



Translation of OCL Invariants into SQL:99 Integrity Constraints

Master Thesis

Submitted by:

Veronica N. Tedjasukmana
veronica.tedjasukmana@tu-harburg.de
Information and Media Technologies
Matriculation Number: 29039

Supervised by:

Prof. Dr. Ralf MÖLLER
STS - TUHH

Prof. Dr. Helmut WEBERPALS
Institut für Rechnertechnologie – TUHH

M.Sc. Miguel GARCIA
STS - TUHH

Hamburg, Germany
31 August 2006

Declaration

I declare that:
this work has been prepared by myself,
all literally or content-related quotations from other sources are clearly pointed out,
and no other sources or aids than the ones that are declared are used.

Hamburg, 31 August 2006

Veronica N. Tedjasukmana

Table of Contents

Declaration	i
Table of Contents	ii
1 Introduction	1
1.1 Motivation	1
1.2 Objective	2
1.3 Structure of the Work	2
2 Constraint Languages	3
2.1 Defining Constraint in OCL	3
2.1.1 Types of Constraints	4
2.2 Defining Constraints in Database	4
2.3 Comparison of Constraint Language	6
3 Design and Implementation	8
3.1 Discussion of different approaches	8
3.2 Object Oriented Query Language with View Approach	9
3.3 Implementation	14
3.3.1 Processing Steps	15
3.4 Problem and Limitation	23
3.5 Summary	24
4 xQL Specification	26
4.1 What is xQL?	26
4.2 Data Types and Values	26
4.2.1 Types from the UML Model	27
4.2.2 Collections	30
4.3 xQL Metamodel	30
4.3.1 Join and Navigation	31
4.3.2 Condition	32
4.4 Operation in xQL	34
4.5 Well-formedness rules of xQL	36
4.6 Summary	42
5 Transformation Recipes: Patterns and Procedures	43
5.1 The Negation of Boolean Expression	43
5.2 Operators	44
5.3 Mapping Procedures	44
5.4 Mapping Patterns	45
5.4.1 Navigation	45
5.4.2 Operation	46

5.4.2.1	Operation on Basic types	46
5.4.2.2	Operations on Collection Types	47
5.4.2.3	Outer Query	56
5.5	Summary	56
6	Introduction to SQL Generator in OctopusEE	57
6.1	Configuration of build path	57
6.2	Generated Files	57
6.3	Testing	58
6.4	Summary	60
7	Summary and Outlook	61
7.1	Summary	61
7.2	Further Work	62
Appendix A: Unmapped Operation		63
Appendix B: Configuration of OctopusEE		65
Appendix C: The Royal and Loyal Model		67
Appendix D: Database Schema or Royal and Loyal		68
References		71

1 Introduction

1.1 Motivation

Integrating object-oriented applications into relational database is no longer state of the art. Since its first emergence, many object relational persistence tools have been developed. The SQL impedance mismatch between object oriented concept and relational database concept such as association, inheritance, polymorphism, composition, and collections has been undoubtedly dealt with. However, these persistence tools are restricted only to class-to-table mapping, mainly based on attributes and associations. The power of Object Constraint Language (OCL) which enables us to write constraint or complex rules over an object model has yet to be utilized. At the moment, most business rules are still placed in the application program. Placing the business rules in the application programs has several disadvantages [5]:

1. *Duplication of effort*. If six different programs deal with various updates to a single table, each of them must include code that enforces the rules relating to the corresponding table.
2. *Lack of consistency*. If several programs written by different programmers handle updates to a table, they will probably enforce the rules somewhat differently.
3. *Maintenance problems*. If the business rules change, the programmers must identify every program that enforces the rules, locate the code, and modify it correctly.
4. *Complexity*. There are often many rules to remember.

By utilizing OCL we could shift the burden of specifying business rules from application layer to database layer. With this approach all the business rules is centralized in the database layer, thus minimizing the programming time and ensuring all applications working on the same database adhering to the same rules. There have been some existing approaches in specifying OCL invariants as constraints in database systems, either in Relational Database Management System (RDBMS) or Object Relational Database Management System (ORDBMS) [2] [3] [7] [13] [12]. In [12], OCL constraints are mapped to SQL constraints by exploiting the query facilities in ORDBMS. One of the advantages of this approach is the simplicity in collecting related records. Unfortunately, ORDBMS is not widely used and implemented. On the other hand, RDMBS is widely used and seems indispensable, but query facilities in it are too verbose especially when specifying relationship between tables.

1.2 Objective

In this project, we propose an approach to specifying OCL invariants as constraints in database systems by combining object oriented query language with the use of view and trigger. This approach is driven by the final release of JSR-220 Enterprise Java Beans 3.0. Along with the final release of EJB 3.0, Java Community Press introduces Java Persistence Query Language, an extension of Enterprise Query Language specified in EJB 2.x. [14]. The Java Persistence query language, also known as EJB3QL, can be compiled to a target language, such as SQL of a database.

Translating OCL to EJB3QL will result in the simplicity of query language, and compiling EJB3QL to SQL afterwards allows us to use the widely deployed RDMBS.

1.3 Structure of the Work

In next chapter we discuss about defining constraint in database and introduce OCL and how to define an invariant in OCL. Chapter 3 discusses approaches taken to define the constraints in relational database from OCL followed by our new approach and the implementation. Chapter 4 introduces xQL, our new Metamodel along with its specification and well-formedness rules. Chapter 5 presents the transformation pattern and procedure from OCL to xQL. Chapter 6 introduces SQL Generator in OcotopusEE along with its configuration in Eclipse. Chapter 7 gives conclusions and future works.

2 Constraint Languages

By definition, a constraint is a restriction on one or more values of (part of) an object-oriented model or system [15]. In this chapter we explore the constraint used in modeling language and constraint in database.

2.1 Defining Constraint in OCL

Merely utilizing a UML diagram is in general not refined enough to specify all the significant aspects of a model. The information presented by such a model has a possibility to be incomplete and imprecise. For instance, in the UML model shown in Figure 2.1, it is reasonable for every loyalty program of Royal and Loyal requires that every customer who enters a loyalty program be of legal age, which is equal or greater than 18. This can be written as an invariant:

```
context Customer
  inv ofAge: self.age >= 18
```

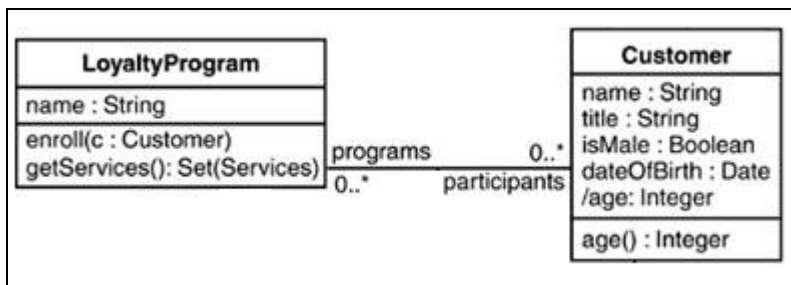


Figure 2.1 Royal and Loyal Model

The rules stated above cannot be expressed in UML. Thus, there is a need to describe additional constraints about the objects in the model. Object Constraint Language (OCL) is developed to fulfill this necessity. OCL is a formal language used to describe expressions on UML models. These expressions typically specify invariant conditions that must hold for the system being modeled or queries over objects described in a model [11]. Characteristics of OCL as taken from various sources:

- OCL is a declarative language. In a declarative language, an expression simply states *what* should be done, but now *how* [15].

- OCL is a pure specification language; therefore, an OCL expression is guaranteed to be without side effects. When an OCL expression is evaluated, it simply returns a value. It cannot change anything in the model [11].
- OCL is not a programming language; therefore, it is not possible to write program logic or flow control in OCL. You cannot invoke processes or activate non-query operations within OCL. Because OCL is a modeling language in the first place, OCL expressions are not by definition directly executable [11].
- OCL is a typed language so that each OCL expression has a type. To be well formed, an OCL expression must conform to the type conformance rules of the language. For example, you cannot compare an Integer with a String. Each Classifier defined within a UML model represents a distinct OCL type. In addition, OCL includes a set of supplementary predefined types. As a specification language, all implementation issues are out of scope and cannot be expressed in OCL [11].

2.1.1 Types of Constraints

There are four types of constraints in OCL:

- *Invariant*
An invariant is a constraint that should be true for an object during its complete lifetime [15]. Invariants often represent rules that should hold for the real-life objects after which the software objects are modeled.
- *Precondition* and *Postcondition*
Preconditions and postconditions are constraints that specify the applicability and effect of an operation without stating an algorithm or implementation. Precondition specifies the conditions that must hold before the operation executes. Postcondition specifies the conditions that must hold after the operation executes. Precondition and postcondition consist of an OCL expression of type Boolean.
- *Guard*
A guard is a constraint that guards the transition, from one state to another state. OCL expression acting as value of a guard is of type Boolean. The condition of guard should be true during the transition.

2.2 Defining Constraints in Database

To preserve the consistency and correctness of its stored data, relational database typically imposes one or more integrity constraints. These constraints restrict the data values that can be inserted into the database or created by a database update.

RDBMS provides four main types of static constraints [1] [5]:

1. *Assertions*

Assertions are the most general form of integrity constraint in SQL. Assertions are intended to specify a constraint over multi tables.

2. *Table constraints*

Table constraints are less general than assertions. It is used to restrict the rows in one particular table only. Table constraints are attached to a particular table by including them into the CREATE TABLE statement defining that table.

3. *Column constraints*

Column constraints are specified as part of a column definition when a table is created. Conceptually, they restrict the legal values that may appear in the column. Column constraints appear in column definition within the CREATE TABLE statement.

4. *Domain Constraints*

Domains are a specialized form of column constraints. They provide a limited capability to define new data types within a database. In effect, a domain is one of the predefined database data types plus some additional constraints, which are specified as part of the domain definition. The columns “inherit” the constraints of the domain.

Among those four types, assertion appears to be the most general constraint since it is not specified inside table or column structure in database. In theory, assertions could cause a very large amount of database processing overhead as they are checked for each statement that might modify the database. In practice, database will analyze the assertion and determine which tables and columns it involves. With assertion we are able to shift the complex constraint or business rule from application layer to database layer. Unfortunately, assertion is not supported by any commercial RDBMS.

Another technique to implement the complex rule in database layer is by utilizing trigger. The concept of a trigger is relatively straightforward. For any event that causes a change in the contents of a table, a user can specify an associated action that the database should carry out. The three events that can trigger an action are attempts to INSERT, DELETE, or UPDATE rows of the table. For example, below is a trigger enforcing every customer to have at least one valid card and raising an application error when an attempted update fails:

Note: For database schema, please refer to Appendix D

```
CREATE OR REPLACE TRIGGER "CUSTOMER_valid"
AFTER
insert or update or delete on "CUSTOMER"
DECLARE
    D NUMBER;
BEGIN
    select count(*) into D from invariant_Custom_Valid;
    IF (D > 0) THEN
        RAISE_APPLICATION_ERROR(-20000, 'constraint
violated');
    END IF;
END;
```

With view invariant_Custom_Valid as follows:

```
create or replace force view
"invariant_Custom_Valid" as
select
    customer0_.id as col_0_0_
from
    Customer customer0_
where
    (select count(f_cards2_.id)
    From Customer customer1_
    inner join CustomerCard f_cards2_
        on customer1_.id=f_cards2_.f_owner_id
    where
        f_cards2_.f_valid<>1
    )<1;
```

2.3 Comparison of Constraint Language

Constraints can be categorized based on various criteria. To classify the constraints related to OCL and integrity constraints in database, the following criteria are used:

- *Time* [1] [5] [3]: In the simple form, constraints in database are checked every time an attempt is made to change the database contents. This activity is categorized as *immediate* constraint. Besides this, SQL:99 specification introduces an additional capability for *deferred* constraint checking. In *deferred* constraint, checking will be deferred until the end of a transaction.
- *Activity*: Constraints can also be categorized as *active* and *passive*. Whereas *active constraints* maintain consistency by executing actions, *passive constraints* only prevent data manipulation operations which violate the consistency.

- *Location*: Constraints can be placed in the application layer or in database layer. Both methods have pro and contra. In this project we will exploit the integrity constraints provided by relational database, thus we will focus on constraints located in database layer.
- *System view* [2]: Constraints can be defined in various views of the system, as the static, dynamic and functional view. In a static view, a constraint usually is an *invariant*, i.e. a condition that should be true for an object during its complete lifetime [15]. In a dynamic view, constraints are used mainly to express the condition under which a transition from one state into another is allowed. In OCL we acknowledge this type of constraint as *guards*. In functional view, the output values and the induced state transformation of an operation are described with respect to the input values. In OCL, this is done by pre- and postconditions.
- *Policy on constraint violations* [2]: There are various actions can be taken when a constraint is not fulfilled: the implementation can be considered as faulty, the recent modification can be made undone, or actions can be taken to automatically correct the state.

Based on the criteria above, we can see that there is a common intersection between the constraint mechanisms of OCL and RDBMS. From all the four types of constraint defined in OCL, the most relevant type to relational database is *invariant*. Constraint in relational database is typically a *static constraint*. In the matter of *active* and *passive constraints*, RDBMS serves both: *active constraints* through TRIGGER statement and *passive constraints* through table constraints, column constraints, domain constraints and assertion. However, as OCL is a declarative language, we cannot invoke processes or activate non-query operations within OCL. So the similarity between OCL and RDBMS in this case is *passive constraint*. Based on the checking time, constraint in OCL as well as in RDBMS serves both *immediate* and *deferred* checking. How often and when to check a constraint depends on how serious the error could be. In policy on constraint violation, OCL as declarative language can only states the condition that has to be fulfilled without specifying any consecutive action, while RDBMS offers a way to rollback or re-establish a correct state.

3 Design and Implementation

3.1 Discussion of different approaches

Some approaches have been proposed for specifying OCL invariants as constraints in database systems, either in Relational Database Management System (RDBMS) or Object Relational Database Management System (ORDBMS). These approaches however have some advantages and also limitations.

In [12] OCL constraints are mapped to SQL constraints by exploiting the query facilities in ORDBMS. One of the advantages of this approach is the database can make use of the relationships between data to easily collect related records. In traditional RDBMS, collecting information from two tables requires a “JOIN”. For example, given that we have two tables: Customer and CustomerCard as shown below:

```
CREATE TABLE "CUSTOMER"
(
  "ID" NUMBER(19,0) NOT NULL ENABLE,
  "F_NAME" VARCHAR2(255 CHAR),
  "F_TITLE" VARCHAR2(255 CHAR),
  "F_ISMALE" NUMBER(1,0) NOT NULL ENABLE,
  "F_GENDER" NUMBER(10,0),
  "F_AGE" NUMBER(5,0),
  PRIMARY KEY ("ID") ENABLE
)

CREATE TABLE "CUSTOMERCARD"
(
  "ID" NUMBER(19,0) NOT NULL ENABLE,
  "F_VALID" NUMBER(1,0) NOT NULL ENABLE,
  "F_COLOR" NUMBER(10,0),
  "F_MYLEVEL_ID" NUMBER(19,0),
  "F_OWNER_ID" NUMBER(19,0),
  "F_MEMBERSHIP_ID" NUMBER(19,0),
  PRIMARY KEY ("ID") ENABLE,
)
```

In traditional RDBMS selecting all the rows in CUSTOMER which has valid CUSTOMERCARD, we have to join the CUSTOMER table and CUSTOMERCARD table.

```
SELECT *
FROM CUSTOMER C, CUSTOMERCARD CC
WHERE CUSTOMERCARD.F_OWNER_ID = CUSTOMER.ID
AND CC.F_VALID = 1
```

The same query in ORDBMS is much simpler:

```
SELECT *
FROM CUSTOMER
WHERE CUSTOMER.CUSTOMERCARD.F_VALID = 1
```

Although some of the ideas of object relational database have largely been adopted by SQL:99 specification, such as allowing user defined datatypes, it excludes the simplicity of query shown above.

Some other approaches are based on traditional RDBMS [2] [3] [7] [13]. In [13], OCL invariant is mapped into stored procedures. The transformation of the constraint can be done by calling this procedure. With this approach, complex loop expression, such as *iterate* - which doesn't have a direct counterpart in declarative SQL syntax - is easy to map. However, this approach depends extremely on programming languages rather than SQL declarative syntax. Furthermore, there is little consistency between DBMSs vendors on stored procedure syntax.

Second approach is implementation of OCL to SQL declarative syntax with assertions [2]. However, up to now, assertion is not supported by any DBMS vendors. Another way to implement integrity constraints in database is with the use of views and triggers [3] [7]. A view is created for each single OCL invariant, and for each data manipulation in corresponding tables, trigger is fired to evaluate generated views. Constraint is violated if view returns any tuples. This approach offers some advantages: view is supported by all DBMS vendors, and it also allows evaluating a complex condition involving arbitrary number of tables. This ability fulfills the vital part of integrity constraint. Other fact that should also taken into consideration is mapping from OCL invariant to declarative SQL code is simpler than the generation of procedural DBMS code in [13].

3.2 Object Oriented Query Language with View Approach

After studying previous approaches and driven by the final release of JSR-220 Enterprise Java Beans 3.0, we propose an approach to specify OCL invariants as constraints in database systems by combining object oriented query language with the use of view and trigger.

Along with the final release of EJB 3.0, Java Community Press introduces Java Persistence Query Language, an extension of Enterprise Query Language specified in EJB 2.x. It adds further operations, including bulk update and delete, JOIN operations, GROUP BY, HAVING, projection, and subqueries; and supports the use of dynamic queries and the use of named

parameters [14]. The Java Persistence query language, also known as EJB3QL, can be compiled to a target language, such as SQL of a database.

Joining Associations. By utilizing the enhanced power of EJB3QL, we are able to simplify the process of specifying OCL invariant as the integrity constraint in database systems and keep using relational databases. The first step in specifying OCL invariant as the integrity constraint is the mapping of OCL invariant to EJB3QL. This mapping process is much simpler compared to mapping the OCL invariant directly to SQL, since both OCL and EJB3QL are still in the object-oriented “world”. For example, given that we have the following OCL Invariant:

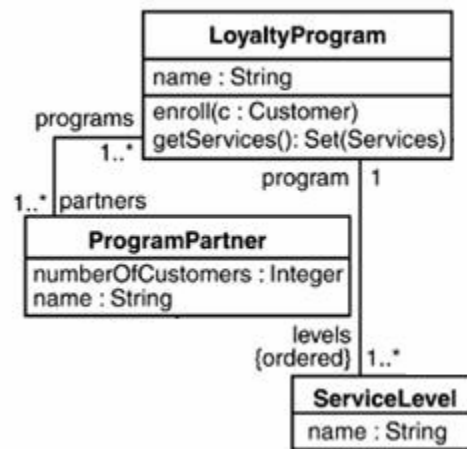


Figure 3.1 Joining Associations

context LoyaltyProgram

inv minServices: partners.deliveredServices->size() >= 1

In our Royal and Loyal example, it would be reasonable to require that a loyalty program offers at least one service to its customers. In order to specify the condition, from the context LoyaltyProgram we have to navigate through its program partners to the services they deliver. In database these objects will be mapped to tables as shown in the following SQL schema:

(Note that the SQL schema shown below only represents the relationship between tables and disregards other information. The full SQL schema of Royal and Loyal example can be found in Appendix D).

```

CREATE TABLE "LOYALTYPROGRAM"
(
  "ID" NUMBER(19,0) NOT NULL ENABLE,
  .....

