Technologies for Information Systems – a. y. 2007 - 2008

Parser Design for PERLA query language

Project by:

Prazzoli Pierpaolo MAT: 708456

Rota Guido MAT: 711294

INDEX

1 - INTRODUCTION

1.1 - PERLA

This part of the document will give the reader a general overview over PERLA infrastructure, starting with some general information about the language and finishing with the query execution flow.

PERLA [1, 2] is a declarative query language for pervasive systems. It has been designed to support heterogeneity and to be easily deployable on most wireless sensor networks. The SQL-like syntax and semantics result in a flat learning curve for users already experienced with other query languages.

The data structures we can work with are divided into two categories: streams and snapshots.

Stream tables are unbounded tables that associate a native timestamp to every record. This kind of table supports the operations of field insertion and field reading. Each insertion operation generates an event that notifies any subquery waiting for data.

Snapshot tables are data structures needed to perform pilot join operations. They are characterized by an internal buffer (write only) and an output buffer (read only). Every time a new record is added to a snapshot, the insertion operation is made in the local buffer. The content of the local buffer is moved to the output buffer with frequency F. Every read operation on a snapshot returns the current content of the output buffer.

Every DataStructure can be instantiated using a CREATE statement, whose syntax is very similar to the classic SQL CREATE.

Interrogations to wireless sensor networks are made using an SQL-like syntax, and can be divided in two groups: low level and high level queries.

Low level queries are used to define sampling operations on a single logical object (a single device or a group of devices abstracted as a single one). They allow the user to specify sampling periods, aggregations, filtering operations, etc. The results of a low level query can be sent either to a stream or a snapshot. Many low level queries can insert their output in a single snapshot (therefore performing a union).

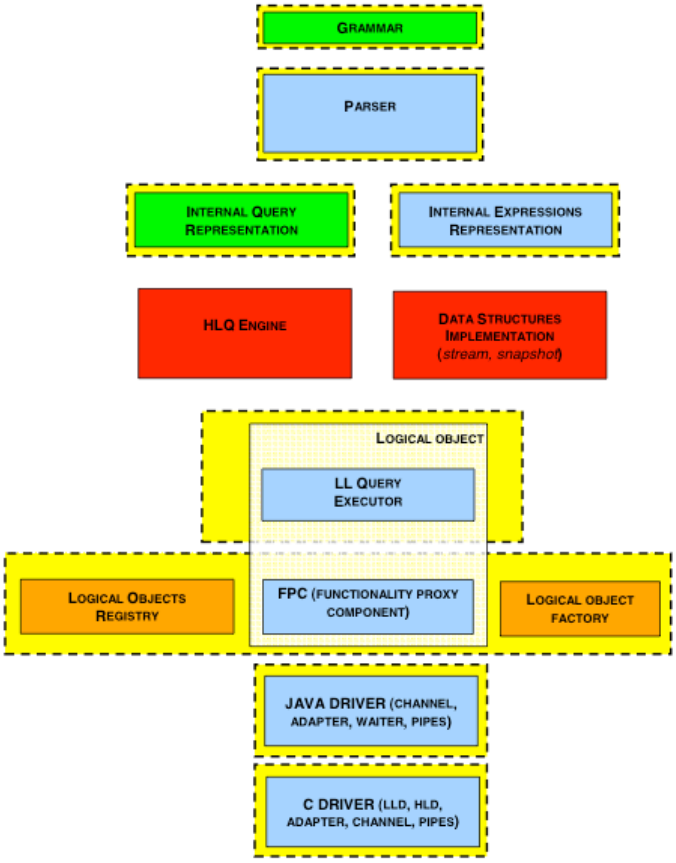
High level queries take one or more streams (generated by other queries) and perform SQL operations on them. They can be activated either on a time-based schedule or following an event-based semantic.

PERLA supports physical heterogeneity using a device abstraction layer. When a new sensor is plugged in the network, its controller sends an XML device descriptor to the system. This descriptor

is used by an object factory to instantiate a high level representation of the concrete device, called Logical Object, that is used to mask all the low-level communication and querying problems.
Every time a query is sent by the user, the system performs a translation process to obtain an object-oriented representation of the interrogation. This step is performed by the parser (more about this later). The low level part of the Query Object is then sent to the Executor component, which undertakes all the actions needed to retrieve the desired information from the sensors. This part of the system doesn't access directly any physical device, but defers all the low level communication problems to the logical objects, that are tailored on each different concrete device using the information contained in the device's XML descriptor. The results coming from the sensors can be used as input for High Level queries or to drive the execution of other Low Level queries (pilot join).

1.2 - Projects

This section is intended as a brief summary about the different ongoing/finished projects related to the PERLA infrastructure. Its only purpose is to give the reader a general knowledge about the various subsystems that compose PERLA, and to explicitly present the boundaries of the work described in this document.



1.3 - The Parser

The parser is part of the query analyzer system, which is the first component that directly interfaces with the user. It performs the following tasks:

- Query validation
- Error checking
- Query translation

The query validation step simply checks if the query is syntactically correct. The error checking system, which works along with the query validation, implements a series of semantic validation steps.
Finally, the query translation subsystem creates the output of the query analyzer, i.e. the Query Object [3]. This is an Object-Oriented representation of the user's input directly used by the executor to know what operations need to be carried out to obtain the desired result.
Our work primarily focused on the extension of the existing parser, which only did the query validation step, adding the semantic error checking layer and the Query Translation system.

2 - PROJECT GOALS

This chapter explains all the goals that led us to the current implementation of the parser. It's fundamental to keep in mind the following objectives to completely understand the design of our implementation.

- **Extend the existing parser to generate the query object structure**: This has been the main goal of our project. The query written by the users needs to be translated in a machine-understandable form before further operations can take place. We extended the original parser (that only did query validation) to create an object structure that represents the query given as input.
- **Keep the grammar as clean as possible**: Maintainability and extendibility are key issues in this project. In order to enable future developers to extend the language with new constructs, we moved most of the code needed to generate the query object outside the grammar file.
- **Error checking and reporting**: To complete our project we had to implement a semantic error-checking system. Moreover we designed and implemented an error-reporting infrastructure.

3 - INTERNAL OPERATIONS

To help the reader to better understand the following sections of the document, we will give some basic information about the grammar and the main classes we introduced to achieve our goals.

As already said in the previous sections, this project aims at extending the existing query parser for the PERLA language.
The parser we began with was created with JavaCC (Java Compiler Compiler) [4], an automatic parser generator. JavaCC takes as input an LL(k) grammar specification file and creates a Java program able to recognize and validate instances of the specified language. The resulting program (the parser) only analyzes and validates the language. Grammar productions are translated as static methods, that are called according to the flow of the parsing procedure. Syntactic errors cause the parser to halt and to throw an exception. Every additional feature (semantic validation, translations, etc.) must be manually coded inside the grammar file using Java syntax. When JavaCC is invoked to generate the parser, these blocks of code are taken as they are and inserted in the parser class.
There are, however, two major drawbacks in this system:

- the JavaCC file contains both the grammar specification and the Java code needed for the additional functionalities of the parser
- JavaCC plugin for Eclipse IDE [5] doesn't provide any form of Java error checking. Debugging is therefore more difficult and time consuming, since the parser has to be recompiled every whenever the Java code included in the grammar is changed

To preserve the grammar understandability and to take advantage from Eclipse's error checking and debugging functionalities, we decided to implement all the additional features outside the grammar. Only the code needed to interface with the external classes is left in the JavaCC grammar file.
The implementation of the system needed to create the query object has been designed using two different sets of classes, located outside the grammar:

- `Context` classes: they keep the current state of the parsing, storing already instantiated fragments of the query object
- `Handler` classes: these classes contain the logic needed to create query object fragments

These classes are mainly composed by static public methods and fields. This design decision has been made to further shrink the size of the code included in the grammar file. In fact result values of grammar productions can only be assigned to a variable, but cannot be used directly in a function call. A classic approach (non static classes and private fields with getters and setters) would significantly increase the lines of Java code in the grammar, since storing a simple variable in a Context class implies the creation of a temporary variable, the assignment of the production result to this variable and finally an invocation of the desired field setter.

This class structure has also the advantage of being modular. New `Context` and `Handler` classes can be added as needed, and old classes can be easily extended to support new grammar constructs.

Semantic error checking has been implemented inside the Handler classes too. Before creating an object, `Handler`'s methods perform a check to assert if the produced result would be semantically valid. If not, a new error message is stored in the `ErrorPool` class. All the problems found are shown to the user at the end of the parsing, since we decided to continue the parser execution when an error is found.

4 - IMPLEMENTATION DETAILS

This chapter will focus on the details regarding the implementation of the PERLA parser. We will see the internals of the main classes we introduced to translate PERLA queries to their corresponding machine representation. Moreover, in order to ease the work of future PERLA developers, all the techniques used to create query statements will be presented and discussed.

4.1 - Object Creation

As stated in the previous chapter, we decided to split our implementation in different classes. In this section we will focus on `Context` and `Handler` classes, which are used for object creation purposes. `Context` classes are used to allow a production to store statement fragments that cannot be immediately created or that are needed in subsequent computations.
This way we avoid unnecessary parameters passing in the grammar, because productions can use the Context to store temporary objects and retrieve information about parts of the query that have already been parsed.
Since we can find the same PERLA construct repeated in different parts of a single query we need to clear Context classes when their content is no longer needed. We usually do this as soon as the corresponding statement object is created.
On the other hand, most of the code related to object creation resides in `Handler` classes.

