

POLITECNICO DI MILANO

V Facoltà di Ingegneria

Corso di Laurea Triennale in Ingegneria Informatica

Dipartimento di Elettronica e Informazione



**DEFINIZIONE ED IMPLEMENTAZIONE DELLA
RAPPRESENTAZIONE INTERNA DELLE
ESPRESSIONI DEL LINGUAGGIO PERLA**

Relatore: Chiar.mo prof. Fabio SCHREIBER

Tesi di Laurea Triennale di:
Stefano VETTOR Matr. N. 668587

Anno Accademico 2007/2008

Ringrazio il Prof. Schreiber per avermi dato l'opportunità di lavorare a questo interessantissimo progetto. L'Ing. Camplani, l'Ing. Marelli e l'Ing. Fortunato per la pazienza ed il supporto datomi durante lo sviluppo del progetto.

Ringrazio inoltre il mio amico Marco per la sua presenza ed il suo supporto morale che mi ha spronato ad impegnarmi e a dare il massimo.

E per ultima, ma non per questo meno importante, ringrazio l'amore della mia vita Giulia che mi è sempre stata accanto sostenendomi.

Stefano

Indice:

Introduzione	8
Nascita del progetto PERLA	8
Linguaggio PERLA	9
<i>Introduzione a PERLA</i>	9
<i>Motivazioni del progetto PERLA</i>	10
Il sistema di gestione e valutazione delle espressioni	12
Struttura interna del sistema	13
Introduzione	13
Costanti	15
Introduzione	15
Struttura delle costanti	16
<i>Introduzione</i>	16
<i>I Metodi Operazioni</i>	17
<i>I Metodi di Controllo di Tipo</i>	18
Classi delle Costanti	19
<i>Superclasse Constant</i>	19
<i>Classe ConstantBuiltin</i>	20
<i>Classe ConstantInteger</i>	20
<i>Classe ConstantFloat</i>	21
<i>Classe ConstantString</i>	22
<i>Classe ConstantLogic</i>	23
<i>Classe ConstantID</i>	24
<i>Classe ConstantTimestamp</i>	24
<i>Classe ConstantNull</i>	25
<i>Classe ConstantUserDefined</i>	26
<i>Classe ConstantVectorInteger</i>	26
Costanti UserDefined	27
<i>Struttura</i>	27
<i>Interazione tra costanti user-defined e built-in</i>	27
<i>Controllo di tipo tra costanti user-defined e built-in</i>	29
Constant Factory	30
Nodi	31
Introduzione	31
<i>Nodi Operazione (NodeOperation)</i>	32
<i>Nodi Costante (NodeConstant)</i>	33

<i>Nodi Aggregazione (Node Aggregation)</i>	33
<i>Nodi Funzione (NodeFunction)</i>	34
Classi dei Nodi	35
<i>Superclasse Node</i>	35
<i>Classe NodeOperation</i>	35
<i>Classe NodeAlgebricalOperation</i>	36
<i>Classe NodeAddition</i>	36
<i>Classe NodeSubtraction</i>	36
<i>Classe NodeMultiplication</i>	37
<i>Classe NodeDivision</i>	37
<i>Classe NodeSign</i>	38
<i>Classe NodeLogicalOperation</i>	38
<i>Classe NodeAND</i>	38
<i>Classe NodeOR</i>	39
<i>Classe NodeXOR</i>	39
<i>Classe NodeLogicTest</i>	39
<i>Classe NodeNOT</i>	40
<i>Classe NodeisNull</i>	40
<i>Classe NodeBetween</i>	40
<i>Classe NodeBitwiseOperation</i>	41
<i>Classe NodeBitwiseAND</i>	41
<i>Classe NodeBitwiseOR</i>	42
<i>Classe NodeBitwiseXOR</i>	42
<i>Classe NodeBitwiseNOT</i>	43
<i>Classe NodeBitwiseRightShift</i>	43
<i>Classe NodeBitwiseLeftShift</i>	44
<i>Classe NodeComparisonOperation</i>	44
<i>Classe NodeLike</i>	44
<i>Classe NodeEqual</i>	45
<i>Classe NodeUnequal</i>	45
<i>Classe NodeGreater</i>	46
<i>Classe NodeStrictlyGreater</i>	46
<i>Classe NodeMinor</i>	47
<i>Classe NodeStrictlyMinor</i>	47
<i>Classe NodeConstant</i>	48
<i>Classe NodeAggregation</i>	48
<i>Classe NodeAggregationStandard</i>	48
<i>Classe NodeAggregationTimeBased</i>	49
<i>Classe NodeAggregationSampleBased</i>	49

<i>Classe NodeExists</i>	49
<i>Classe NodeExistsAttribute</i>	50
<i>Classe NodeExistsAll</i>	50
<i>Classe NodeLogicalObjectAttribute</i>	50
Eccezioni	51
Introduzione	51
Le eccezioni	51
<i>DivisionByZeroException</i>	51
<i>ConstantRuntimeException</i>	51
<i>ConstantCastException</i>	51
<i>ConstantCreationException</i>	51
<i>OperationNotSupportedException</i>	51
Conclusioni	52
Stato dell'arte	53
Bibliografia	54

Introduzione

Nascita del progetto PERLA

Nel luglio 2005 da un gruppo di università italiane, coordinate dal Politecnico di Milano, è nato il progetto **ART DECO** (**Adaptive InfRasTructures for DECentralized Organizations**). Il progetto, finanziato dal Ministero dell'Università e della Ricerca, mira allo sviluppo di tecniche e strumenti per favorire la diffusione delle “networked enterprise” tra le piccole e medie imprese italiane. Uno dei casi di studio di **ART DECO** concerne il processo produttivo e distributivo di vini pregiati da parte di un'azienda vinicola leader del settore. Al fine di garantire e certificare la qualità dei propri prodotti l'azienda può essere interessata a monitorare i principali parametri influenzanti il ciclo produttivo come ad esempio l'acidità ed umidità del terreno o le temperature e le vibrazioni a cui sono state esposte le bottiglie durante il trasporto verso i rivenditori. Si mirerebbe inoltre a garantire la completa tracciabilità degli interventi effettuati su ciascuna partita di vino prodotta, dalla raccolta dell'uva fino alla fase di etichettatura.

Per poter soddisfare le richieste dell'azienda è necessario realizzare una vasta rete eterogenea di sensori (da reti wireless di sensori per monitorare la temperatura a cui si trovano i prodotti in ogni attimo del ciclo produttivo, alle etichette RFID da applicare sulle bottiglie per identificarle durante il trasporto, fino ai dispositivi GPS per avere l'esatta posizione dei lotti durante la fase di trasferimento ai rivenditori). Con i dati così ottenuti sarà possibile, durante la fase produttiva, evitare che sbalzi improvvisi dei parametri possano andare ad influenzare negativamente la qualità dei prodotti, ed in fase di vendita fornirne ai clienti l'intera storia a partire dall'uva nella vigna fino ad arrivare alla bottiglia nel negozio (rafforzando così la certificazione di qualità)!

A causa del grande numero di tecnologie coinvolte, ognuna con la propria interfaccia di controllo ed interrogazione, il quantitativo di lavoro e di conoscenze specifiche da parte dell'utente del sistema per poter correttamente recuperare e manipolare i dati ottenuti sarebbe tale da rendere il sistema difficilmente utilizzabile. Si è quindi posto il problema di rendere omogenea la visione del sistema all'utente finale mascherandone la complessità.

Una soluzione a questo problema è la definizione di un linguaggio di alto livello completamente dichiarativo che permetta all'utente di interrogare un sistema pervasivo in un modo estremamente simile a come si interrogherebbe una base di dati. Un progetto del Politecnico di Milano (nato come tesi di Laurea specialistica di Marco Marelli e Marco Fortunato) ha definito un linguaggio, di nome **PERLA** (**PER**vasive **L**anguage), che consente l'interrogazione di un sistema pervasivo utilizzando una sintassi simile a quella dell'SQL standard. L'ovvio vantaggio

dell'utilizzo di un tale linguaggio sta nel fatto che la complessità dell'intero sistema viene nascosta all'utente finale sicché questi non debba preoccuparsi delle caratteristiche tecniche dei singoli sensori ma possa concentrarsi unicamente sulla manipolazione dei dati da essi ricevuti. Il punto di partenza dello sviluppo di questo progetto è stato TinyDB, un sistema per l'interazione con reti omogenee di sensori, che però ha notevoli limitazioni, fatto che ha quindi spinto a realizzare un nuovo linguaggio.

Linguaggio PERLA

Introduzione a PERLA

PERLA (PERvasive LAnguage) è un linguaggio completamente dichiarativo che permette all'utente di interrogare un sistema pervasivo in modo simile a come interrogherebbe una base di dati utilizzando SQL. Un sistema pervasivo è una grande rete eterogenea composta da diversi dispositivi, ognuno dei quali può utilizzare diverse tecnologie, come ad esempio reti di sensori wireless (WSN), sistemi RFID, GPS e molti altri tipi di sensori.

Per linguaggio dichiarativo si intende un linguaggio con cui si stabilisce che cosa debba fare il programma, cioè che relazione intercorre fra input ed output, a differenza dei linguaggi imperativi, con i quali si stabilisce come un problema deve essere risolto, cioè come derivare un output da uno o più input.

Lo scopo principale del linguaggio PERLA è quindi nascondere all'utente l'elevata complessità del sistema (dovuta all'eterogeneità ed al numero dei sensori con cui si andrà ad operare) e della programmazione di basso livello, fornendogli una interfaccia simile a quella di una base di dati, sicché possa recuperare i dati dal sistema in modo semplice e veloce.

Il linguaggio è stato pensato per essere eseguito su un'architettura costituita da tre livelli:

1. livello di accesso fisico ai dispositivi
2. livello degli oggetti logici
3. livello applicativo

All'interno dell'ultimo livello è presente il parser: esso è il componente che riceve le query sottoposte dall'utente in formato di stringhe, e ne verifica la correttezza sintattica. Obiettivo del parser è quello di tradurre ciascuna query in una struttura ad oggetti semanticamente equivalente, ma che risulti facilmente utilizzabile dal motore di esecuzione della query.

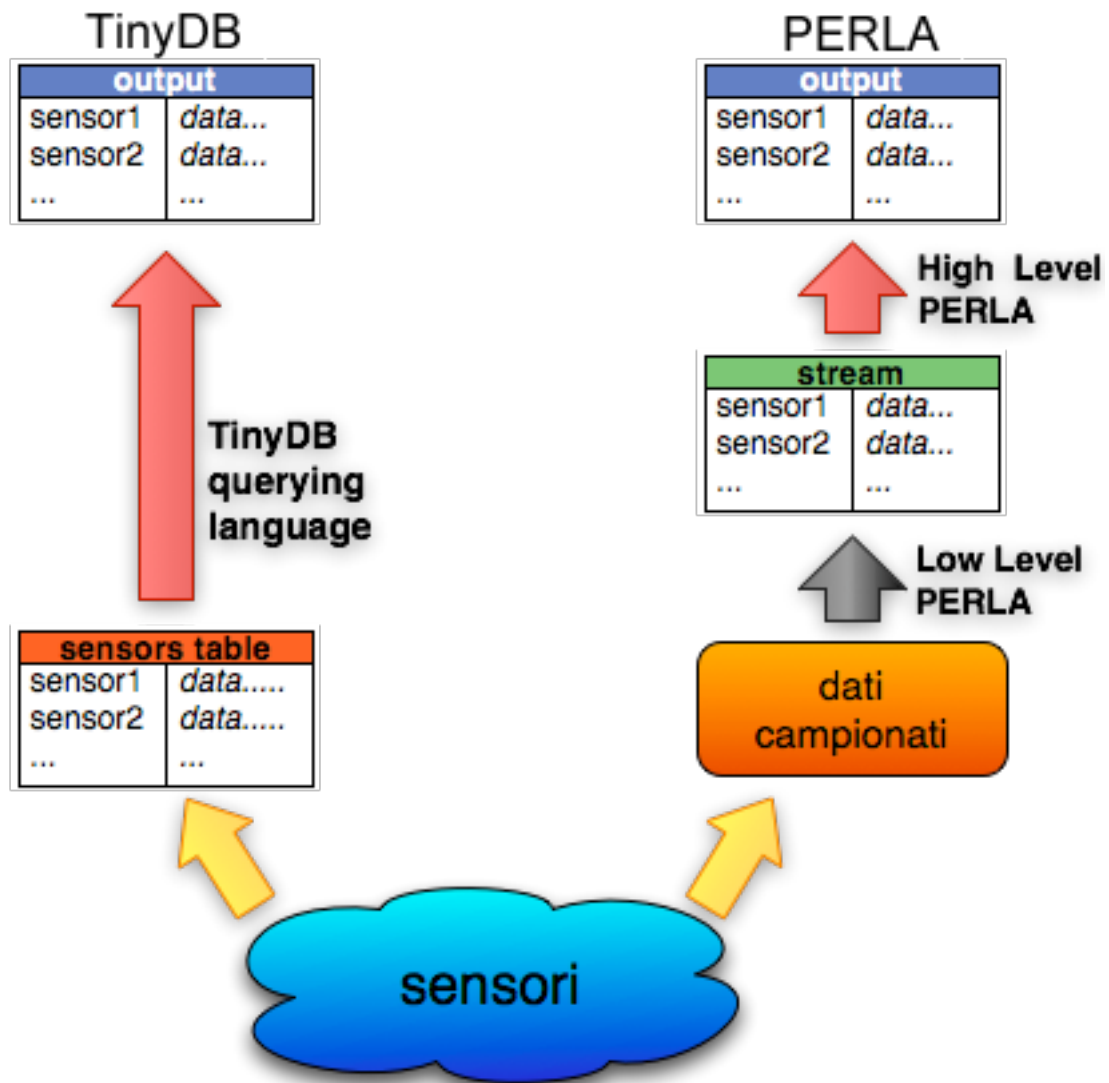
Motivazioni del progetto PERLA

PERLA non è stato il primo tentativo di astrarre una rete di sensori wireless cercando di mascherarla come se fosse una base di dati, ne esistono altri tra cui ci sono anche GSN (Global Sensor Network) e TinyDB che è, tra l'altro, il progetto più famoso.

La sua fama gli deriva dall'esser stato il primo tentativo di estrarre informazioni da una rete di sensori wireless, utilizzando questo tipo di approccio: esso permette infatti all'utente di interrogare la rete tramite un linguaggio SQL-like semplificando notevolmente l'operazione fruizione dei dati.

