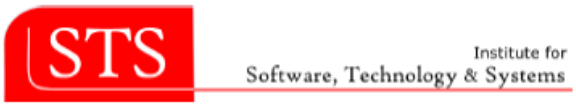

IDE Customization to Support Language Embeddings

Master's Thesis
submitted by
Rakesh Prithiviraj
Matriculation Number: 33745
Master of Science
Information and Media Technologies

supervised by
Prof. Dr. Ralf Möller (STS)
Prof. Dr. Dieter Gollmann (SVA)
Miguel Garcia, M.Sc. (STS)

Hamburg University of Technology (TUHH)

Hamburg, Germany

October 24, 2008

Abstract

Different types of data sources exist and different query languages exist to query them. For example, SQL to query relational database, XQuery to query XML data, JPQL to query object domain etc. The query languages can be used with high level programming languages like Java but they stand isolated, as in most of the cases the queries are embedded as strings inside the program. Therefore the syntactic and semantic meaning of the query is opaque to the programming language and IDE will not be able to provide features like content assist, auto completion, displaying problem markers (for syntactic violation) etc, for constructing the queries. One possible solution to make these features available for the queries is to embed them using the constructs of the host programming language. In the first part of this thesis, an approach to embed JPQL in Java using a sequence of method calls (method chaining) is presented. Another possibility is to extend the programming language itself to include the query mechanism, which is the case with LINQ that adds data querying capabilities to .NET languages like C#, VB. The second part of the thesis describes a LINQ to JPQL translator as an interim step toward the proposed query integration.

Declaration

I hereby declare that the following master thesis, done in the Institute for Software Systems (STS) at the Hamburg University of Technology, has been carried out independently by me, only with the help of the reference material that has been mentioned in the master thesis report.

Rakesh Prithiviraj
(email: marudhar_rakesh@yahoo.co.in)

Hamburg, October 24, 2008

Acknowledgement

I would like to extend my sincere gratitude to Prof. Dr. Ralf Möller for giving me an opportunity to do my master thesis in the Institute for Software Systems (STS) at the Hamburg University of Technology (TUHH), Germany.

I am very much thankful to M.Sc. Miguel Garcia, for supervising this master thesis and for providing valuable feedback and comments to carry out this thesis and to organize the report in a better way.

Last but not the least, I thank my parents and my friends who constantly motivated me during the course of my master thesis.

Rakesh Prithiviraj
Hamburg, October 24, 2008

Contents

List of Figures

Chapter 1

Introduction

This chapter introduces the concept of Domain Specific Languages (DSLs) and their types, briefs about the query languages LINQ and JPQL (as examples of DSL) , presents the objective of this master thesis and concludes with the recent works in the area of embedding DSLs in general purpose programming languages like Java.

1.1 Domain Specific Languages

Domain Specific Languages (DSLs) is a topic that has gained popularity recently. Different knowledge domains demand different types of support from programming languages. Domain engineers often use DSLs to overcome this problem. A DSL is commonly described as a computer language targeted at a particular kind of problem domain. DSLs have been formally studied for many years. The definition of DSL is given in *DSLs: An Annotated Bibliography* as, A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.

Simplicity is critical to the success of a DSL. A person familiar with the language's domain must easily understand it. DSLs allow solutions to be expressed at the level of abstraction of the problem domain. Consequently, domain experts themselves can understand, validate, modify, and often even develop domain-specific language programs. Well-known examples are for DSLs are SQL, CSS, and HTML.

1.2 Types of DSLs

There are mainly two types of DSLs: Internal and External DSLs. Martin Fowler has written extensively on these types of DSL in [Fow08].

> An internal DSL is created with the main language of an application without requiring the creation (and maintenance) of custom compilers and interpreters.

In other words, internal DSLs use the same general purpose programming language that the wider application uses, but uses that language in a particular and limited style. Internal DSLs are also referred to as embedded DSLs.

> External DSLs are DSLs that use a different language to the main language of the application that uses them.

External DSLs have their own custom syntax and a parser to process them. Many XML configurations have ended up as external DSLs. The key strength of an external DSL is that the user is free to use any form of syntax. This results in ability to express the domain in the easiest form possible to read and modify. The major limitation is to build a translator/parser that can parse the DSL and produce something executable, usually in the base language.

It is difficult to differentiate between a DSL and an API. In the case of internal DSLs, they are essentially the same. When thinking in terms of DSL, we exploit the host language to create a readable API with a limited scope. "Internal DSL" is more or less a fancy name for an API that has been created thinking in terms of readability and focusing on a particular problem of a specific domain.

1.3 Internal DSLs in Java

Alex Ruiz and Jeff Bay discusses [1] on creating internal DSLs in Java. Lately, with the advent of Ruby and other dynamic languages, there has been a growing interest in DSLs amongst programmers. These loosely structured languages offer an approach to DSLs which allow a minimum of grammar and therefore the most direct representation of a particular language. However, discarding the ability to use the most powerful modern IDEs such as Eclipse and IntelliJ IDEA is a definite disadvantage with this approach.

On the bright side, by using the Java language we can take advantage of mature IDEs, which can make creation, usage and maintenance of DSLs easier thanks to features like auto-complete, automatic refactoring and debugging. Any internal DSL is limited to the syntax and structure of its base language. In the case of Java, the obligatory use of curly braces, parenthesis and semicolons, and the lack of closures and meta-programming may lead to a DSL that is more verbose that one created with a dynamic language. But this limitation can be made insignificant with new language features in Java 5 (generics, varargs and static imports) as they can help us create a more compact API than previous versions of the language.

In general, a DSL written in Java will not lead to a language that a business user can create from scratch. It will lead to a language that is quite readable by a business user, as well as being very intuitive to read and write from the perspective of the programmer. It has the advantage over an external DSL or a DSL written in a dynamic language that the compiler can enforce correctness along the way, and flag inappropriate usage where Ruby or Pearl would accept nonsensical input and fail at run-time. This reduces the verbosity of testing substantially and can dramatically improve application quality.

[1] An Approach to Internal Domain-Specific Languages in Java, `http://www.infoq.com/articles/internal-dsls-java`

1.4 Query Languages as DSLs

The Java Persistence Query Language (JPQL) is used to define queries against persistent entities (object model) independent of the mechanism used to store those entities. It uses the abstract persistence schemas of entities, including their relationships, for its data model, and it defines operators and expressions based on this data model.

Language Integrated Query (LINQ) is a Microsoft .NET Framework component that adds native data querying capabilities to .NET languages like C#, VB. LINQ defines a set of query operators that can be used to query, project and filter data in arrays, enumerable classes, XML, relational database, and third party data sources.

1.5 Objective

The main objectives of this thesis work are

1. To embed a DSL in Java, supporting features like content assist, compile time type checking etc. JPQL is taken as case study for embedding.
2. To explore the implementation of an integrated query language for Java (similar to LINQ) and to develop a translator that generates a JPQL query from a LINQ query (can be described as LINQ to JPQL provider) as a step towards realizing LINQ for Java.

One of the techniques used by us to embed JPQL is fluent interface[2] or Method Chaining. A fluent interface provides a easy-to-read representation of the domain problem we want to model and it is a usual way to implement an internal DSL. The implementation in our approach involves developing an Ecore metamodel for JPQL and using this metamodel to generate the needed APIs for embedding JPQL.

The translation from a LINQ query to an equivalent JPQL query is implemented based on a visitor pattern. An Ecore metamodel to realize the LINQ standard query operators is developed.

1.6 Related Works

The concept of embedding a query language in Java is not new. In the past and at present, many promising efforts were made to improve language integration by providing fluent interfaces or other lightweight wrappers around SQL, JDBC. This section specifies some of these related efforts and provides references to them.

1. **JPA Query Tool**: This project [3] aims at creating a tool that can be used to browse persistence unit entities and properties, edit and run queries in a GUI editor, browse queries results etc.

[2]Domain Specific Languages, `http://martinfowler.com/dslwip/index.html`
[3]JPA Query Tool, `https://jpaquerytool.dev.java.net/`

2. **Silver** [Sil08]: An attribute grammar specification language that can be used to extend general purpose languages like Java. The authors added new features to Java 1.4 language with modular language extensions. For example, one of the extension that is added is to support SQL queries in Java such that the extension statically check for syntax and type errors in SQL queries.

3. **JEQUEL** [Hun08]: Java Embedded QUEry Language, a DSL for embedding SQL in Java. The APIs needed for fluent interface were developed manually. Listing 1.1 shows fluent interface approach in JEQUEL.

Listing 1.1: *Fluent Interface in JEQUEL*

```
1   public void testSimpleSql() {
2         final SqlString sql =
3                   select(ARTICLE.OID)
4                           .from(ARTICLE, ARTICLE_COLOR)
5                           .where(ARTICLE.OID.eq(ARTICLE_COLOR.ARTICLE_OID)
6                                   .and(ARTICLE.ARTICLE_NO.is_not(NULL)));
7
8         assertEquals("select ARTICLE.OID" +
9                       " from ARTICLE, ARTICLE_COLOR" +
10                      " where ARTICLE.OID = ARTICLE_COLOR.ARTICLE_OID" +
11                      " and ARTICLE.ARTICLE_NO is not NULL", sql.toString());
12    }
```

4. **Quaere** [NM08]: Quaere is an internal DSL that adds a querying syntax similar to SQL to Java. The language is modeled with basis in Microsoft's Language Integrated Query (LINQ) project, which adds similar capabilities to a range of Microsoft .NET languages. The Quaere project allows users to use an internal DSL to filter, enumerate and create projections over a number of collections and other queryable resources using a common, expressive syntax. Example in Listing 1.2 is from Quaere Project.

Listing 1.2: *Fluent interface in Quaere*

```
1
2   public class GettingStartedWithQuaere {
3     public static void main() {
4         String[] cities = {
5             "London",
6             "New York",
7             "Tokyo",
8             "Stockholm",
9             "Barcelona",
10            "Sydney"
11        };
12
13        Iterable<String> allCities =
14            from("city").in(cities).select("city");
15        for (String city: allCities) {
16            System.out.println(city);
17        }
18    }
19  }
```

5. **Typesafe SQL** [KR08]: This embeds the whole of the SQL as a typesafe embedded DSL. Relies on the use of the Java 5 Generics to improve the type safety properties of the DSLs. SQL query embedded in Java is shown in Listing 1.3.

Listing 1.3: *SQL query embedded as fluent interface*

```
1   List<Tuple3<String, Integer, Date>> rows =
2 new QueryBuilder(datasource)
3 .from(p)
4 .where(gt(p.height, 170))
5 .select(p.name, p.height, p.birthday)
6 .list();
```

1.7 Organization of the Report

This thesis work is carried out in two phases. Chapter 2 and Chapter 3 elaborates the first phase. In this phase, we present our approach to use the Java language as a tool for creating internal DSLs. Chapter 4 and Chapter 5 are part of the second phase and mainly deals with the efforts spent towards developing an integrated query for Java. The rest of the report is organized as follows.

Chapter 2 gives the motivation for embedding JPQL in Java using a tool called DSL2JDT, followed by the explanation on the development of a metamodel for JPQL based on its BNF grammar. In this chapter, we also listed the well-formedness rules (WFRs) for the queries from the specification that are not captured by the grammar.

Chapter 3 addresses one of the shortcoming of the metamodel developed in Chapter 2. It uses Java Generics to implement type checking at compile time for embedding JPQL.

Chapter 4 introduces the concept of LINQ queries, Lambda expressions and Expression trees. It also provides an overview of building and visiting an expression tree.

Chapter 5 presents the challenges involved in developing a LINQ provider for JPQL. It elaborates on the metamodel developed for LINQ and the translation algorithm for generating a JPQL query from a LINQ query.

Chapter 6 explores the area for future works and discusses conclusions.

Chapter 2

JPQL as Internal DSL

This chapter describes an approach for embedding JPQL in Java. JPQL is a object oriented query language for Java Persistence that operates on the logical entity model (object model) as opposed to the physical data model. The object graph being queried results from the Object-Relational Mapping (ORM). Embedding JPQL provides an internal DSL for building JPQL statements. Our approach to embed JPQL involves building an Ecore metamodel based on JPQL BNF grammar and then automatically generating the APIs required for method chaining (fluent interface) from the metamodel using DSL2JDT tool [Gar08]. This tool generates a class (expression builder [Fow08]) that encapsulates all of the methods generated from the metamodel.

2.1 Motivation

To understand the importance of embedding query languages like JPQL in Java, consider the following example that makes use of traditional JPQL. This example creates a query and then

Listing 2.1: *JPQL query as a string*

```
1 Query query = em.createQuery("SELECT e.name FROM Employee e");
2 Collection empNames = query.getResultList();
```

executes it to obtain all of the employee names in the database. As the query is specified as string, errors in the query like misspelled keyword or entity name go unnoticed during compile time. Similarly there are other errors that are only detected at runtime rather than at compile time, when the query is specified as string. One way to catch such errors at compile time is to embed the whole of the JPQL as a typesafe embedded DSL. With the internal DSL approach, the query in Listing 2.1 may look like in Listing 2.2.

Listing 2.2: *Embedded JPQL*

```
1 List<Tuple1<String>> names =
2         new QueryBuilder(datasource).from(employee).as(e).select(e.name).list();
```

2.2 Pros and Cons of Embedding JPQL in Java

There are advantages and disadvantages of embedding JPQL in Java as internal DSL. Most of these pros and cons are due to the fact that JPQL is implemented as internal DSL.

