement : 'item' ID;
```

Questa regola ha bisogno di individuare esplicitamente il carattere di fine file allo scopo di gestire il caso del carattere di fine linea mancante. Se non è possibile individuare il carattere di fine file, si usi questa grammatica:

```
catalog : line* statement?;
```

```
line : statement? EOL;
statement : 'item' ID;
```

Si perde in chiarezza ma si ha il vantaggio di non aver bisogno del carattere di fine file.

Quando usarlo

Decidere di utilizzare i terminatori di linea come separatori comporta in realtà due decisioni separate: decidere di avere i separatori e poi decidere di usare questi caratteri allo scopo.

La struttura limitata di DSL spesso implica che si può sopravvivere anche senza i separatori tra istruzioni. Il parser può solitamente capire il contesto nel parsing dalle diverse parole chiave che si usano.

I separatori di istruzioni possono facilitare l'individuazione degli errori. A tale scopo il parser ha bisogno di un marker di controllo che indichi a che punto del parsing ci si trova. Senza questi marker un errore su una certa linea dello script potrebbe essere individuata soltanto molte linee dopo, generando errori fuorvianti. I separatori di istruzioni possono spesso adempiere a questo ruolo (anche le parole chiave posso servire allo stesso scopo).

Se si decide di utilizzare i separatori di istruzioni, la scelta da fare è tra un carattere visibile, come il punto e virgola, e il terminatore di linea. La cosa più interessante dei terminatori di linea è che il più delle volte, avendo un'istruzione per linea, non introducono rumore sintattico al DSL. Ciò è molto importante quando si ha a che fare persone diverse dai programmatori, anche se pure i programmatori li preferiscono. Il rovescio della medaglia è che la traduzione Syntax Directed deve essere fatta utilizzando le tecniche descritte sopra. Bisogna anche assicurarsi di avere dei test che coprano i casi problematici.

I.Symbol Table

Una locazione dove memorizzare tutti gli oggetti identificabili durante il parsing allo scopo di risolvere i riferimenti.

Molti linguaggi offrono la possibilità di rifersi ad un oggetto in più punti all'interno del codice. Se abbiamo un linguaggio che definisce una configurazione di task e delle loro dipendenze, abbiamo bisogno di un modo per i task di riferirsi al task da cui dipendono nella loro stessa definizione.

A tale scopo definiamo un insieme di simboli per ciascun task; elaborando lo script inseriamo questi simboli all'interno di una Symbol Table che memorizza il link tra il simbolo e l'istanza dell'oggetto che contiene le informazioni.

Come funziona

L'obiettivo principale della Symbol table è quello di mappare il simbolo utilizzato per riferirsi ad un oggetto in DSL script con l'oggetto a cui il simbolo si riferisce. Strutture dati come le mappe ben si prestano a realizzare questo tipo di mapping perciò non c'è da stupirsi se, nella maggior parte delle implementazioni della Symbol table, vengono utilizzate le mappe sfruttando il simbolo stesso come chiave e un oggetto del modello semantico come valore.

Una questione da considerare è il tipo di oggetto che andrebbe utilizzato per la chiave nelle Symbol table. Per la maggior parte dei linguaggi, la scelta più comune è il tipo stringa visto che il testo di un DSL è anch'esso una stringa.

La maggior parte delle volte in cui c'è bisogno di utilizzare un tipo diverso dalla stringa è quando il linguaggio supporta un tipo per i simboli. I simboli sono simili alle stringhe dal punto di vista strutturale, infatti un simbolo non è altro che un sequenza di caratteri, ma differiscono per quanto riguarda l'aspetto comportamentale. Molti degli operatori utili per le stringhe (concatenazione, generazione di sottostringhe, ecc.) non hanno senso per i simboli. I simboli vengono usati principalmente nelle ricerche e i tipi simbolo vengono progettati tenendo a mente ciò. Quindi, mentre due stringhe, "foo" e "foo", sono spesso oggetti diversi che vanno comparati osservando il contenuto, i simboli :foo e :foo si riferiscono sempre allo stesso oggetto e possono essere comparati più velocemente.

Le performance possono essere una buona ragione per preferire i tipi simbolo alle stringhe, ma per piccoli DSL potrebbe non fare molta differenza. Il motivo principale per preferire il tipo simbolo è che comunica più chiaramente le proprie intenzioni quando lo si usa.

Dichiarando qualcosa di tipo simbolo, stai anche dicendo come pensi di utilizzarla e questo rende il tuo codice più facile da comprendere.