```

```

        PRIMARY KEY ("ID") ENABLE
    )

CREATE TABLE "PROGRAMPARTNER"
(
    "ID" NUMBER(19,0) NOT NULL ENABLE,
    .....
    PRIMARY KEY ("ID") ENABLE
)

CREATE TABLE "LOYALTYPROGRAM_PROGRAMPARTNER"
(
    "F_PROGRAMS_ID" NUMBER(19,0) NOT NULL ENABLE,
    "F_PARTNERS_ID" NUMBER(19,0) NOT NULL ENABLE,
    PRIMARY KEY ("F_PROGRAMS_ID", "F_PARTNERS_ID") ENABLE,
    CONSTRAINT "FK403144A5830FFA37" FOREIGN KEY ("F_PROGRAMS_ID")
        REFERENCES "LOYALTYPROGRAM" ("ID") ENABLE,
    CONSTRAINT "FK403144A5C19CF8C1" FOREIGN KEY ("F_PARTNERS_ID")
        REFERENCES "PROGRAMPARTNER" ("ID") ENABLE
)

CREATE TABLE "SERVICE"
(
    "ID" NUMBER(19,0) NOT NULL ENABLE,
    .....
    "F_PARTNER_ID" NUMBER(19,0),
    PRIMARY KEY ("ID") ENABLE,
    CONSTRAINT "FKD97C5E951D4B56" FOREIGN KEY ("F_PARTNER_ID")
        REFERENCES "PROGRAMPARTNER" ("ID") ENABLE
)

```

Since LOYALTYPROGRAM have many-to-many association with PROGRAMPARTNER, the Join Table LOYALTYPROGRAM_PROGRAMPARTNER is required. The relationship between LOYALTYPROGRAM and LOYALTYPROGRAM_PROGRAMPARTNER is represented as the foreign key F_PROGRAMS_ID in LOYALTYPROGRAM_PROGRAMPARTNER. The relationship between LOYALTYPROGRAM_PROGRAMPARTNER and PROGRAMPARTNER is represented as the foreign key F_PARTNERS_ID in LOYALTYPROGRAM_PROGRAMPARTNER. The relationship between PROGRAMPARTNER and SERVICE is represented as the foreign key F_PARTNER_ID in SERVICE.

To encompass the entire objects, in SQL we have to combine them using *join* as shown in FROM clause below:

```

from LoyaltyProgram loyaltypro1_
join LoyaltyProgram_ProgramPartner f_partners2_
join ProgramPartner programpar3_
join Service f_delivere4_
where loyaltypro1_.id=f_partners2_.f_programs_id
        f_partners2_.f_partners_id=programpar3_.id
        programpar3_.id=f_delivere4_.f_partner_id

```

In traditional SQL above to combine tables, besides *join*, we also have to describe the join condition of combined tables, either in the *FROM clause* or in the *WHERE clause*. To retrieve the service delivered by a loyalty program we have to match ID of `LoyaltyProgram` to `f_programs_id` of `LoyaltyProgram_ProgramPartner`, `f_partner_id` of `LoyaltyProgram_ProgramPartner` to ID of `ProgramPartner` and ID of `ProgramPartner` to `f_partner_id` of `Service`. Moreover, to connect `LoyaltyProgram` to `ProgramPartner` we have to go through the Join Table `LoyaltyProgram_ProgramPartner` which is not visible from the OCL Invariant.

Let us compare with the following EJB3QL for the same OCL invariant:

```
from LoyaltyProgram loyaltyprogram
  join loyaltyprogram.f_partners i_ProgramPartner
  join i_ProgramPartner.f_deliveredServices service
```

EJBQL introduces path expression, an identification variable followed by the navigation operator (`.`) and a state-field or association-field [14]. Utilizing path expression, we not need to specify join condition explicitly. With path expression, EJBQL has enough information in the mapping document to then deduce the table join expression. This helps make mapping navigation in OCL invariant easier and in the same time make queries less verbose and more readable.

For example, In the *FROM clause* to map the navigation from `LoyaltyProgram` to `ProgramPartner`, we declare `loyaltyprogram.f_partners i_ProgramPartner`. The identification variable `i_ProgramPartner` evaluates to any `ProgramPartner` value directly reachable from `LoyaltyProgram`. The association-field `f_partners` is a collection of instances of the abstract schema type `ProgramPartner` and the identification variable `i_ProgramPartner` refers to an element of this collection. The type of `i_ProgramPartner` is the abstract schema type of `ProgramPartner`. The same explanation also applies to declaration `i_ProgramPartner.f_deliveredServices service`.

As we can infer from two query languages described above, translating OCL invariant to EJB3QL is simpler and hence it will produce less error-prone. Another advantage is time-saving in programming because using EJB3QL we need not care the Join Table, of which we should care if we navigate through many-to-many objects such as `LoyaltyProgram` and `ProgramPartner`.

Polymorphic Queries. By default, all queries in EJB3QL are polymorphic. That is, the FROM clause of a query designates not only instances of the specific entity class(es) to which it explicitly refers, but subclasses as well [14]. For example in our Royal and Loyal model, selecting Transaction will not only return instances of Transaction but also instances of Burning and Earning.

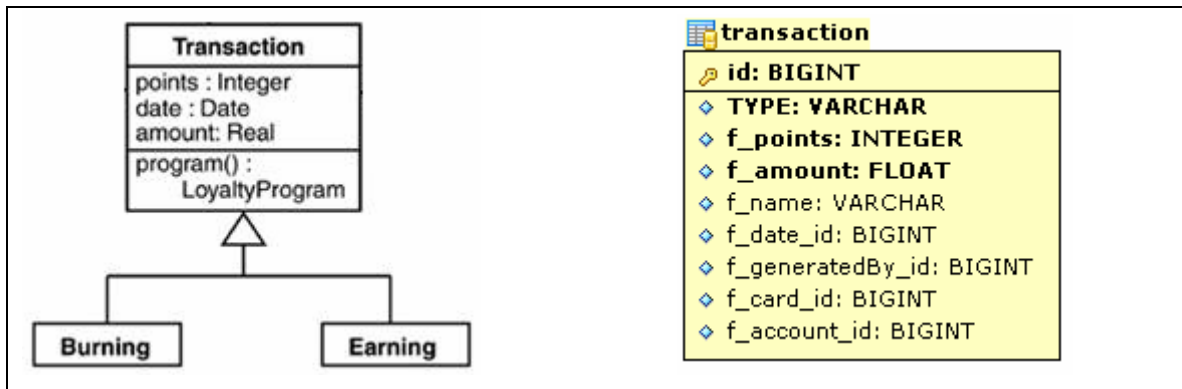


Figure 3.2 Polymorphic Queries

However, queries against the subclasses might be not as trivial as query against superclass. We might define a rule which states that a customer cannot obtain more than 50 points of bonus point as follows:

```
context Burning
  inv maxbonus: self.points < 50
```

In traditional SQL, we would expect that we have BURNING table and simply make a restriction in the WHERE clause that points should be less than 50. Unfortunately, following the table per class hierarchy approach [10], we do not have BURNING table. Instead, we only have TRANSACTION table with type discriminator column to represent subclasses (See Figure 3.2). Fortunately, disregarding which approach is taken on mapping class inheritance, we can swiftly write a query in EJB3QL as follows:

```
select burning.id
from Burning burning
where not burning.f_points < 50
```

And still abide by our goal; we compiled the EJB3QL to our traditional SQL as follows:

```
select
  burning0_.id as col_0_0_
from
  Transaction burning0_
```

```
where
  burning0_.DTYPE='Burning'
  and burning0_.f_points >= 50
```

Creating View. The translated SQL Query is used as the <SELECT statement> in constructing a view. We can create a view using a CREATE VIEW command as follows:

```
CREATE VIEW <view name> [(<column list>)] AS <SELECT statement>
```

Following above rule, a created view for example on Burning subclass is as follows:

```
create view maxbonus as
select
  burning0_.id as col_0_0_
from
  Transaction burning0_
where
  burning0_.DTYPE='Burning'
  and burning0_.f_points >= 50
```

3.3 Implementation

To implement our approach, the domain model of OCL invariant should be mapped to relational databases beforehand. For this prerequisite we choose OctopusEE, an extended version of Octopus which implements MDA-driven generation of EJB3 persistence artifacts. Octopus itself is an MDA tools which is able to transform UML model along with its OCL expressions into Java code. Octopus also able to statically check OCL expressions. It checks the syntax, as well as the expression types, and the correct use of model elements like association roles and attributes [8].

OctopusEE uses Hibernate as its ORM tool. Hibernate provides libraries of classes which are able to map EJB3QL to SQL automatically. One of the advantages of using Hibernate libraries is it supports multiple SQL dialect, such as: Oracle 8i, 9i, 10g, DB2 7.1, 7.2, 8.1, Microsoft SQL Server 2000, Sybase 12.5 (JConnect 5.5), MySQL 3.23, 4.0, 4.1, 5.0, PostgreSQL 7.1.2, 7.2, 7.3, 7.4, 8.0, 8.1, TimesTen 5.1, HypersonicSQL 1.61, 1.7.0, 1.7.2, 1.7.3, 1.8, and SAP DB 7.3¹. The mapping of UML model to a relational database in OctopusEE is addressed in [6].

¹ Database Supported by Hibernate Team [<http://hibernate.org/260.html>]

3.3.1 Processing Steps

The processing step of translating OCL invariants into SQL is depicted in the diagram below:

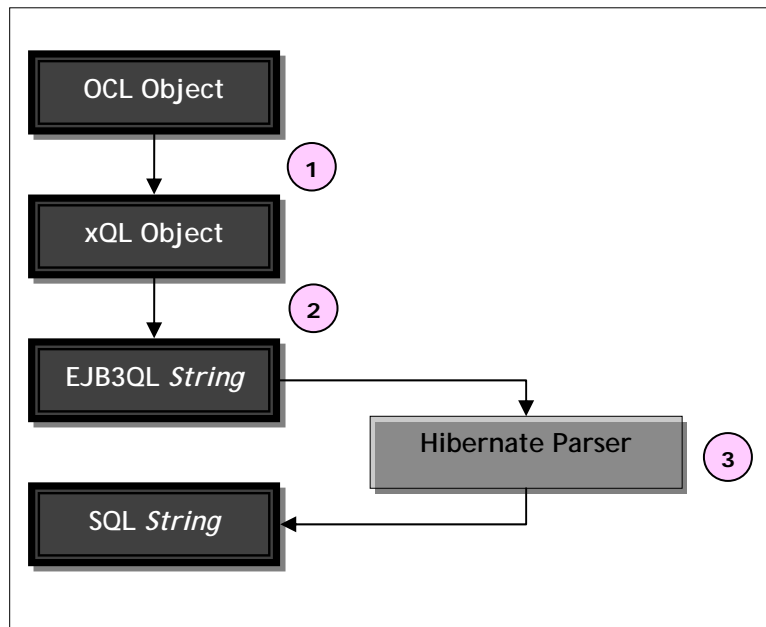


Figure 3.3 The processing step of translating OCL invariants into SQL

First, the parsed OCL expression is taken as input for xQL. xQL plays as an intermediate layer between OCL to SQL. Abstract syntax of OCL expression is transformed to xQL model following the transformation pattern described in Chapter 5. After the abstract syntax of xQL is well-built, it is serialized to HQL String, which is later become the input for Hibernate Parser. In Hibernate, SQL String will be generated from EJB3QL String. The more detail explanation is as follows:

① **OCL Object to xQL Object.** Following the pipe and filter architectural pattern, transformation of OCL expression to xQL Metamodel is done through several sequential processing steps. In the first step, we restructure the OCL expression from inline structure to expression structure in xQL model. Inline structure is a way of structuring an expression object into a sequence based on which object appear first in the expression. Expression structure is a way of structuring an object expression into an operand-operator structure where the base object is the operation expression object and the other object is considered as the operand and used as the argument for this operation expression object. To see the difference, please compare the AST in inline structure and AST in expression structure for OCL invariant below:

```

context Customer
inv: cards->select( valid = true )->size() > 1

```

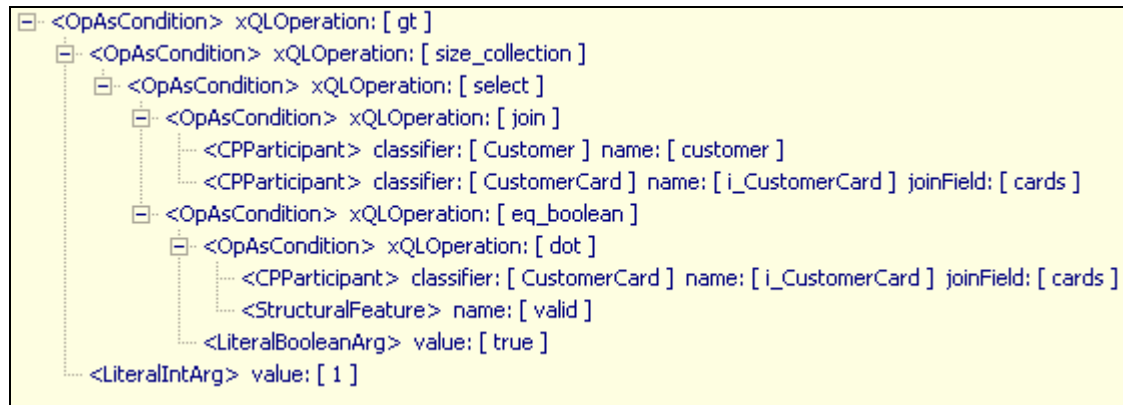


Figure 3.4 Expression Structure

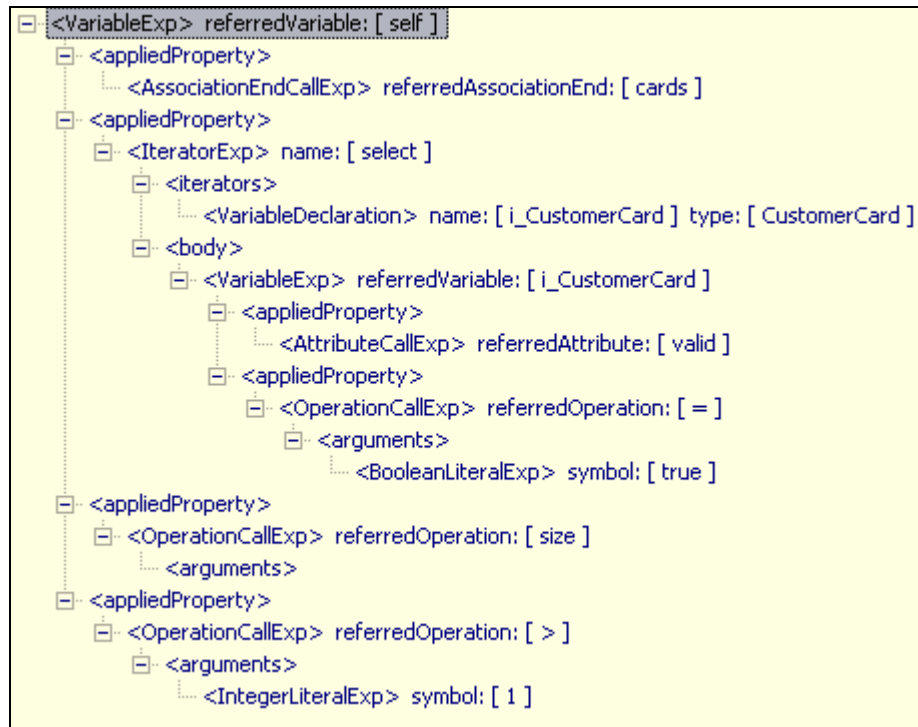


Figure 3.5 Inline Structure

The second step is mapping the restructured OCL invariant from previous step into xQL Metamodel. The transformation recipe from OCL to xQL can be found in Chapter 5, while the specification of xQL can be found in Chapter 4.

The transformation in first step as well as in the second step is done by visiting the abstract syntax tree of OCL invariants. These steps are connected by the data flow through the system; the output data of a step is the input to the subsequent step. The restructuring of OCL expression to xQL model is sequentially done by two visitors: `NavigationVisitor` class and `OperationVisitor` class. First, OCL Expression as the data source is the input for first transforming layer, `NavigationVisitor` Class. The output of this class as well as the OCL Expression is an input for the second layer, `OperationVisitor` Class. At the end of the second layer, the initial xQL object is achieved. In initial xQL object, we have all the necessary nodes and operations but not in the query structure but only focusing in the condition part. In the last layer, `xQL2Visitor` Class will transform the initial xQL object into a complete xQL model in query structure. The hierarchy of visitor classes involved in the transformation is depicted in Figure 3.6.

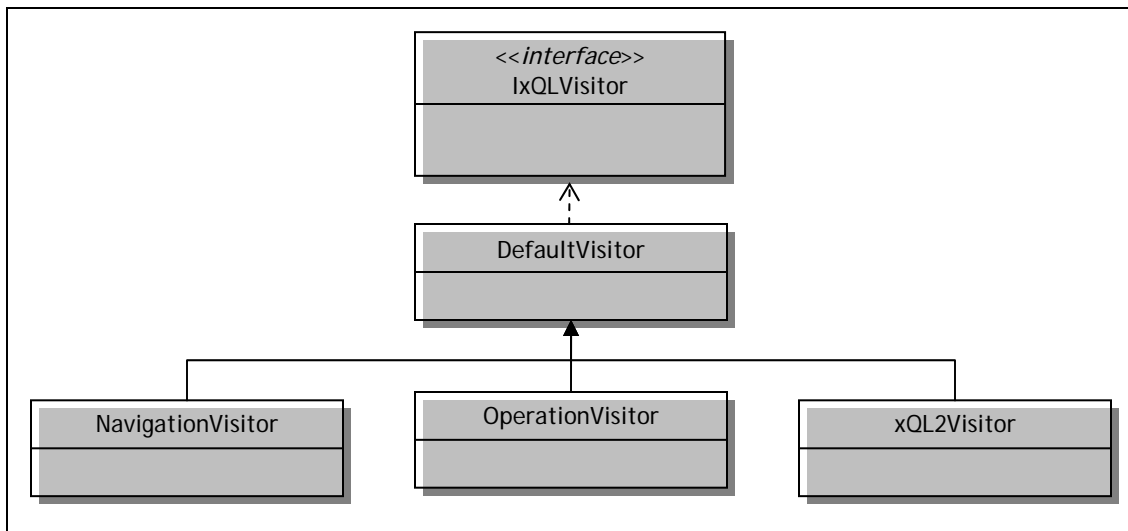


Figure 3.6 Hierarchy of xQLVisitor classes

To get a detail insight on how xQL Metamodel is generated out of OCL Expression, first we should take a peek at `ASTxQLViewer.openViewOn` method.

