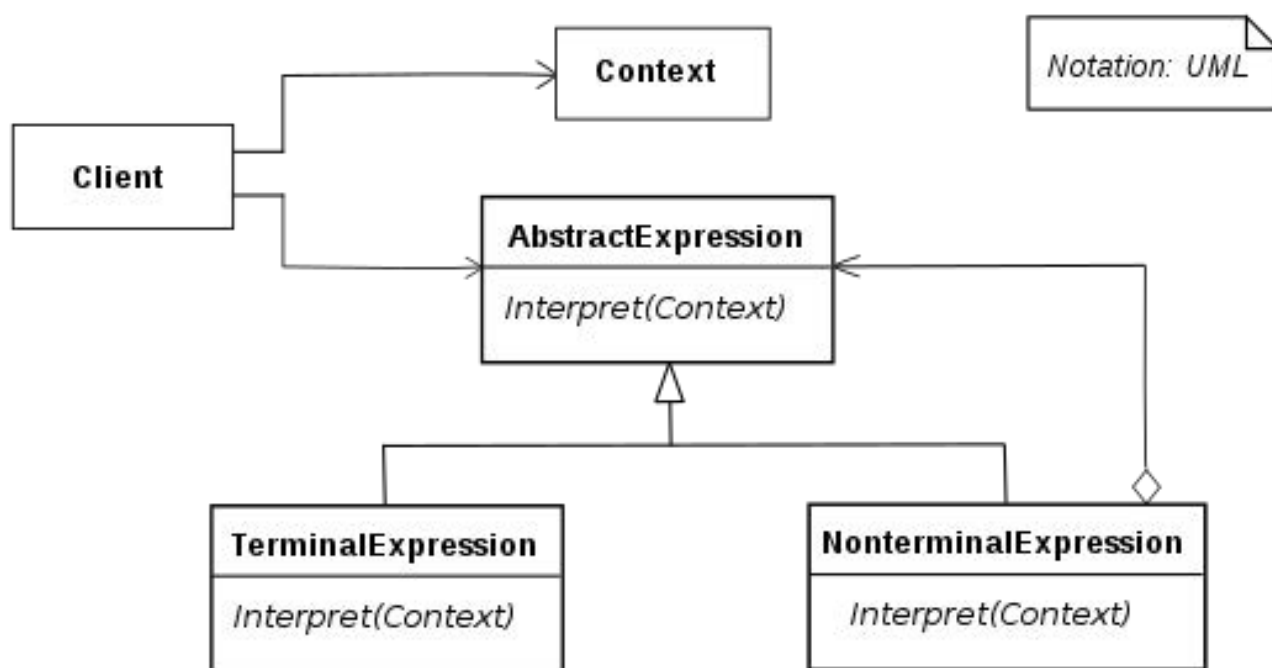


Progetto ingegneria del software

Anno Accademico 2016/2017

“Espressioni Condizionali”



Morelli Lorenzo
169153

Analisi dei requisiti

E' stato richiesto lo sviluppo di un'applicazione in grado di interpretare un'espressione condizionale che segua una data grammatica. Dopo aver costruito l'albero sintattico corrispondente all'espressione l'applicazione deve fornire la possibilità di poter fare una visita simmetrica e una postfissa dell'albero sintattico ed inoltre la valutazione di tale espressione. La valutazione è effettuata a partire da un contesto che contiene i valori delle variabili intere usate nelle espressioni aritmetiche. Inoltre l'applicazione deve permettere di salvare un contesto su un file per poter poi essere ripristinato e riutilizzato.

Le interazioni tra utente e sistema che consentono l'utilizzo di tali funzionalità sono definite in dettaglio dai seguenti casi d'uso

ID1	Costruzione Espressione Condizionale
Descrizione:	Permette all'utente di inserire un'espressione condizionale
Attore:	Utente
Svolgimento	1. L'utente inserisce l'espressione condizionale desiderata. 2. Il sistema analizza la correttezza grammaticale dell'espressione. 3a. Il sistema visualizza l'espressione immessa. 3b. L'espressione non è corretta e il sistema richiede nuovamente di immettere l'espressione.
Postcondizioni in caso di successo(a)	L'utente può ora inserire il contesto su cui valutare l'espressione.
Postcondizioni in caso di fallimento(b)	L'utente può reinserire una nuova espressione condizionale.

ID2	Creazione Contesto
Descrizione:	Permette all'utente di creare un contesto
Attore:	Utente
Svolgimento	<ol style="list-style-type: none"> 1. L'utente inserisce il nome del file .properties per popolare il contesto. 2a. Il contesto viene popolato. 2b. Il file non esiste o non è .properties pertanto il sistema chiede nuovamente di inserire il nome del file.
Postcondizioni in caso di successo(a)	L'utente può ora svolgere tutte le operazioni, inclusa la valutazione dell'espressione.
Postcondizioni in caso di fallimento(b)	L'utente può reinserire il nome del file.

ID3	Ripristino Contesto
Descrizione:	Permette all'utente di ripristinare un contesto.
Attore:	Utente
Precondizioni	Deve essere già stato creato il contesto
Svolgimento	<ol style="list-style-type: none"> 1. L'utente sceglie di ripristinare un contesto. 2. L'utente inserisce il nome del file testo da cui ripristinare un vecchio contesto. 3a. Il contesto viene ripristinato. 3b. Il file non esiste o non ha il formato corretto e il sistema chiede di immettere nuovamente il nome del file.
Postcondizioni in caso di successo(a)	L'utente può ora svolgere la valutazione dell'espressione con il nuovo contesto.
Postcondizioni in caso di fallimento(b)	L'utente può reinserire il nome del file oppure può uscire dal programma.

