



# Support for DSL in Eclipse

## Master Thesis

Submitted by:

LIU Yao

liuyao1981918@hotmail.com

IMT

Matriculation Number:

27497

Supervised by:

Prof. Dr. Ralf Möller

M.Sc. Miguel GARCIA

Hamburg, Germany

2006-09-18

# 1. Introduction

Domain Specific Language (DSL) is a kind of modeling language used by domain modeling experts. And it is used to describe the problem domain and consists of the domain terms and rules. The emergence of DSL is due to improving the productivity in software development.

Developer productivity has been improved by high-level programming language, such as Java and C++, modeling languages like UML, and even software engineering ways like UML based Model-Driven Architecture (MDA), which means UML is used to describe all the domain models. However, this seemed still not satisfying enough, and people were after for some new ways to improve the productivity and hoped raise the abstraction level of models. The DSL and Domain Specific Modeling (DSM) came up.

The idea is to use a domain specific language (DSL) that fits the problem domain and generate realization with specialized code generators. In a nutshell, Domain Specific Modeling means that you as a developer will implement and use your own code generators to avoid repetitive manual coding and therefore increase efficiency. It raises the level of abstraction and automates a lot of otherwise hand written code. This is also a kind of MDA not only with UML/OCL but also with DSL.

Since a DSL has its own constructs and rules, if modeling with DSL in just a text pad, it is labor consuming and error-prone. An efficient approach is to be in control of the DSL and the code generation tools, which could be free tools in Eclipse. The tools should be adopted to fill the needs of your system. As developer you will define the DSL and implement the code generators yourself. Further on, it will make you feel comfortable with the result, since it is under your control. This is a different approach from many of the general purpose MDA tools that are popping up in this area. Vendor specific code generation tools often look good at the first demo, but later on turns out to be inadequate and hard to evolve as the requirements changes.

Therefore, giving some support to DSL as tools in Eclipse is desirable. And this kind of tool for DSL should have the functionalities such as editing no matter text based or tree based, syntax checking, well-formedness checking, further transforming, and maybe graphical generating and editing, etc. All of the above possible functionalities need to be implemented on the base of modeling of the DSL itself, i.e. metamodeling. We can say that the metamodel is a very important task in providing support to a DSL. In this report, we will take EJB3QL as a case study to see how important a well-formed metamodel for DSL is, how to build a well-formed metamodel, and how to further make use of the metamodel.

# 2. EJB3 Persistence and EJB3QL

## 2.1 Introduction of EJB3 Persistence

From May 6th, 2004 Linda DeMichiel at the first time announced the large changes of EJB3.0 compared with the previous versions, to 2006 the final release of EJB3.0 specification was issued, the dispute and discussion on EJB3.0 never stopped. Not compared with other frameworks such as Spring, JDO, and so on, but just with EJB2.1, indeed EJB3.0 has been largely improved.

Firstly, the portability and usability are improved. In the time of EJB2.1, EJBs are heavy weight to the developers, because one has to follow several steps to build an EJB, for example, creating a bunch of files for interfaces of the bean, and writing a configuration file that is complex and error-proned. EJB3.0 exploits deeply the usage of Java annotations, with which one only has to add some annotations to the bean class when building an EJB.

Then, Dependency Injection (DI, a kind of IoC pattern) is imported. DI is also used in Spring with a lot of XML files. With DI, JNDI is no longer used to look up resources, and instead the framework injects the resource or service objects into POJOs to bind the resource or service to them. And the framework and POJOs do not have to care the binding. Using DI could improve the loose coupling between POJOs and resources.

Finally, entity bean CMP is simplified. Gavin King's joining into the design of EJB3.0 Persistence brought the experiences of Hibernate. Hibernate as one of the most popular ORM frameworks to some extent attracted a lot of J2EE developers who complained the heavy weight and complexity of entity bean CMP for a long time. Gavin King as the developer of Hibernate criticized JDO unmercifully and raised the dispute of choosing the persistence technique for EJB3.0. The success of Hibernate finally made EJB3.0 almost based on Hibernate to simplify the entity bean CMP, so we have the EJB3.0 Persistence specification (JSR-220).

Although there were already several choices of the framework for building persistence layer of enterprise applications, such as Oracle TopLink, BEA Kodo and so on, there is no persistence standard that could be portable for both J2SE and Java EE platform. The EJB3 specification recognizes the interest and the success of the transparent object/relational mapping paradigm. The EJB3 specification standardizes the basic persistence APIs, which is called EJB3 Java Persistence API (JPA), and the

metadata needed for any object/relational persistence mechanism. Furthermore, JSR-220 has been widely accepted by the main ORM venders, such JBoss Hibernate, BEA Kodo, and other application server venders and JDO venders.

We can take one of those implementations of EJB3 Persistence specification as an example. Hibernate could be regarded as the base of the EJB3.0 Persistence specification, as mentioned above. Although the EJB3 Persistence API is different from that of Hibernate, the ideas in the design of EJB3 Persistence almost come from Hibernate. Compared with Hibernate API, it would be found that Gavin King just encapsulated Hibernate3, and converted the call of EJB3.0 Persistence API to that of Hibernate internal API, so that those two set of APIs are compatible. Actually, EJB3.0 does not promise calling its APIs without a container, if you would like to try on EJB3.0, your EJB3.0 components have to run in some container provided by some EJB3 vender. For example the container by JBoss is used, the call of EJB3 Persistence API (entity bean API) will be converted to the call of Hibernate API, which means that, Hibernate provides two set of APIs: one is the Hibernate native API; the other is compatible to EJB3.0 EntityBean API. For the developers who need support for distributed invocations and EJB containers, the latter of the two set of APIs are suitable; whereas for those who do not need EJB containers, the Hibernate Native API is suitable enough.

To be more concrete, JBoss Hibernate *Hibernate EntityManager* implements the programming interfaces and lifecycle rules as defined by the EJB3.0 persistence specification. Together with *Hibernate Annotations*, this wrapper implements a complete (and standalone) EJB3 persistence solution on top of the mature Hibernate core. One might use a combination of all three together, annotations without EJB3 programming interfaces and lifecycle, or even pure native Hibernate, depending on the business and technical needs of your project. One could at all times fall back to Hibernate native APIs, or if required, even to native JDBC and SQL.

Although J2EE was strictly criticized widely, for example, Rod Johnson, the founder of Spring, published a book *J2EE Development without EJB*, nowadays' pervasive acceptance and support of EJB3.0, especially its Persistence part, proves that the large changes on EJB have revived Java EE and made it lively.

## 2.2 Introduction of EJB3 Query Language

Querying on entities is an important aspect of persistence. Java Persistence Query Language (JPQL) is used for query. JPQL is the extension of EJBQL, and is imported as a part of the EJB2.0 specification. However, EJB3 Persistence API overcomes all the limitations of EJBQL (in EJB2.0), and enriches with some new

features, so that EJBQL becomes a more powerful query language. As EJB3 specification gives JPQL some exciting changes, and it actually operates on EJBs, in the following chapters, we just simply call JPQL (for EJB3) EJB3QL.

Some new features of EJB3QL over EJB2QL are as follows:

- Simplification of the syntax for query;
- Batch updates and deletes;
- JOIN operation, which is to declare joins between entities (database tables) in the FROM clause;
- GROUPBY and HAVING clauses: GROUPBY clause is to group the query results by some item to be queried that might be id variables or path expressions (refer to the following chapters), and HAVING clause contains a boolean condition to filter the grouped result;
- Subquery: an embedded query which could be used as a function returning some result, and largely improve the functionality of such query language;
- Dynamic query and Named query: 1. before EJB3.0, the queries in EJBQL could not be created and executed at runtime, but written in the configuration file and loaded as context; in EJB3.0, EJB3QL queries are allowed to be created and executed at runtime, just like SQL used as a script in programs. 2. Named queries are created and stored with its corresponding entity by using a Java annotation,

```
@Entity
@NamedQuery(name="findAllEmployee",
  query="select e from Employee e where e.empNo > p")
 public abstract class Employee implements Serializable {
 }
```

……

In order to execute the query above, EJB3 Persistence API contains the following method to get the named query (query is an instance of Class Query):

```
query = em.createNamedQuery("findAllEmployee");
query.setParameter(p,100);
return query.getResultList();
```

- Named parameter: in EJB2QL, only positional parameter is allowed; whereas named parameter is allowed as a plus, e.g. in the example above, the query could be changed to `select e from Employee e where e.empNo > :empNo` , and the code for setting the parameter could be changed to `query.setParameter("empNo",100)`.

With those new features, the functionality, usability and portability of EJBQL are really improved. After the introduction of EJB3 Persistence and EJB3QL, we will start our discussion on EJB3QL metamodel in the following chapters.

# 3. EJB3QL Metamodel

EJB3QL was issued with EJB3 Persistence Specification (JSR-220). Because EJB3 Persistence mechanism is designed mainly based on Hibernate, the query language, EJB3QL, is similar to HQL accordingly. Compared with EJBQL for EJB2.1, the new one is more precise and more powerful, and it could be created and executed at runtime, which makes this object query language more important and used more often. Given the more powerful persistence mechanism and some advantages of this new query language, EJB3QL could be used not only to operate the data in the database, but also to check the integrity of the persistence layer.

Therefore, an EJB3QL metamodel as a case study DSL model is meaningful and should be built. Moreover, in my colleague Xinhua Gu's work, MDA is applied to EJB3 area for generating EJB3 artifacts out of the domain model in UML. On the other side, OCL could be used to check the integrity of the persistence layer. For translating of OCL invariants into EJB3QL or generating EJB3QL out of some other languages, a metamodel of EJB3QL is really necessary and always needed. In this chapter, a newly designed and carefully fined metamodel of EJB3QL will be introduced, and in the next chapter, the well-formedness rules accompanying the metamodel will be described. This metamodel and the well-formedness rules are all designed based on the latest specification (JSR-220).

## 3.1 Previous Attempts and Hints for EJB3QL Metamodel

As far as we know, there is no well defined metamodel for EJB3QL. Although EJBQL 2.1 and other previous versions have been already existing for several years, the power of this persistence query language didn't show off until EJB3QL was issued. The runtime execution makes EJB3QL more useful and satisfy the critics. It's time to build a metamodel for it.

To tell the truth, we tried several times and in different ways to build this model. At the beginning, our goal is just to translate OCL invariants or some other queries into EJB3QL queries, which means that generating strings EJB3QL queries is desirable. To make the translation easier, we built a simplified EJB3QL metamodel, for which we took the OCL metamodel as a reference. But the translation of an OCL invariant into an EJB3QL query with building its AST is really hard. Although both OCL and EJB3QL have the similar capability of object query, this capability is only exerted

when object navigations or path expressions appear. Actually, OCL is quite different from EJB3QL even in the format. For example, even if the FROM clause of an EJB3QL query could be easily translated out of an OCL invariant (not that easy in fact), the where clause could be very difficult to get, for which a lot of translation recipes need to be discussed and worked out. However, finally the real translation with those recipes could still not be implemented, or implemented with lot of problems which might be caused by data types and well-formedness.

Then we dropped the simplified EJB3QL metamodel and built a so-called xQL model instead to represent an arbitrary persistence query language with considering the translation of OCL into SQL. This model is focusing on the Cartesian products or joins between tables in the database or abstract schemas in the domain persistence layer, so that the object navigations or path expressions could be satisfyingly modeled. And this xQL model also takes into account the multiply occurrences of an abstract schema, let's take a look at an example from the EJB3QL specification:

```
SELECT DISTINCT o1
FROM Order o1, Order o2
WHERE o1.quantity > o2.quantity AND
      o2.customer.lastname = 'Smith' AND
      o2.customer.firstname= 'John'
```

o1 and o2 are two occurrences of the same entity (abstract schema) Order. This case happens very often but was neglected in the last simplified EJB3QL metamodel, while in xQL it's modeled as a separate class called CPChaptericipant. And we also discussed the data retrieval path based on the so-called DependsOn tree, in which the paths for retrieving data encapsulated in objects are structured in trees with the concepts of Cartesian products or joins. However, conditional expressions and other operations as sub-conditions in a where clause was modeled as built-in operations, which is actually similar to how the OCL operations are modeled in Octopus. The difference is that, Octopus has a well defined data type system for OCL which abides by the data types defined in the OCL specification, and each built-in operation in Octopus is clearly declared with arguments' and return value's data types; whereas xQL doesn't have a data type system and its built-in operations are not clear about their arguments' and return value's data types. My colleague Veronica Novita later added some data types to xQL, but those might not well match the data types in Java and database.

Besides, we have to consider the issue of the well-formedness embedded in the metamodel. The importance of well-formedness of a model will be discussed in the next chapter. Here we only raise a problem around well-formedness in a transformation process. When we build a transformation process, which might involve more than one transformation steps, we must guarantee the correctness of the result AST of each transformation step, so that the final result, whatever an AST or

strings, could be correct and desirable. For example,

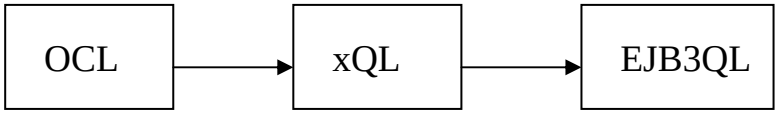



Fig 3.1 Transformation Process of OCL-xQL-EJB3QL

In the transformation process drawn above, in order to get the correct EJB3QL strings at the end, we must guarantee that the final EJB3QL AST is correct, and cascadingly the xQL AST is correct (the correctness of the OCL AST is guaranteed by Octopus). But xQL is lack of the implementation of enough well-formedness rules, for example, the well-formedness rules around data types, the duplication of id variables in the same declaration scope (e.g. o2 in the example above is no longer o2 but o1 instead, which means that an entity occurrence is declared twice with a duplicate id variable o1), and a lot of other well-formedness rules. Although the EJB3QL AST could be built correctly sometimes, this transformation is still vulnerable and couldn't be correct all the time.

The data types and well-formedness are greatly important to metamodels of any languages (including DSLs such as EJB3QL). So I worked out another metamodel which I just called EJB3QL-R (based on a metamodel called RQL we built). In this metamodel, we took into account data types, and modeled the data types system using data-typed expressions, which means that, all the expressions which could return a value must be data-typed by categorizing. For example, an expression which returns a string should be put into the category of string expressions. To model the categorizing, an interface for each category, e.g. an interface called StringExp for string expressions, should be created, and all the expressions in this category implement this interface. Obviously, the expressions which return a value are all data-typed. So the operations or expressions are modeled as classes instead of built-in operations. And the arguments' data types are also clear, for example, the logical operation ANDExp, both its two arguments should be of Boolean type, so two uni-directional associations to BooleanExp are added to ANDExp. Besides those data-typed expressions and stricter obeying the grammar, a lot of OCL invariants were also composed for well-formedness rules.

It seems that this EJB3QL-R metamodel is quite complete. But some problems still popped up when we talked about some issues. For example, aggregate expressions are important constructs in EJB3QL (also in SQL) and used very often in the real applications, first, an aggregate expression could return a value of numeric, string, or datetime, which means that the return type is not clear but depends on its arguments; second, an aggregate could be used not only in the select clause, but anywhere else where a value of numeric, string or datetime is needed, so it could appear in the select clause, where clause, having clause, or subqueries. And also some complex

problems existed at data types: for an entity type, an occurrence of that entity could be also treated as a type or not, and if not, what its type should be and that occurrence might be referred as an argument of some expressions which require a data-typed argument; as a query usually return a collection of values (or tuples) and a collection-valued path expression is standing for a collection of entity values, how those collection-valued expressions should be typed.

Obviously, EJB3QL-R could give us an answer to the problems talked above. But it gave us some hints such as using interfaces to make expressions data-typed. And more importantly, previous tries told us that, 1, Well-formedness is significant to transformations between models, and those around data types are an considerable chapter of all the well-formedness rules; 2, Data type system is necessary for the metamodel of any language, and all the expressions which might return a value should be clear about its return type and arguments' types; 3, A valued expression should be of a single type, and have no other possibilities in data types; 4, A metamodel of a language must be created strictly based on its grammar and specification, which contains all the well-formedness rules of that language; 5, For EJB3QL, the entity data types must be taken into account and well handled.

Given the hints above, we can come to the latest EJB3QL metamodel in the following sections. We will first give an overview and then describe each chapter of it.

## 3.2 Overview of EJB3QL Metamodel

In the last section, some previous attempts we have made to model EJB3QL are talked, and we get some hints for the latest metamodel of EJB3QL we will discuss. All the hints are around well-formedness, which is actually could be guaranteed in two ways, one is to embody in the metamodel with using UML, and the other is to compose OCL invariants and other queries. Most data type related well-formedness could be expressed when modeling with UML, for example, the arguments of an operation could be modeled using associations to other data-typed expressions. And other well-formedness should be expressed by OCL. In fact, some well-formedness could be modeled using UML and also could be expressed by OCL, more classes lead to less OCL queries, whereas less classes lead to more OCL queries; that's a tradeoff. Therefore, well-formedness of this metamodel will be mainly discussed in the next chapter, and in this chapter modeling of data types is described.

This EJB3QL metamodel was created strictly according to the EBNF grammar of EJB3QL (Appendix) given in the specification (JSR-220). Almost all the terminals and non-terminals which appear in the EBNF grammar of EJB3QL are modeled in the

metamodel, and of course some assistant constructs are also needed. And these classes in the metamodel are structured into several packages as follows, due to their functionality in EJB3QL queries.

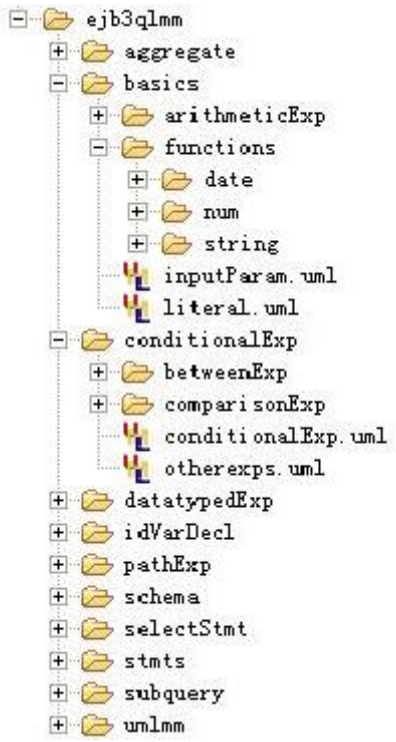



Fig 3.2 Package Structure of EJB3QL Metamodel

Package **schema** contains the data type related classes: some are for the primitive types, whereas the others are for the entity types. It's important that the abstract schema, which is corresponding to a database table, is modeled, because the abstract schema is the base for entities, id variable declarations, object navigations, state fields, and so on. Besides, primitive types are related to Java data types and some database data types.

Package **pathExp** is modeling the object navigation paths. Association paths and state field paths are modeled separately; meanwhile the state field is also modeled.

Package **idVarDecl** is modeling entity id variable declarations. An id variable is standing for an occurrence of an entity, and it must be declared before used. This package contains the classes for all the constructs where an id variable could be declared.

Package **aggregate** models the aggregate expressions related constructs. Because of the return type of an aggregate expression, it's necessary to create different classes due to the different return types, also with considering its function type, such as SUM, AVG, etc.

Package **datatypedExp** contains a bunch of interfaces as a part of data type system. Any expression which might return a value should implement one of those interfaces, which is taken from EJB3QL-R.

Package **basics** contains some basic constructs which should not be structured into other packages, such as literals, input parameters, EJB3QL functions which contains varieties due to the return type, arithmetic expressions, etc.

Package **conditionalExp** models those expressions which could be used as a condition in the where and having clauses. Each of them returns a Boolean value. Two sub-packages are created for some complex expressions.

Package **subquery** models the subquery related contructs. Although a subquery is similar to a select statement, it is still different, because it always returns values of a single type and thus could be used as an argument of some expressions. So it should be modeled independently.

Package **selectStmt** contains classes and interfaces used to model a select statement. The constructs modeled in the packages above would be combined and made use of in the more complex constructs, such as the select statement, so a lot of associations to the constructs in other packages are needed.

Package **stmts** models the update statement and delete statement, because the select statement is more complex and important, it is modeled in a separate package.

Package **umlmm** only contains a class that is standing for the classifiers in a UML model, which is referred by an entity in the schema, because as we know, an entity in the persistence layer is always corresponding to a classifier in the domain model.

The components and the role in the metamodel of all the packages are briefly described above. And according to those as a guideline, in the following section, we will detailedly go through each part of this EJB3QL metamodel and explain some design issues.

## 3.3 Design of EJB3QL Metamodel

Modeling a language needs to consider both abstract conceptual and implementational problems, otherwise the generated code would have redundancy and errors. Of course, some problems about the code generation depend on the modeling and code generation tool, which might be not consummate and cause some errors, and we will

discuss this issues later. As to the language itself, EJB3QL is a DSL supporting Java persistence operations, therefore, when we design this metamodel, Java and even database schema should be in our consideration, such as the modeling of the most fundamental package **schema**.