I linguaggi che supportano i simboli solitamente hanno per loro una particolare sintassi. Ruby utilizza :*aSymbol*, Smalltalk utilizza *#aSymbol* e Lisp tratta tutti gli identificatori come simboli. Questo fa in modo che i simboli risaltino ancor più negli internal DSL.

I valori nella symbol table possono essere degli oggetti di tipo final oppure degli intermediate builders. Utilizzare oggetti del modello fa in modo che l'obiettivo della symbol table diventi più quello di memorizzare i risultati, che è utile per situazioni non molto complesse, inserire invece un oggetto builder come valore garantisce una maggior flessibilità al prezzo di un po' di lavoro in più.

Molti linguaggi hanno bisogno di riferirsi a diversi tipi di oggetti: utilizzare una unica mappa come symbol table comporta che tutte le ricerche di qualsiasi simbolo vengono fatte sulla stessa mappa. Una conseguenza immediata è che non si può utilizzare lo stesso simbolo per diversi tipi di oggetto, il che può essere un vincolo utile a ridurre la confusione nel DSL ma rende più difficile leggere il codice di elaborazione dato che è meno chiaro che tipo di oggetto viene manipolato quando ci si riferisce ad un simbolo.

Utilizzando mappe multiple si ha una mappa separata per ogni tipo di oggetto a cui ci si vuole riferire. Concettualmente si può pensare alla symbol table come una singola symbol table logica oppure come a tre diverse symbol table.

In certi casi, avviene che gli oggetti vengono riferiti prima di essere propriamente definiti, questa caratteristica viene chiamata forward references. I DSL solitamente non hanno regole stringenti circa la dichiarazione di identificatori prima del loro uso quindi le forward references spesso hanno senso. Se si permettono le forward references, hai bisogno di garantire che qualsiasi riferimento ad un simbolo popolerà l'entry nella symbol table a meno che non sia già presente. Ciò spesso ti spinge ad utilizzare builders come valori nella symbol table ma in ogni caso gli oggetti del modello già garantiscono molta flessibilità.

Se non esiste una dichiarazione esplicita dei simboli, devi controllare che i simboli siano scritti correttamente, il che può essere fonte di errori.

Linguaggi più complicati spesso hanno scope innestati, dove i simboli sono definiti solo in un sottoinsieme dell'intero programma. Ciò è molto comune per i linguaggi general purpose ma molto raro in un DSL. Se si deidera supportare questa funzionalità si possono sfruttare le Symbol Table per scope innestati.

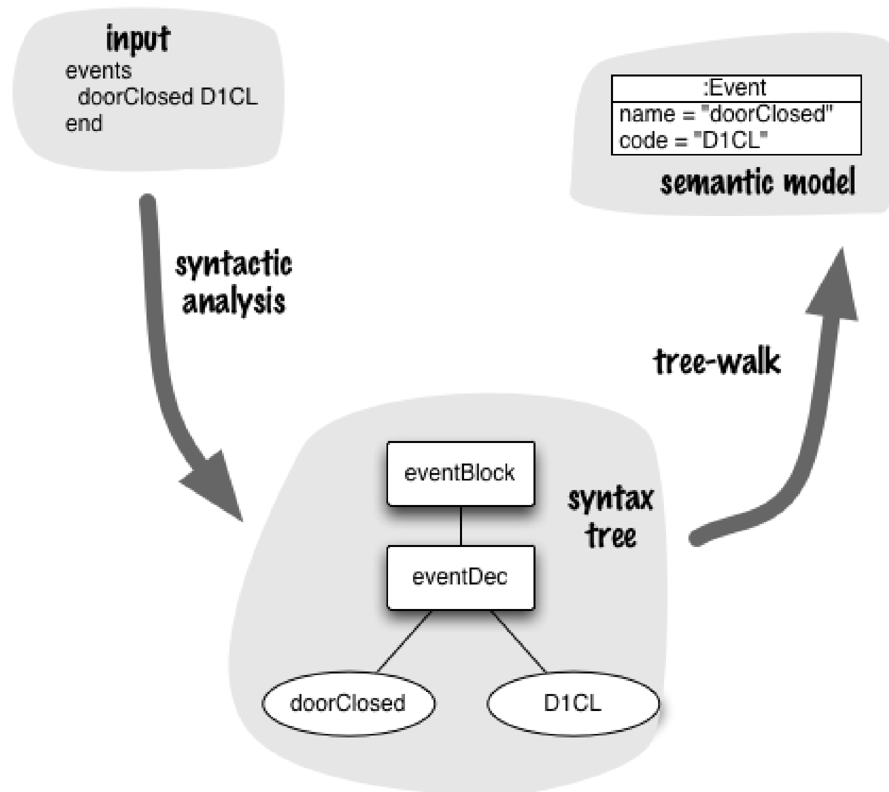
Quando è utile

Le Symbol Tables sono comuni a tutte gli esercizi sui linguaggi di elaborazione esercizio fisico, e si avrà quasi sempre bisogno di usarli.