Code 1 openViewOn method in ASTxQLViewer Class

```

public void openViewOn(IOclExpression elem) {
    if (elem != null) && elem instanceof OclExpression) {
        AstWalker w = new AstWalker();
        XQLCollections collections = new XQLCollections();
        NavigationVisitor nv = new NavigationVisitor(collections);
        w.walk(elem,nv);
        OperationVisitor ov = new OperationVisitor(collections);
        w.walk(elem,ov);

        xQLWalker wx = new xQLWalker();

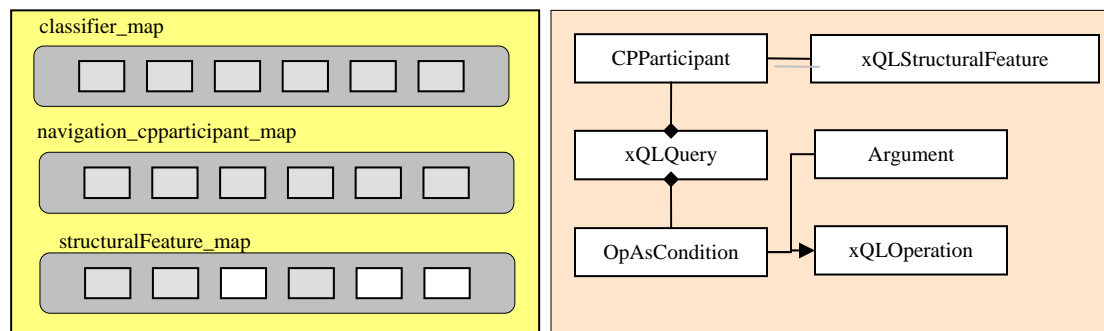
        //show xql2
        xQL2Visitor xql2w = new xQL2Visitor();
        IxQLQuery xql = (IxQLQuery) wx.walk(collections.getQuery().
            getCondition(),xql2w);

        XmlxQLVisitor xml = new XmlxQLVisitor();
        Element tree = (Element) wx.walk(xql,xml);
        Element root = new Element("root");

        root.addContent(tree);
        viewer.setInput( root);
        viewer.refresh();
        viewer.expandAll();
        XQLToString sml = new XQLToString();
        String str = wx.walk(xql,sml).toString();

        HqlTest.accept(str);
    }
}

```

**Figure 3.7 Attributes of XQLCollections Class**

In `openViewOn` method, an instance of `AstWalker` and `XQLCollections` are created. `XQLCollections` class (see Figure 3.7) has 3 `HashMap` objects (`classifier_map`, `structuralFeature_map`, and `navigation_cppparticipant_map`) and `xQLQuery` object as its attribute. These maps are used for storing `CPPParticipants` and `xQLStructuralFeature` objects as a result of `AstWalker` walking through the `OCLExpression` carrying `NavigationVisitor`. The value of these maps will be retrieved when processing `OCL Expression` in `NavigationVisitor` and later in `OperationVisitor` class.

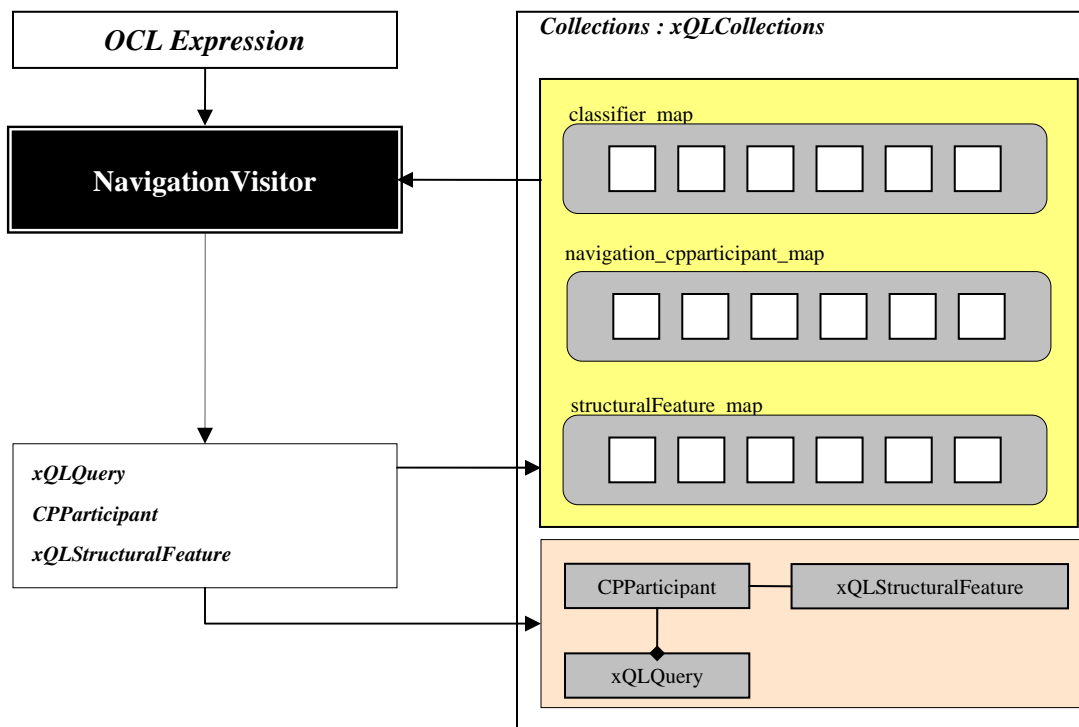


Figure 3.8 The processing steps of constructing XQLQuery Model – Step 1

In the first step of constructing xQL model, `NavigationVisitor` takes `OCLExpression` as its input. Here, we only implement some methods which produce `CPParticipant` and `xQLStructuralFeature`. `CPParticipant` and `xQLStructuralFeature` are owned by `xQLQuery`. These objects are stored in `HashMap` so that they could easily be retrieved in next process (`OperationVisitor`). This first step is depicted in Figure 3.8.

The second step is walking through the `OCLExpression` with `OperationVisitor` class. Here, `OpAsCondition`, `xQLOperation` and `Argument` object is created. `Argument` object might contain some literal objects. In the process of creating these objects, we often access the `HashMap` to retrieve `CPParticipant` or `xQLStructuralFeature` object. This process is depicted in Figure 3.9. In this step the initial xQL object is ready.

The last step is mapping the operation initial xQL object according to transformation pattern described in Chapter 5 to well-formed xQL object in a query structure. In this step, `xQL2Visitor` walks through the initial xQL object and transforms all the visited OCL operation to xQL operation.

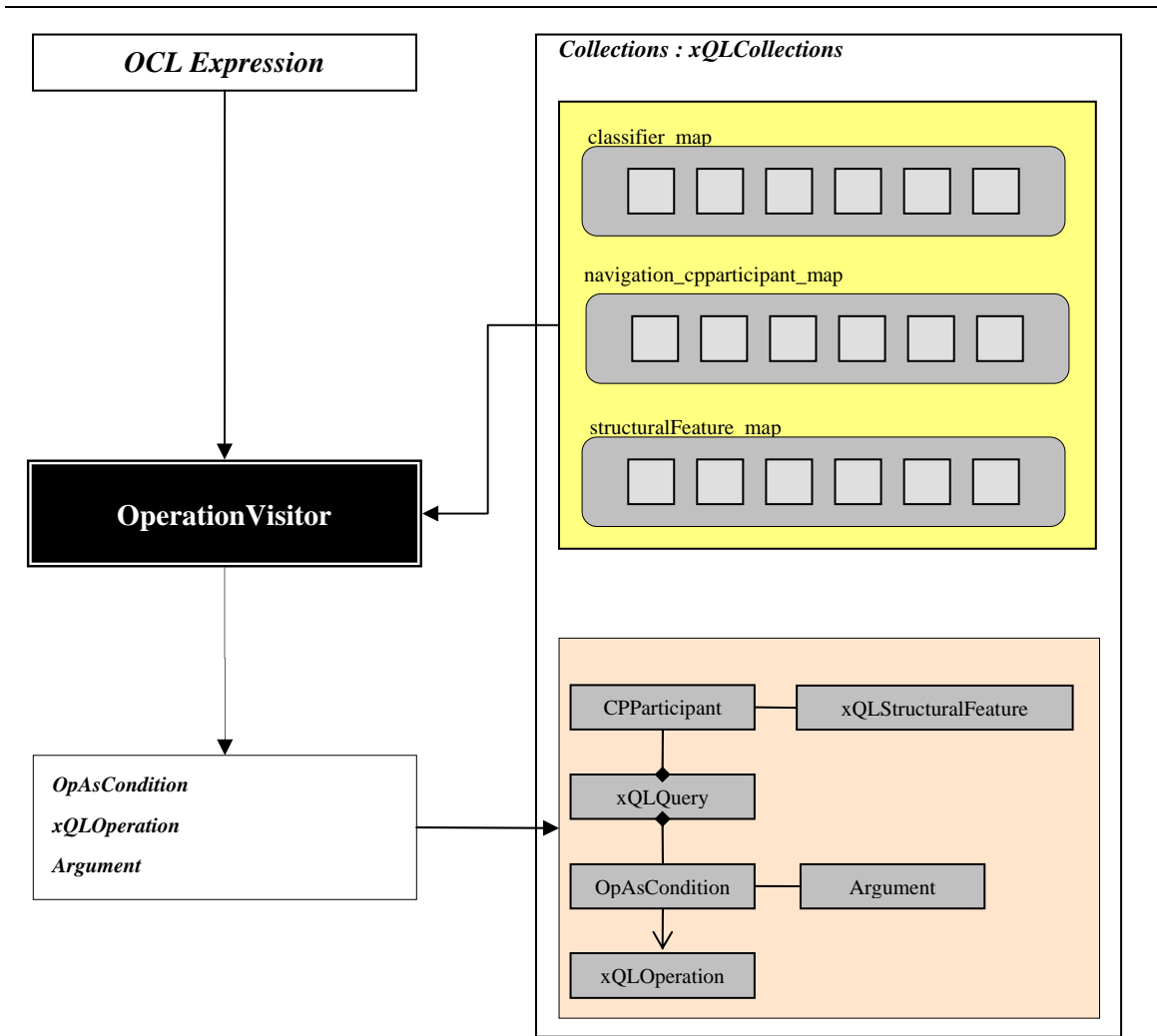
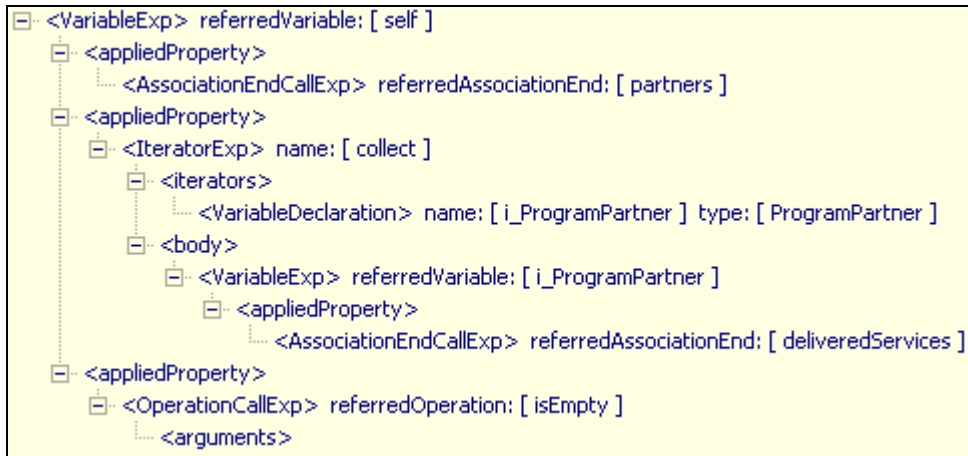


Figure 3.9 The processing steps of constructing XQLQuery Model – Step 2

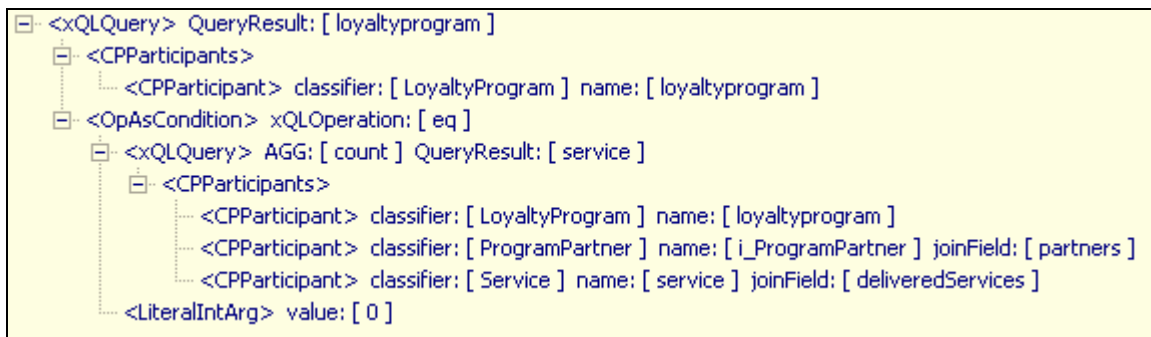
For example given that we have the following OCL invariant:

```
context LoyaltyProgram
inv: self.Membership.account->one( number < 10000 )
```


The abstract syntax tree of OCL above is:



And the abstract syntax tree of xQL for above OCL invariant is:



② **xQL Object to EJB3QL String.** After the xQL object is well-formed, we serialize the object to EJB3QL String. xQLWalker class walks through the xQL object by carrying xQLtoString class. xQLtoString will visit each object in abstract syntax of xQL and generate their corresponding EJB3QL String. For example for above example the generated EJB3QL string is as follows:

```

select loyaltyprogram.id
from LoyaltyProgram loyaltyprogram
where not
  (select COUNT(service.id)
   from LoyaltyProgram loyaltyprogram
   join loyaltyprogram.f_partners i_ProgramPartner
   join i_ProgramPartner.f_deliveredServices service )= 0
  
```

As we can see from above example, `<xQLQuery>` object will be serialized as a *SELECT statement*. *SELECT statement* is composed of *SELECT clause*, *FROM clause* and optional *WHERE clause*.

SELECT clause. Each `xQLQuery` has `QueryResult` attribute and sometimes aggregate function (marked with `AGG:[]` in the AST of xQL above) is also appeared. `QueryResult` attribute will be serialized as *SELECT clause*, and when aggregate function appears, the `QueryResult` will be aggregated according to which aggregate function is used. The serialized xQL can only have one object to be selected.

FROM clause. The `<CPParticipants>` node will be serialized as *FROM clause*. `CPParticipants` node can have arbitrary number of `CPParticipant`. Each `CPParticipant` consists of a classifier, a name and except the first `CPParticipant`, a `joinField`. Name will be serialized as an identification variable while both classifier and `joinField` will compose a path expression. A path expression is an identification variable followed by the navigation operator (`.`) and a state-field or association-field [14]. If a `CPParticipant` has no `joinField`, the classifier is serialized as an abstract schema type.

WHERE clause. The `<OpAsCondition>` node will be serialized as *WHERE clause*. The *WHERE clause* of a query consists of a conditional expression used to select objects or values that satisfy the expression. The *WHERE clause* restricts the result of a select statement [14]. The root `xQLOperation` always an instance of boolean expression. An `xQLQuery` could be exists in `OpAsCondition` and will be serialized as a subquery.

The constructed EJB3QL string is following the Java Persistence Query Language specification defined in [14].

③ **EJB3QL String to SQL String.** After we have the EJB3QL string in hand, the final step is to create a SQL view construct out of it. SQL string is generated from EJB3QL by utilizing Hibernate library called `QueryTranslator` as shown in Code 2.

Code 2 generateSQL method in SQLGenerator.java

```

public static String generateSQL(final SessionFactory sf,
                                final String query) {
    Session session = null;
    SessionFactoryImpl sfimpl = (SessionFactoryImpl) sf;
    HQLQueryPlan plan = new HQLQueryPlan(query, false, Collections.
        EMPTY_MAP, sfimpl);
    StringBuffer str = new StringBuffer(256);
    String sql = null;
    QueryTranslator[] translators = plan.getTranslators();
    for (int i = 0; i < translators.length; i++) {
        QueryTranslator translator = translators[i];
        Iterator sqls = translator.collectSqlStrings().iterator();
        while (sqls.hasNext()) {
            sql = (String) sqls.next();
        }
    }
    return formatForScreen(sql);
}

```

The generated SQL from generateSQL method will be used as a <SELECT statement> in constructing a view. A create view script is the final result of our OCL invariant to SQL translation.

3.4 Problem and Limitation

In this project we have shown that it is possible to specify OCL invariants as constraints in database systems by combining object oriented query language with the use of view and trigger. Our goal to translate the OCL invariants to EJB3QL has been achieved with some limitations. Differences in operation behavior of OCL and EJB3QL cause some operation in OCL cannot be translated into EJB3QL, such as *iterate*. The complete list of unmapped operation can be found in Appendix A. Another limitation comes from our dependency on class-to-table mapping technique taken by OctopusEE. Two main limitations in this case are:

- To navigate through classes which are linked with association class, the navigation class should be explicitly mentioned in the navigation paths. For example:

<pre> context LoyaltyProgram inv: self.participants->size() < 10000 </pre>
--

Should be written like this:

<pre> context LoyaltyProgram inv: self.Membership.participants->size() < 10000 </pre>

- @Transient datatype is not mapped into a column in database, so we cannot translate the OCL invariant involving this attribute. For example:

```
context Customer
  inv ofAge: age >= 18
```

3.5 Summary

Some approaches have been proposed for specifying OCL invariants as constraints in database systems, either in Relational Database Management System (RDBMS) or Object Relational Database Management System (ORDBMS). These approaches however have some advantages and also limitations. In [12], OCL constraints are mapped to SQL constraints by exploiting the query facilities in ORDBMS. One of the advantages of this approach is the database can make use of the relationships between data to easily collect related records. Unfortunately ORDBMS is not widely used. Some other approaches are based on traditional RDBMS [2] [3] [7] [13]. Here, OCL invariant is mapped into stored procedures [13], assertions [2] or views and triggers [3] [7].

In this project, we propose an approach to specify OCL invariants as constraints in database systems by combining object oriented query language with the use of view and trigger. Along with the final release of EJB 3.0, Java Community Press introduces Java Persistence Query Language. The Java Persistence query language, also known as EJB3QL, can be compiled to a target language, such as SQL of a database. With this approach we could gain some advantages:

- **Joining Associations.** By utilizing the enhanced power of EJB3QL, we are able to simplify the process of specifying OCL invariant as the integrity constraint in database systems and keep using relational databases. EJBQL introduces path expression, an identification variable followed by the navigation operator (.) and a state-field or association-field [14]. Utilizing path expression, we not need to specify join condition explicitly. With path expression, EJBQL has enough information in the mapping document to then deduce the table join expression. This helps make mapping navigation in OCL invariant easier and in the same time make queries less verbose and more readable.
- **Polymorphic Queries.** By default, all queries in EJB3QL are polymorphic. That is, the FROM clause of a query designates not only instances of the specific entity class(es) to which it explicitly refers, but subclasses as well [14]. We might define a rule which involves subclasses that, following the table per class hierarchy approach [10], are not

mapped into a table. With polymorphic queries, no matter what approach is taken in mapping class inheritance, we can swiftly write a query in EJB3QL.

