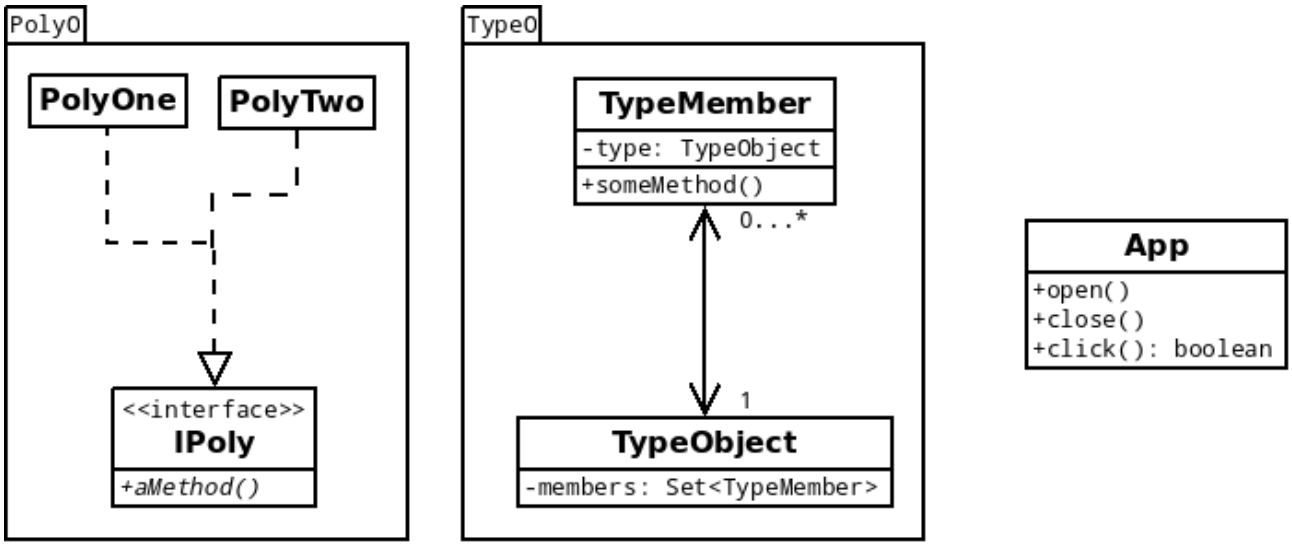
DESIGN PATTERNS – Parte 5

Facade
Strategy
Composite
Interpreter

FACADE

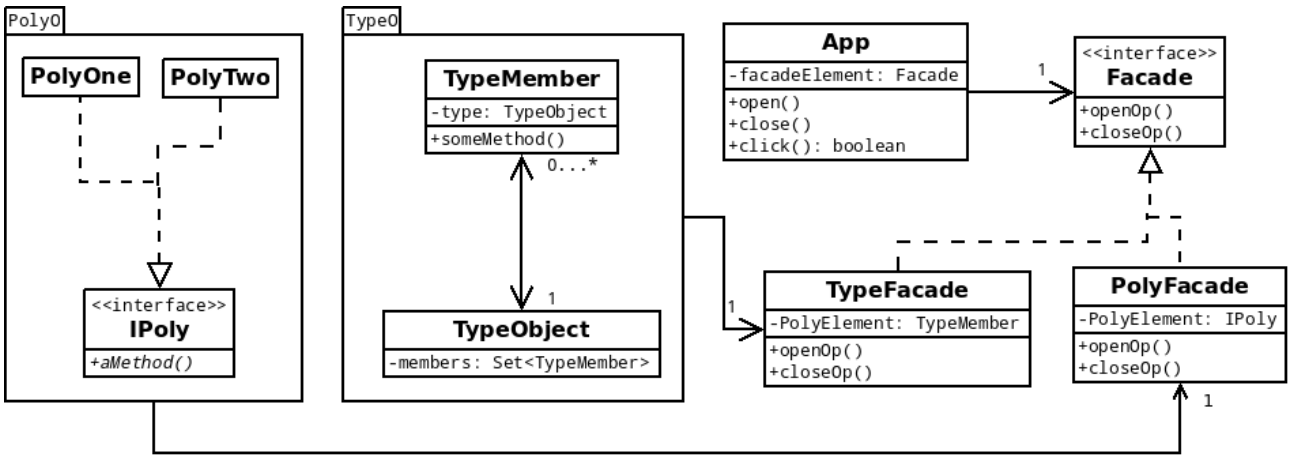
Per proteggere la nostra applicazione sfruttando il Protect Variation possiamo usare il pattern Facade. Questo pattern è ottimo per implementare nuove interfacce sperimentabili... facciamo un esempio:
ho una classe che gestisce dei prodotti, non sappiamo ancora se usare il polimorfismo o il type object. Il Facade farà quindi come uno switch e potremmo sfruttarlo per fare varie operazioni da vari package come se fosse uno solo senza dover modificare tutta la classe (in questo somiglia molto all'Adapter. Si ricorda che l'Adapter è utilizzato più per quando si ha una situazione client-server).