E' bene notare, però, che ci sono momenti in cui non sono strettamente necessari. Con Tree Construction, si può sempre girare attorno all'albero di sintassi per trovare le cose. Spesso, una ricerca sul modello semantico che si sta costruendo potrebbe semplificare il lavoro ma a volte, è necessario un po di stoccaggio intermedio che facilita il tutto.

J. Tree Construction

Il parser crea e restituisce una rappresentazione ad albero del sorgente che viene manipolato in seguito.



Come funziona

Qualsiasi parser che utilizza la traduzione Syntax Directed costruisce un albero mentre esegue il parsing, si costruisce l'albero sullo stack, tagliando i rami su cui ha terminato il lavoro. Sfruttando la tecnica Tree Construction, si definiscono delle azioni nel parser che creano il syntax tree in memoria durante il parsing. Una volta che il parsing è terminato, si ha a disposizione l'intero syntax tree dello script DSL. Si possono apportare ulteriori modifiche sulla base del syntax tree. Se si sta utilizzando un modello semantico, si può scrivere del codice che naviga il syntax tree e popola il modello semantico.

Il syntax tree che viene creato in memoria non deve necessariamente corrispondere al parse tree creato dal parser, infatti solitamente non lo è. Al contrario, viene creato ciò

solitamente è chiamato abstract syntax tree. Un AST è una semplificazione del parse tree che fornisce un migliore rappresentazione del linguaggio in input.

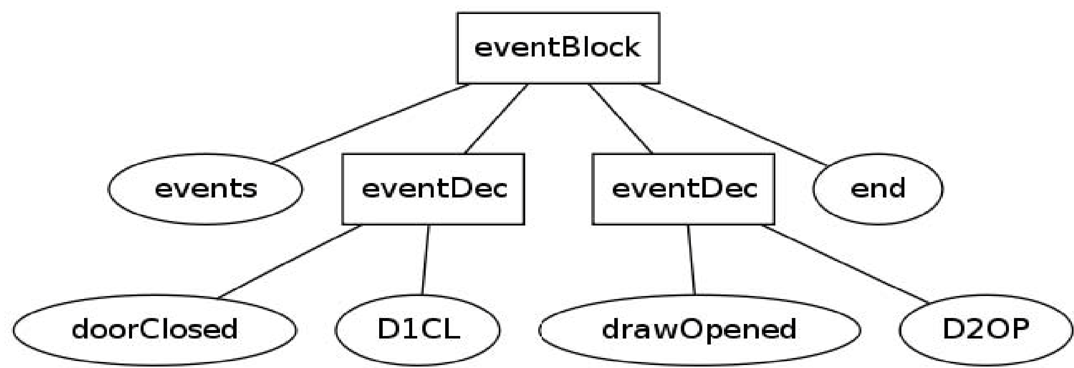
Si osservi il breve esempio qui sotto.

events
doorClosed D1CL
drawOpened D2OP
end

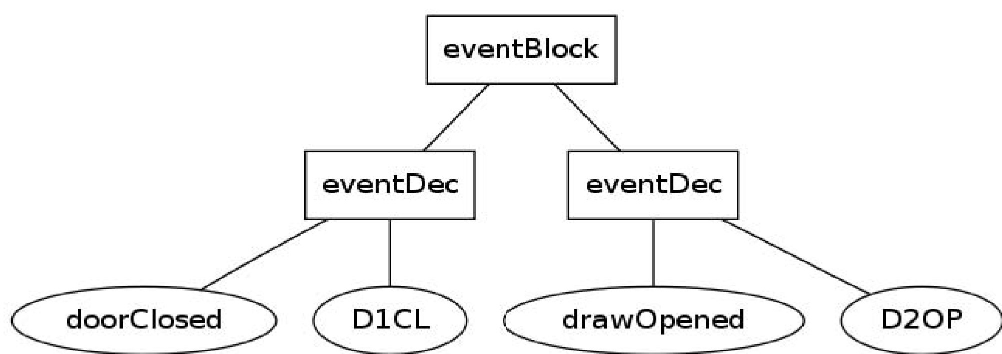
Il parsing viene effettuato con questa grammatica:

declarations : eventBlock commandBlock;
eventBlock : Event-keyword eventDec End-keyword;*
eventDec : Identifier Identifier;
commandBlock : Command-keyword commandDec End-keyword;*
commandDec : Identifier Identifier;

producendo l'albero in figura:



Osservando l'albero, si dovrebbe notare che i nodi *events* e *end* non sono necessari. Le parole erano utili nel testo in input allo scopo di contrassegnare i confini della dichiarazione degli eventi, ma dopo il parsing non ne abbiamo più bisogno. E' quindi più pulito rappresentare l'input con il seguente syntax tree :



Questo albero non è una fedele rappresentazione dell'input, ma è ciò di cui si ha bisogno per elaborare gli eventi. E' un'astrazione dell'input che meglio si presta allo scopo. Ovviamente, si può avere bisogno di diversi AST per vari motivi; se si ha bisogno di elencare i codici degli eventi, si possono eliminare i nodi del nome e *eventDec* lasciando solamente i codici.