The processing step of translating OCL invariants into SQL is depicted in the diagram below:

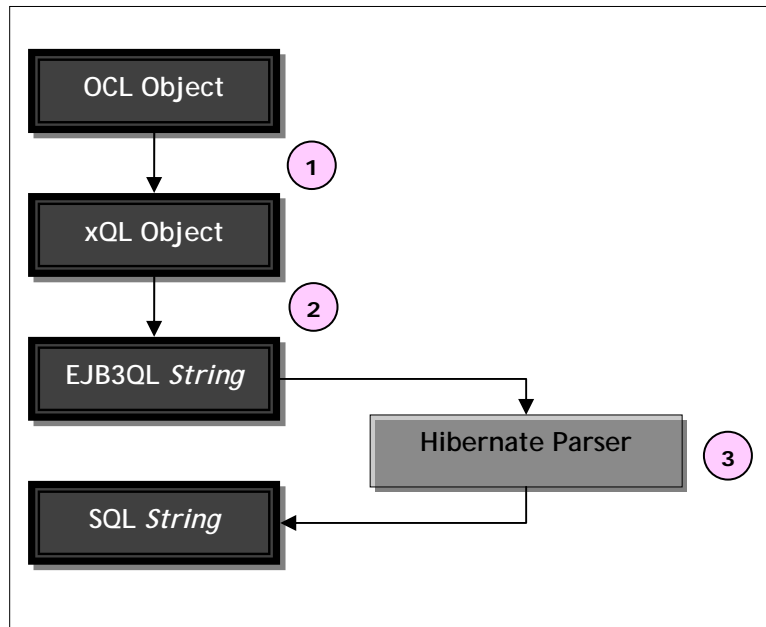


Figure 3.10 The processing step of translating OCL invariants into SQL

First, the parsed OCL expression is taken as input for xQL. xQL plays as an intermediate layer between OCL to SQL. Abstract syntax of OCL expression is transformed to xQL model following the transformation pattern described in Chapter 5. After the abstract syntax of xQL is well-built, it is serialized to HQL String, which is later become the input for Hibernate Parser. In Hibernate, SQL String will be generated from EJB3QL String.

We have shown that it is possible to specify OCL invariants as constraints in database systems by combining object oriented query language with the use of view and trigger. Our goal to translate the OCL invariants to EJB3QL has been achieved with some limitations. Differences in operation behavior of OCL and EJB3QL cause some operation in OCL cannot be translated into EJB3QL, such as *iterate*. The complete list of unmapped operation can be found in Appendix A. Another limitation comes from our dependency on class-to-table mapping technique taken by OctopusEE. Two main limitations in this case are (1) To navigate through classes which are linked with association class, the navigation class should be explicitly mentioned in the navigation paths, and (2) we cannot translate OCL invariant which involves @Transient datatype, since it is not mapped into a column in database.

4 xQL Specification

This chapter introduces xQL, an intermediate level of OCL invariant - SQL translation. The description is divided into several sections. The first section gives description of xQL. The second section describes the model of xQL. The third section describes all the operation used in xQL, and the last section defines the well-formedness rules of xQL.

4.1 What is xQL?

xQL is an intermediate step of translating OCL invariant to SQL. It is developed to ease the transformation step. xQL is mainly composed of OCL expression and HQL expression. While xQL borrows some of HQL operations to make expressions, the data types is mainly taken from OCL data types. At the end, to complete the translation steps, we will serialize the abstract syntax of xQL to HQL String.

OCL invariant basically built based on navigation and boolean expression. Navigation involves one or more objects. In xQL, we see the OCL invariant from database perspective where invariant is a way to query a database with the condition specified, which should not return any result otherwise the constraint is broken. Object participating in navigation is seen as the join between tables in the *FROM clause* and boolean expression is seen as condition appear in the *WHERE clause*.

4.2 Data Types and Values

In xQL, a number of basic types are predefined. The most basic value in xQL is a value of one of the basic types. The basic types defined in the xQL are Integer, Real, String, and Boolean. The basic types of xQL, with corresponding examples of their values, are shown in the following table.

Table 4.1 Basic xQL Types

Types	Values
Boolean	True, false
String	“This is a string”
Integer	1, 2, 3, ...
Real	0.5, 0.75, 1.25, ...

Real

The standard type Real represents the mathematical concept of real. Note that Integer is a subclass of Real, so for each parameter of type Real, you can use an integer as the actual parameter.

Integer

The standard type Integer represents the mathematical concept of integer.

String

The standard type String represents strings, which can be either ASCII or Unicode.

Boolean

The standard type Boolean represents the common true/false values.

4.2.1 Types from the UML Model

Each xQL expression is the translation from OCL expression which is written in the context of a UML model, a number of classifiers, their features and associations, and their generalizations. OCL expressions can refer to Classifiers (types, classes, interfaces, associations classes) and all attributes, association-ends, methods, and operations without side-effects that are defined on it can be used. xQL wraps types from UML Model with CPParticipant and xQLStructuralFeature. For the purpose of this project, we will refer only to attributes, association-ends, and association class.

Attributes

For example, an invariant in OCL stating the age of a Customer is always greater than zero is written as follows:

```
context Customer
inv: self.age > 0
```

The value of the subexpression *self.age* is the value of the *age* attribute on the particular instance of Customer identified by *self*. The type of this subexpression is the type of the attribute *age*, which is the standard type Integer. Using attributes and operations defined on the basic value types, we can express calculations etc. over the class model.

In xQL, attribute age is wrapped in StructuralFeature as shown in the following picture:

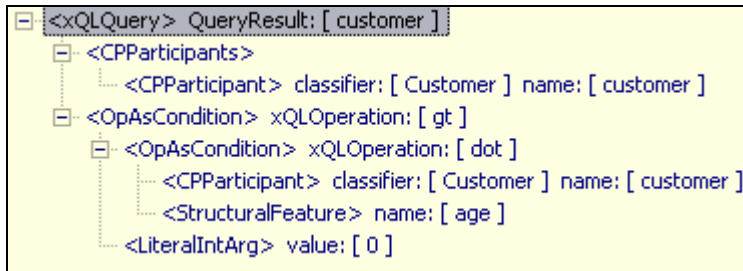


Figure 4.1 Attribute in xQL

After the xQL is serialized into HQL, age becomes one of the columns in table Customer:

```

select customer.id
from Customer customer
where not customer.f_age > 0

```

AssociationEnds, Association Class and Navigation

In OCL, starting from a specific object, we can navigate an association on the class diagram to refer to other objects and their properties. To do so, we navigate the association by using the opposite association-end. If the multiplicity of the association-end has a maximum of one then the navigation results in object. If the multiplicity of the association-end is more than one, then the navigation results in collection of object. Other means of navigation is using association class. We can navigate from the association class itself to the objects that participate in the association.

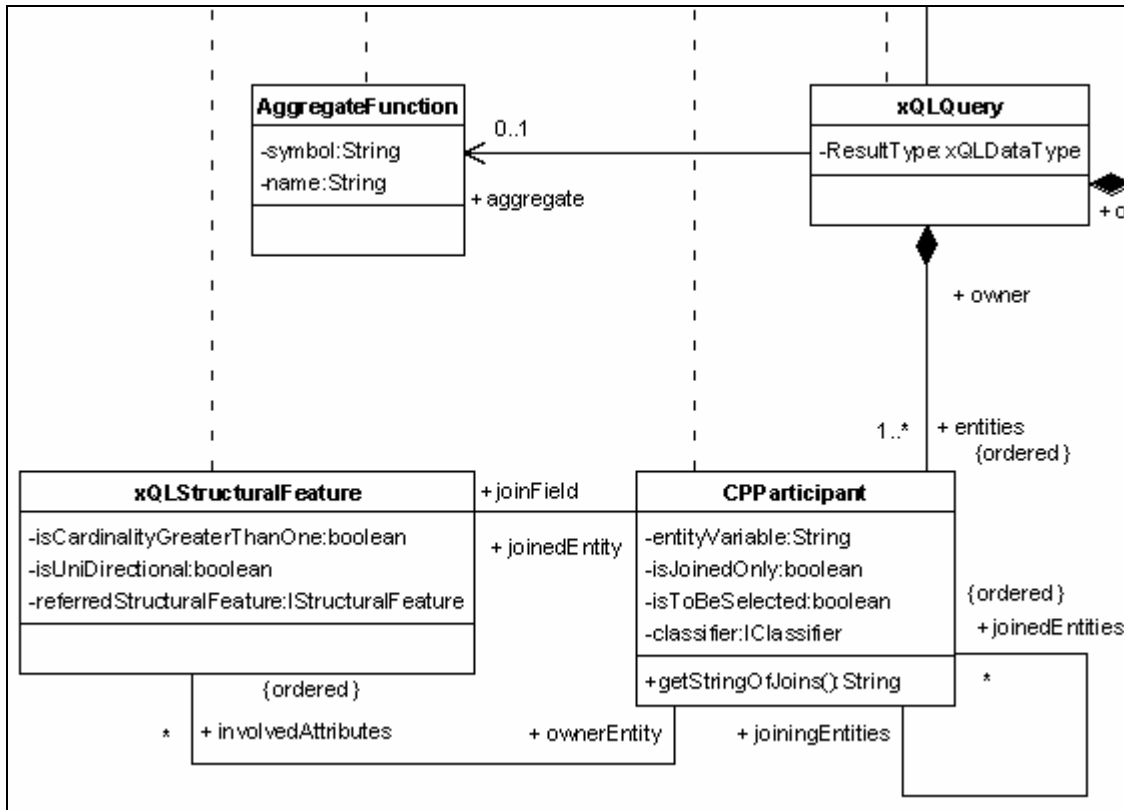


Figure 4.2 CPParticipant and xQLQuery

In xQL, each object participating in navigation is mapped into a CPParticipant. So, for every classifier appears in the navigation, call it the context, association end, or association class, a CPParticipant is instantiated. The instantiated CPParticipants exist as entities of an xQLQuery, as shown in the class diagram above. Later when we serialize the AST of xQL, these CPParticipant will be the join between tables. The result of an xQLQuery is always a collection although it might consist of only one element.

For example, an OCL invariant which involves association end with multiplicity not greater than one:

```

context Membership
inv: self.account.points > 5
  
```

In xQL expression `self.account` will instantiate CPParticipant membership and CPParticipant loyaltyaccount, each corresponding to its Classifier, Membership and LoyaltyAccount. `points` as an attribute of LoyaltyAccount will be treated as the QueryResult of xQLQuery with type Integer. The AST of created xQL corresponding to the example above is as follows:

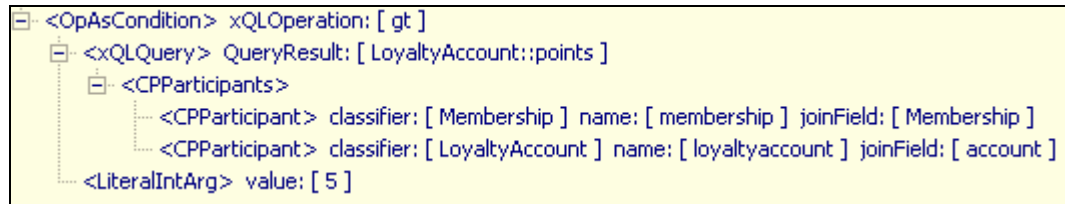


Figure 4.3 Example of navigation involving association end with multiplicity less than 1

Although the result of this navigation is an object, xQL treats all its entities as a join between tables, so the result is always a collection.

4.2.2 Collections

Collections in xQL can be produced in two ways: as a result of xQLQuery or define literally by the user. The type Collection is predefined in OCL. The element of the collection take the type of basic types which is either a String, Integer, Real or Boolean.

Collection Literals

Collection can be specified by a literal in xQL. Users can specify each element in the collection by using curly brackets to surround the elements of the collection. The elements in the collection are written within, separated by commas.

```
collection {1, 2, 5, 88}
```

Another way to define collection literals is to specify the interval of the element in the collection which is called collection range. Collection range consists of two expressions of type Integer, separated by '..'.

```
sequence{1..10}
```

in the model, a collection literals is be hold in a List and a collection range will be hold in a HashMap.

4.3 xQL Metamodel

xQL starts with xQLQuery, the class which hold the main part of the model: CPParticipant object and OpAsCondition. When we later serialize the xQL AST to HQL, CPParticipant represent the SELECT clause and FROM clause while OpAsCondition represent the WHERE clause. In this section we will describe the xQL Metamodel in two parts, first we discuss the entire object which built the SELECT clause and FROM clause and the second we will discuss the objects which involves in building the condition in the WHERE clause.

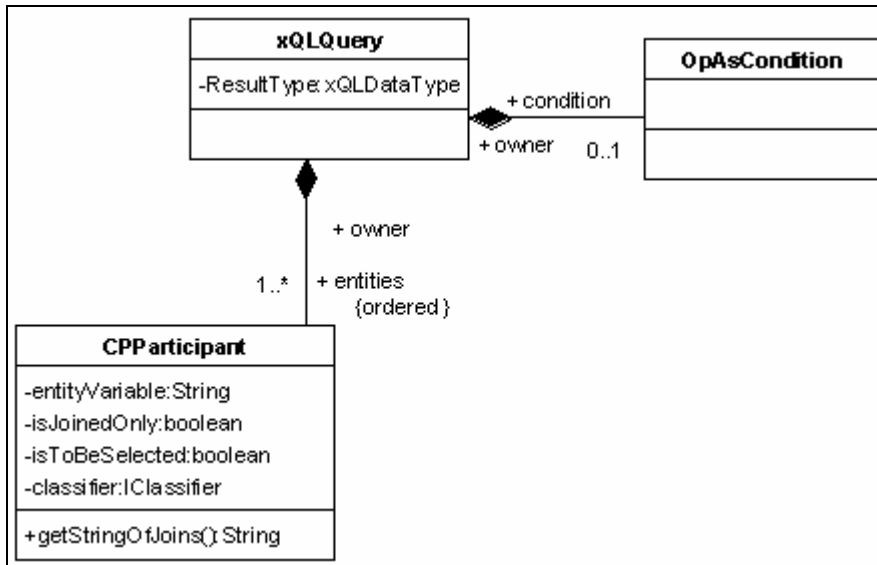


Figure 4.4 CPParticipant and OpAsCondition of xQL Metamodel

4.3.1 Join and Navigation

To handle the navigation in OCL invariant, **xQLQuery** and **CPParticipant** is made. **xQL** always starts with **xQLQuery** which forms the query object. Navigation in OCL is mapped to **CPParticipant**.

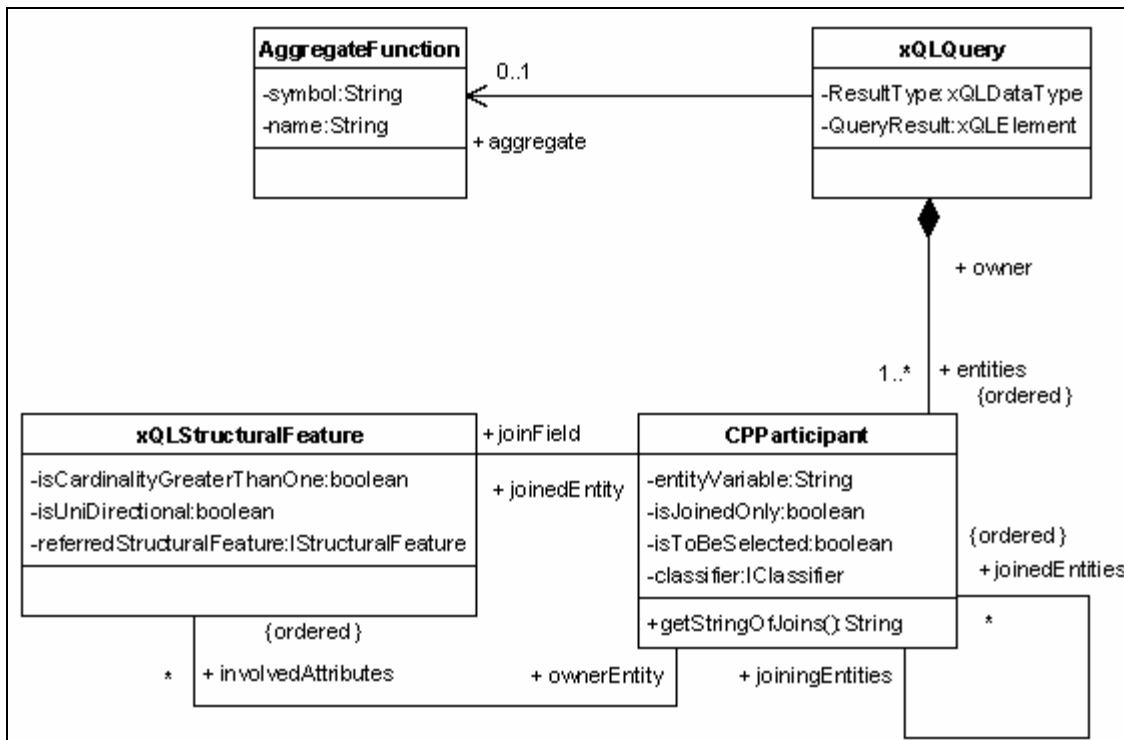


Figure 4.5 CPParticipant in xQL

xQLQuery

xQLQuery corresponds to query as a whole. ResultType defines the type of the query result. The result of xQLQuery always in the form of collection, except when AggregateFunction appears. QueryResult defines the *SELECT clause*. Not like ordinary query, xQL only select one column of the table.

AggregateFunction

AggregateFunction is used when we want to have a single aggregate value over a QueryResult. xQL provides two aggregate functions: SUM and COUNT. While SUM computes the sum of an expression over all rows in the query result; COUNT returns the number of element in the collection. When AggregateFunction is used in xQLQuery, the query will return one single value in the type of Integer. AggregateFunction is used whenever we found a *size()* and *sum()* operation in OCL invariant.

CPParticipant

Every time an association end appears in navigation, a CPParticipant is instantiated corresponds to its Classifier and the association end is stored in xQLStructuralFeature. CPParticipant is also instantiated for the context of the OCL invariant and association class. From database point of view, CPParticipant represent the table in the database. In serialization of xQL, CPParticipant will appear in the *FROM clause*, where each of them will be joined by JOIN expression.

xQLStructuralFeature

Besides holding the association end as already mentioned before, xQLStructuralFeature also holds the information on attribute. Attributes associates with CPParticipant as its ownerEntity. xQLStructuralFeature wraps the information from UML in referredStructuralFeature.

4.3.2 Condition

The condition part which lies in the *WHERE clause* is represented in OpAsCondition as shown in the class diagram below.