Prendiamo quindi di avere il seguente diagramma delle classi, e vogliamo testare la nostra applicazione App con i package TypeO e PolyO.



Come possiamo vedere i due package sono configurati in maniera differente e non sembra esserci modo perchè App possa usarli a propria scelta se non modificando il comportamento dei metodi di App.

Una soluzione è usare Facade che farà da interfaccia per i due package. Questa interfaccia userà dei nomi standard per identificare i metodi e delegherà l'uso dei metodi corripsondenti alle classi dei due package.



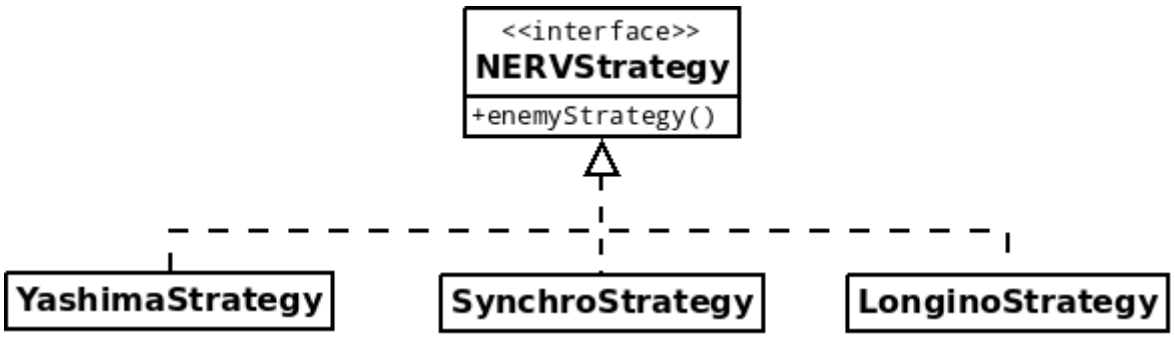
La nostra interfaccia Facade ci farà da tramite per i due package implementando due versioni differenti dei metodi che interesserebbe ad App usare. Questo significa che tutto ciò che ci sarà da modificare nell'applicazione è il tipo di creazione dell'oggetto Facade e niente di più.

STRATEGY

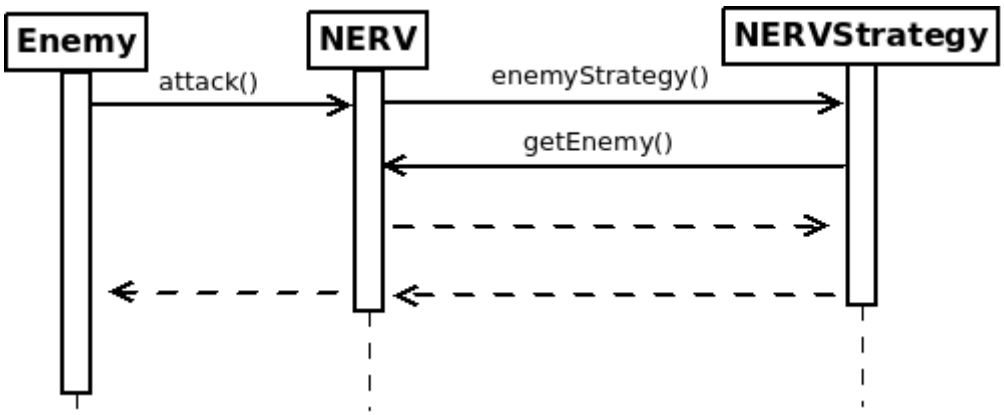
Questo pattern nasce dall'esigenza di decidere a chi assegnare delle responsabilità in termini di algoritmi agli oggetti che si ritiene più adatti.
Fingiamo di avere il seguente caso:
abbiamo un classe “NERV” che a variazione di attacchi da parte dei nemici userà strategie diverse, per esempio:

- in caso di attacco da parte di cubi giganti userà il piano Yashima
- in caso di attacco di esseri doppi userà il piano Synchro
- in caso di attacco di esseri spaziali userà il piano Longino

e via discorrendo se ci sono altri piani.
Vediamo dunque il diagramma delle classi come sarà composto:



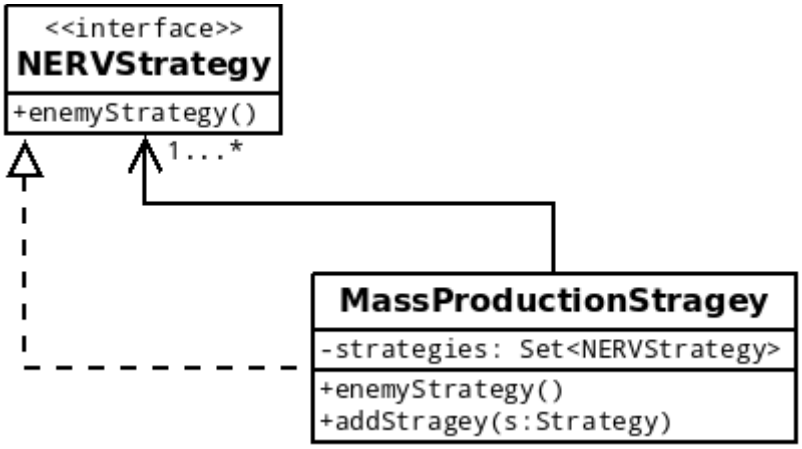
Per comprendere meglio l'effettivo uso dell'interfaccia NERVStrategy vediamo un diagramma di sequenza, nel quale avremmo una classe Enemy che utilizzerà il metodo attack() sulla classe NERV che invocherà quindi il metodo enemyStrategy() invocato dalla corrispondente Strategy adatta.



Una tecnica utile è quella di sfruttrare nell'oggetto Strategy i pattern Factory e Singleton.