### 3.3.1 package ejb3qlmm.schema

The class diagram of this package is in Fig 3.3. Here only attributes and necessary methods of the classes are shown, the methods for well-formedness rules and serialization are all hidden for concision. The classes in this package mainly contribute to the database schema and definition of the persistence layer, which provide the base for EJB3QL queries. And also some of the classes deal with the data types.

The following words are cited from the specification (JSR-220):
*The persistent fields or properties of an entity may be of the following types: Java primitive types; java.lang.String; other Java serializable types (including wrappers of the primitive types, java.math.BigInteger, java.math.BigDecimal, java.util.Date, java.util.Calendar[4], java.sql.Date, java.sql.Time, java.sql.Timestamp, user-defined serializable types, byte[], Byte[], char[], and Character[]); enums; entity types and / or collections of entity types; and embeddable classes.*

Class **SupportedJavaType** is standing for any data types supported in Java language except entity data types, and it contains a lot of static methods (not shown in the diagram) to each return a static instances of **SupportedJavaType** to represent the type of a certain expression. And the name of such a static instance indicates which data type this instance is standing for.

Those static methods all return instances for four kinds of primitive data types, numeric, boolean, string, and datetime. Actually, string and datetime are not primitive data types in Java, but here we treat them as primitives, as they are different from collection types and entity types. In Java, java.lang.String is the only one type for strings, whereas the other three kinds of primitive data types have various concrete types or classes correspondingly. For numerics, Java provides not only primitives like int, short, long, float, double, byte and char (byte and char are treated like integer), but also wrapper classes for each primitive like Integer, Short, Long, Float, Double, Byte and Character in package java.lang. Besides, two more classes for arithmetic in package java.math, BigInteger and BigDecimal, are also should be taken into account. And the boolean type is corresponding to the primitive type boolean and its wrapper java.lang.Boolean. Finally, the datetime type is supported in both programming languages and databases, so classes Date and Calendar in package java.util, and classes Date, Time, Timestamp in package java.sql should be included. Now all the

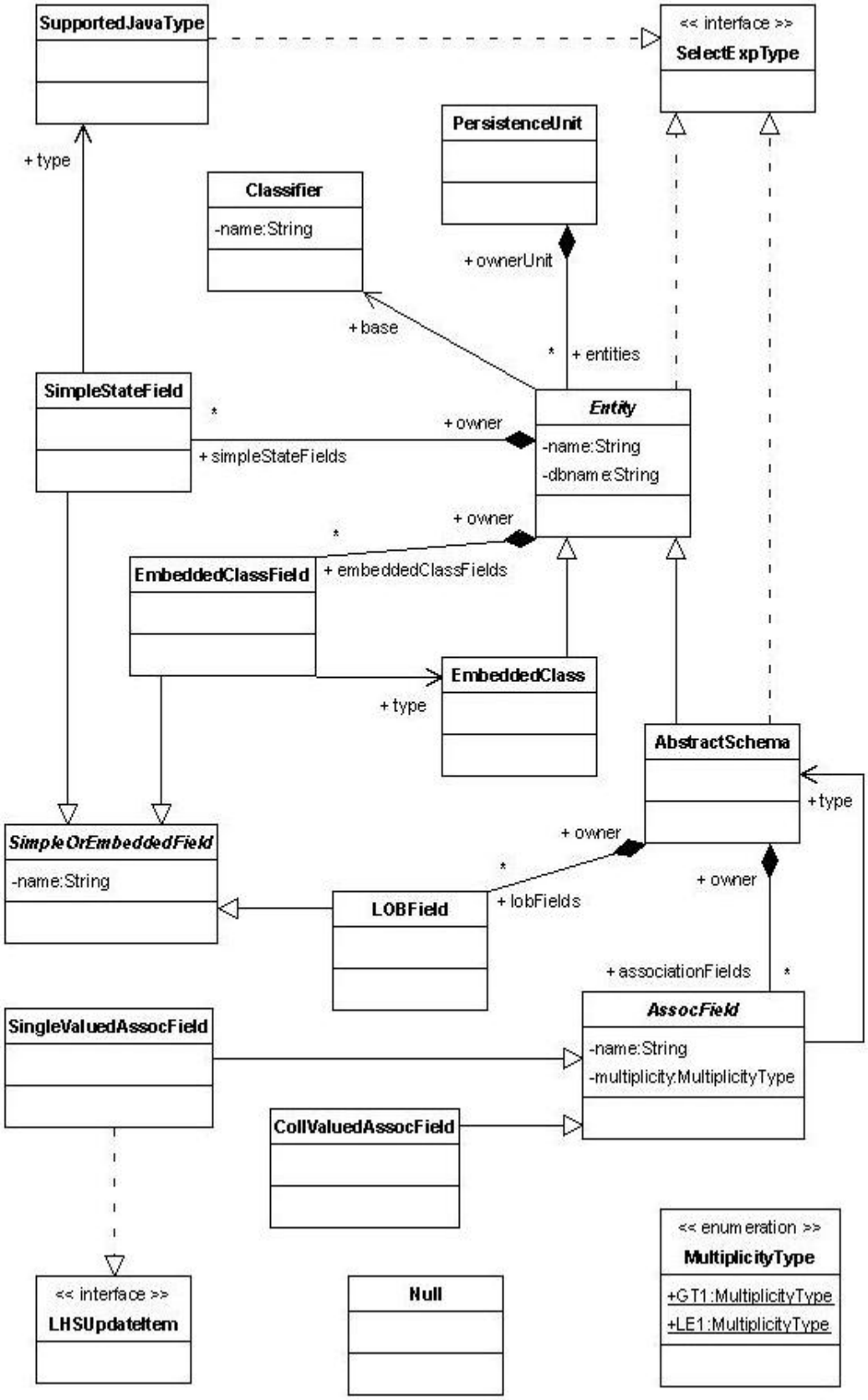



Fig 3.3 Class Diagram of Package ejb3qlmm.schema

static instances and methods are ready for the primitive data types supported in Java.

There are still some static methods for primitive data types are needed to check the type matching, such as isNumeric(arg : SupportedJavaType) : Boolean is to check a given type is numeric or not. Since Java provides various primitive types and classes for numeric values, and those of different numeric types are allowed to be calculated together, but not allowed for other types, then another static method is needed to check whether two given values are compatible in type or not, which is areTypeCompatible( t1 : SupportedJavaType, t2 : SupportedJavaType ) : Boolean.

Let's turn to collection types. Notice that some instances of **SupportedJavaType** additional to those returned by the static methods are necessary. Let's suppose that, a collection-valued field in a Java class has no annotations marking it as implementing an association end, and then it will be stored as a BLOB by the ORM engine. The standard specification does not require that its elements can be referred to in EJB3QL queries (this model does not either). Those fields will be therefore simply recognized as **SimpleStateField** according to the EJB3QL specification, and thus an instance of **SupportedJavaType** is needed at this moment to represent their data types. For example, a UML attribute "`lineItems[1..10] : LineItem`" or a Java field "`java.util.List lineItems = new ArrayList<LineItem>()`" without an association-related annotation will be mapped as BLOB, thus an instance of **SupportedJavaType** is needed to represent the type of such field, i.e. an instance whose name is "java.util.ArrayList<LineItem>". This issue is relevant for example in EJB3QL queries which compare two such fields for equality. The AST of such expression is well-formed, if and only if two fields having the same type are compared, i.e. whose type() (explained later) operations return the same instance of **SupportedJavaType**. If only one catch-all instance (e.g. BLOB) of **SupportedJavaType** is made available at runtime, then the expressions where a lineItems **SimpleStateField** is compared for equality with an addresses **SimpleStateField** are well-formed, because they have the same **SupportedJavaType** instance that has the name "java.util.ArrayList". In contrast, if one instance of **SupportedJavaType** is devoted to "java.util.ArrayList<LineItem>" and another to "java.util.ArrayList<Address>", then that sub-expression will be reported as not well-formed.

Those types discussed above are provided by Java, or we can call them Java built-in types and classes. However, the user-defined types and classes are also regarded as data types. Enumerations are special user-defined types, because values of enumerations are actually integers and can be computed. This is different from ordinary classes which are standing for entity types; therefore, although they are user-defined, we still make them in **SupportedJavaType**.

An important type can not be missed that is **Null**. Although only a few places may be NULL in EJB3QL queries, e.g. in the update statement a field to be updated could

be NULL, it is absolutely indispensable.

Then we come to user-defined classes, i.e. entity types. There are two kinds of entities, **AbstractSchema** and **EmbeddedClass**. An abstract schema is always treated by the ORM engine as an independent persistence entity not pertaining to other entities, and its class is marked with an entity annotation, and it can have entity id variables declared in EJB3QL queries. Different from abstract schema, embedded classes are not independent persistence entities, but pertain to other entities, although they have corresponding database tables; that is, their instances often serve other entities as some composite attributes. For example, given a class FootballTeam and a class Member, a football team might have over 20 members, and a team member should have a name, a number and his position in a match, then the class FootballTeam is a persistence entity, whereas the class Member could be a persistence entity or an embedded class. If it is an embedded class and pertains to FootballTeam, the instances of Member only acts as the elements of a collection attribute of class FootballTeam, and can not be used or even exist independently.

No matter an abstract schema or an embedded class it is, an entity always corresponds to a classifier in the domain model and consists of fields, which are among four different kinds (only the first two kinds of fields in the following for embedded class). First, **SimpleStateField**, which refers to a primitive data type or an enumeration type; Second, **EmbeddedClassField**, which refers to an embedded class as its type; third, **AssocField**, standing for association field, which refers to an abstract schema as its type; Fourth, **LOBField**.

As to association fields, when two tables, more exactly two persistence entities, are associated or joined with each other, there must be an association field; that is, only persistence entities could be involved in a join, and no embedded classes are allowed for being joined, therefore, an association field (association end) is just standing for its pointing abstract schema. An association field always has a multiplicity of two types, greater than one and less than or equal to one, which are represented by the enumeration **MultiplicityType** with two values, **GT1** and **LE1**. Depending on the multiplicity, an association field could represent to a single object or a collection of objects of the abstract schema that is its referring type, and thus we should have class **SingleValuedAssocField** and **CollValuedAssocField**.

Some of the lob fields are caused by collection-valued fields not marking with an association annotation, which is already discussed in a previous paragraph. Those fields whose values are stored as LOBs are not typed as **SupportedJavaType**, and are not typed as **Entity** (neither as **AbstractSchema** nor as **EmbeddedClass**). So it does not refer to a type.

As to **PersistenceUnit**, let’s take a look at its definition from the specification:
*A persistence unit defines the set of all classes that are related or grouped by the*

*application and which must be colocated in their mapping to a single database.*

A persistence unit contains the domain classes which are to be persisted, no matter abstract schema or embedded classes they are.

### 3.3.2 Package ejb3qlmm.pathExp

A path expression is representing an object navigation, which could be used almost everywhere in EJB3QL queries. According to the specification, there are two kinds of path expressions, **AssocPathExp** and **SingleValuedPathExp**. The grammar is as follows:

**association_path_expression ::=**
**collection_valued_path_expression | single_valued_association_path_expression**

**single_valued_path_expression ::=**
**state_field_path_expression | single_valued_association_path_expression**

Although they have the same alternative, single_valued_association_path_expression, what they focus on is different. An association path expression is focusing on the joins between abstract schemas, and it is joins that enable the object navigations. Joins always involve associations and of course association fields (association ends). Every association path should have an entity id variable standing for an occurrence of an abstract schema, which acts as the starting point in the chain of joins. Thus class **AssocPathExp** has two uni-directional associations with **IdVarDecl** (from package ejb3qlmm.idVarDecl) and **AssocField** (from package ejb3qlmm.schema). Since the multiplicities of association ends could be greater than one or not greater than one, an association path might result in a collection of objects or a single object, and then an association path expression should be specialized into **CollValuedAssocPathExp** and **SingleValuedAssocPathExp**. Accordingly, class **AssocPathExp** has a method, `isCollValued():boolean`, to check an instance of it is collection-valued or single-valued. The well-formedness rules for the chaining of association fields will be discussed in the next chapter.

Single-valued path expressions are focusing on the value it results in. The value could be of primitive or entity type, not a collection. **SingleValuedAssocPathExp** is a specialization of **SingleValuedPathExp**, since it is always valued to a single object as discussed above. A **StateFieldPathExp** must terminate at a **StateField**, and there are two possibilities for the construct leading to the state field, an entity id variable or a single-valued association path expression, because a state field is owned by some entity, and should be resulted by an instance of the entity. Just take the example in section 3.1, `o1.customer.lastname` and `o1.number` are both allowed.

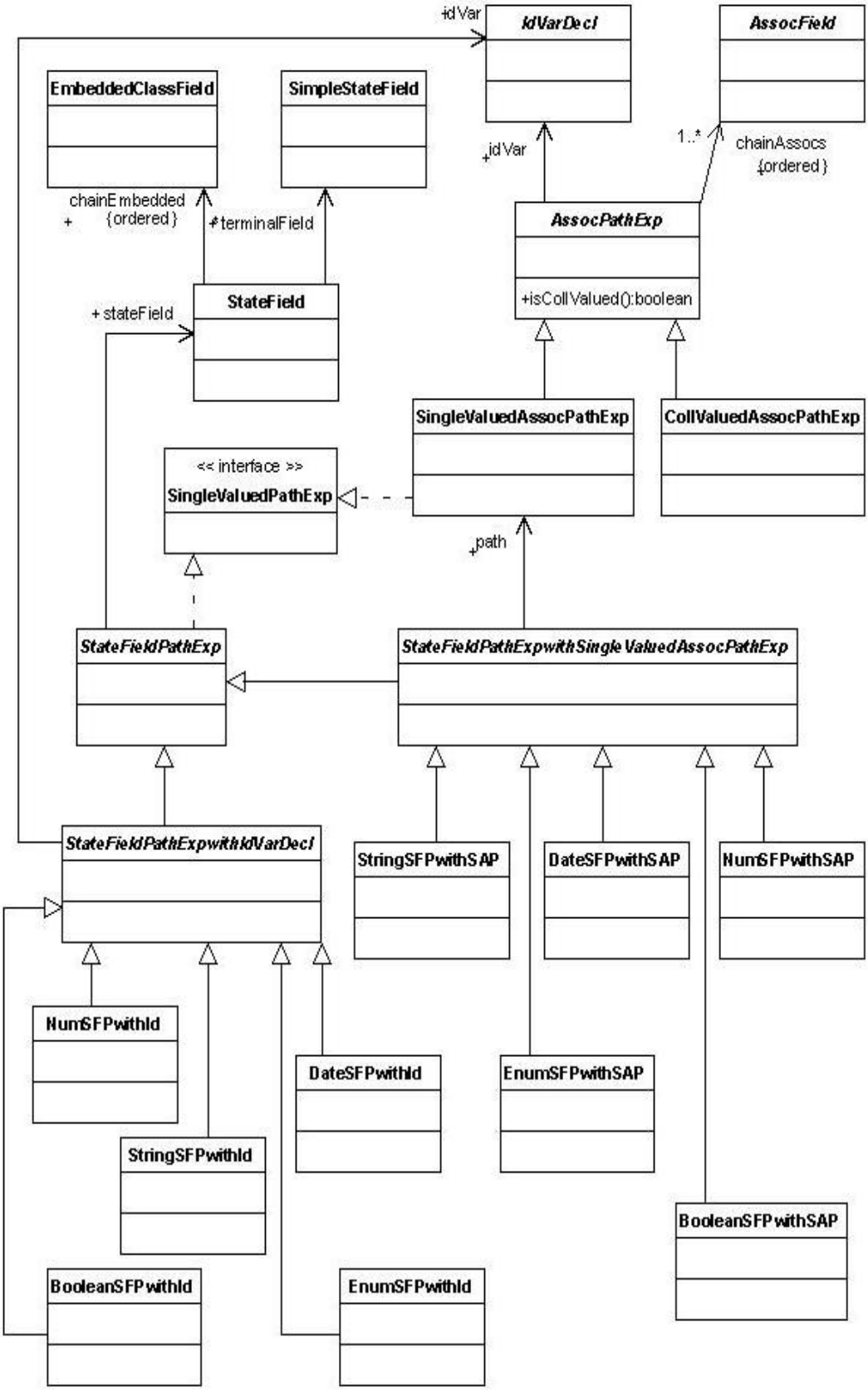



Fig 3.4 Class Diagram of Package ejb3qlmm.pathExp

As a state field path expression results in a primitive value (although an embedded class field might exist, it finally terminates at a simple state field), this value could be of different primitive types or enumeration types. Based on the idea of data-typed expressions, the state field path expressions are also data-typed, so subclasses of the two kinds of state field path expressions are created for each primitive types and enumeration types.

### 3.3.3 Package ejb3qlmm.idVarDecl

An entity id variable represents an instance of an abstract schema, which is an occurrence of that abstract schema in the EJB3QL query, so an instance of class IdVarDecl should always have a name, attribute **alias**, for the id variable it represents. According to the specification, it is only possible for an id variable to be declared in the FROM clause, and in four forms:

1) In a range variable declaration, where an id variable for an explicitly given abstract schema is declared to range over it and a chain of joins could be started;
2) In a collection member declaration, where an id variable is declared to represent one of the instances resulted by a collection-valued association path expression;
3) In a join declaration, where a join to the preceding abstract schema (maybe appearing explicitly in a preceding range variable declaration or implicitly in a preceding join declaration) happens and leads to a new id variable;
4) In an association path expression that must be in a subquery's FROM clause, which is the only position that an id variable is allowed to be declared in association path expression.

In the following example, which is composed based on the Loyalty&Royalty model given in the appendix, the four cases that an id variable is declared are shown:

```
SELECT  c
FROM  Customer  c                        -- 1)
       JOIN  c.cards  cd ,                       -- 3)
       IN ( cd.transactions )  t                    -- 2)
WHERE  t.points = 100  AND
        EXISTS(  SELECT  lp
                 FROM  c.programs  lp                -- 4)
                 WHERE  lp.name = 'Pizza'  )
```

Based on the four possibilities above, there should be four subclasses of **IdVarDecl** for each of them, then we have **RangeVarDecl**, **CollMemberDecl**, **JoinDecl**, and **SubqueryIdVarDeclwithAssocPath**, which is actually declared and included in package **subquery** and will be introduced when we move to that package. As to the

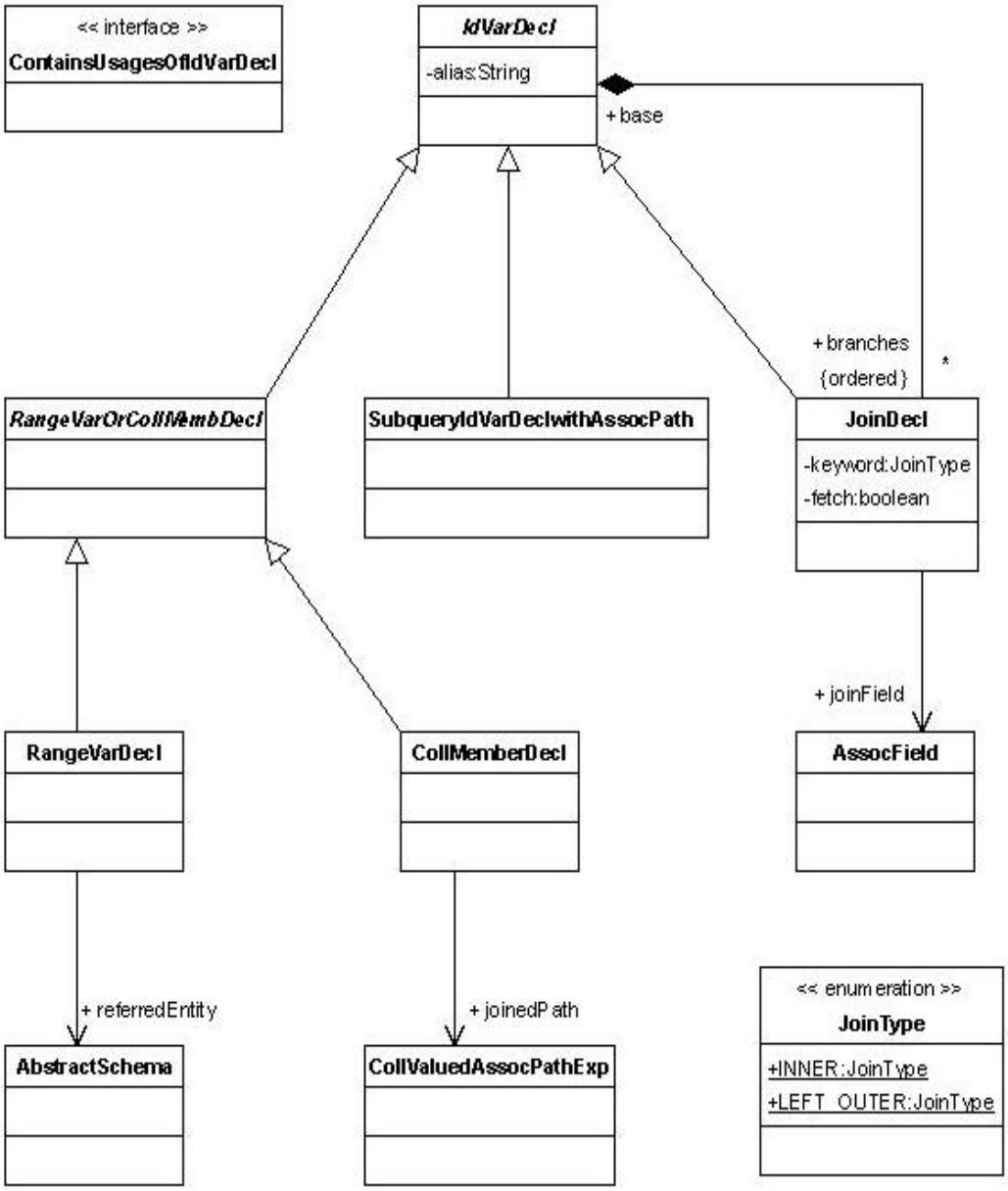



Fig 3.5 Class Diagram of Package ejb3qlmm.idVarDecl

class **RangeVarOrCollMembDecl**, this generalization is created due to the grammar as follows:

```
from_clause ::= FROM identification_variable_declaration
                {, {identification_variable_declaration | collection_member_declaration}} *
```

Here this identification_variable_declaration is equivalent to our class **RangeVarDecl** in the metamodel. So it's clear that **CollMemberDeclaration** is treated in a similar way as **RangeVarDecl**, although a collection member declaration is functionally

similar to a join. And thus the generalization is created to make easier some operations like toString().