Allo scopo di costruire il syntax tree, si possono utilizzare le azioni all'interno della grammatica BNF. In particolare, la capacità delle azioni di ritornare, per certo nodo, un valore è molto utile in questa tecnica; ogni azione assembla la rappresentazione di un nodo nel syntax tree.

Alcuni Parser Generators vanno oltre, dando la possibilità a un DSL di specificare il syntax tree. In ANTLR, per esempio, si può creare l'AST di prima utilizzando una regola come questa:

```
eventDec : name=ID code=ID -> ^(EVENT_DEC $name $code);
```

L'operatore -> introduce la regola per costruire l'albero. Il corpo della regola è una lista dove il primo elemento è il tipo di nodo (*EVENT_DEC*) seguito dai nodi figli, che in questo caso sono i token del nome e del codice.

Utilizzare un DSL per la Tree Construction può semplificare molto la costruzione dell'AST. Spesso, i Parser Generator che supportano questa funzionalità restituiscono il parse tree se non si forniscono regole per la Tree Construction, ma non si vuole pressochè mai il parse tree. Di solito quindi si preferisce utilizzare le regole per semplificarlo il parse tree in un AST.

Un AST costruito in questo modo sarà costituito da oggetti generici che contengono i dati per l'albero. Nell'esempio sopra il nodo *eventDec* è un generico nodo con nome e codice come nodi figli. Entrambi i nodi nome e codice sono token generici. Se si costruiscono da sè gli alberi usando le azioni, si possono creare veri oggetti, come un oggetto evento con nome e codice nei campi. Si preferisce utilizzare un generico AST e in un secondo momento trasformarlo nel modello semantico. E' comunque meglio avere due trasformazioni semplici rispetto ad una più complicata.

Quando è utile

Sia la Tree Construction che l'Embedded Translation sono approcci utili a popolare il modello semantico durante il parsing. L'Embedded Translation effettua la trasformazione in un singolo passo, mentre la Tree Construction ne effettua due utilizzando un AST come modello intermedio. Un motivo per utilizzare la Tree Construction è che spezza una singola trasformazione in due trasformazioni più semplici. Quando sia utile avere a che fare con

modello intermedio dipende molto dalla complessità delle trasformazioni, più complicata è la trasformazione più utile diventa il modello intermedio.

Quando sullo script si vogliono effettuare molte elaborazioni, c'è bisogno di un modo per controllare la complessità. Tecniche come le forward references possono essere un po' più complicate se le elaborazioni sono effettuate in unico passo. Con la Tree Construction, invece, è facile navigare l'albero molte volte nelle successive elaborazioni.

Un fattore che può spingere ad utilizzare la Tree Construction è se il Parser Generator fornisce o meno tool che permettono di costruire agilmente l'AST. Alcuni Parser Generator non danno scelta, bisogna usare la Tree Construction mentre altri danno la possibilità di utilizzare anche l'Embedded Translation, ma se il Parser Generator rende veramente semplice costruire un AST, l'uso della Tree Construction è preferibile.

La tree construction è probabile che occupi più memoria rispetto agli altri approcci, perchè ha bisogno di memorizzare l'AST. Nella maggior parte dei casi, comunque, non è una variazione apprezzabile.

E' possibile analizzare l'AST in modi diversi per popolare vari modelli semantici riutilizzando il parser. Ciò può essere comodo, ma se la costruzione dell'albero nel parser è semplice allora potrebbe essere più semplice utilizzare più AST ciascuno per uno scopo preciso. Potrebbe essere ancora meglio utilizzare il modello semantico come base per realizzare le altre rappresentazioni.