ID4	Salvataggio Contesto
Descrizione:	Permette all'utente di salvare il contesto.
Attore:	Utente
Precondizioni	Deve essere già stato creato il contesto
Svolgimento	<ol style="list-style-type: none"> 1. L'utente decide di salvare il contesto corrente. 2. L'utente inserisce il nome di un file testo in cui salvare il contesto. 3. Il sistema salva il contesto nel file desiderato.
Postcondizioni	L'utente può ripristinare il contesto salvato in questa o in una altra sessione di lavoro.

ID5	Valutazione Espressione
Descrizione:	Permette all'utente di valutare l'espressione.
Attore:	Utente
Precondizioni	Deve essere già stato creato il contesto ed immessa l'espressione
Svolgimento	<ol style="list-style-type: none"> 1. L'utente sceglie di valutare l'espressione corrente. 2. Il sistema valuta l'espressione sulla base del contesto corrente. 3. Il sistema mostra il risultato dell'espressione

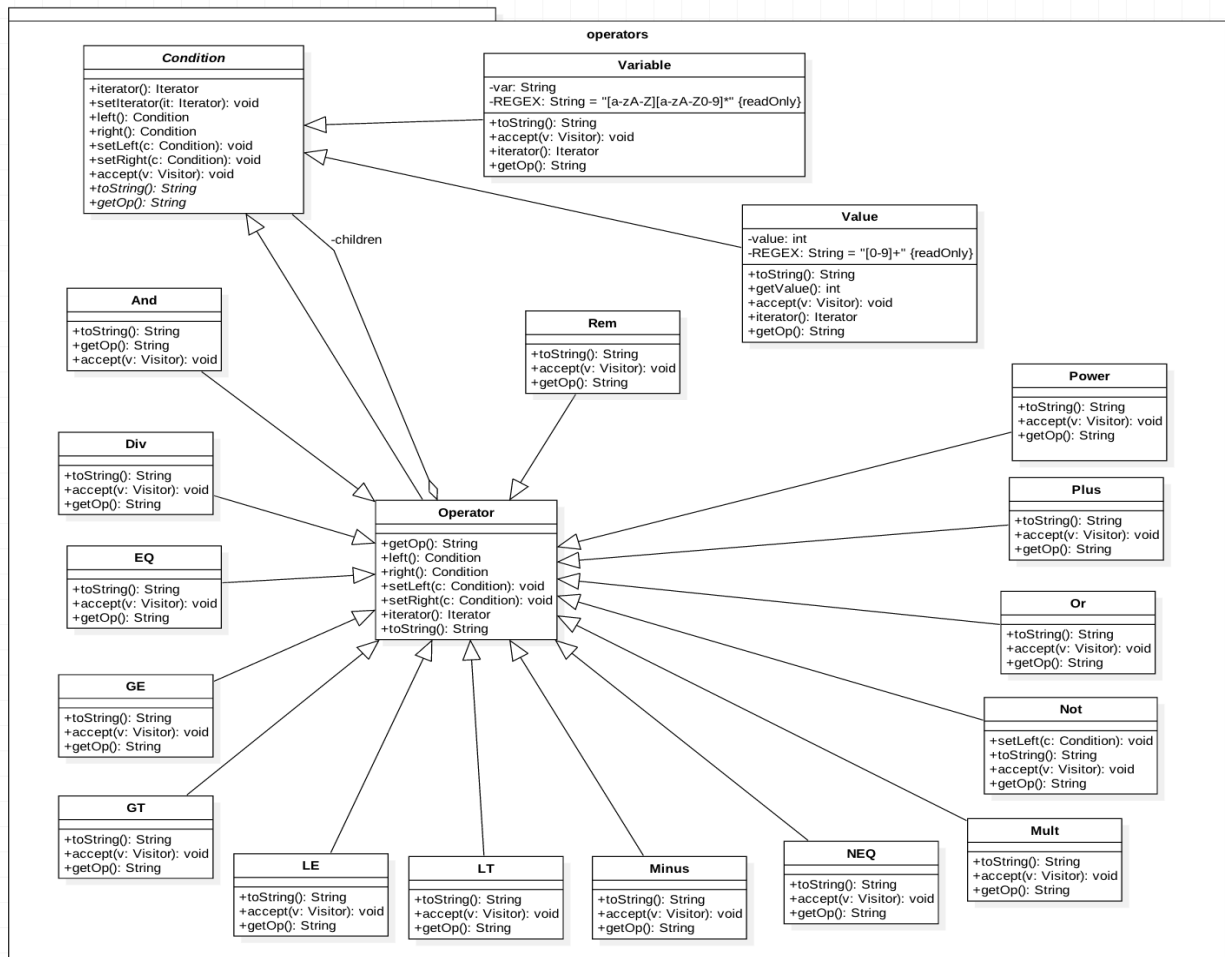
ID6	Visita Simmetrica
Descrizione:	Permette all'utente di fare una visita simmetrica.
Attore:	Utente
Precondizioni	Deve essere già stata immessa un'espressione valida
Svolgimento	<ol style="list-style-type: none"> 1. L'utente sceglie di fare una visita simmetrica dell'albero sintattico. 2. Il sistema mostra la visita simmetrica richiesta.