Pros:

1. Model driven approach
2. Java IDE can be used to enforce static semantics of JPQL
3. Methods required to build fluent interface can be automatically generated from the metamodel using DSL2JDT
4. Content Assist or Code completion for query keywords and operations
5. Use of Java Generics to enforce type safety
6. Enhanced readability of code
7. Easy for domain users to understand

Cons:

1. Queries may be more verbose than the ones created with dynamic languages like Ruby
2. May need to code manually few classes to improve usability

2.3 JPQL Metamodel Development

JPQL is an extension of the Enterprise Java Beans Query Language (EJB QL). It adds further operations, including bulk update and delete, JOIN operations, GROUP BY, HAVING, projection, subqueries and supports the use of dynamic queries and the use of named parameters. The full definition of the query language (BNF Grammar) can be found in Java Persistence 2.0 specification (JSR 317). The complete JPQL grammar is reproduced in Appendix A for quick reference. A metamodel for JPQL (also known as EJB3QL) based on the JSR 220 specification has already been developed as part of the master's thesis work of Liu Yao [Yao06]. During the development of this metamodel, the focus was not on embedding JPQL queries in Java but was more on implementing tools to support a DSL (JPQL in this case) in eclipse. So we did not use this metamodel but developed a new one based on the JSR 317 spec, focusing on the embedding of JPQL queries in Java. Almost all the terminals and non-terminals which appear in the BNF grammar are modeled. The classes in the resulting Ecore model are grouped into different packages.

The approach followed in building the metamodel for JPQL can be described as follows: We first identified the part of the metamodel that does not depend on other parts of the metamodel or packages. And for a query language, this corresponds to database schema, as it is being

referred by all the queries and does not refer any other part of metamodel. So we first built the package `schema`. From the grammar, we inferred that path expressions are used by the queries most often and hence `pathExpr` is built after the schema. Next comes the packages `from` which refers mainly to `schema` and package `primaries` which will be referred by the queries. Now the option is to continue the metamodel for the select, update and delete statements. Select and subquery grammar are similar though not same and hence they are modeled together in `selectSubQ`. We identified the functions and expressions in the grammar and modeled them in package `functions` and `expressions` respectively. Grammar for update and delete statements is simple and they are clubbed together in `updateDel` Figure 2.1 shows these packages. The following section describes the stages in building the metamodel (.ecore) from the grammar.

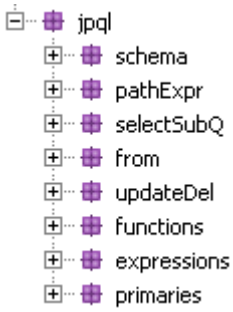

Figure 2.1: *JPQL Metamodel Packages*

2.3.1 Package: Schema

JPQL is a language for querying entities. Instead of tables and rows, the language queries operate on the set of entities known as the abstract persistence schema, from which results may be retrieved. This package defines the entities, their state and their relationships. The definition of persistent unit and abstract persistence schema taken from the specification is as follows:

A persistence unit defines the set of all classes that are related or grouped by the application and which must be co-located in their mapping to a single database.

The term abstract persistence schema refers to the persistent schema abstraction (persistent entities, their state, and their relationships) over which Java Persistence queries operate. Queries over this persistent schema abstraction are translated into queries that are executed over the database schema to which entities are mapped.

In query expressions, entities are referred by name. This name is the abstract schema name of the entity in the context of a query. The class diagram for the package schema is shown in Figure 2.2.

2.3.2 Package: PathExpr

In the grammar, there are many productions that deal with path expressions (Listing 2.3). This package models them. Path expressions are used to navigate out from an entity, either across a

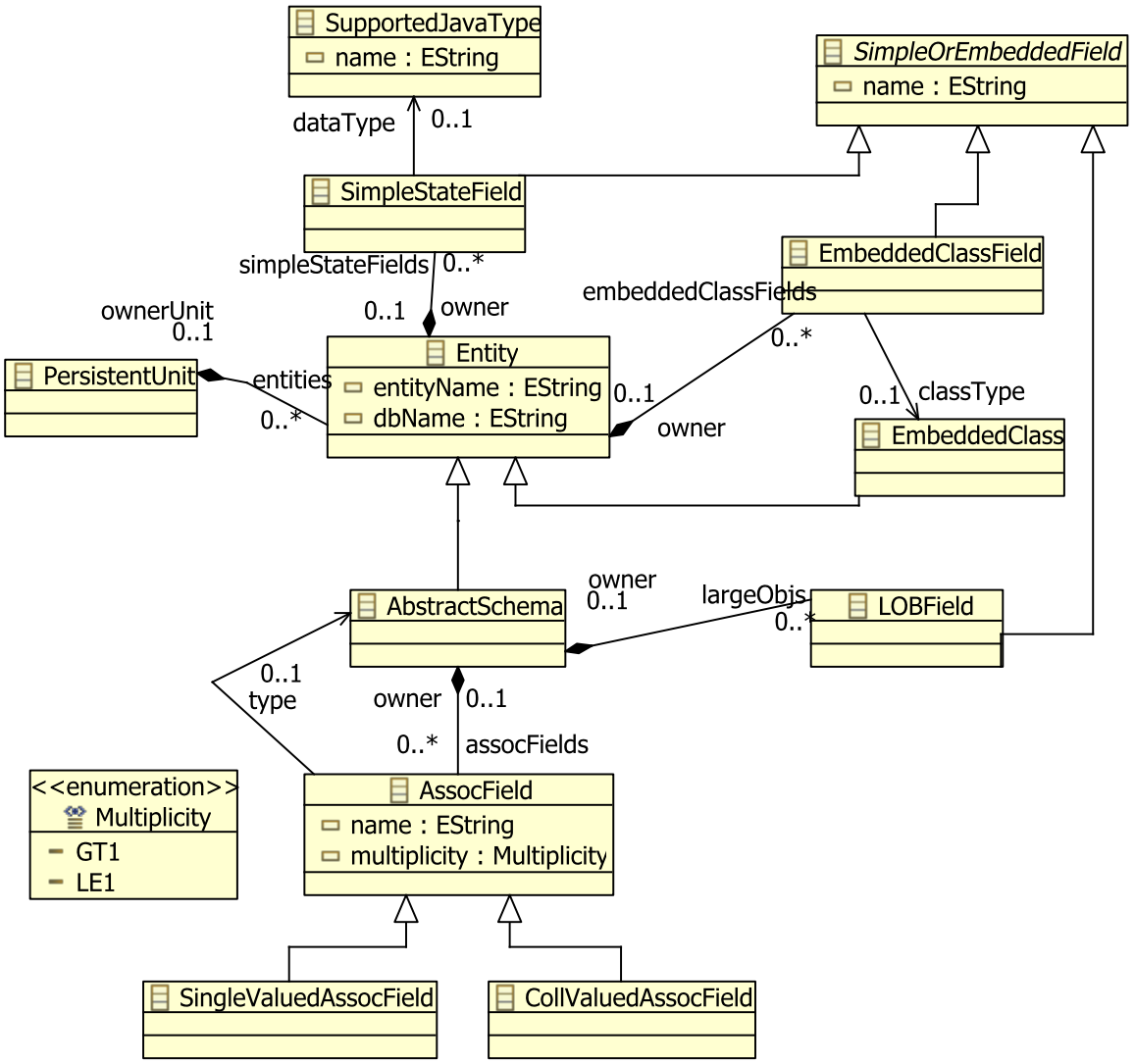

Figure 2.2: *Class Diagram of Package Schema*

relationship to another entity (or collection of entities) or to one of the persistent properties of an entity. Navigation that results in one of the persistent state fields (either field or property) of an entity is referred to as a state field path. Navigation that leads to a single entity is referred to as a single-valued association path, while navigation to a collection of entities is referred to as a collection-valued association path.

The dot operator (.) signifies path navigation in an expression. For example, if the Employee entity has been mapped to an identification variable `e`, then `e.name` is a state field path expression resolving to the employee name. Likewise, the path expression `e.department` is a single-valued association from the employee to the department to which he or she is assigned.

The class diagram for the package pathExpr is shown in Figure 2.3.

Listing 2.3: *Path Expressions Grammar*

```
1
2  single_valued_path_expression ::=
3  state_field_path_expression | single_valued_association_path_expression
4
5  state_field_path_expression ::=
6  {identification_variable | single_valued_association_path_expression}.state_field
7
8  single_valued_association_path_expression ::=
9  identification_variable.{single_valued_association_field.}*
10 single_valued_association_field
11
12 collection_valued_path_expression ::=
13 identification_variable.{single_valued_association_field.}*
14 collection_valued_association_field
```

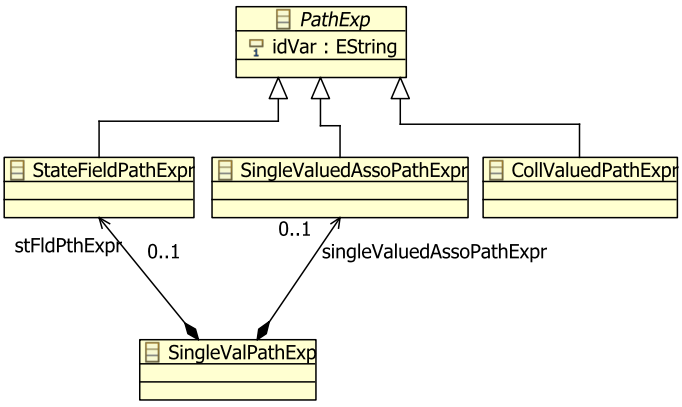

Figure 2.3: *Class Diagram of Package PathExpr*

2.3.3 Package: From

This package models the FROM clause. The FROM clause is used to declare one or more identification variables, optionally derived from joined relationships, that form the domain over which the query should draw its results. The syntax of the FROM clause consists of one or more identification variables and join clause declarations. The identification variable is the starting point for all query expressions. Every query must have at least one identification variable defined in the FROM clause, and that variable must correspond to an entity type. When an identification variable declaration does not use a path expression (that is, when it is a single entity name), it is referred to as a range variable declaration.

Range variable declarations use the following syntax: <entity_name> [AS] <identifier>. The identifier must follow the standard Java naming rules and may be referenced throughout the query in a case-insensitive manner. Multiple declarations may be specified by separating them with commas. Path expressions may also be aliased to identification variables in the case of joins and subqueries. Listing 2.4 shows the part of the grammar that corresponds to the class diagram shown in Figure 2.4

Listing 2.4: *From Clause Grammar*

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

identification_variable_declaration ::= range_variable_declaration
                    { join | fetch_join }*

range_variable_declaration ::= abstract_schema_name [ AS ] identification_variable

join ::= join_spec join_association_path_expression [ AS ] identification_variable

fetch_join ::= join_spec FETCH join_association_path_expression

association_path_expression ::=
collection_valued_path_expression | single_valued_association_path_expression

join_spec::= [ LEFT [ OUTER ] | INNER ] JOIN

join_association_path_expression ::= join_collection_valued_path_expression |
join_single_valued_association_path_expression

join_collection_valued_path_expression::=
identification_variable.collection_valued_association_field

join_single_valued_association_path_expression::=
identification_variable.single_valued_association_field

collection_member_declaration ::=
IN (collection_valued_path_expression) [ AS ] identification_variable
```

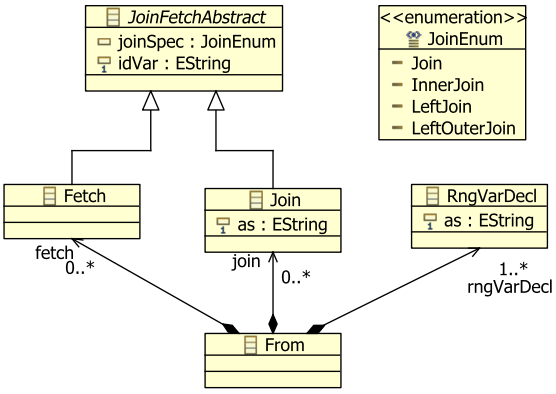

Figure 2.4: *Class Diagram of Package From*

2.3.4 Package: SelectSubQ

This package models the select queries and subqueries. Select queries are the primary query type and facilitate the bulk retrieval of data. They are also the most common form of query used in applications. The overall form of a select query is as shown in Listing 2.5.

The simplest form of a select query consists of two mandatory parts, the SELECT clause and the FROM clause. The SELECT clause defines the format of the query results, while the FROM clause defines the entity or entities from which the results will be obtained. The SELECT clause

Listing 2.5: *Select Statement Grammar*

```
1
2  select_statement ::= select_clause from_clause [where_clause]
3  [groupby_clause] [having_clause] [orderby_clause]
4
5  select_clause ::= SELECT [ DISTINCT ] select_expression {, select_expression}*
6
7  select_expression ::=
8  single_valued_path_expression |
9  aggregate_expression |
10 identification_variable |
11 OBJECT (identification_variable) |
12 constructor_expression
13
14 constructor_expression ::=
15 NEW constructor_name ( constructor_item {, constructor_item}* )
16
17 constructor_item ::= single_valued_path_expression | aggregate_expression
```

of a query can take several forms, including simple and complex path expressions, transformation functions, multiple expressions (constructor expressions), and aggregate functions. A form of SELECT clause involving multiple expressions is the constructor expression, which specifies that the results of the query are to be stored using a user-specified object type.

The WHERE clause of a query is used to specify filtering conditions to reduce the result set. The WHERE clause is simply the keyword WHERE, followed by a conditional expression.

The GROUP BY clause defines the grouping expressions over which the results will be aggregated. A grouping expression must either be a single-valued path expression (state field or single-valued association field) or an identification variable. In the absence of a GROUP BY clause, the entire query is treated as one group, and the SELECT list may contain only aggregate functions.

Listing 2.6: *SubQuery Expressions Grammar*