TinyDB è un sistema di elaborazione di query, progettato dall'università di Berkeley, per l'estrazione di informazioni da reti di sensori basate su TinyOS. In ogni sua implementazione è presente una tabella dei sensori chiamata *sensors*, di dimensione variabile, che contiene i campionamenti dei sensori provenienti da tutti i nodi. Ogni colonna (rappresentante un attributo) della tabella si riferisce ad un tipo di sensore (per esempio temperatura, pressione, ecc.) e sono inoltre presenti un attributo per identificare univocamente un nodo della rete, e un attributo per il *TIMESTAMP*. Ogni record della tabella è il risultato di un campionamento eseguito da un nodo della rete. Se un sensore non è presente in un nodo, tutti i record che si riferiscono a quel nodo conterranno un valore nullo nel campo corrispondente all'attributo. Il risultato di una query sulla tabella dei sensori può essere memorizzato in una tabella di materializzazione, di dimensione fissa.

Nonostante l'indubbia innovatività del progetto TinyDB è comunque afflitto da alcuni limiti nell'implementazione che lo rendono nella pratica un prodotto di difficile utilizzazione. PERLA invece, benché utilizzi anch'esso un'astrazione che rende la rete di sensori simile ad una base di dati, ha alcune differenze con TinyDB che lo rendono più adatto per l'utilizzo in situazioni reali. La prima importante differenza riguarda le classi di dispositivi supportate dai due linguaggi: TinyDB infatti può essere utilizzato solo su reti wireless di sensori omogenee utilizzando TinyOS, al contrario PERLA è stato progettato per poter lavorare contemporaneamente con diverse classi di dispositivi che possono andare dai sensori di temperatura, ai lettori RFID o a dei PDA. La seconda grande differenza concerne invece il processo del campionamento; secondo l'approccio adottato da PERLA c'è l'idea che il campionamento non possa essere del tutto nascosto all'utente in quanto poter variarne i parametri a runtime può rivelarsi estremamente utile. TinyDB invece maschera totalmente il processo di campionamento tramite la tabella *sensors*, e quindi impedisce all'utente qualsiasi tipo di controllo sul procedimento eccezion fatta per il periodo. Questo rende sì il sistema di facile utilizzazione e comprensione ma anche adatto solamente a situazioni in cui i dispositivi della rete sono tutti dello stesso tipo e le caratteristiche note a priori (in fase di scrittura del software).



In un sistema eterogeneo l'operazione di campionamento deve invece essere gestita in modo più complesso, dando all'utente la possibilità di definire criteri sofisticati per decidere quali nodi dovranno eseguire il campionamento e come. Proprio grazie all'importanza che PERLA riserva alla fase di campionamento, il linguaggio è suddiviso in due parti:

1. un livello basso che gestisce la procedura di campionamento
2. un livello alto che manipola i dati campionati producendo il risultato dell'interrogazione.

Ciò consente una maggiore flessibilità e potenza del linguaggio, a scapito della semplicità d'uso.

Il sistema di gestione e valutazione delle espressioni

In questo documento si parlerà in particolare del sistema di gestione, rappresentazione e valutazione delle espressioni del Query Analyzer di PERLA.

Il Query Analyzer è stato scritto in Java™ e così anche il sistema di valutazione delle espressioni è stato scritto utilizzando lo stesso linguaggio.

Il linguaggio PERLA offre all'utente la possibilità di creare vari tipi di espressioni complesse sfruttando un'ampia gamma di tipi di costanti predefinite oppure create dall'utente stesso. Gli elementi chiave di questo sistema sono le Costanti (che rappresentano gli elementi su cui poi verrà valutata l'espressione) ed i Nodi (che aggregano le costanti secondo una specifica legge di composizione binaria - o ternaria in alcuni casi).

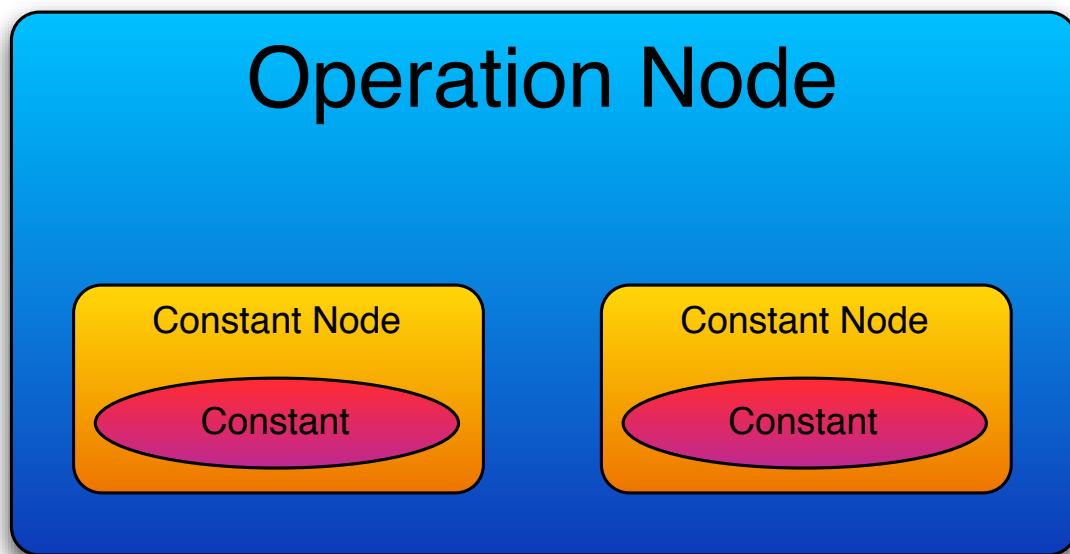
Struttura interna del sistema

Introduzione

Come già precedentemente detto i due elementi principali del sistema di rappresentazione e valutazione delle espressioni sono le costanti ed i nodi.

Le costanti, la cui superclasse è *Constant*, sono le classi che rappresentano gli elementi su cui si effettuano tutte le operazioni (includono il tipo di costante ed il suo valore. I nodi, la cui superclasse è *Node*, si occupano dell'aggregazione delle costanti secondo una particolare operazione (unaria, binaria o ternaria... a seconda dell'operazione). Sebbene i nodi associno le costanti mediante un'operazione è la costante stessa ad occuparsi dell'effettiva implementazione dell'operazione.

I nodi sono essenzialmente di due tipi: nodi costanti e nodi operazione, i nodi operazione, come detto prima, si occupano di aggregare più nodi costanti che contengono gli oggetti costanti e vi accedono.



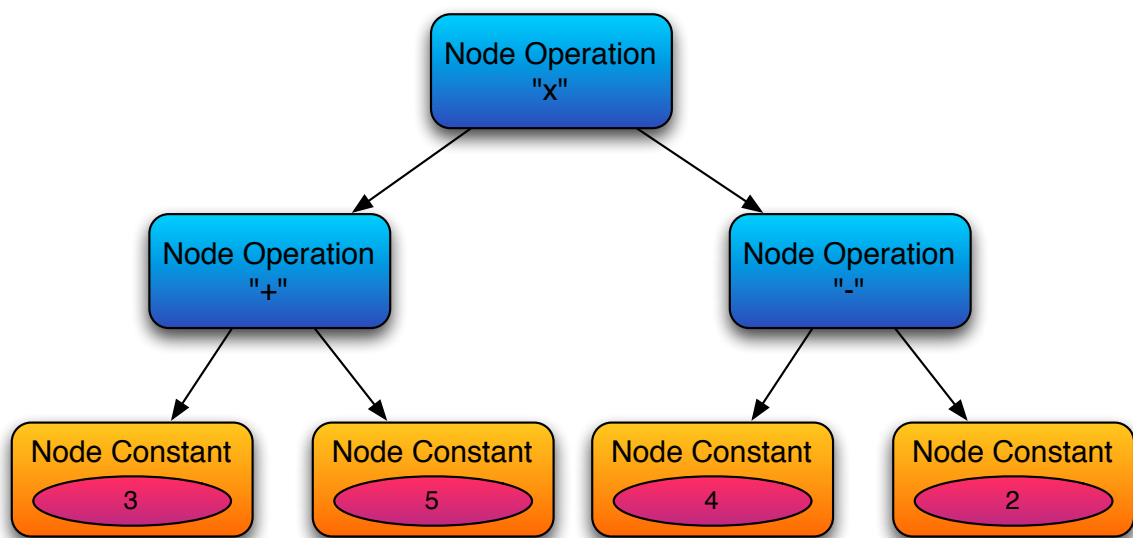
Ogni nodo operazione come propri elementi può contenere un qualsiasi tipo di nodo, quindi indifferentemente nodi costanti o nodi operazioni. In questo modo si può creare una struttura ad albero per la rappresentazione delle espressioni che si risolve in cascata partendo dalle foglie dell'albero, risalendo fino alla radice.

Infatti ogni nodo ritorna al proprio superiore il risultato della propria operazione.

Un esempio può essere osservato nella struttura qua sotto:

come è possibile vedere l'albero ha 4 foglie (che sono sempre nodi costanti) ognuna contenente una costante, a loro volta sono poi contenute in un nodo operazione che è ancora a sua volta contenuto in un altro nodo operazione che è la radice dell'albero. L'esempio qua sotto quindi rappresenta l'espressione $(3+5)*(4-2)$.

In fase di valutazione il nodo "+" ed il nodo "-" restituiranno al nodo "x" una costante rappresentante i loro rispettivi risultati sicché possa essere effettuato il calcolo finale.



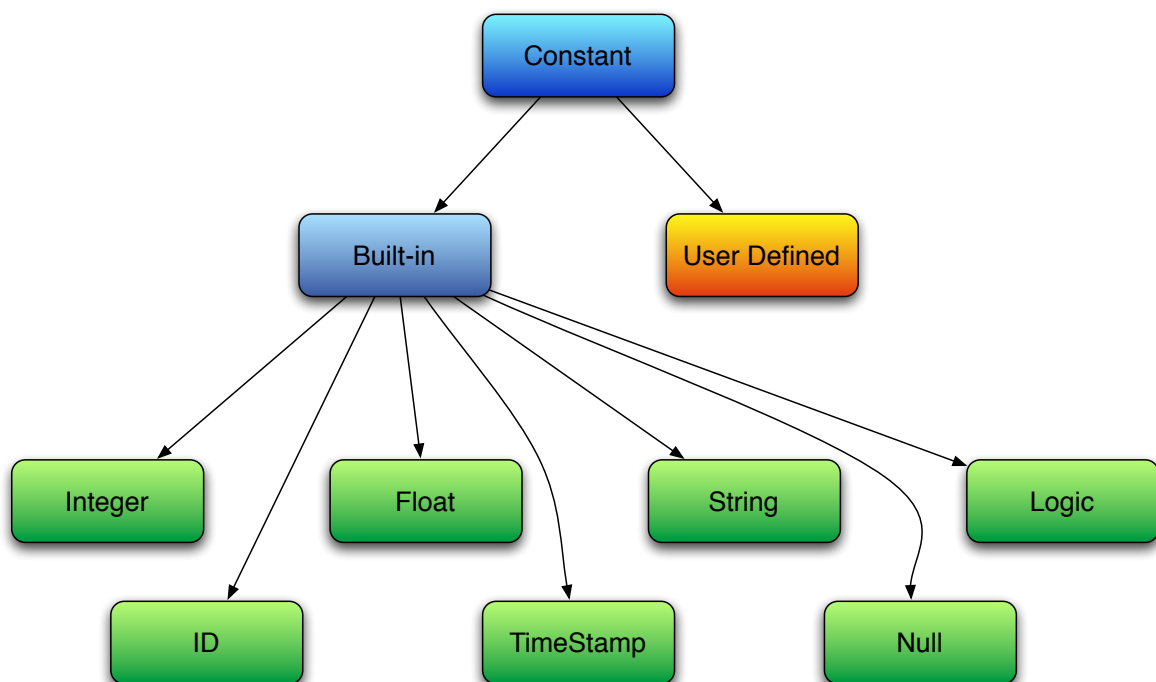
Costanti

Introduzione

Come già detto le costanti sono gli elementi base della rappresentazione e della valutazione delle espressioni, ogni foglia contiene una costante ed ogni nodo operazione ritorna una costante contenente il risultato dell'operazione. La superclasse delle costanti è la classe astratta *Constant* che viene estesa da tutte le altre classi costante.

Le costanti si suddividono in due tipologie: quelle Built-in, che sono quelle disponibili di default, e quelle User-defined, che sono definite dall'utente e caricate a runtime.

Il diagramma qua sotto dà una rappresentazione visiva della suddivisione delle costanti.



Struttura delle costanti

Introduzione

Ogni classe che estende `Constan` è strutturata essenzialmente allo stesso modo:

- ha una variabile che contiene il dato memorizzato (ad esempio nel caso di una costante intera una variabile Java™ `int`)
- i costruttori per istanziare la classe, di cui uno deve essere obbligatoriamente mediante il passaggio di una stringa di caratteri
- i metodi per il casting della costante in altri tipi di costante (ove questo sia possibile)
- i metodi per implementare le varie operazioni sulla costante, difatti sono le costanti stesse ad implementare le varie operazioni



Variabile

Come sopra detto, ogni costante contiene una variabile adibita alla memorizzazione del dato della costante; nel caso esista un equivalente Java™ del tipo sarà una variabile di quello (una *ConstantInteger* [costante intera] avrà una variabile *int*, una *ConstantString* [costante stringa] avrà una variabile *String*). Nel caso non esista si provvede a crearne una apposita. Una volta inizializzata, la variabile non può essere alterata.

Costruttori

I costruttori ricevono un particolare tipo di variabile ed inizializzano la variabile contenente il dato, i tipi accettati variano da costante a costante tranne per uno... difatti il costruttore che riceve come parametro una stringa rappresentante il dato è obbligatorio in ogni costante (in maniera tale che possa essere passata direttamente dalla query PERLA).

Metodi Casting

I metodi di casting permettono per l'appunto di trasformare la classe in altri tipi di costante, ove questo sia possibile, per permetterle l'interazione con altri oggetti costanti di tipo diverso (ad esempio il casting di una *ConstantInteger* in *ConstantFloat* per l'interazione con un altro *ConstantFloat*).

