Progetto Ingegneria del Software # 1 - A.A. 2018/2019: Espressione aritmetica intera con contesto

Mattia Gatto

Matricola:182926

III anno Ingegneria Informatica

Traccia

1. Espressione aritmetica intera con contesto

È assegnata la seguente grammatica EBNF di un “linguaggetto” per espressioni aritmetiche intere, in presenza delle usuali regole di precedenza degli operatori.

<espressione>::=<termine>{<addop><termine>}

<termine>::=<fattore>{<mulop><fattore>}

<fattore>::=<costante>|<variabile>|(<espressione>)

<costante>::=<interosenzasegno>

<variabile>::=<identificatore>

<interosenzasegno>::=<cifra>{<cifra>}

<identificatore>::=<lettera>{<lettera>|<cifra>}

<cifra>::=0|...|9

<lettera>::=a|…|z|A|…|Z

<addop>::=+|-

<mulop>::=*|/

Occorre realizzare un'applicazione, dotata di una minima interfaccia grafica, che includa almeno le seguenti funzionalità:

- valutazione dell'espressione rispetto ad un dato contesto (modificabile a tempo di esecuzione);
- sostituzione di una variabile con un'assegnata espressione per ottenerne una nuova;
- salvataggio e ripristino di un'espressione in un opportuno formato esterno.

Nello sviluppo del progetto si devono utilizzare i Design Pattern ritenuti più adeguati motivandone opportunamente la scelta. le fasi del processo di sviluppo devono essere documentate ricorrendo, ove necessario, all'uso di diagrammi UML.

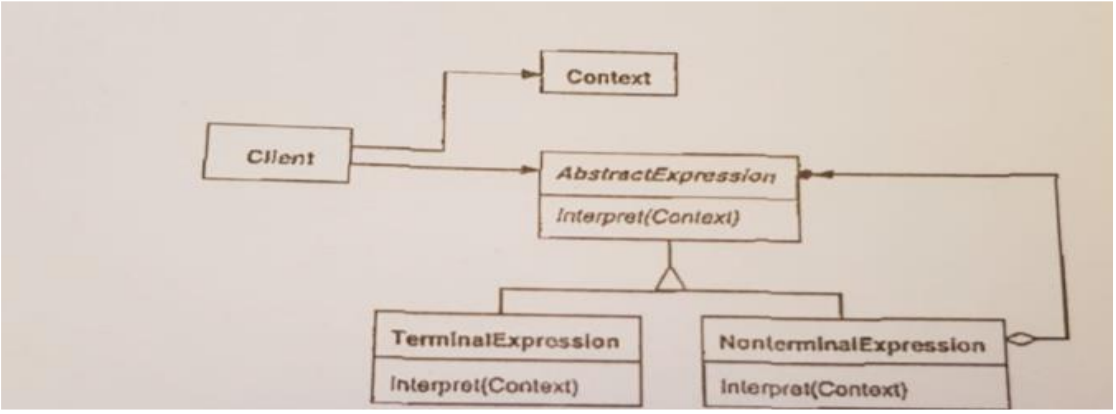
Il contesto mantiene coppie <nome_variabile,valore> in cui il nome di una variabile è associato al suo valore. Anche il contesto deve poter essere salvato e ripristinato dal file system ad es. utilizzando un formato basato su testo semplice o file properties o xml).

Si richiede inoltre di effettuare il testing di uno o più moduli significativi impiegando un opportuno criterio e sfruttando le funzionalità offerte dal framework JUnit.