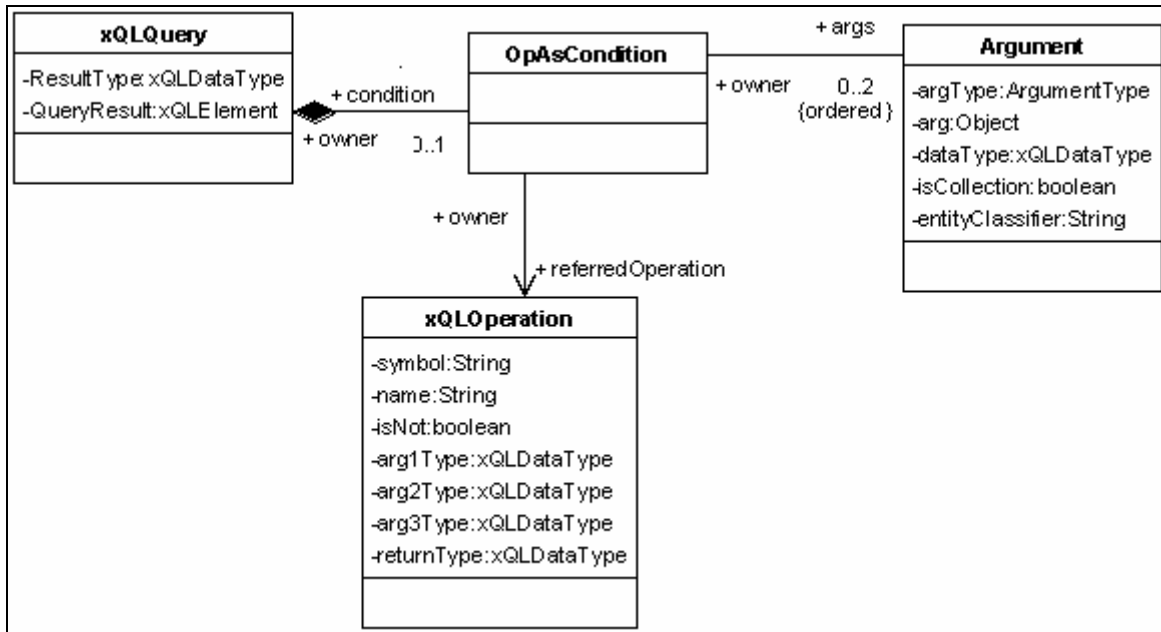


Figure 4.6 OpAsCondition in xQL

OpAsCondition

OpAsCondition is the wrapper of xQLOperation and Argument. If OpAsCondition associates with xQLQuery, it is considered as the root operation. The root operation must be a boolean expression, hence its referredOperation should be an instance of ComparisonOperation or LogicalOperation.

Argument

OpAsCondition can have an arbitrary number of argument depends on xQLOperation. Argument could be a collection or one of the basic types, such as String, Integer, Real and Boolean. ArgumentType must conform to the argument type specified by xQLOperation.

xQLOperation

xQLOperation defines the operation of the condition. It consists of 4 important subclasses: StringOperation, LogicalOperation, ComparisonOperation, and ArithmeticOperation. Each of the subclasses will be described in the following section.

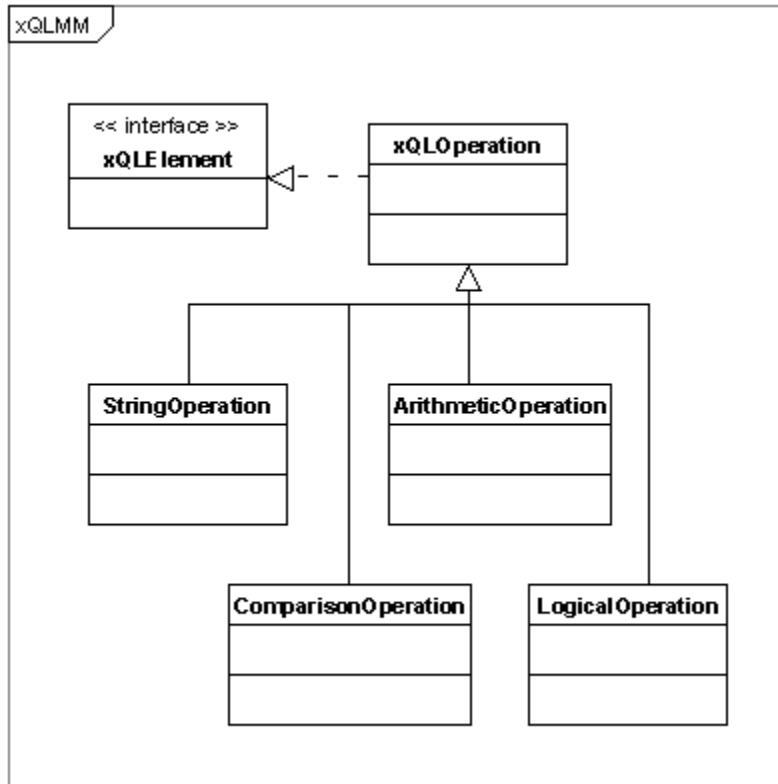


Figure 4.7 xQLOperation and its subclasses

4.4 Operation in xQL

Operation in xQL can be considered similar to operation in HQL. The operation is contained in the referredOperation. OpAsCondition acts as a wrap of operation and its arguments. OpAsCondition may contain a list of argument expressions if the operation is defined to have parameters. In this case, the number and types of the arguments must match the parameters.

Arithmetic operators perform mathematical operations on two expressions of any data types in the numeric datatype category. We use the term 'Numeric' to represent Integer and Real.

Table 4.2 Arithmetic Operation

Operation	Meaning	Data Type		
		Argument 1	Argument 2	Return
+	Addition	Numeric	Numeric	Numeric
-	Subtraction	Numeric	Numeric	
*	Multiplication	Numeric	Numeric	
/	Division	Numeric	Numeric	

Table 4.3 Comparison Operation

Operation	Meaning	Data Type		
		Argument 1	Argument 2	Return
=	Equal to	Numeric/ Boolean/ String	Numeric/ Boolean/ String	boolean
>	Greater than	Numeric	Numeric	
<	Less than	Numeric	Numeric	
>=	Greater than or equal to	Numeric	Numeric	
<=	Less than or equal to	Numeric	Numeric	
<>	Not equal to	Numeric/ Boolean/ String	Numeric/ Boolean/ String	

Table 4.4 Unary Operation

Operation	Meaning	Data Type	
		Argument 1	Return
+	Numeric value is negative	Numeric	Numeric

Table 4.5 String Operation

Operation	Meaning	Data Type			
		Argument 1	Argument 2	Argument 3	Return
Concat ()	Appends two or more literal expressions, attributes values together into one string	String	String	-	String
Lower	Converts a string to all lowercase characters	String	-	-	String
Upper	Converts a string to all lowercase characters	String	-	-	String
Substring	Extracts a portion of string	String	Integer	Integer	String

Table 4.6 Logical Operation

Operation	Meaning	Data Type			Return
		Argument 1	Argument 2	Argument 3	
IN	TRUE if the operand is equal to one of the element in the list	String/ Numeric	Collection	-	Boolean
BETWEEN	TRUE is the operand is within a range	Numeric	Numeric	Numeric	
AND	TRUE if both Boolean expression are TRUE	Boolean	Boolean	-	
OR	TRUE if either Boolean expression are TRUE	Boolean	Boolean	-	
NOT	Reverses the value of any other Boolean operator	Boolean	-	-	

4.5 Well-formedness rules of xQL

xQLQuery

[1] Every xQLQuery must have one condition and at least one entity.

```

context xQLQuery
inv: self.condition->notEmpty()
       inv: self.entities.size(>)0

```

[2] If aggregate's name equal to SUM, the type of QueryResult must be a numeric.

```

context xQLQuery
inv: self.aggregate->notEmpty() and self.aggregate.name = 'SUM'
       implies self.QueryResult.refferedStructuralFeature.type =
       StdlibPrimitiveType::Integer

inv: self.aggregate->notEmpty() and self.aggregate.name = 'SUM'
       implies self.QueryResult.refferedStructuralFeature.type =
       StdlibPrimitiveType::Real

```


- [3] QueryResult is either a type of xQLStructuralFeature or type of CPParticipant.

```
context xQLQuery
inv: self.QueryResult.ocIsTypeOf(xQLStructuralFeature) or
     self.QueryResult.ocIsTypeOf(CPParticipant)
```

- [4] If QueryResult is a type of CPParticipant then ReturnType is a Numeric.

```
context xQLQuery
inv: self.QueryResult.ocIsTypeOf(CPParticipant) implies
     self.ReturnType = xQLDataType::Numeric
```

CPParticipant

- [1] entityVariable of CPParticipants associate with the same owner must be unique.

```
context xQLQuery
inv: self.entities.isUnique(entityVariable)
```

- [2] Except the first and last entities, all entities must have a joinField.

```
context xQLQuery
inv: -
```

- [3] The referredStructuralFeature of joinField of the corresponding CPParticipant must be the type of AssociationEndImpl.

```
context CPParticipant
inv: self.joinField.ocIsTypeOf(AssociationEndImpl)
```

- [4] If the classifier is an instance of AssociationClassImpl then the corresponding CPParticipant does not have a joinField.

```
context CPParticipant
inv: self.ocIsTypeOf(AssociationClassImpl) implies
     self.joinField->isEmpty()
```

- [5] The referredStructuralFeature of involvesAttributes of the corresponding CPParticipant must be the type of AttributeImpl.

```
context CPParticipant
inv: self.involvesAttributes.ocIsTypeOf(AttributeImpl)
```

OpAsCondition

- [1] If the owner of the current OpAsCondition is an instance of xQLQuery, the referredOperation must be an instance of LogicalOperation or ComparisonOperation.

```
context OpAsCondition
inv: self.owner.ocIsTypeOf(xQLQuery) implies
     self.referredOperation.ocIsTypeOf(LogicalOperation) or
     self.referredOperation.ocIsTypeOf(ComparisonOperation)
```

- [2] The argType of each argument of OpAsCondition must adhere to the argument type specified by xQLOperation.

```
context OpAsCondition
inv: -
```

Argument

- [1] if arg is an instance of CPParticipant then argType equals to ENTITY_VAR

```
context Argument
inv: self.arg.oclIsTypeOf(CPParticipant) implies self.argType =
    ArgumentType::ENTITY_VAR
```

- [2] if arg is an instance of OpAsCondition then argType equals to SUB_COND

```
context Argument
inv: self.arg.oclIsTypeOf(OpAsCondition) implies self.argType =
    ArgumentType::SUB_COND
```

- [3] if arg is an instance of IxQLStructuralFeature then argType equals to
STRUCTURAL_FEATURE

```
context Argument
inv: self.arg.oclIsTypeOf(IxQLStructuralFeature) implies
    self.argType = ArgumentType::STRUCTURAL_FEATURE
```

- [4] if arg is an instance of Boolean then argType equals to L_BOOL

```
context Argument
inv: self.arg.oclIsTypeOf(Boolean) implies self.argType =
    ArgumentType::L_BOOL
```

- [5] if arg is an instance of Integer then argType equals to L_INT

```
context Argument
inv: self.arg.oclIsTypeOf(Integer) implies self.argType =
    ArgumentType::L_INT
```

- [6] if arg is an instance of Double then argType equals to L_DOUBLE

```
context Argument
inv: self.arg.oclIsTypeOf(Double) implies self.argType =
    ArgumentType::L_DOUBLE
```

- [7] if arg is an instance of String then argType equals to L_STR

```
context Argument
inv: self.arg.oclIsTypeOf(String) implies self.argType =
    ArgumentType::L_STR
```

- [8] if arg is an instance of Integer then argType equals to L_INT

```
context Argument
inv: self.arg.oclIsTypeOf(Integer) implies self.argType =
    ArgumentType::L_INT
```

- [9] if arg is an instance of HashMap then argType equals to COLL_RANGE

```

context Argument
  inv: self.arg.ocIsTypeOf(HashMap) implies self.argType =
    ArgumentType::COLL_RANGE

```

[10] If arg is an instance of List then argType equals to COLL_ITEM

```

context Argument
  inv: self.arg.ocIsTypeOf(List) implies self.argType =
    ArgumentType::COLL_ITEM

```

Arithmetic Operation

[1] The number of argument is two.

```

context OpAsCondition
  inv: self.referredOperation.ocIsTypeOf(ArithmeticOperation)
    implies self.arguments->size() = 2

```

[2] Arguments must be a type of Integer or Real

```

context OpAsCondition
  inv: self.referredOperation.ocIsTypeOf(ArithmeticOperation)
    implies self.arguments.dataType=xQLDataType::Integer or
    self.arguments.dataType=xQLDataType::Real

```

Comparison Operation

[1] The number of argument is two.

```

context OpAsCondition
  inv: self.referredOperation.ocIsTypeOf(ComparisonOperation)
    implies self.arguments->size() = 2

```

[2] Argument of *greater*, *greater than*, *less*, *less than* is either Integer or Real.

```

context OpAsCondition
  inv: self.referredOperation.name='ge' implies
    self.arguments.dataType = xQLDataType::Real or
    self.arguments.dataType=xQLDataType::Integer

```

```

context OpAsCondition
  inv: self.referredOperation.name='gt' implies
    self.arguments.dataType = xQLDataType::Real or
    self.arguments.dataType=xQLDataType::Integer

```

```

context OpAsCondition
  inv: self.referredOperation.name='le' implies
    self.arguments.dataType = xQLDataType::Real or
    self.arguments.dataType=xQLDataType::Integer

```

```

context OpAsCondition
  inv: self.referredOperation.name='lt' implies
    self.arguments.dataType = xQLDataType::Real or
    self.arguments.dataType=xQLDataType::Integer

```

- [3] If comparison operation is equal or not equal, the argument type could be a String, Boolean or Numeric, both both argument must be on the same type.

```

context OpAsCondition
  inv: self.refferedOperation.name='eq' implies
        self.arguments.dataType = xQLDataType::Real or
        self.arguments.dataType=xQLDataType::Integer or
        self.arguments.dataType = xQLDataType::String or
        self.arguments.dataType=xQLDataType::Boolean

context OpAsCondition
  inv: self.refferedOperation.name='ne' implies
        self.arguments.dataType = xQLDataType::Real or
        self.arguments.dataType=xQLDataType::Integer or
        self.arguments.dataType = xQLDataType::String or
        self.arguments.dataType=xQLDataType::Boolean

context OpAsCondition
  inv: self.refferedOperation.name='eq' implies self.arguments-
        >first().dataType = self.arguments->last().dataType

context OpAsCondition
  inv: self.refferedOperation.name='ne' implies self.arguments-
        >first().dataType = self.arguments->last().dataType

```

String Operation

- [1] If String Operation equals to concat, the number of argument is two and must be a type of String.

```

context OpAsCondition
  inv: self.refferedOperation.name='concat' implies
        self.arguments->size() = 2

  inv: self.refferedOperation.name='concat' implies
        self.arguments.dataType = xQLDataType::String

```

- [2] If String Operation equals to lower or upper, the number of argument is one and must be a type of String

```

context OpAsCondition
  inv: self.refferedOperation.name='lower' implies
        self.arguments->size() = 1

  inv: self.refferedOperation.name='lower' implies
        self.arguments.dataType = xQLDataType::String

  inv: self.refferedOperation.name='upper' implies
        self.arguments->size() = 1

  inv: self.refferedOperation.name='upper' implies
        self.arguments.dataType = xQLDataType::String

```

- [3] If String Operation equals to substring, the number of argument is three. The first argument must be the type of String, the second and third argument must be the type of Integer.

```

context OpAsCondition
  inv: self.refferedOperation.name='substring' implies
        self.arguments->size() = 3

  inv: self.refferedOperation.name='substring' implies
        self.arguments->first().dataType = xQLDataType::String

  inv: self.refferedOperation.name='substring' implies
        self.arguments->at(2).dataType = xQLDataType::Integer

  inv: self.refferedOperation.name='substring' implies
        self.arguments->last().dataType = xQLDataType::Integer

```

Logical Operation

- [1] If the Logical Operation equals to IN, the number of arguments is two and the first argument must be in the same type in the type of collection's element.

```

context OpAsCondition
  inv: self.refferedOperation.name='in' implies self.arguments-
        >size() = 2

  inv: self.refferedOperation.name='in' implies self.arguments-
        >first().dataType = self.arguments->last().dataType

  inv: self.refferedOperation.name='in' implies self.arguments-
        >last().isCollection = true

```

- [2] If the Logical Operation equals to BETWEEN, the number of argument is three and must be in the type of Integer.

```

context OpAsCondition
  inv: self.refferedOperation.name='in' implies self.arguments-
        >size() = 3

  inv: self.refferedOperation.name='in' implies
        self.arguments.oclIsTypeOf(Integer)

```

- [3] If the Logical Operation equals to AND or OR, the number of arguments is two and must be in the type of Boolean.

```

context OpAsCondition
  inv: self.refferedOperation.name='and' implies self.arguments-
        >size() = 2

  inv: self.refferedOperation.name='or' implies self.arguments-
        >size() = 2

```

- [4] If the Logical Operation equals to NOT, the number of arguments is one and must be in the type of Boolean

```
context OpAsCondition
  inv: self.refferedOperation.name='not' implies self.arguments-
    >size() = 1

  inv: self.refferedOperation.name='not' implies
    self.arguments.oclIsTypeOf(Integer)
```

4.6 Summary

xQL is an intermediate step of translating OCL invariant to SQL. It is developed to ease the transformation step. xQL is mainly composed of OCL expression and HQL expression. While xQL borrows some of HQL operations to make expressions, the data types is mainly taken from OCL data types. At the end, to complete the translation steps, we will serialize the abstract syntax of xQL to HQL String. From database point of view xQL is composed of a *SELECT clause*, *FROM clause* and *WHERE clause*. Path and navigation of OCL is translated into *SELECT clause* and *WHERE clause* in xQL, whereas the Boolean expression is taken as *WHERE clause*. Some operation involved in the *WHERE clause* is also discussed in this chapter along with their well-formedness rules.