Metodi operazioni

I metodi delle operazioni implementano vari tipi di operazioni (che possono andare dalla somma e moltiplicazione al confronto di uguaglianza) tra la *Constant* ed un'altra che le viene passata come argomento del metodo restituendo un nuovo oggetto *Constant* contenente il risultato dell'operazione.

Metodi di controllo di tipo

I metodi che consentono il controllo a runtime dei tipi ritornati da determinate operazioni tra determinati tipi di costanti. Sono metodi statici e quindi non richiedono l'istanziamento di un oggetto per essere invocati.

I Metodi Operazioni

I metodi operazioni delle classi che estendono *Constant* possono essere facilmente suddivisi in gruppi a seconda della loro funzione:

- Legge di composizione unaria, binaria o ternaria sulle costanti (ad esempio la moltiplicazione, la somma, ecc...). Questi metodi hanno esattamente il nome dell'operazione che eseguono e ricevono come argomento del metodo il secondo parametro dell'operazione (il primo è la costante che esegue il metodo). Questi metodi ritornano un oggetto *Constant* contenente il risultato dell'operazione.
- Operazioni di confronto operano esclusivamente confronti tra costanti (ad esempio di uguaglianza, disuguaglianza, maggioranza, ecc...) e ritornano una *ConstantLogic* come risultato dell'operazione di confronto. Anche in questo caso i metodi hanno il nome dell'operazione di confronto preceduto da *is* (uguaglianza = *isEqual*, maggioranza = *isGreater*).

Per fare un esempio per sommare l'intero 4 contenuto nella costante *constantFour* e l'intero 5 contenuto nella costante *constantFive* procederemo con:

```
constantFour.addition(constantFive);
```

l'istruzione scritta sopra ritornerà quindi un nuovo oggetto costante contenente l'intero 9.

I Metodi di Controllo di Tipo

i metodi di controllo di tipo possono essere facilmente suddivisi a seconda dei loro compiti:

- Tipo di risultato ritornato da un'operazione, hanno nome del tipo *nomeoperazioneResutType()*, sono metodi statici che ricevono come parametro un oggetto di tipo *class* e ritornano un altro oggetto di tipo *class* che è il tipo di costante che ritorna una determinata operazione (ad esempio `ConstantInteger.additionResultType(ConstantInteger.class)` ritorna *ConstantInteger.class* in quanto il risultato di una somma tra interi è ancora un intero).
- il metodo `isEquallyComparable(class object)` è un metodo statico che ritorna `TRUE` se la classe passata come argomento del metodo supporta le operazioni di uguaglianza e disuguaglianza con la classe che esegue il metodo.
- il metodo `isFullyComparable(class object)` è un metodo statico che ritorna `TRUE` se la classe passata come argomento del metodo supporta tutte le operazioni di confronto con la classe che esegue il metodo.

Classi delle Costanti

Superclasse Constant

La superclasse Constant è una superclasse astratta che viene estesa da tutte le altre classi costanti e possiede le definizioni di tutti i metodi.

org.dei.perla.parser.expressions.Constant
<ul style="list-style-type: none">● isNull(): boolean● isUserDefined(): boolean● getValueInt(): int● getValueFloat(): double● getValueLogic(): LogicValue● getValueString(): String● getValueID(): String● getValueTimestamp(): long● addition(parConst: Constant): Constant● <u>additionResultType(parConst: Class<? extends Constant>): Class<? extends Object></u>● subtraction(parConst: Constant): Constant● <u>subtractionResultType(parConst: Class<? extends Constant>): Class<? extends Object></u>● multiplication(parConst: Constant): Constant● <u>multiplicationResultType(parConst: Class<? extends Constant>): Class<? extends Object></u>● division(parConst: Constant): Constant● <u>divisionResultType(parConst: Class<? extends Constant>): Class<? extends Object></u>● isEqual(parConst: Constant): ConstantLogic● isUnequal(parConst: Constant): ConstantLogic● isGreater(parConst: Constant): ConstantLogic● isStrictlyGreater(parConst: Constant): ConstantLogic● isMinor(parConst: Constant): ConstantLogic● isStrictlyMinor(parConst: Constant): ConstantLogic● and(parConst: Constant): ConstantLogic● or(parConst: Constant): ConstantLogic● xor(parConst: Constant): ConstantLogic● not(): ConstantLogic● andInverse(parConst: Constant): ConstantLogic● orInverse(parConst: Constant): ConstantLogic● xorInverse(parConst: Constant): ConstantLogic● additionInverse(parConst: Constant): Constant● subtractionInverse(parConst: Constant): Constant● multiplicationInverse(parConst: Constant): Constant● divisionInverse(parConst: Constant): Constant● isGreaterInverse(parConst: Constant): ConstantLogic● isStrictlyGreaterInverse(parConst: Constant): ConstantLogic● isMinorInverse(parConst: Constant): ConstantLogic● isStrictlyMinorInverse(parConst: Constant): ConstantLogic● isBetween(parConstBegin: Constant, parConstEnd: Constant): ConstantLogic● bitwiseAND(parConst: Constant): Constant● <u>bitwiseANDResultType(parConst: Class<? extends Constant>): Class<? extends Object></u>● bitwiseOR(parConst: Constant): Constant● <u>bitwiseORResultType(parConst: Class<? extends Constant>): Class<? extends Object></u>● bitwiseXOR(parConst: Constant): Constant● <u>bitwiseXORResultType(parConst: Class<? extends Constant>): Class<? extends Object></u>● bitwiseNOT(): Constant● <u>bitwiseNOTResultType(): Class<? extends Object></u>● bitwiseRightShift(parConst: Constant): Constant● <u>bitwiseRightShiftResultType(parConst: Class<? extends Constant>): Class<? extends Object></u>● bitwiseLeftShift(parConst: Constant): Constant● <u>bitwiseLeftShiftResultType(parConst: Class<? extends Constant>): Class<? extends Object></u>● bitwiseANDInverse(parConst: Constant): Constant● bitwiseORInverse(parConst: Constant): Constant● bitwiseXORInverse(parConst: Constant): Constant● bitwiseRightShiftInverse(parConst: Constant): Constant● bitwiseLeftShiftInverse(parConst: Constant): Constant● isLike(parConst: Constant): ConstantLogic● isLikeInverse(parConst: Constant): ConstantLogic● is(parConst: Constant): ConstantLogic● sign(): Constant● <u>additionInverseResultType(parConst: Class<? extends Constant>): Class<? extends Object></u>● <u>subtractionInverseResultType(parConst: Class<? extends Constant>): Class<? extends Object></u>● <u>multiplicationInverseResultType(parConst: Class<? extends Constant>): Class<? extends Object></u>● <u>divisionInverseResultType(parConst: Class<? extends Constant>): Class<? extends Object></u>● <u>bitwiseANDInverseResultType(parConst: Class<? extends Constant>): Class<? extends Object></u>● <u>bitwiseORInverseResultType(parConst: Class<? extends Constant>): Class<? extends Object></u>● <u>bitwiseXORInverseResultType(parConst: Class<? extends Constant>): Class<? extends Object></u>● <u>bitwiseRightShiftInverseResultType(parConst: Class<? extends Constant>): Class<? extends Object></u>● <u>bitwiseLeftShiftInverseResultType(parConst: Class<? extends Constant>): Class<? extends Object></u>● isEquallyComparable(parConst: Class<? extends Constant>): boolean● isFullyComparable(parConst: Class<? extends Constant>): boolean● isLikeComparable(parConst: Class<? extends Constant>): boolean● isLikeInverseComparable(parConst: Class<? extends Constant>): boolean● exist(parConst: Constant): ConstantLogic

Classe ConstantBuiltin

La classe ConstantBuiltin è una classe astratta che non implementa nulla e serve unicamente per differenziare le classi delle costanti definite dal sistema da quelle definite dall'utente. Estende Constant.

Classe ConstantInteger

La classe ConstantInteger implementa le costanti di tipo intero e tutte le operazioni da esso supportate. Estende ConstantBuiltin.

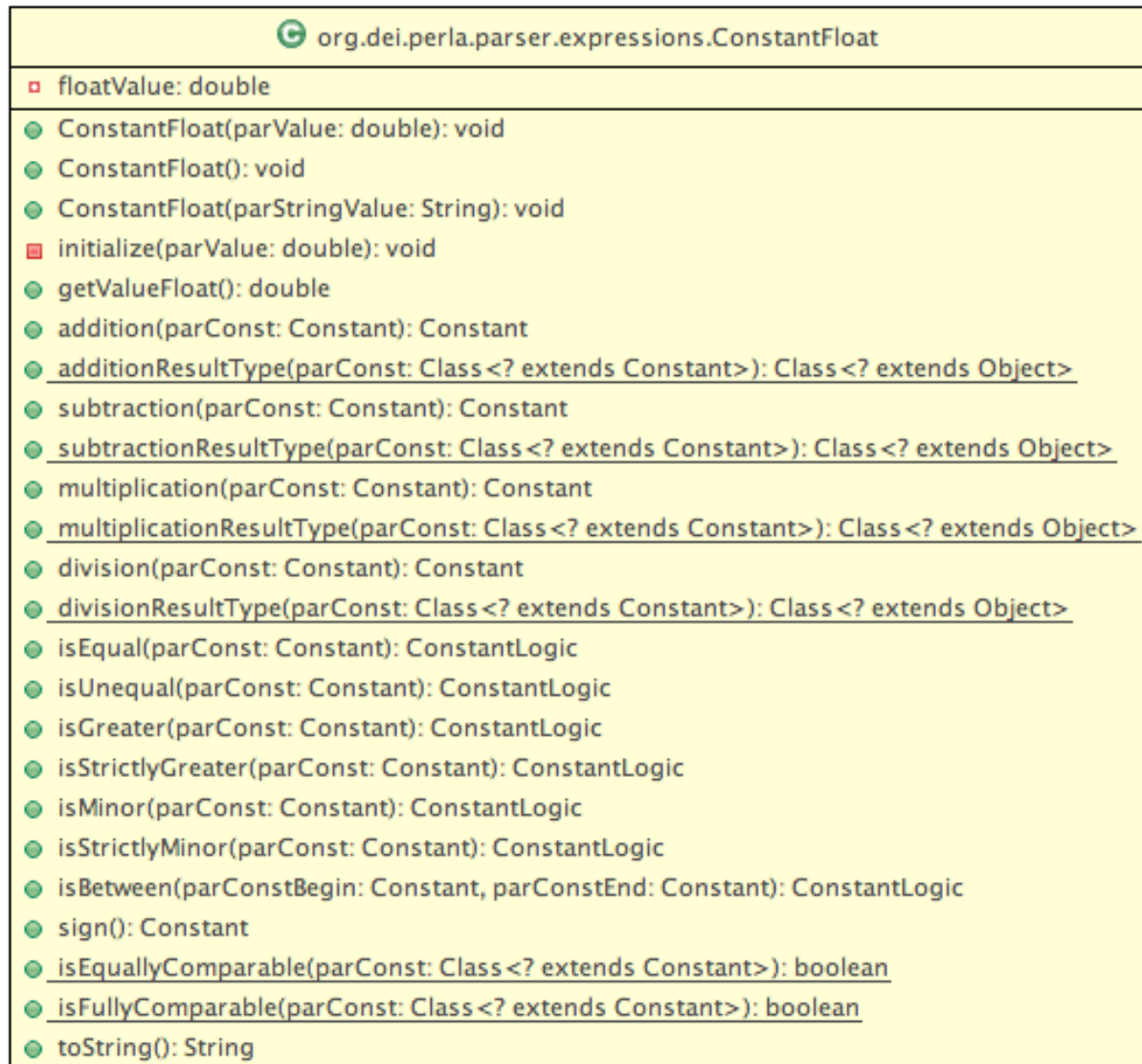
Diagramma UML



Classe ConstantFloat

La classe ConstantFloat implementa le costanti di tipo in virgola mobile e tutte le operazioni da esso supportate. Estende ConstantBuiltin.

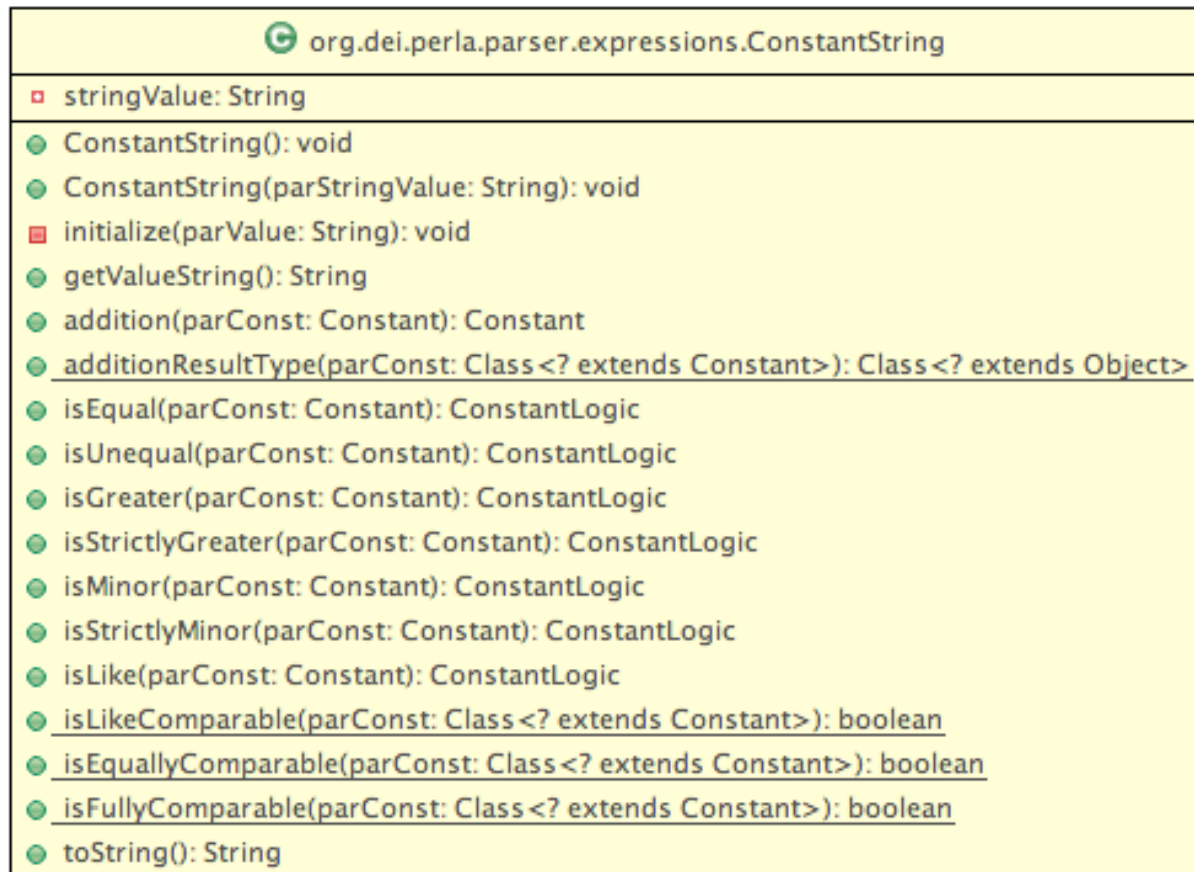
Diagramma UML



Classe ConstantString

La classe ConstantString implementa le costanti di tipo stringa e tutte le operazioni da esso supportate. Estende ConstantBuiltin.