Each of the following points focuses on a particular creation technique used in our work:

a) `Handler` classes are used inside productions to create query objects. Their methods are invoked to initialize the query class objects, using the attributes passed as arguments by the production and/or variables stored in appropriate `Context` classes. These functions, when it's needed by the caller, return the created object.

```
SelectClause SelectClause(ExpressionType parExpressionType) :
{}
{
    <KEYWORD_SELECT>
    [
        <KEYWORD_DISTINCT> { ClausesContext.selectionDistinct = true; }
        |
        <KEYWORD_ALL>
    ]
    FieldSelectionList(parExpressionType)

    { return ClausesHandler.getSelectClause(); }
}
```

In this example an object is created and returned using only context variables (`ClausesHandler.getSelectClause()`)

```
/**
 * Costruisce e ritorna la clausola SELECT
 * @return La clausola SelectClause creata
 */
public static SelectClause getSelectClause()
{
    SelectClause select = new SelectClause();

    // recupera l'attributo di distinct dal contesto
    select.setDistinct(ClausesContext.selectionDistinct);

    SelectFieldList list = select.getFields();

    // copia i campi della select dalla lista nel contesto
    for(int i = 0; i < ClausesContext.selectFieldList.size(); i++)
    {
        list.add(ClausesContext.selectFieldList.get(i));
    }

    // Controlla quale tipo di statement stiamo trattando (LL, HL) e cancella il contesto di conseguenza
    if (StatementContext.statementType == StatementType.HL)
        ClausesContext.clear();

    return select;
}
```

`ClausesHandler` method that creates and returns a `SelectClause` object

b) Enumerations and objects that can be create just using the constructor ("basic" objects) are usually created inside a production primarily using a token-based choice to instantiate and to return the appropriate element. If more than a simple constructor invocation is needed, the object creation takes place in a `Handler` class. If all the objects returned by a production are derived from the same super-class or they implement the same interface, the return type is set to the common superclass/interface. `Handler` methods differ from this behavior, since they return the specific type of the instantiated object (except for particular situations where the type distinction is impossible).

```
TimeUnit TimeUnit() :
{ Token t; }
{
    (
        t = <TIMEUNIT_S>
    |
        t = <TIMEUNIT_M>
    |
        t = <TIMEUNIT_H>
    |
        t = <TIMEUNIT_MS>
    |
        t = <TIMEUNIT_D>
    |
        t = <TIMEUNIT_MT>
    )
    { return EnumHandler.getTimeUnitFromName(t.image); }
}
```

Example of a “simple” production which returns the object created in the `Handler`, based on the option chosen with the token

```
AggregationType AggregationOperator() :
{}
{
    <FUNCTION_AVG> { return AggregationType.AVG; }
    |
    <FUNCTION_MIN> { return AggregationType.MIN; }
    |
    <FUNCTION_MAX> { return AggregationType.MAX; }
    |
    <FUNCTION_SUM> { return AggregationType.SUM; }
}
```

Example of a “simple” production where the enumeration object is immediately returned

```
NodeComparisonOperation ComparisonOperator() :
{}
{
    <OPERATOR_GREATER> { return new NodeStrictlyGreater(); }
    |
    <OPERATOR_LESS> { return new NodeStrictlyMinor(); }
    |
    <OPERATOR_GREATER_EQUAL> { return new NodeGreater(); }
    |
    <OPERATOR_LESS_EQUAL> { return new NodeMinor(); }
    |
    <OPERATOR_EQUAL> { return new NodeEqual(); }
    |
    <OPERATOR_NOT_EQUAL> { return new NodeUnequal(); }
}
```

Example of a production that creates objects derived from the same super-class (`NodeComparisonOperation`) which is also used as return type

c) The creation of lists with at least one element is handled calling appropriate functions of the `Context` classes, which create (either using argument parameters or `Context` variables) list of elements that are added to a temporary Collection in the `Context` itself. Everything is handled in the single element production, so the enclosing list production contains no extra code. No return value is used, because all the objects are stored in the `Context` after their creation.

```
void FieldDefinitionList() :
()
{
    "("
    FieldDefinition()
    (
        ","
        FieldDefinition()

    )*
    ")"
}
```

List production, no extra java code needed

```
void FieldDefinition() :
{ Token tokenId; FieldType type; Constant defaultValue = null; }
{
    tokenId = Identifier() /* DataStructureField */
    type = FieldType()
    [
        <KEYWORD_DEFAULT>
        defaultValue = SignedConstant()
    ]
    { StatementContext.addField(tokenId, type, defaultValue); }
}
```

Production of a single element of the list with `Context` function call

```java
/**
 * Aggiunta di un Field a tokenDataStructureFieldList
 * @param tokenName Token contenente di dati del Field
 * @param type Tipo del Field
 * @param defaultValue Costante di default del Field
 */
public static void addField(Token tokenName, FieldType type, Constant defaultValue)

    if(tokenDataStructureFieldList == null) {

        tokenDataStructureFieldList = new ArrayList<PairValue<Token, Field>>();
    }

    PairValue<Token, Field> value = new PairValue<Token, Field>();
    Field field = new Field();
    field.setDefaultValue(defaultValue);
    field.setName(tokenName.image);
    field.setType(type);

    value.setFirst(tokenName);
    value.setSecond(field);

    tokenDataStructureFieldList.add(value);
}
```

`StatementContext` method used to create an element for the `tokenDataStructureFieldList` list, which is used by a `Handler` function to populate a query object attribute

d) Optional elements are handled using null initialized variables, that are set by the production only if the corresponding keywords appear in the query. The functions that receive these variables as parameters have to execute a null test to check how an object has to be created (using defaults or the values specified in the query).

```
ExecuteIfClause ExecuteIfClause() :
{ Node n; RefreshClause refresh = null; }
{
    <KEYWORD_EXECUTE>
    <KEYWORD_IF>
    n = Expression(ExpressionType.LOW_LEVEL_NO_AGGR_NO_PILOT)
    [
        refresh = RefreshClause()
    ]
    { return ClausesHandler.getExecuteIfClause(n, refresh); }
}
```

Production with an optional refresh element that is assigned to null at the beginning of the function

```java
/**
 * Costruisce e recupera la clausola EXECUTE IF
 * @param condition Condizione della clausola
 * @param refresh Clausola di refresh associata
 * @return La clausola ExecuteIfClause creata
 */
public static ExecuteIfClause getExecuteIfClause(Node condition, RefreshClause refresh)

    ExecuteIfClause executeIf = new ExecuteIfClause();
    executeIf.setCondition(condition);

    if(refresh != null) {

        executeIf.setRefreshClause(refresh);
    } else {

        // crea la clausola di refresh di default
        executeIf.setRefreshClause(new RefreshNever());
    }

    return executeIf;
}
```

`ClausesHandler` method used to create and return an `ExecuteIfClause`.

A null test is executed on the refresh option to check if the object attribute has to be set to the default valure or not

e) The creation of query expression classes uses a particular technique needed to deal with nested expressions. This peculiarity is handled with a stack data structure stored inside the `ExpressionContext` class, which enables to work with the right context (always located at the top of the stack). `Context`s inside the stack are instances of `ExpressionContext` (this class cannot be completely static because many `ExpressionContexts` can be simultaneously active). The stack structure helps us to easily suspend and resume the creation of nested query expressions and to have the current context always at the top.

When an expression is created, a function is called to check its validity. If this fails, the expression doesn't conform to the correct semantics and an exception is thrown to immediately stop the parser. Due to ongoing changes to the expression implementation this feature is currently disabled. It can be enabled setting to true the constant *TYPE_VALIDATION*, which is located at the beginning of the Parser class definition inside the grammar file.

The expression creation system works using return values. Expression nodes created in deep productions are returned back to the callers. If no new nodes are added the returned object is left unchanged, otherwise `ExpressionHandler` functions are called to add the new expression nodes.

```
Node Expression(ExpressionType parExpressionType) :
{ Node n; }
{
    { ExpressionsContext.addNewContext(); }

    n = ExpressionBooleanTerm(parExpressionType)
    (
        { ExpressionsContext.addLogicalNode(n, ExpressionTypes.OR); }
        <OPERATOR_OR>
        n = ExpressionBooleanTerm(parExpressionType)
    )*
    {
        n = ExpressionsHandler.getLogicalNode(n, ExpressionTypes.OR);
        ExpressionsContext.removeContext();
        ExpressionsHandler.checkTypeValid(n);
        return n;
    }
}
```

Production of the expressions' "root", with Context life and type check highlighted.

f) More complex productions are handled using one of the following methods:

- calls to overloaded `Handler` functions are used to manage the creation of different objects which derive from the same superclass, which is also the return type of the production;

```
TerminateAfterClause TerminateAfterClause() :
{ Duration interval; int selections; }
{
    <KEYWORD_TERMINATE>
    <KEYWORD_AFTER>
    (
        LOOKAHEAD(2)
        interval = Duration()
        { return ClausesHandler.getTerminateAfterClause(interval); }
        |
        selections = SelectionsNumber()
        { return ClausesHandler.getTerminateAfterClause(selections); }
    )
}
```

`ClausesHandler.getTerminateAfterClause()` method has two possible overloads. We can exploit this technique to make the extra code in the grammar more homogeneous and to use the superclass of the created objects as return type

```java
/**
 * Costruisce e ritorna la clausola di TERMINATE AFTER time-based
 * @param interval Oggetto Duration associato
 * @return La clausola TerminateAfterTimeBased creata
 */
public static TerminateAfterTimeBased getTerminateAfterClause(Duration interval)
{
    TerminateAfterTimeBased terminateClause = new TerminateAfterTimeBased();
    terminateClause.setInterval(interval);
    return terminateClause;
}

/**
 * Costruisce e ritorna la clausola di TERMINATE AFTER selection-based
 * @param selections Numero di selezioni della clausola
 * @return La clausola TerminateAfterSelectionsBased creata
 */
public static TerminateAfterSelectionsBased getTerminateAfterClause(int selections)
{
    TerminateAfterSelectionsBased terminateClause = new TerminateAfterSelectionsBased();
    terminateClause.setSelections(selections);
    return terminateClause;
}
```

Two overloaded methods whose return type corresponds to the created objects' class.

- use of Context classes to store partial objects that will be completed by other productions

```
void SamplingEveryClause() :
{ Node n; TimeUnit time; }
{
    <KEYWORD_EVERY>
    n = Expression(ExpressionType.LOW_LEVEL_NO_AGGR)
    time = TimeUnit()
    { ClausesContext.addSamplingIfEvery(n, time); }
}
```

Results of other productions used to create an object to keep inside `ClausesContext`

- parameters passed to other productions (used only to handle the `Query` object in the first productions)

```
void Statement(Query q) :
{}
{

    CreationStatement(q)
    |
    InsertionStatement(q)
}

void CreationStatement(Query q) :
{}
{
    <KEYWORD_CREATE>
    [
        <KEYWORD_OUTPUT> { StatementContext.isOutput = true; }
    ]
    (
        StreamCreationStatement(q)
        |
        SnapshotCreationStatement(q)
    )
}
```

The only extra parameter passed to productions for the `Query` object

4.2 - Semantic check and tracking system

Deferring the creation of objects in external `Handler` classes enabled us to add additional semantic checks. This extra control is necessary because queries with wrong semantics are accepted by the parser itself, that only performs syntactic controls. As an example. without this type of inspection, we would validate queries that use wrong table names, access fields never created, or that contain similar errors.