ID7	Visita Postfissa
Descrizione:	Permette all'utente di fare una visita postfissa.
Attore:	Utente
Precondizioni	Deve essere già stata immessa un'espressione valida
Svolgimento	<ol style="list-style-type: none"> 1. L'utente sceglie di fare una visita postfissa dell'albero sintattico. 2. Il sistema mostra la visita postfissa richiesta.

ID8	Modifica Espressione Condizionale
Descrizione:	Permette all'utente di modificare l'espressione condizionale
Attore:	Utente
Precondizioni	Deve essere già stata immessa un'espressione valida
Svolgimento	<ol style="list-style-type: none"> 1. L'utente decide di inserire una nuova espressione condizionale. 2. L'utente inserisce la nuova espressione condizionale desiderata. 3. Il sistema analizza la correttezza grammaticale dell'espressione. 3a. Il sistema visualizza l'espressione immessa. 3b. L'espressione non è corretta e il sistema richiede nuovamente di immettere l'espressione.
Postcondizioni in caso di successo(a)	L'utente può svolgere qualsiasi operazione sulla nuova espressione.
Postcondizioni in caso di fallimento(b)	L'utente può reinserire l'espressione condizionale.

Composite

Il package *operators* contiene le classi che rappresentano i singoli elementi utilizzati per la costruzione di un'espressione condizionale.



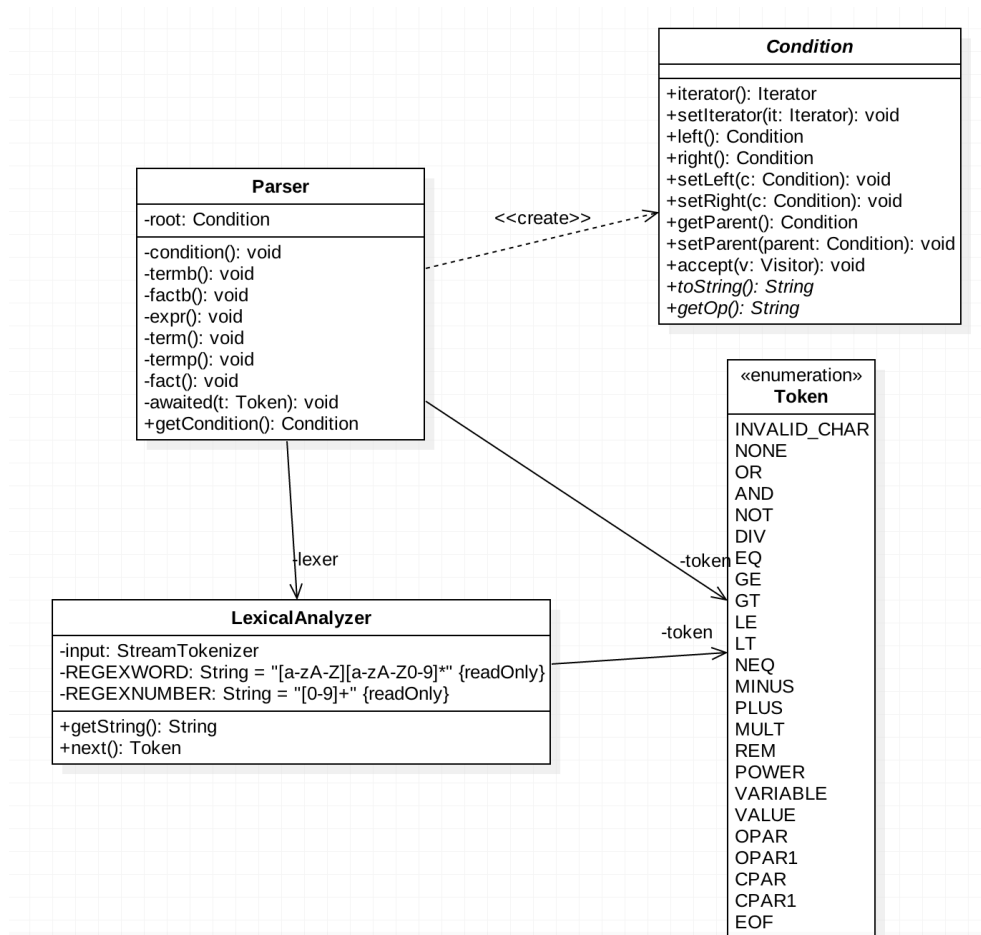
Dal diagramma delle classi è possibile osservare l'utilizzo del pattern *Composite* per definire i vari pezzi dell'espressione, che permette di trattare le classi composte e quelle non composte in maniera uniforme.

La classe *Operator* rappresenta l'oggetto composito che viene estesa da tutte quelle classi che rappresentano operatori aritmetici, di confronto o booleani. Queste classi mantengono un riferimento

ai propri operandi, ossia contengono a loro interno due oggetti di tipo *Condition*. Pertanto un “figlio” di un *Operator* potrebbe essere un altro *Operator*, oppure una *Variable* o un *Value*, ossia una variabile o un valore. Queste ultime due classi rappresentano le foglie dell’albero sintattico e non potranno avere figli. Infatti richiamando i metodi *right()* e *left()* su di essi verrà lanciata un’eccezione per come sono definiti nella classe *Condition*, che estendono direttamente. Al contrario la classe *Operator* ridefinisce tali metodi e quindi le sue classi eredi potranno gestire i propri figli. Infine è possibile notare all’interno delle classi *Variable* e *Value* una regex che verrà utilizzata per stabilire la correttezza sintattica dell’operando in base ai vincoli del linguaggio forniti nella descrizione del progetto mediante grammatica EBNF.