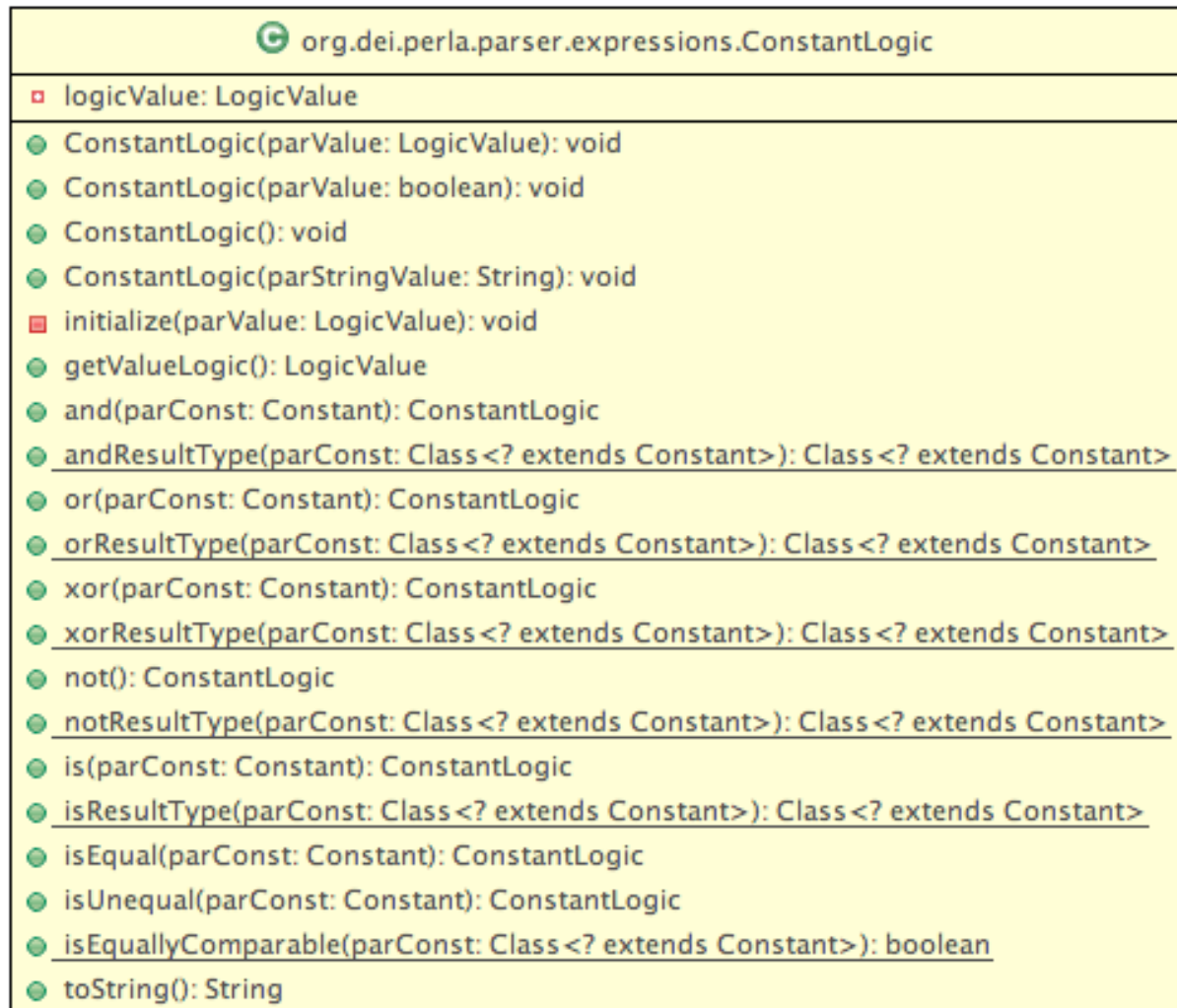
Diagramma UML



Classe ConstantLogic

La classe ConstantLogic implementa le costanti di tipo booleane e tutte le operazioni da esso supportate. Estende ConstantBuiltin.

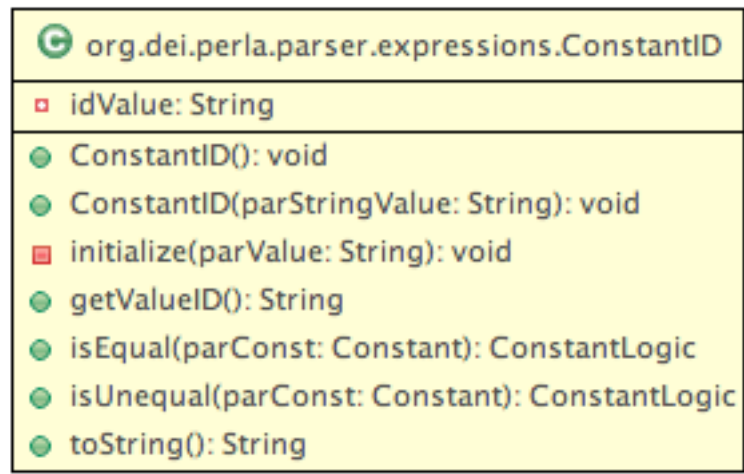
Diagramma UML



Classe ConstantID

La classe ConstantID implementa le costanti di tipo ID e tutte le operazioni da esso supportate. Estende ConstantBuiltin.

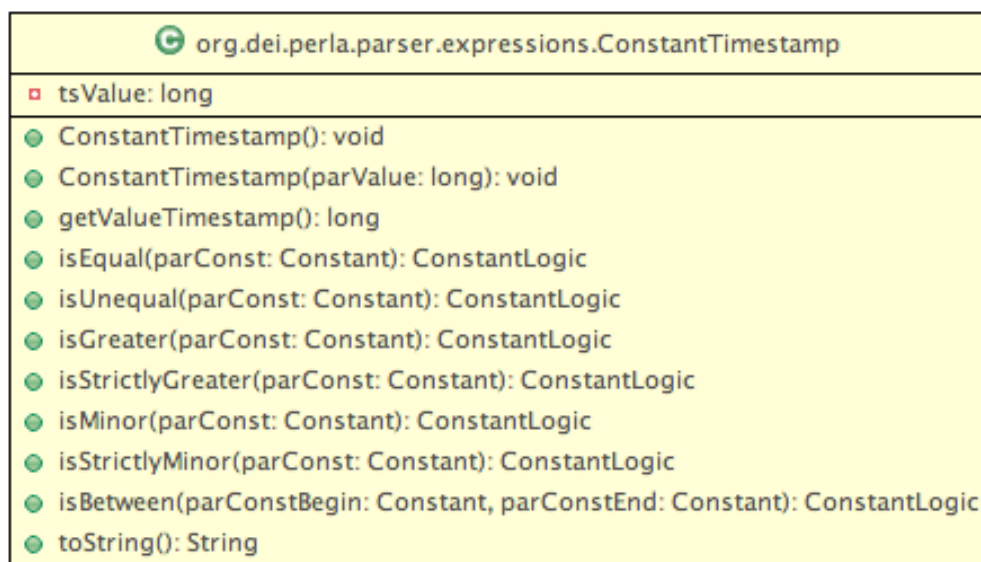
Diagramma UML



Classe ConstantTimestamp

La classe ConstantTimestamp implementa le costanti di tipo TimeStamp e tutte le operazioni da esso supportate. Estende ConstantBuiltin.

Diagramma UML



Classe ConstantNull

La classe ConstantNull implementa le costanti di tipo Null e tutte le operazioni da esso supportate. Estende ConstantBuiltin.

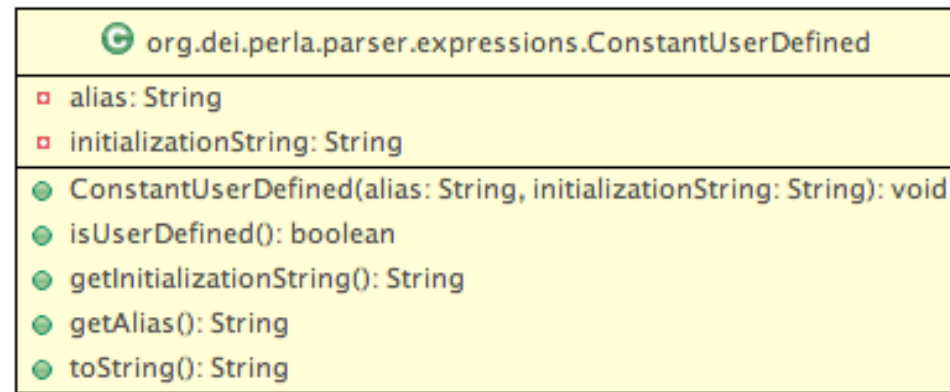
Diagramma UML



Classe ConstantUserDefined

La classe ConstantUserDefined estende Constant e deve essere estesa da tutte le classi constant definite dall'utente.

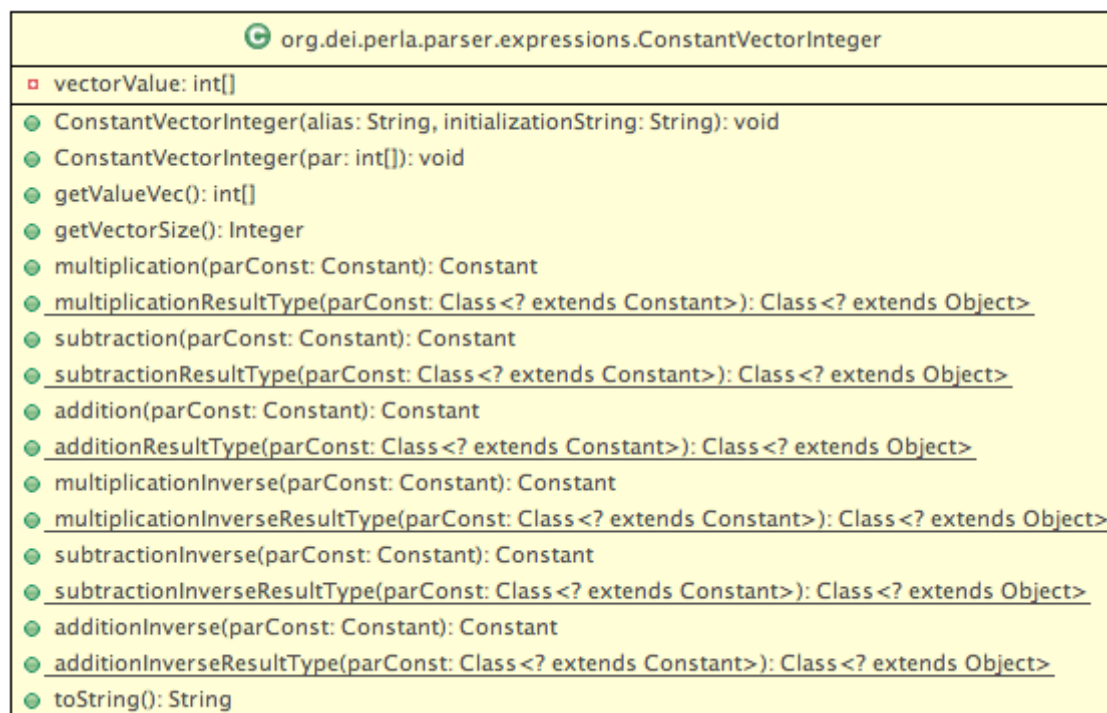
Diagramma UML



Classe ConstantVectorInteger

La classe ConstantVectorInteger implementa le costanti di tipo vettori interi e tutte le operazioni da esso supportate. Estende ConstantUserDefined ed è stata realizzata come classe di test per le classi User Defined.

Diagramma UML



Costanti UserDefined

Una delle peculiarità principali del sistema di rappresentazione e valutazione delle espressioni è la possibilità per l'utente di creare dei propri tipi di costanti e di farli interagire con le costanti BuiltIn senza dover modificare anche solo una riga di codice del progetto originale ma scrivendo unicamente la classe relativa alla costante che desidera realizzare seguendo alcuni parametri di progettazione.

Struttura

La classe `ConstantUserDefined` fornisce due variabili stringa una denominata *alias* che contiene il nome della costante e l'altra chiamata *initializationString* che contiene la stringa di inizializzazione usata per istanziare la costante.

Al contrario di tutte le costanti estendenti *ConstantBuiltin* quelle estendenti *ConstantUserDefined* ritornano **TRUE** all'invocazione del metodo *isUserDefined()*.

Ogni costante User-Defined per ogni metodo relativo ad un'operazione (ad esempio il metodo somma sarebbe `oggettoCostante.addition(altroOggettoCostante)`) deve implementare un metodo di operazione inversa che esegua la stessa operazione, ma invertendo gli operandi per permettere alla classe user-defined di implementare la tecnica, spiegata nel dettaglio in seguito, di interazione con le costanti built-in (ad esempio per l'operazione di addizione il metodo sarebbe `oggettoCostante.additionInverse(altroOggettoCostante)`).

Per i controlli di tipo è obbligatorio che ogni costante user-defined implementi dei metodi *nomeoperazioneResultType()* e *nomeoperazioneInverseResultType()*.

Per permettere l'interazione tra le costanti user-defined e quelle built-in senza dover andare a modificare il codice sorgente del progetto si è fatto anche ricorso alla tecnica reflection offerta da Java™, tecnica di cui parleremo in dettaglio più avanti.