**JoinDecl** is undoubtedly the most complex part of this package, which involves how joins are declared and organized in the FROM clause. The following complex example will show us the joins' organization:

```
SELECT  lp
FROM  LoyaltyProgram  lp  JOIN  lp.partners  pt
                      JOIN  pt.deliveredServices  ds
                      JOIN  lp.levels  sl
WHERE  EXISTS (  SELECT  as
                 FROM  sl.availableServices  as
                 WHERE  ds = as  )
```

This query is to search some loyalty programs, each of whose partners provides the services that are already be available at the programs' service levels. This is just an example to show joins, even if it is not that meaningful. In this example, three joins are not resulted by four tables joining end to end in a line with each other, but instead there are two branches from the entity `LoyaltyProgram`, which makes a tree structure of the joins as follows:

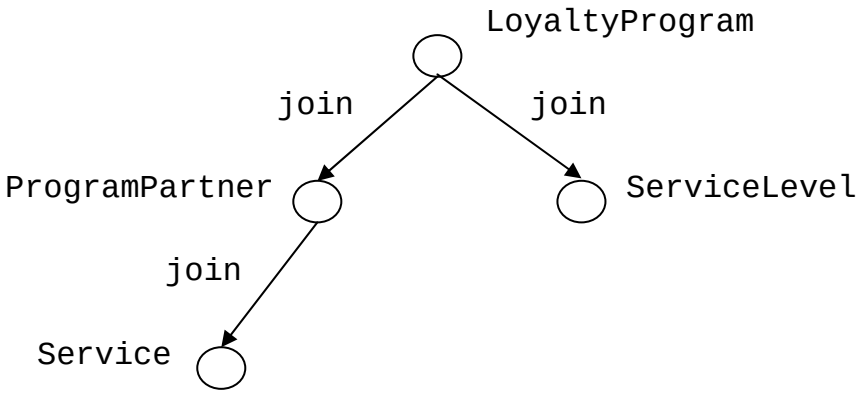



Fig 3.6 Structure of Joins

Therefore, the association between **IdVarDecl** and **JoinDecl** is suitable and necessary. Here the IdVarDecl must be down casted to subclasses **RangeVarDecl** and **JoinDecl**, because only these two constructs could have joins. And thus each range variable declaration or join declaration could have arbitrary entities joined with it as its branches to form a subtree.

The enumeration **JoinType** has two alternative values for inner join and outer join, and for fetch join, **JoinDecl** has a boolean attribute to indicate it.

The interface ContainsUsagesOfIdVarDecl is created to retrieve the usages of id variables and check the validity of them, which contributes to the well-formedness and will be discussed in the next chapter.

### 3.3.4 Package ejb3qlmm.aggregate

Aggregate expressions are used like an operation in EJB3QL queries, because an aggregate expression takes as its argument an id variable, a state field path expression, or a single-valued association path expression, and returns a value of numeric, string, or datetime.

The return type of an aggregate expression depends on its operator and argument:

1) When the operator is **COUNT**, the return value must be a numeric value.
2) When the operator is **MAX** or **MIN**, if the argument, which must be a state field path expression, is of type numeric, the return type is numeric; or if the argument is of type datetime, the return type is datetime.
3) When the operator is **SUM** or **AVG**, the argument must be a state field path expression with a numeric value, and the return type is undoubtedly numeric.

Given those possibilities above, the class **AggregateExp** should be specialized into two subclasses, **AggregateExpCount** and **AggregateExpNonCount**. The former obviously represents the aggregate expression with operator **COUNT**, and this should be data-typed as a numeric expression. The latter should be further specialized, because all the expressions returning a value should be exactly data-typed. Then three subclasses are created, **AggExpNonCountString, AggExpNonCountNum** and **AggExpNonCountDate**. They are respectively data-typed as string expression, numeric expression and datetime expression.

The enumeration **AggregateType** is to indicate the operator that an aggregate expression might have. However, it only contains four types, **MAX**, **MIN**, **SUM** and **AVG**, but no **COUNT**, as the aggregate expressions with **COUNT** are already modeled independently as **AggregateExpCount**. And this enumeration is only used by **AggregateExpNonCount**.

There is another point to mention: aggregate expressions, especially those with **SUM** and **AVG**, are often, but not always, used with a GROUPBY clause in an ejb3ql query. In most cases, a GROUPBY clause is needed at the point of view of semantics, but according to the JSR-220 spec (page 102), some usages of aggregate expressions without a GROUPBY clause are also allowed and semantically correct, no matter the aggregate expressions appear in the WHERE clause or some other places.

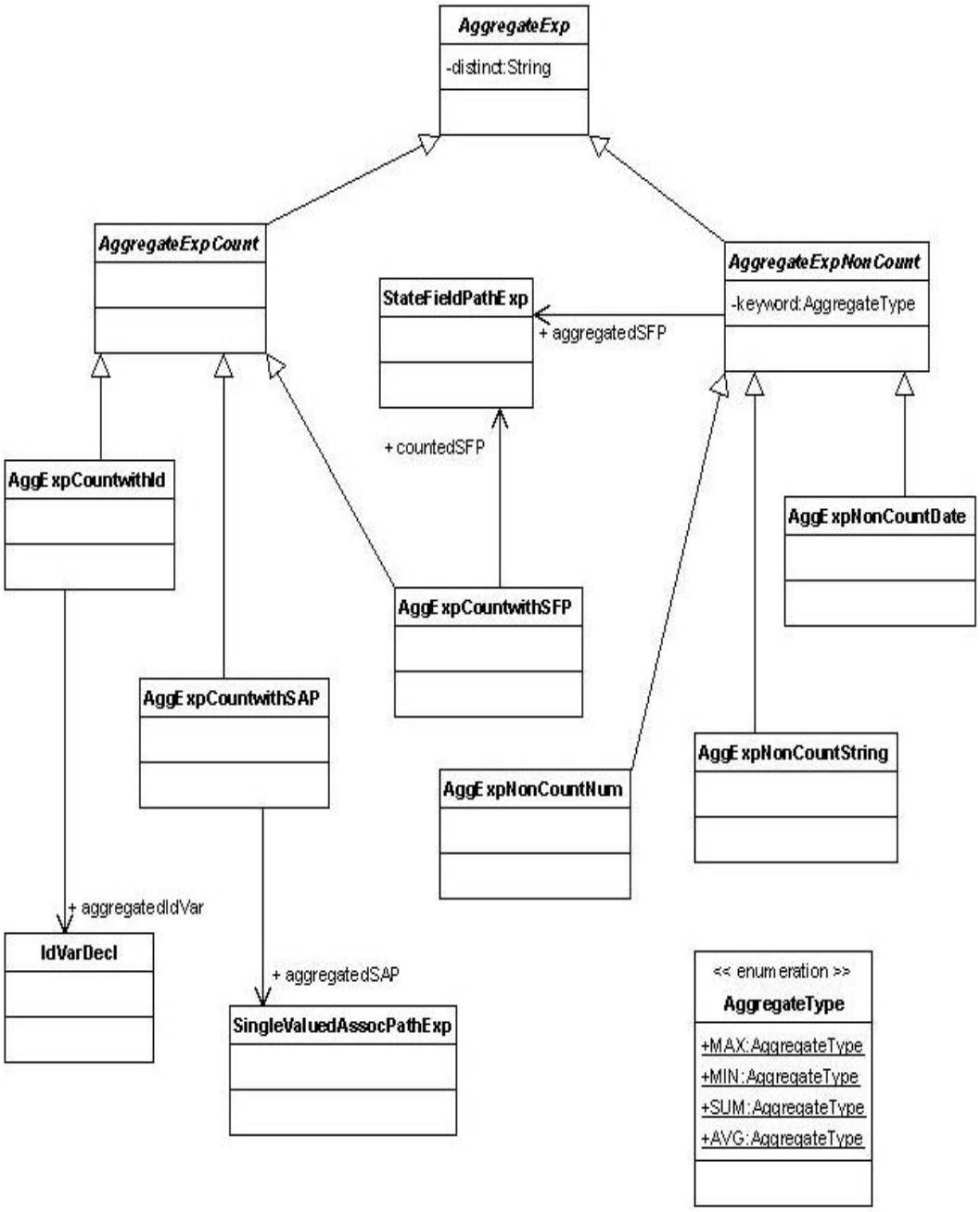



Fig 3.7 Class Diagram of Package ejb3qlmm.aggregate

With using an aggregate expression in the SELECT clause, a subquery, which is allowed for containing only one object to be selected in its SELECT clause, could have a return value of some primitive type (if a state field path expression for a primitive type is in its SELECT clause and only return a single value, the return type of this subquery is also primitive), and thus subqueries could be used as operations in EJB3QL queries. Details about subqueries will be discussed in the section on them.

### 3.3.5 Package ejb3qlmm.datatypedExp

As discussed in the previous sections, the data-typed expressions are adopted from EJB3QL-R to be a part of the data type system of this EJB3QL metamodel. And the expressions should be data-typed through implementing a certain interface which represents a kind of data-typed expressions. Therefore, those interfaces are necessary and declared in this package (interfaces for numeric type are declared in package `ejb3qlmm.basics.arithmeticExp`), as shown in the following picture.

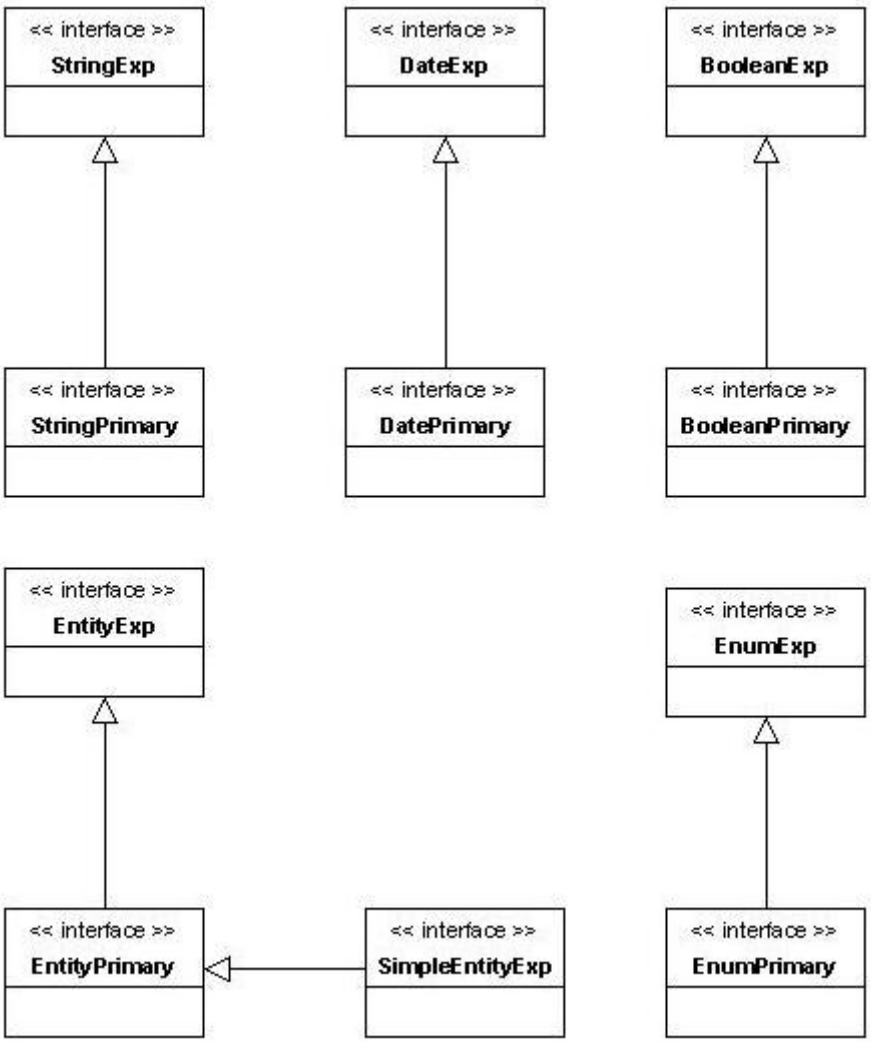



Fig 3.8 Class Diagram of Package ejb3qlmm.datatypedExp

Those in the picture above are only the interfaces, and it's necessary to know which classes (EJB3QL constructs) implement them. Actually, in the previous discussion, we have already met with some classes which should be data-typed, for example, in

package `ejb3qlmm.aggregate`, **AggExpNonCountString** should be of string type, and implement **StringPrimary**, whereas in package `ejb3qlmm.pathExp`, **BooleanSFPwithSAP** should be of boolean type, and implement **BooleanPrimary**. However, this is not complete, and there are a lot of expressions to be data-typed. In this section, we are about to take a look at which expressions should be data-typed by implementing which interface.

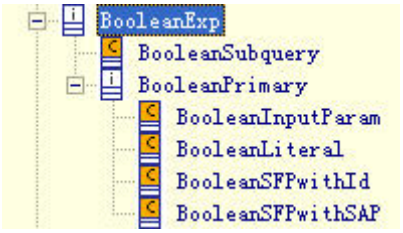



Fig 3.9 Type Hierarchy of BooleanExp

Fig 3.9 gives all the expressions implementing **BooleanExp**. As discussed before, a subquery could return a value of primitive type that includes boolean type. To distinguish the subquery and other simple data-typed expressions, a sub-interface, like **BooleanPrimary** and **StringPrimary**, etc., is created for each typing interface. For **BooleanPrimary**, besides the state field path expressions we have already talked, the input parameter and literal of boolean type are also included in this hierarchy, which will be introduced in the next section.

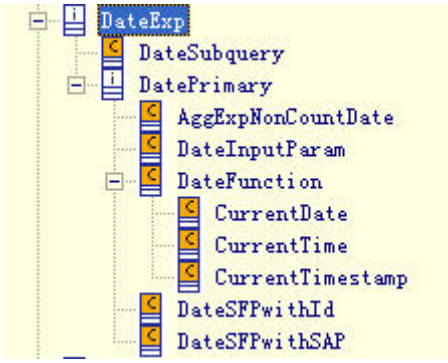



Fig 3.10 Type Hierarchy of DateExp

Fig 3.9 shows the expressions implementing **DateExp**. Different from **BooleanExp**, datetime expressions includes aggregate expressions returning a datetime value and a sub-hierarchy for datetime functions, but no datetime literals. DateFunction has three sub-classes, **CurrentDate**, **CurrentTime**, and **CurrentTimestamp**, each of which represents a function returning a datetime value. These functions are implemented by the framework that implements JSR-220 calling the built-in functions of the background database.

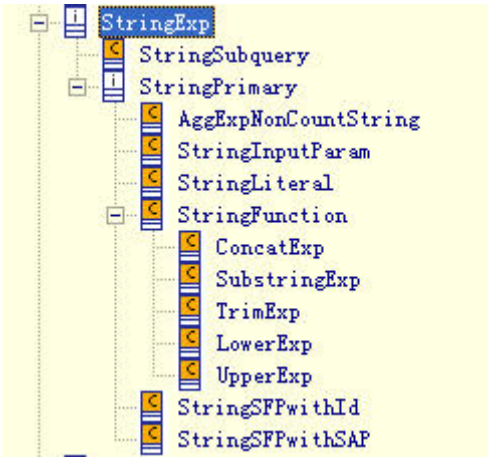



Fig 3.11 Type Hierarchy of StringExp

The type hierarchy of StringExp shown in Fig 3.10 is similar to that of DateExp, except that there is **StringLiteral**.

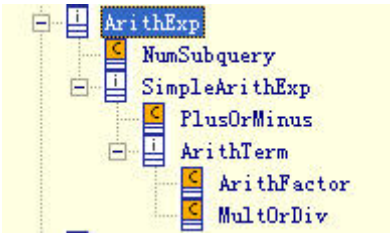



Fig 3.12 Type Hierarchy of ArithExp

Actually **ArithExp** and its subclasses are not declared in this package but in package `ejb3qlmm.basics.arithmeticExp`. However, they indeed contribute to the data-typed expressions, so it's necessary to mention here. You might find that the type hierarchy of **ArithExp** for numeric expressions is simpler than what you expect, in that numeric expressions includes the arithmetic calculating operations which are already modeled as subclasses in this diagram, like **PlusOrMinus**, **MultOrDiv** and **ArithFactor**, and these operations' arguments might be themselves or other numeric expressions that we can call simple numeric expressions in contrast to those arithmetic calculating operations that we can call composite numeric expressions. And those simple numeric expressions are modeled in another hierarchy as the subclasses of **ArithPrimary**, which we will discuss in the next section.

For the expressions returning an entity instance, we would better first refer to the related part of the grammar in JSR-220:

**entity_expression ::=**
**single_valued_association_path_expression | simple_entity_expression**

```
simple_entity_expression ::= identification_variable  |  input_parameter
```

Obviously, entity expressions include single-valued association path expressions, input parameters and id variables. But if a subquery returns instances of an entity, should it be data-typed as an entity expression? Yes, I think so, and let's take a look at an example as follows:

```
SELECT  lp
FROM  LoyaltyProgram  lp
        JOIN  lp.partners  pt
        JOIN  pt.deliveredServices  s
        JOIN  s.level  l
WHERE  l =  ANY ( SELECT  sl
                  FROM  lp.levels  sl )
```

This query is to search some loyalty programs, each of whose partners provides the services that refer to one of the programs' service levels. Although the subquery returns a collection of instances of entity **ServiceLevel** in RandL model, and the **ANY** expression makes it like a single-valued expression, the data type of elements of the collection returned by the subquery is that entity, and the return type of that **ANY** expression just follows that of the subquery. Although **ANY** or **ALL** expressions with subquery only happen in comparison for equality, we still should include subquery returning entity value in the type hierarchy of **EntityExp** as one of its implementations. This could also explain why we have a subquery of different types in each data-typed expressions hierarchy.

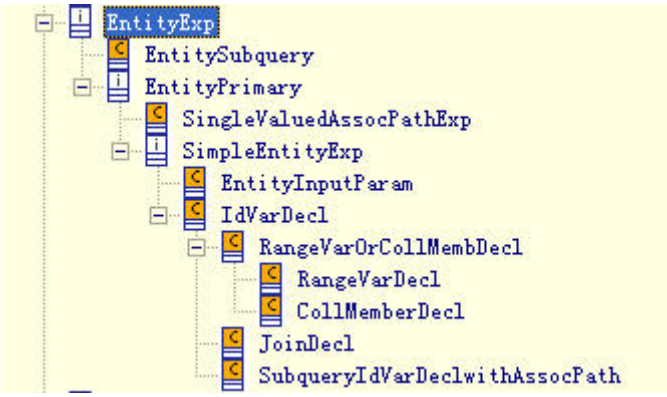



Fig 3.13 Type Hierarchy of EntityExp

Besides **EntitySubquery** returning perhaps more than one entity values, other implementations of **EntityExp** in this hierarchy all return a single entity value. This sub-hierarchy, in which the constructs return a single entity value, is rooted at **EntityPrimary**, and according to the grammar, there are three possibilities that a single entity value could appear:

1) A single-valued association path expression always returns one entity instance.
2) An entity instance could appear in the form of an input parameter.
3) An id variable is standing for an occurrence of an entity, or a single instance of that entity.

Therefore, **SingleValuedAssocPathExp**, **EntityInputParam**, **IdVarDecl** and its subclasses are all data-typed under **EntityPrimary**. Since these constructs except single-valued association path expressions could appear as new values in UPDATE statements, a subtree rooted at **SimpleEntityExp** is modeled to be distinguished from single-valued association path expressions.

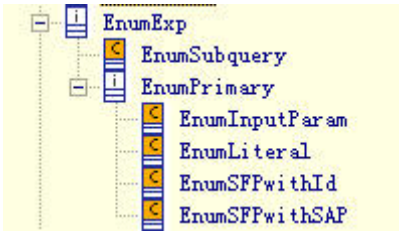



Fig 3.14 Type Hierarchy of EnumExp

For the type hierarchy of EnumExp, it is similar to others talked above, and we will not give unnecessary details.