Package builder

Il package builder contiene le classi utili per la costruzione dell’albero sintattico. In particolare modo la classe responsabile della creazione è *Parser* che prende in ingresso nel suo costruttore la stringa, fornita dall’utente, relativa all’espressione condizionale. A partire da tale espressione il *Parser* istanzierà la radice, che è un oggetto *Condition* che mantiene al suo interno, e ne costruirà i figli e ricorsivamente fino ad arrivare ad oggetti *Variable* e *Value*. Per far ciò è stato applicato il pattern *Recursive Descent Parser Builder*, che è tipicamente utilizzato per la costruzione dell’albero sintattico. Attraverso l’utilizzo di tale pattern il Parser avrà sia le funzionalità del Director sia quelle del Builder.



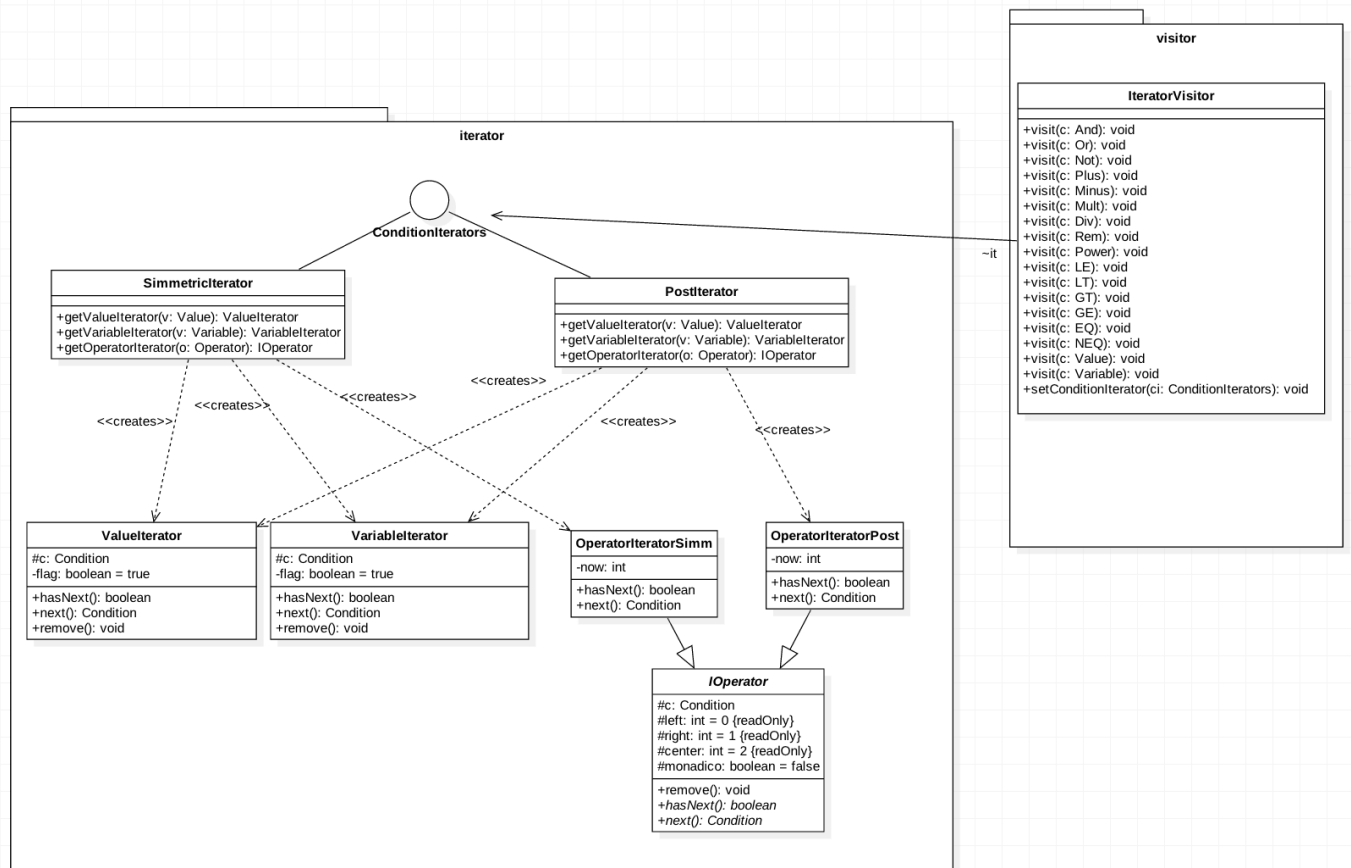
Per la costruzione dell'albero il Parser utilizza un oggetto della classe *LexicalAnalyzer* il quale utilizza uno *StreamTokenizer* per riconoscere i vari pezzi dell'espressione condizionale. Una volta riconosciuto un pezzo dell'espressione esso verrà associato ad un oggetto di *Token*, un'enumerazione le cui istanze rappresentano i vari pezzi di un'espressione, che è possibile fornire al Parser tramite il metodo `next()`. Se durante la costruzione dell'albero il Parser riscontra delle anomalie, ossia i pezzi dell'espressione fornita non rispettano i vincoli grammaticali, allora lancerà una *SyntaxException*.