The implementation of the semantic checking system required a tracking infrastructure. Without that we wouldn't be able to keep track of what we've already declared and what we can use during queries.

The tracking system works using 2 classes:

- `IdTracker`: keeps the association between an alias and the corresponding object
- `Helper`: provides all the features needed to access the tracking system, like methods to add or retrieve objects

The objects tracked with their aliases inside `IdTracker` are:

- `Datastructure`: mapping between an alias and the associated `Datastructure` object

- `Field`: mapping between a key and an associated `Field` object. The key is defined with the `Datastructure` name containing the `Field`, followed by an underscore and by the `Field` name (dsOwnerName + "_" + fieldName)
- `FromSource`: mapping between an alias and the associated `FromSource` object
- `ArrayList<FromSource>`: mapping between a `Field` name and a list of `FromSource` objects which have a Field with that name

The `Helper` class is used while parsing queries to check if an alias has already been declared, to retrieve already created objects, to control that the fields we are using really belong to the DataStructures declared and to check if every DataStructure has been declared before being used.
In case of errors, exceptions are thrown.

```
/**
 * Costruisce e ritorna la clausola EVERY (High Level) event-based
 * @param samples Numero di campioni
 * @param tokenStream Token dell'oggetto Stream associato
 * @return La clausola HLEveryEventBased creata
 */
public static HLEveryEventBased getHLEveryClause(int samples, Token tokenStream) {

    HLEveryEventBased everyClause = new HLEveryEventBased();
    Stream stream = Helper.getStream(tokenStream);
    everyClause.setSamples(samples);
    everyClause.setStream(stream);
    return everyClause;
}
```

Example of an `Helper` class used to retrieve a `Stream` through its associated token

```
/**
 * Inizializzazione della struttura dati
 * @param table Struttura dati da inizializzare
 */
private static void setDataStructure(DataStructure table) {

    // Imposta il nome ed il flag isOutput prendendoli dal contesto corrente
    table.setName(StatementContext.currentStatementToken.image);
    table.setOutput(StatementContext.isOutput);

    // Aggiunta della DataStructure al sistema di tracking
    Helper.trackDataStructure(table, StatementContext.currentStatementToken);

    // Aggiunta alla DataStructure dei suoi Field parserizzati in precedenza
    DataStructureFieldList fieldList = table.getFields();
    for(int i = 0; i < StatementContext.tokenDataStructureFieldList.size(); i++) {

        PairValue<Token, Field> value = StatementContext.tokenDataStructureFieldList.get(i);
        Field field = value.getSecond();

        // Aggiunta dell'oggetto Field al sistema di tacking
        Helper.trackField(table, field, value.getFirst());

        fieldList.add(field.getName(), field);
    }
}
```

Example of the `Helper` class used to add in the tracking a DataStructure and its `Fields` through the associated tokens

4.3 - Errors

As stated in the introductory part of this document, one of the goals of this work has been the implementation of an error management system. This section will describe the design decisions that led to its current implementation and its underlying working principles.

The characteristics of this error management system are:

- More than one error can be stored in the system at a time (no need to run the parser multiple times to correct different errors)
- A priority is associated to every error (HIGH, MEDIUM or LOW)
- Errors are shown to the user according to their priority

4.3.1 - Error hierarchy

To fully support different types of errors, and to easily customize their associated output, a common parent class has been implemented. This class, called `AbstractError`, just contains attributes used

to store the values shared among all the different kinds of errors, e.g. a general description and the error priority.
`AbstractError`'s implementations must:

- Define attributes needed to store specific information about the error and provide a class constructor to populate them
- Implement the `getCompleteErrorDetails` method specifying the desired output format.

This particular design allows future developers to easily add support for new errors, and to autonomously decide how the collected information should be presented to the end user. Moreover the existing error classes are not affected by a new addition and don't need to be modified.

Currently, only two different `AbstractError`'s implementations have been made: `GeneralError` and `ParsingError`.
The first class is a bare-bone implementation of the superclass, and is used whenever it's impossible to collect many information about the error. The only information stored in it are a general description of the error and the associated priority. The following lines represent an example of the `GeneralError`'s output:

```
[HIGH] Due tipi User Defined sono stati dichiarati con lo stesso alias, Alias:
                                    Point
```

`ParsingError` is a more complex implementation of `AbstractError`, and provides additional attributes needed to store errors generated inside the parser. The output provided by this class is the following:

```
[MEDIUM] Riga: 5, Colonna: 15 – Il campo specificato non e' presente in alcuna
                    DataStructure, Nome campo: Temperature
```

4.3.2 - Error Managing System

The error managing system is composed of the `ErrorPool` class, that performs the following operations:

- Creation of new error instances
- Storage of the errors retrieved during the parsing procedure
- Display of the errors to the user

The `ErrorPool` class is accessed where the errors are generated, in order to be able to collect as many information as possible about the problem that has been found. If the error doesn't allow the system to continue the parsification procedure an exception is thrown.

```
// Errore, sono stati dichiarati piu' tipi built in con lo stesso alias
ParserLogger.getInstance().printMessage(Type.Error, Verbosity.Low, "DefinedTypes.loadUserDefinedTypes()",
        "Un tipo User Defined e' stato dichiarato con lo stesso nome di un tipo Built-in", "Alias: " + alias);
ErrorPool.getInstance().addError("Un tipo User Defined e' stato dichiarato con lo stesso nome di un tipo Built-in",
        "Alias: " + alias, ErrorPriority.HIGH);
```

Insertion of an error in the ErrorPool

This class is also accessed when the parsing is completed and the system reports all the collected errors.

```
/**
 * Visualizza a video gli errori memorizzati
 */
public void printOnConsole() {

    // Ordina i vari errori per priorita'
    Collections.sort(errors);

    // Controllo presenza di errori ed eventuale printout
    if (errors.size() == 0) {

        System.out.println("Nessun errore trovato.");
    } else {

        // Visualizza il contenuto di ogni errore
        for (AbstractError error : errors) {

            System.out.println(error.getCompleteErrorDetails() + "\n");
        }
    }
}
```

ErrorPool.printOnConsole(): Every single error class is responsible about the printout

4.3.3 - Detected errors and examples

The following list presents some of the errors currently checked by the parser.

- INSERT INTO fields must correspond to the DataStructure's fields
- The DataStructure used in the INSERT INTO must exist
- It's impossible to have different CREATE referring to the same DataStructure
- Different DataStructures declared with same alias are not allowed
- Lack of field qualificators when needed

- Fields belonging to DataStructures not declared in the FROM clause
- Multiple constants defined with same name
- Sign modifications made to constants that are not integer or float
- Multiple User Defined Types with same alias
- Multiple User Defined Types with same class
- User Defined Types with alias already used by Built-In types

Example 1

```
/* DATA STRUCTURES */
CREATE OUTPUT STREAM Table (readerID ID, ts TIMESTAMP);

 /* LL STATEMENTS   */
INSERT INTO STREAM NotDeclaredTable (readerID, ts) LOW:
      EVERY 1 SAMPLES
      SELECT ALL lastReader, GROUP_TS
      SAMPLING
            ON EVENT lastReaderChanged
            WHERE TRUE
      EXECUTE IF (ID = "tag")
            REFRESH NEVER
```

Here an error is produced because the INSERT INTO clause uses *NotDeclaredTable* which has never been declared.

Example 2

```
/* DATA STRUCTURES */
CREATE OUTPUT STREAM Table (readerID ID, ts TIMESTAMP);

 /* LL STATEMENTS   */
INSERT INTO STREAM Table (readerID, ts, newField1, newField2)
LOW:
      EVERY 1 SAMPLES
      SELECT ALL lastReader, GROUP_TS
      SAMPLING
            ON EVENT lastReaderChanged
```

```
        WHERE TRUE
    EXECUTE IF (ID = "tag")
        REFRESH NEVER
```

The error in this query regards an insertion query, which tries to put elements in a DataStructure's field that doesn't exist