### 3.3.6 Package ejb3qlmm.basics

This package mainly contains some basic constructs, such as literals, input parameters, EJB3QL functions, arithmetic expressions, etc.

Fig 3.14 and Fig 3.15 shows class **InputParam**, **Literal** and their subclasses. Because either an input parameter or a literal is standing for a value, this must be also data-typed. And thus their subclasses are created due to the data types which are possible for the constructs (there is no entity literal or enum literal). Besides those subclasses, associations for explicitly referring to their types are also necessary, so **Literal** and subclasses of **InputParam** are each associated with an type class, **SupportedJavaType** or **Entity**.

EJB3QL functions were already mentioned a little in the last section about functions for datetime type. Here we only introduce the EJB3QL functions for other data types.

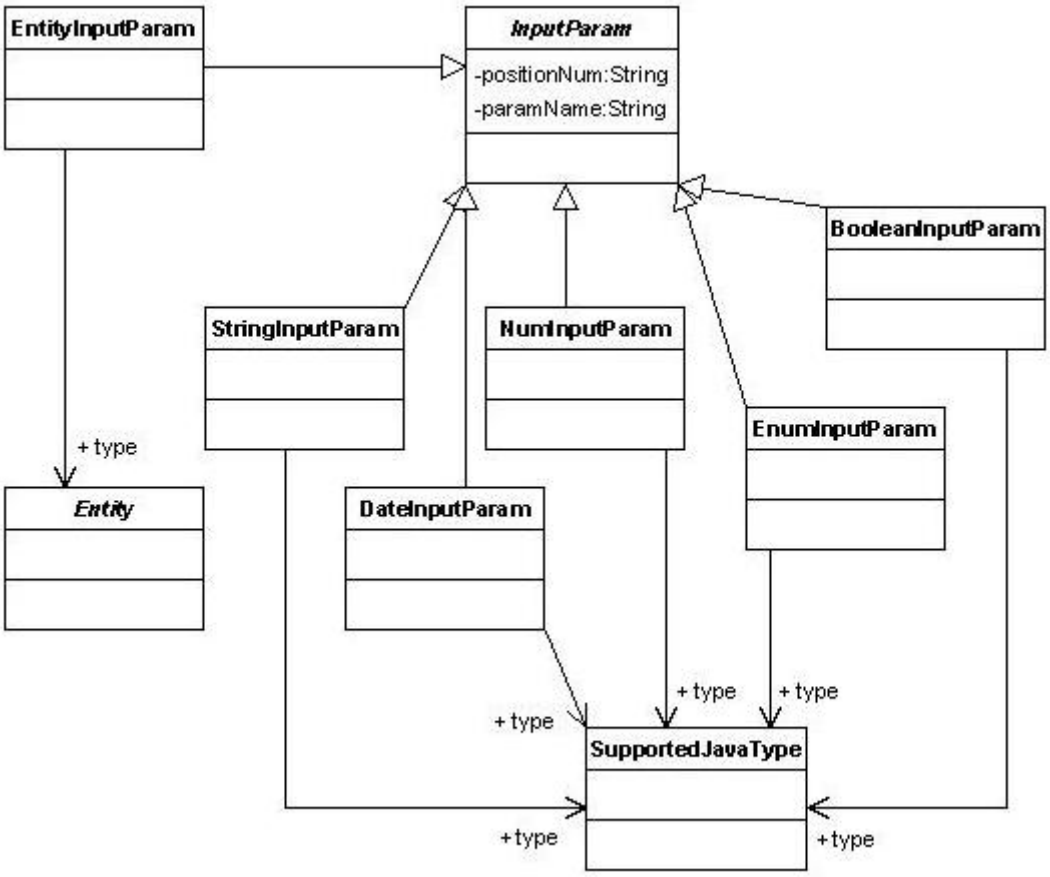



Fig 3.15 Class Diagram of InputParam and its subclasses

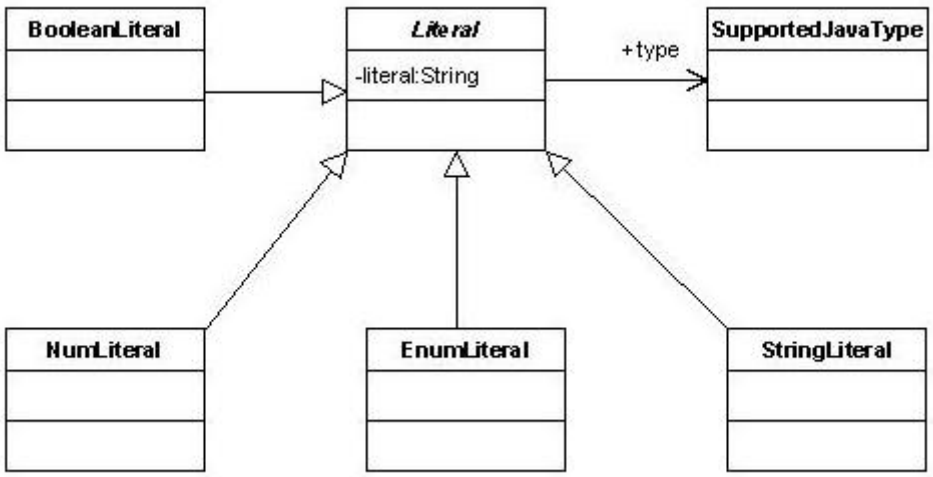



Fig 3.16 Class Diagram of Literal and its subclasses

In Fig 3.16, all functions returning a numeric value are the subclasses of the abstract class **NumFunction**. Although **LengthExp** and **LocateExp** are operating on string

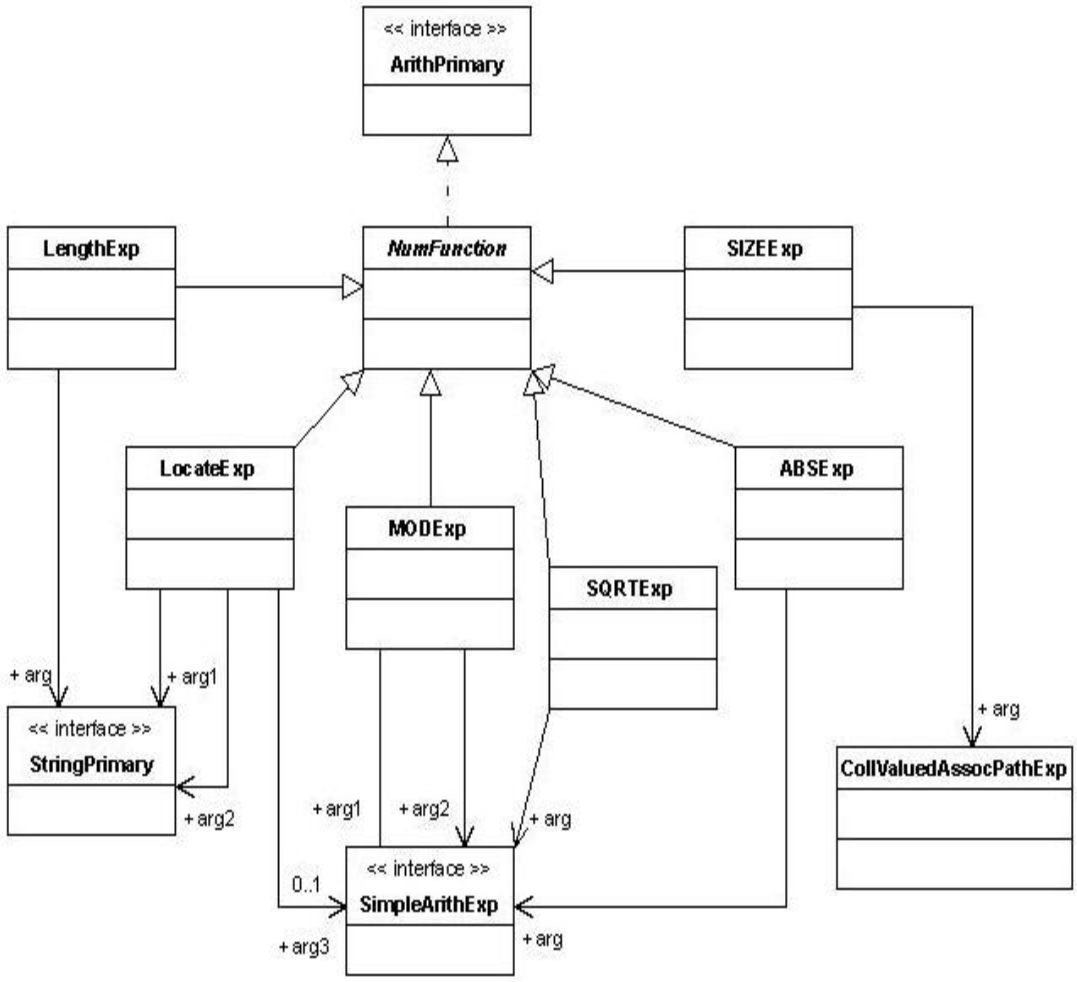



Fig 3.17 Class Diagram of Package ejb3qlmm.basics.functions.num

values, they always return a numeric value and thus are numeric functions. As mentioned a little before, data-typed subqueries could only be used in a few places, such as Like expression, Between expression and comparison expression, therefore, **StringPrimary**, **SimpleArithExp** and other analogous interfaces are necessary. The grammar of those numeric functions is as follows:

```
functions_returning_numerics::=
        LENGTH ( string_primary )   |
        LOCATE (string_primary, string_primary [, simple_arithmetic_expression ] )   |
        ABS ( simple_arithmetic_expression )   |
        SQRT ( simple_arithmetic_expression )   |
        MOD ( simple_arithmetic_expression, simple_arithmetic_expression )   |
        SIZE ( collection_valued_path_expression )
```

There are five functions defined in EJB3QL spec returning a string value, and they all operate on string values. **TrimExp** is used to trim the specified character from a string. If the character to be trimmed is not specified, it is assumed to be space.

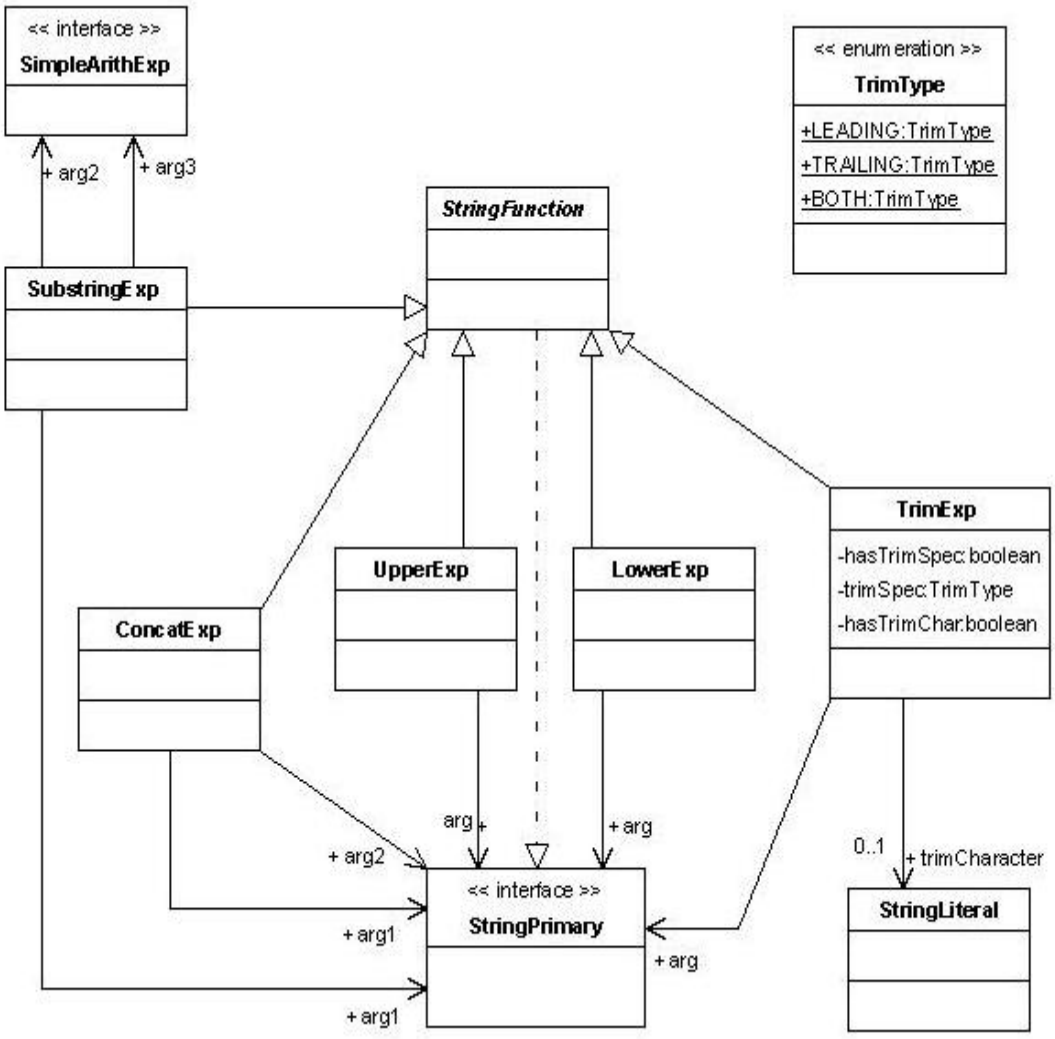



Fig 3.18 Class Diagram of Package ejb3qlmm.basics.functions.string

Because of the pattern of trimming, an enumeration **TrimType** for those trimming pattern is needed, which has three values, If a trim type is not provided, **BOTH** is assumed. The grammar of these string functions is as follows:

```
functions_returning_strings ::=
        CONCAT ( string_primary, string_primary )   |
        SUBSTRING ( string_primary,
                simple_arithmetic_expression, simple_arithmetic_expression )   |
        TRIM ( [ [ trim_specification ] [ trim_character ] FROM ] string_primary )   |
        LOWER ( string_primary )   |
        UPPER ( string_primary )
trim_specification ::=   LEADING   |   TRAILING   |   BOTH
```

In the last section, the type hierarchy of **ArithExp** was already discussed, and besides this hierarchy, Fig 3.19 also shows the associations between them, which are mainly

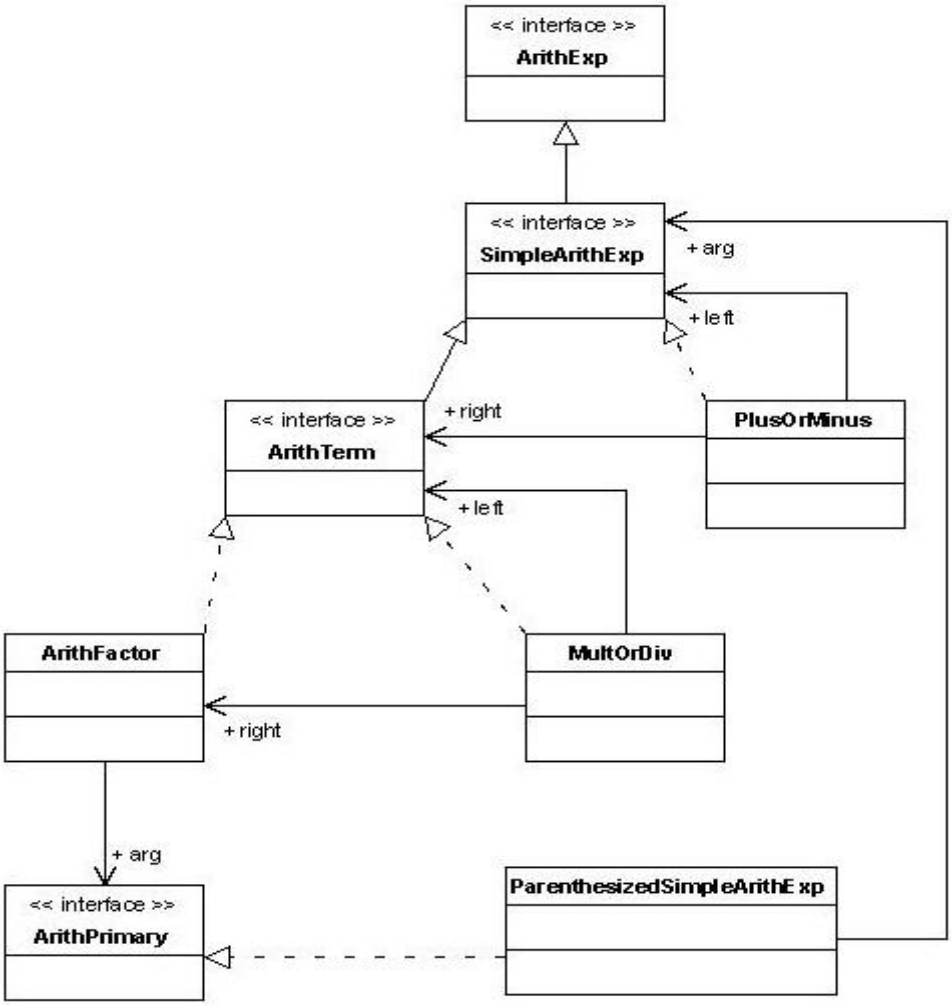



Fig 3.19 Class Diagram of Package ejb3qlmm.basics.arithmeticExp

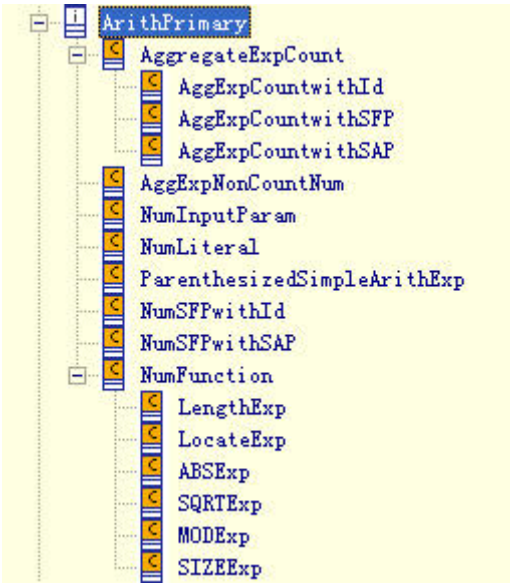



Fig 3.20 Type Hierarchy of ArithPrimary

for arguments of plus/minus or multiplication/division operations. In Fig 3.19, **ArithFactor** has an argument, a generalization of all simple numeric expressions (refer to the last section), which is called **ArithPrimary**. Because the hierarchy of **SimpleArithExp** is focusing on the arithmetic operations (+, -, *, /, etc.) and expressions which we called composite numeric expressions, and all the other simple numeric expressions are used as arguments of those arithmetic operations, **ArithPrimary** is created to generalize simple numeric expressions and separate them from the composite ones. Fig 3.20 gives us its implementations, including aggregate expressions, numeric functions, state field path expressions, literal, and input parameter, etc.

### 3.3.7 Package ejb3qlmm.conditionalExp

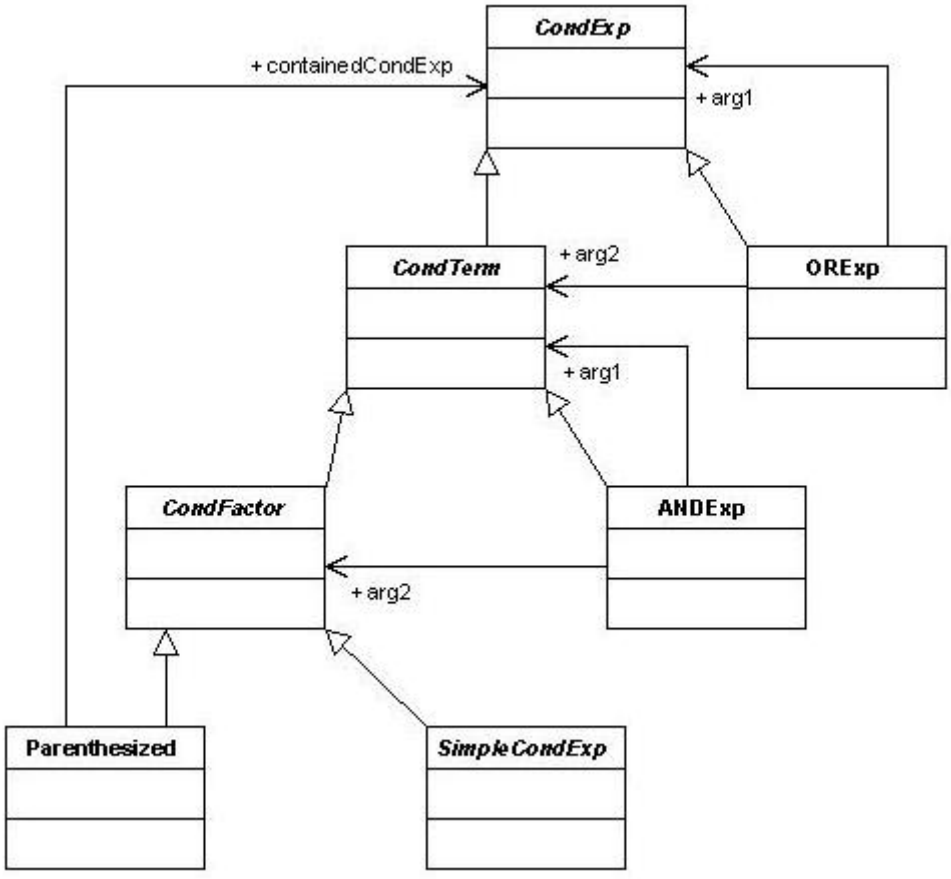



Fig 3.21 Class Diagram of Package ejb3qlmm.conditionalExp (1)

Fig 3.21 shows some conditional expressions including binary and unary logical operations such as AND, OR, NOT, which we call composite boolean expressions, because they could take themselves as arguments. This case is similar to arithmetic expressions, and there are also some simple boolean expressions, which are

generalized as **SimpleCondExp**. The following picture gives all simple boolean expressions and the associations representing their arguments.