```
1
2  subquery ::= simple_select_clause subquery_from_clause [where_clause]
3                                          [groupby_clause] [having_clause]
4
5  subquery_from_clause ::=
6  FROM subselect_identification_variable_declaration
7  {, subselect_identification_variable_declaration}*
8
9  subselect_identification_variable_declaration ::=
10 identification_variable_declaration |
11 association_path_expression [ AS ] identification_variable |
12 collection_member_declaration
13
14 simple_select_clause ::= SELECT [ DISTINCT ] simple_select_expression
15
16 simple_select_expression::=
17 single_valued_path_expression |
18 aggregate_expression |
19 identification_variable
```

The HAVING clause defines a filter to be applied after the query results have been grouped. It is effectively a secondary WHERE clause, and its definition is the same, the keyword HAVING followed by a conditional expression. The key difference with the HAVING clause is that its conditional expressions are limited to state fields or single-valued association fields previously identified in the GROUP BY clause. Conditional expressions in the HAVING clause may also make use of aggregate functions. In many respects, the primary use of the HAVING clause is to restrict the results based on the aggregate result values. Subqueries may be used in the

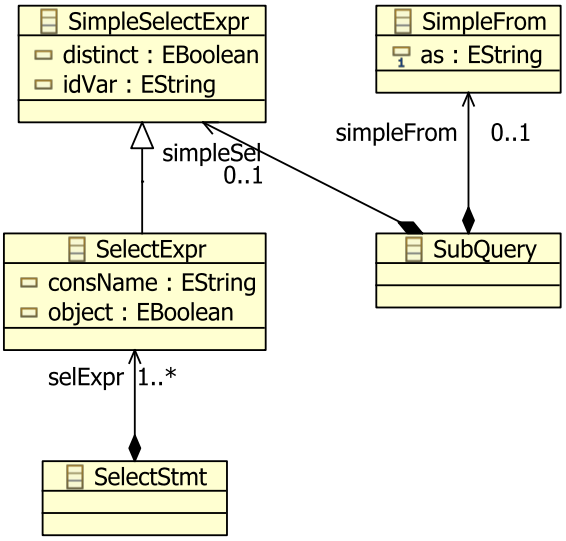

Figure 2.5: *Class Diagram of Package SelectSubQ*

WHERE and HAVING clauses of a query. A subquery is a complete SELECT query inside a pair of parentheses that is embedded within a conditional expression. The results of executing the subquery (which will either be a scalar result or a collection of values) are then evaluated in the context of the conditional expression.

Subqueries are a technique for solving the complex query scenarios. The scope of an identifier variable name begins in the query where it is defined and extends down into any subqueries. Identifiers in the main query may be referenced by a subquery, and identifiers introduced by a subquery may be referenced by any subquery that it creates. If a subquery declares an identifier variable of the same name, then it overrides the parent declaration and prevents the subquery from referring to the parent variable. The part of the grammar that specifies the syntax of subquery is shown in Listing 2.6. The class diagram for the package selectSubQ is shown in Figure 2.5.

2.3.5 Package: UpdateDel

This package models the update and delete statements. Update and delete statements provide an equivalent to the SQL UPDATE and DELETE statement but with JPQL conditional expressions. The form of an update and delete query is shown in Listing 2.7. Each UPDATE statement consists of a single-valued path expression, assignment operator (=), and an expression. Expression choices for the assignment statement are slightly restricted compared to regular conditional expressions. The right side of the assignment must resolve to a literal, simple expression resolving to a basic type, function expression, identification variable, or input

parameter. The result type of that expression must be compatible with the simple association path or persistent state field on the left side of the assignment.

Listing 2.7: *Update Statement Grammar*

```
1
2  update_statement ::= update_clause [where_clause]
3
4  update_clause ::= UPDATE abstract_schema_name [ [ AS ] identification_variable]
5  SET update_item {, update_item}*
6
7  update_item ::= [identification_variable.]
8  {state_field | single_valued_association_field} = new_value
9
10 new_value ::=
11 simple_arithmetic_expression |
12 string_primary |
13 datetime_primary |
14 boolean_primary |
15 enum_primary
16 simple_entity_expression |
17 NULL
18
19 delete_statement ::= delete_clause [where_clause]
20
21 delete_clause ::= DELETE FROM abstract_schema_name [[ AS ] identification_variable]
```

Delete queries are polymorphic. Any entity subclass instances that meet the criteria of the delete query will also be deleted. The WHERE clause of an UPDATE statement functions the same as a SELECT statement and may use the identification variable defined in the UPDATE clause in expressions.

The class diagram for the package UpdateDel is shown in Figure 2.6.

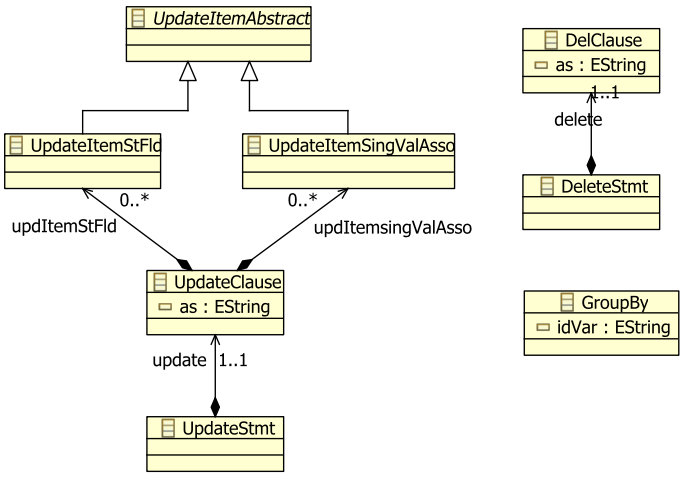

Figure 2.6: *Class Diagram of Package UpdateDel*

2.3.6 Package: Functions

This package models the functions that are often used in JPQL queries. Conditional expressions may leverage a number of functions that can be used to modify query results in the WHERE and HAVING clauses of a select query. Figure 2.7 shows the class diagram of functions package.

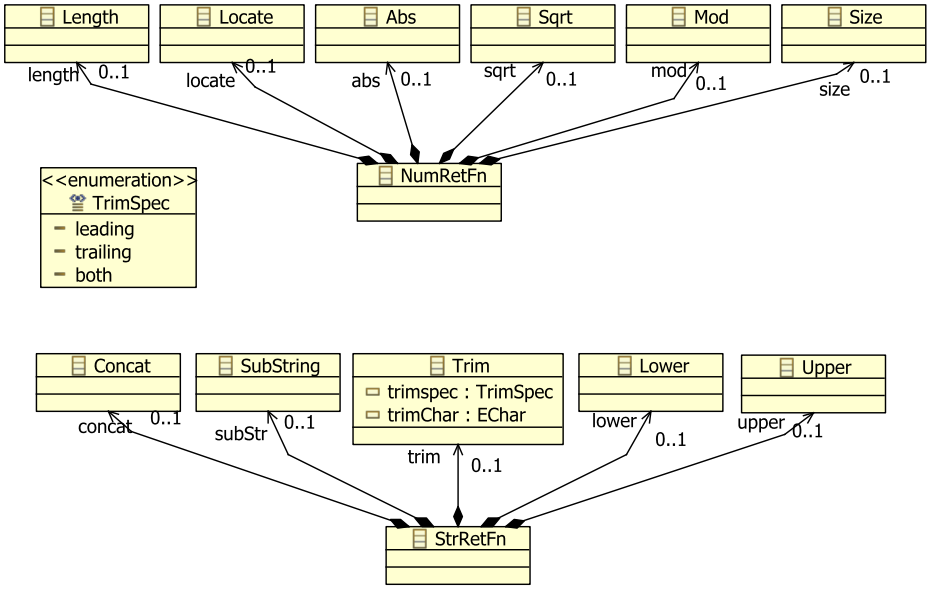

Figure 2.7: *Class Diagram of Package Functions*

2.3.7 Package: Expressions

Much of the conditional expression support in JPQL is borrowed directly from SQL. The key difference between conditional expressions in JPQL and SQL is that JPQL expressions can leverage identification variables and path expressions to navigate relationships during expression evaluation. Conditional expressions are constructed using a combination of logical operators, comparison expressions, primitive and function operations on fields, and so on.

BETWEEN expressions are used to determine whether or not the result of an expression falls within an inclusive range of values. Numeric, string, and date expressions may be evaluated in this way. LIKE Expressions provides a limited form of string pattern matching. Each LIKE expression consists of a string expression to be searched and a pattern string and optional escape sequence that defines the match conditions. The IN expression may be used to check whether a single-valued path expression is a member of a collection. The collection may be defined inline as a set of literal values or may be derived from a subquery. Queries may check whether a collection association path resolves to an empty collection or has at least one value using collection expressions. The EXISTS expression returns true if a subquery returns any rows. The ANY, ALL, and SOME operators may be used to compare an expression to the results of a subquery.

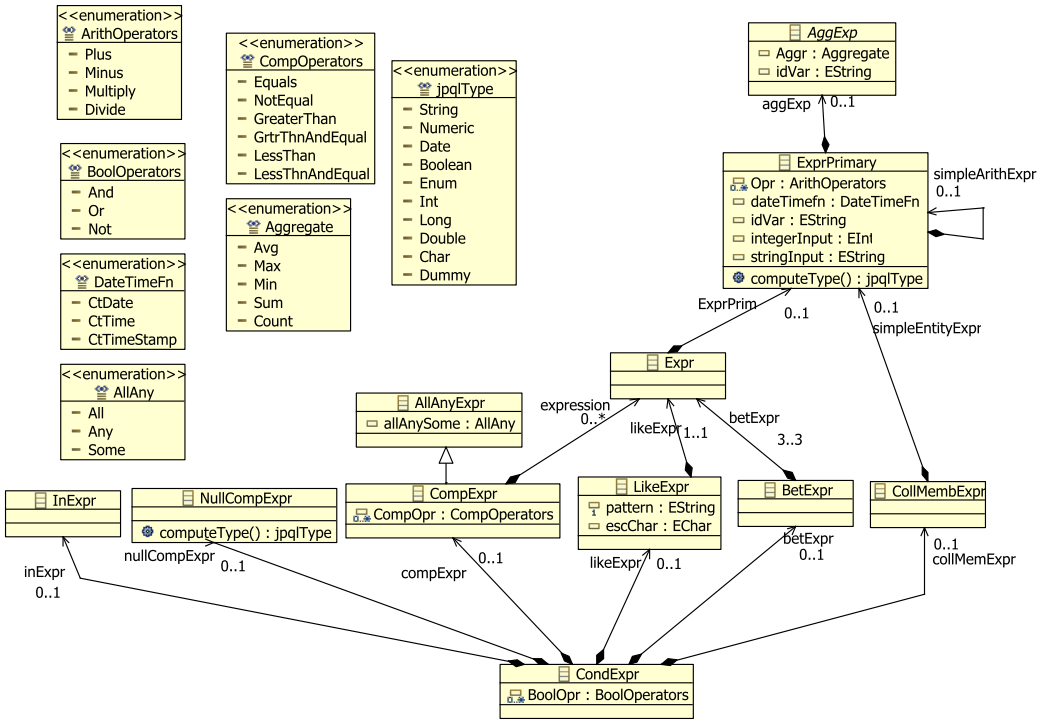

Figure 2.8: *Class Diagram of Package Expressions*

2.3.8 Package: Primaries

This package contains Identification Variable interface and Input parameter which is used in several expressions.

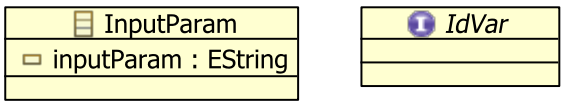

Figure 2.9: *Class Diagram of Package Primaries*

2.4 Well-Formedness Rules (WFRs)

Now that the metamodel `.ecore` is ready, this can be used as input to the DSL2JDT tool to generate the methods to embed JPQL query. But before doing this, we need to carry out the task of implementing constraints in the form of Object Constraint Language (OCL) or Java to check well-formedness of the embedded query. The metamodel is developed based on the BNF grammar given in the JPA 2.0 specification [Gro08]. But the grammar alone does not capture all of the constraints of the query language. For example as per the section 4.3.1 of the specification, entity names must be unique within the persistence unit. But this constraint is not depicted in the grammar. More examples for the shortcomings of the JPQL BNF grammar to capture all the well-formedness rules are given in the article by Miguel [Gar06]. This section

lists the constraints taken from the specification along with the section number in which the constraint is specified. Examples are provided wherever needed.

1. Section 4.4.2: All identification variables must be declared in the FROM clause.
2. Section 4.4.4: It is syntactically illegal to compose a path expression from a path expression that evaluates to a collection. The path expression `department.employees.name` is not valid because `department.employees` is a collection.
3. Section 4.4.5.3: The association referenced by the right side of the FETCH JOIN clause must be an association that belongs to an entity that is returned as a result of the query.
4. Section 4.6.1: For enum literals, the enum class name must be specified.
5. Section 4.6.2: All identification variables used in the WHERE or HAVING clause of a SELECT or DELETE statement must be declared in the FROM clause. The identification variables used in the WHERE clause of an UPDATE statement must be declared in the UPDATE clause.
6. Section 4.6.3: It is illegal to use a collection_valued_path_expression within a WHERE or HAVING clause as part of a conditional expression except in an empty_collection_comparison_expression, in a collection_member_expression, or as an argument to the SIZE operator.
7. Section 4.6.4: Constraint for input parameters: Positional and named parameters may not be mixed in a single query.
8. Section 4.6.8: Constraint for In Expressions: The state_field_path_expression must have a string, numeric, or enum value.
9. Section 4.6.15: Subqueries are restricted to the WHERE and HAVING clauses.
10. Section 4.6.16: Built-in functions are restricted to the WHERE and HAVING clauses.
11. Section 4.7:
    - Any item that appears in the SELECT clause (other than as an argument to an aggregate function) must also appear in the GROUP BY clause. So the query in Listing 2.8 is invalid as `e.salary` is not listed in the GROUP BY clause.

Listing 2.8: *Invalid JPQL query*