Example 3

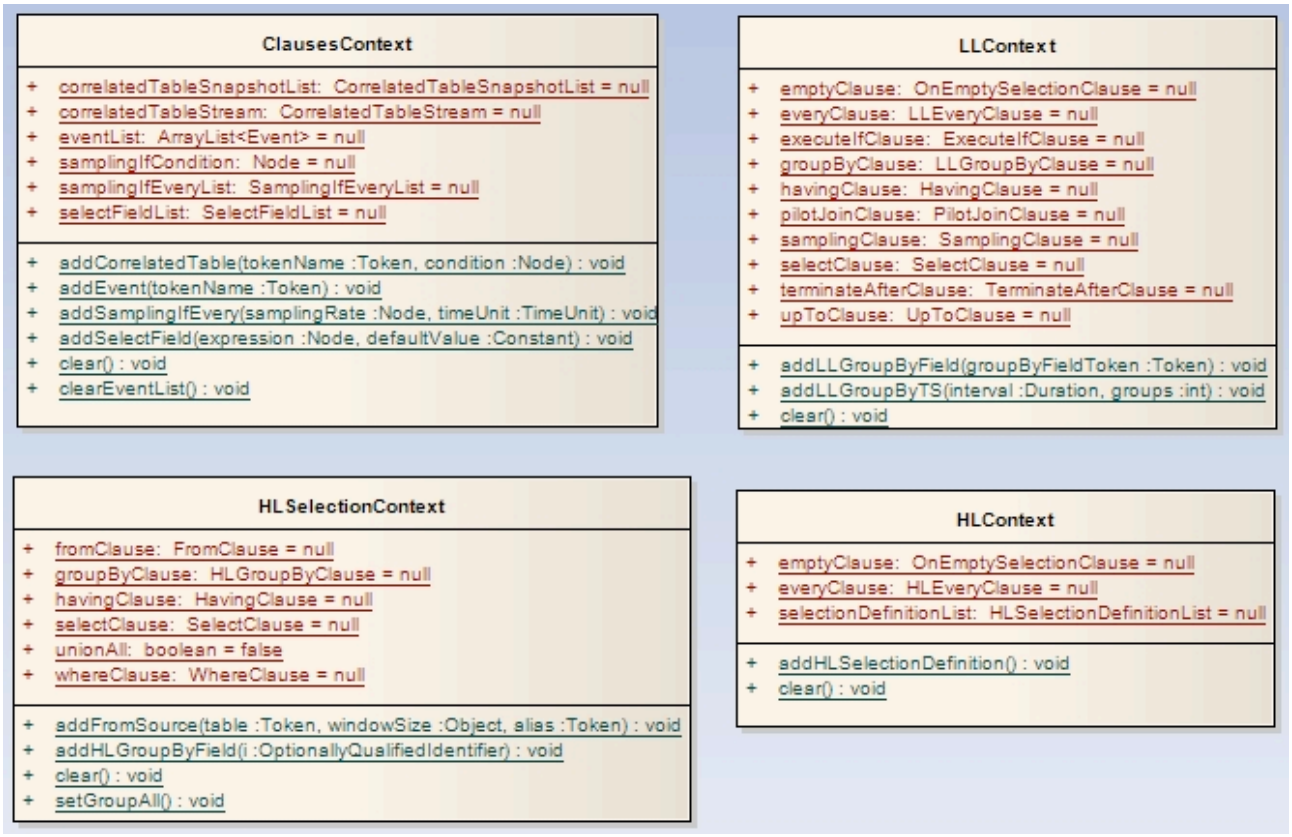
```
<UserDefinedConstants>
    <UserDefinedConstant enabled="true">
        <package>org.dei.expressions.userDefined</package>
        <class>Point</class>
        <alias>Point</alias>
    </UserDefinedConstant>
    <UserDefinedConstant enabled="true">
        <package>org.dei.expressions.userDefined</package>
        <class>Point</class>
        <alias>Dot</alias>
    </UserDefinedConstant>
</UserDefinedConstants>
```

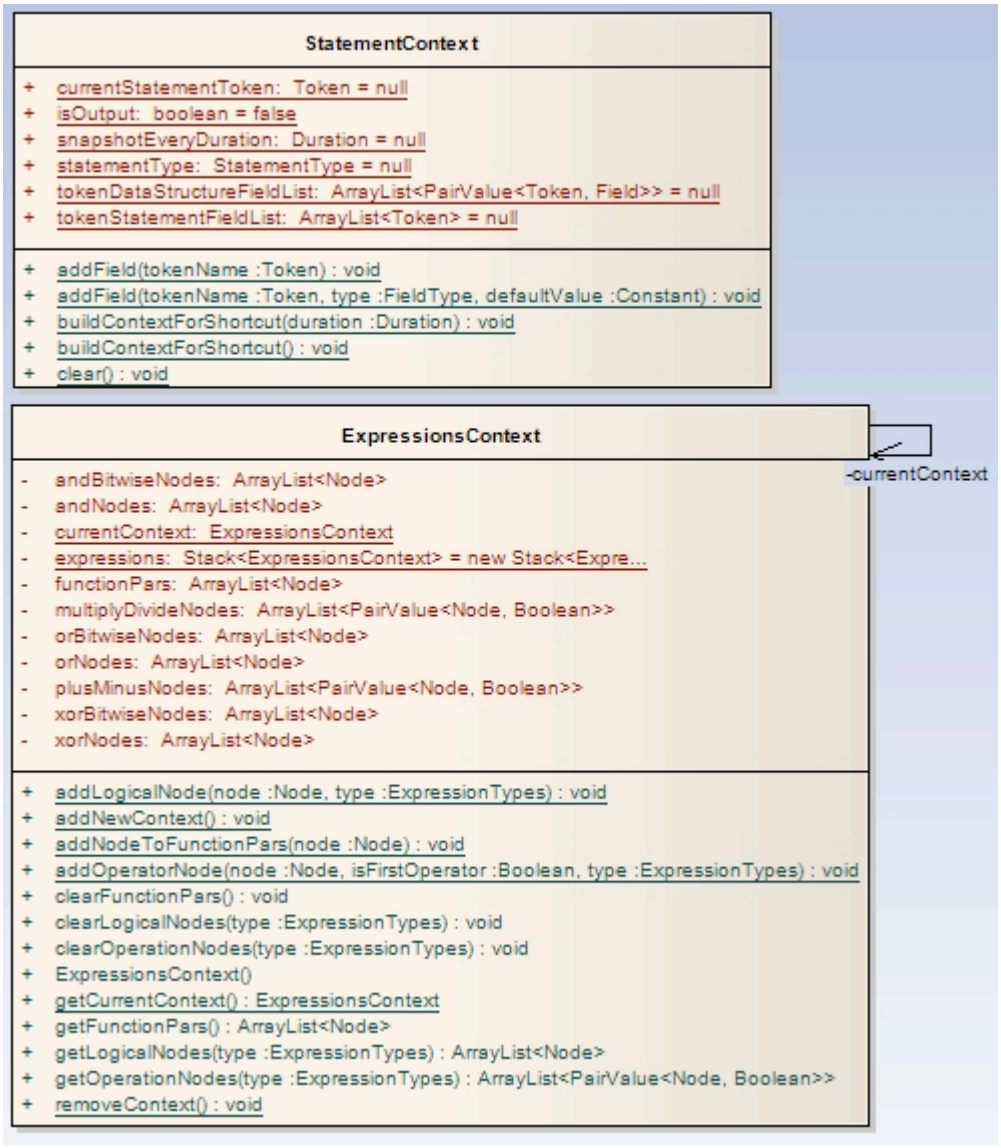
This last error is relative to the declaration of a UserDefinedType. The XML descriptor shown above contains 2 different types, that are declared with the same class.

4.4 - Class Diagrams

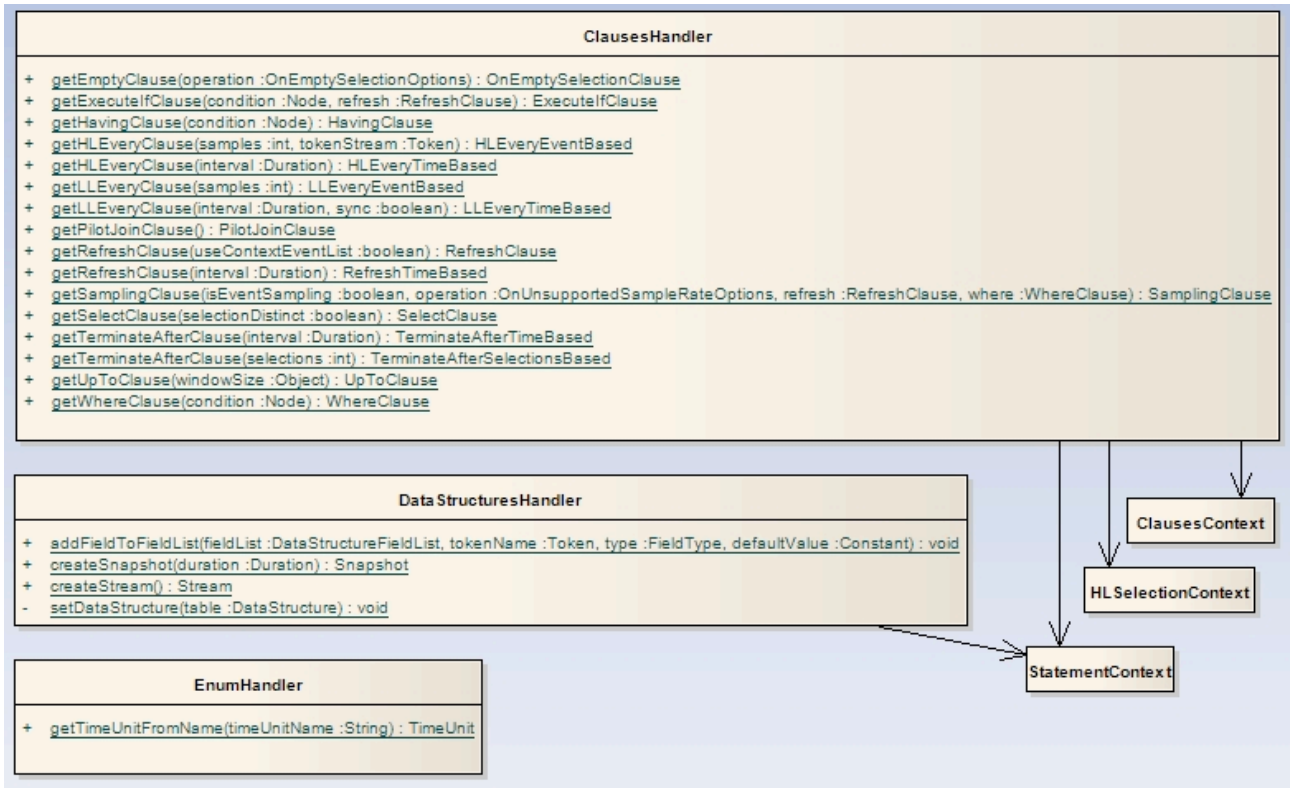
To give an overview of the classes and to better understand their content, we show the class diagrams of the project.