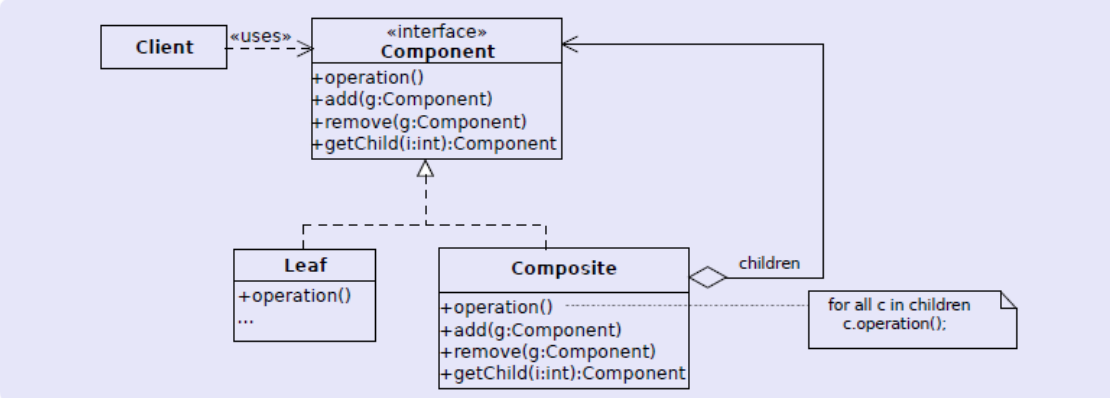
Relazione

Per la realizzazione del progetto ho fatto uso dei pattern:

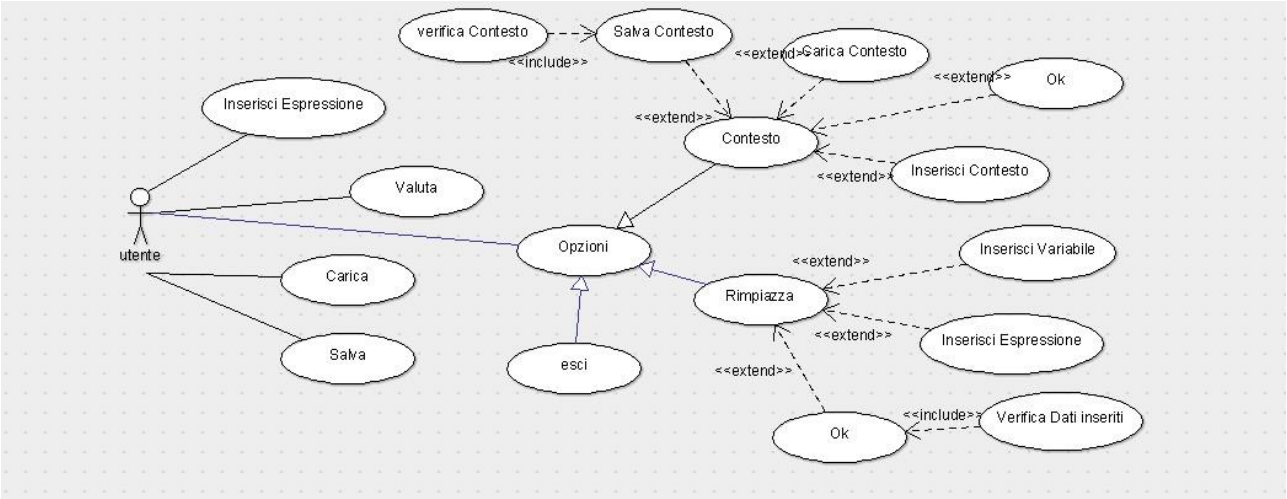
- Interpreter:
  Di norma interpreter è un pattern il cui scopo è di tipo comportamentale e il cui raggio d'azione è orientato alle classi, ossia usare l'ereditarietà per descrivere algoritmi e flussi di dati. Esso per natura si lega a strutture di tipo composite.
  Interpreter illustra come definire una grammatica per linguaggi semplici, rappresentare proposizioni nel linguaggio definito e interpretare queste proposizioni.
  Il pattern usa una classe per rappresentare ciascuna produzione della grammatica. I simboli nella parte destra sono attributi di queste classi.
  Il pattern Interpreter in generale è applicabile quando la grammatica è semplice (grammatiche difficili creerebbero gerarchie troppo grandi ed ingestibili) e quando l'efficienza non deve essere un aspetto cruciale (per avere una maggiore efficienza bisognerebbe tradurre l'albero sintattico in altre forme).
  La struttura quindi risulta essere così fatta:
  AbstractExpression: definisce un metodo astratto Interpret comune a tutti i nodi dell'albero sintattico;
  Terminal Expression: il metodo Interpret è associato a simboli terminali della grammatica;
  NonTerminalExpression: implementa il metodo Interpret con i simboli non terminali; solitamente chiama ricorsivamente se stesso; è necessaria una classe per ogni regola della grammatica.
  Context: Contiene le informazioni comuni all'interprete e la stringa da analizzare;
  Client: costruisce (o gli viene fornito) un albero sintattico astratto composto;