Package visitor

Per introdurre delle funzionalità sulla espressione condizionale è stato utilizzato il pattern *Visitor*. Tale pattern permette di introdurre funzionalità future, non previste all'inizio della progettazione, senza modificare il codice delle classi su cui svolge le operazioni. Infatti all'interno della classe *Condition* non è visibile alcun metodo che svolge operazioni come la visita dell'albero o la sua valutazione. L'unico metodo visibile è *accept(Visitor)*, che richiama una specifica funzione in base allo specifico Visitor passato come parametro. Per tale motivo per aggiungere una funzionalità non occorre che creare una nuova classe che implementi l'interfaccia *Visitor*.

Iterazione e visite

Per quanto riguarda gli oggetti che compongono l'espressione condizionale, essi contengono al loro interno un oggetto di tipo *Iterator*. In tal modo si rende l'albero sintattico iterabile per consentire la navigazione su di esso a prescindere dalla sua particolare struttura. E' stato pertanto applicato il pattern *Iterator*, facendo in modo che l'accesso ai dati avvenga tramite un oggetto conforme all'interfaccia *Iterator* contenuto nello specifico oggetto *Condition*. In questo modo è possibile fornire più metodi di attraversamento della struttura cambiando il tipo di iteratore utilizzato.



Per settare l'iteratore ad un oggetto *Condition* è stato progettato un *Visitor* di installazione, *IteratorVisitor*. È possibile lasciare inalterato il visitor di installazione a patto di fornire allo stesso, in qualche modo, la famiglia di iteratori che si desidera applicare ai vari nodi della gerarchia composite. Si può pertanto utilizzare il pattern *AbstractFactory* per creare le famiglie di iteratori corrispondenti ad un assegnato criterio di iterazione. Su queste classi può quindi basarsi il visitor di installazione per completare il suo compito.

La classe *IteratorVisitor* avrà come istanza un oggetto conforme all'interfaccia *ConditionIterators* in modo tale da poter creare l'iteratore da settare nella struttura. Per settare un iteratore simmetrico utilizzeremo un oggetto *SimmetricIterator* mentre per uno postfissa l'oggetto *PostIterator*. Se si volesse dunque introdurre un nuovo tipo di attraversamento dell'albero basterà

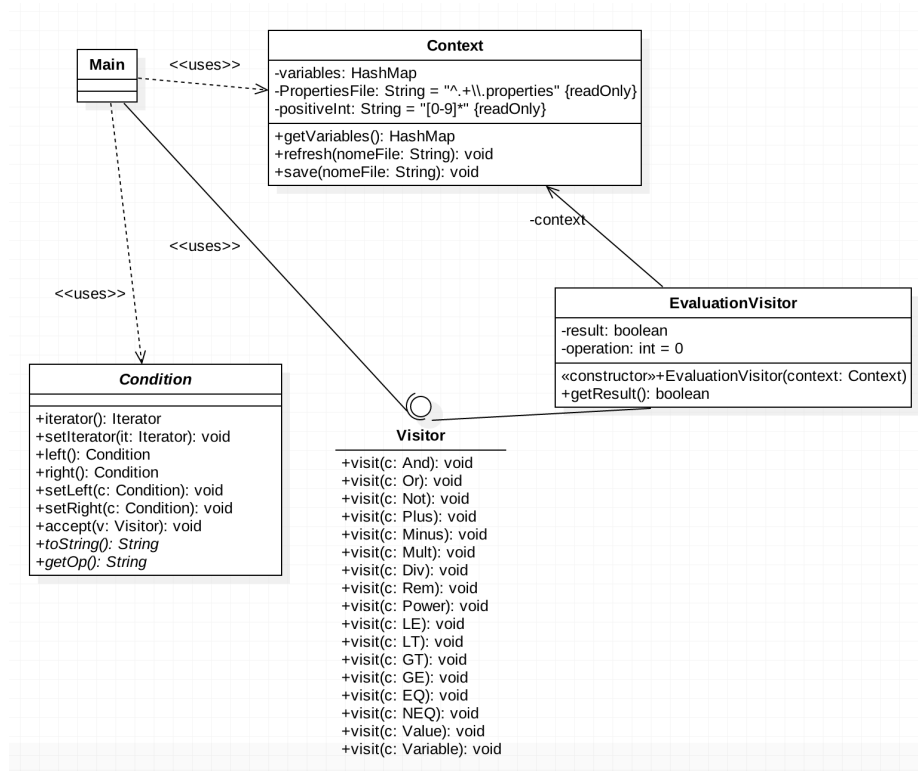
introdurre un oggetto conforme all'interfaccia *ConditionIterators* e poi settarlo al visitor di installazione.

Per quanto riguarda gli iteratori associati alle classi foglia, essi non cambiano poiché devono soltanto restituire il valore contenuto al suo interno. Ciò che invece cambia nel tipo di visita di un albero sono gli iteratori degli operatori. Per questo motivo è stata introdotta una classe astratta *IOperator* che potrà essere concretizzata da un *OperatorIteratorSimm* o da un *OperatorIteratorPost*. Nel primo caso la visita di tale iteratore è simmetrica per cui l'ordine con cui restituisce gli elementi *next()* sono figlio sinistro, radice, figlio destro, mentre nel secondo la visita è postfissa per cui l'ordine con cui restituisce gli elementi *next()* è figlio sinistro, figlio destro, radice. Per ottenere l'iteratore del nostro albero occorrerà richiamare il metodo *iterator()* sulla radice che restituirà l'iteratore che è stato settato in quel momento sull'albero tramite l'operazione *setConditionIterator* della classe *IteratorVisitor*.