```
SELECT e.name, e.salary FROM employee e  GROUP BY e.name
```

    - Grouping by an entity is permitted. In this case, the entity must contain no serialized state fields or lob-valued state fields.
12. Section 4.8.4:
    - For all aggregate functions except COUNT, the path expression that is the argument to the aggregate function must terminate in a state-field.
    - The path expression argument to COUNT may terminate in either a state-field or a association-field, or the argument to COUNT may be an identification variable.

- Arguments to the functions SUM and AVG must be numeric.
- Arguments to the functions MAX and MIN must correspond to orderable state-field types (i.e., numeric types, string types, character types, or date types).

13. Section 4.9: When the ORDER BY clause is used in a query, each element of the SELECT clause of the query must be one of the following:

    (a) an identification variable x, optionally denoted as OBJECT(x)

    (b) a single_valued_association_path_expression

    (c) a state_field_path_expression

    In the first two cases, each orderby_item must be an orderable state-field of the entity abstract schema type value returned by the SELECT clause. In the third case, the orderby_item must evaluate to the same state-field of the same entity abstract schema type as the state_field_path_expression in the SELECT clause. Therefore the query in Listing 2.9 is invalid as the ORDER BY item is `e.age` instead of `e.name`.

Listing 2.9: *Invalid JPQL query*

```
SELECT e.name FROM employee e ORDER BY e.age
```

14. Section 4.10: Only one entity abstract schema type may be specified in the FROM or UPDATE clause of delete and update queries respectively.

    Apart from the above mentioned WFRs, another important check is type compatibility check. For example the LHS and RHS items in assignment and conditional expressions should be type compatible.

2.5 JPQL Type System - Computing Expression Types

To embed JPQL in a typesafe manner, it is important to know the type to which a given JPQL expression will evolve. This section tabulates the rules to compute the type of expressions given the type of its subexpressions. The classification is based on the result type to which a expression evaluates. In the following tables (Table 2.1-2.4), Op1, Op2 and Op3 represent Operands.

It has to be noted that these tables are developed based on the JPQL specification (JSR 317). Tables shown here does not cover the cases where the result type of an expression is unknown. For example, as per the specification, if the value of a Op1 in an IN expression is NULL or unknown, the value of the expression is unknown. As we need to know the datatype of a expression when checking WFR, we modeled the JPQL datatype as a enumeration `jpqlType` in the metamodel.

2.6 JUnit Test to Check Well-formedness

As mentioned earlier, IDE can be used to verify the static semantics of the embedded JPQL queries. Two possible approaches to achieve this have been mentioned in the article by Miguel

JPQL Expression	Op1 Type	Op2 Type	Op3 Type	Result Type
Op1 Between Op2 And Op3	String	String	String	Boolean
	Numeric	Numeric	Numeric	Boolean
	Date	Date	Date	Boolean
Op1 In Op2	String	String+	-	Boolean
	Numeric	Numeric+	-	Boolean
	Enum	Enum+	-	Boolean
Op1 Like Op2 [escape Op3]	String	String	Char	Boolean
Op1 Is Null	State Field	-	-	Boolean
	Entity	-	-	Boolean
Op1 Is Empty	Entity*	-	-	Boolean
Op1 Member [Of] Op2	Entity	Entity*	-	Boolean
Exists Op1	Any Type	-	-	Boolean
Op1 (> | >= | < | <=) Op2	String	String	-	Boolean
	Numeric	Numeric	-	Boolean
	Date	Date	-	Boolean
Op1 (= | <>) Op2	String	String	-	Boolean
	Numeric	Numeric	-	Boolean
	Date	Date	-	Boolean
	Boolean	Boolean	-	Boolean
	Enum	Enum	-	Boolean
	Entity	Entity	-	Boolean

Table 2.1: *Expressions that return Boolean Type*

JPQL Expression	Op1 Type	Op2 Type	Op3 Type	Result Type
Length (Op1)	String	-	-	Integer
Locate (Op1, Op2, [Op3])	String	String	Integer	Integer
Mod (Op1, Op2)	Numeric	Numeric	-	Integer
Size (Op1)	Entity*	-	-	Integer
Count (Op1)	State Field	-	-	Long
	Entity	-	-	Long
Sqrt (Op1)	Numeric	-	-	Double
Avg (Op1)	Numeric	-	-	Double
Sum (Op1)	Numeric	-	-	Numeric
(Max | Min) Op1	Numeric	-	-	Numeric
Abs (Op1)	Numeric	-	-	Numeric

Table 2.2: *Expressions that return Numeric Type*

[Gar08], namely the pragmatic and the grand plan way. The following description is based on this article. In the pragmatic approach, the JUnit support of the JDT is leveraged to check the WFRs of JPQL queries. The procedure to do the same is as follows:

1. Related JPQL expressions are embedded in a Java method that returns the root node of the Abstract Syntax Tree (AST) of the embedded expression.
2. The root node returned in the step 1 is passed to the EMF validation method.

JPQL Expression	Op1 Type	Op2 Type	Op3 Type	Result Type
Concat (Op1, Op2)	String	String	-	String
Lower (Op1)	String	-	-	String
Upper (Op1)	String	-	-	String
Substring (Op1, Op2, Op3)	String	Numeric	Numeric	String
Trim [Op1] from Op2	Char	String	-	String
(Max | Min) Op1	String	-	-	String

Table 2.3: *Expressions that return String Type*

JPQL Expression	Op1 Type	Op2 Type	Result Type
(Max | Min) Op1	Date	String	Date
CURRENT_DATE	-	-	Date
CURRENT_TIME	-	-	Date
CURRENT_TIMESTAMP	-	-	Date

Table 2.4: *Expressions that return Date Type*

3. The EMF validation method is invoked within JUnit's `assertTrue()`

It should be noted that EMF validation method can check for the WFRs of all the nodes under the root node i.e. all child nodes of the root node, if the relation between the root node and child nodes is of composition type (black diamond in UML diagram). So it is enough to pass the root node of the AST without the need to explicitly invoke EMF validation for the child nodes.

Let's see an example for the above procedure. In this example, we will embed a JPQL SELECT statement with a BETWEEN conditional expression and we will check the WFR of the BETWEEN conditional expression. The syntax of the BETWEEN expression and the SELECT query to embed is shown in Listing 2.10.

Listing 2.10: *BETWEEN expression syntax and SELECT statement*

```
1 expression1 [ NOT ] BETWEEN expression2 AND expression3
2
3 JPQL Query to embed
4 SELECT e FROM Employee e WHERE '200' BETWEEN 40000 AND 45000
```

One of the constraint for BETWEEN expression is that all of its three expressions should be of same type. All three expressions can be either arithmetic expression or string expression or date-time expression. So the query in Listing 2.10 is not well-formed. We can verify this constraint as part of the EMF validation. Listing 2.11 shows the Java method `selStmt` in which the SELECT query is embedded. This method returns the node in AST that corresponds to `SelectStmt`.

Now we will invoke the EMF Validation method `MyEcoreUtil.isWellFormed` for the SelectStmt node from the JUnit's `assertTrue()`. This is shown in the Listing 2.12. The method `MyEcoreUtil.isWellFormed` encapsulates the invocation of the EMF validation. It is shown

in Listing 2.13. The WFR to check can be specified in Java or in OCL [Gro06]. In both cases an annotation with source `http://www.eclipse.org/emf/2002/Ecore` has to be added in the Ecore model to the appropriate class (in this example, class `BetExpr`). This is shown in Figure 2.10. Adding the constraint this way to the model will result in generation of validation code. The generated validation code is complete if the WFR is specified as OCL in the model. If not, then we have to manually code the WFR check in the validation method. More information on annotating an Ecore model with constraints can be found in the article [Dam06] by Christian W.Damus.

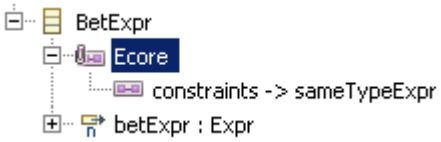

Figure 2.10: *BetExpr with annotation*

In our example, as we have not specified the OCL in the model, our generated validation method `validateBetExpr_sameTypeExpr` is incomplete and looks like in Listing 2.14. To verify the constraint and to complete the code in Listing 2.14 we need to know the datatype of the expressions used in the BETWEEN statement. For this, we added an operation to EClass `ExprPrimary` that returns the type of an expression as shown in Figure 2.11. The code snippet for the added operation is shown in Listing 2.15.

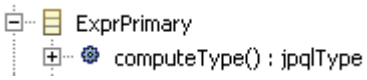

Figure 2.11: *Operation to compute expression type*

Now that we have a method to return the expression type, we can use this to complete the code in Listing 2.14. This is shown in Listing 2.16. Executing the JUnit test (Listing 2.12) will throw assertion error as `MyEcoreUtil.isWellFormed(selStmExpr)` which is the argument to `assertTrue` returns `false`.

2.7 Improving Usability

One useful observation from the code shown in Listing 2.11 is embedding a query like a SELECT with a WHERE clause based on the methods generated from the DSL2JDT tool may lead to verbose embedding if we do not adapt any other techniques. Also from the user perspective, query embedding like in Listing 2.11 is less useful. Usability can be improved by following different techniques. For example, in the project work [KR08], the author argues that mixing the fluent interface with static functions, metadata and closures provides for a better user experience than pure method chaining. For our case of embedding JPQL, user experience can be improved by extending the expression builder class generated by the DSL2JDT tool. Though this involves manual coding, it is still desirable considering the advantage that we get by class extension. This section gives an example on how to achieve this.

In the Listing 2.11, we extended the expression builder `JpqlExprBuilder` by the `private static class E` and used it to embed the query. Alternatively, we can have a class with

methods coded manually to improve the usability and this class then extends the expression builder. We call these hand coded methods as intermediate methods. This is shown in Listing 2.17. Here the class `MyClass` extends the expression builder class `JpqlExprBuilder`. Therefore at the point where we want to embed JPQL queries, it is enough to extend `MyClass` instead of extending `JpqlExprBuilder`. This provides the intermediate methods to appear in the content assist when embedding a query. Given that there are relevant intermediate methods, then Listing 2.18 shows a embedded JPQL query.

The intermediate methods are basically wrappers for the APIs generated by the DSL2JDT. The intermediate methods pass the parameters they receive to the methods generated by the DSL2JDT. This way they provide a layer of abstraction to the generated methods. Also overloading of the intermediate methods can be made use of to provide default values for the optional parameters in the query. This relieves the user from specifying the default value. For example, consider the LOCATE funtion (syntax: `LOCATE(string1, string2 [, start])`) used in the JPQL queries which returns the position of string2 in string1, optionally starting at the position indicated by start. This means that we can write the intermediate methods in two ways, one with the `start` parameter and one without it as shown in Listing 2.19 and the user can choose any of these at the time of embdding a query.

2.8 Summary

This chapter first described the motivation for embedding JPQL as internal DSL in Java and then elaborated the development of a `.ecore` metamodel for JPQL based on the BNF grammar of JSR 317. This metamodel is used to generate the APIs needed for method chaining using DSL2JDT tool. We also listed the WFRs for the queries from the specification that are not captured by the grammar. Type system of JPQL which plays an important role in the correctness of JPQL queries was explained. Steps to implement the WFRs and to check well-formedness during editing are also mentioned in this chapter. We inferred that embedding the JPQL queries, only based on the methods generated from the ecore model may lead to verbose embedding. This chapter proposed the technique of using intermediate class that extends the expression builder to provide better user experience in embedding a query. One of the shortcoming of the metamodel developed in this chapter and which will also be addressed in the next chapter is the type safety. For an instance, the current metamodel may allow the assignment of string to a numeric simple state field. We propose the use of new feature introduced in Java 5, Generics to implement type checking at compile time for embedding JPQL.

Listing 2.11: *JPQL embedded in Java*

```
public class SelExpr {

  public static SelectStmt selStmt() {
   base.jpql.schema.Entity emp;
   base.jpql.selectSubQ.SelectExpr selEmp;
   base.jpql.from.RngVarDecl rngVarDecl;
   base.jpql.from.From fromExpr;

  /*
   * Invalid query
   * JPQL Query to embed SELECT e FROM Employee e WHERE
   * '200' BETWEEN 40000 AND 45000
   */

  emp = E.entity().entityName("employee").simpleStateFields()
.embeddedClassFields().toAST();

  selEmp = E.selectExpr().idVar("e").toAST();

  rngVarDecl = E.rngVarDecl().entity(emp).as("e").toAST();
  fromExpr = E.from().rngVarDecl(rngVarDecl).join().fetch().toAST();

  ExprPrimary exprPrim1 = E.exprPrimary().Opr().stringInput("200").toAST();
  Expr expr1 = E.expr().ExprPrim(exprPrim1).toAST();

  ExprPrimary exprPrim2a = E.exprPrimary().Opr().integerInput(40000).toAST();
  Sqrt sqrt1 = E.sqrt().number(exprPrim2a).toAST();
  NumRetFn num1 = E.numRetFn().sqrt(sqrt1).toAST();
  ExprPrimary exprPrim2b = E.exprPrimary().Opr().numFunc(num1).toAST();
  Expr expr2 = E.expr().ExprPrim(exprPrim2b).toAST();