- **Composite**
  È un design pattern strutturale basato sugli oggetti. Ha lo scopo di comporre oggetti in strutture ad albero per rappresentare gerarchie "parte-tutto" e consentire ai client di trattare oggetti singoli e composizioni in modo uniforme. Possono esistere applicazioni (quali editor grafici o ambienti per la progettazione di circuiti) che consentono agli utenti di costruire diagrammi complessi a partire da semplici componenti. Componenti semplici possono essere raggruppati per costruire oggetti più complessi che a loro volta possono essere utilizzati come parti di componenti ancor più complessi. Solitamente vengono introdotte alcune classi per modellare gli oggetti semplici ed altre per rappresentare gli oggetti ottenuti per composizione. Il pattern Composite introduce un'interfaccia comune per gli oggetti semplici e per quelli compositi in modo che il codice cliente li possa trattare uniformemente.



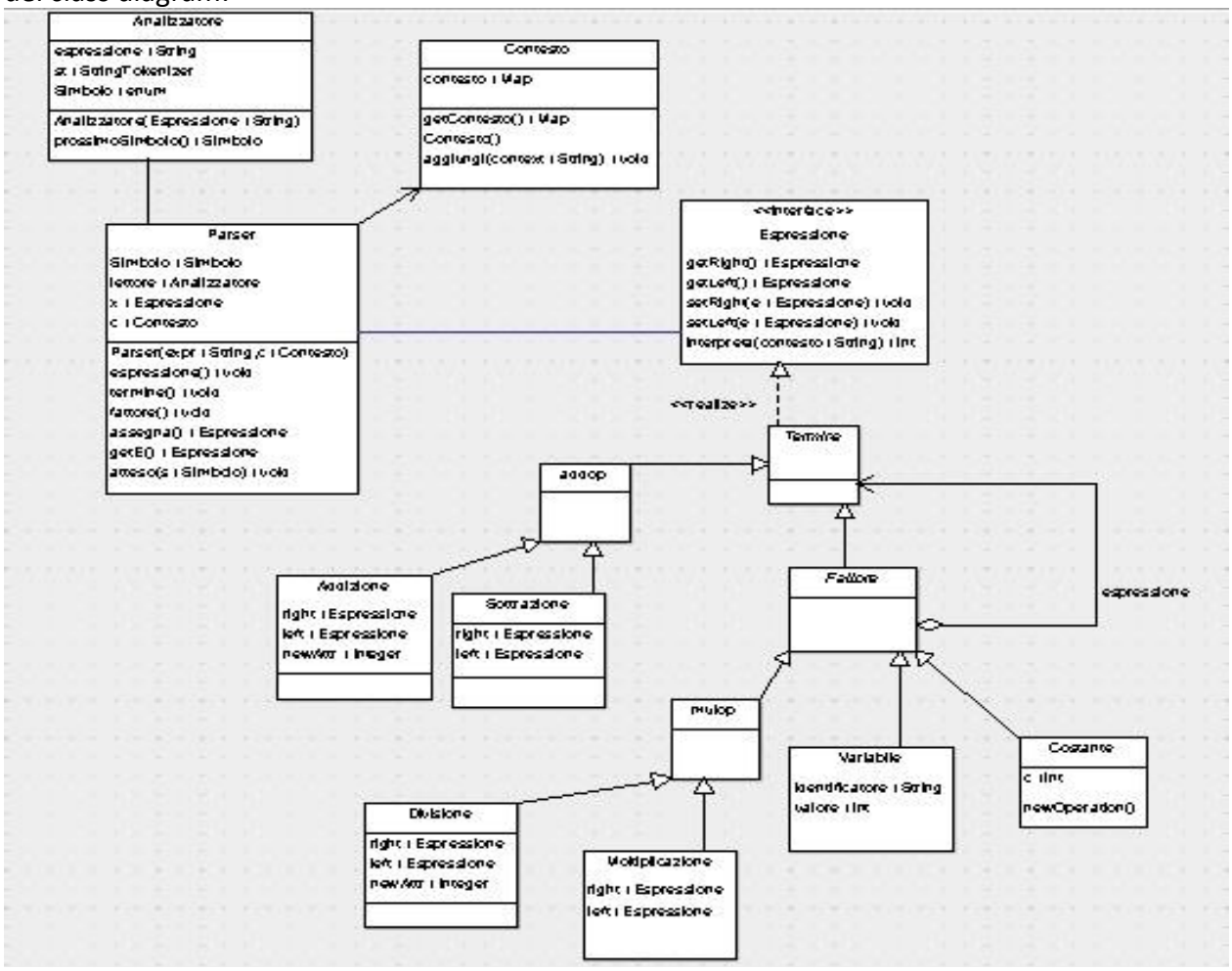
Per rappresentare la modalità di uso del programma faccio ricorso ad un caso d' uso ossia uno scenario generale che può avere l'utente con il programma attraverso l'interfaccia grafica sviluppata:



1. L'utente all'inizio si trova davanti un'interfaccia che presenta una casella di testo nella quale si può inserire un'espressione aritmetica intera, la quale può essere valutata attraverso la pressione del tasto valuta, il quale andrà a calcolare il risultato finale che sarà settato in una casella di testo non modificabile chiamata risultato;
2. Attraverso il tasto 'carica', il programma va nel contesto (un file expr.txt presente nel package) e carica un'espressione, precedentemente calcolata dal programma e salvata nel contesto attraverso la pressione del tasto 'salva', nella casella di testo 'espressione' e il suo corrispettivo risultato nella casella di testo 'risultato'.
3. Attraverso il tasto 'carica', il programma va nel contesto e salva l'espressione inserita nella casella nella casella di testo 'espressione' e il suo corrispettivo risultato nella casella di testo 'risultato'.
4. Attraverso il tasto opzioni è possibile accedere ad un menù che presenta tre opzioni:
    a. Esci: chiude l'applicazione.
    b. Contesto: apre un'interfaccia grafica che possiede a sua volta:
        i. Salva contesto: che salva in un file contesto (un file context.txt presente nel package) il testo inserito nella casella di testo 'Contesto' che deve essere del tipo (x=10 oppure x=5 s=9 ecc.).
        ii. Carica contesto: che carica da un file contesto (un file context.txt presente nel package) il testo del tipo (x=10 oppure x=5 s=9 ecc.) che sarà settato nella casella di testo.
        iii. Ok: permette di ritornare all' interfaccia originaria nella quale si potrà utilizzare la variabile del contesto al posto del valore corrispettivo per risolvere l'espressione;

c. Rimpiazza: apre un'interfaccia grafica che a sua volta possiede:
    i. Inserisci espressione: che prende l'espressione inserita e la sostituirà alla variabile inserita nel campo 'variabile' nel campo di testo 'espressione' nell' interfaccia principale subito dopo la pressione di 'ok'.
    ii. Inserisci variabile: che prende la variabile inserita che deve essere sostituita con l'espressione inserita nel capo di testo 'expr' nella casella di testo espressione nell' interfaccia principale subito dopo la pressione di 'ok'.
    iii. Ok: avvia i punti precedentemente accennati e ritorna all' interfaccia principale.

Ora parliamo di come sono state strutturate le classi all' interno del progetto attraverso l'utilizzo del class diagram:



Attraverso l'utilizzo della classe Parser è stato possibile riuscire a collegare i singoli componenti in modo tale che le classi appartenenti al linguaggio siano utilizzate dai metodi appartenenti alla classe Parser che rappresentano l'ordine giusto da utilizzare per svolgere le operazioni e attendere i rispettivi simboli nell' ordine giusto dove simboli sono i segni di addizione, sottrazione, moltiplicazione, divisione, parentesi tonda aperta e parentesi tonda chiusa.
Ecco qui il Parser e l'analizzatore:

```
package EAV;

import java.util.StringTokenizer;
public class Analizzatore {
```

```
	private String espressione;
	private StringTokenizer st;
	public enum Simbolo {
		PIU,MENO,PER,DIVISO,COSTANTE,VARIABILE,TONDA_A,TONDA_C,FINE;
		private String variabile;
		private int costante;
		Simbolo() {
			variabile = null;
			costante = 0;
		}
		public String getVariabile() {
			return variabile;
		}
		public void setVariabile(String s) {
			this.variabile = s;
		}
		public int getCostante() {
			return costante;
		}
		public void setCostante(int n) {
			this.costante = n;
		}
	}//simbolo
```