Utilizzando dunque congiuntamente pattern iterator e pattern visitor possiamo cambiare il metodo di navigazione attraverso una semplice visita dell'albero. Questo vantaggio risulta evidente quando dobbiamo effettuare un'identica operazione con navigazioni di tipo diverso come la stampa dell'albero. Infatti per effettuare una stampa dell'albero è stato inserito il solo visitor *PrintVisitor* che si basa su un iteratore per prendere il successivo elemento da stampare. E' così possibile stampare il risultato di una vista simmetrica e di una postfissa semplicemente settando l'iteratore desiderato sull'albero sintattico.

Valutazione

Per effettuare la valutazione dell'albero sintattico è stato creato un visitor apposito, ossia *EvaluationVisitor*



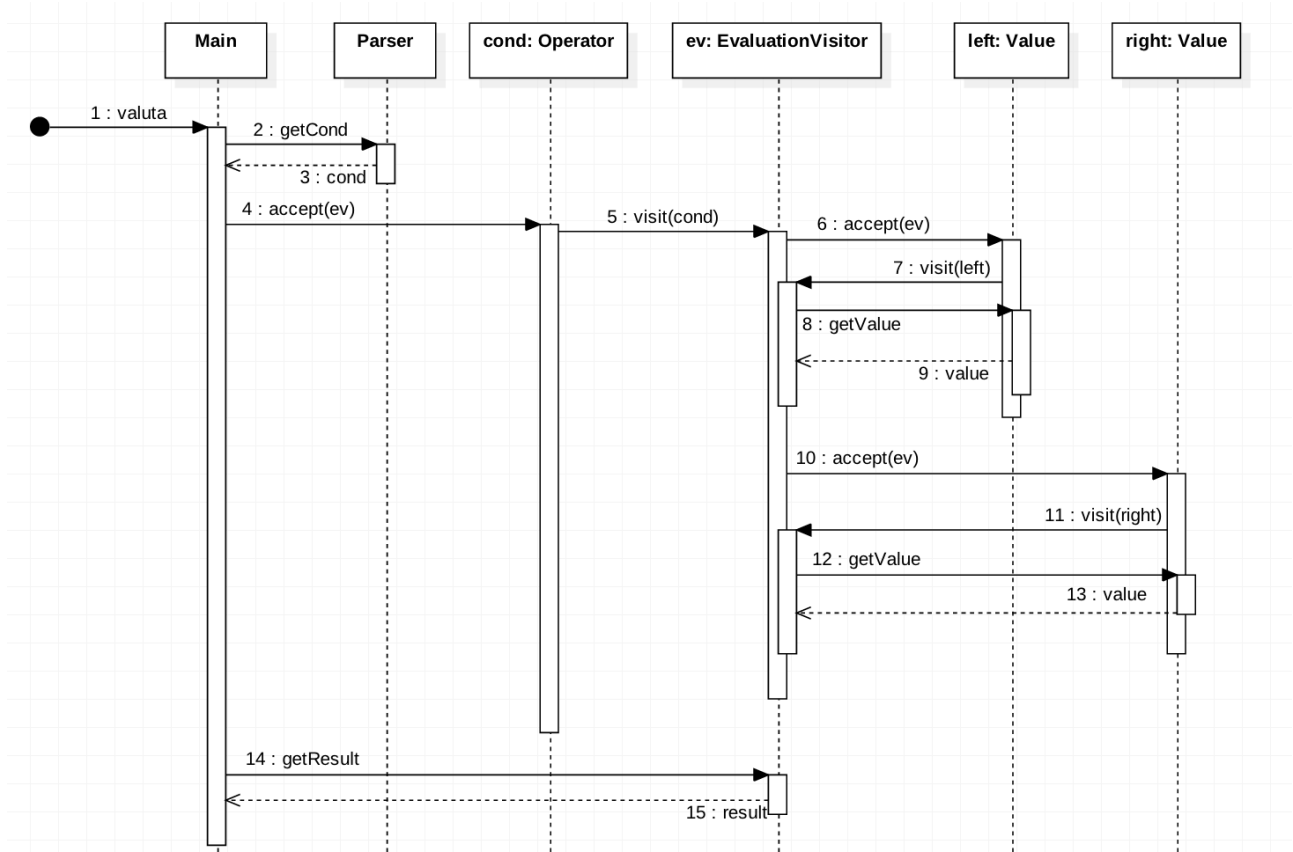
EvaluationVisitor fungerà da interprete per la grammatica definita in precedenza. Definire una grammatica ed il suo relativo interprete è l'obbiettivo che si pone il pattern *Interpreter* che è stato pertanto adottato nel progetto.

Come mostra il diagramma un oggetto *EvaluationVisitor* per essere istanziato ha bisogno di un oggetto *Context* che mantiene all'interno di un *HashMap* i nomi delle variabili e i valori ad esse associati. La classe *Context* mette a disposizione il metodo `refresh` per ripristinare un vecchio contesto da un file il cui nome è passato come parametro e il metodo `save` che permette di salvare il contesto in un file il cui nome è passato come parametro.

La valutazione dell'espressione condizionale avviene tramite una visita postfissa dell'albero, dove ricorsivamente vengono calcolati il risultato del sottoalbero sinistro e poi di quello destro. Quando si giunge alla foglie, se si ha un oggetto di tipo *Value* allora ne verrà restituito il suo valore, mentre se di tipo *Variable* si ricerca il nome della variabile dentro l'HashMap di Context, se è presente gli verrà assegnato quel valore, altrimenti il suo valore sarà 0.