  ExprPrimary exprPrim3a = E.exprPrimary().Opr().integerInput(45000).toAST();
  Sqrt sqrt2 = E.sqrt().number(exprPrim3a).toAST();
  NumRetFn num2 = E.numRetFn().sqrt(sqrt2).toAST();
  ExprPrimary exprPrim3b = E.exprPrimary().Opr().numFunc(num2).toAST();
  Expr expr3 = E.expr().ExprPrim(exprPrim3b).toAST();

  BetExpr betExpr1 = E.betExpr().betExpr(expr1, expr2, expr3).toAST();

  CondExpr betCond = E.condExpr().BoolOpr().betExpr(betExpr1).toAST();

  SelectStmt selStmt = E.selectStmt().selExpr(selEmp).from(fromExpr)
.where(betCond).groupBy().orderBy().toAST();

  return selStmt;

 }

  private static class E extends MyClass {

  }
}
```

Listing 2.12: *JUnit test to check well-formedness*

```
public class TestSelect extends TestCase {
        public void testSelStmt() {
                SelectStmt selStmExpr = SelExpr.selStmt();
                assertTrue(MyEcoreUtil.isWellFormed(selStmExpr));
        }
}
```

Listing 2.13: *EMF validation method Invocation*

```
public class MyEcoreUtil {
        public static boolean isWellFormed(EObject root) {
                Diagnostician diagnostician = new Diagnostician();
                final Diagnostic diagnostic = diagnostician.validate(root);
                boolean res = diagnostic.getSeverity() == Diagnostic.OK;
                return res;
        }
}
```

Listing 2.14: *Generated validation code - Incomplete*

```
public boolean validateBetExpr_sameTypeExpr(BetExpr betExpr,
  DiagnosticChain diagnostics, Map<Object, Object> context) {
  // TODO implement the constraint
  // -> specify the condition that violates the constraint
  // -> verify the diagnostic details, including severity, code, and message
  // Ensure that you remove @generated or mark it @generated NOT
  if (false) {
   if (diagnostics != null) {
           diagnostics.add
                (createDiagnostic
            (Diagnostic.ERROR,
                 DIAGNOSTIC_SOURCE,
                 0,
                 "_UI_GenericConstraint_diagnostic",
                 new Object[] { "sameTypeExpr", getObjectLabel(betExpr, context) },
                 new Object[] { betExpr },
                 context));
        }
        return false;
  }
  return true;
}
```

Listing 2.15: *Operation for finding expression return type added to class ExprPrimary*

```
public jpqlType computeType() {
  NumRetFn numfn = this.getNumFunc();
  StrRetFn strfn = this.getStrFunc();
  String estr = this.getStringInput();

  if (numfn instanceof NumRetFn )
        return jpqlType.NUMERIC;
  if (strfn instanceof StrRetFn || estr instanceof String)
        return jpqlType.STRING;
  else
        return jpqlType.DUMMY;
}
```

Listing 2.16: *WFR check for BETWEEN Expr*

```
public boolean validateBetExpr_sameTypeExpr(BetExpr betExpr,
DiagnosticChain diagnostics, Map<Object, Object> context) {
boolean wfr_failed = true;
jpqlType[] type1={jpqlType.CHAR, jpqlType.CHAR, jpqlType.CHAR};
EList<Expr> bet= betExpr.getBetExpr();
int i =0;
  for (Expr e: bet){
        type1[i] = e.getExprPrim().computeType();
        i++;
        }

  if (type1[0].equals(type1[1]))
        if (type1[1].equals(type1[2]))
              wfr_failed = false;

  if (wfr_failed) {
        if (diagnostics != null) {
          diagnostics.add
           (createDiagnostic
           (Diagnostic.ERROR,
                DIAGNOSTIC_SOURCE,
                0,
                "_UI_GenericConstraint_diagnostic",
                new Object[] { "sameTypeExpr", getObjectLabel(betExpr, context) },
                new Object[] { betExpr },
                context));
         }
         return false;
   }
   return true;
}
```

Listing 2.17: *Intermediate class*

```
public class MyClass extends JpqlExprBuilder {
        /*
         * Hand coded methods to be used in the method chaining goes here
         */

}
```

Listing 2.18: *Usage of Intermediate methods in lines 22 24 and 26*

```
1  public class SelExpr {
2
3    public static SelectStmt selStmt() {
4     base.jpql.schema.Entity emp;
5     base.jpql.selectSubQ.SelectExpr selEmp;
6     base.jpql.from.RngVarDecl rngVarDecl;
7     base.jpql.from.From fromExpr;
8
9    /*
10    * Invalid query
11    * JPQL Query to embed SELECT e FROM Employee e WHERE
12    * '200' BETWEEN 40000 AND 45000
13    */
14
15   emp = E.entity().entityName("employee").simpleStateFields()
16 .embeddedClassFields().toAST();
17
18   selEmp = E.selectExpr().idVar("e").toAST();
19
20   rngVarDecl = E.rngVarDecl().entity(emp).as("e").toAST();
21   fromExpr = E.from().rngVarDecl(rngVarDecl).join().fetch().toAST();
22
23   Expr expr1 = E.expr().ExprPrim().stringInput("200").toAST();
24
25   Expr expr2 = E.expr().ExprPrim().numRetFn().sqrt(40000).toAST();
26
27   Expr expr3 = E.expr().ExprPrim().numRetFn().sqrt(45000).toAST();
28
29   BetExpr betExpr1 = E.betExpr().betExpr(expr1, expr2, expr3).toAST();
30
31   CondExpr betCond = E.condExpr().BoolOpr().betExpr(betExpr1).toAST();
32
33   SelectStmt selStmt = E.selectStmt().selExpr(selEmp).from(fromExpr)
34 .where(betCond).groupBy().orderBy().toAST();
35
36   return selStmt;
37
38  }
39
40   private static class E extends MyClass {
41
42   }
43 }
```

Listing 2.19: *Overloading of Intermediate methods*

```
1 locate(string1,string2,start);
2 locate(string1,string2);
```

Chapter 3

Typesafe Language Embedding

Java was intended to be a type safe language. It disallows incompatible type substitutions. For example, the Java statement `int i = 3.4` throws a type mismatch error at compile time. IDEs usually rely on problem markers to display such errors in the editor as shown in Figure 3.1. One of the objective in embedding JPQL in Java is to perform the embedding in a type-safe manner. For example, when the user embeds a JPQL expression of type `Integer` where type `String` is expected, then the IDE should indicate such errors at edit time by problem markers. To achieve this, we modify the `.ecore` metamodel developed in the previous chapter to include Java Generics or parameterized types.

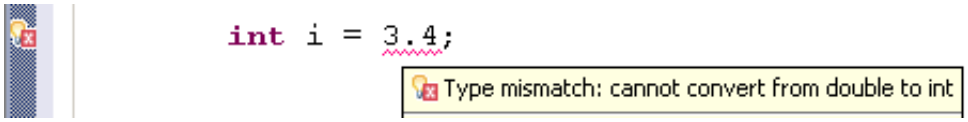

Figure 3.1: *Problem marker for type mismatch*

This chapter shows how the current metamodel lacks the ability to enforce type safety and then explains the steps to modify the metamodel and the expression builder generated by the DSL2JDT tool to embed JPQL in a type safe manner. We will use the same JPQL query shown in Listing 2.10 to embed in a type-safe manner.

3.1 Well-Formedness Check

In the previous chapter, we have derived WFRs from the JSR specification and not all of them are enforced with the current metamodel. For example, section 4.8.4 of the spec (item number 12 in the WFR list of previous chapter), specifies that arguments to the functions SUM and AVG must be numeric. With the current metamodel, if the user passes a non-numeric argument to the functions SUM or AVG, it is not reported to the user by means of problem markers in the editor. We showed that JUnit tests can be used to enforce WFRs. But this is not the only way, a metamodel with Generics can also be used to enforce WFRs related to type system. In the previous chapter, we checked the constraint of the BETWEEN expression (all parameters should be of same type) using JUnit test. In this chapter we will modify the metamodel and the expression builder to support Generics and we show that violation of the BETWEEN expression

constraint can be captured at compile time and reported to the user through problem markers.

3.2 Metamodel with Generics

Generics are introduced in J2SE 5.0. They enable classes and methods to operate on well defined parametric types that can be substituted by a suitable Java type at the compile time. From EMF 2.3 onwards, Generics can be modeled directly in the Ecore itself. More details on this can be found at [Ric08]. This section explains the stages of adding the generics to relevant classes in the metamodel to support type-safe BETWEEN expression. The type of

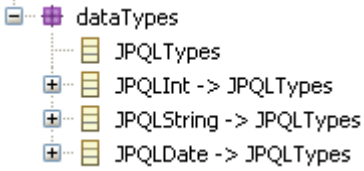

Figure 3.2: *Datatypes of BETWEEN expression*

all three expressions in BETWEEN (`syntax: expr1 [ NOT ] BETWEEN expr2 AND expr3`) can be either arithmetic expression or string expression or date-time expression. To model these types, we added classes `JPQLInt`, `JPQLString`, `JPQLDate` to the metamodel in the package `dataTypes`. These classes inherit the class `JPQLTypes` as shown in Figure 3.2.

The first class in our metamodel that has to be made generic is the class that models the conditional expression (`CondExpr`) as BETWEEN is one of the conditional expression. A type variable "T" that extends the class `JPQLTypes` is added to `CondExpr`. This is shown in emfatic [DG08a] code in Listing 3.1. As type variable "T" extends `JPQLTypes`, it can be instantiated

Listing 3.1: *CondExpr - Generic type*

```
1
2  class CondExpr <T extends dataTypes.JPQLTypes> {
3    val BetExpr<T> betExpr;
4    attr BoolOperators[*] BoolOpr;
5    . . . . .
6  }
```

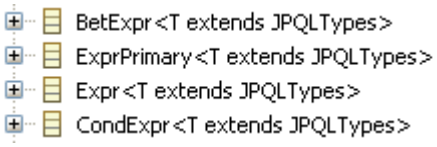

Figure 3.3: *Generic classes*

with the subclasses of `JPQLTypes`. This particular type is then used to instantiate `BetExpr`. For example, `CondExpr<JPQLInt>` leads to `BetExpr<JPQLInt>`. `BetExpr` contains expressions (`Expr`) which in turn contains primary expressions (`ExprPrimary`) and hence the type variable "T" can be used to make these classes generic as well. This is shown in Figure 3.3. Primary expressions can be constructed from state field path expressions and a state field path expression

can end in a simple state field . So classes `StateFieldPathExpr`, `SimpleStateField` are also made generic. As `SimpleStateField` carries the type information, the attribute `datatype` can be omitted to avoid redundancy. The packages in the metamodel which are modified to support generics for type-safe BETWEEN expression is given in the form of emfatic code in Appendix B.

3.3 Expression Builder with Generics

As of now, the expression builder generated by the DSL2JDT tool does not suppport the automatic generation of generic classes and generic methods from the `.genmodel` that has generic classes defined in it. We need to manually change the generated classes and methods to generic type wherever needed in the expression builder. The insight gained in the process of manually changing the expression builder to include generics can be used as input to extend the DSL2JDT tool to support automatic generation of generic classes and methods.

When we generate the expression builder using our new metamodel that has generic classes, we get warnings and problem markers related to generics in the expression builder. One such problem occurrence is shown in Figure 3.4 for the method `simpleStateFields`. The cause

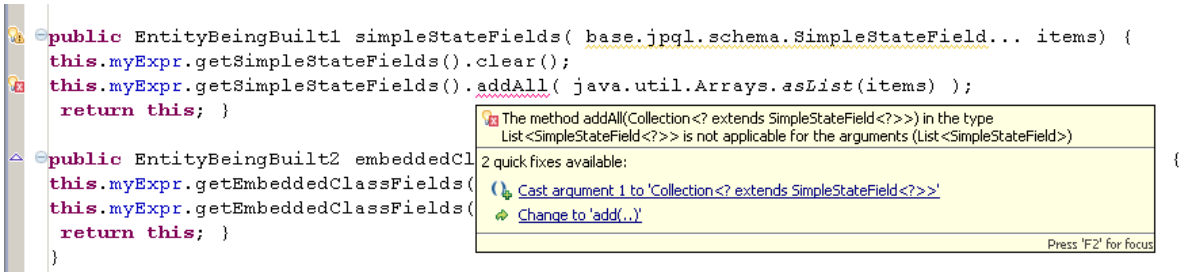

Figure 3.4: *Problem markers in expression builder output of DSL2JDT*

for this error is obvious from the message. As we changed the `SimpleStateField` to generic type in the metamodel, the argument of the method `addAll` is no longer valid for list of raw type, i.e., no longer valid for list of `SimpleStateField` but valid for list of generic type `SimpleStateField<?>`. The solution would be to change the method shown in Figure 3.4 to generic method in the class and in the interface as shown in Listing 3.2. All the remaining errors in the expression builder can be fixed in a similar fashion.

The warnings shown in the expression builder are of the form

`classname` is a raw type. References to generic type `classname<T>` should be parameterized.

These kind of warnings can be removed by making the appropriate classes and methods to generic type. Listing 3.3 shows such modifications for the class `SimpleStateFieldBeingBuilt`.

Listing 3.2: *Generic method in expression builder*