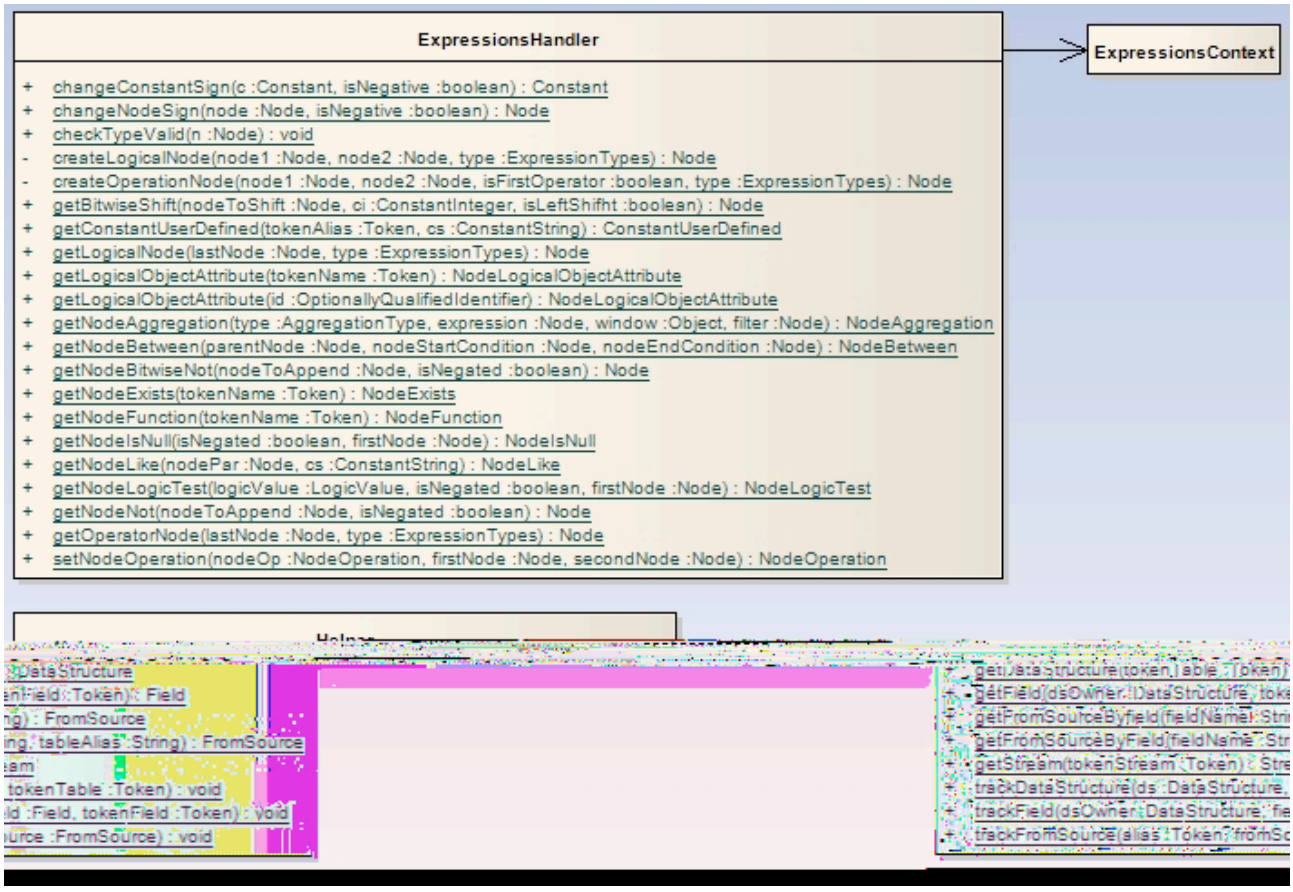
4.4.1 - Context classes

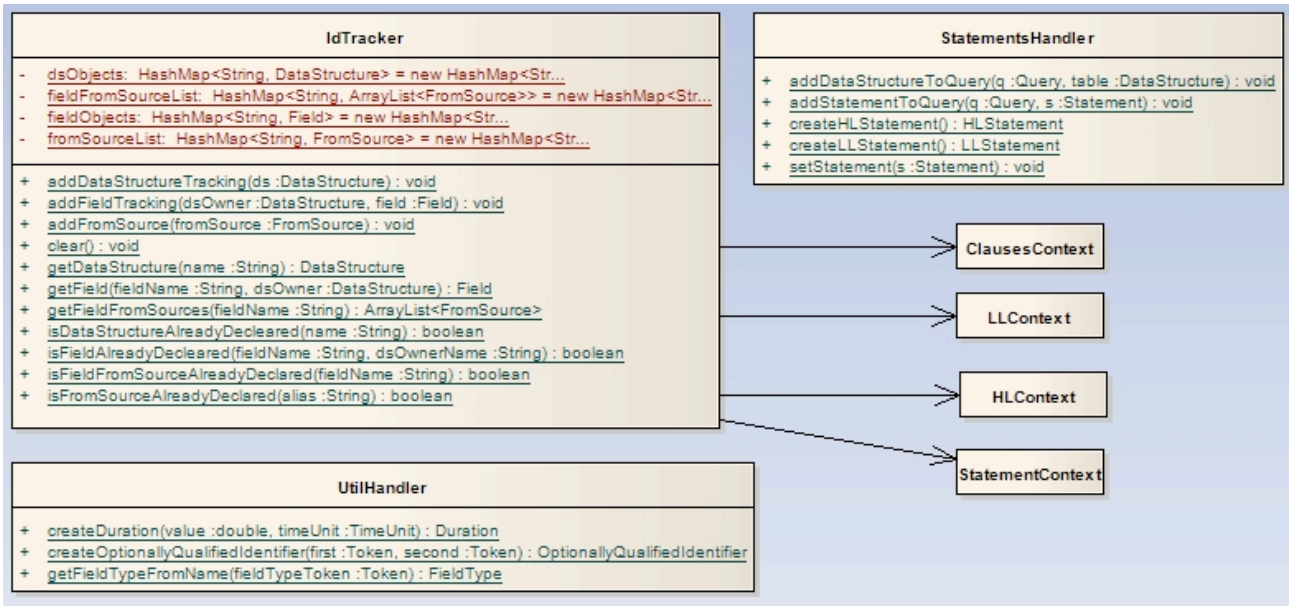



StatementContext
+ currentStatementToken: Token = null
+ isOutput: boolean = false
+ snapshotEveryDuration: Duration = null
+ statementType: StatementType = null
+ tokenDataStructureFieldList: ArrayList<PairValue<Token, Field>> = null
+ tokenStatementFieldList: ArrayList<Token> = null
+ addField(tokenName :Token) : void
+ addField(tokenName :Token, type :FieldType, defaultValue :Constant) : void
+ buildContextForShortcut(duration :Duration) : void
+ buildContextForShortcut() : void
+ clear() : void
ExpressionsContext
- andBitwiseNodes: ArrayList<Node>
- andNodes: ArrayList<Node>
- currentContext: ExpressionsContext
- expressions: Stack<ExpressionsContext> = new Stack<Expre...
- functionPars: ArrayList<Node>
- multiplyDivideNodes: ArrayList<PairValue<Node, Boolean>>
- orBitwiseNodes: ArrayList<Node>
- orNodes: ArrayList<Node>
- plusMinusNodes: ArrayList<PairValue<Node, Boolean>>
- xorBitwiseNodes: ArrayList<Node>
- xorNodes: ArrayList<Node>
+ addLogicalNode(node :Node, type :ExpressionTypes) : void
+ addNewContext() : void
+ addNodeToFunctionPars(node :Node) : void
+ addOperatorNode(node :Node, isFirstOperator :Boolean, type :ExpressionTypes) : void
+ clearFunctionPars() : void
+ clearLogicalNodes(type :ExpressionTypes) : void
+ clearOperationNodes(type :ExpressionTypes) : void
+ ExpressionsContext()
+ getCurrentContext() : ExpressionsContext
+ getFunctionPars() : ArrayList<Node>
+ getLogicalNodes(type :ExpressionTypes) : ArrayList<Node>
+ getOperationNodes(type :ExpressionTypes) : ArrayList<PairValue<Node, Boolean>>
+ removeContext() : void
-currentContext