E' infine possibile notare all'interno dei metodi *visit(And a)* e *visit(Or o)* di *EvaluationVisitor* è applicata la condizione di cortocircuito sugli operatori AND e OR facendo subito un controllo sul risultato ottenuto dalla valutazione del sottoalbero sinistro. Infatti se il risultato parziale nel caso della AND è false il metodo non avanza, stessa cosa per la OR se il risultato parziale è true.

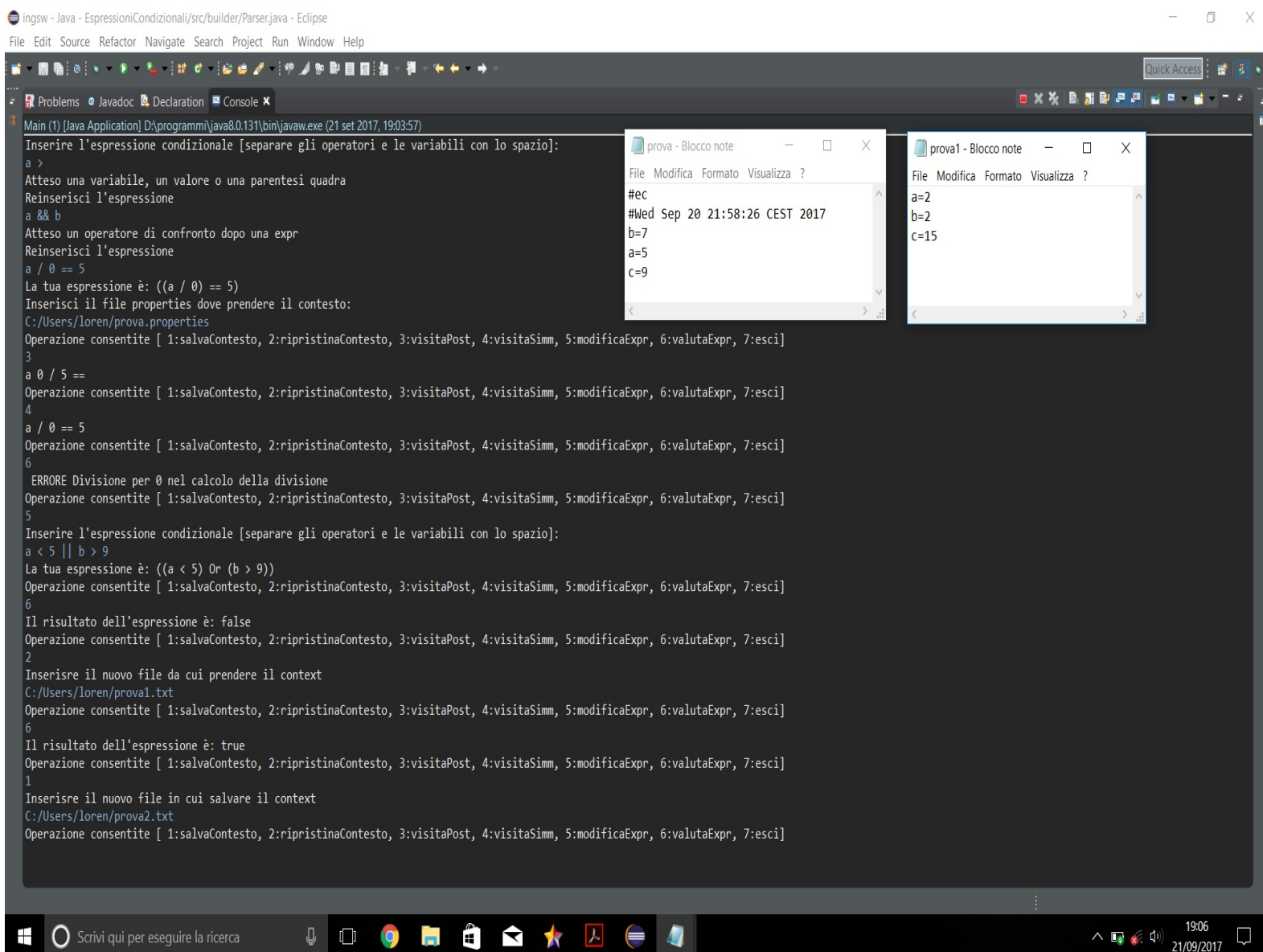
Di seguito è mostrato la valutazione di una semplice espressione del tipo valore, operatore, valore.



Main

E' stata inclusa nel progetto una classe di prova *Main* che permette di utilizzare le funzionalità sviluppate nel progetto. Una volta avviato chiede all'utente di inserire l'espressione condizionale desiderata, e nel caso in cui essa non rispetti le regole il sistema chiede di reinserirla. In seguito viene richiesto di inserire il nome del file per popolare il contesto. Nel caso in cui il file non sia .properties o non esiste il sistema chiede all'utente di reinserirlo. In seguito l'utente può interagire con il sistema scrivendo l'operazione che desidera fare.