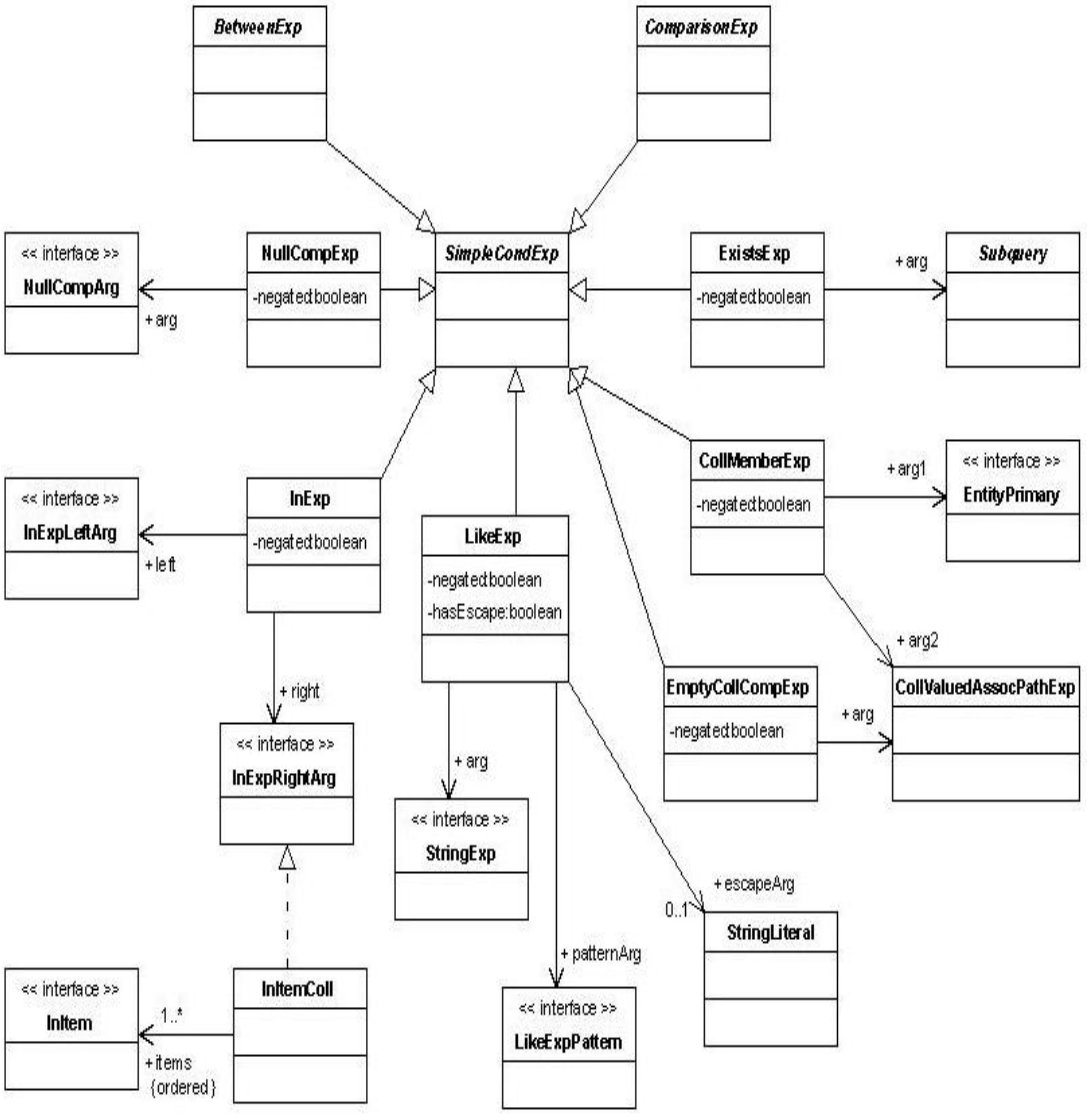



Fig 3.22 Class Diagram of Package ejb3qlmm.conditionalExp (2)

**ExistsExp**, **CollMemberExp**, **EmptyCollCompExp** are all created exactly like what the grammar describes, and not discussed here. **NullCompExp** represents the comparison expression with keyword NULL, whose argument might be of an input parameter or a single-valued path expression, so interface **NullCompArg** is created to be the generalization of them as the possible argument of NULL expression. For **LikeExp**, **LikeExpPattern** represents the pattern value used in a LIKE expression, and could be a string literal or a string-valued input parameter, so it is also a generalization of the two possible pattern values.

The expression with keyword IN deserves more words. Let's first take a look at its grammar:

```
in_expression ::=
        state_field_path_expression [NOT] IN ( in_item {, in_item}*  |  subquery )
in_item ::= literal  |  input_parameter
```

It seems not so complex as what is modeled in the metamodel. But in section *4.6.8 In Expressions* of the JSR-220 specification, we found the following words: "*The state_field_path_expression must have a string, numeric, or enum value.*" That means the left side of an IN expression must be of one of the three data types. According to the semantics of the IN expression, the data type of its left and right sides must match. So the right side should also be of one of the three data types (for in_items, all the items must be of the same data type). Therefore, the interface **InExpLeftArg** is created as a generalization of the state field path expressions for those three data types. **InExpRightArg** and **InItem** are necessary to be created, and this data type problem should also be taken into account, when they are modeled. The following pictures show how these interfaces are modeled:

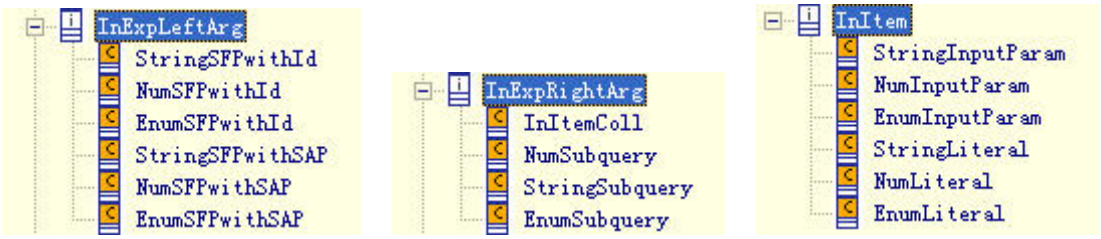



Fig 3.22 Hierarchy of InExp related Interfaces

The problem around these interfaces could be regarded as a well-formedness rule, and modeling with UML only constrains the scope of the data types applicable in this case, the type matching between the left and right sides and the type consistency among in_items are still left to OCL invariants, and will discussed in the next chapter.

The Between expression are straightforward, and has three arguments of the same type, which must be one of numeric, string and datetime. Therefore, **BetweenExp** should be specialized into three subclasses due to different data types, each of which is associated with an interface that is the generalization of a kind of data-typed expressions, such as **ArithExp**.

Comparison expressions are complex more or less, because of the ALL or ANY expressions. Fig 3.23 gives the class diagram of a branch of comparison expression. Similar to **BetweenExp**, due to the data types of arguments, **ComparisonExp** also has subclasses for all data types. Here in Fig 3.23, only the branch for numeric type is shown, and others are similar. The corresponding grammar is as follows:

```
comparison_expression ::=  arithmetic_expression comparison_operator
                                { arithmetic_expression  |  all_or_any_expression }
all_or_any_expression ::= {  ALL  |  ANY  |  SOME } ( subquery )
```

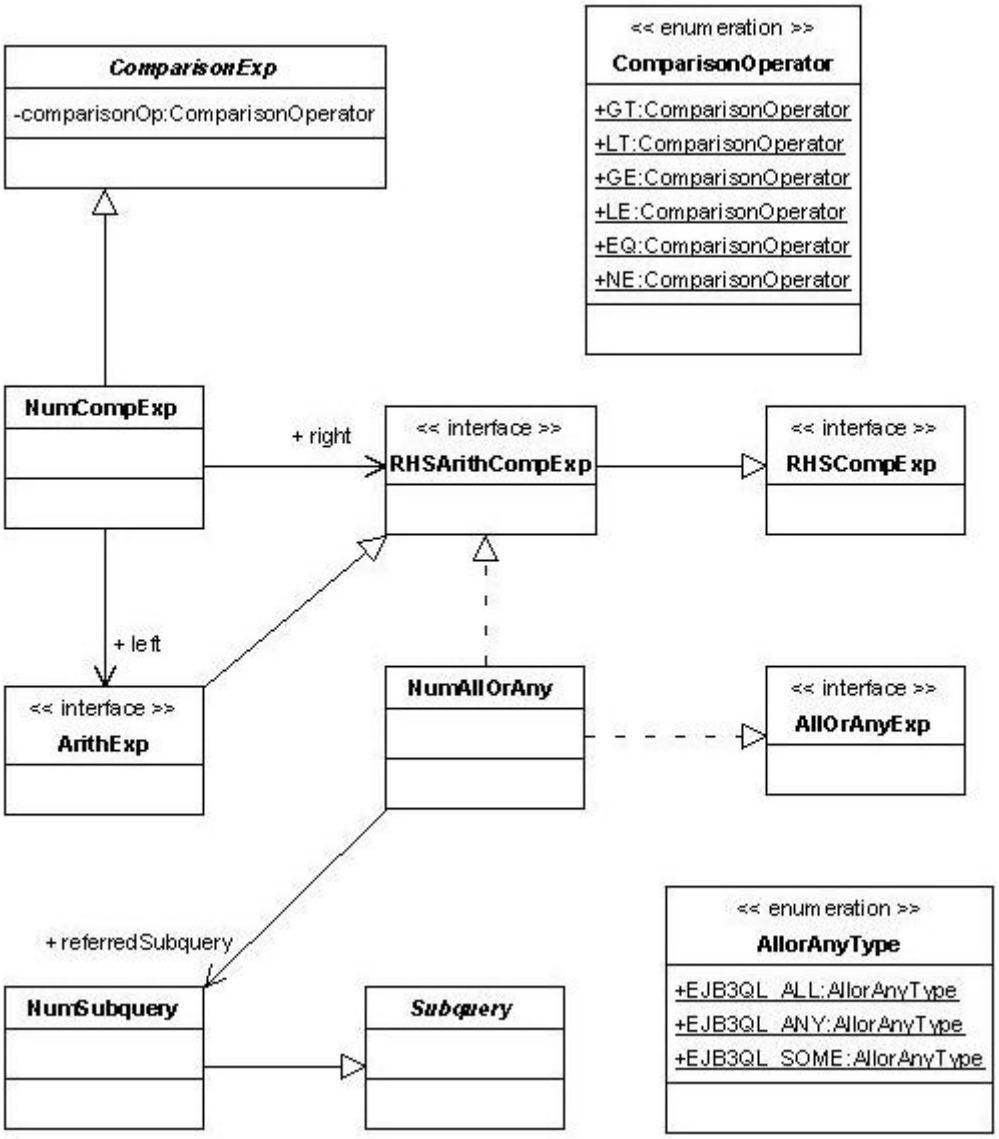



Fig 3.23 Class Diagram for (Numeric) Comparison Expression

According to the grammar, two possible constructs could appear at the right side, an arithmetic expression or an ALLorANY expression with a subquery. So interface **RHSArithCompExp** is created as a generalization of the two possible constructs. As more interfaces like **RHSArithCompExp** are needed for other data types such as **RHSStringCompExp**, interface **RHSCompExp** is built as the root of the type hierarchy of those interfaces. Since the left side is of numeric type, the right side should be of numeric type, too. That means the ALLorANY expression at the right side must be of numeric type, therefore, **AllOrAnyExp** should also data-typed, and then we have class **NumAllOrAny** and other classes for the other data types. **NumAllOrAny** is used at the right side and of numeric type, and obviously, it should implement **RHSArithCompExp**. In fact, the data type of an ALLorANY expression depends on the subquery it refers, and subquery is data-typed, and undoubtedly, **NumAllOrAny** should be associated with **NumSubquery**. The comparison

expressions for other data types are modeled almost in the same way.

### 3.3.8 Package ejb3qlmm.subquery

The following picture is the class diagram of this package. **GroupByItem** is actually declared in package ejb3qlmm.selectStmt, and we will discuss it in that section.

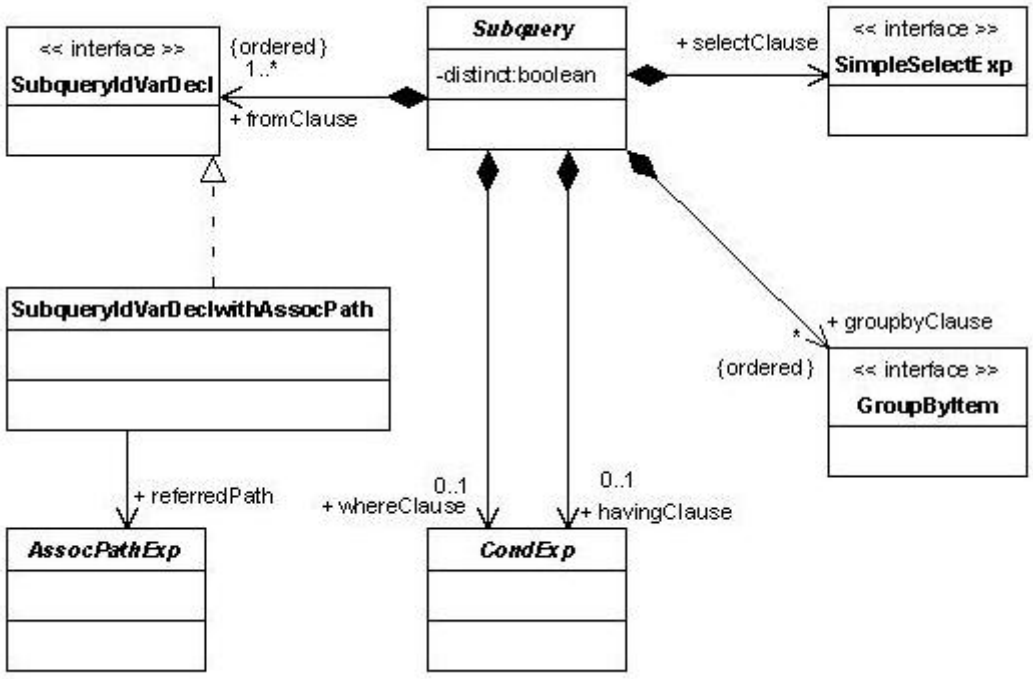



Fig 3.25 Class Diagram of Package ejb3qlmm.subquery

Besides, two interfaces respectively for the FROM clause and SELECT clause deserve some discussion. Although a subquery is similar to a select statement, they are still different at some points, such as the expression that could appear in the SELECT clause and the id variable declaration used in the FROM clause. In the SELECT clause of a subquery, only one expression is allowed to be selected. And the possible expressions are generalized to the interface **SimpleSelectExp**, and the hierarchy of them is as follows:


Fig 3.26 Hierarchy of SimpleSelectExp

Aggregate expreesions, id variables and single-valued path expressions could be used

in the SELECT clause. For the FROM clause, because a simple association path expression with an id variable is allowed to be used as an id variable declaration in a subquery., the class **SubqueryIdVarDeclwithAssocPath** is created. It was already introduced a little in the section for id variable declaration. The grammar and the hierarchy of **SubqueryIdVarDecl** are as follows:

```
subselect_identification_variable_declaration ::=
                        identification_variable_declaration   |
                        association_path_expression [AS] identification_variable   |
                        collection_member_declaration
```

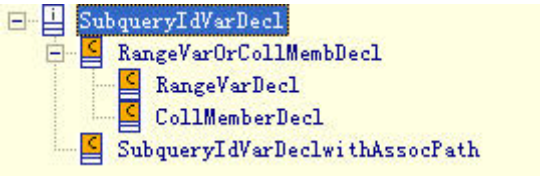



Fig 3.27 Hierarchy of SubqueryIdVarDecl

There is no **JoinDecl** in this hierarchy, because all these constructs could be independently used in the FROM clause, whereas join declarations must be attached to range variable declarations.

As we have mentioned a lot of times that the subquery is used as a function and return some values, so that they are data-typed, now we come to its type hierarchy:

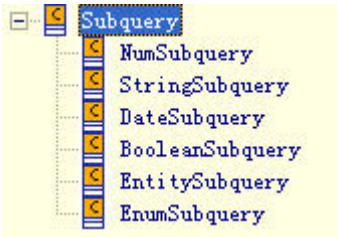



Fig 3.28 Type Hierarchy of Subquery

### 3.3.9 Package ejb3qlmm.selectStmt

Class **SelectStmt** inherits **EJB3QLStmt** in package ejb3qlmm.stmts, which has an association for the WHERE clause, so in this package, **SelectStmt** has no association for its WHERE clause. The HAVING clause is similar to the WHERE clause, as either of them takes a conditional expression (**CondExp**).

For the FROM clause, a sequence of **RangeVarOrCollMembDecl** instances compose it. Different from the FROM clause of a subquery, an association path expression

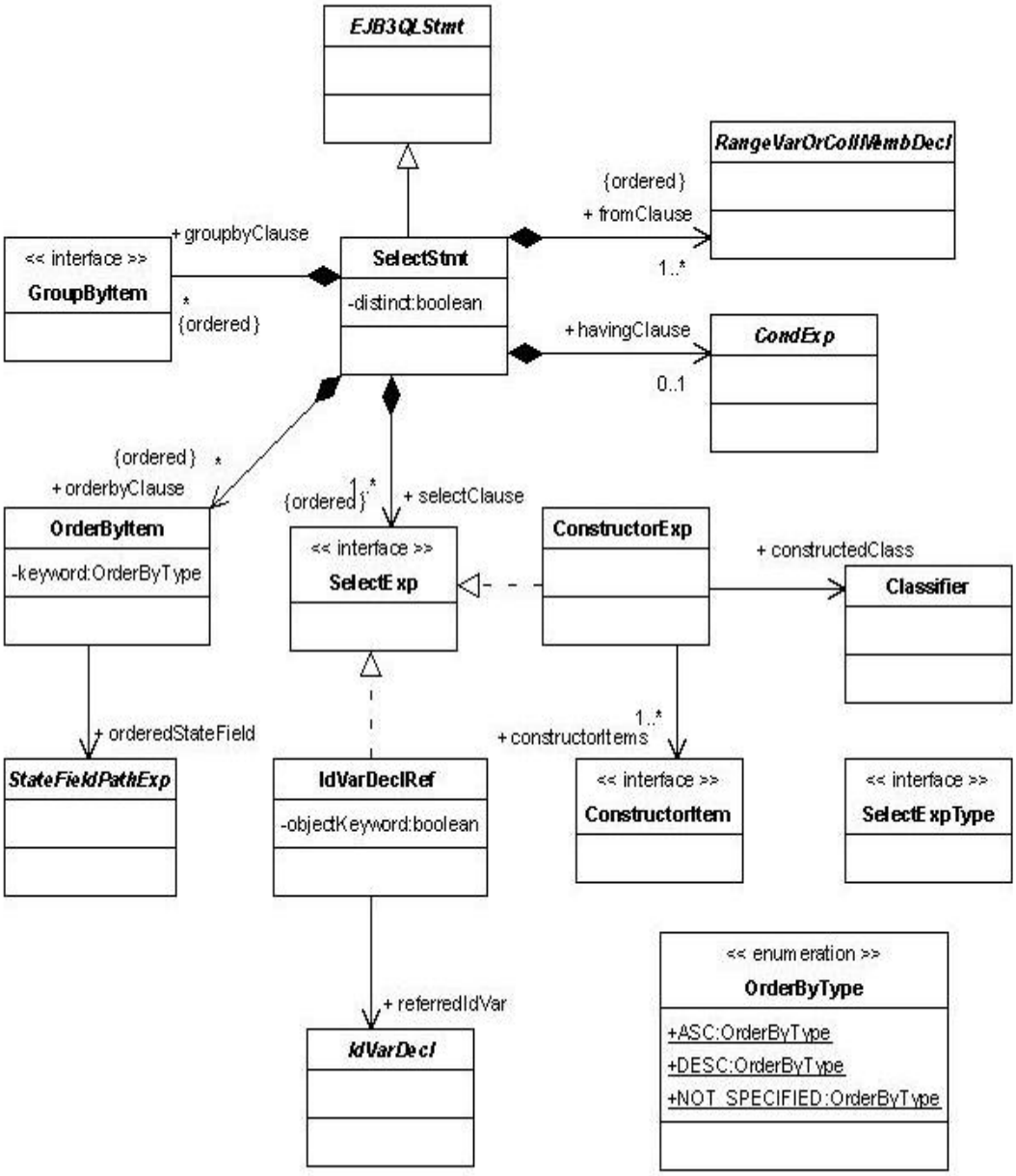



Fig 3.29 Class Diagram of Package ejb3qlmm.selectStmt

with an id variable declared is not allowed to be independently used in the FROM clause of a select statement.