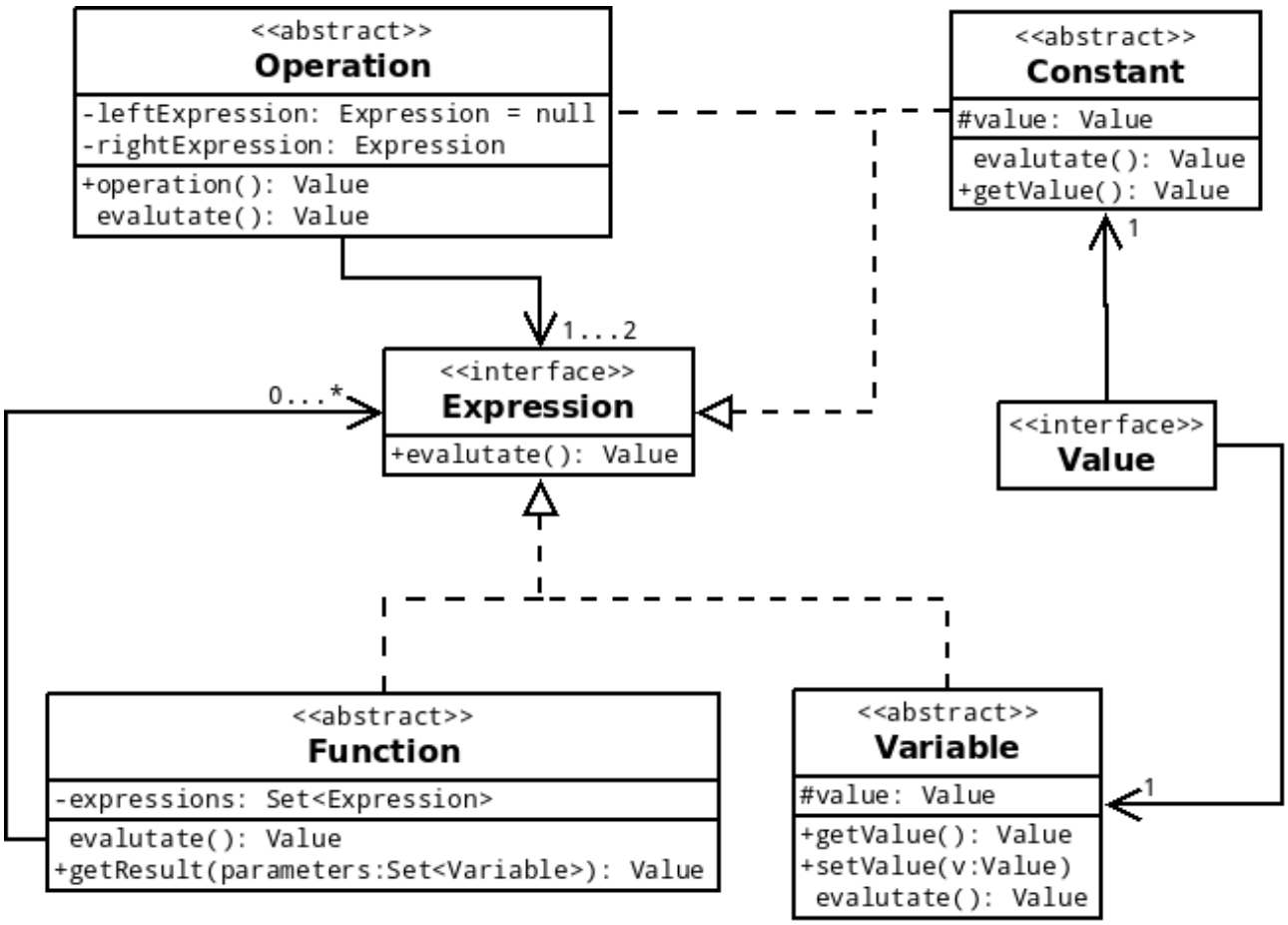
COMPOSITE

Il Composite pattern nasce per risolvere il problema del Strategy che è applicabile per un'unico caso. Se avessimo più casi applicabili nello stesso momento e vorremmo applicare quello più adatto?
Il pattern Composite nasce per questo motivo. Vediamo da subito il diagramma della classi con applicato il CompositePattern per una nuova strategy che chiameremo MassProductionStrategy.



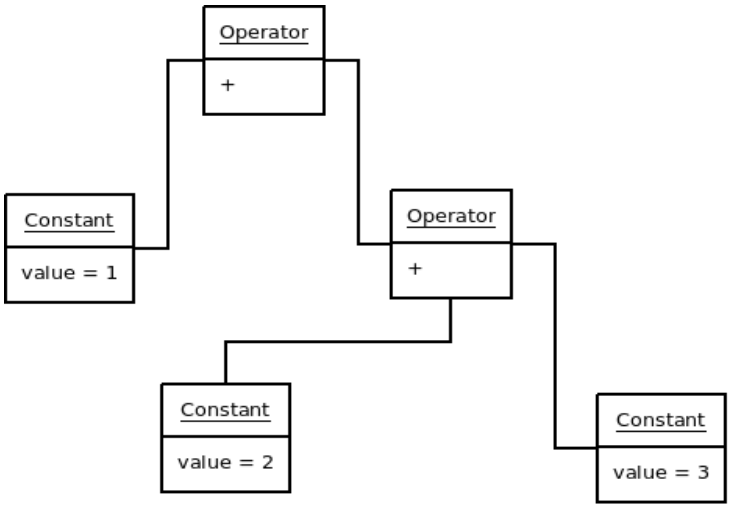
INTERPRETER

Interpreter è un pattern che nasce dall'esigenza di poter “creare” un proprio piccolo linguaggio all'interno della propria stessa applicazione. Lo si può usare implementando una struttura ad albero, detto albero sintattico, dove le radici sono gli operatori e i figli le espressioni di questi operatori.. o uno stack. Noi vedremmo il primo caso.

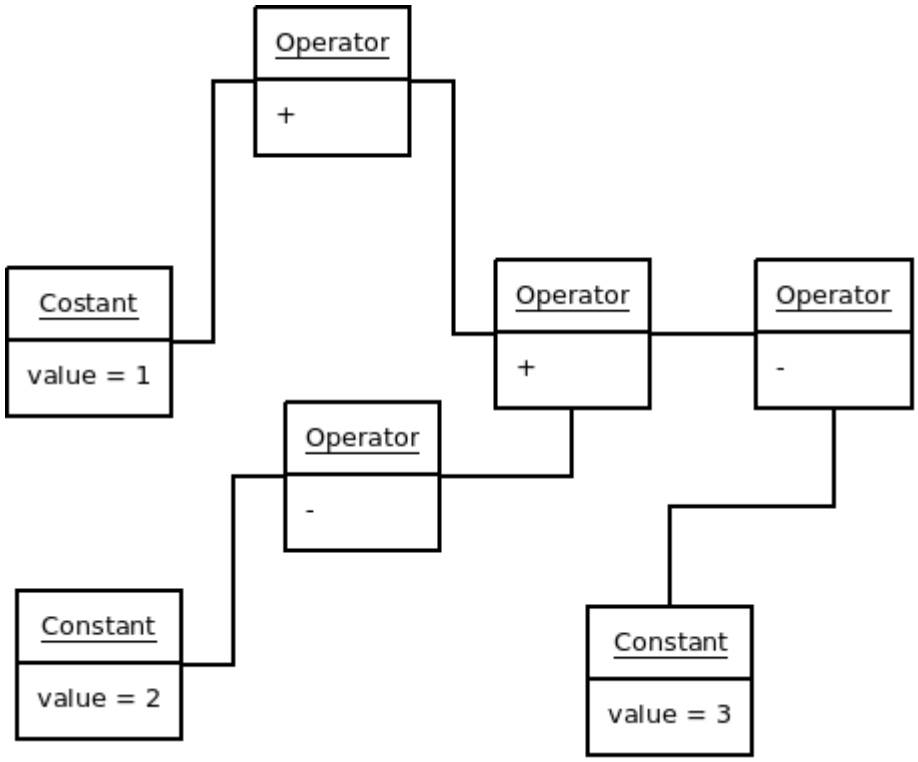


Il nostro albero sintattico sarà quindi fatto delle seguenti componenti.
Ricordiamo che per ognunga di queste componenti, essendo tutte astratte saranno implementabili nel modo che il programmatore meglio vorrà, noi ci limitiamo a mostrare ciò che devono avere di minimo.
Per comprendere come creare l'albero sintattico vediamo alcuni esempi utilizzando il diagramma degli oggetti per alcune espressioni.