Interazione tra costanti user-defined e built-in

Per permettere l'interazione tra classi built-in e user-defined senza dover andare a modificare le prime ogni volta, si delega tutto il lavoro alle seconde: nel caso ad una costante che estende `ConstantBuiltin` venga passato come parametro una classe estendente `ConstantUserDefined` invece di eseguire l'operazione tra lei e la user defined, come farebbe nel caso di un'altra built-in, richiama il metodo dell'operazione inversa (stessa operazione, ma con gli operandi invertiti) della user-defined passandosi come argomento del metodo e poi ritornando il risultato ritornato dall'operazione svolta dalla user-defined.

Per semplificare la comprensione si riporta qua sotto un esempio con i relativi frammenti di codice riguardanti la moltiplicazione tra una *ConstantInteger* ed un *ConstantVectorInteger*.

Codice del metodo multiplication() di ConstantInteger:

```
public Constant multiplication(Constant parConst) throws OperationNotSupportedException,
ConstantCastException {
    if (parConst.getClass() == ConstantInteger.class)
        return new ConstantInteger(this.integerValue * parConst.getValueInt());
    else if (parConst.getClass() == ConstantFloat.class)
        return new ConstantFloat((double)this.integerValue * parConst.getValueFloat());
    else if (parConst.isUserDefined())
        return parConst.multiplicationInverse(this);
    else if (parConst.getClass() == ConstantNull.class)
        return new ConstantNull();
    else
        return super.multiplication(parConst);
}
```

Codice del metodo multiplicationInverse() di ConstantVectorInteger:

```
public Constant multiplicationInverse(Constant parConst) throws ConstantCastException {
    if (parConst.getClass() == ConstantInteger.class) {
        int[] parArray = new int[vectorValue.length];
        for (int i = 0; i < vectorValue.length; i++) {
            parArray[i] = parConst.getValueInt() * vectorValue[i];
        }
        return new ConstantVectorInteger(parArray);
    }
    else
        return new ConstantNull();
}
```

Poniamo quindi di avere un oggetto *ConstantInteger* contenente il valore intero 4 chiamato *constantFour* e di avere un oggetto *ConstantVectorInteger* con un vettore di due elementi interi [3, 5] chiamato *constantVec* e si voglia procedere con la moltiplicazione di *constantFour* con *constantVec*, allora si scriverà il codice:

```
constantFour.multiplication(constantVec);
```

Sicchè *constantVec*, passando nei controlli di tipo del metodo multiplication di *constantFour* ritornerà **TRUE** a *constantVec.isUserDefined()* verrà richiamato il suo metodo di moltiplicazione inversa e *constantFour* passerà se stessa come argomento:

```
constantVec.multiplicationInverse(constantFour);
```

constantVec eseguirà quindi i calcoli a lui noti per eseguire la moltiplicazione con un intero e ritornerà a *constantFour* un nuovo oggetto *Constant* contenente il risultato dell'operazione e a sua volta il metodo *multiplication* di *constantFour* ritornerà l'oggetto ricevuto da *constantVec*.

I metodi di operazione inversa eseguono l'operazione invertendo l'ordine degli operandi per far sì che il risultato sia corretto anche quando l'operazione non gode della proprietà commutativa.

Controllo di tipo tra costanti user-defined e built-in

Per far sì che una costante estendente *ConstantBuiltin* possa sapere che tipo ritorna una sua operazione con una costante estendente *ConstantUserDefined*, che non è nota in fase di scrittura del codice (e quindi non si sa che operazioni e relative operazioni inverse avremo a disposizione), ci affidiamo alla tecnica della reflection di Java™ che permette di richiamare metodi di classi che non sono note in fase di scrittura del codice, ma solamente a runtime.

un esempio del codice utilizzato da *ConstantInteger* per il controllo del tipo ritornato da una moltiplicazione con una costante user-defined è:




```
return (Class<? extends Object>) parConst.getMethod("multiplicationInverseResultType",  
                                                    Class.class).invoke(null, ConstantInteger.class);
```

Come possiamo vedere *ConstantInteger* passa la propria classe tramite la reflection al metodo *multiplicationInverseResultType()* della costante user-defined a noi non nota.

Constant Factory

La Constant factory è una classe che si occupa della creazione automatica di costanti tramite il passaggio della loro definizione in stringa (ricordiamo che obbligatoriamente ogni costante implementa un costruttore che accetta come parametro una stringa che poi viene analizzata ed utilizzata per l'istanziazione della costante).

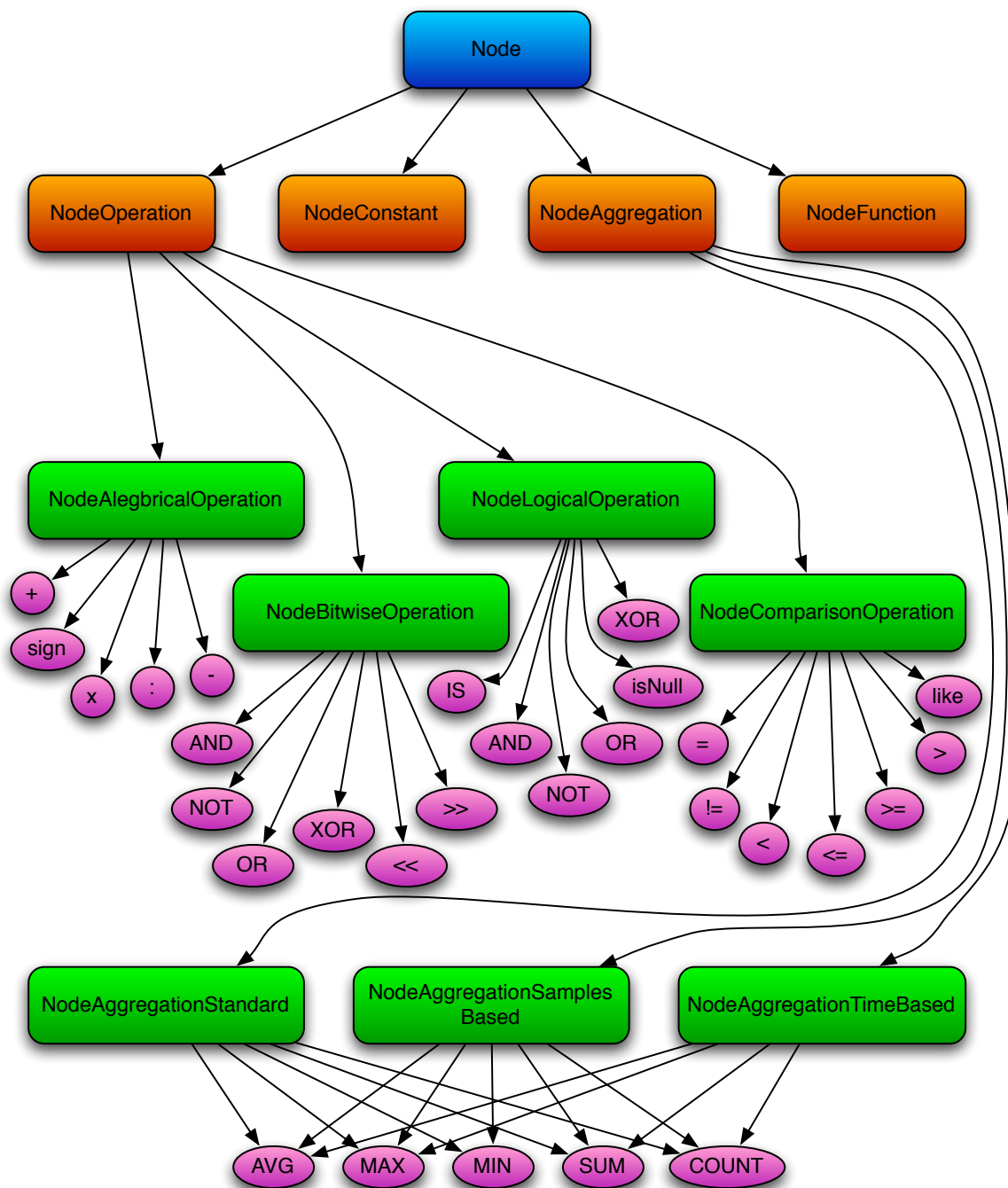
I metodi della Constant Factory sono statici sicché sia possibile richiamarli senza creare un oggetto ConstantFactory vero e proprio.

 <code>org.dei.perla.parser.expressions.ConstantFactory</code>
 <code><u>create(parDataType: FieldType, parValue: String): Constant</u></code>
 <code><u>create(parDataType: FieldType): Constant</u></code>

Nodi

Introduzione

I nodi, che estendono tutti la superclasse Node, sono gli elementi di base che vanno a formare l'albero dell'espressione. Sono essenzialmente di 4 tipi: nodi operazione (NodeOperation), nodi costanti (NodeConstant), nodi aggregati (Node Aggregation) e nodi funzione (NodeFunction).



Nodi Operazione (NodeOperation)

Ogni nodo associa secondo una determinata operazione 2 nodi (in casi particolari solo no o anche 3), che sono le radici dei due sottoalberi che andranno a generare gli operandi su cui verrà eseguita l'operazione. Benché ogni nodo rappresenti una particolare operazione esso si limita ad invocare il metodo di quella operazione dalla costante ritornata dal suo primo nodo passando come argomento la costante ritornata dal secondo nodo.



Il metodo `getResultType()` permettere di conoscere il tipo ritornato dal calcolo senza dover realmente valutare l'espressione, `isValid` invece, sempre senza valutare l'espressione, si assicura che un eventuale valutazione non generi eccezioni o errori dovuti a combinazioni errate di tipi (ad esempio la somma di una stringa per un intero). Infine il metodo `getResult()` richiama il `getResult()` del primo nodo e, sulla costante ritornata dall'operazione, richiama il metodo relativo all'operazione rappresentata dal nodo passando come argomento la costante ritornata dal metodo `getResult()` del secondo nodo; un esempio di `getResult()` del nodo addizione è riportato qua in basso:

```
public Constant getResult() {  
    return this.getFirstNode().getResult().addition(this.getSecondNode().getResult());  
}
```


Nodi Costante (NodeConstant)

I nodi costante (o ConstantNode) sono utilizzate unicamente come foglie dell'albero e come variabile hanno un unico oggetto Constant, il metodo getResult() ritorna banalmente l'oggetto Constant memorizzato, getResultType() invece ritorna la classe di appartenenza dell'oggetto costante.



Nodi Aggregazione (Node Aggregation)

I nodi aggregazione rappresentano le operazioni di aggregazione tra espressioni, le operazioni di: valore massimo (MAX), valore minimo (MIN), valore medio (AVG), somma totale (SUM) e conteggio (COUNT). I Nodi aggregati contendono due nodi contenenti ognuno un'espressione, una è l'espressione da aggregare e l'altra è l'espressione di filtro.

I tipi di aggregazione sono fondamentalmente tre:

- Standard, ovvero come accadrebbe nell'SQL standard
- Time based, ovvero basata sui tempi di campionamento dei sensori della WSN (Wireless Sensor Network)
- Sample based, ovvero basata sui singoli campioni provenienti dai sensori della WSN.

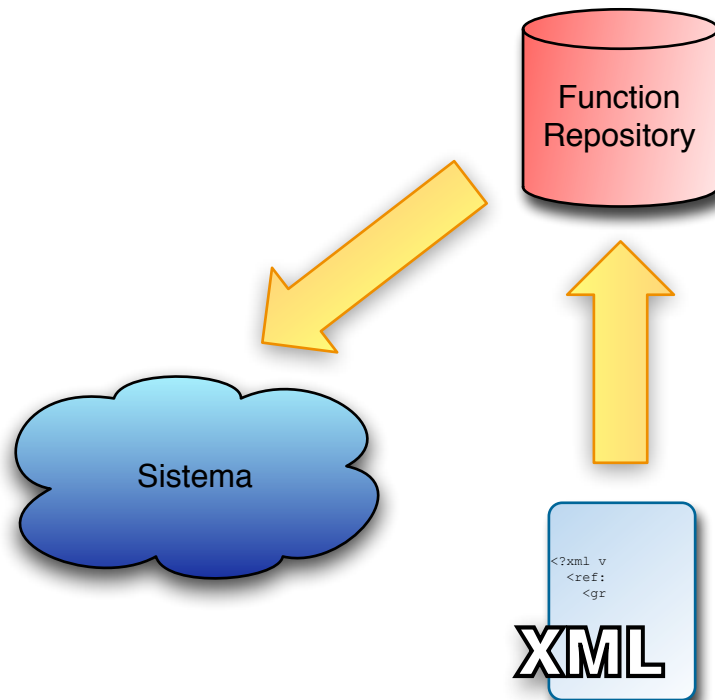
Nodi Funzione (NodeFunction)

I nodi funzione contengono come variabile un oggetto funzione (Function) che implementa una funzione che, dati dei parametri in ingresso, restituisce un'uscita.

Anche in questo caso la chiamata di getResult() del nodo provocherà il ritorno di un oggetto Constant contenente il risultato della valutazione della funzione. Le funzioni non sono inserite di default nel sistema ma vengono create dall'utente estendendo la classe Function e scrivendone la definizione in un file XML che verrà letto dal repository di funzioni (FunctionRepository) che le renderà disponibili a runtime.

Nel file XML vanno specificati il nome della funzione, i parametri ed il tipo ritornato; un esempio di file XML di definizione è riportato qua sotto:

```
<functions>
  <function>
    <name>Funzione 1</name>
    <parameter>
      <parName>Par1</parName>
      <parType>org.dei.perla.parser.expressions.ConstantVectorInteger</parType>
    </parameter>
    <returnedType>org.dei.perla.parser.expressions.ConstantVectorInteger</returnedType>
  </function>
</functions>
```

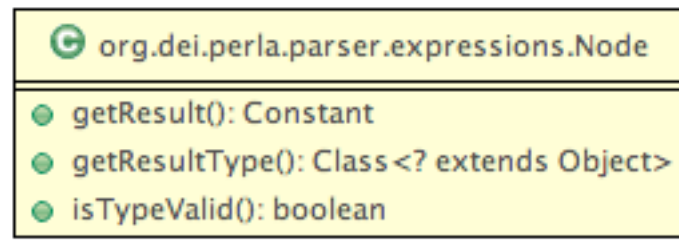


Classi dei Nodi

Superclasse Node

La superclasse astratta Node viene estesa da tutte le sottoclassi di tipo nodo e fornisce le definizioni dei metodi comuni a tutti i nodi.