//qua mi sono creato un enum Simobolo che può essere uno di questi elencati sopra e 4 metodi per settare o prendere una variabile o una costante.

```
	public Analizzatore(String espressione) {
		this.espressione = espressione;
		st = new StringTokenizer(this.espressione,"()+-*/",true);
	}
	public Simbolo prossimoSimbolo() {
		String intero = "[\\d]+";
		String lettera = "[(a-z)|(A-Z)]+";
		String identificatore = lettera+"([\\d]*|[(a-z)|(A-Z]*)";
		if(!st.hasMoreTokens()) {
			return Simbolo.FINE;
		}
		String s = st.nextToken();
		if(s.equals("+")) { return Simbolo.PIU; }
		if(s.equals("-")) { return Simbolo.MENO; }
		if(s.equals("*")) { return Simbolo.PER; }
		if(s.equals("/")) { return Simbolo.DIVISO; }
		if(s.equals("(")) { return Simbolo.TONDA_A; }
		if(s.equals(")")) { return Simbolo.TONDA_C; }
		if(s.matches(intero)) {
				int c = Integer.parseInt(s);
				Simbolo simbolo = Simbolo.COSTANTE;
				simbolo.setCostante(c);
				return simbolo;
		}
		if(s.matches(identificatore)) {
			Simbolo simbolo = Simbolo.VARIABILE;
			simbolo.setVariabile(s);
			return simbolo;
		}
		return Simbolo.FINE;
	}
}//AnalizzatoreLessicale
```

//Prossimo simbolo è un metodo che ho creato con lo scopo di riuscire a capire quale sia il prossimo simbolo cioè se sia un segno o una parentesi o un intero o una costante, se intero o costante settiamo anche a che valore corrispondono.
Ed ecco il parser:

```
package EAV;
import EAV.Analizzatore.Simbolo;
public class Parser{
	private Simbolo simbolo;
```

```
private Analizzatore lettore;
private Espressione x = null;
private Contesto c = new Contesto();
public Parser(String expr, Contesto c) {
        lettore = new Analizzatore(expr);
        this.c = c;
        espressione();
}
```

//Il costruttore una volta passato un'espressione ed un contesto io passo automaticamente il metodo espressione che attraverso la struttura sotto riportata va a risolvere l'espressione.

```
public void espressione() {
        termine();
        while(simbolo == Analizzatore.Simbolo.PIU ||simbolo == Analizzatore.Simbolo.MENO) {
                Termine t = (Termine) assegna();
                t.setLeft(x);
                termine();
                t.setRight(x);
                x = t;
        }
}
```

//Espressione come esposta dal 'linguaggetto' è composta da

<espressione>::=<termine>{<addop><termine>}

infatti richiamo il metodo termine e subito dopo attendo fino a quando non finiscono i simboli di addizione e sottrazione, nel contempo prendo il termine t che sarà una classe concreta di espressione(o add o molt o div o sott), imposto come parte sinistra di t uguale a x che è la corrente espressione, applico di nuovo il metodo termine e alla fine di questo imposto la parte destra di t uguale a x e infine imposto x=t, dove x è l ' espressione che alla fine sarà uguale ad un numero intero .

```
public void termine() {
        fattore();
        while(simbolo == Analizzatore.Simbolo.PER || simbolo == Analizzatore.Simbolo.DIVISO) {
                Fattore f = (Fattore) assegna();
                f.setLeft(x);
                fattore();
                f.setRight(x);
                x = f;
        }
}
```