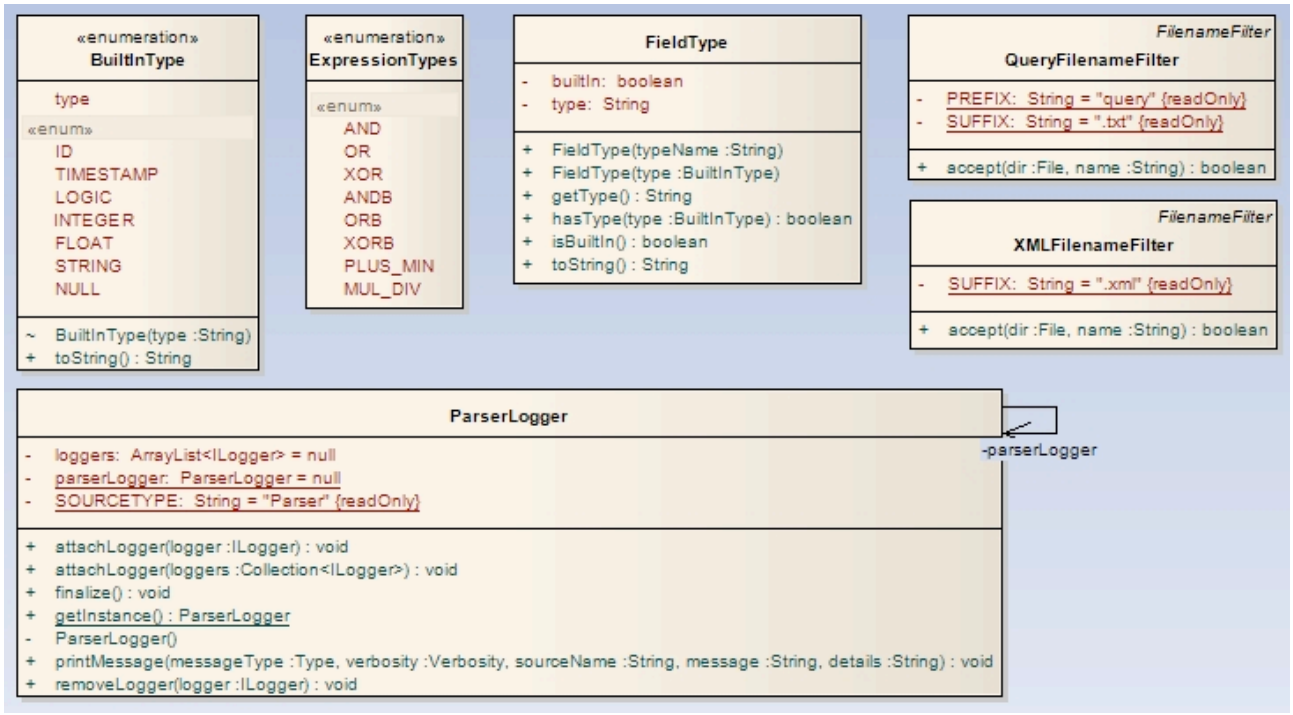
4.4.2 - Handler classes

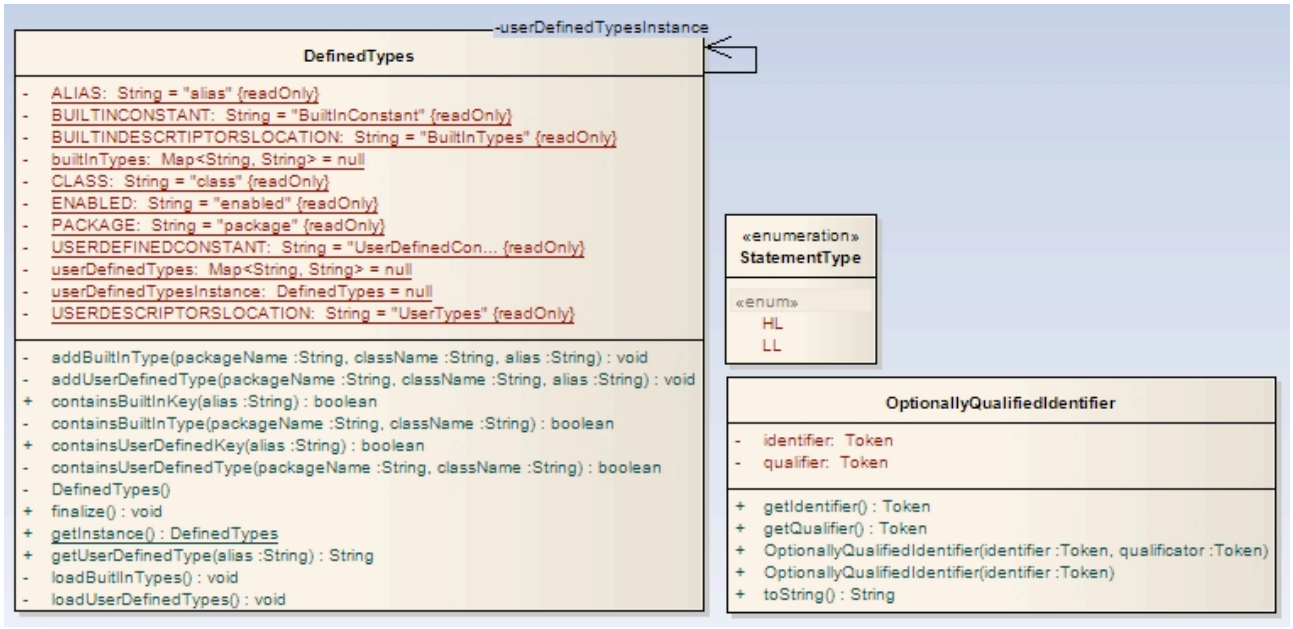




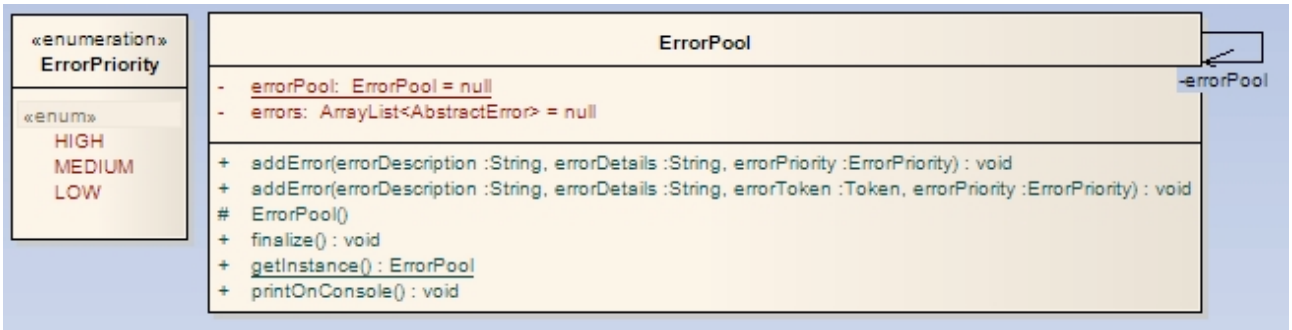

IdTracker
- dsObjects: HashMap<String, DataStructure> = new HashMap<Str...
- fieldFromSourceList: HashMap<String, ArrayList<FromSource>> = new HashMap<Str...
- fieldObjects: HashMap<String, Field> = new HashMap<Str...
- fromSourceList: HashMap<String, FromSource> = new HashMap<Str...
+ addDataStructureTracking(ds :DataStructure) : void
+ addFieldTracking(dsOwner :DataStructure, field :Field) : void
+ addFromSource(fromSource :FromSource) : void
+ clear() : void
+ getDataStructure(name :String) : DataStructure
+ getField(fieldName :String, dsOwner :DataStructure) : Field
+ getFieldFromSources(fieldName :String) : ArrayList<FromSource>
+ isDataStructureAlreadyDecleared(name :String) : boolean
+ isFieldAlreadyDecleared(fieldName :String, dsOwnerName :String) : boolean
+ isFieldFromSourceAlreadyDeclared(fieldName :String) : boolean
+ isFromSourceAlreadyDeclared(alias :String) : boolean
StatementsHandler
+ addDataStructureToQuery(q :Query, table :DataStructure) : void
+ addStatementToQuery(q :Query, s :Statement) : void
+ createHLStatement() : HLStatement
+ createLLStatement() : LLStatement
+ setStatement(s :Statement) : void
ClausesContext
LLContext
HLContext
StatementContext
UtilHandler
+ createDuration(value :double, timeUnit :TimeUnit) : Duration
+ createOptionallyQualifiedIdentifier(first :Token, second :Token) : OptionallyQualifiedIdentifier
+ getFieldTypeFromName(fieldTypeToken :Token) : FieldType