The ORDERBY clause is composed by a sequence of **OrderByItem** instances, each of which must be a state field path expression.

The classes for modeling the SELECT clause in this package might be the most complex, because there are several possible constructs that could be appear in the SELECT clause and the modeling of the SELECT clause involves the data type problem of the select statements.


Fig 3.30 Hierarchy of SelectExp

Fig 3.30 gives the four possible constructs that could be selected in a select statement. When a query is to search some objects of an entity, the corresponding id variable should appear in the SELECT clause, because a keyword OBJECT might be added optionally, **IdVarDeclRef** is created as a wrapper of **IdVarDecl** in this case.

A constructor expression is to return some instances of a certain Java class. The constructor expression acts similarly as a "constructor" of a class in Java language, so it must have a corresponding Java class. And this specified class is not required to be an entity or to be mapped to the database, and the constructor name must be fully qualified. There is an example from the specification:

```
SELECT NEW com.acme.example.CustomerDetails(c.id, c.status, o.count)
FROM Customer c JOIN c.orders o
WHERE o.count > 100
```

In this example, the specified class `com.acme.example.CustomerDetails` must already exist. It is to create some instances of this class with the data that is retrieved on the condition specified in the FROM and WHERE clause. The fields of the class to be instantiated are modeled as **ConstructorItem**, which might be aggregate expressions or single-valued path expressions, as shown in the following picture.

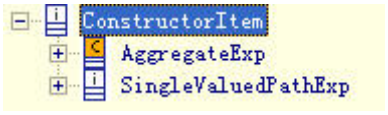



Fig 3.31 Hierarchy of ConstructorItem

As we know, a select statement might contain a sequence of expressions to be searched in the SELECT clause, which means the select statement might return a sequence of values of different data types. In order to get the return types of a select statement for well-formedness goals, interface **SelectExpType** is created to generalize some data types as follows.

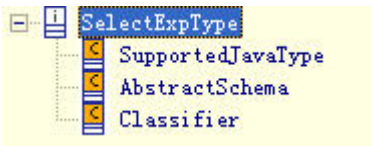



Fig 3.32 Hierarchy of SelectExpType

### 3.3.10 Package ejb3qlmm.stmts

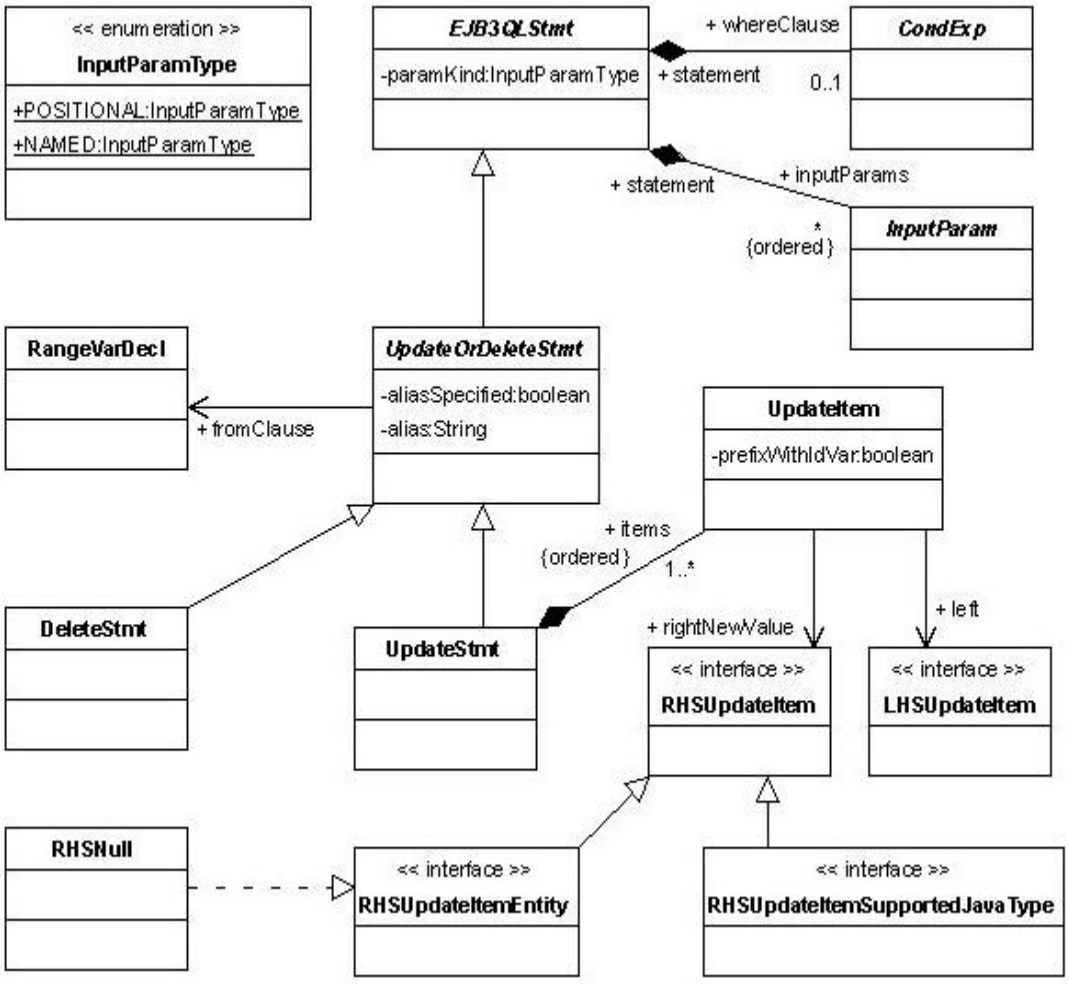



Fig 3.33 Class Diagram of Package ejb3qlmm.stmts

A statement should have all the input parameters declared, which might be used in it. This is related to a well-formedness rule which requires that all input parameter used in a statement be already declared.

Other constructs in this package are straightforward, only except **UpdateItem**. The grammar of update statement and update item is as follows:

```
update_clause ::= UPDATE abstract_schema_name [ [AS] identification_variable ]
                                SET update_item { , update_item }*
update_item ::= [ identification_variable. ] { state_field    |
                                single_valued_association_field }    =    new_value
```

An update item always consists of a equation, which has the left hand side and right

hand side. For the left side, it could be a state field or a single-valued association field, which are generalized to interface **LHSUpdateItem**. For the right side, there are several possibilities: **SimpleEntityExp**, **SimpleArithExp**, **BooleanPrimary**, **StringPrimary**, **DatePrimary**, **EnumPrimary**, and **RHSNull** that represents NULL value. These possible expressions are clearly interfaces of data-typed expression, which could be easily divided into two groups, entity type and primitive data type. The generalization of the two groups' expressions will help compose OCL invariants for the related well-formedness rules, such as type matching of the left side and the right side. The following picture shows the type hierarchy.

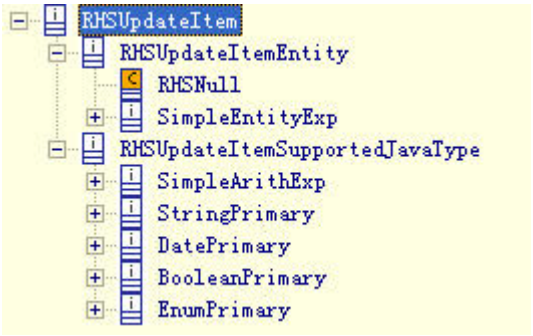



Fig 3.33 Hierarch of RHSUpdateItem

Up to now, we have already detailedly described the metamodel and involved a little around well-formedness, and the data type related discussions and others provide a solid background for the later discussion on well-formedness rules in the next chapter.

# 4. Well-Formedness Rules

From the description in the last chapter, we might find that the metamodel is complex and detailed enough to describe and model EJB3QL queries very well. However, although UML is so powerful to build such a complex model with considering many tiny problems, it is still not able to solve some problems which can not be expressed in the model, for example, the type matching between the two sides of a comparison expression, and chaining between the path sections of a path expression, etc. This kind of inability of UML is just why we need OCL and why OCL could come out with UML 2.0. In this chapter, we will use OCL to make up the well-fomedness related part of EJB3QL metamodel. Firstly, the importance of well-formedness of a metamodel will be reemphasized; then several important kinds of well-formedness rules and their algorithm and designs will be described; then we go to talk about the other well-formedness rules briefly and some other rules which can not be perfectly implemented in a while and might be solved in the future.

## 4.1 Importance of Well-Formedness

When a transformation happens between a source model and a target model, there are two important points deserving our more attention: 1, the mapping rules, or the transformation recipes; 2, the well-formedness rules. Generally, people prefer to talk about and devote much effort to the transformation recipes, which seems the most difficult part in a transformation. Actually, a transformation recipe provides a scheme of what the transformation result would look like and how the source model should be transformed into the target model.

However, the transformation recipes might only guarantee that the source models could be transformed into the target models, but can’t guarantee that the source models could be transformed correctly into the target models, which means the transformation results might be grammatically incorrect, or not well-formed.

For example, we plan to translate OCL invariants into EJB3QL queries. Octopus acts as the parser and AST builder of OCL, and so it can guarantee the OCL AST is well-formed. On the other hand, EJB3QL ASTs should be also well-formed, so that the EJB3QL strings finally generated out of those EJB3QL ASTs could be accordingly correct and the ORM engine could execute them successfully. Therefore, in this chain of transformation, the grammatically correct OCL invariants are parsed into

well-formed OCL ASTs, which is then transformed into well-formed EJB3QL ASTs to generate the query strings that are fed to the ORM engine; that is, each section is correct or well-formed, which leads to the correct final result. Let's suppose that EJB3QL AST is not well-formed, and check an example.

There is a similar feature in both OCL and EJB3QL, object navigation, for example, in the following OCL invariant,

```
context LoyaltyProgram
  inv minServices: self.partners.deliveredServices->size() >= 1
```

`self`, `partners`, `deliveredServices` are all corresponding to some entity instances; and in EJB3QL,

```
SELECT  lp
FROM  LoyaltyProgram  lp
     JOIN  lp.partners  p
     JOIN  p.deliveredServices  s
WHERE  s.serviceNr > 0
```

`lp`, `partners`, `p`, `deliveredServices`, `s` are all corresponding to entity instances. The difference in navigation between OCL and EJB3QL is that OCL allows for the navigation containing association ends with cardinality greater than 1, e.g. partners that results in a collection of `ProgramPartner` and could be further composed to `deliveredServices`; whereas EJB3QL doesn't allow this. In EJB3QL, a navigation involving associations with cardinality greater than 1 can not appear in WHERE clause; and in FROM clause, JOIN is used to solve this problem. Therefore, if this fact was not taken into account when designing EJB3QL metamodel, then a path would consist of several path sections including the ones with cardinality greater than 1 and without using JOIN, which looks like the navigations in OCL, but could not be successfully parsed or executed by the ORM engine, for example, `lp.partners.deliveredServices`.

After a lot of explanation above and in the last chapter, well-formedness is obviously necessary to be considered when we build a metamodel. For EJB3QL metamodel, besides the well-formedness problems involving navigations, others involving data type problems are another important part to be concerned. Actually, well-formedness around data types is already considered in the design of EJB3QL metamodel as we mentioned in the last chapter, but modeling could not completely solve all the data type related well-formedness problems, so in the following sections, firstly those around data types will be further discussed, and then we turn to the others.

## 4.2 Complex Well-Formedness Rules

### 4.2.1 Data Type Related Well-Formedness Rules

The data type related well-formedness mainly involves type ckecking and matching. In order to perform those, the data type of the expressions to be checked should be firstly retrieved. We have defined a bunch of methods, which are named type(), for the classes in EJB3QL metamodel. Their return types are different, and some of them return an instance of **SupportedJavaType**, whereas the others return an instance of **AbstractSchema**. **EmbeddedClass** as a type is only for embedded class fields, which in a state field path expression is finally evaluated to a value of a supported Java type. Since most of the expressions are already data-typed, the return type of their type() methods could be accordingly defined to be **SupportedJavaType** or **AbstractSchema**. For example, an association path expression is always evaluated to an entity instance or a collection of instances of a certain entity, which means it should be of an entity type and its type should be an **AbstractSchema** instance, as the following OCL code shows:

```
context AssocPathExp :: type() : ejb3qlmm::schema::AbstractSchema
 body : self.chainAssocs -> last().type
```

The type of a collection-valued association path expression should be that of its last association field, whereas a single-valued association path expression only has one navigation section in the chain. And a state field path expression always terminates as a state field which should be of a **SupportedJavaType**, even though it is an embedded class field which is just explained above. The type() method body is as follows:

```
context StateField :: type() : ejb3qlmm::schema::SupportedJavaType
 body : self.terminalField.type

context StateFieldPathExp :: type() :
                                  ejb3qlmm::schema::SupportedJavaType
 body : self.stateField.type()
```

The type of a state field path expression depends on its state field. And the type of a state field should be that of its terminal simple state field.

From these two examples, it is clear that the type of an expression is sure to be that of its terminal construct, for example, a state field path expression terminates at a state field and its type is that of the state field. This is semantically correct and applied to

other data-typed constructs. Then we will take a look at some typical type() methods of some constructs in the metamodel.

```
context RangeVarDecl :: type() : ejb3qlmm::schema::AbstractSchema
 body : self.referredEntity

context CollMemberDecl :: type() : ejb3qlmm::schema::AbstractSchema
 body : self.joinedPath.type()

context JoinDecl :: type() : ejb3qlmm::schema::AbstractSchema
 body : self.joinField.type

context SubqueryIdVarDeclwithAssocPath :: type() :
                              ejb3qlmm::schema::AbstractSchema
 body : self.referredPath.type()
```

The type of an id variable declaration is definitely an AbstractSchema, and depends on which kind of construct id refers. For example, the type of a collection member declaration is that of its joined path which is a collection-valued association path expression; and the type of a join declaration is that of its referring join field, etc.

```
context AggregateExpNonCount :: type() :
                              ejb3qlmm::schema::SupportedJavaType
 body : self.aggregatedSFP.type()
```

An aggregate expression with COUNT are sure to return a numeric value, and the type of it, an instance of **SupportedJavaType**, could be any numeric type and dynamically assigned at runtime by the parser or AST builder, so we don't have to define its body. For an aggregate expression with another operator, the type depends on its argument, a state field path expression, because AVG and SUM can not take an argument of non numeric type, which is guaranteed by another invariant.

```
context DateSubquery::type() : SupportedJavaType
 body :
 if (self.selectClause.oclIsKindOf(ejb3qlmm::aggregate::AggExpNonCountDate))
   then self.selectClause.oclAsType(ejb3qlmm::aggregate::AggExpNonCountDate).type()
   else
     if (self.selectClause.oclIsKindOf(ejb3qlmm::pathExp::DateSFPwithId))
       then self.selectClause.oclAsType(ejb3qlmm::pathExp::DateSFPwithId).type()
       else self.selectClause.oclAsType(ejb3qlmm::pathExp::DateSFPwithSAP).type()
     endif
 endif
```

The type() method of a subquery could return the type of its select clause, which contains only one expression.

For functions like **ABSExp**, its return type could be easily determined. And for arithmetic operations (+, -, *, /), determining their return type involves the promotion of Java data types, for example, if a value of double type is multiplied with a value of int type, the result must be of double type, therefore, the type() methods of those arithmetic expressions contain a lot of checking arguments' types and determining the return type, and we won't write them here because of the large amount of the code.

With those type() methods, the data type of the expressions could be retrieved, and then we can perform the type checking and matching.

a) Type Matching

A typical problem lies in the comparison expressions, where the left side argument must be matched with the right side argument. So we have the following invariant:

```
context EntityCompExp
  inv : self.left.type() = self.right.type()

context EnumCompExp
  inv typeCompatibility : self.left.type() = self.right.type()
```

Since the comparison expressions for primitive data types, such as **StringCompExp**, and their arguments, such as **StringExp** and **RHSStringCompExp**, are data-typed, for those primitive data typed expressions, we do not have to check the type matching of their arguments; whereas the comparison expressions for entity or enumeration types must have type matching performed, because within a persistence unit the abstract schema or enumeration might be different, for example, we can not compare an instance of entity "Person" with an instance of entity "Department".

However, for some expressions involving primitive data types, the type matching is still need to be checked, for example, **InExp** (refer to Fig 3.22). Type matching of primitive types is a little more complex than that of entity or enumeration types, because of the numeric type. Notice that we have talked in section 3.3.1 that Java provides a lot of different numeric data types, such as short, int and their wrappers and other numeric types. For other primitive data types, we could just compare them with an equation; for numeric types, we can not do like that, but make sure that two instances of **SupportedJavaType** are both numeric types. So we can check the compatibility of two arbitrary primitive types as follows:

```
context SupportedJavaType::areTypeCompatible( t1 : SupportedJavaType,
                                  t2 : SupportedJavaType) : Boolean
 body: if t1 = t2
       then true
       else
```

```
            let b1 : Boolean = SupportedJavaType::isNumeric(t1) ,
                b2 : Boolean = SupportedJavaType::isNumeric(t2)
            in (b1 and b2)
      endif
```

In class SupportedJavaType, several methods are defined to check whether a given instance belongs to an specified primitive type, such as isNumeric().
Now we come to the type matching of **InExp**. "The state_field_path_expression [i.e. the argument before the IN keyword] must have a string, numeric, or enum value." This well-formedness rule is actually guaranteed by the types in the UML model alone we have discussed with the type hierarchies (Fig 3.22) in the last chapter, as the only classes supporting the interface **InExpLeftArg** are those subclasses of **StateFieldPathExp** referring to a field whose type can only be string, numeric, or enum. According to the type matching, the right side of the IN expression must be also of string, numeric, or enum, no matter it consists of a subquery or a collection of input parameters or literals. The UML model actually restricts the possible data types for IN expressions to the three types, but it is still lack of something to determine the matching between the two sides. Therefore, the following invariant is obviously necessary:

```
context InExp
  inv WFR_4_6_8 :
    ejb3qlmm::schema::SupportedJavaType::areTypeCompatible(
    self.left.type(),
    self.right.type() )
```

Because the right side argument of an IN expression could be a collection of instances of **InItme**, the data type of this collection in this case should be the data type of anyone of its elements. The following method is defined:

```
context InItemColl :: type() : ejb3qlmm::schema::SupportedJavaType
 body : self.items->any(true).type()
```

And here a well-formedness rule is implied, that the elements in the collection must be of the same data type.

```
context InItemColl
  inv allItemsOfTypeCompatibleTypes:
    items->forAll( i1, i2 : ejb3qlmm::conditionalExp::InItem |
       ejb3qlmm::schema::SupportedJavaType::areTypeCompatible(
            i1.type(), i2.type() ) )
```

The invariant above is to iterate over all the items with the collection and check the compatibility of the data type of every two items.

There are also other well-formedness rules on type matching, such as those for some other expressions like **CollMemberExp**, and those for **UpdateItem** which has two sides involving the matching problems, etc., and they could be solved in the same way as what we have explained here.

b) Type Checking

Type checking is to check whether an argument of an expression is of the data type that is expected. For example,

```
context AggregateExpNonCount
  inv WFR_4_8_4_A :
    (keyword = AggregateType::SUM or keyword = AggregateType::AVG)
    implies
 ejb3qlmm::schema::SupportedJavaType::isNumeric(aggregatedSFP.type())
```

According to the specification, the arguments of the aggregate expressions with SUM and AVG must be numeric, therefore, we have to check whether the argument is of a numeric type given either of the two mentioned operators. And the aggregate expressions with other operators also need the type checking on their arguments.

Besides the type checking on arguments of operations, some basic constructs like input parameters and literals need to be checked. For example,

```
context NumLiteral
  inv typeIsNum :
         ejb3qlmm::schema::SupportedJavaType::isNumeric(self.type)
```

```
context NumInputParam
  inv typeIsNum : ejb3qlmm::schema::SupportedJavaType::isNumeric(
       self.type.oclAsType(ejb3qlmm::schema::SupportedJavaType))
```

As we have talked in the last chapter, both literal and input parameter are data-typed, and each data-typed subclass of them contains an instance of **SupportedJavaType** or **AbstractSchema** as its data type. So it is necessary to check whether the data type instance that an input parameter or a literal refers is correctly corresponding to the data type by which it is data-typed. The examples above perform this checking.

The data type related well-formedness rules could be mainly categorized into type matching and type checking that are discussed above with some typical examples, and other rules on type matching and checking could be referred to the OCL files in the metamodel. After discussing the problems around data types, we come to another important issue in the next section, id variables and other id related constructs.