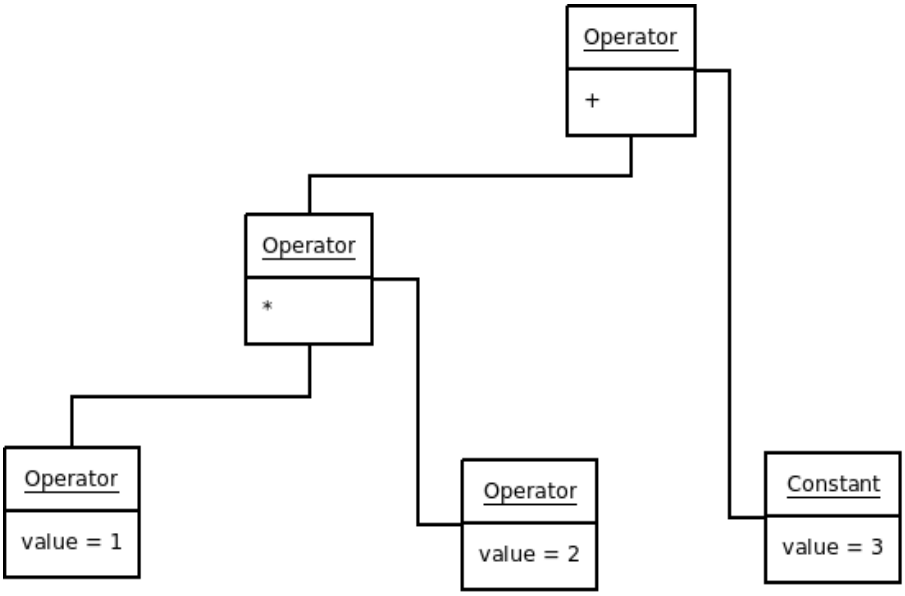
ESPR_1: 1+2+3
Con l'operatore somma useremo una lettura da sinistra verso destra.



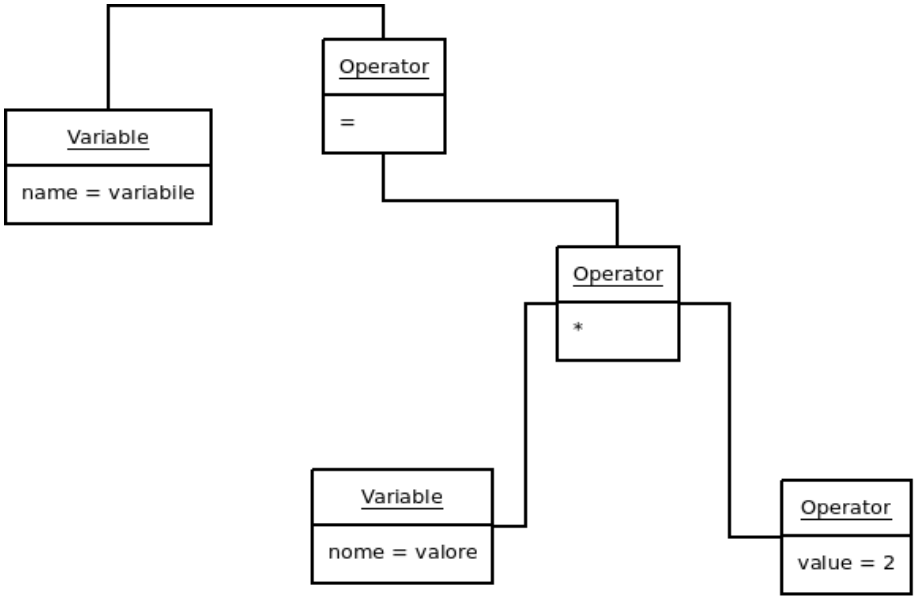
ESPR_2: 1-2-3
Attenzione! Per questa espressione possiamo valutarla in due modi:
in maniera classica, quindi come operatore binario -, e quindi saremmo costretti a usare una lettura da destra verso sinistra.
Oppure vedendo il – come operatore unario, ovvero come 1+(-2)+(-3).
Noi useremo la seconda opzione.