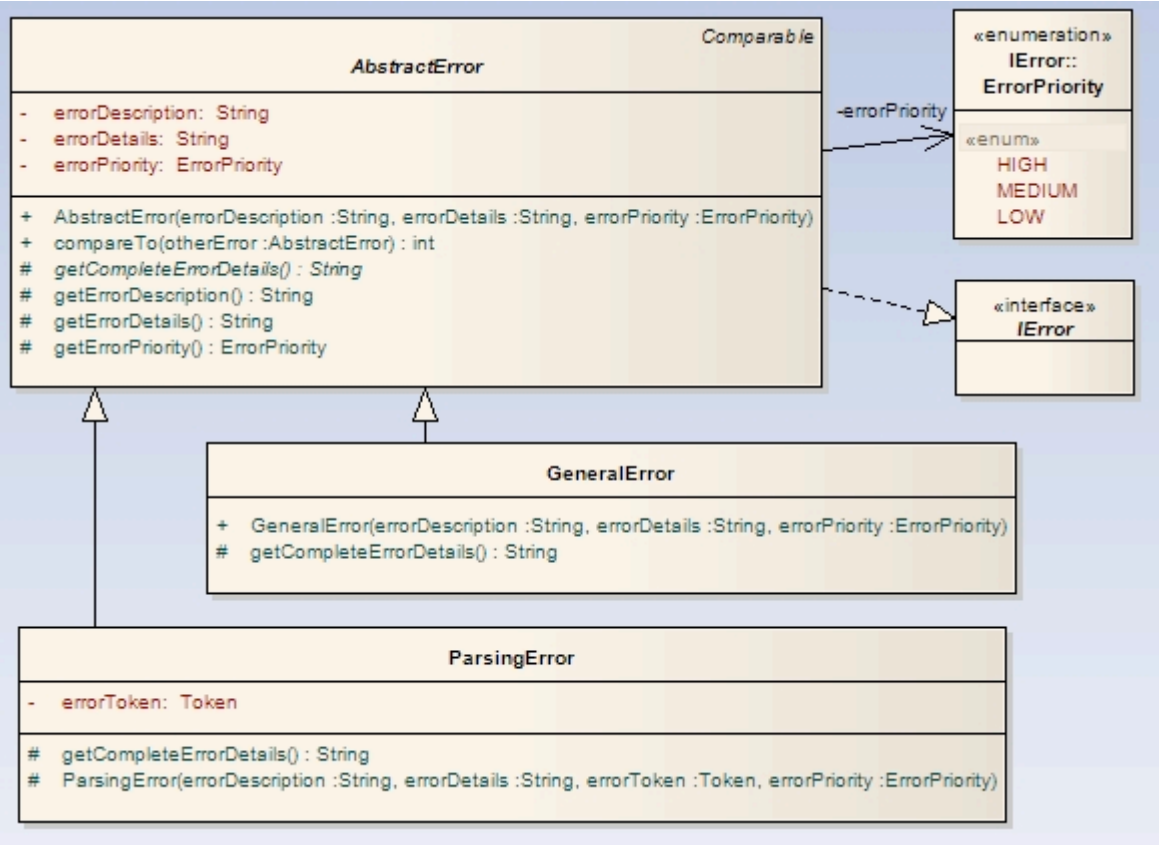
4.4.3 - Util classes





4.4.4 - Error classes





4.4.5 - Parser


class parser
Parser
+ definedTypes: DefinedTypes
+ TYPE_VALIDATION: boolean = false {readOnly}
+ Aggregate(parExpressionType :ExpressionType) : NodeAggregation
+ AggregationOperator() : AggregationType
+ ComparisonOperator() : NodeComparisonOperation
+ Constant() : Constant
+ ConstantBoolean() : ConstantLogic
+ ConstantFloat() : ConstantFloat
+ ConstantInteger() : ConstantInteger
+ ConstantNull() : ConstantNull
+ ConstantString() : ConstantString
+ ConstantUserDefined() : ConstantUserDefined
+ CorrelatedTable() : void
+ CorrelatedTableList() : void
+ CreateSnapshotClause() : Snapshot
+ CreateStreamClause() : Stream
+ CreationStatement(q :Query) : void
+ Duration() : Duration
+ EventList() : void
+ ExecuteIfClause() : ExecuteIfClause
+ ExistsAttribute() : NodeExists
+ Expression(parExpressionType :ExpressionType) : Node
+ ExpressionBit(parExpressionType :ExpressionType) : Node
+ ExpressionBitFactor(parExpressionType :ExpressionType) : Node
+ ExpressionBitShift(parExpressionType :ExpressionType) : Node
+ ExpressionBitTerm(parExpressionType :ExpressionType) : Node
+ ExpressionBitTest(parExpressionType :ExpressionType) : Node
+ ExpressionBooleanFactor(parExpressionType :ExpressionType) : Node
+ ExpressionBooleanPrimary(parExpressionType :ExpressionType) : Node
+ ExpressionBooleanTerm(parExpressionType :ExpressionType) : Node
+ ExpressionBooleanTest(parExpressionType :ExpressionType) : Node
+ ExpressionValue(parExpressionType :ExpressionType) : Node
+ ExpressionValueFactor(parExpressionType :ExpressionType) : Node
+ ExpressionValuePrimary(parExpressionType :ExpressionType) : Node
+ ExpressionValueTerm(parExpressionType :ExpressionType) : Node
+ FieldDefinition() : void
+ FieldDefinitionList() : void
+ FieldGroupingBy() : void
+ FieldGroupingByList() : void
+ FieldGroupingByTs() : void
+ FieldList() : void
+ FieldSelection(parExpressionType :ExpressionType) : void
+ FieldSelectionList(parExpressionType :ExpressionType) : void
+ FieldType() : FieldType
+ FromClause() : void
+ FunctionCall(parExpressionType :ExpressionType) : NodeFunction
+ GroupByClause() : void
+ GroupClause() : void
+ HavingClause(parExpressionType :ExpressionType) : HavingClause
+ HighEveryClause() : HLEveryClause
+ HighSelectionStatement(parExpressionType :ExpressionType) : void
+ HighSelectionStatementDefinition(parExpressionType :ExpressionType) : void
+ Identifier() : Token
+ InsertionStatement(q :Query) : void
+ InsertSnapshotClause() : void
+ InsertStreamClause() : void
+ LogicTest(firstNode :Node) : NodeLogicTest
+ LogicValue() : LogicValue
+ LowEveryClause() : LLEveryClause
+ LowSelectionStatement(parExpressionType :ExpressionType) : void
+ main(args :String[]) : void
+ OnEmptySelectionClause() : OnEmptySelectionClause
+ OnEventClause() : void
+ OnUnsupportedSRClause() : OnUnsupportedSampleRateOptions
+ OptionallyQualifiedIdentifier() : OptionallyQualifiedIdentifier
+ PilotJoinClause() : PilotJoinClause
+ QualifiedIdentifier() : void
+ RefreshClause() : RefreshClause
+ SamplesNumber() : int
+ SamplingClause() : SamplingClause
+ SamplingEveryClause() : void
+ SamplingIfClause() : void
+ SamplingIfEveryClause() : void
+ SelectClause(parExpressionType :ExpressionType) : SelectClause
+ SelectionsNumber() : int
+ Sign() : boolean
+ SignedConstant() : Constant
+ SnapshotCreationStatement(q :Query) : void
+ SnapshotInsertionStatement(q :Query) : void
+ SnapshotSelectionStatement() : Statement
+ Statement(q :Query) : void
+ StatementSequence() : Query
+ StreamCreationStatement(q :Query) : void
+ StreamInsertionStatement(q :Query) : void
+ StreamSelectionStatement() : Statement
+ TerminateAfterClause() : TerminateAfterClause
+ TimeUnit() : TimeUnit
+ UpToClause() : UpToClause
+ WhereClause(parExpressionType :ExpressionType) : WhereClause
+ WindowDefinition() : void
+ WindowDefinitionList() : void
+ WindowFieldList() : void
+ WindowSize() : Object
«enumeration»
Parser::ExpressionType
«enum»
HIGH_LEVEL_ALL
LOW_LEVEL_ALL
LOW_LEVEL_NO_AGGR
LOW_LEVEL_NO_AGGR_NO_PILOT
org.dei.perla.parser.grammar.context
org.dei.perla.parser.grammar.handlers
org.dei.perla.parser.grammar.errors

5 - USER DEFINED TYPES MANAGEMENT

The work related to the PERLA Parser also included the implementation of a sub-system that enables the user to define and use new data types (constants).
Every User Defined Type is managed using a class (the implementation of the type) and a simple XML descriptor (needed to notify the parser about the presence of a new constant type).
The following information are stored in the descriptor:

- *package*: package which contains the Type's class
- *class*: name of the class that implements the Type
- *alias*: name that identifies the Type inside the queries

The XML descriptors are loaded before the system begins the parsing procedure. The classes corresponding to the UserDefinedTypes used in a query are loaded at runtime using Java Reflection.

UserDefinedTypes XML descriptor's DTD**:**

```
<!ELEMENT UserDefinedConstants ( UserDefinedConstant+ ) >

<!ELEMENT UserDefinedConstant ( package, class, alias ) >
<!ATTLIST UserDefinedConstant enabled (true|false) "true" >

<!ELEMENT alias ( #PCDATA ) >

<!ELEMENT class ( #PCDATA ) >

<!ELEMENT package ( #PCDATA ) >
```

```
<UserDefinedConstants>
    <UserDefinedConstant enabled="true">
        <package>org.dei.expressions</package>
        <class>Point</class>
        <alias>Point</alias>
    </UserDefinedConstant>
    <UserDefinedConstant enabled="true">
        <package>org.dei.expressions</package>
        <class>ConstantVector</class>
        <alias>Vector</alias>
    </UserDefinedConstant>
</UserDefinedConstants>
```

XML descriptor example

Due to the need to keep track of the built-in types in a more dynamic way, and to be able to check if a user named a new type with an alias that has already been assigned to a built-in type, this system has been extended to load PERLA original types from XML descriptors as well.

BuiltInTypes XML descriptor's DTD**:**

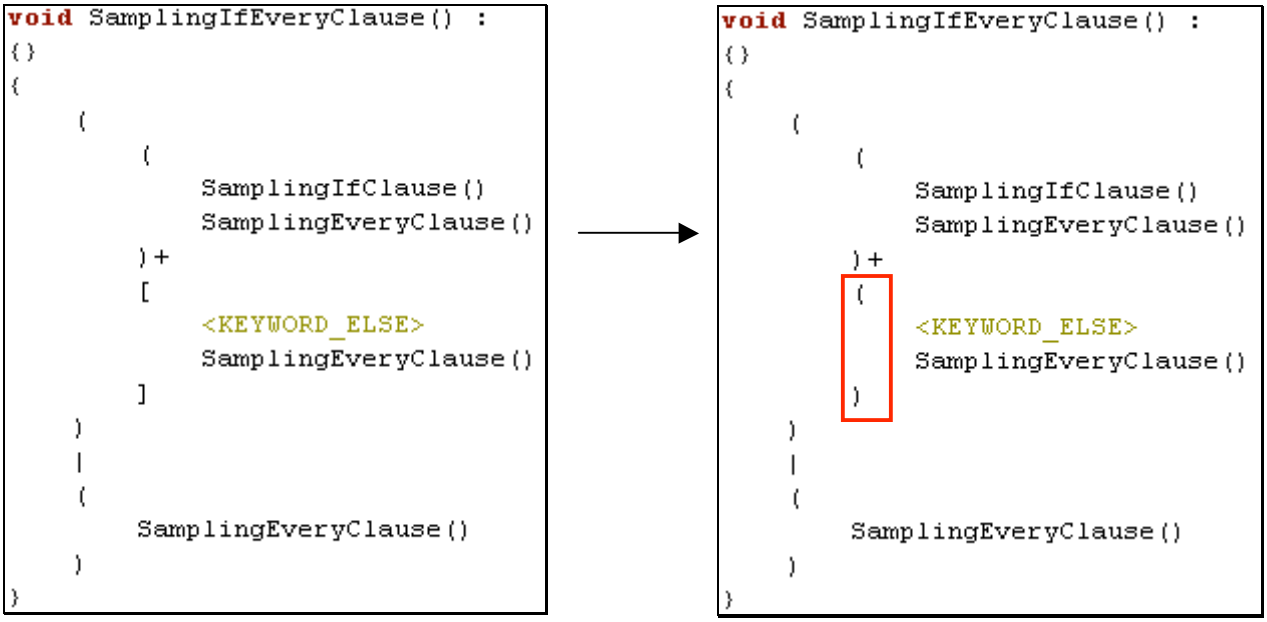
```
<!ELEMENT BuiltInConstants ( BuiltInConstant+ ) >

<!ELEMENT BuiltInConstant ( package, class, alias ) >
<!ATTLIST BuiltInConstant enabled (true|false) "true" >

<!ELEMENT alias ( #PCDATA ) >

<!ELEMENT class ( #PCDATA ) >

<!ELEMENT package ( #PCDATA ) >
```

The current implementation of the Type management system still considers user defined and built-in types as separated entities (built in types are entirely recognized using grammar productions, while user defined types are parsed using additional classes and loaded at runtime). It's however possible to consolidate these two behaviors, and use the system described in this chapter to load built-in types at runtime too.

6 - GRAMMAR CHANGES

During the development of the project, some grammar changes has been made to fix some error and to add a new functionality.

The first corrected error was caused by an optional `else` path in the `SamplingIfEveryClause` production when at least an `if` path was present. The correct behavior corresponds to the second image, since the `else` path is mandatory in that situation.