### 4.2.2 Id Variable Related Well-Formedness Rules

An id variable is standing for an instance of an entity and an occurrence of that entity in an EJB3QL query. Every EJB3QL query has to involve at least one entities, so that we can not compose EJB3QL queries without id variables. Given that id variables are indispensable, there are certainly some well-formedness rules about them. The following rule is from section 4.6.2 of the specification:

*All identification variables used in the WHERE or HAVING clause of an EJB QL SELECT or DELETE statement must be declared in the FROM clause. The identification variables used in the WHERE clause of an UPDATE statement must be declared in the UPDATE clause.*

The words above imply that the appearances of id variables are divided into id variables' usages and declarations. The id variables must be declared in the FROM clause (for select statement and delete statement) or UPDATE clause (for update statement); whereas they could be used in the WHERE and HAVING clauses, and actually also in SELECT clause as the object to be selected. And all the id variables that are being used must be already declared.

Based on the discussion above, the algorithm for this well-formedness rule could be like follows:

- Collecting id variable declarations into collections for an EJB3QL query, and each its subqueries no matter how deep they are embedded;
- Checking whether all id variables being used in a given constructs are already declared in the collection of id variable declarations got in the first step;
- Checking whether all id variables in the WHERE and HAVING clauses are already declared in the FROM or UPDATE clause.

Since the id variables declarations and usages could appear in subqueries, we have to import a concept, scope. Every variable has its own scope where it is declared and can be recognized, for example, in a Java class, a variable defined in a method can not be recognized in another method, so its scope is the method where it is declared and if the variable is defined as an attribute of the class, it can be recognized by all the methods within the class which is its scope. That is the same for id variables in EJB3QL. In Fig 4.1, there is a tree rooted at an EJB3QL query, and subqueries are embedded in depth of 2 in the tree. In each node id variables could be declared and used. And the scope concept implies that, for example, the id variables declared in Subquery 1 can not be used in EJB3QL Query and Subquery2, but can be used in Subquery1 itself and Subquery3 embedded in it.

Therefore, for the second step of the algorithm above, "the collection of id variable

declarations" should include all the id variable declarations in the current subquery itself and all its owner queries (if the current is the root query, no owner queries), for example, if the given construct (e.g. an aggregate expression) with the some id variables is used in Subquery3, the collection should include all the id variable declarations in Subquery3, Subquery1 and EJB3QL Query. Here we just call this collection Scope. It is expected that all the id variables in the given construct belongs to that scope where it is used.

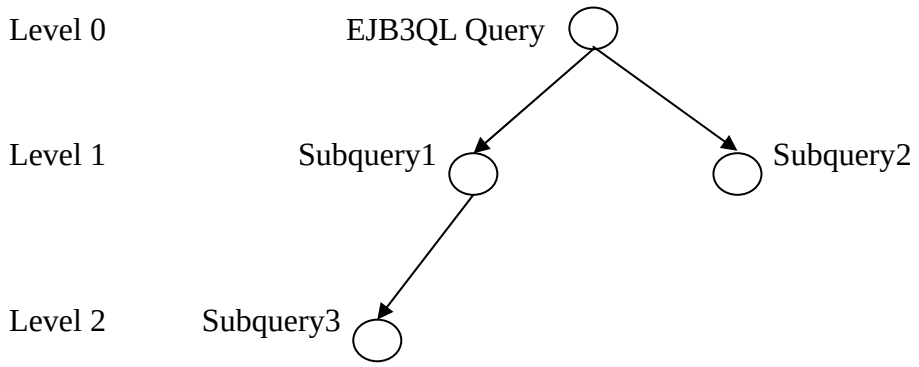



Fig 4.1 Scope of Id Variables

a) Computing Scope

With the concept Scope, we describe the first step of algorithm as computing scope. The algorithms for computing scope of an EJB3QL query and a subquery should be distinguished as follows:

- For an EJB3QL query, just retrieve the id variable declarations in the FROM clause, which might contains joins that form several join trees;
- For a subquery, just combine the id variable declarations within it and those in all its owner queries.

A subquery also has FROM clause, so retrieving id variable declarations from the FROM clause is pivotal, which relies on how those **IdVarDecl** instances could be retrieved from the join trees.

```
context IdVarDecl :: allIdVarDeclaredUnder() : Set(IdVarDecl)
 body : let result : Set(IdVarDecl) = Set{}
        in result -> union(
             self.branches -> iterate(
                  jd : JoinDecl;
                  idVars : Set(IdVarDecl) = Set{} |
                  idVars -> union(jd.allIdVarDeclaredUnder())) )
```

```
            -> including(self)
```

A method `allIdVarDeclaredUnder()` is defined in class **IdVarDecl** to deal with the retrieving. To traverse a tree and meanwhile do some processing, the recursion is needed, and we also adopt it in this method. Every time an instance of **IdVarDecl** is reached, the collection of all the instances of **IdVarDecl** as joins in its subtree are collected and incorporate the current **IdVarDecl** instance itself.

With this method, the id scope of an EJB3QL query and the locally declared id variables in a subquery could be easily retrieved.

```
context SelectStmt :: locallyDeclaredIdVars() :
                                  Set(ejb3qlmm::idVarDecl::IdVarDecl)
 body : self.fromClause -> iterate(
        rvd : ejb3qlmm::idVarDecl::RangeVarOrCollMembDecl;
        idVars : Set(ejb3qlmm::idVarDecl::IdVarDecl) = Set{} |
        idVars->union ( rvd.allIdVarDeclaredUnder() ) )
```

In the FROM clause of a select statement, there might be a sequence of range variable declarations and collection member declarations. Because a collection member declaration has no join branches, its method `allIdVarDeclaredUnder()` only returns itself. This `locallyDeclaredIdVars()` iterates the FROM clause and could return all id variables that appear as collection member declarations, and those as range variable declarations and joins under them.

```
context Subquery :: locallyDeclaredIdVars() : Set(IdVarDecl)
 body : self.fromClause -> iterate(
           sivd : ejb3qlmm::idVarDecl::SubqueryIdVarDecl;
           idVars : Set(ejb3qlmm::idVarDecl::IdVarDecl) = Set{} |
           if sivd.oclIsTypeOf(ejb3qlmm::idVarDecl::RangeVarDecl)
           then idVars -> union (
                  sivd.oclAsType(ejb3qlmm::idVarDecl::RangeVarDecl)
                  .allIdVarDeclaredUnder() )
           else if sivd.oclIsTypeOf(
                        ejb3qlmm::idVarDecl::CollMemberDecl)
                then idVars -> including(sivd.oclAsType(
                          ejb3qlmm::idVarDecl::CollMemberDecl))
                else idVars -> including(sivd.oclAsType(
                ejb3qlmm::subquery::SubqueryIdVarDeclwithAssocPath))
                endif
           endif
        )
```

For method `locallyDeclaredIdVars()` in class **Subquery**, another id variable

declaration must be taken into account, **SubqueryIdVarDeclwithAssocPath**. It is similar to collection member declaration and has no join branches. With this method, it is easy to combine its own id variable declarations and those of its owner queries.

b) Checking Validity

This step needs a lot of methods implemented in OCL files. So an interface **ContainsUsagesOfIdVarDecl** is created in package `ejb3qlmm.idVarDecl` with a method `areAllReferredVarsVisible ( varsInScope : Set(IdVarDecl) ) : Boolean`. This method is to check whether all id variables contained in a given construct are included in its scope. Almost all the constructs that might contain id variables implement this interface, and we need to give the implementations of the method in each class to fulfill this step of the algorithm.

```
context ORExp::areAllReferredVarsVisible ( varsInScope :
                                        Set(IdVarDecl) ) : Boolean
 body : self.arg1.areAllReferredVarsVisible(varsInScope)
        and
        self.arg2.areAllReferredVarsVisible(varsInScope)
```

In the example above, arg1 and arg2 are two arguments of an **ORExp** that represents the binary logical operation OR as a subclass of **CondExp**. To make sure the validity of all id variables the OR expression contains, we must guarantee that all id variables contained in its two arguments are valid. This kind of implementation could be applied to any other constructs that have arguments, for example,

```
context LocateExp :: areAllReferredVarsVisible ( varsInScope :
                     Set(ejb3qlmm::idVarDecl::IdVarDecl) ) : Boolean
 body : self.arg1.areAllReferredVarsVisible(varsInScope)
        and
        self.arg2.areAllReferredVarsVisible(varsInScope)
        and
        if self.arg3 -> notEmpty()
        then self.arg3.areAllReferredVarsVisible(varsInScope)
        else true
        endif
```

**LocateExp** represents the built-in operation LOCATE, and the implementation of this method of it also consists of the checking on each of its arguments.

Most of constructs are similar to the examples above in this case. Now we check the others. Literals and input parameters do not contain any id variables, but as the terminal of an EJB3QL AST they have to implement this method like follows:

```
context Literal :: areAllReferredVarsVisible ( varsInScope :
                    Set(ejb3qlmm::idVarDecl::IdVarDecl) ) : Boolean
 body : true

context InputParam :: areAllReferredVarsVisible ( varsInScope :
                    Set(ejb3qlmm::idVarDecl::IdVarDecl) ) : Boolean
 body : true
```

There are some other construct that directly contains id variables (e.g. **AggExpCountwithId**) or are themselves id variables (e.g. **IdVarDecl** and its subclasses). Let's take a look at the their implementations of the method:

```
context AggExpCountwithId::areAllReferredVarsVisible ( varsInScope :
                                  Set(IdVarDecl) ) : Boolean
 body : varsInScope->includes(aggregatedIdVar)

context AssocPathExp::areAllReferredVarsVisible ( varsInScope :
                                  Set(IdVarDecl) ) : Boolean
 body : varsInScope->includes(self.idVar)

context StateFieldPathExpwithIdVarDecl :: areAllReferredVarsVisible
                    ( varsInScope : Set(IdVarDecl) ) : Boolean
 body : varsInScope->includes(self.idVar)
```

For the path expressions, their starting id variable of the navigation should be valid. And the id variable counted in an aggregate expression should be also valid.

```
context IdVarDecl :: areAllReferredVarsVisible ( varsInScope :
                     Set(ejb3qlmm::idVarDecl::IdVarDecl) ) : Boolean
 body : varsInScope -> includes(self)
```

Sometimes the checking is delegated to id variables themselves, and should be performed as above.

c) Checking Validity of Id Variables in Queries

Up to now we already have most constructs contain and implement the method, and then we can come to the third step to check the validity of id variables in statements and subqueries.

```
context SelectStmt
  inv WFR_4_6_2_A:
      ( not self.whereClause->isEmpty()
        implies
```

```
          self.whereClause.areAllReferredVarsVisible
                    ( self.locallyDeclaredIdVars() )
      ) and (
         not self.havingClause->isEmpty()
         implies
         self.havingClause.areAllReferredVarsVisible
                    ( self.locallyDeclaredIdVars() )
      )
```

The collection of id variables declared in the FROM clause are the scope passed to the method, and the checking is delegated cascadingly from the conditional expressions to their arguments, and further to other expressions, and finally to the terminals that might be id variables, literals, input parameters, etc. The checking in update statement or delete statement is similar to the OCL invariant above.

```
context UpdateOrDeleteStmt
  inv WFR_4_6_2:
     let idvars : Set(ejb3qlmm::idVarDecl::IdVarDecl) = Set{}
     in
         not self.whereClause->isEmpty()
         implies
         self.whereClause.areAllReferredVarsVisible(
            idvars -> including(self.fromClause) )
```

Here the `fromClause` represents the FROM clause in delete statement and the UPDATE clause in update statement, and is only one range variable declaration without any joins.

For subqueries, actually the checking is not an OCL invariant but an implementation of that method. We didn't talk about it in the second part but in this part just because it is similar to and more complex than the checking in the select statement.

```
context Subquery :: areAllReferredVarsVisible ( varsInScope :
                    Set(ejb3qlmm::idVarDecl::IdVarDecl) ) : Boolean
 body : let newVarsInScope : Set(ejb3qlmm::idVarDecl::IdVarDecl)
             = varsInScope -> union(self.locallyDeclaredIdVars())
        in
       ( not self.whereClause->isEmpty()
         implies
         self.whereClause.areAllReferredVarsVisible(newVarsInScope)
       )
       and
       ( not self.havingClause->isEmpty()
         implies
```

```
        self.havingClause.areAllReferredVarsVisible(newVarsInScope)
      )
```

What deserves our attention is the scope for the checking in the example above. The OCL operation `union()` is used to combine the locally declared id variables of a subquery and the scope of its direct owner query that might be an EJB3QL query or another subquery; if the direct owner query is a subquery, this subquery's scope is also computed with `union()` to combine its local id variables and the scope of its direct owner query; finally the local id variables of the top EJB3QL query are incorporated. This is a down-to-top recursive combination of id variables in the subquery tree (refer to Fig 4.1). And the checking in a subquery is of course based on the computed scope.

Besides checking the validity of id variables, the names of the id variables and other ids should be constrained. The specification requires that an id's name can not be the same as any reserved identifier's name (keywords in EJB3QL), and an id variable's name can not be the same as any reserved identifier's name or any entity's name within a persistence unit. So we compose the following OCL invariant.

```
context IdVarDecl
 inv WFR_4_4_2 : let reservedIds : Set(String) =
          Set{ 'SELECT', 'FROM', 'WHERE', 'UPDATE', 'DELETE', 'JOIN',
               'OUTER', 'INNER', 'LEFT', 'GROUP', 'BY', 'HAVING',
               'FETCH', 'DISTINCT', 'OBJECT', 'NULL', 'TRUE', 'FALSE',
               'NOT', 'AND', 'OR', 'BETWEEN', 'LIKE', 'IN', 'AS',
               'UNKNOWN', 'EMPTY', 'MEMBER', 'OF', 'IS', 'AVG', 'MAX',
               'MIN', 'SUM', 'COUNT', 'ORDER', 'ASC', 'DESC', 'MOD',
               'UPPER', 'LOWER', 'TRIM', 'POSITION',
               'CHARACTER_LENGTH', 'CHAR_LENGTH', 'BIT_LENGTH',
               'CURRENT_TIME', 'CURRENT_DATE', 'CURRENT_TIMESTAMP',
               'NEW', 'EXISTS', 'ALL', 'ANY', 'SOME'  },
         entityNames : Set(String) =
                 ejb3qlmm::schema::Entity::allInstances() -> iterate(
                       et : ejb3qlmm::schema::Entity;
                       names : Set(String) = Set{} |
                       names -> including(et.name.toUpper()) )
      in  reservedIds -> excludes(self.alias.toUpper())
          and
          entityNames -> excludes(self.alias.toUpper())
```

The collection of keywords of EJB3QL and the collection of all entity names should exclude a given id variable's name (alias). The part for checking an id's name against keywords could be applied to input parameters which might have names. The uniqueness of id variables' names is not mentioned in the specification, so we do

not whether duplication of id variable's names is allowed.

### 4.2.3 Well-Formedness Rules on Chaining

A path expression consists of a sequence of path sections, which form a chain of end-to-end joins. Actually those path sections are join fields (association ends). So there is a problem of chaining: each association end is actually standing for its entity, e.g. in the EJB3QL query example in section 3.3.5, there is a path `lp.partners` where `lp` is an id variable of abstract schema **LoyaltyProgram** and partners is a join field standing for abstract schema **ProgramPartner** that is to be joined. Notice that here is a precondition that there exist a join between two entities and a join field in the preceding entity to stand for the latter entity, e.g. **LoyaltyProgram** contains join field `parterners` for **ProgramPartner**. This precondition is what we call the rule on chaining that must be satisfied when a path happens. Therefore, OCL invariants should be composed to implement this rule.

```
context AssocPathExp
  inv : let pathLength : Integer = self.chainAssocs -> size()
        in self.chainAssocs -> forAll(
                        f : ejb3qlmm::schema::AssocField  |
                  if self.chainAssocs -> indexOf(f) > 1
                  then  let predec : Integer
                           = (self.chainAssocs -> indexOf(f))-1
                        in   self.chainAssocs ->
                               at(predec).type.associationFields
                                     -> includes(f)
                  else true
                  endif
               )
```

In this invariant `forAll()` is used to iterate over all sections of a path expression to check whether each path section (join field) is visible at the type (abstract schema) of its predecessor. Here the visibility is what is discussed above as the precondition. Since a path expression always starts with an id variable that stands for an entity, this rule should also be applied to it and the first section of the path expression as follows.

```
context AssocPathExp
  inv : self.idVar.type().oclAsType(ejb3qlmm::schema::AbstractSchema)
          .associationFields -> includes( self.chainAssocs -> first() )
```

Here we must complement another prerequisite for the object navigation. Different from that in OCL, the object navigation in EJB3QL requires that all the navigation

sections (association ends) except the last one in an association path expression must be single valued. For example, the path expression `lp.partners` can't be further navigated, because the join field `partners` is a collection-valued association end. The following OCL invariant is to check every join field except the last one that it is an instance of **SingleValuedAssocField**.

```
context AssocPathExp
  inv : let pathLength : Integer = self.chainAssocs -> size()
        in self.chainAssocs -> forAll(
               f : ejb3qlmm::schema::AssocField |
               if self.chainAssocs -> indexOf(f) < pathLength
               then f.oclIsTypeOf(
                       ejb3qlmm::schema::SingleValuedAssocField)
               else true
               endif
           )
```

Not only should path expressions meet this condition, but state fields should satisfy the chaining rule, because a state field might contain a sequence of embedded class fields and only one simple field. All the embedded class fields in a state field should be in a chain. So an OCL invariant like follows is also needed.

```
context StateField
  inv : let pathLength : Integer = self.chainEmbedded -> size()
        in if self.chainEmbedded -> notEmpty()
           then self.chainEmbedded -> forAll(
                   f : ejb3qlmm::schema::EmbeddedClassField |
                   if self.chainEmbedded -> indexOf(f) > 1
                   then let predec : Integer
                            = (self.chainEmbedded -> indexOf(f))-1
                        in self.chainEmbedded ->
                             at(predec).type.embeddedClassFields
                                  -> includes(f)
                   else true
                   endif
                )
           else true
           endif
```

If there exist at least one embedded class fields, the simple state field as the terminal should be declared in the type entity of the last embedded class field in that sequence.

```
context StateField
  inv : if self.chainEmbedded -> notEmpty()
```

```
        then self.chainEmbedded -> last().type.simpleStateFields
            -> includes(self.terminalField)
        else true
        endif
```

It's time to turn to a missing construct, state field path expressions. There are two kind of state field path expressions in our metamodel, those starting with an id variable and with a single-valued association path expression. For a state field path expression starting with a single valued path expression, its state field should be visible at the type of its last association field: if the state field has a sequence of embedded class fields, the first one should be visible at the type of its last association field; if the state field has no embedded class field, the simple state field should be visible at the type of its last association field.

```
context StateFieldPathExpwithSingleValuedAssocPathExp
  inv : let sf : ejb3qlmm::pathExp::StateField = self.stateField
        in if sf.chainEmbedded -> isEmpty()
           then self.path.type().oclAsType(ejb3qlmm::schema::Entity)
                   .simpleStateFields -> includes(sf.terminalField)
           else self.path.type().oclAsType(ejb3qlmm::schema::Entity)
                 .embeddedClassFields ->
                 includes( sf.chainEmbedded -> first() )
           endif
```

The invariant for state field path expressions starting with an id variable is similar to but simpler than the one above, and we do not talk about it here.

Actually join declarations in the FROM clause also form a chain. Therefore, the join field of any join declaration should be visible at the type of its base that might be a range variable declaration or a join declaration.

```
context IdVarDecl
inv : self.branches->forAll( b |
           self.type().associationFields-> includes(b.joinField) )
```

### 4.2.4 Other Important Well-Formedness Rules

According to the specification, there are a lot of well-formedness rules to be considered in our metamodel. And up to now, we have covered most of them by grouping and explaining the typical ones, and the other rules of them are omitted. Here we have to talk about some rules not in the three groups in the last sections.

The GROUPBY clause and HAVING clause deserve some discussion. Usually, the HAVING clause is regarded as a filter condition on the items in the GROUPBY clause, and it could not be independently used without the existing of the GROUPBY clause in the same query. But this is wrong. The existing of the HAVING clause doesn't depend on that of the GROUPBY clause. Due to the specification, if there is no GROUPBY clause and the HAVING clause is used, the result is treated as a single group, and the select list can only consist of aggregate functions. The OCL invariant for this rule is as follows:

```
context SelectStmt
 inv WFR_4_7_D: if self.groupbyClause -> isEmpty()
                    and
                    self.havingClause -> notEmpty()
                then self.selectClause -> forAll(sc :
                          ejb3qlmm::selectStmt::SelectExp |
                          sc.oclIsKindOf(
                             ejb3qlmm::aggregate::AggregateExp) )
                else true
                endif
```

We have another rule about the GROUPBY clause. "*Grouping by an entity is permitted. In this case, the entity must contain no serialized state fields or lob-valued state fields.*" The following invariant is to get the abstract schemas, the id variables of which appear as groupby items, and to check each of those contains no LOB fields.

```
context SelectStmt
  inv WFR_4_7_B:
    let entitiesUsedAsGroupBy : Set(ejb3qlmm::schema::AbstractSchema)
        = groupbyClause->iterate(
               gbi : ejb3qlmm::selectStmt::GroupByItem ;
               r : Set(ejb3qlmm::schema::AbstractSchema) = Set{} |
              if gbi.oclIsKindOf(ejb3qlmm::idVarDecl::IdVarDecl)
              then r->including(gbi
                   .oclAsType(ejb3qlmm::idVarDecl::IdVarDecl).type())
              else r
              endif
          )
    in entitiesUsedAsGroupBy->forAll(
           as : ejb3qlmm::schema::AbstractSchema |
            as.lobFields->isEmpty() )
```

The fields whose values are stored as LOBs are not typed as **SupportedJavaType**, and are not typed as **Entity** (not as **AbstractSchema** and not as **EmbeddedClass**).