5 Transformation Recipes: Patterns and Procedures

5.1 The Negation of Boolean Expression

Invariant in OCL Expression is a Boolean expression that should be true for an object during its complete lifetime. In Database system, integrity is checked by querying the database with false condition. Integrity is assured when there query returns no result. For example, take this simple OCL invariant:

```
context Customer
inv: self.cards->size() < 5
```

OCL invariant above states that the current customer can have in maximum 4 cards. In database point of view, this invariant seen as ‘there is not exist a customer that have 5 or more cards’ and is written as follows:

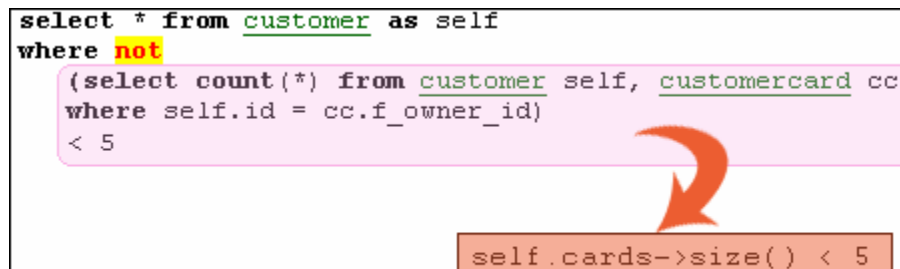
```
select * from customer as self
where not
  (select count(*) from customer self, customercard cc
   where self.id = cc.f_owner_id)
  < 5
;
```

Considering this, to assure the constraint is not broken, we have to select from the database with the negated condition and to hold the integrity, the selection should never return any result. Hence, in database constraint, to put the boolean expression as the condition in *WHERE clause*, we should always negate it first :

```
not self.cards->size() < 5
```

Another crux point is the OCL invariant is taken as the *WHERE clause* in database.

```
select * from customer as self
where not
  (select count(*) from customer self, customercard cc
   where self.id = cc.f_owner_id)
  < 5
```



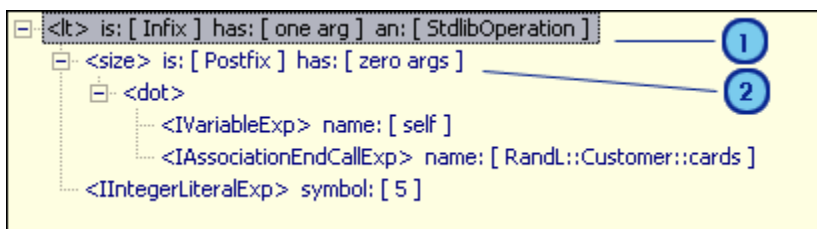
```
self.cards->size() < 5
```

5.2 Operators

Boolean expression can be produced by an operator which return a boolean value. For example, in OCL invariant above boolean operator lesser than ('<') returns a boolean value. Based on this, we can divide operators in two main group: operator which return a boolean value, and operator which return other than boolean value. Operator which return a non-boolean value won't be considered as the candidate for negated operator.

Problem in determining which expression should be negated may occur when invariant contains more than one operator. However, this problem could easily solved if we see the OCL Expression in a operator hierarchical way. Here the OCL Expression is drawn based on the operator, from the most outer operator to the most inner operator.

In above example, we actually have 2 operators, size() and lesser than ('>'). In the hierarchical AST this invariant is drawn as follows:



With lesser than (<lt>) as the outer operator (marked with 1) and size() (<size>) as the inner operator (marked with 2). It is clear that the negated operator is lesser than since it is the root operator. In short, in mapping of OCL invariant to database constraint, root operator should always be negated.

In the AST above, OCL invariants is considered as group of operator and operand. Each operator has one or more operand, and each operand could be another operator (which also consist of operands) and navigation. Operand could also consists navigation with no further applied operator. As in example above, <lt> as the outer operator has 2 operands, call them A and B. A is another operand, size() and B is the operand 5. Operator size() has 1 operand which is cards.

5.3 Mapping Procedures

Mapping is divided into two main groups, first is the translation of navigation, and the second is translation of operation:

a. Navigation

Navigation is translated to *SELECT clause* and *FROM clause*.

Collection type of OCL are divided into set and bag, however in database, data are always in the form of tuples.

b. Operation

Simple operation such as arithmetic or boolean operation are easy to map since their direct counterpart is available in xQL. However, mapping of operation for collection type is quite challenging, since some of the operations do not have a direct counter part in xQL. To overcome this problem, we need to transform the operation to another equivalent OCL operation. For example *isUnique* operation is first transformed to its equivalent expression using *forAll(expr)* operation before being translated into xQL.

Complete list of mapping patterns can be found in the following section.

5.4 Mapping Patterns

5.4.1 Navigation

Operands in OCL could be divided into 3 main types: Classifier, StructuralFeature and Literal. A Classifier is equivalent to a Class, StructuralFeature is equivalent to an Attribute, and Literal could take the type of String, Numeric, Boolean or a Collection. Since literal is transformed also as literal, what left is the transformation of Classifier and StructuralFeature.

In xQL, Classifier, Association End, and Association Class are mapped to CPParticipant. However, unlike OCL, CPParticipant is the occurrence of the Classifier, so there could be a possibility that we have more than one instance of CPParticipant correspond to the same Classifier, as shown in the following example:

```
context LoyaltyProgram
inv lp_2: levels->includesAll( Membership.currentLevel )
```

Above invariant states that the service level of each membership must be a service level known to the loyalty program for which the invariant holds. For this invariant, following CPParticipants are instantiated:

- *loyaltyprogram*: instantiated from the context LoyaltyProgram, so it is referring to LoyaltyProgram Classifier.

- *i_Membership*: instantiated from the association class Membership, so it is referring to Membership Classifier.
- *servicelevel_11857510*: instantiated from the association end levels, which its Classifier is ServiceLevel.
- *servicelevel*: instantiated from the association end currentLevel, which its Classifier is also ServiceLevel.

The AST from above example is depicted in Figure 5.1.



Figure 5.1 AST of OCL Invariant lp_2: levels->includesAll(Membership.currentLevel)

Attributes from OCL is mapped to xQLStructuralFeature. xQLStructuralFeature holds the information about the attribute's name and the owner.

5.4.2 Operation

Based on the type of expression, OCL Operation can be divided into 2 main groups, operation on basic types and operation on collection type. Besides these two main categories, there also exists user-defined operation. The last mentioned type of operation is out of this project scope.

5.4.2.1 Operation on Basic types

Operation on primitive data type is divided into several sections based on the input type: Boolean type, Numeric type and String type.

Table 5.1 Standard operations for Boolean type

OCL	xQL	Result Type
a or b	a or b	boolean
a and b	a and b	
a xor b	(a or not b) and (not a or b)	
not a	not a	
a = b	a = b	
a <> b	a <> b	
a implies b	not a or b	

For operation on Boolean type, the direct counterpart can be found almost for all operation except exclusive or and implies. However, the translation is still possible by using the equivalent operation.

Table 5.2 Standard operations for Numeric type

OCL	xQL	Result Type
a = b	a = b	boolean
a <> b	a <> b	
a < b	a < b	
a > b	a > b	
a <= b	a <= b	
a >= b	a >= b	
a + b	a + b	numeric
a - b	a - b	
a * b	a * b	
a / b	a / b	

Table 5.3 Standard operations for String type

OCL	xQL	Result Type
string.concat(string)	string1 string2	String
string.toLowerCase()	lower(string)	
string.toUpperCase()	upper(string)	
string.substring(int,int)	substring(string, int,int)	

5.4.2.2 Operations on Collection Types

OCL defines many operations on collection types. These operations are specifically meant to enable a flexible and powerful way of projecting new collections from existing ones. The different constructs are described in the following sections.

Standard Operations

To look at the concise version of the mapping please refer to Table 5.4.

Table 5.4 Standard operations on all collection types

OCL	xQL	Result Type
collection->count(object)	select [count(object)] from ... where object collection [not] in collection	integer
collection ->excludes(object)		boolean
collection ->excludesAll(collection)		
collection ->includes(object)		
collection ->includes All(collection)		
collection ->isEmpty()		
collection ->notEmpty()	count(*) from .. <> 0	integer
collection ->size()	count(*)	
collection ->sum()	sum(collection)	

The count, excludes, excludesAll, includes, includesAll operation

The *count*, *excludes*, *excludesAll*, *includes*, and *includesAll* operation actually has the same basic operation; they check whether the object or collection in the body parameter exists or not exists in the collection. All of these operations return a boolean value except count operation which return the number of occurrences of the object in the collection. In xQL the verification of whether a certain object exists in the corresponding collection is solved with IN operation. IN returns true if the object exists in the corresponding collection, and false coverseely. IN operation can also be used to write certain types of subqueries.

We have the same basic recipe for *count*, *excludes*, *excludesAll*, *includes*, *includesAll* operation. The collection appeared in left side of the arrow is mapped as an xQLQuery and become the second argument of IN operation. The body of those operations is mapped as another xQLQuery and become the outer query where IN is used in the WHERE clause. First argument of IN operation is QueryResult of the outer xQLQuery. For *excludes* and *excludesAll* we use NOT IN operation, and for COUNT operation, we use the aggregate function COUNT in the Query Result of the outer xQLQuery.

For example, given that we have the following invariant which specifies that the actual service level of a membership must be one of the service levels of the program so which the membership belongs:

```
context Membership
inv: programs.levels ->includes(currentLevel)
```

For the body of *includes*, an `xQLQuery` is instantiated with `membership` and `servicelevel_33199009` as its `CPParticipants`. For the first argument of `IN` operation, the `QueryResult`, `servicelevel_33199009`, is used and for the second argument another `xQLQuery` is instantiated with `membership`, `loyaltyprogram`, and `servicelevel` as its `CPParticipants`.



Figure 5.2 AST of `IN` operation

And the serialization of the `xQL` model above into `HQL` String after we negate the root operation is as follows:

```

select servicelevel_13596360.id
from Membership membership
  join membership.f_currentLevel servicelevel_13596360
where not servicelevel_13596360
  in
  (select servicelevel.id
   from Membership membership
    join membership.f_programs loyaltyprogram
    join loyaltyprogram.f_levels servicelevel)
  
```

The *isEmpty* and *notEmpty* Operation

The *isEmpty* operation returns true if the collection contains no elements, and *notEmpty* operation returns true if the collection contains at least one element. So actually these two operations are the reverse operation of each other. To map this operation we have to query the database count the result with aggregate function. For *isEmpty* the result should be 0 and for *notEmpty* the result should be not equal to 0.

For example, given that we have the following OCL invariant which states a loyalty program does not deliver any Service to its customer.

```

context LoyaltyProgram
inv: partners.deliveredServices->isEmpty()
  
```

For this invariant, first we instantiate an `xQLQuery` object which has `loyaltyprogram`, `i_ProgramPartner` and `service` as its `CPPParticipant`. This `CPPParticipant` is taken from the context `LoyaltyProgram`, association end `partners` and association end `deliveredServices`. The last `CPPParticipant` is taken as the `QueryResult` follows with the application of aggregate function `COUNT` on it. The next step is comparing the `xQLQuery` with `Literal Integer 0`. For `isEmpty` we compare the `xQLQuery` with equal (`'='`) operator, and for `notEmpty` we compare the `xQLQuery` with not equal (`'<>'`) operator. The result of this comparison is a boolean value.

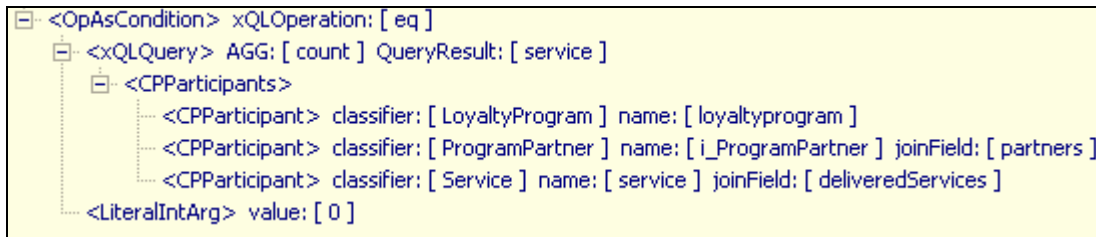


Figure 5.3 Example for `isEmpty` and `notEmpty` operation

The *size* Operation

The *size* operation returns the number of elements in the collection. In xQL, *size* is mapped by adding the aggregate function `COUNT` in the `QueryResult`.

For example given that we have the following OCL invariant which states that the current customer can have in maximum 4 cards:

```
context Customer
inv: self.cards->size() < 5
```

size operation is mapped by the aggregate function `COUNT` in the inner `xQLQuery` as shown in the figure below:

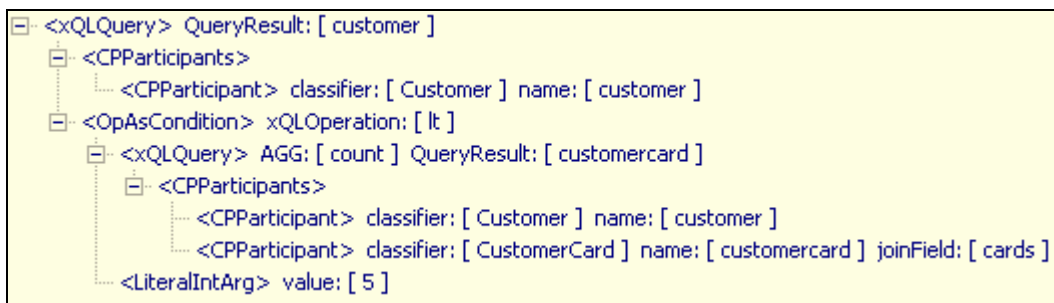


Figure 5.4 Example for *size* Operation

And the serialization of the xQL model above into HQL String after we negate the root operation is as follows:

```
select customer.id
from Customer customer
where not
    (select COUNT(customercard.id)
     from Customer customer
     join customer.f_cards customercard ) < 5
```

The *sum* Operation

The sum operation returns the addition of all elements in the collection. The elements must be of a type supporting addition (such as Real or Integer). In xQL, *sum* is mapped by adding the aggregate function SUM in the QueryResult.

Loop Operations or Iterators

A number of standard OCL operations enable you to loop over the elements in a collection. These operations take each element in the collection and evaluate an expression on it. Loop operations are also called *iterators* or *iterator* operations. Every loop operation has an OCL expression as parameter. This is called the body, or body parameter, of the operation. The following sections explain each of the loop operations in more detail. Table 5.5 shows an overview of the loop operations defined on the collection types.

Table 5.5 Loop Operations or Iterators (IteratorExp)

OCL	HQL	Return Type
collection->collect(element)	<i>navigation</i>	collection
collection->collectNested(element)	<i>navigation</i>	
collection->any(boolean expr)	select *	element
collection->reject(boolean expr)	from <i>table</i>	collection
collection->select(boolean expr)	where [not] <i>boolean exp</i>	
Collection->forAll(boolean expr)	(select count(*) from <i>table</i>) = (select count(*) from <i>table</i> where <i>boolean expr</i>)	boolean
collection->isUnique(element)	use forAll	
collection->one(boolean exp)	(select count(*) from <i>table</i> where <i>boolean exp</i>) = 1	
Collection->exists(boolean expr)	(select count(*) from <i>table</i> where <i>boolean expr</i>) > 1	boolean

The *collect* and *collectNested* Operation

The *collect* operation iterates over the collection, computes a value for each element of the collection, and gathers the evaluated values into a new collection. We consider *collect* as a part of navigation, since *collect* is mostly used to navigate from the source object to destination object. *Collect* can also be written using dot notation ('.').

OCL differentiates iteration over a set and iteration over a bag with *collect* and *collectNested*. However, in xQL we consider *collect* and *collectNested* as the same operation. In xQL we see it as list of CPParticipants, and later when it is serialized; it will be the join between tables.

The *any*, *select*, and *reject* Operation

The basic idea behind these three operations is they enables us to specify a selection from the original collection based that fulfill the condition stated in the parameter. In xQL, the condition stated in the body parameter will be mapped as OpAsCondition. The operand and operator of OpAsCondition depend on the boolean expression in the body parameter. In order to return an element or a collection, we have to make the outer xQLQuery and put the OpAsCondition as its condition.

While *any* and *select* return a selection from the original collection which fulfills the condition, *reject* return all elements from the collection for which the expression evaluates to false. The difference is reflected by the use of *not* operation. This pattern is the basic translation of other iterator exp such as *forAll*, *exists* and *one*.

Again, taking Royal and Loyal model as an example, the following expression from the context of LoyaltyProgram results in a loyalty account randomly picked from the set of accounts in the program that have a number lower than 10,000:

```
self.Membership.account->any( number < 10000 )
```

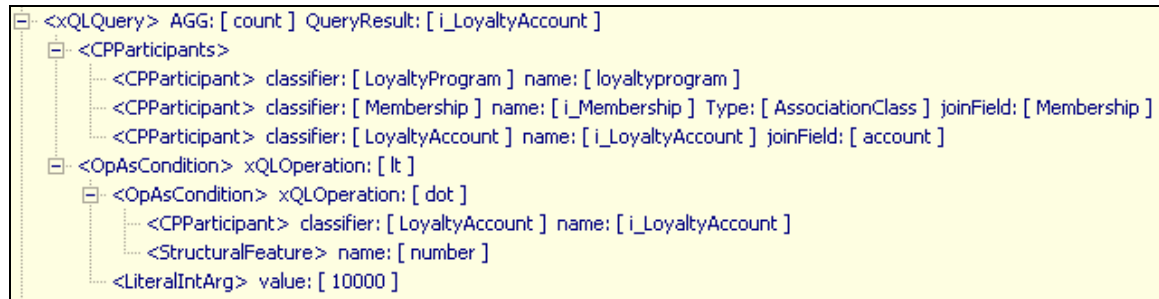



Figure 5.5 Example for any Operation

The *exists* Operation

Exists operation is used to specify whether there is at least one object in a collection for which a certain condition holds. Using the basic translation for iterator expression described in the previous operation, in *exists* Operation; we add an aggregate function *count*. Next step is instantiating an *OpAsCondition* which will hold the *xQLQuery*, literal integer 0 and greater than as the comparison operation. *Exists* returns true if the number of result is greater than zero, otherwise returns false.

To specify whether there is at least one service with the name = 'basic' in the corresponding LoyaltyProgram, we define the following OCL invariant:

```

context LoyaltyProgram
inv lp_3: self.levels->exists(name = 'basic')
  
```

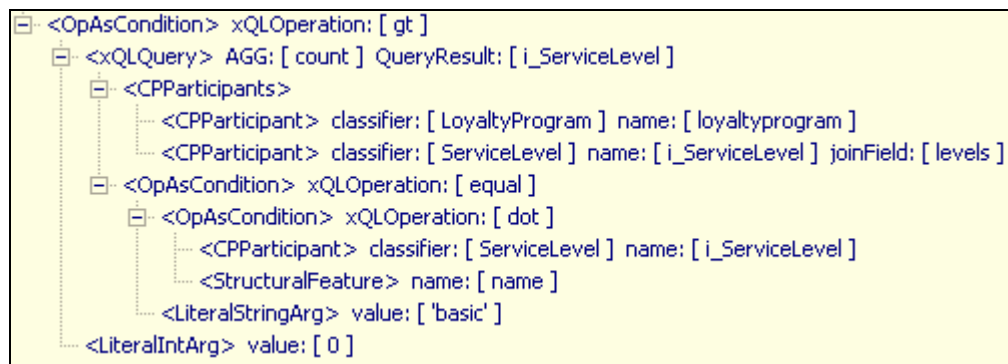


Figure 5.6 Example of exists Operation

The *forAll* Operation

We often want to specify that a certain condition must hold for all elements of a collection. The *forAll* operation on collections can be used for this purpose. The result of the *forAll* operation is a boolean value. It is true if the expression is true for all elements of the collection. If the

expression is false for one or more elements in the collection, then *forall* results in false. Using the operations we have in xQL, the equivalent operation for *forall* operation is to compare the number of results of xQLQuery with and without the condition. If the number of results is the same, it means that the condition is valid for all the elements in the collection.