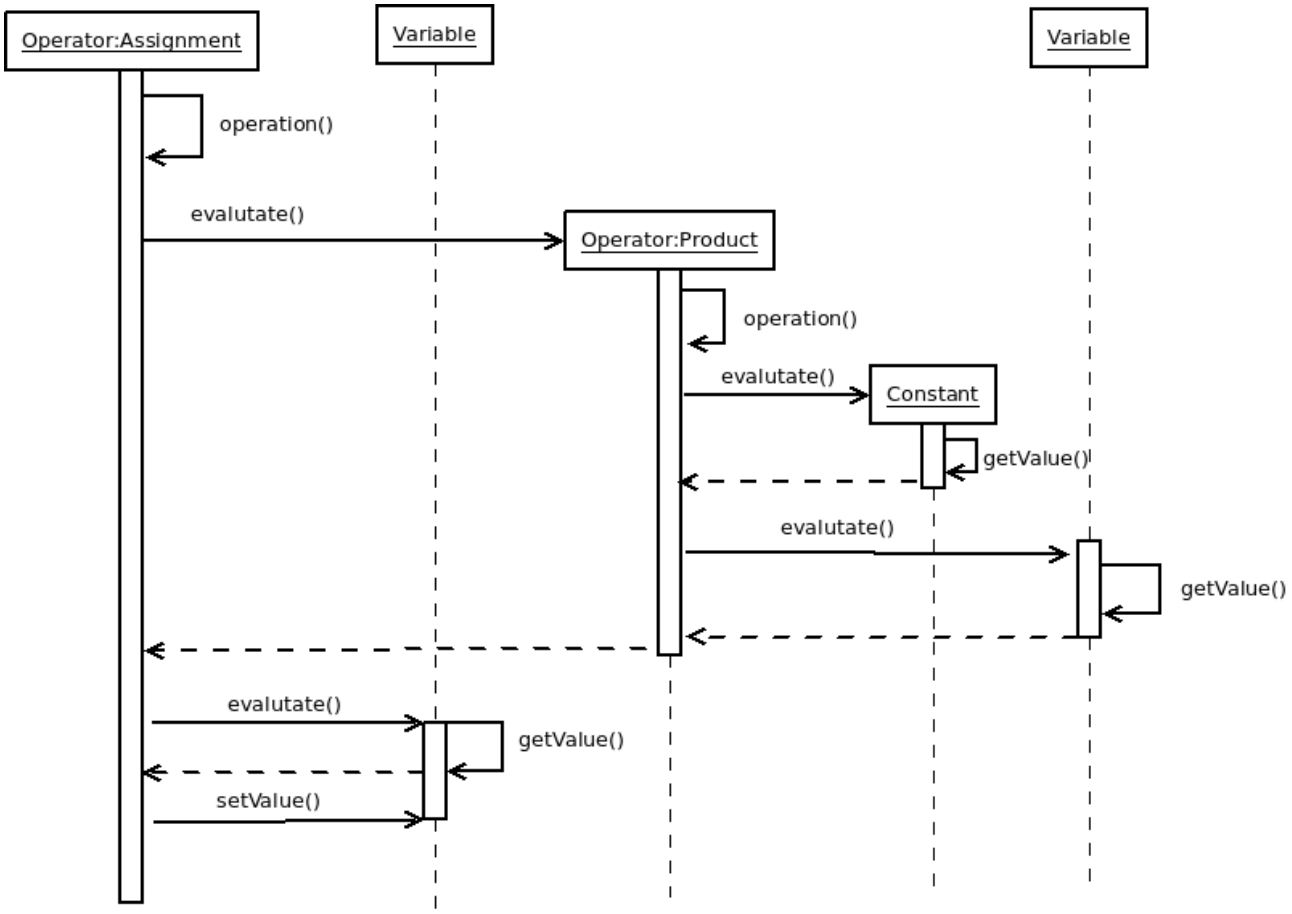
ESPR_3: 1*2+3
Qui invece dobbiamo lavorare sulla priorità degli operatori. L'operatore prodotto sarà ovviamente più prioritario dell'operatore +.



ESPR_4: variabile = valore*2
Per comprendere meglio il funzionamento delle cose in questo caso non ci avalremmo solo del diagramma degli oggetti ma anche di un diagramma di sequenza.