And we can identify them because their field `type` returns true for the OCL operation `oclIsUndefined()`. Notice that `self.type->isEmpty()` for those fields returns false because at runtime Java evaluates `{ null }->isEmpty()` which is not empty, the collection has one item, null .

There are also some problems around uniqueness of entities within a persistence unit. Entity names are scoped within the persistence unit and must be unique within the persistence unit. And in case an UML model is accessible, no two different entities may be persisted to the same Java class. And then we have the following two invariants:

```
context PersistenceUnit
  inv WFR_4_3_1 : self.entities -> isUnique(name)

context PersistenceUnit
  inv differentEntitiesAreDifferentUMLClasses :
        self.entities -> isUnique(base)
```

Of course, there are still some more well-formedness rules implemented as OCL invariants, but it's not necessary to explain each of them detailedly, because they are straightforward to understand with cross checking against the specification.

In this chapter, we emphasize again the importance of well-formedness in metamodels, such as our EJB3QL metamodel. And we explain and discuss several complex and important well-formedness rules of EJB3QL and their OCL implementations and algorithms for our metamodel.

# 5. Conclusion

As a typical DSL, EJB3QL is deeply described and discussed in this report. And we build a well-formed EJB3QL metamodel which might be the first well-formed metamodel for EJB3QL over the MDA world, as we have not found any existing one. In this final chapter, some further points around the metamodel will be discussed, and some work to do in the future will be briefly mentioned.

## 5.1 Incomplete Well-Formedness Rules

Although our metamodel could be almost considered well-formed, there are still only a few rules which are not implemented by OCL invariants. The incompleteness is mainly caused by two reasons:

a) Non-specifying in the specification. For example, if two id variables are declared with the same name respectively in a scope and its sub scope, and these two id variables appear in the same scope, the specification does not clarify whether this is allowed or the one declared in the owner scope would be hidden. Another example also deserves consideration. As we discussed the well-formedness rules for data type matching, e.g. the left side's and the right side's expressions of a comparison expression should be of the same data type, there exists a possibility that the two expressions are both entity expressions and should be of the same abstract schema type, but one is of an abstract schema type which is the subclass of the others' abstract schema type. The specification is silent for the inheritance appearing EJB3QL queries. So we don't know whether this is treated as an error or not.

b) Hardness in implementing those with OCL. When one of the arguments of an expression in EJB3QL is null, the value of this expression is unknown. For example, in a LIKE expression, anyone of its three arguments is null, the value of this expression is unknown; if an input parameter's value is null, comparison operations or arithmetic operations involving the input parameter will return an unknown value; If the value of a state field path expression in an IN or NOT IN expression is null or unknown, the value of the expression is unknown. The three cases just mentioned are clearly indicated in the specification. But this is difficult to be expressed by OCL invariants. Although there is OclAny type which is the super type of any other types in OCL, OCL does not have a type for

null, and the OclAny type does not mean the so called “unknown”. Of course, we can just write an invariant to constrain that those argument that might incur “unknown” value can not be null, i.e. not empty.

## 5.2 Further Exploration of EJB3QL and this Metamodel

What we desire after creating EJB3QL metamodel is to build an AST which is a bunch of instances of the metamodel classes, for a given EJB3QL query. This is a process from plain text to AST. A parser is obviously necessary to parse the given query in plain text into a tree. There are some alternative tools to generate such a parser for EJB3QL from its grammar. For example, ANTLR is a popular parser generator which requires that the grammar of the target language, EJB3QL in this case, should be described in a .g file that is recognized by ANTLR as an EBNF grammar with some scripts.

Fortunately, there already exists an EJB3QL grammar described in a .g file, which is provided by GlassFish project. The GlassFish project is an open source application server, through which Application Server Platform Edition 9 is developed and developers could access the source code of the Application Server and contribute to its development, implementing Java EE 5 specifications, including JSR-220. Of course, GlassFish supports EJB3QL. However, the comments on the Glassfish’s EJB3QL grammar don’t claim that it’s complete at this point, but they also don't mention where it is incomplete; actually, it contains a lot of constructs incomplete and some errors. Another big open source project for Java Persistence API is OpenJPA, which offer a JavaCC grammar file for JPQL (actually an extension of EJB3QL, refer to Chapter 2).

Although the application servers utilize the parser generator to get a parser for JPQL/EJB3QL, some well-formedness rules are still not taken into account, because the parser is generated only from the EBNF grammar, which could imply a few well-formedness rules but not all, whereas the other well-formedness rules are described in the specification’s text. In a certain community, some discussion on whether a few EJB3QL queries could be executed was going on, as someone found that those EJB3QL queries were executed successfully in several application servers, but not on some others. So this is a place that a well-formed EJB3QL metamodel should be used.

With a well-formed EJB3QL metamodel, a syntax and well-formedness checking before executing an EJB3QL query could be performed. And if a translation involving EJB3QL is carried out no matter in which direction, the well-formedness is

guaranteed on the step of EJB3QL and its successor.

Besides using this metamodel like the above words, some other processing could be done. For example, this metamodel described in UML could be transformed to an EMF model in an emfatic file, from which EMF related artifacts, such as a tree based EJB3QL editor as an Eclipse plug-in, could be generated. And also with a Rational tool, good looking class diagrams of those packages in the metamodel could be generated.

## 5.3 Concluding Words

We already have such a well-formed EJB3QL metamodel, and could apply it to some usages, such as EJB3QL queries' editing, checking, etc. And this metamodel constitutes a base for further improving the functionalities of Octopus with Xinhua Gu's work to check the integrity against the persisted domain model. And with a tree based EJB3QL editor plug-in generated from the EMF version of this metamodel, well-formed queries could be easily composed. And also an Eclipse text editor might be generated in the future. Although as a case study, we still have a lot of work to do with EJB3QL in the future, all those prove that a well-formed metamodel is the key task for adding support for a DSL to Eclipse or its plug-ins like Octopus.

# References


[1] Improving Developer Productivity with Lightweight Domain Specific Modeling
*by Patrik Nordwall*
http://www.theserverside.com/tt/articles/article.tss?l=LightweightModeling

[2] DSL: Organizing Business Rules based on Rule System
by Anders Lin
http://yimlin.cnblogs.com/archive/2006/06/30/439480.html

[3] DSL: How to get it
by Anders Lin
http://www.cnblogs.com/yimlin/archive/2006/07/08/445673.html

[4] DSM: Domain Specific Modeling
http://www.jdon.com/mda/dsm.html

[5] JSR-220: Enterprise JavaBeans, Version 3.0, Java Persistence API, Final Release
*by EJB3.0 Expert Group May 2, 2006*
*Specification Lead: Linda DeMichiel Sun Microsystems,*
*Michael Keith Oracle Corporation*

[6] POJO Application Framework: Comparison between Spring and EJB3.0
*by Michael Juntao Yuan, Lory Liu*
http://www.matrix.org.cn/resource/article/43/43718_Spring_EJB.html

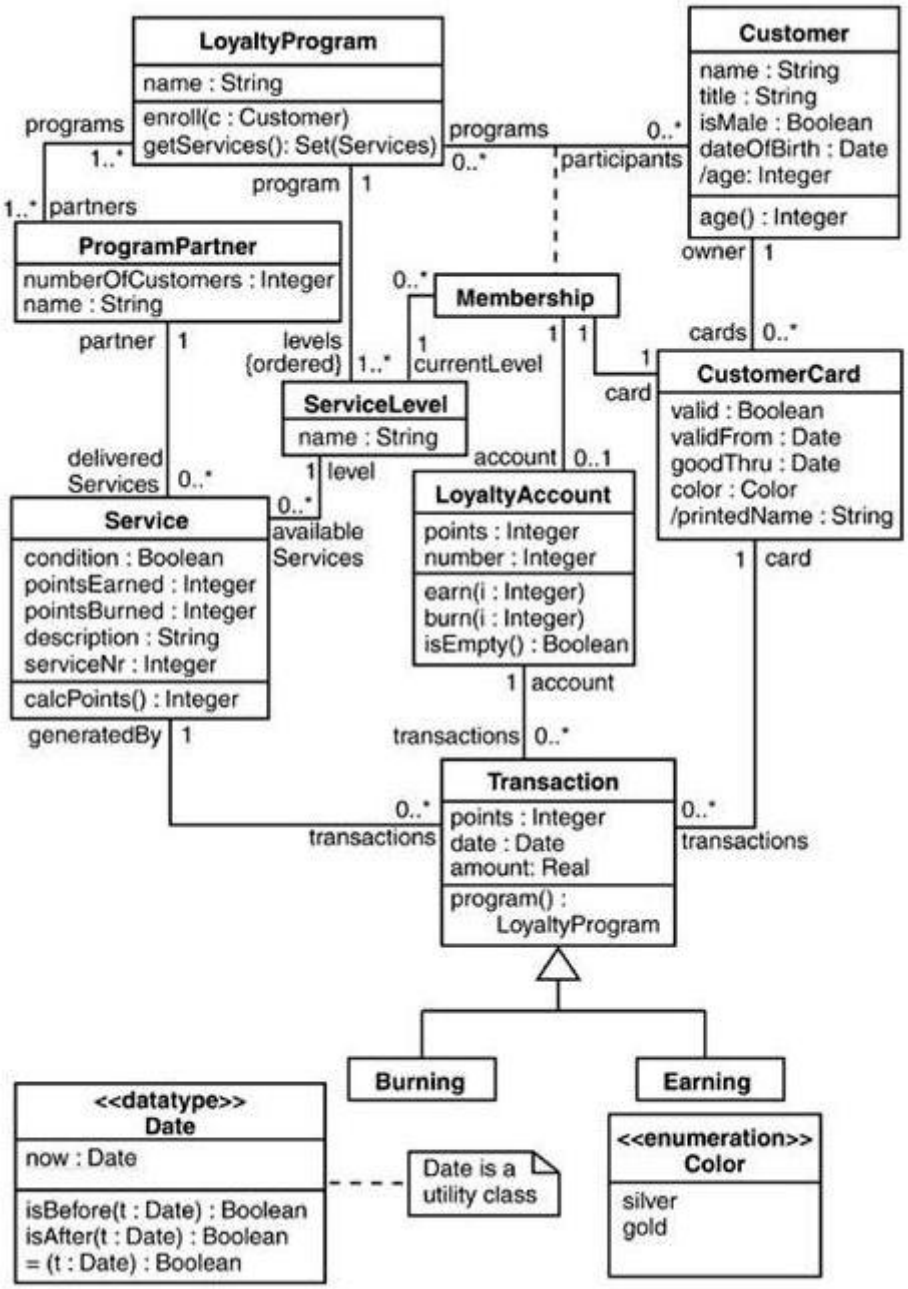
[7] EJB3.0 and Clamorous TSS Annual Meeting by Chenyang Peng
http://www.jdon.com/artichect/EJB3.0.htm

[8] http://www.jspcn.net/htmlnews/11453812900311571.html

[9] Standardizing Java Persistence with the EJB3 Java Persistence API
*by Debu Panda*
http://www.onjava.com/pub/a/onjava/2006/05/17/standardizing-with-ejb3-java-persistence-api.html

[10] Object Constraint Language, The: Getting Your Models Ready for MDA, Second Edition
*by Jos Warmer, Anneke Kleppe August 29, 2003*

# Appendix I RandL Model



# Appendix II EJB3QL Grammar

*QL_statement ::= select_statement | update_statement | delete_statement*
*select_statement ::= select_clause from_clause [where_clause] [groupby_clause]*
*[having_clause] [orderby_clause]*
*update_statement ::= update_clause [where_clause]*
*delete_statement ::= delete_clause [where_clause]*
*from_clause ::=* ***FROM*** *identification_variable_declaration*
*{****,*** *{identification_variable_declaration | collection_member_declaration}}**
*identification_variable_declaration ::= range_variable_declaration { join | fetch_join }**
*range_variable_declaration ::= abstract_schema_name [****AS****] identification_variable*
*join ::= join_spec join_association_path_expression [****AS****] identification_variable*
*fetch_join ::= join_spec* ***FETCH*** *join_association_path_expression*
*association_path_expression ::=*
*collection_valued_path_expression | single_valued_association_path_expression*
*join_spec::= [* ***LEFT*** *[****OUTER****] |* ***INNER*** *]* ***JOIN***
*join_association_path_expression ::= join_collection_valued_path_expression |*
*join_single_valued_association_path_expression*
*join_collection_valued_path_expression::=*
*identification_variable****.****collection_valued_association_field*
*join_single_valued_association_path_expression::=*
*identification_variable****.****single_valued_association_field*
*collection_member_declaration ::=*
***IN (****collection_valued_path_expression****)*** *[****AS****] identification_variable*
*single_valued_path_expression ::=*
*state_field_path_expression | single_valued_association_path_expression*
*state_field_path_expression ::=*
*{identification_variable | single_valued_association_path_expression}.state_field*
*single_valued_association_path_expression ::=*
*identification_variable.{single_valued_association_field.}*single_valued_association_field*
*collection_valued_path_expression ::= identification_variable****.***
*{single_valued_association_field.}*collection_valued_association_field*
*state_field ::= {embedded_class_state_field.}*simple_state_field*
*update_clause ::=* ***UPDATE*** *abstract_schema_name [[****AS****] identification_variable]*
***SET*** *update_item {, update_item}**
*update_item ::= [identification_variable.]*
*{state_field | single_valued_association_field}* **=** *new_value*
*new_value ::= simple_arithmetic_expression |*
*string_primary |*
*datetime_primary |*
*boolean_primary |*

*enum_primary*
*simple_entity_expression* |
***NULL***

*delete_clause ::=* ***DELETE FROM*** *abstract_schema_name [[**AS**] identification_variable]*

*select_clause ::=* ***SELECT*** *[**DISTINCT**] select_expression {, select_expression}**

*select_expression ::= single_valued_path_expression* |
*aggregate_expression* |
*identification_variable* |
***OBJECT(****identification_variable**)* |
*constructor_expression*

*constructor_expression ::=* ***NEW*** *constructor_name* ***(*** *constructor_item {, constructor_item}** ***)***

*constructor_item ::= single_valued_path_expression* | *aggregate_expression*

*aggregate_expression ::=*
*{* ***AVG*** | ***MAX*** | ***MIN*** | ***SUM*** *}* ***(****[**DISTINCT**] state_field_path_expression**)*** |
***COUNT (****[**DISTINCT**] identification_variable* | *state_field_path_expression* |
*single_valued_association_path_expression**)***

*where_clause ::=* ***WHERE*** *conditional_expression*

*groupby_clause ::=* ***GROUP BY*** *groupby_item {, groupby_item}**

*groupby_item ::= single_valued_path_expression* | *identification_variable*

*having_clause ::=* ***HAVING*** *conditional_expression*

*orderby_clause ::=* ***ORDER BY*** *orderby_item {**,** orderby_item}**

*orderby_item ::= state_field_path_expression [* ***ASC*** | ***DESC*** *]*

*subquery ::= simple_select_clause subquery_from_clause [where_clause]*
*[groupby_clause] [having_clause]*

*subquery_from_clause ::=* ***FROM*** *subselect_identification_variable_declaration*
*{, subselect_identification_variable_declaration}**

*subselect_identification_variable_declaration ::=*
*identification_variable_declaration* |
*association_path_expression [**AS**] identification_variable* |
*collection_member_declaration*

*simple_select_clause ::=* ***SELECT*** *[**DISTINCT**] simple_select_expression*

*simple_select_expression::= single_valued_path_expression* |
*aggregate_expression* |
*identification_variable*

*conditional_expression ::= conditional_term* | *conditional_expression* ***OR*** *conditional_term*

*conditional_term ::= conditional_factor* | *conditional_term* ***AND*** *conditional_factor*

*conditional_factor ::= [* ***NOT*** *] conditional_primary*

*conditional_primary ::= simple_cond_expression* | ***(****conditional_expression**)***

*simple_cond_expression ::= comparison_expression* |
*between_expression* |
*like_expression* |
*in_expression* |
*null_comparison_expression* |
*empty_collection_comparison_expression* |

*collection_member_expression* |
*exists_expression*

*between_expression ::= arithmetic_expression [**NOT**] **BETWEEN***
*arithmetic_expression **AND** arithmetic_expression* |
*string_expression [**NOT**] **BETWEEN***
*string_expression **AND** string_expression* |
*datetime_expression [**NOT**] **BETWEEN***
*datetime_expression **AND** datetime_expression*

*in_expression ::= state_field_path_expression [**NOT**] **IN (** in_item {, in_item}* | subquery**)***

*in_item ::= literal | input_parameter*

*like_expression ::= string_expression [**NOT**] **LIKE** pattern_value [**ESCAPE** escape_character]*

*null_comparison_expression ::= {single_valued_path_expression |*
*input_parameter} **IS** [**NOT**] **NULL***

*empty_collection_comparison_expression ::=*
*collection_valued_path_expression **IS [NOT] EMPTY***

*collection_member_expression ::= entity_expression*
*[**NOT**] **MEMBER** [**OF**] collection_valued_path_expression*

*exists_expression::= [**NOT**] **EXISTS (**subquery**)***

*all_or_any_expression ::= { **ALL** | **ANY** | **SOME**} **(**subquery**)***

*comparison_expression ::= string_expression comparison_operator*
*{string_expression | all_or_any_expression}* |
*boolean_expression { **=|<>**}*
*{boolean_expression | all_or_any_expression}* |
*enum_expression { **=|<>**}*
*{enum_expression | all_or_any_expression}* |
*datetime_expression comparison_operator*
*{datetime_expression | all_or_any_expression}* |
*entity_expression { **=** | **<>** }*
*{entity_expression | all_or_any_expression}* |
*arithmetic_expression comparison_operator*
*{arithmetic_expression | all_or_any_expression}*

*comparison_operator ::= **=** | **>** | **>=** | **<** | **<=** | **<>***

*arithmetic_expression ::= simple_arithmetic_expression | **(**subquery**)***

*simple_arithmetic_expression ::= arithmetic_term |*
*simple_arithmetic_expression { **+** | **-** } arithmetic_term*

*arithmetic_term ::= arithmetic_factor | arithmetic_term { ***** | **/** } arithmetic_factor*

*arithmetic_factor ::= [{ **+** | **-** }] arithmetic_primary*

*arithmetic_primary ::= state_field_path_expression |*
*numeric_literal |*
*(simple_arithmetic_expression) |*
*input_parameter |*
*functions_returning_numerics |*
*aggregate_expression*

*string_expression ::= string_primary | **(**subquery**)***

*string_primary ::= state_field_path_expression* |
*string_literal* |
*input_parameter* |
*functions_returning_strings* |
*aggregate_expression*
*datetime_expression ::= datetime_primary* | ***(****subquery****)***
*datetime_primary ::= state_field_path_expression* |
*input_parameter* |
*functions_returning_datetime* |
*aggregate_expression*
*boolean_expression ::= boolean_primary* | ***(****subquery****)***
*boolean_primary ::= state_field_path_expression* |
*boolean_literal* |
*input_parameter* |
*enum_expression ::= enum_primary* | ***(****subquery****)***
*enum_primary ::= state_field_path_expression* |
*enum_literal* |
*input_parameter*
*entity_expression ::= single_valued_association_path_expression* | *simple_entity_expression*
*simple_entity_expression ::= identification_variable* | *input_parameter*
*functions_returning_numerics::=* ***LENGTH(****string_primary****)*** |
***LOCATE(****string_primary****,*** *string_primary[****,*** *simple_arithmetic_expression]****)*** **|**
***ABS(****simple_arithmetic_expression****)*** |
***SQRT(****simple_arithmetic_expression****)*** **|**
***MOD(****simple_arithmetic_expression, simple_arithmetic_expression****)*** **|**
***SIZE(****collection_valued_path_expression****)***
*functions_returning_datetime ::=* ***CURRENT_DATE***|
***CURRENT_TIME*** |
***CURRENT_TIMESTAMP***
*functions_returning_strings ::=* ***CONCAT(****string_primary****,*** *string_primary****)*** |
***SUBSTRING(****string_primary****,***
*simple_arithmetic_expression****,*** *simple_arithmetic_expression****)|***
***TRIM(****[[trim_specification] [trim_character]* ***FROM****] string_primary****)*** |
***LOWER(****string_primary****)*** |
***UPPER(****string_primary****)***
*trim_specification ::=* ***LEADING | TRAILING | BOTH***