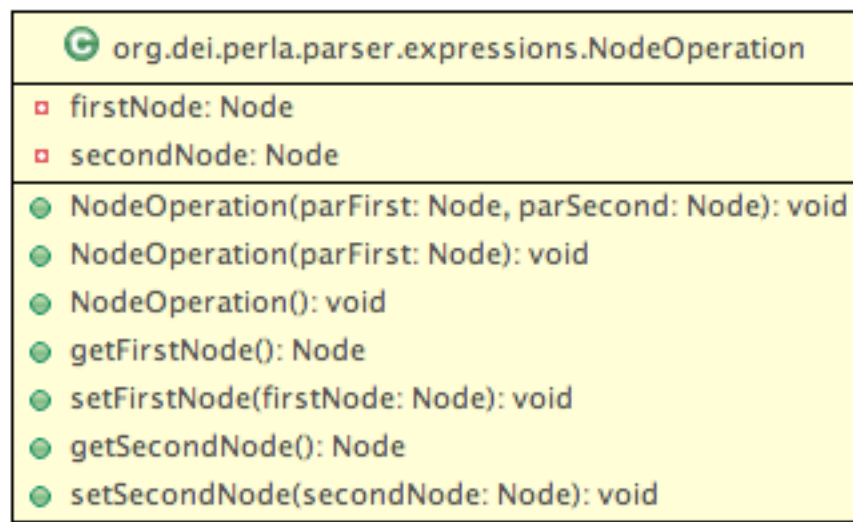
Diagramma UML



Classe NodeOperation

La superclasse astratta che viene estesa da tutti i nodi che eseguono operazioni. Estende Node.

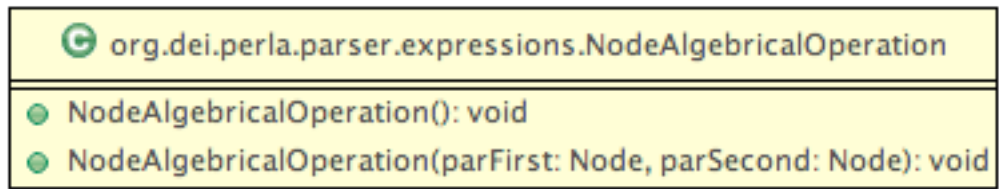
Diagramma UML



Classe NodeAlgebricalOperation

La superclasse astratta che viene estesa da tutti i nodi che eseguono operazioni algebriche. Estende NodeOperation.

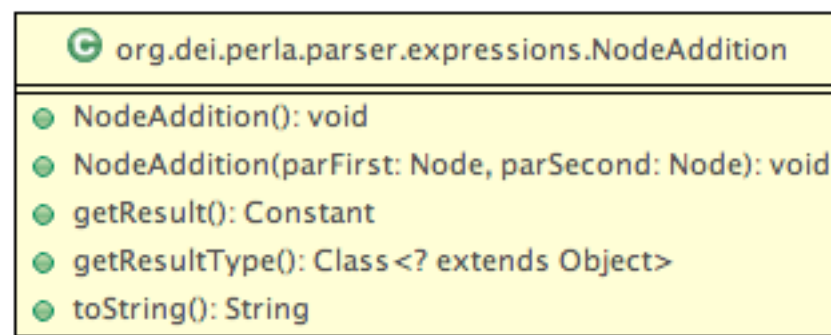
Diagramma UML



Classe NodeAddition

La classe NodeAddition rappresenta un nodo con l'operazione di addizione. Estende NodeAlgebricalOperation.

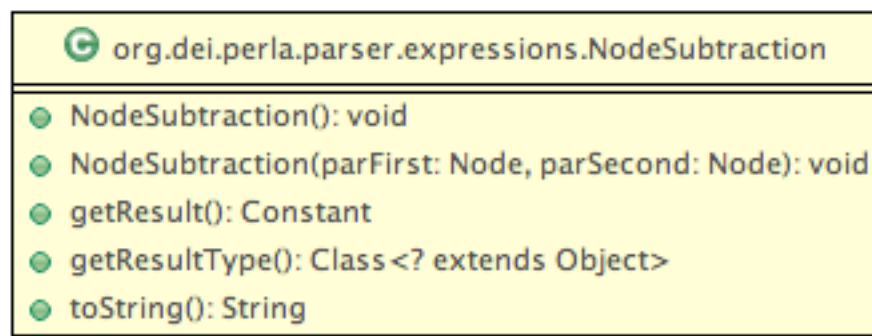
Diagramma UML



Classe NodeSubtraction

La classe NodeSubtraction rappresenta un nodo con l'operazione di sottrazione. Estende NodeAlgebricalOperation.

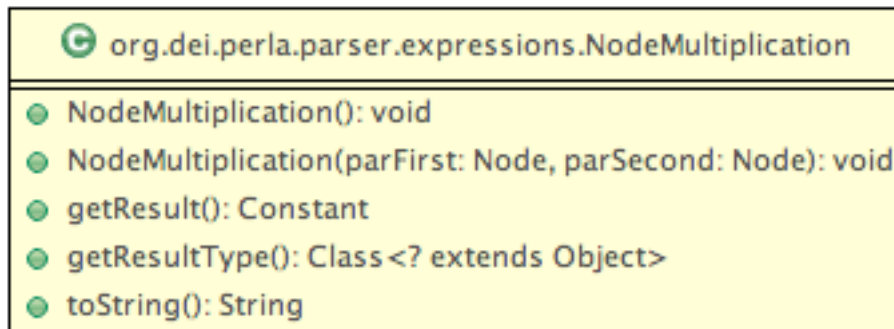
Diagramma UML



Classe NodeMultiplication

La classe NodeMultiplication rappresenta un nodo con l'operazione di moltiplicazione. Estende NodeAlgebraicalOperation.

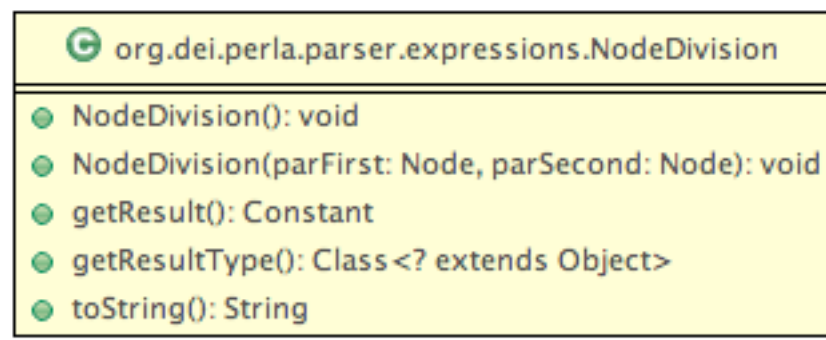
Diagramma UML



Classe NodeDivision

La classe NodeDivision rappresenta un nodo con l'operazione di divisione. Estende NodeAlgebraicalOperation.

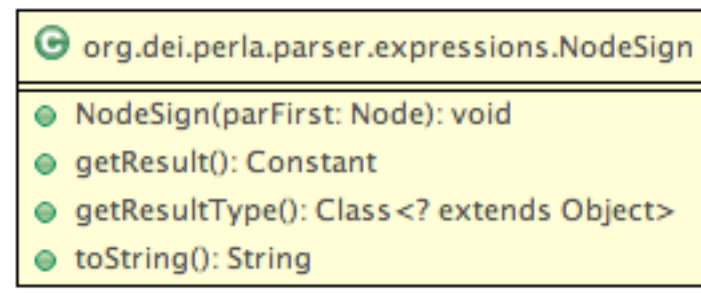
Diagramma UML



Classe NodeSign

La classe NodeSign rappresenta un nodo con l'operazione di inversione di segno. Estende NodeAlgebricalOperation.

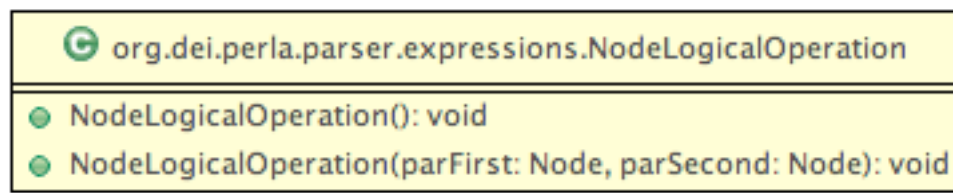
Diagramma UML



Classe NodeLogicalOperation

La classe astratta NodeLogicalOperation viene estesa da tutte le classi nodo che rappresentano un'operazione logica (che ritorna una ConstantLogic). Estende NodeOperation.

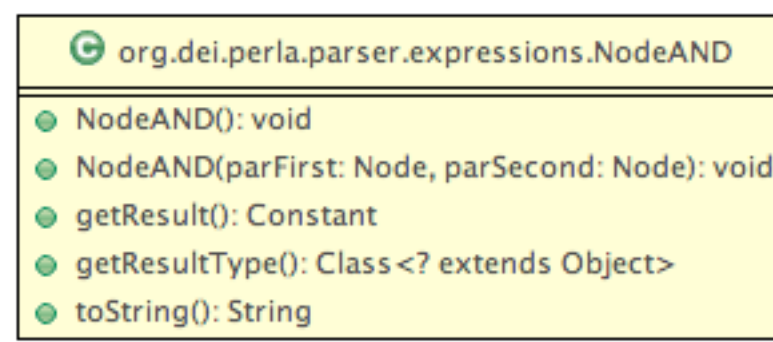
Diagramma UML



Classe NodeAND

La classe NodeAND rappresenta un nodo con l'operazione di AND. Estende NodeLogicalOperation.

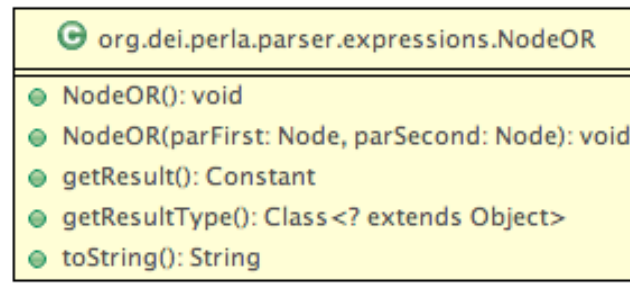
Diagramma UML



Classe NodeOR

La classe NodeOR rappresenta un nodo con l'operazione di OR. Estende NodeLogicalOperation.

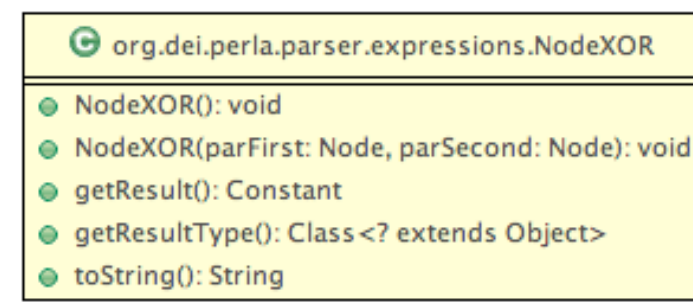
Diagramma UML



Classe NodeXOR

La classe NodeXOR rappresenta un nodo con l'operazione di XOR. Estende NodeLogicalOperation.

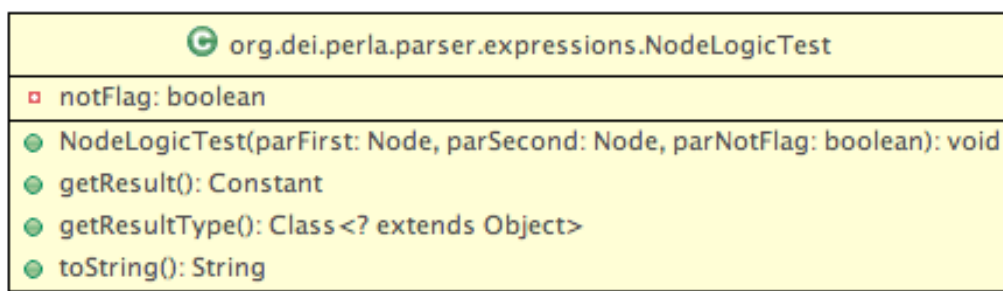
Diagramma UML



Classe NodeLogicTest

La classe NodeLogicTest rappresenta un nodo con l'operazione di IS (IS standard SQL). Estende NodeLogicalOperation.

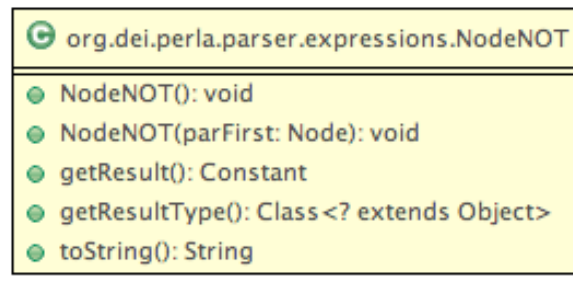
Diagramma UML



Classe NodeNOT

La classe NodeNOT rappresenta un nodo con l'operazione di NOT. Estende NodeLogicalOperation.

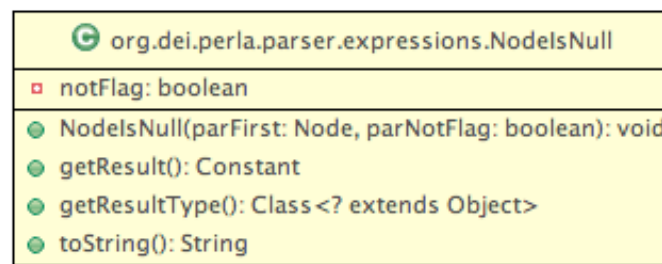
Diagramma UML



Classe NodeIsNull

La classe NodeIsNull rappresenta un nodo con l'operazione di controllo di nullità. Estende NodeLogicalOperation.