Di seguito è mostrato un possibile funzionamento



```
ingsw - Java - EspressioniCondizionali/src/builder/Parser.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help

Main (1) [Java Application] D:\programmi\java8.0.131\bin\javaw.exe (21 set 2017, 19:03:57)
Inserire l'espressione condizionale [separare gli operatori e le variabili con lo spazio]:
a >
Atteso una variabile, un valore o una parentesi quadra
Reinserisci l'espressione
a && b
Atteso un operatore di confronto dopo una expr
Reinserisci l'espressione
a / 0 == 5
La tua espressione è: ((a / 0) == 5)
Inserisci il file properties dove prendere il contesto:
C:/Users/loren/prova.properties
Operazione consentite [ 1:salvaContesto, 2:ripristinaContesto, 3:visitaPost, 4:visitaSimm, 5:modificaExpr, 6:valutaExpr, 7:esci]
3
a 0 / 5 ==
Operazione consentite [ 1:salvaContesto, 2:ripristinaContesto, 3:visitaPost, 4:visitaSimm, 5:modificaExpr, 6:valutaExpr, 7:esci]
4
a / 0 == 5
Operazione consentite [ 1:salvaContesto, 2:ripristinaContesto, 3:visitaPost, 4:visitaSimm, 5:modificaExpr, 6:valutaExpr, 7:esci]
6
ERRORE Divisione per 0 nel calcolo della divisione
Operazione consentite [ 1:salvaContesto, 2:ripristinaContesto, 3:visitaPost, 4:visitaSimm, 5:modificaExpr, 6:valutaExpr, 7:esci]
5
Inserire l'espressione condizionale [separare gli operatori e le variabili con lo spazio]:
a < 5 || b > 9
La tua espressione è: ((a < 5) Or (b > 9))
Operazione consentite [ 1:salvaContesto, 2:ripristinaContesto, 3:visitaPost, 4:visitaSimm, 5:modificaExpr, 6:valutaExpr, 7:esci]
6
Il risultato dell'espressione è: false
Operazione consentite [ 1:salvaContesto, 2:ripristinaContesto, 3:visitaPost, 4:visitaSimm, 5:modificaExpr, 6:valutaExpr, 7:esci]
2
Inserire il nuovo file da cui prendere il context
C:/Users/loren/prova1.txt
Operazione consentite [ 1:salvaContesto, 2:ripristinaContesto, 3:visitaPost, 4:visitaSimm, 5:modificaExpr, 6:valutaExpr, 7:esci]
6
Il risultato dell'espressione è: true
Operazione consentite [ 1:salvaContesto, 2:ripristinaContesto, 3:visitaPost, 4:visitaSimm, 5:modificaExpr, 6:valutaExpr, 7:esci]
1
Inserire il nuovo file in cui salvare il context
C:/Users/loren/prova2.txt
Operazione consentite [ 1:salvaContesto, 2:ripristinaContesto, 3:visitaPost, 4:visitaSimm, 5:modificaExpr, 6:valutaExpr, 7:esci]
```

prova - Blocco note

```
File Modifica Formato Visualizza ?
#ec
#Wed Sep 20 21:58:26 CEST 2017
b=7
a=5
c=9
```

prova1 - Blocco note

```
File Modifica Formato Visualizza ?
a=2
b=2
c=15
```

Test

Sono stati effettuati alcuni test di tipo black box sui moduli *EvaluationVisitor*, *Parser*, *Value* e *Variable*.

La prima sequenza di test è definita nella classe *EvaluationVisitorTest* che definisce il metodo `@BeforeClass buildEvaluation()` che costruisce un file `.properties` con nessuno valore all'interno e istanzia un *EvaluationVisitor*. L'insieme dei possibili input, quindi delle possibili valutazioni di espressioni condizionali, è stato partizionato in due sottoinsiemi: il primo contiene espressioni condizionali valutabili, il secondo no per la presenza di operazioni non consentite.

Sono eseguiti sequenzialmente tre metodi di test. Il primo, `@Test positiveEvaluation()`, che verifica che il risultato dell'espressione sia true. Il secondo, `@Test CortoCircuitEvaluation()`, mostra che una divisione per zero sia correttamente corto circuitata. Il terzo, `@Test negativeEvaluation()`, mostra come sia sollevata un'eccezione nel caso di divisione per 0.

La seconda sequenza di test è definita nella classe *ParserTest*. L'insieme dei possibili input, quindi delle possibili espressioni condizionali, è stato partizionato in due sottoinsiemi: il primo contiene espressioni condizionali corrette, il secondo no.

Sono eseguiti sequenzialmente tre metodi di test. Il primo, *@Test positiveParser()*, che mostra una corretta costruzione dell'espressione. Il secondo, *@Test negativeParser()*, mostra un'espressione scorretta per l'assenza di una parentesi tonda finale. Il terzo, *@Test negativeParser2()*, mostra un'espressione scorretta per la presenza di due variabili vicine.

La terza sequenza di test è definita nella classe *ValueTest*. L'insieme dei possibili input, quindi dei possibili valori, è stato partizionato in due sottoinsiemi: il primo contiene valori corretti, il secondo no.

Sono eseguiti sequenzialmente due metodi di test. Il primo, *@Test positiveValue()*, che mostra che è stata rispettata la regola grammaticale. Il primo, *@Test negativeValue()*, che mostra che non è stata rispettata la regola grammaticale inserendo un carattere e non un valore.

La quarta sequenza di test è definita nella classe *VariableTest*. L'insieme dei possibili input, quindi dei possibili variabili, è stato partizionato in due sottoinsiemi: il primo contiene variabili corrette, il secondo no.

Sono eseguiti sequenzialmente due metodi di test. Il primo, *@Test positiveVariable()*, che mostra che è stata rispettata la regola grammaticale. Il primo, *@Test negativeVariable()*, che mostra che non è stata rispettata la regola grammaticale inserendo un numero all'inizio del nome della variabile.