//Termine come esposta dal 'linguaggetto' è composta da

<termine>::=<fattore>{<mulop><fattore>}

infatti richiamo il metodo fattore e subito dopo attendo fino a quando non finiscono i simboli di moltiplicazione e divisione, nel contempo prendo il fattore f che sarà una classe concreta di espressione(o add o molt o div o sott), imposto come parte sinistra di f uguale a x che è la corrente espressione, applico di nuovo il metodo fattore e alla fine di questo imposto la parte destra di f uguale a x e infine imposto x=f, dove x è l ' espressione che alla fine sarà uguale ad un numero intero .

```
public void fattore() {
        simbolo = lettore.prossimoSimbolo();
        if(simbolo == Analizzatore.Simbolo.COSTANTE) {
                x = new Costante(simbolo.getCostante());
                simbolo = lettore.prossimoSimbolo();
        }
        else if(simbolo == Analizzatore.Simbolo.VARIABILE) {
```

```
                    x = new Variabile(simbolo.getVariabile(),c.getContesto().get(simbolo.getVariabile()));
                    simbolo = lettore.prossimoSimbolo();
                }
                else if(simbolo == Analizzatore.Simbolo.TONDA_A) {
                    espressione();
                    atteso(Analizzatore.Simbolo.TONDA_C);
                }
                else {
                    throw new RuntimeException("Valore in ingresso inatteso!");
                }
        }
```

//Fattore come esposta dal 'linguaggetto' è composta da

<fattore>::=<costante>|<variabile>|(<espressione>)

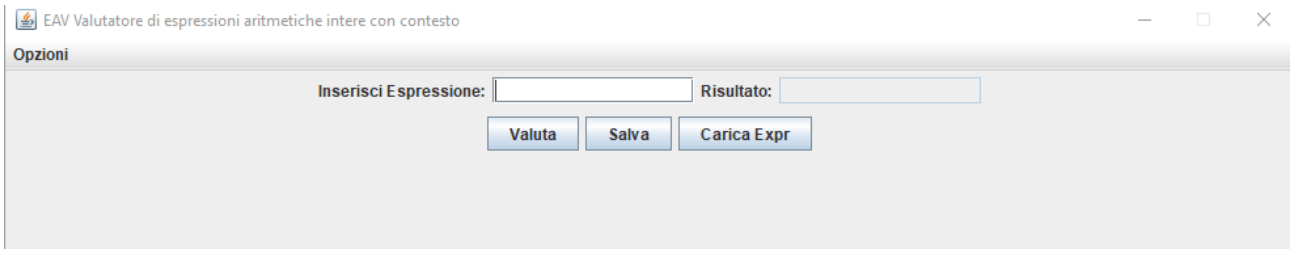
Infatti, prendo per prima cosa il prossimo simbolo corrente attraverso l'analizzatore 'lettore' e subito dopo controllo se esso sia uguale a:

- Una costante: allora imposto x che è l'espressione corrente uguale ad una nuova costante che corrisponde al valore intero dell'espressione, e aggiorno il prossimo simbolo.
- Una variabile: allora imposto x che è l'espressione corrente uguale ad una nuova variabile, della quale viene preso il valore dal contesto che corrisponde ad un valore intero che il valore dell'espressione, e aggiorno il prossimo simbolo.
- Una tonda aperta richiamo espressione perchè all' interno della tonda si trova una nuova espressione da risolvere e attendo il simbolo della parentesi tonda chiusa perché mi darà il risultato dell'espressione tra le parentesi e continua con il prossimo simbolo.
- Altrimenti è un'espressione malformata.

```
    private Espressione assegna() {
        switch(simbolo) {
            case PIU:
                return new Addizione();
            case MENO:
                return new Sottrazione();
            case PER:
                return new Moltiplicazione();
            case DIVISO:
                return new Divisione();
            default: return null;
        }
    }
    public Espressione getE() {
            return x;
    }
    private void atteso(Simbolo s) {
            if(simbolo != s) {
                throw new RuntimeException();
            }
            simbolo = lettore.prossimoSimbolo();
    }
}//Parser
```

Il metodo attendi è usato per la parentesi e fino a quando non si trova la parentesi chiusa di aggiorna il simbolo uguale al prossimo simbolo.

L'interfaccia appare in questo modo:


EAV Valutatore di espressioni aritmetiche intere con contesto
Opzioni
Inserisci Espressione:
Risultato:
Valuta
Salva
Carica Expr