```
public <T extends JPQLTypes> EntityBeingBuilt1 simpleStateFields(
                            base.jpql.schema.SimpleStateField<T>... items) {
        this.myExpr.getSimpleStateFields().clear();
        this.myExpr.getSimpleStateFields().addAll(
            java.util.Arrays.asList(items));
        return this;
 }

public interface EntityBeingBuilt0 {
        public EntityBeingBuilt0 entityName(java.lang.String arg);
        public EntityBeingBuilt0 dbName(java.lang.String arg);
        public <T extends JPQLTypes> EntityBeingBuilt1 simpleStateFields(
                        base.jpql.schema.SimpleStateField<T>... items);
 }
```

Listing 3.3: *Generic class in expression builder*

```
public static class SimpleStateFieldBeingBuilt<T extends JPQLTypes> {
  private final base.jpql.schema.SimpleStateField<T> myExpr;
  public base.jpql.schema.SimpleStateField<T> toAST() {
          return this.myExpr;
  }
  SimpleStateFieldBeingBuilt(base.jpql.schema.SimpleStateField<T> arg) {
          this.myExpr = arg;
  }
  public SimpleStateFieldBeingBuilt<T> name(java.lang.String arg) {
          this.myExpr.setName(arg);
          return this;
  }
}
```

3.3.1 Static Methods

There are new warnings for some of the static methods in the expression builder because of the changes done as per Listing 3.3. One of the static method that gives warning is shown in Listing 3.4. These warnings looks like

Listing 3.4: *Static method in expression builder*

```
public static SimpleStateFieldBeingBuilt simpleStateField() {
        return new SimpleStateFieldBeingBuilt(
                        base.jpql.schema.SchemaFactory.eINSTANCE
                                        .createSimpleStateField());
}
```

The constructor `constructor name` *belongs to the raw type* `classname`*. References to generic type* `classname<T>` *should be parameterized.*

Since we made some of the classes and its constructor method to be of generic type, the constructor call for these classes should also need to be parameterized to overcome such warnings. This is shown in Listing 3.5. The factory method `createSimpleStateField` in Listing 3.4 is generated by EMF as a generic method as shown in Listing 3.6. So invoking this kind of

Listing 3.5: *Parametrized Static method in expression builder*

```
public static <T extends JPQLTypes>
              SimpleStateFieldBeingBuilt<T> simpleStateField() {
        return new SimpleStateFieldBeingBuilt<T>(
                        base.jpql.schema.SchemaFactory.eINSTANCE
                                          .<T> createSimpleStateField());
}
```

factory methods from the expression builder without type instantiation also raises warnings. These warnings can be eliminated if we invoke these factory methods with type instantiation as shown in Listing 3.7

Listing 3.6: *Factory method auto generated by EMF*

```
/**
 * Returns a new object of class '<em>Simple State Field</em>'.
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @return a new object of class '<em>Simple State Field</em>'.
 * @generated
 */
<T extends JPQLTypes> SimpleStateField<T> createSimpleStateField();
```

Listing 3.7: *Factory method invokation for each datatype*

```
SimpleStateField<JPQLString> ssf = E.<JPQLString>simpleStateField()
                                    .name("name").toAST();
```

As we modeled three valid types of BETWEEN expression as classes `JPQLInt`, `JPQLString`, `JPQLDate`, we can invoke the factory method for each type once.

3.4 Constraint Validation in Editor

In the previous chapter, we have validated the constraint of the BETWEEN expression by JUnit test. Now as our metamodel and the expression builder have been modified to support generics, we can use them to perform the validation of the expression and report the errors, if any, to the user through problem markers shown in the editor. For example, we have embedded a JPQL SELECT query with conditional BETWEEN expression as shown in Listing 3.8. This query is not well-formed as the three expressions of BETWEEN are not of same type. This is indicated in the editor by the problem marker as shown in Figure 3.5.

3.5 Proposed Improvement for JPQL embedding

Embedding JPQL in Java involves building a Abstract Syntax Tree (AST) [KT06] of the emdedded expression. Apart from using this AST can be used to check Well-formedness of the JPQL expression, it can also be used to improve the presentation of the program, ie, improving

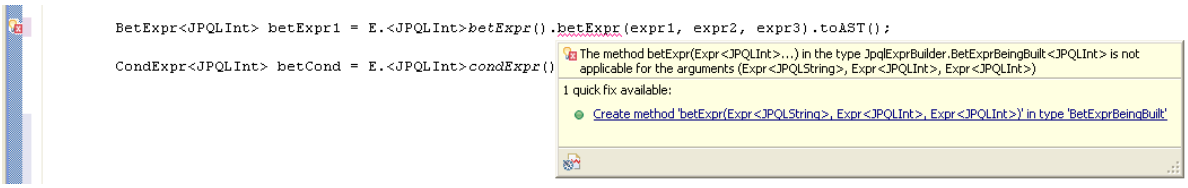

Figure 3.5: *Problem markers for constraint violation*

how the program is displayed to the user. This idea of making programs more expressive is based on the work of Andrew Eisenberg [EK07]. In his work, a language called CAL is embedded in Java and the embedded syntax of CAL is shown in its own editor inside JDT as shown in Figure 3.6.

```
public ProcessedMessage processMessage(Message message) {
    return (ProcessedMessage)
        let
            processorFunction = getNextProcessor message;
            maybeProcessedMessage = processorFunction message;
        in
            case maybeProcessedMessage of
                Just processedMessage -> processedMessage;
                Nothing -> message;
        ;
}
```

Figure 3.6: *CAL embedded in Java*

In our case, we can show the embedded JPQL query in a popup to the user when the cursor is moved over the root node of the embedded expression as shown in Figure 3.7

```
ExprPrimary exprPrim3a = E.exprPrimary().Opr().integerInput(45000)
        .toAST();
Sqrt sqrt2 = E.sqrt().number(exprPrim3a).toAST();
NumRetFn num2 = E.numRetFn().sqrt(sqrt2).toAST();
ExprPrimary exprPrim3b = E.exprPrimary().Opr().numFunc(num2).toAST();
Expr expr3 = E.expr().ExprPrim(exprPrim3b).toAST();

BetExpr betExpr1 = E.betExpr().betExpr(expr1, expr2, expr3).toAST();

CondExpr betCond = E.condExpr().BoolOpr().betExpr(betExpr1).toAST();

SelectStmt selStmt = E.selectStmt().selExpr(selEmp).from(fromExpr)
        .wh
return sels

}

private static
```

SELECT e FROM Employee e WHERE e.salary BETWEEN 40000 AND 45000

Figure 3.7: *Proposed improvement: showing embedded query in popup*

3.6 Summary

In this chapter, we modified the metamodel to support Generics and to aid us in type checking at compile time for embedded JPQL statements. We also showed the steps needed to manually change the methods and classes in the expression builder generated by the DSL2JDT tool, to support the new metamodel. We concluded the chapter by showing an example that generates a problem marker for a BETWEEN expression that is not well formed. So far, our efforts are towards embedding a JPQL query in Java by method chaining approach. If we look at the Microsoft .NET technologies, it provides the querying capabilities by means of LINQ, which is the discussed in next chapter.

Listing 3.8: *Embedded BETWEEN expression - Not Well-formed*

```
public class SelExprGen {
 public static SelectStmt selStmt() {
  base.jpql.schema.Entity emp;
  base.jpql.selectSubQ.SelectExpr selEmp;
  base.jpql.from.RngVarDecl rngVarDecl;
  base.jpql.from.From fromExpr;

  /*
   * Invalid query to check WFR:
   *  JPQL Query to embed
   *  SELECT e FROM employee e WHERE e.name BETWEEN 40000 AND 45000
   */

  selEmp = E.selectExpr().idVar("e").toAST();
  emp = E.entity().entityName("employee").simpleStateFields()
         .embeddedClassFields().toAST();

  rngVarDecl = E.rngVarDecl().entity(emp).as("e").toAST();
  fromExpr = E.from().rngVarDecl(rngVarDecl).join().fetch().toAST();

  SimpleStateField<JPQLString> ssf = E.<JPQLString>simpleStateField()
                                      .name("name").toAST();
  StateFieldPathExpr<JPQLString> sfpe = E.<JPQLString>stateFieldPathExpr()
  .idVar("e").singleValuedAssoField().embeddedClsField().simpleStateField(ssf)
  .toAST();
  ExprPrimary<JPQLString> exprPrim1 = E.<JPQLString>exprPrimary()
                                     .stFldPathExpr(sfpe).Opr().toAST();
  Expr<JPQLString> expr1 = E.<JPQLString>expr().ExprPrim(exprPrim1).toAST();

  ExprPrimary<JPQLInt> exprPrim2 = E.<JPQLInt>exprPrimary().integerInput(40000)
                                      .toAST();
  Expr<JPQLInt> expr2 = E.<JPQLInt>expr().ExprPrim(exprPrim2)
                         .toAST();

  ExprPrimary<JPQLInt> exprPrim3 = E.<JPQLInt>exprPrimary().integerInput(45000)
                                      .toAST();
  Expr<JPQLInt> expr3 = E.<JPQLInt>expr().ExprPrim(exprPrim3).toAST();

  BetExpr<JPQLInt> betExpr1 = E.<JPQLInt>betExpr().betExpr(expr1, expr2, expr3)
                                 .toAST();

  CondExpr<JPQLInt> betCond = E.<JPQLInt>condExpr().betExpr(betExpr1).toAST();

  SelectStmt selStmt = E.selectStmt().selExpr(selEmp).from(fromExpr)
                .where(betCond).groupBy().orderBy().toAST();

  return selStmt;
 }
private static class E extends JpqlExprBuilder {
 }
}
```

Chapter 4

LINQ Expression Trees

Language Integrated Query (LINQ)) is a Microsoft .NET Framework component that adds data querying capabilities to .NET languages such as C# and Visual Basic. LINQ defines a set of query operators that can be used to query, project and filter data in arrays, enumerable classes, XML, relational database, and third party data sources. More details on LINQ can be found at [LIN07] and [Pia08].

We plan to implement a translation algorithm that translates a LINQ query to an equivalent JPQL query. For this translation, we do not plan to support all LINQ operators but only a subset of operators. The LINQ operators [PR07] which can be mapped to a JPQL construct are supported by the translation.

This chapter gives a brief introduction to lambda expressions and expression trees, which are used in LINQ. It also explores the functioning of a tool called *Expression Tree Visualizer* which comes with Visual Studio 2008. Then the techniques used for translating a LINQ query to a native query of the persistent store are discussed. Examples used in this chapter are based on C# language syntax.

4.1 LINQ Queries

LINQ query operations consist of three distinct actions: obtain the data source, create the query and then execute the query. The example in Listing 4.1 shows how the three parts of a query operation are expressed in source code. LINQ queries can be expressed in two forms, namely, *query expression* and *method syntax*. Not all operations can be expressed in query expression syntax, but it is simple and easy to write. query expressions provide nine basic operations (from, join, join...into, group...by, orderby, where, let, select, into). Each of these basic operations is expressed as a clause. The clauses are then chained together to provide a declaration of query. These clauses are translated by the language compliers into invocations of methods that are sequentially applied to the target of the query. Query expression to method syntax translation rules are explained by Wes Dyer in his article on Comprehending Comprehensions [1]

[1] Yet Another Language Geek, `http://blogs.msdn.com/wesdyer/archive/2006/12/21/comprehending-comprehensions.aspx`

Listing 4.1: *LINQ query*

```
class IntroToLINQ
{
    static void Main()
    {
        // The Three Parts of a LINQ Query:
        //  1. Data source.
        int[] numbers = new int[7] { 0, 1, 2, 3, 4, 5, 6 };

        // 2. Query creation using query expression syntax
        // numQuery is an IEnumerable<int>
        var numQuery =
            from num in numbers
            where (num % 2) == 0
            select num;

        // 3. Query execution.
        foreach (int num in numQuery)
        {
            Console.Write("{0,1} ", num);
        }
    }
}
```

4.2 Lambda Expressions and Expression Trees

A Lambda Expression is a evolution of anonymous functions, which allows the body of a function to be written in-line. The basic form of a lambda expression is,

```
argument-list => expression
```

For example, a lambda expression in C# can be written as

```
Func<int,int> LambdaExp = x => (x + 1) * 2
```

Expression trees are built from a LINQ query and can be navigated to process the query in the desired manner, for example, to generate a SQL query from the LINQ query. They consists of nodes in a tree-shaped structure. Each node in the expression tree represents an expression, for example a method call or a binary operation such as x < y. An expression tree can be obtained from a lambda expression by using `Expression<TDelegate>` syntax. For example,