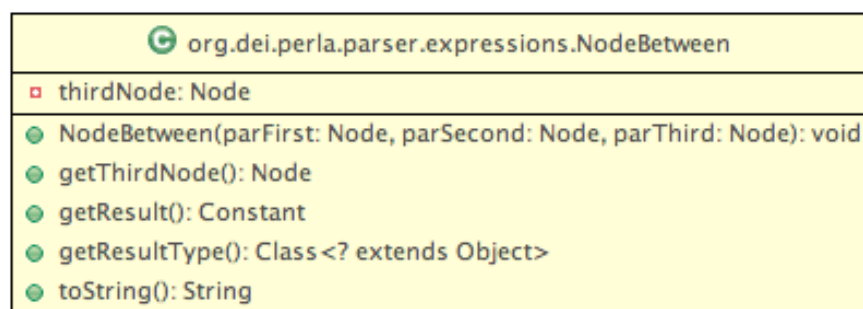
Diagramma UML



Classe NodeBetween

La classe NodeBetween rappresenta un nodo con l'operazione di controllo del fatto che un numero stia tra altri due (between SQL). Estende NodeLogicalOperation.

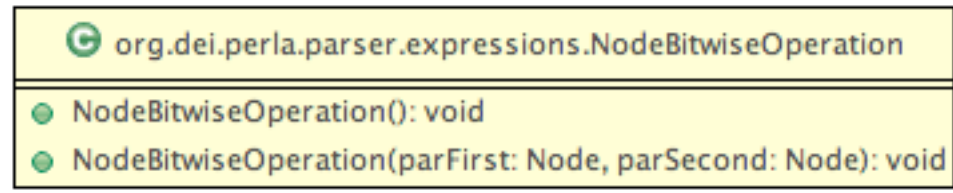
Diagramma UML



Classe NodeBitwiseOperation

La classe astratta NodeBitwiseOperation viene estesa da tutte le classi nodo che rappresentano un'operazione bitwise. Estende NodeOperation.

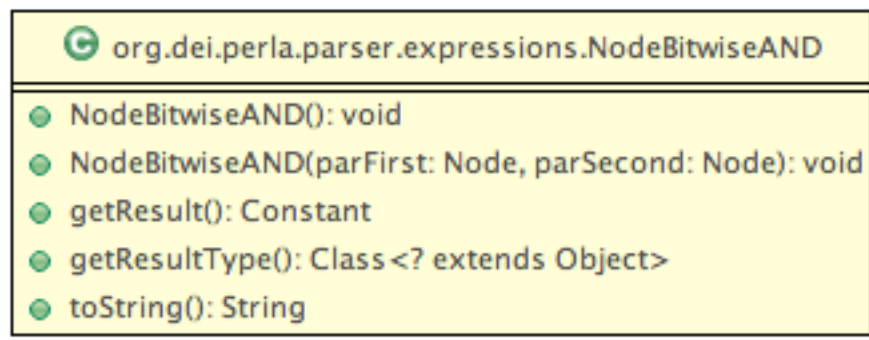
Diagramma UML



Classe NodeBitwiseAND

La classe NodeBitwiseAND rappresenta un nodo con l'operazione di and bit a bit. Estende NodeBitwiseOperation.

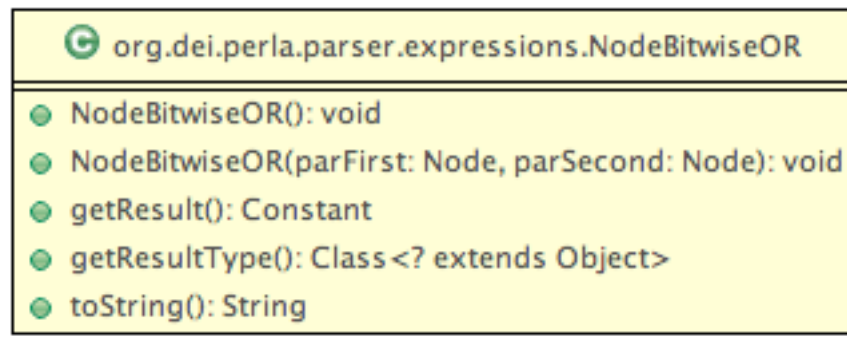
Diagramma UML



Classe NodeBitwiseOR

La classe NodeBitwiseOR rappresenta un nodo con l'operazione di or bit a bit. Estende NodeBitwiseOperation.

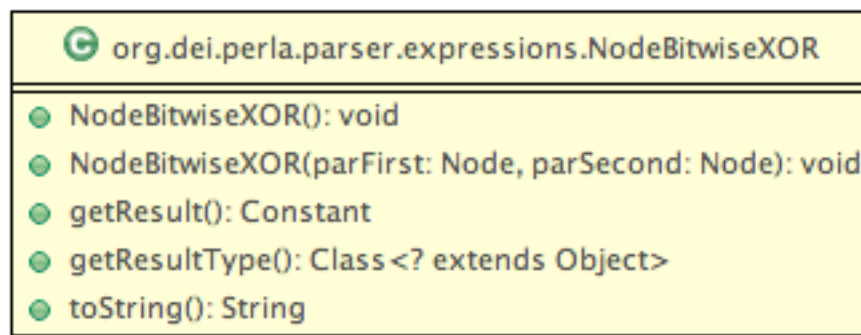
Diagramma UML



Classe NodeBitwiseXOR

La classe NodeBitwiseXOR rappresenta un nodo con l'operazione di xor bit a bit. Estende NodeBitwiseOperation.

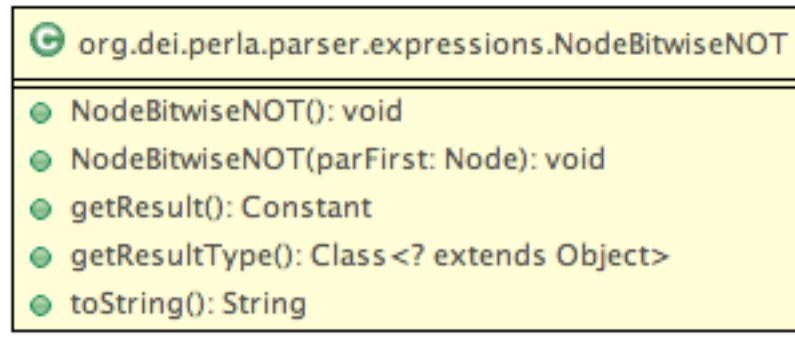
Diagramma UML



Classe NodeBitwiseNOT

La classe NodeBitwiseNOT rappresenta un nodo con l'operazione di not bit a bit. Estende NodeBitwiseOperation.

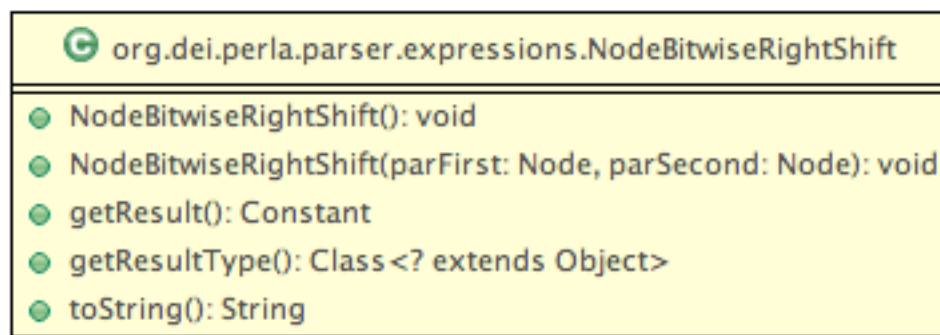
Diagramma UML



Classe NodeBitwiseRightShift

La classe NodeBitwiseRightShift rappresenta un nodo con l'operazione di right shift bit a bit. Estende NodeBitwiseOperation.

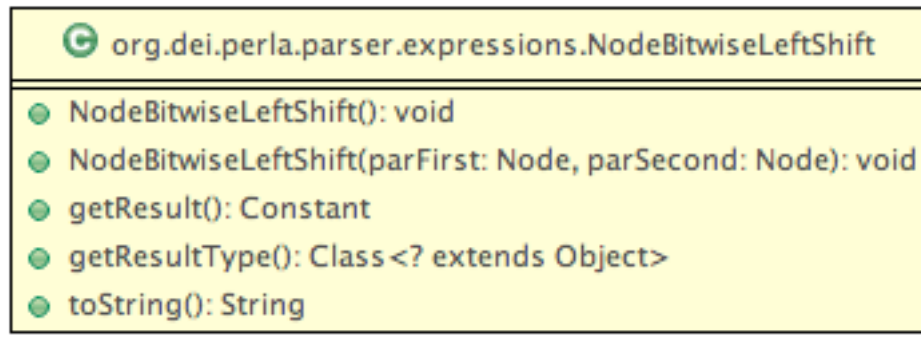
Diagramma UML



Classe NodeBitwiseLeftShift

La classe NodeBitwiseLeftShift rappresenta un nodo con l'operazione di left shift bit a bit. Estende NodeBitwiseOperation.

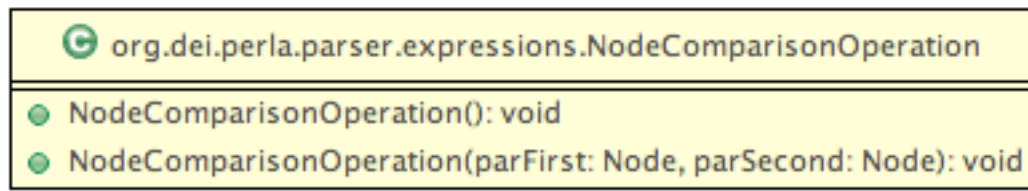
Diagramma UML



Classe NodeComparisonOperation

La classe astratta NodeComparisonOperation viene estesa da tutte le classi nodo che rappresentano un'operazione di comparazione. Estende NodeOperation.

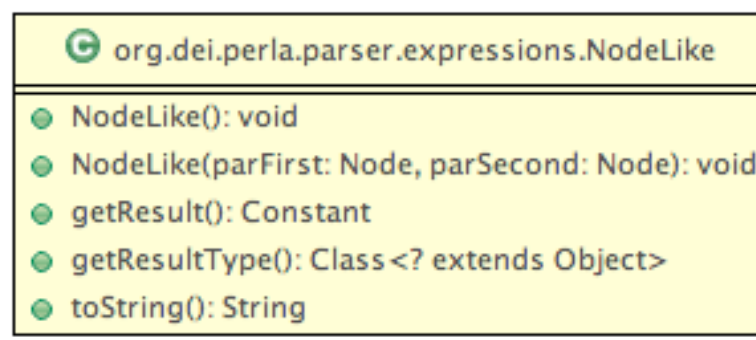
Diagramma UML



Classe NodeLike

La classe NodeLike rappresenta un nodo con l'operazione like dell'SQL standard su una stringa. Estende NodeComparisonOperation.

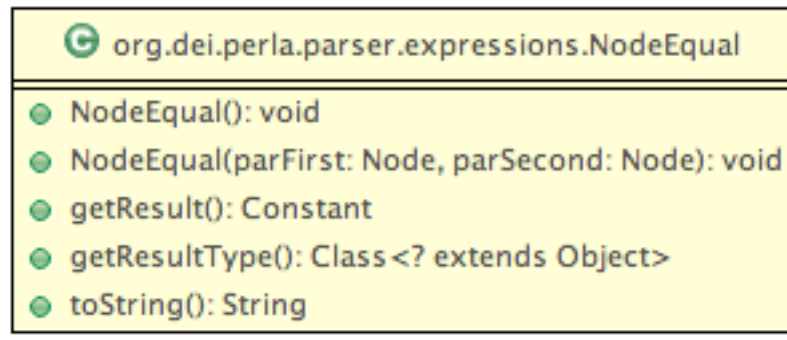
Diagramma UML



Classe NodeEqual

La classe NodeEqual rappresenta un nodo con l'operazione di confronto di uguaglianza. Estende NodeComparisonOperation.

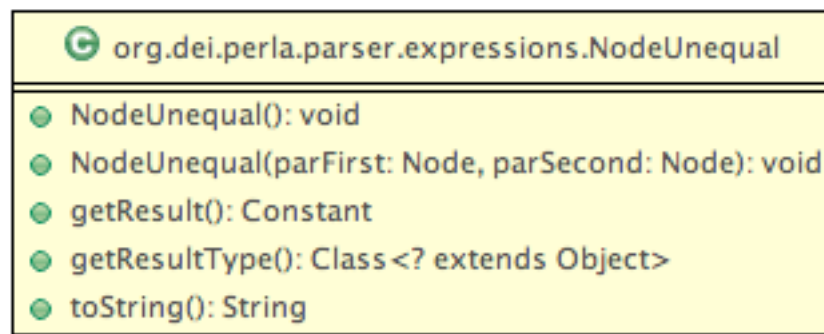
Diagramma UML



Classe NodeUnequal

La classe NodeUnequal rappresenta un nodo con l'operazione di confronto di disuguaglianza. Estende NodeComparisonOperation.

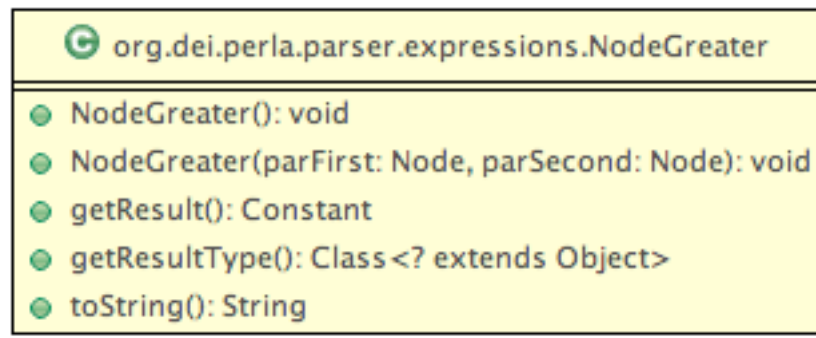
Diagramma UML



Classe NodeGreater

La classe NodeGreater rappresenta un nodo con l'operazione di confronto di maggioranza non stretta (maggiore o uguale). Estende NodeComparisonOperation.

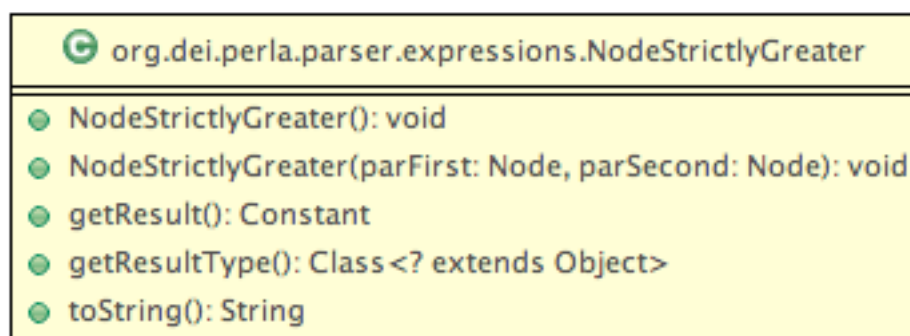
Diagramma UML



Classe NodeStrictlyGreater

La classe NodeStrictlyGreater rappresenta un nodo con l'operazione di confronto di maggioranza stretta (maggiore ma non uguale). Estende NodeComparisonOperation.