Again, using the basic translation for iterator expression, we will instantiate two xQLQuery, one xQLQuery with OpAsCondition and the other without OpAsCondition. Next step is adding the aggregate function *count* to both of xQLQuery. Last step is comparing the result of both xQLQuery with equal comparison operator.

Given that we want to specify that the participant of the corresponding program is open only for male:

```
context LoyaltyProgram
  inv : self.Membership.participants->forall(isMale = true)
```

For above example, we have the OpAsCondition with equal as its xQLOperation and two aggregated xQLQuery, one with the condition `isMale = true` and the other without the condition.

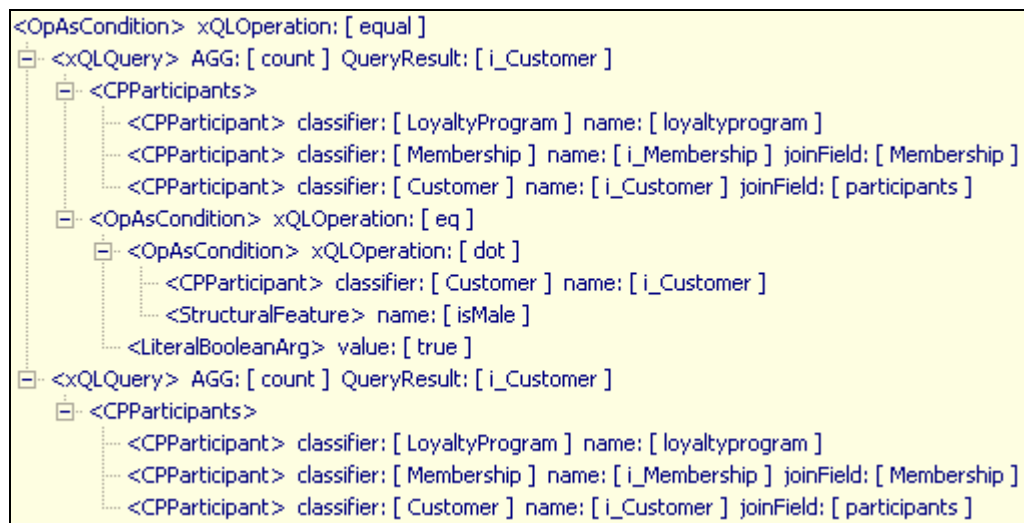


Figure 5.7 Example of forall Operation

The *isUnique* Operation

Quite often in a collection of elements, we want a certain aspect of the elements to be unique for each element in the collection. For instance, in a collection of employees of a company, the employee number must be unique. To state this fact, we can use the *isUnique* operation. The parameter of this operation is usually a feature of the type of the elements in the collection. The result is either true or false. The operation will loop over all elements and compare the values by

calculating the parameter expression for all elements. If none of the values is equal to another, the result is true; otherwise, the result is false.

To resolve the translation of `isUnique` we will firstly translate it to its equivalent operation using nested `forall` [15]. The latter translation is following the pattern of `forall`.

The *one* Operation

The *one* operation gives a boolean result stating whether there is exactly one element in the collection for which a condition holds. The body parameter of this operation, stating the condition, is a boolean expression. If there is exactly one such element, then the result is true; otherwise, the result is false.

Following the basic pattern of iterator expression, we take the resulting `xQLQuery` and add an aggregate function `count` in the `QueryResult`. Next, we instantiate an `OpAsCondition` with `equal` operator as its `xQLOperation`, taking `xQLQuery` as its first argument and `Literal Integer 1` as the second argument.

Taking the number attribute of the `LoyaltyAccount` class in the R&L system as an example, the following invariant states that there may be only one loyalty account that has a number lower than 10,000:

```
context LoyaltyProgram
inv: self.Membership.account->one( number < 10000 )
```

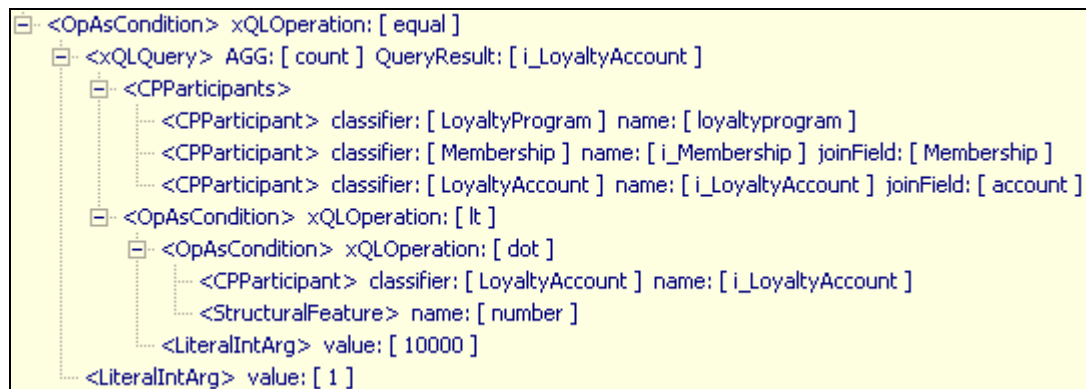


Figure 5.8 Example of one Operation

5.4.2.3 Outer Query

The translation often results not in xQLQuery, hence for this type of result an outer query must be made. For example the translation of *one* operation as mentioned above. Translating merely the one operation will result in OpAsCondition, since after the xQLQuery with the condition has been made we have to aggregate the result and check whether the value is equal to one. If one is the root operation, we have to create an extra outer query to hold the OpAsCondition, so the final result is as shown in the following picture:



Figure 5.9 Translation of one Operation with outer query

5.5 Summary

Translation of OCL to xQL could be divided into two major parts: translation of the navigation and translation of operation. The navigation will be mapped into CPParticipant and xQLStructuralFeature. The OCL operation will be mapped into xQL operation. Apart from the translation recipe we have to deal also with the creation of view where in order to hold the integrity constraint we have to query the database with the negated condition. The integrity is ensure where the query return no result.

6 Introduction to SQL Generator in OctopusEE

To run OctopusEE in Eclipse, please refer to OctopusEE Configuration in Appendix C. However an extra setup is required to generate SQL from OCL invariant in OctopusEE.

6.1 Configuration of build path

To run SQL Generator in OctopusEE, the following jar files is required in your build path. To prevent compatibility mismatch, the version of each jar files is provided as well.

- *hibernate3.jar* (version: 3.2.0.cr3)
- *hibernate-annotations.jar* (version: 3.2.0.CR1)
- *hibernate-entitymanager.jar* (version: 3.1beta4)
- *ejb3-persistence.jar*
- *hibernate-tools.jar*

In this project we use Oracle Database 10g Express Edition with following JDBC driver:

- *classes12.jar* (Oracle JDBC Driver version - 10.1.0.4.0)

6.2 Generated Files

In addition to java generated code in folder `src`, SQL Generator in OctopusEE creates additional folder named `hql`. In this folder, HQL files - the translation result of OCL invariant - are placed. Each `hql` file in `hql` folder correspond to `ocl` file in `expressions` folder: `Customer.hql` is an HQL script corresponds to OCL invariant in `Customer.ocl`, `CustomerCard.hql` is an HQL script corresponds to OCL invariant in `CustomerCard.ocl`, and so on.

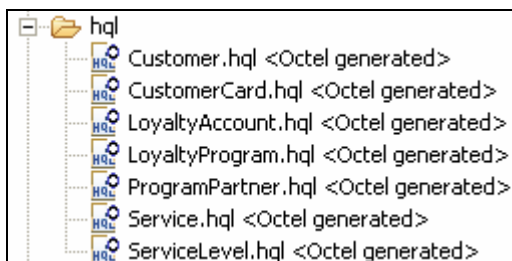


Figure 6.1 additional hql folder

SQL Generator in OctopusEE also output a new “`SQLGenerator.java`” in “`utilities`” package, which is used for generating SQL script out of HQL files. The dialect of generated SQL

script depends on the JDBC driver specified in hibernate configuration file. The translation of HQL to SQL will be written in one single file, `viewscript.sql`, and placed in the root path.

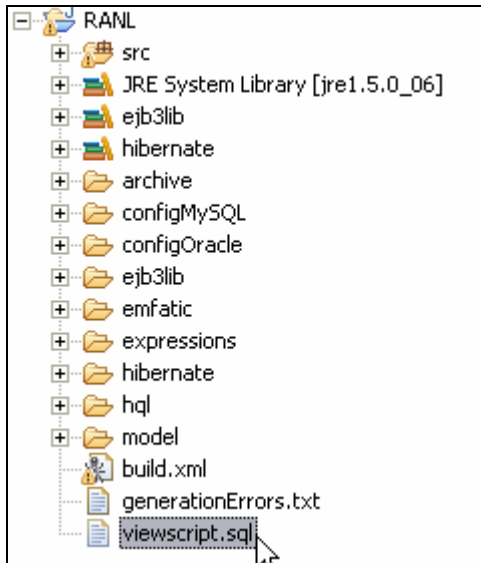


Figure 6.2 Location of `viewscript.sql`

6.3 Testing

To test the SQL script, a database corresponds to the project should exist beforehand. To generate the database schema, please refer to section 6.4.1 in Appendix C. The first step to test the SQL script is by uploading the script to Oracle.

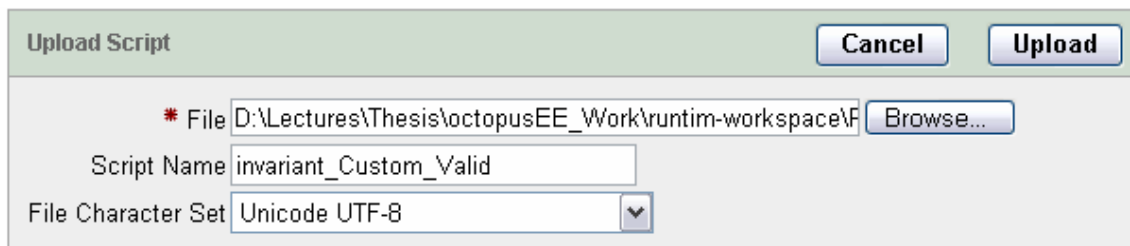


Figure 6.3 Upload Script in Oracle

After the script is run and compiled we can see the created views as depicted in the following picture:

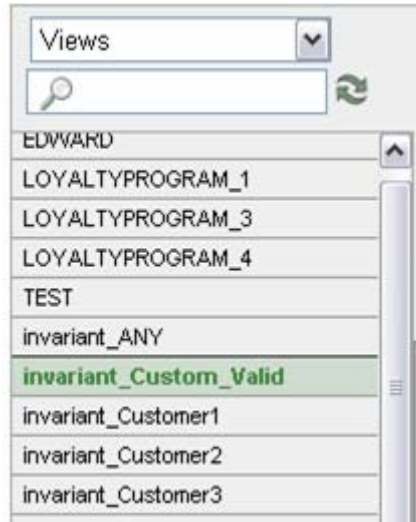


Figure 6.4 Created View from SQL Script

The new created views are then used by triggers which evaluate the constraints after each critical data manipulation operation. When any constraint violation is found, the trigger should rollback the current transaction and sends an appropriate error message to the invoking application.

To gain a better understanding of this concept, we will take one example of OCL invariant which states that every customer must have at least one valid card:

```
context Customer
  inv Custom_Valid: cards->select( valid = true )->size() > 1
```

For above invariant we have following SQL script:

```
create or replace force view
"invariant_Custom_Valid" as
select
  customer0_.id as col_0_0_
from
  Customer customer0_
where
  (select count(f_cards2_.id)
   From Customer customer1_
   inner join CustomerCard f_cards2_
     on customer1_.id=f_cards2_.f_owner_id
   where
     f_cards2_.f_valid<>1
  )<=1;
```

Next, we evaluate the given constraint by creating the following trigger which will evaluate every time a manipulation occurred in table Customer:

```
CREATE OR REPLACE TRIGGER "CUSTOMER_valid"
AFTER
insert or update or delete on "CUSTOMER"
DECLARE
    D NUMBER;
BEGIN
    select count(*) into D from invariant_Custom_Valid;
    IF (D > 0) THEN
        RAISE_APPLICATION_ERROR(-20000, 'constraint
violated');
    END IF;
END;
```

If data, which will violate the constraint, is entered into database, an error message will be raised and the transaction will be rolled back.

```
error ORA-20000: constraint violated ORA-06512: at "NOVI.CUSTOMER_valid", line 6 ORA-04088
```

For the conciseness of the report, the given example of OCL invariant above only involves one table in evaluating a business rule. However, a more complex integrity view could use more than one table of the database to evaluate a business rule. As a result, the constraint evaluation must be done after manipulation of all of these tables.

6.4 Summary

In this chapter, we introduce SQL Generator in OctopusEE and how to configure it in Eclipse as well as how to upload the generated SQL script in RDBMS, in our case Oracle.

7 Summary and Outlook

7.1 Summary

In this project, we have reported our approach to translate OCL invariants as constraints in database systems by combining object oriented query language with the use of view and trigger. Along with the final release of EJB 3.0, Java Community Press introduces Java Persistence Query Language. The Java Persistence query language, also known as EJB3QL, can be compiled to a target language, such as SQL of a database.

By utilizing the enhanced power of EJB3QL, we are able to simplify the process of specifying OCL invariant as the integrity constraint in database systems and keep using relational databases. With this approach we could gain some advantages:

- **Joining Associations.** EJBQL introduces path expression, an identification variable followed by the navigation operator (.) and a state-field or association-field [14]. Utilizing path expression, we do not need to specify join condition explicitly. With path expression, EJBQL has enough information in the mapping document to then deduce the table join expression. This helps make mapping navigation in OCL invariant easier and in the same time make queries less verbose and more readable.
- **Polymorphic Queries.** By default, all queries in EJB3QL are polymorphic. That is, the FROM clause of a query designates not only instances of the specific entity class(es) to which it explicitly refers, but subclasses as well [14]. We might define a rule which involves subclasses that, following the table per class hierarchy approach [10], are not mapped into a table. With polymorphic queries, no matter what approach is taken in mapping class inheritance, we can swiftly write a query in EJB3QL.

The use of view and trigger offers some advantages: view is supported by all DBMS vendors, and it also allows evaluating a complex condition involving arbitrary number of tables. This ability substitutes the task of *assertion* and fulfills the vital part of integrity constraint.

We have shown that it is possible to specify OCL invariants as constraints in database systems by combining object oriented query language with the use of view and trigger. Our goal to translate the OCL invariants to EJB3QL has been achieved with some limitations. Differences in operation behavior of OCL and EJB3QL cause some operation in OCL cannot be translated into EJB3QL,

such as *iterate*. However, this seems not to be a serious problem, since in practical OCL specification the *iterate* operator is rarely used [2], and all OCL constructs derived from *iterate* (like *forAll* and *select*) can be mapped properly. The complete list of unmapped operation can be found in Appendix A. Another limitation comes from our dependency on class-to-table mapping technique taken by OctopusEE. Two main limitations in this case are (1) To navigate through classes which are linked with association class, the navigation class should be explicitly mentioned in the navigation paths, and (2) we cannot translate OCL invariant which involves `@Transient` datatype, since it is not mapped into a column in database.

7.2 Further Work

In this project, we only exploit the use of OCL *invariant* to define integrity constraint in relational database. Further works can be done in completing the constraint by using *precondition* and *postcondition* and *guard*. Moreover, the generation of SQL construct can be extended from creating a view construct to automatically create a trigger for each involved tables in the view construct.

In the area of EJB3QL, a further work can be done in developing a complete Metamodel of EJB3QL. In our xQL model, we only use the subset of EJB3QL, thus although we are able to serialized a well-formed EJB3QL out of xQL model, the result is limited. For example, translating OCL invariant to EJB3QL syntax, we need a single `QueryResult` in the *SELECT clause*, thus in our xQL model, the association between *SELECT clause* and `QueryResult` is one-to-one. In the EJB3QL specification, a *SELECT clause* can consists of arbitrary `QueryResult` including arbitrary aggregate function.

Appendix A: Unmapped Operation

Some operations are not mapped into xQL syntax, for several reasons:

- Counterpart of corresponding OCL operation is not available in declarative EJB3QL, such as *iterate(...)* operation.
- Some OCL operations are not used in invariant. For example: *append()* and *prepend()* is used to add an element to a sequence as the last or first element, respectively. This kind of operation is not used to check integrity constraint in SQL and need not to be translated.
- Some OCL operations are applicable only to Set or Bag. In EJB3QL we do not have such collection. So these types of operations not need to be translated.

List of unmapped operation is shown in table below:

Operation	Description
append(object)	Add an element to a sequence as the last element
asBag()	Applying <i>asBag</i> on a sequence or <i>asSet</i> on an ordered set means that the ordering is lost.
asOrderedSet()	Applying <i>asOrderedSet</i> on a set or bag means that the elements are placed randomly in some order in the result.
asSequence()	Applying <i>asSequence</i> on a set or bag means that the elements are placed randomly in some order in the result.
at(index)	The <i>at</i> operation results in the element at the given position.
excluding(object)	The <i>excluding</i> operation results in a new collection with an element removed from the original collection.
first()	The <i>first</i> operations result in the first elements of the collection.
flatten()	The <i>flatten</i> operation changes a collection of collections into a collection of single objects.
including(object)	The <i>including</i> operation results in a new collection with one element added to the original collection.
indexOf(object)	The <i>indexOf</i> operation results in an integer value that indicates the position of the element in the collection.

Operation	Description
insertAt(index, object)	The insertAt operation results in a sequence or ordered set that has an extra element inserted at the given position.
last()	The last operations result in the last elements of the collection.
prepend(object)	The prepend operations add an element to a sequence as the first element.
subSequence(lower, upper)	The subSequence operation may be applied to sequences only, and results in a sequence that contains the elements from the lower index to the upper index, inclusive, in the original order.
symmetricDifference(coll)	The symmetricDifference operation results in a set containing all elements in the set on which the operation is called, or in the parameter set, but not in both.
subOrderedSet(lower, upper)	The subOrderedSet operation may be applied to ordered sets only. Its result is equal to the subSequence operation, although it results in an ordered set instead of a sequence.
iterate(...)	Iterates over all elements in the source collection
let	Defines local variable to represent the value of the sub-expression
a.max(b)	Arithmetic operation
a.min(b)	Arithmetic operation

Appendix B: Configuration of OctopusEE²

7.3 Requirements

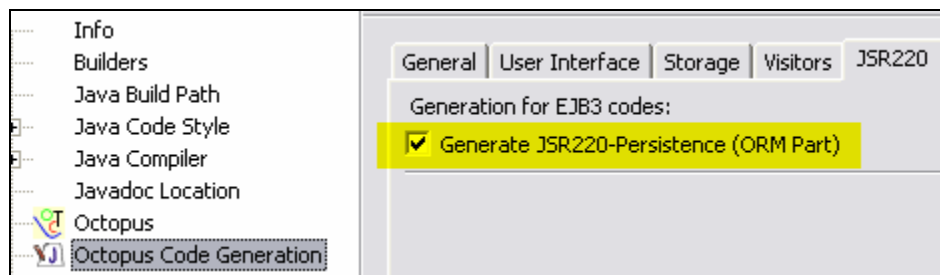
EJB3 runs on Java 1.5 VM or above, thus the installation of Java 1.5 is an essential requirement. Besides this, an Entity Manager needs to be set for providing the EJB3 persistence environment (also referred to as “persistence engine” or “ORM engine”). We recommend using “Hibernate Entity Manager” which is founded on “Hibernate Core” and “Hibernate Annotation”.