```
Expression<Func<int, int» ExpTree = x => (x + 1) * 2
```

gives an expression tree whose graphical representation is shown in Figure 4.1 It has to be noted that the nodes of an expression tree are immutable. This means that an expression tree can be changed only by creating another one from the existing tree, adding the newly created nodes.

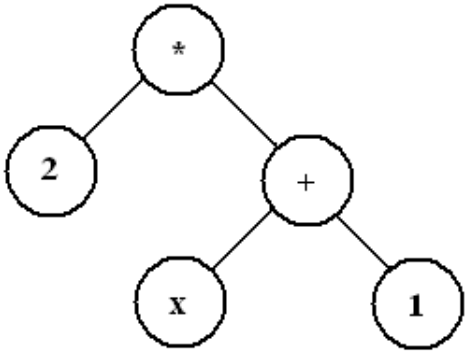

Figure 4.1: *Expression tree - graphical representation*

4.3 Expressions Namespace

In C#, namespaces are used to logically arrange classes, structs, interfaces, enums etc. The .NET framework 3.5 has its own library to handle expression trees, which is formed by classes defined in the `System.Linq.Expressions` namespace. Figure 4.2 shows the classes, interfaces and enumerations that are available in the `System.Linq.Expressions` namespace. These classes can be used directly by users to create expression trees, and also by compilers to translate an expression written in C# or Visual Basic into an expression tree created at runtime. Figure 4.3 shows the `Expression` class and its subclasses. As there are 15 subclasses that

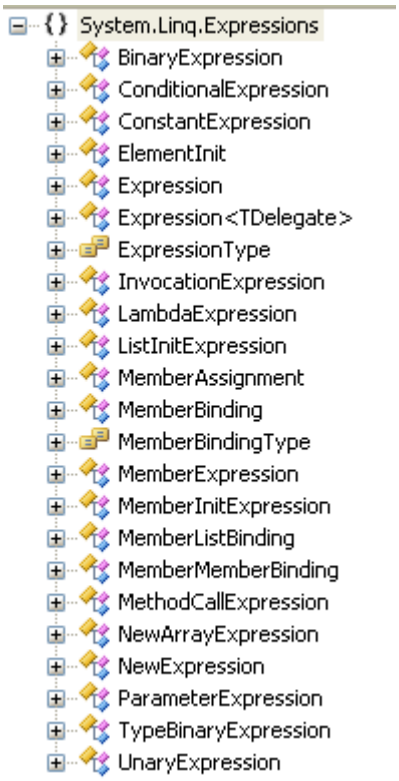

Figure 4.2: *System.Linq.Expressions Namespace*

inherit the `Expression` class, the class diagram is shown in two parts. The abstract class `Expression` provides the root of a class hierarchy used to model expression trees. The classes

in this namespace that derive from `Expression`, are used to represent nodes in an expression tree. The `Expression` class contains static factory methods to create expression tree nodes of the various types. The enumeration type `ExpressionType` specifies the unique node types. For example, *Add* can be a node type for the *BinaryExpression* node.

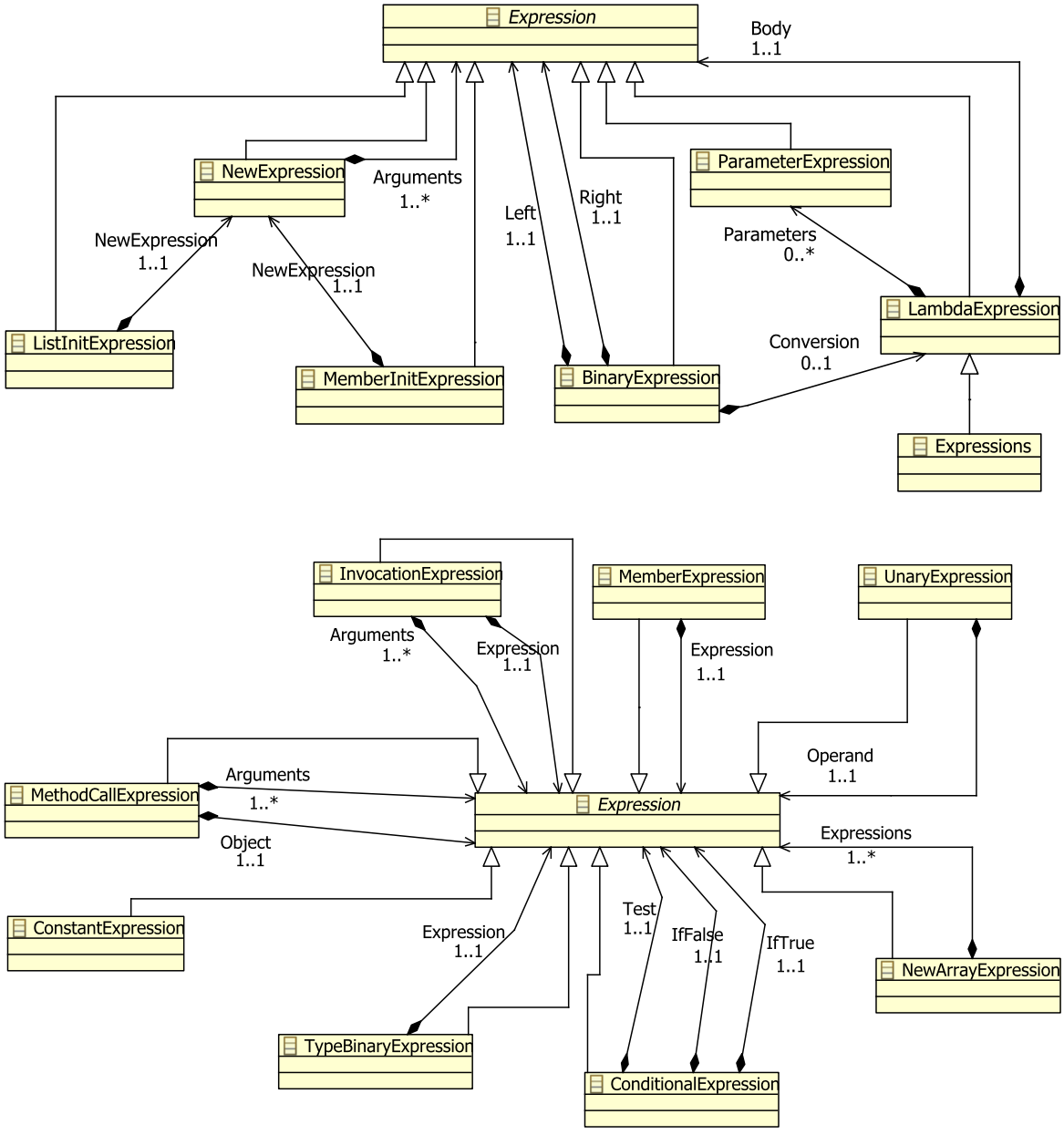

Figure 4.3: *Expression and its subclasses*

4.4 Expression Tree Visualizer

A tool for visualizing the expression trees is available in Visual studio 2008. This tool, called *Expression Tree Visualizer* is useful for analyzing the structure of an expression tree while

debugging. This tool is available as a project in the installation path of the Visual studio, in the folder `LinqSamples`. To start using this tool, the following steps should be followed.

1. Build the Visual studio solution called `ExpressionTreeVisualizer.sln` found in the `ExpressionTreeVisualizer` folder (sub-directory of `LinqSamples`). After the build successfully completes, a file called `ExpressionTreeVisualizer.dll` is created in the `\ExpressionTreeVisualizer\bin\Debug` sub-directory. This is the actual visualizer DLL used by Visual Studio.

2. Copy the `ExpressionTreeVisualizer.dll` to the Visualizer folder. For example, copy it to `My Documents\Visual Studio 2008\Visualizers` folder.

3. Restart the Visual Studio

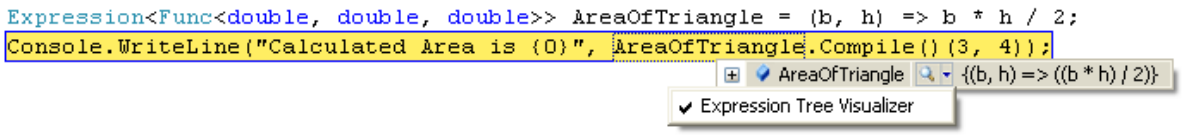

Figure 4.4: *Invoking Expression Tree Visualizer*

After completing the above steps, when debugging an application with an `Expression<T>` instance, having a breakpoint on a line of code with an `Expression<T>`, a magnifying glass icon can be seen (refer Figure 4.4), indicating that a visualizer is available for the object under the cursor. Clicking on the magnifying glass will pop up the Expression Tree Visualizer as shown in Figure 4.5.

4.5 Building an Expression Tree

This section explores the approach followed by the *Expression Tree Visualiser* to build an expression tree. Major part of the work in constructing an expression tree is done by the class `ExpressionTreeNode` in the file `TreeNode.cs`. Its constructor method takes the expression as an argument(in our example, `(b,h)=> b*h/2`). The method `GetType` is used to get the object `Type` of the root node of the expression and the method `ObtainOriginalName(Type)` is invoked on this object to get the name of the type itself. In our case, this name corresponds to `Expression<Func<Double, Double, Double> >`. Then based on the type of expression, its static and instance properties are obtained [using reflection] to further construct the expression tree. The static and instance properties of `Expression` are namely, *Body, Parameters, NodeType* and *Type*. For each of these properties, a subnode (also called `AttributeNode`) is added. For each node that is added, its static and instance properties are obtained and the above process is recursively applied to construct the appropriate subnodes. In this way, the expression tree is built until all the subnodes are processed. The complete expression tree is constructed with the help of two methods, `ExpressionTreeNode(Object value)` and `AttributeNode(Object attribute, PropertyInfo propertyInfo)`. This means that all the different node types are identified while building the tree inside these two methods. This is a special case if we compare with the typical approach of building and processing an expression tree using visitor pattern. In a typical visitor approach, an individual visit method exists for each of the different type of node. Processing code is written inside the visit method,

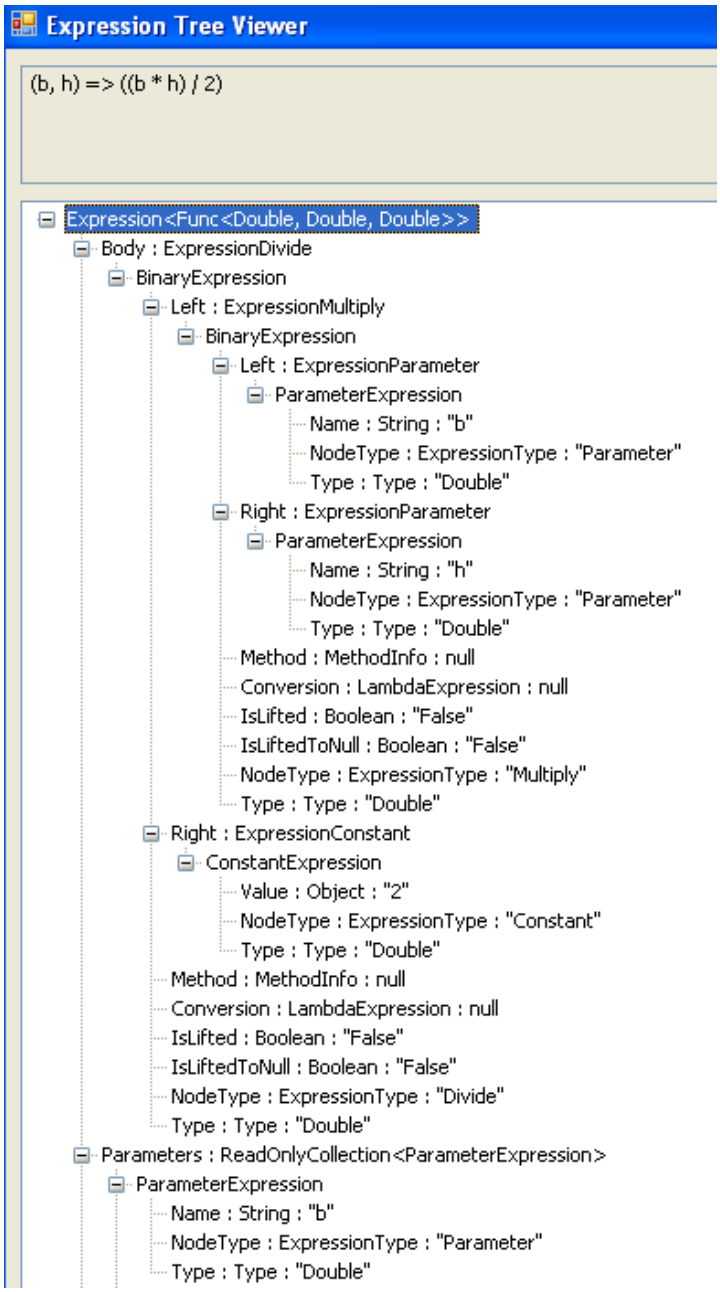

Figure 4.5: *Expression Tree Visualizer*

based on the visited node. The next section describes visitor pattern approach for processing an expression tree.

4.6 Expression Tree Visitor

Visiting an expression tree is useful in many cases. For example, a LINQ to SQL provider uses a visitor pattern to walk through the expression tree identifying different expression types and then constructs an equivalent SQL statement which is executed against the database. LINQ expression tree consists of nodes of different types and the properties of the node affect the nav-

igation of the tree. For instance, `BinaryExpression` node has two properties (`Left`, `Right`) and hence it has two children while the `ConditionalExpression` node has three properties (`Test`, `IfTrue`, `IfFalse`) and hence it has three children in an expression. One approach to handle the differences that arise in navigating an expression tree based on the node type is to write a dedicated visit method for each type of node, like `VisitBinaryExp(exp)` method to handle nodes of type `BinaryExpression` and `VisitConditionalExp(exp)` method for nodes of type `ConditionalExpression`. This pattern is used by the `ExpressionVisitor` abstract class published in the product documentation of Visual Studio 2008. This class has a method `Visit(exp)` that visits a node and recursively calls itself for all the child nodes to visit the entire tree. Listing 4.2 shows a part of the `Visit(exp)` method. As mentioned before, a dedicated

Listing 4.2: *Fragment of a cloning visitor for LINQ expression trees*