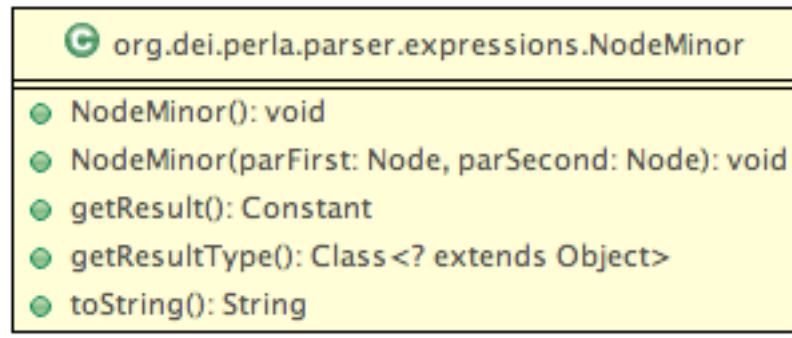
Diagramma UML



Classe NodeMinor

La classe NodeMinor rappresenta un nodo con l'operazione di confronto di minoranza non stretta (minore o uguale). Estende NodeComparisonOperation.

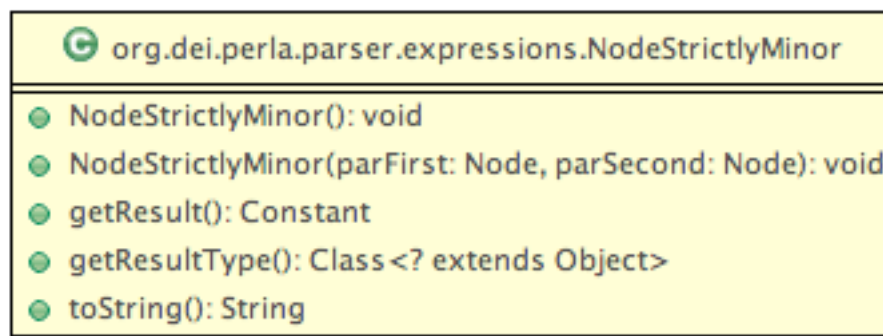
Diagramma UML



Classe NodeStrictlyMinor

La classe NodeStrictlyMinor rappresenta un nodo con l'operazione di confronto di minoranza stretta (minore ma non uguale). Estende NodeComparisonOperation.

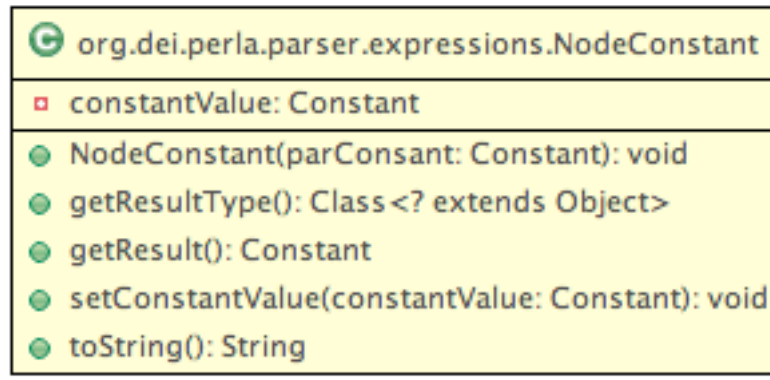
Diagramma UML



Classe NodeConstant

La classe NodeConstant contiene una classe Constant e funge da foglia dell'albero dell'espressione. Estende Node.

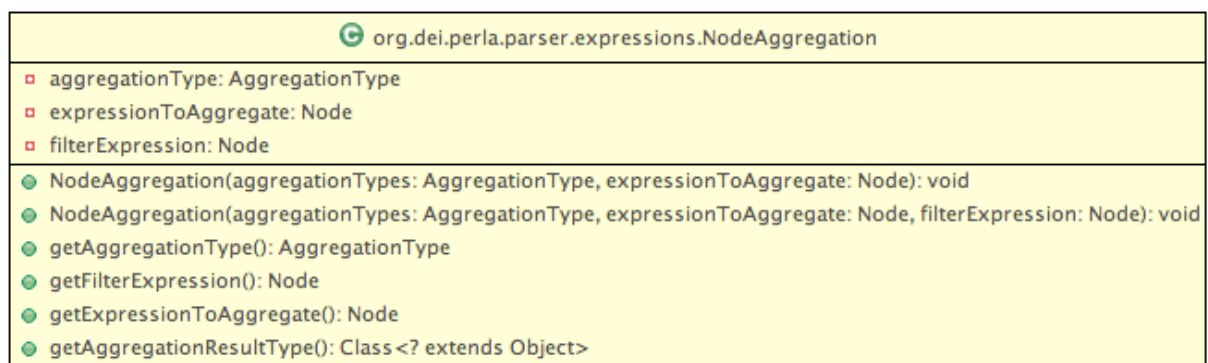
Diagramma UML



Classe NodeAggregation

La classe astratta NodeAggregation funge da superclasse per tutte le classi di aggregazione. Estende Node.

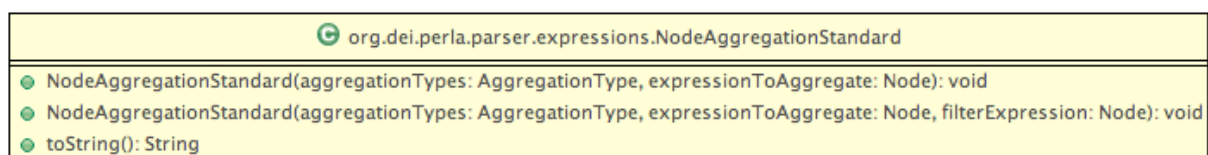
Diagramma UML



Classe NodeAggregationStandard

La classe NodeAggregationStandard funge da classe per le operazioni di aggregazione stanandard (come un aggregato SQL). Estende NodeAggregation.

Diagramma UML



Classe NodeAggregationTimeBased

La classe NodeAggregationTimeBased funge da classe per le operazioni di aggregazione basati sui tempi di campionamento dei sensori della WSN. Estende NodeAggregation.

Diagramma UML

org.dei.perla.parser.expressions.NodeAggregationTimeBased
interval: Duration
NodeAggregationTimeBased(interval: Duration, aggregationTypes: AggregationType, expressionToAggregate: Node): void
NodeAggregationTimeBased(interval: Duration, aggregationTypes: AggregationType, expressionToAggregate: Node, filterExpression: Node): void
getInterval(): Duration
toString(): String

Classe NodeAggregationSampleBased

La classe NodeAggregationSampleBased funge da classe per le operazioni di aggregazione basati sui campioni provenienti dai sensori della WSN. Estende NodeAggregation.

Diagramma UML

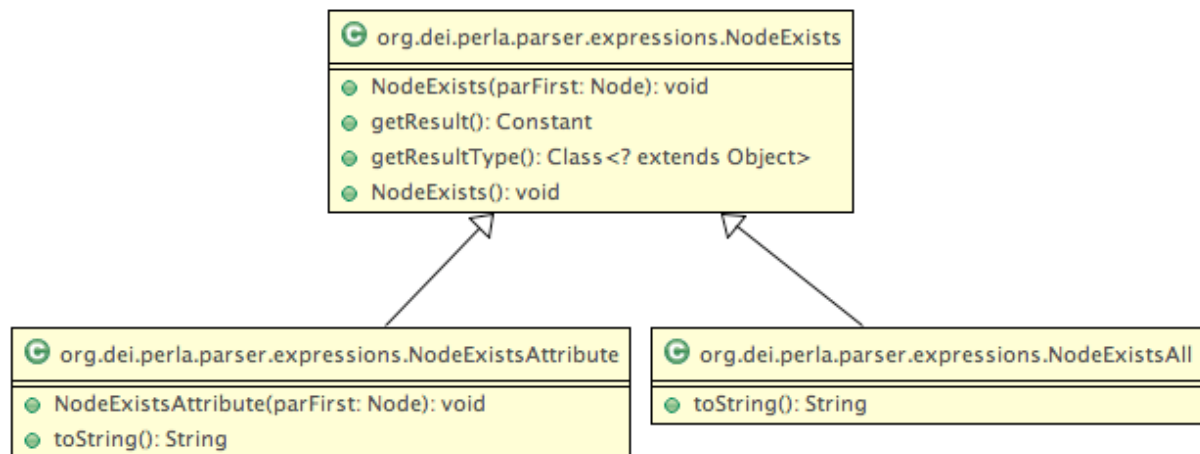
org.dei.perla.parser.expressions.NodeAggregationSamplesBased
samples: int
NodeAggregationSamplesBased(samples: Integer, aggregationTypes: AggregationType, expressionToAggregate: Node): void
NodeAggregationSamplesBased(samples: Integer, aggregationTypes: AggregationType, expressionToAggregate: Node, filterExpression: Node): void
getSamples(): int
toString(): String

Classe NodeExists

La classe astratta NodeExists funge da superclasse per le classi di controllo sull'esistenza di oggetti logici (astrazione software dei sensori della WSN) e dei loro eventuali attributi (sincerarsi per esempio che ci sia un attributo temperatura). Estende NodeOperation.

Diagramma UML

org.dei.perla.parser.expressions.NodeExists
NodeExists(parFirst: Node): void
getResult(): Constant
getResultType(): Class <? extends Object>
NodeExists(): void



Classe NodeExistsAttribute

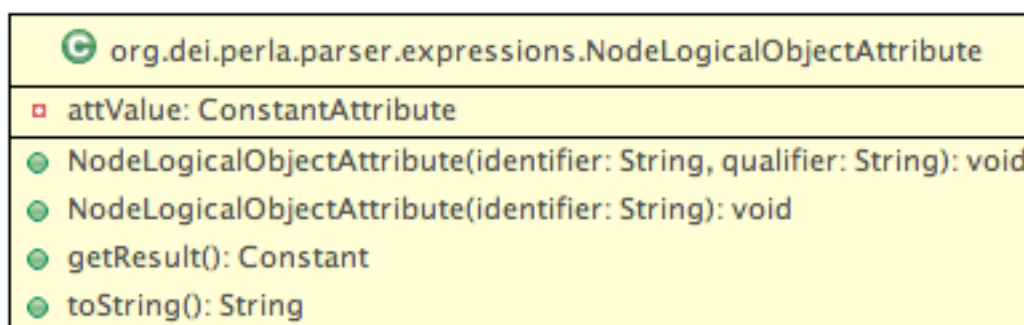
La classe `NodeExistsAttribute` è adibita al controllo della presenza di determinati attributi (ex. temperatura, pressione, ecc...) negli oggetti logici (astrazione software dei dispositivi collegati alla WSN). Estende

Classe NodeExistsAll

La classe `NodeExistsAll` è adibita al controllo della presenza di un intero set di attributi (ex. temperatura, pressione, ecc...) negli oggetti logici.

Classe NodeLogicalObjectAttribute

La classe rappresenta un attributo di un oggetto logico (la temperatura, la pressione, ecc...). Estende `Node`.



Eccezioni

Introduzione

Per gestire tutti i possibili errori generati dal sistema, errori che vanno dalla divisione per zero a calcoli tra tipi incompatibili, è stato creato un insieme di eccezioni che permettono di individuare con esattezza dove il problema si è verificato. E' altresì vero che i sistemi di controllo sul tipo riducono al minimo la possibilità di imbatterci.

Il sistema è dotato di cinque diversi tipi di eccezione:

- `DivisionByZeroException`
- `ConstantRuntimeException`
- `ConstantCastException`
- `ConstantCreationException`
- `OperationNotSupportedException`

Le eccezioni

`DivisionByZeroException`

Come è facile intuire questa eccezione viene lanciata unicamente da quelle classi discendenti di `Constant` che supportano la divisione, e viene lanciato nel caso si tenti di effettuare una divisione per zero che, come ben si sa, è impossibile.

`ConstantRuntimeException`

Questa eccezione viene lanciata nel caso nella costante avvenga un errore non coperto da alcuna delle altre eccezioni.

`ConstantCastException`

Questa eccezione viene lanciata nel caso in cui viene tentato di trasformare una `Constant` in un tipo da lei non supportato per il cast (se ad esempio si tenta di trasformare la stringa "ciao" in un intero si provocherebbe il lancio di questa eccezione).

`ConstantCreationException`

Questa eccezione viene lanciata dalla `Constant Factory` quando incontra un errore nella creazione di una costante.

`OperationNotSupportedException`

Questa eccezione viene lanciata quando si tenta di effettuare un'operazione non supportata tra due `Constant` (ad esempio se si prova a moltiplicare una stringa per un float).

Conclusioni

In questo lavoro si è cercato di realizzare un sistema di rappresentazione e valutazione delle espressioni completo e funzionale per il Query analyzer del linguaggio PERLA. Particolare attenzione è stata prestata al sistema di costanti User-Defined in quanto si è ritenuto che la possibilità di aggiungere tipi personalizzati di costanti al set di default avrebbe aumentato notevolmente la potenza e la versatilità del linguaggio. E' stata anche ovviamente prestata particolare attenzione al fatto di rendere il più semplice possibile l'aggiunta di tipi di costanti all'utente finale creando metodi appositi per l'interazione delle costanti builtin con le user-defined e stendendo le linee guida per la loro creazione.

Stato dell'arte

Ad ora per il progetto PERLA sono state completate la stesura della grammatica, il parser delle query, il sistema di classi per la rappresentazione interna delle query ed il sistema di valutazione delle espressioni. In fase di progettazione troviamo invece lo sviluppo dei driver C per i sensori, i driver JAVA per l'interazione con l'esecutore delle query, il Functionality proxy component ed il Low Level Query Executor.

Bibliografia

[1] <http://artdeco.elet.polimi.it>

[2] M. Fortunato, M. Marelli “Design of a declarative language for pervasive systems”

[3] C. Horstmann “Java: Fondamenti”

[4] C. Horstmann “Java: Tecniche Avanzate”