7.4 Setup in Eclipse

7.4.1 Configuration in Property page

In the “Properties” page of your “octopus project”, make sure that the “JDK Compiler compliance level” is set to “5.0”

In order to let OctopusEE generate EJB3 artifacts, you need to turn the option for EJB3 generation on. This switch can be found under “Properties”→“octopus code generation”→“JSR220”



7.4.2 Configuration of build path:

The following .jar files are needed in the build path of your “octopus project”:

² Taken from Generation of EJB3 Artifacts in a Modeling Platform, master thesis by Xinhua Gu.

- *hibernate-entitymanager.jar* from root directory of Hibernate Entity Manager package.
- *ejb3-persistence.jar* (the core library for EJB3 persistence) and
- *hibernate-annotation.jar* to be found in the lib directory of the Hibernate Entity Manager installation
- *hibernate3.jar* from root directory of Hibernate Core package. Add the whole lib directory in build path. We also need *hibernate-tool.jar* from Hibernate Tool package for generation of DDL file. In order to get *hibernate-tool.jar* working, some additional jars are required, you can take these information from “chapter 4 Ant tools” of the Hibernate Tool document.

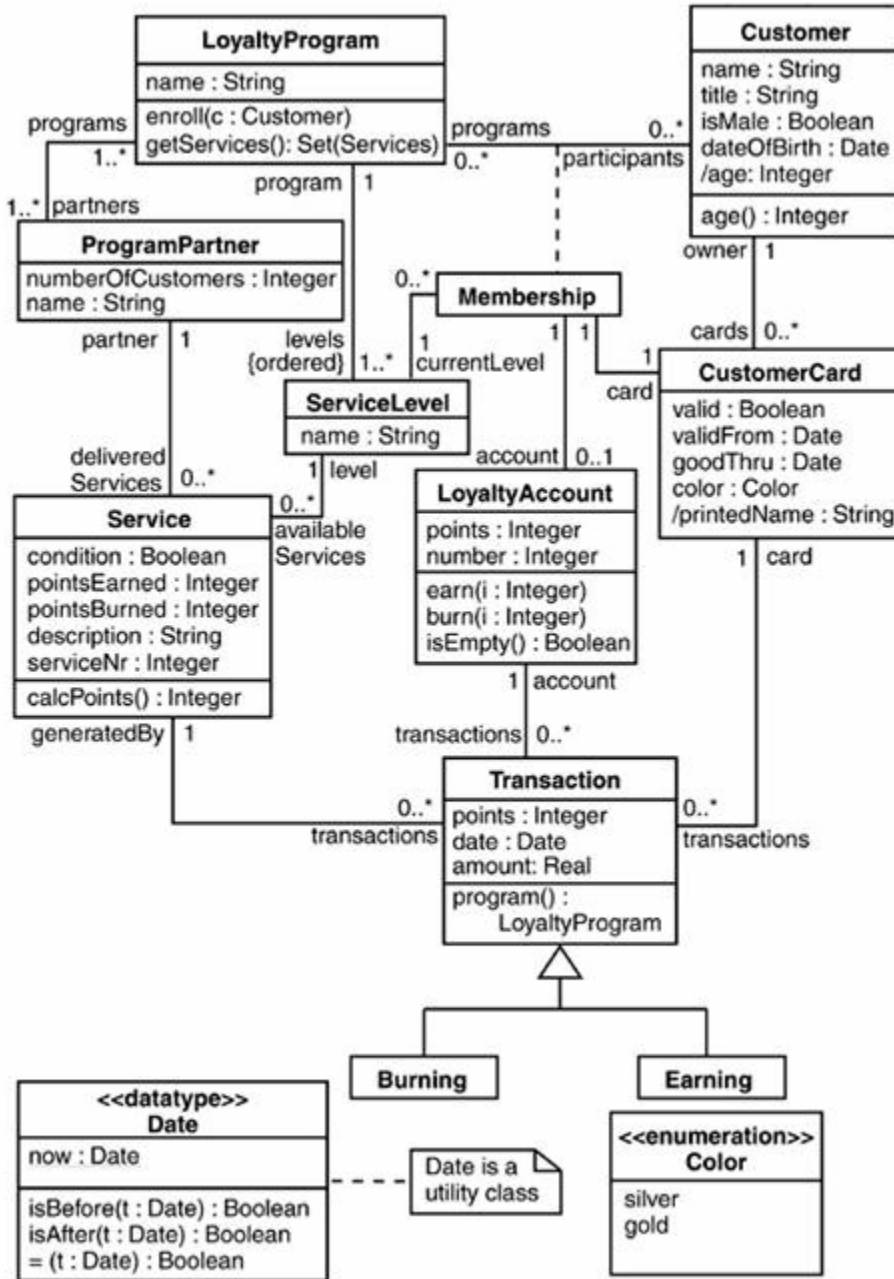
7.5 Generated files

The original Octopus distribution will generate an “utilities” package in addition to the package defined in the .uml file. OctopusEE will output a new “DDLGenerator.java” in “utilities” package. This file is used for generating the database schema by means of DDL. It can be configured to let the DDL be executed directly by the DBMS during code generation.

A log4j configuration file “log4j.properties” is created under “src” directory. From default configuration, log information will be displayed in console.

Furthermore, OctopusEE will generate two more packages. One is “META-INF” and the other is “test”. The “META-INF” folder contains the XML configuration files for the project. They are “hibernate.cfg.xml” which is used for DDL generation and “persistence.xml” which contains the ORM mapping information to be used by the Entity Manager at runtime. In the “test” package, a simple JUnit file is created.

Appendix C: The Royal and Loyal Model



Appendix D: Database Schema or Royal and Loyal

```

CREATE TABLE "CUSTOMER"
  ( "ID" NUMBER(19,0) NOT NULL ENABLE,
    "F_NAME" VARCHAR2(255 CHAR),
    "F_TITLE" VARCHAR2(255 CHAR),
    "F_ISMALE" NUMBER(1,0) NOT NULL ENABLE,
    "F_GENDER" NUMBER(10,0),
    PRIMARY KEY ("ID") ENABLE
  )
/
CREATE TABLE "SERVICELEVEL"
  ( "ID" NUMBER(19,0) NOT NULL ENABLE,
    "F_NAME" VARCHAR2(255 CHAR),
    "I_INDEX1" NUMBER(10,0) NOT NULL ENABLE,
    PRIMARY KEY ("ID") ENABLE
  )
/
CREATE TABLE "MEMBERSHIP"
  ( "ID" NUMBER(19,0) NOT NULL ENABLE,
    "F_MEMBERSHIPATTR" NUMBER(10,0) NOT NULL ENABLE,
    "I_INDEX1" NUMBER(10,0) NOT NULL ENABLE,
    "F_CURRENTLEVEL_ID" NUMBER(19,0),
    "F_PARTICIPANTS_ID" NUMBER(19,0),
    "F_PROGRAMS_ID" NUMBER(19,0),
    PRIMARY KEY ("ID") ENABLE,
    CONSTRAINT "FK26EF63F6830FFA37" FOREIGN KEY ("F_PROGRAMS_ID")
      REFERENCES "LOYALTYPROGRAM" ("ID") ENABLE,
    CONSTRAINT "FK26EF63F68D72F9A6" FOREIGN KEY ("F_PARTICIPANTS_ID")
      REFERENCES "CUSTOMER" ("ID") ENABLE,
    CONSTRAINT "FK26EF63F67145822C" FOREIGN KEY ("F_CURRENTLEVEL_ID")
      REFERENCES "SERVICELEVEL" ("ID") ENABLE
  )
/
CREATE TABLE "CUSTOMERCARD"
  ( "ID" NUMBER(19,0) NOT NULL ENABLE,
    "F_VALID" NUMBER(1,0) NOT NULL ENABLE,
    "F_COLOR" NUMBER(10,0),
    "F_MYLEVEL_ID" NUMBER(19,0),
    "F_OWNER_ID" NUMBER(19,0),
    "F_MEMBERSHIP_ID" NUMBER(19,0),
    PRIMARY KEY ("ID") ENABLE,
    CONSTRAINT "FK3F6DA42E115A15A8" FOREIGN KEY ("F_MEMBERSHIP_ID")
      REFERENCES "MEMBERSHIP" ("ID") ENABLE,
    CONSTRAINT "FK3F6DA42E403A3B45" FOREIGN KEY ("F_OWNER_ID")
      REFERENCES "CUSTOMER" ("ID") ENABLE,
    CONSTRAINT "FK3F6DA42E7F425EB1" FOREIGN KEY ("F_MYLEVEL_ID")
      REFERENCES "SERVICELEVEL" ("ID") ENABLE
  )
/
CREATE TABLE "IC1"
  ( "ID" NUMBER(10,0) NOT NULL ENABLE,
    "SEQUENCE" NUMBER(10,0) NOT NULL ENABLE,
    "OWNER_ID" NUMBER(19,0),
    "ITEM_ID" NUMBER(19,0),
    PRIMARY KEY ("ID") ENABLE,
    CONSTRAINT "FK11A5793132C" FOREIGN KEY ("ITEM_ID")
      REFERENCES "CUSTOMER" ("ID") ENABLE,

```



```
        CONSTRAINT "FK11A57F82E29FD" FOREIGN KEY ("OWNER_ID")
            REFERENCES "SERVICELEVEL" ("ID") ENABLE
    )
/
CREATE TABLE "IC2"
(
    "ID" NUMBER(10,0) NOT NULL ENABLE,
    "OWNER_ID" NUMBER(19,0),
    PRIMARY KEY ("ID") ENABLE,
    CONSTRAINT "FK11A58F82E29FD" FOREIGN KEY ("OWNER_ID")
        REFERENCES "SERVICELEVEL" ("ID") ENABLE
)
/
CREATE TABLE "IC3"
(
    "ID" NUMBER(10,0) NOT NULL ENABLE,
    "ITEM" VARCHAR2(255 CHAR),
    "OWNER_ID" NUMBER(10,0),
    PRIMARY KEY ("ID") ENABLE,
    CONSTRAINT "FK11A59A7363EB8" FOREIGN KEY ("OWNER_ID")
        REFERENCES "IC2" ("ID") ENABLE
)
/
CREATE TABLE "LOYALTYACCOUNT"
(
    "ID" NUMBER(19,0) NOT NULL ENABLE,
    "F_POINTS" NUMBER(10,0) NOT NULL ENABLE,
    "F_NUMBER" NUMBER(10,0) NOT NULL ENABLE,
    "F_TOTALPOINTSEARNED" NUMBER(10,0) NOT NULL ENABLE,
    "F_MEMBERSHIP_ID" NUMBER(19,0),
    PRIMARY KEY ("ID") ENABLE,
    CONSTRAINT "FKFABEEF27115A15A8" FOREIGN KEY ("F_MEMBERSHIP_ID")
        REFERENCES "MEMBERSHIP" ("ID") ENABLE
)
/
CREATE TABLE "LOYALTYPROGRAM"
(
    "ID" NUMBER(19,0) NOT NULL ENABLE,
    "F_NAME" VARCHAR2(255 CHAR),
    PRIMARY KEY ("ID") ENABLE
)
/
CREATE TABLE "PROGRAMPARTNER"
(
    "ID" NUMBER(19,0) NOT NULL ENABLE,
    "F_NUMBEROFCUSTOMERS" NUMBER(10,0) NOT NULL ENABLE,
    "F_NAME" VARCHAR2(255 CHAR),
    PRIMARY KEY ("ID") ENABLE
)
/
CREATE TABLE "LOYALTYPROGRAM_PROGRAMPARTNER"
(
    "F_PROGRAMS_ID" NUMBER(19,0) NOT NULL ENABLE,
    "F_PARTNERS_ID" NUMBER(19,0) NOT NULL ENABLE,
    PRIMARY KEY ("F_PROGRAMS_ID", "F_PARTNERS_ID") ENABLE,
    CONSTRAINT "FK403144A5830FFA37" FOREIGN KEY ("F_PROGRAMS_ID")
        REFERENCES "LOYALTYPROGRAM" ("ID") ENABLE,
    CONSTRAINT "FK403144A5C19CF8C1" FOREIGN KEY ("F_PARTNERS_ID")
        REFERENCES "PROGRAMPARTNER" ("ID") ENABLE
)
/
CREATE TABLE "LOYALTYPROGRAM_SERVICELEVEL"
(
    "F_PROGRAM_ID" NUMBER(19,0) NOT NULL ENABLE,
    "F_LEVELS_ID" NUMBER(19,0) NOT NULL ENABLE,
    CONSTRAINT "FKF1EA2510B73CDDD4" FOREIGN KEY ("F_PROGRAM_ID")
        REFERENCES "LOYALTYPROGRAM" ("ID") ENABLE,
    CONSTRAINT "FKF1EA25102CC2B128" FOREIGN KEY ("F_LEVELS_ID")
        REFERENCES "SERVICELEVEL" ("ID") ENABLE
)
```

```
/
CREATE TABLE "SERVICE"
(
  "ID" NUMBER(19,0) NOT NULL ENABLE,
  "F_CONDITION" NUMBER(1,0) NOT NULL ENABLE,
  "F_POINTSEARNED" NUMBER(10,0) NOT NULL ENABLE,
  "F_POINTSBERNED" NUMBER(10,0) NOT NULL ENABLE,
  "F_DESCRIPTION" VARCHAR2(255 CHAR),
  "F_SERVICENR" NUMBER(10,0) NOT NULL ENABLE,
  "F_LEVEL_ID" NUMBER(19,0),
  "F_PARTNER_ID" NUMBER(19,0),
  "F_LOYALTYACCOUNT_ID" NUMBER(19,0),
  PRIMARY KEY ("ID") ENABLE,
  CONSTRAINT "FKD97C5E95439318E8" FOREIGN KEY
("F_LOYALTYACCOUNT_ID")
  REFERENCES "LOYALTYACCOUNT" ("ID") ENABLE,
  CONSTRAINT "FKD97C5E9570A6C745" FOREIGN KEY ("F_LEVEL_ID")
  REFERENCES "SERVICELEVEL" ("ID") ENABLE,
  CONSTRAINT "FKD97C5E951D4B56" FOREIGN KEY ("F_PARTNER_ID")
  REFERENCES "PROGRAMPARTNER" ("ID") ENABLE
)
/
CREATE TABLE "TRANSACTION"
(
  "DTYPE" VARCHAR2(31 CHAR) NOT NULL ENABLE,
  "ID" NUMBER(19,0) NOT NULL ENABLE,
  "F_POINTS" NUMBER(10,0) NOT NULL ENABLE,
  "F_AMOUNT" FLOAT(126) NOT NULL ENABLE,
  "F_NAME" VARCHAR2(255 CHAR),
  "F_GENERATEDBY_ID" NUMBER(19,0),
  "F_CARD_ID" NUMBER(19,0),
  "F_ACCOUNT_ID" NUMBER(19,0),
  PRIMARY KEY ("ID") ENABLE,
  CONSTRAINT "FKE30A7ABE164A6F1B" FOREIGN KEY ("F_GENERATEDBY_ID")
  REFERENCES "SERVICE" ("ID") ENABLE,
  CONSTRAINT "FKE30A7ABE51ADD134" FOREIGN KEY ("F_ACCOUNT_ID")
  REFERENCES "LOYALTYACCOUNT" ("ID") ENABLE,
  CONSTRAINT "FKE30A7ABE77234BE6" FOREIGN KEY ("F_CARD_ID")
  REFERENCES "CUSTOMERCARD" ("ID") ENABLE
)
/
CREATE TABLE "TRANSACTIONREPORT"
(
  "ID" NUMBER(19,0) NOT NULL ENABLE,
  "F_CARD_ID" NUMBER(19,0),
  PRIMARY KEY ("ID") ENABLE,
  CONSTRAINT "FK8058359277234BE6" FOREIGN KEY ("F_CARD_ID")
  REFERENCES "CUSTOMERCARD" ("ID") ENABLE
)
/
CREATE TABLE "TRANSACTIONREPORTLINE"
(
  "ID" NUMBER(19,0) NOT NULL ENABLE,
  "F_TRANSACTION_ID" NUMBER(19,0),
  "F_REPORT_ID" NUMBER(19,0),
  PRIMARY KEY ("ID") ENABLE,
  CONSTRAINT "FK876340A6AE3BD70C" FOREIGN KEY ("F_TRANSACTION_ID")
  REFERENCES "TRANSACTION" ("ID") ENABLE,
  CONSTRAINT "FK876340A667F4B778" FOREIGN KEY ("F_REPORT_ID")
  REFERENCES "TRANSACTIONREPORT" ("ID") ENABLE
)
/
```

References

- [1] Behrend, Andreas, Rainer Manthey and Birgit Pieper. An Amateur's Introduction to Integrity Constraints and Integrity Checking in SQL, University of Bonn: 2001.
- [2] Demuth, Birgit and Heinrich Hussmann. Using OCL/UML Constraints for Relational Database Design. Dresden University of Technology: 1999.
- [3] Demuth, Birgit, Heinrich Hussmann, and Sten Loecher. OCL as a Specification Language for Business Rules in Database Applications. Dresden University of Technology: 2001.
- [4] Donahoo, Michael J. and Gregory D. Speegle. SQL Practical Guide for Developers. Morgan Kaufmann: 2005.
- [5] Groff, James R. and Paul N. Weinberg. SQL: The Complete Reference. Osborne/McGraw-Hill, 2003.
- [6] Gu, Xinhua. Generation of EJB3 Artifacts in a Modeling Platform. Hamburg University of Technology, Master Thesis, 2006.
- [7] Heidenreich, Florian. SQL-Codegenerierung in der metamodellbasierten Architektur des Dresden OCL Toolkit. Technische Universität Dresden, Grosser Beleg, 2005.
- [8] Klasse Objecten – Octopus: OCL Tool for Precise Uml specifications
[<http://www.klasse.nl/octopus/index.html>]
- [9] Kline, Kevin and Daniel Kline. SQL in a Nutshell. O'Reilly & Associates, Inc: 2001.
- [10] King, Gavin and Christian Bauer. Hibernate in Action. Manning Publications Co, 2005.
- [11] Object Management Group. Object Constraint Language OMG Available Specification Version 2.0, 2006.
- [12] Ritter, Norbert and Hans-Peter Steiert. Enforcing Modeling Guidelines in an ORDBMS-based UML Repository, International Resource Management Association Conference 2000, 2000.
- [13] Schmidt, A. Untersuchungen zur Abbildung von OCL-Ausdrucken auf SQL. Dresden University of Technology, Diploma Thesis, 1998.
- [14] Sun Microsystems. JSR 220: Enterprise JavaBeans Version 3.0, 2006.
- [15] Warmer, Jos and Anneke Kleppe. Object Constraint Language, The: Getting Your Models Ready for MDA Second Edition. Addison Wesley, 2003.