Mandatory *else* path fixed

The second error was caused by the inversion of the `HavingClause` and the `UpToClause` calls in the `LowSelectionStatement`. The productions have been changed to respect the original order of the grammar.

```
void LowSelectionStatement(ExpressionType parExpressionType) :
()
{
    LLContext.selectClause = SelectClause(parExpressionType)
    [
        GroupByClause()
    ]
    [
        LLContext.upToClause = UpToClause()
    ]
    [
        LLContext.havingClause = HavingClause(parExpressionType)
    ]
```

```
void LowSelectionStatement(ExpressionType parExpressionType) :
()
{
    LLContext.selectClause = SelectClause(parExpressionType)
    [
        GroupByClause()
    ]
    [
        LLContext.havingClause = HavingClause(parExpressionType)
    ]
    [
        LLContext.upToClause = UpToClause()
    ]
```

Change to the *LowSelectionStatement* production

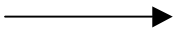
The last grammar change has been made to add a new language functionality: User Defined Types. To have this working, we had to add a new production, called `ConstantUserDefined`, which is identified by the token `NEW` followed by 2 parameters enclosed in parentheses. The first parameter is the type identifier, while the second one is a string used for initialization purposes.

The `Constant` production has been changed as well.

```
ConstantUserDefined ConstantUserDefined() :
{ Token t; ConstantString cs; }
{
    <NEW>
    "("
        t = <IDENTIFIER>
        ","
        cs = ConstantString()
    ")"

    { return ExpressionsHandler.getConstantUserDefined(t, cs); }
}
```

The new `ConstantUserDefined` production

```
Constant Constant() :
{ Constant c; }
{
    (
        c = ConstantNull()
        |
        c = ConstantBoolean()
        |
        c = ConstantString()
        |
        LOOKAHEAD(1)
        c = ConstantInteger()
        |
        c = ConstantFloat()
    )
    { return c; }
}
```



```
Constant Constant() :
{ Constant c; }
{
    (
        c = ConstantNull()
        |
        c = ConstantBoolean()
        |
        c = ConstantString()
        |
        LOOKAHEAD(1)
        c = ConstantInteger()
        |
        c = ConstantFloat()
        |
        c = ConstantUserDefined()
    )
    { return c; }
}
```

Change to the `Constant` production

7 - QUERY CLASSES CHANGES

While developing this project we faced some discrepancy between the grammar definition and the query classes implementation. In order to have a coherent implementation, we had to correct them.
NOTE: changes made to the expression classes are not listed in this chapter, since their development was assigned to another person.

The first significant change concerned the lack of union's *ALL* flag.
A High Level query (`HLSelectionDefinition`) is composed by different High Level selection statements (`HighSelectionStatementDefinition`) joined together using the UNION clause. However, the UNION alone only returns distinct values. When also duplicates are important, the user also need to specify the ALL particle.
In order to implement this behavior, a new boolean variable has been added to the `HighSelectionStatementDefinition` object. This flag is set whenever the ALL particle is found by the parser.
It's important to notice that, if there are *n* different High Level Selection Statements in a single High Level selection Query, only *n-1* UNIONs are needed to merge the results. Therefore the decision to ignore the ALL flag associated to the first `HighSelectionStatementDefinition`.

```
HighSelectionStatementDefinition(parExpressionType)
(
    <KEYWORD_UNION>
    [
        <KEYWORD_ALL> { HLSelectionContext.unionAll = true; }
    ]
    HighSelectionStatementDefinition(parExpressionType)
)*
```

Grammar part where *UNION [ALL]* flag can be specified

The second change regarded the attribute `value` inside `Duration` class. It was defined with type integer, but the grammar specifies a double precision floating point value.

The third modification concerned `FieldType` enumeration, that had to be changed to add extra checks for the new UserDefinedTypes system.
The original implementation used a simple enumeration to distinguish among different types, whereas the new `FieldType` has been provided with a boolean flag, which indicates if the instance

refers to a built-in type or to a user defined one, and a String field which stores the name of the UserDefinedType's alias. An additional constructor, specifically written to handle user defined types, checks if the alias given as parameter really corresponds to a type loaded during the parser startup. If not, an exception is thrown.

8 - RESULTS

Every component of the Query Object everrides the `toString()` method in order to allow correctness checking (that is our main goal).
It's important to notice that the text returned by this `toString()` doesn't perfectly correspond with the original query, but it represents an equivalent form. The most important differences are related to the presence of abbreviated forms.
The following examples will show some of the differences that can be found comparing the original query and the equivalent `toString()` representation.

Example 1

Input query:

```
CREATE OUTPUT STREAM Table (readerID ID, ts TIMESTAMP) AS
LOW:
      EVERY ONE
      SELECT lastReader, GROUP_TS
      SAMPLING ON EVENT lastReaderChanged
      EXECUTE IF ID = "tag"
```

`query.toString()` printout:

```
 /* DATA STRUCTURES */

CREATE OUTPUT STREAM Table (readerID ID, ts TIMESTAMP)
;

 /* LL STATEMENTS   */

INSERT INTO STREAM Table (readerID, ts)
LOW:
      EVERY 1 SAMPLES
      SELECT ALL lastReader, GROUP_TS
      HAVING TRUE
      UP TO ONE
      ON EMPTY SELECTION INSERT NOTHING
      SAMPLING
            ON EVENT lastReaderChanged
            WHERE TRUE
      EXECUTE IF (ID = "tag")
            REFRESH NEVER
```

The first noticeable difference between the two representations is the absence of the token AS in the equivalent form, that leads to the decomposition of the query in two parts (declarative statements and insertion statements are now clearly separated).

Moreover, it's possible to see that the equivalent `toString()` explicitly contains all the clauses that have been omitted in the original form, such as ON EMPTY INSERT NOTHING, UP TO ONE, etc.

Example 2

Input query:

```
CREATE STREAM TanksPositions (gpsID ID, baseStationID ID, distanceFromP FLOAT)
AS
LOW:
      EVERY ONE
      SELECT ID, baseStationID, dist(P, location)
      SAMPLING EVERY 1 h
      EXECUTE IF deviceType = "GPS"
;

CREATE SNAPSHOT NearestTank (gpsID ID, baseStationID ID)
WITH DURATION 1 h AS
HIGH:
      SELECT TanksPositions.gpsID, TanksPositions.baseStationID
      FROM TanksPositions (1 h)
      WHERE TanksPositions.distanceFromP = MIN(TanksPositions.distanceFromP)
;

CREATE OUTPUT STREAM Temperatures (sensorID ID, temp FLOAT) AS
LOW:
      EVERY ONE
      SELECT ID, temp
      SAMPLING EVERY 1 m
      PILOT JOIN NearestTank ON NearestTank.baseStationID = baseStation
```

`query.toString()` printout:

```
/* DATA STRUCTURES */

CREATE STREAM TanksPositions (gpsID ID, baseStationID ID, distanceFromP FLOAT)
;

CREATE SNAPSHOT NearestTank (gpsID ID, baseStationID ID) WITH DURATION 1.0 h
;

CREATE OUTPUT STREAM Temperatures (sensorID ID, temp FLOAT)
;


 /* LL STATEMENTS   */

INSERT INTO STREAM TanksPositions (gpsID, baseStationID, distanceFromP)
LOW:
      EVERY 1 SAMPLES
      SELECT ALL ID, baseStationID, dist(P, location)
      HAVING TRUE
      UP TO ONE
```

```
      ON EMPTY SELECTION INSERT NOTHING
      SAMPLING
            EVERY 1 h
            ON UNSUPPORTED SAMPLE RATE DO NOT SAMPLE
            REFRESH NEVER
            WHERE TRUE
      EXECUTE IF (deviceType = "GPS")
            REFRESH NEVER
;

INSERT INTO STREAM Temperatures (sensorID, temp)
LOW:
      EVERY 1 SAMPLES
      SELECT ALL ID, temp
      HAVING TRUE
      UP TO ONE
      ON EMPTY SELECTION INSERT NOTHING
      SAMPLING
            EVERY 1 m
            ON UNSUPPORTED SAMPLE RATE DO NOT SAMPLE
            REFRESH NEVER
            WHERE TRUE
      PILOT JOIN NearestTank ON (NearestTank.baseStationID = baseStation)
      EXECUTE IF TRUE
            REFRESH NEVER
;


 /* HL STATEMENTS   */

INSERT INTO SNAPSHOT NearestTank (gpsID, baseStationID)
HIGH:
      SELECT ALL TanksPositions.gpsID, TanksPositions.baseStationID
      FROM TanksPositions(1.0 h) AS TanksPositions
      WHERE (TanksPositions.distanceFromP = MIN(TanksPositions.distanceFromP))
      ON EMPTY SELECTION INSERT NOTHING
```

9 - CONCLUSIONS

Among all the possible implementations devised to achieve the project goals, we believe we've chosen the most flexible one. Implementation of new grammar constructs and changes in the existing ones can be easily done with the current class structure. Moreover, the small amount of Java code present in the JavaCC file makes all the modifications to the grammar easy to implement. The modular approach simplifies the extension of the parser and enables the programmers to add new functionalities without having to modify big parts of the existing code (thus avoiding errors). A classic approach to perform an addition to the grammar should be:

- change the JavaCC file to support the new constructs
- select, among all the object creation styles listed in this document, the most convenient for the work to be done
- implement, if necessary, the new classes which will be used to store the new query elements
- implement new Handler and Context classes as needed or extend the existing ones
- complete the JavaCC file with the code needed to interface with the newly created classes/methods

Currently, the insertion of a new statement to set logical object parameters (simple actuation queries) has already been planned and we verified that the aforementioned checklist can be successfully used. The new grammar construct will have the following form:
SET fieldName_1 = value_1, fieldName_2 = value2, ... EXECUTE IF ...

Bibliography

1] F.A. Schreiber, R. Camplani, M. Fortunato, M. Marelli: “Design of a Declarative Data Language for Pervasive Systems” - Art Deco R. A. 11.1b, January 2008.

2] F. A. Schreiber, R. Camplani, M. Fortunato, M. Marelli and F. Pacifici: “PERLA: a Data Language for Pervasive Systems”, in Proceedings of Sixth Annual IEEE International Conference on Pervasive Computing and Communications (PerCom 2008). Honk Kong, pp. 282-287, 2008

3] A. Perrucci, “Definizione di una struttura ad oggetti per la rappresentazione interna delle interrogazioni del linguaggio Perla”, Thesis, Politecnico di Milano, pp. 1 - 55

4] JavaCC reference
Official site: *https://javacc.dev.java.net/*

5] JavaCC Eclipse Plugin
Official site: *http://eclipse-javacc.sourceforge.net/*