Indice delle figure

Figura 1: Architettitura generale di un IDE........ 13

Figura 2: Syntax highlighting di un sorgente Java in Eclipse 14

Figura 3: Esempio di autocompletamento........ 14

Figura 4: Archittettura generale di Eclipse........ 20

Figura 5: Relazione tra DSL e Semantic model ....

Figura 6: Esempio di un passo in un Wizard generato con WDL 30

Figura 7: Relazione tra Entity Type, Business Objects e Wizard........

Figura 8: Esempio di espressione con la Dot Notation 34

Figura 9: Esempio di espressione in Dot Notation con filtro 35

Figura 10 : Esempio di Foreign Code CSS in WDL........ 37

Figura 11: Grammatica BNF del linguaggio WDL in JavaCC........ 42

Figura 12: Uso dei predicati semantici in WDL 43

Figura 13: Fase di interpretazione in WDL ....

Figura 14: Architettura J2EE per l'applicativo ....

Figura 15: Piattaforma EJB 3 49

Figura 16 : Architettura logica del sistema........ 56

Figura 17: Soluzione basata su JavaScript code editor ....

Figura 18: Tabella per il confronto delle funzionalità tra Ace e CodeMirror 62

Figura 19: Analisi dei sorgenti di CodeMirror ....

Figura 20: Alcune classi di stile del file *CodeMirror.css*........ 65

Figura 21: Interazione tra WizardEditorMB e CodeMirror........

Figura 22: Il metodo *getLanguageKeywords()*........ 67

Figura 23: Il metodo *help()*........ 67

Figura 24: Il metodo *checkSyntax()* 68

Figura 25: Il metodo *Save()*........ 68

Figura 26:Aspetto della pagina JSP 70

Figura 27:Configurazione di CodeMirror 71

Figura 28: Creazione di un’ istanza dell’ editor CodeMirror 72

Figura 29: Archittetura a layer del modo ....

Figura 30: Funzione *register* per la memorizzazione di variabili nel contesto 78

Figura 31: Implementazione Comment/Uncomment........ 80

Figura 32:Soluzione basata sul framework Xtext 84

Bibliografia

[1] T. Teitelbaum e T. Reps, «The Cornell Program Synthesizer: A Syntax Directed Programming Environment,» *Communication of the ACM,* Settembre 1981.

[2] JetBrains, «IntelliJ Idea: Overview,» JetBrains, 2012. [Online]. Available: http://www.jetbrains.com/idea/.

[3] G. M. Ricci, «Html.it : Introduzione e breve storia di Visual Studio,» Gruppo Html, 2012. [Online]. Available: http://www.html.it/pag/18832/introduzione-e-breve-storia-di-visual-studio/.

[4] M. Fowler, Domain Specific Languages, Addison-Wesley Professional, 2010.

[5] Itemis, «About: Xtext,» Itemis, 2012. [Online]. Available: http://xtext.itemis.com/.

[6] L. Bettini, «An Eclipse-based IDE for Featherweight Java implemented in Xtext,» 2010.

[7] Reverso, «Ricerca: Wizard,» Softissimo, [Online]. Available: http://dizionario.reverso.net/. [Consultato il giorno 2012].

[8] i4C S.r.l, «Training Resources,» i4C S.r.l, 2012. [Online]. Available: http://web2.i4c.it/confluence/dashboard.action.

[9] N. Fanizzi, «Linguaggi di programmazione:Manuale JavaCC,» 10 Maggio 2012. [Online]. Available: http://lacam.di.uniba.it/~nico/corsi/lingpro/materiale/manualetto-javacc.pdf.

[10] G. Sicari, «Articoli: Introduzione a Enterprise JavaBeans (EJB),» 2009. [Online]. Available: http://www.giuseppesicari.it/articoli/introduzione-enterprise-javabeans-ejb/.

[11] Corriere della sera, «Dizionario di Italiano: Debugger,» RCS MediaGroup Spa, 2012. [Online]. Available: http://dizionari.corriere.it/dizionario_italiano/D/debugger.shtml. [Consultato il giorno 2012].

[12] Wikipedia, «Source Code Editor,» Wikimedia Foundation Inc, 2012. [Online]. Available: http://en.wikipedia.org/wiki/Source_code_editor.

E. Lodolo, «Tecnologie Web: Lezioni,» 2010. [Online]. Available:

13] http://lia.deis.unibo.it/Courses/TecnologieWeb0910/lezioni/3.01.JavaScript.pdf.

M. Haverbeke, «CodeMirror,» 2012. [Online]. Available: http://codemirror.net/.
14]