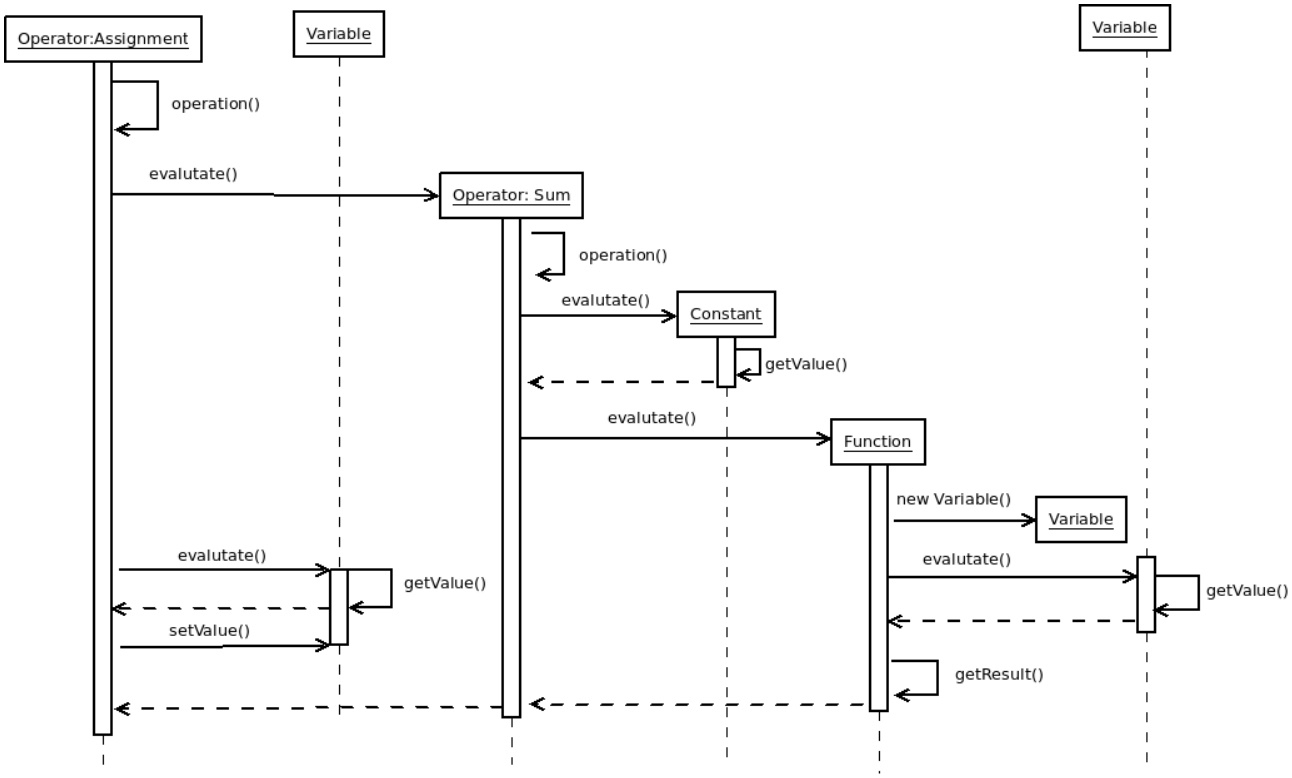


Ed il diagramma di sequenza corrispondente:



ESPR_5: variabile = fun(bar, 2)+3

Similmente a prima anche la funzione fun farà le stesse cose, se non con l'aggiunta di un passaggio di parametri sulla parte della funzione.
Questa volta ci limitamo al solo diagramma di sequenza:



Ora vedremo il codice dei vari componenti per una maggior comprensione degli schemi. Prima di vedere il codice si ricorda perché l'esempio sia completo abbiamo bisogno di una classe Parser che per l'appunto parserà i sorgenti e quindi valuterà le varie espressioni ottenendo i valori necessari. Per motivi di leggibilità abbiamo lasciato stare l'implementazione di questa classe e mostreremo solo una classe astratta coi metodi che essa deve avere per essere usata.

```
public abstract class Parser {
     protected String source; protected Parser self = null;
     protected Parser(String source) {
          this.source = source;
     }
     public Parser getParser(String s) {
          if(self == null) self = new Parser(s); return self;
     }
     //legge un carattere
     public abstract char readChar();
     //legge una espressione
     public abstract String readExpression();
     //salta gli spazi
     public abstract void skipSpaces();
}
```

La classe Parser come vediamo è stata fatta in modo che sia una singleton. I suoi metodi principali sono quelli di lettura di un'espressione, di un carattere e di saltare gli spazi bianchi. Cominciamo dunque a vedere pian piano gli attori, ovvero le classi che compongono il nostro interprete.
Interfaccia Operation e sue implementazioni: somma, moltiplicazione, assegnamento e negazione.

```
public abstract class Operation implements Expression {

     protected Expression leftExpression;
     protected Expression rightExpression;

     public abstract Value operation();

     public abstract Value evalutate();

}
```

La classe Operation rappresenta sia le operazioni binarie che unarie, dunque potremmo definire nelle sotto-classi costruttori differenti. Nel caso di una sotto-classe unaria useremo uno solo dei due attributi.
Implementazione: Sum

```
public class Sum extends Operation {

     public Sum(Expression r, Expression l) {
          this.leftExpression = l;
          this.rightExpression = r;
     }

     public Value operation() {
          Value al = leftExpression.evalutate();
          Value ar = rightExpression.evalutate();

          int il = Integer.parseInt(al.toString());
          int ir = Integer.parseInt(ar.toString());

          return new Number(il+ir);
      }

     public Value evalutate() {
          return operation();
     }
}
```

Similmente sarà la classe Multiplication che vedremo tra poco. Ma prima vediamo in dettaglio questa. Il metodo ereditato evalutate fa da wrapper a operation che a sua volta per valutare l'operazione da fare non fa altro che valutare le due espressioni, sommarle tra loro e restituire una classe Number (che è sottoclasse di Value) come Value dell'espressione.

Implementazione: Multiplication

```
public class Multiplication extends Operation {

     public Sum(Expression r, Expression l) {
          this.leftExpression = l;
          this.rightExpression = r;
     }

     public Value operation() {
          Value al = leftExpression.evalutate();
          Value ar = rightExpression.evalutate();

          int il = Integer.parseInt(al.toString());
          int ir = Integer.parseInt(ar.toString());

          return new Number(il*ir);
      }

     public Value evalutate() {
          return operation();
     }
}
```

Come visto prima, nulla cambia. Vediamo ora una classe leggermenete diversa, l'operatore Negazione ovvero il segno -.
Implementazione: Negative

```
public class Negative extends Operation {

     public class Negative(Expression e) {
          this.rightExpression = e;
     }

     public Value operation() {
          Value a = rightExpression.evalutate();

          int i = Integer.parseInt(a.toString());

          return new Number(i*(-1));
      }

     public Value evalutate() {
          return operation();
     }
}
```

Essendo l'operazione unaria userà una sola espressione invece che due.
Finiamo la nostra carrelata di operatori con l'operatore Assign che vedremo essere un attimo differente dalle 3 implementazioni qui viste.

Implementazione: Assign

```
public class Assign extends Operation {

     public Sum(Expression r, Expression l) {
          this.leftExpression = l;
          this.rightExpression = r;
     }

     public Value operation() {
          Variable var = (Variable)leftExpresson;
          Value expr = rightExpression.evalutate();

          int i = expr.toString();
          Value val = new Number(i);
          var.setValue(val);

          return val;
      }

     public Value evalutate() {
          return operation();
     }
}
```

Come possiamo vedere Assign non fa proprio nella stessa maniera di Sum e Product. Innanzitutto Assign sa che alla sua sinistra c'è sicuramente una variabile (vedi il cast) che riceverà il valore dell'espressione di destra (vedi metodo setValue()). Alla fine dell'assegnazione verrà tornato il valore assegnato alla variabile.. questo perché potremmo avere delle assegnazioni in catena, per esempio var1 = var2 = 0
Visti gli operatori passiamo alle costanti e ai tipi delle variabili (in questo caso uno solo, Number).

```
public interface Value {

     String toString();
}

public class Number implements Value {

     private String value;

     public Number(int val) { value = val+”“; }

     public String toString() { return value; }
}
```

Prima di vedere le costanti, vediamo un po' come sono composte l'interfaccia Value ed il tipo Number. L'interfaccia ha solo un metodo comune per ritornare un tipo di riconoscimento del valore, mentre il tipo Number viene costruito a

partire da un valore.
Vediamo ora la Costante che è una delle tante classi che sono sotto-classi di Espressione.

```
public abstract class Constant implements Expression {

     protected Value value;

     public Constant(Value val) {  value = val; }

     public Value getValue() { return value; }

     public Value evalutate() { return getValue(); }

}
```

La costante si comporta, anche se non lo abbiamo ancora visto, in maniera molto simile sia alla Variabile che alla Funzione. È giunto dunque tempo di vedere anche queste due strutture.

```
public abstract class Variable implements Expresson {

     protected Value value;

     public Variable(Value val) {
          value = val;
     }

     public Value getValue() {
          return value;
     }

     public void setValue(Value v) {
          value = v;
     }

     public Value evalutate() {
          return getValue();
     }
}
```

La variabile definisce molte poche cose, e come si era detto è uguale alla Constant, se non per il fatto che la Variable, può variare.

La classe Function, che vedremmo fra poco, invece sarà strutturata in una maniera differente, dove ovviamente le operazioni su di essa saranno definite dall'utilizzatore. Da notare che il corpo della funzione non è mai stato definito nella nostra implementazione, questo non significa che il programmatore non possa farlo.

```
public abstract class Function implements Expression {

     protected Set<Expression> expressions;

     public Function(Set<Expression> expr) {
          this.expressions = expr;
     }

     public abstract Value getResult(Set<Variable> params);

     public abstract Value evalutate();

}
```

La classe Function ovviamente dipende dal tipo di funzione, e soprattutto cosa deve fare. Per questo i suoi metodi getResult ed evalutate restano abstract.
Finiamo il nostro interprete vedendo l'interfaccia Expression:

```
public interface Expression {
     Value evalutate();
}
```