```
protected virtual Expression Visit(Expression exp)
   {
       if (exp == null)
           return exp;
       switch (exp.NodeType)
       {
           case ExpressionType.Negate:
           case ExpressionType.NegateChecked:
           case ExpressionType.Not:
           case ExpressionType.Convert:
           case ExpressionType.ConvertChecked:
           case ExpressionType.ArrayLength:
           case ExpressionType.Quote:
           case ExpressionType.TypeAs:
               return this.VisitUnary((UnaryExpression)exp);
           case ExpressionType.Add:
           case ExpressionType.AddChecked:
           case ExpressionType.Subtract:
           case ExpressionType.SubtractChecked:
           case ExpressionType.Multiply:
           case ExpressionType.MultiplyChecked:
           case ExpressionType.Divide:
           case ExpressionType.Modulo:
           case ExpressionType.And:
           case ExpressionType.AndAlso:
           case ExpressionType.Or:
           case ExpressionType.OrElse:
           case ExpressionType.LessThan:
           case ExpressionType.LessThanOrEqual:
           case ExpressionType.GreaterThan:
           case ExpressionType.GreaterThanOrEqual:
           case ExpressionType.Equal:
           case ExpressionType.NotEqual:
           case ExpressionType.Coalesce:
           case ExpressionType.ArrayIndex:
           case ExpressionType.RightShift:
           case ExpressionType.LeftShift:
           case ExpressionType.ExclusiveOr:
               return this.VisitBinary((BinaryExpression)exp);
```

visit method is implemented for each node type like `UnaryExpression`, `BinaryExpression`. Listing 4.3 shows the visit method for `BinaryExpression` which calls `Visit` for *Left, Right* and *Conversion* properties. If any of the internal nodes were changed during the visit, a new `BinaryExpression` instance is created with the new properties. This is the case for non-leaf

nodes. As leaf nodes are those without any children, their visit method does not contain a recursive call to another visit method. Listing 4.4 shows the leaf node `ConstantExpression`. In order to carry out some useful work , the abstract class `ExpressionVisitor` can be inherited to implement particular actions when visiting a node. For instance, Matt Warren, in his series of post on *Building an IQueryable Provider* [War08], has inherited the `ExpressionVisitor` to translate a LINQ query to SQL query.

Listing 4.3: *Non-Leaf node: VisitBinary method*

```
protected virtual Expression VisitBinary(BinaryExpression b)
{
 Expression left = this.Visit(b.Left);
 Expression right = this.Visit(b.Right);
 Expression conversion = this.Visit(b.Conversion);
 if (left != b.Left || right != b.Right || conversion != b.Conversion)
  {
   if (b.NodeType == ExpressionType.Coalesce && b.Conversion != null)
    return Expression.Coalesce(left, right, conversion as LambdaExpression);
   else
    return Expression.MakeBinary(b.NodeType,left,right,b.IsLiftedToNull,b.Method);
  }
 return b;
}
```

Listing 4.4: *Leaf node: VisitConstant method*

```
protected virtual Expression VisitConstant(ConstantExpression c)
  {
    return c;
  }
```

4.7 Partial Evaluation of an Expression Tree

When building an expression tree, the compiler tries to optimize the construction by evaluating the possible expressions to their final value. For instance, if a binary expression involves only constants, then its final value can be computed and used in building the expression tree. This process involves, constructing a new tree for every reduction that can be applied as expression trees are immutable and hence cannot be changed directly. Figure 4.6 shows such a partial evaluation applied to a part of an expression tree. In this example, the compiler calculates the value for the binary expression with only constants (2+1) and substitutes that part of the tree with a new node containing the calculated value (3). In reality, not just one node is added, but all the nodes till the root node in the hierarchy are changed. So in our case, two nodes are modified with respect to the original expression tree, the `BinaryExpression` node with `NodeType` *Multiply* and `ConstantExpression` node with value 3. Matt Warren [War08], gives an example implementation of partial evaluator class that resolves the local variable references in a LINQ query.

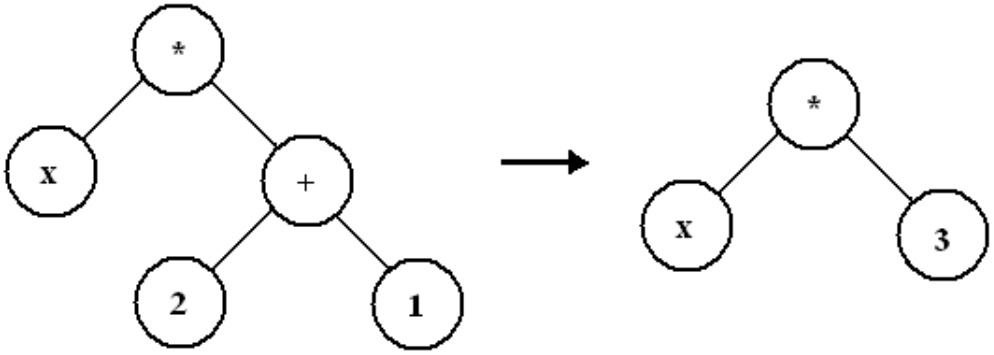

Figure 4.6: *Partial evaluation of an expression tree*

4.8 A Visitor for Implementing Derivatives

To better understand on writing a visitor for an expression tree, Luke Marshall's [Mar08] example of a visitor that implements first order derivative is helpful. This visitor `Derivative` inherits the `ExpressionVisitor` and overrides some of the *Visit* methods to implement the derivative. In particular, the methods *VisitBinary, VisitParamater* and *VisitConstant* are overridden. In the *VisitBinary* method, the node type of is examined to implement the product rule [`(fg)' = f'g + fg'`] for node type *Multiply* and the quotient rule [`(f/g)' = (f'g - fg') / (g*g)`] for the node type *Divide*. The Visitor class is reproduced in Listing 4.5.

4.9 Expression Tree Nodes

This section gives a brief introduction on some of the nodes that are common in an expression tree. In general, understanding these nodes will help us to better analyze and manipulate expression trees. The node types which we are going to describe are *MethodCallExpression, UnaryExpression, Expression<T>, ConstantExpression, BinaryExpression, MemberExpression, ParameterExpression, NewExpression.* To aid in our understanding, we constructed a query that involves all these node types. Suppose `empList` is a list of employee objects, then the query to project all the names and IDs of employees whose salary is 1200 is shown in Listing 4.6 along wih the compiler translated code for the query The diagrammatic representation of expression tree `empQuery.Expression` is shown in Figure 4.7.

MethodCallExpression : The call to methods like `Where, Select` of our expression tree is represented by an instance of *MethodCallExpression* and the value of the NodeType property of a *MethodCallExpression* object is *Call.* This method has a *Method* property pointing to the code to execute, *Object* property pointing to the instance itself, and a collection of *Arguments* which stores the remaining part of the expression tree.

UnaryExpression : It represents an expression that has a unary operator and the value of the NodeType property of *UnaryExpression* in our expression tree is *Quote.* Quoting an expression causes the expression not be evaluated but instead to return the structure of the expression. Lambdas can be converted either to delegates or to expression trees (quoted lambdas)

Listing 4.5: *Visitor to implement first order derivative*

```
class Derivative : ExpressionVisitor
{
    public Expression Eval(Expression exp)
    {
        return this.Visit(Evaluator.PartialEval(exp));
    }

    protected override Expression VisitBinary(BinaryExpression b)
    {
        // Product Rule: (fg)' = f'g + fg'
        if (b.NodeType == ExpressionType.Multiply)
        {
            return Expression.Add(
                Expression.Multiply(Eval(b.Left), b.Right),
                Expression.Multiply(b.Left, Eval(b.Right)));
        }

        // Quotient Rule: (f/g)' = (f'g - fg') / (g*g)
        if (b.NodeType == ExpressionType.Divide)
        {
            return Expression.Divide(
                Expression.Subtract(
                    Expression.Multiply(Eval(b.Left), b.Right),
                    Expression.Multiply(b.Left, Eval(b.Right))),
                Expression.Multiply(b.Right, b.Right));
        }

        return base.VisitBinary(b);
    }

    // Parameter Derivation: f(x)' = 1
    protected override Expression VisitParameter(ParameterExpression p)
    {
        return Expression.Constant(1.0);
    }

    // Constant Rule f(a)' = 0
    protected override Expression VisitConstant(ConstantExpression c)
    {
        return Expression.Constant(0.0);
    }
}
```

depending on the usage. Delegates can be invoked while expression trees preserve the structure of the lambda. This is useful when we need to analyze a lambda expression which should not be evaluated, so that we can modify or translate the query into a form that is useful. For example, translating a LINQ query to SQL query. More details on *Quote* can be found at [Dye08].

Expression<T> : This class is derived from *LambdaExpression* class which is the first node of any expression tree generated from the C/ lambda expression syntax. It has two properties, namely *Body*, which is the first node of the expression defined in the lambda expression and *Parameters* is a collection of *ParameterExpression*, which references the parameters defined in the context of lambda expressions.

Listing 4.6: *C# query and the compiler generated code and IDs*

```
var empQuery = from e in empList.AsQueryable()
           where e.salary == 1200
           select new { e.name, e.id };

ConsoleApplication1.employee[].Where(e => (e.salary = 1200)).Select(e => new
<>f__AnonymousType0`2(name = e.name, id = e.id))
```

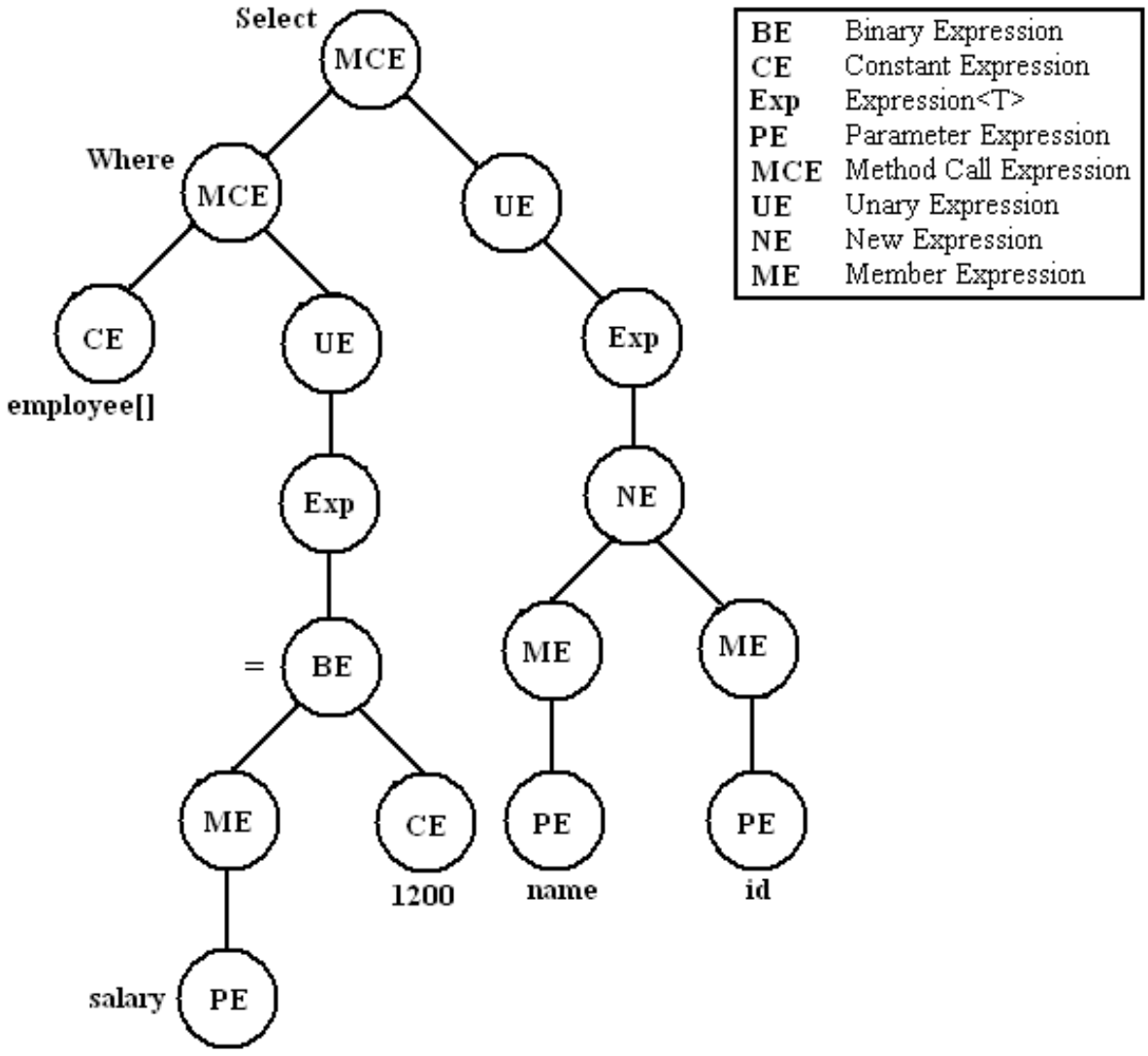

Figure 4.7: *Expression tree for query in Listing 4.6*

ParameterExpression : This class specifies the number and type of arguments involved in a lambda expression and hence is useful when building a delegate from an expression tree (i.e., compiling expression tree). They are always a leaf node in an expression tree.

ConstantExpression : Like *ParameterExpression*, Constant Expression also always appear as leaf nodes in an expression tree. This class has a property *Value* that returns the value stored in the node.

BinaryExpression : The instance of this class represents a binary operator and the value of the NodeType property of *BinaryExpression* can be *Add, Subtract, Multiply, Divide, And, Or, Equal, LessThan, GreaterThan* etc. This class has two properties, *Left* and *Right* which points to the two child nodes of a binary operator.

MemberExpression : The instance of this class represents accessing a field or property. This class has two properties, *Expression* (i.e., the target) and *Member*.

NewExpression : LINQ queries make use of the C# 3.0 feature called *Anonymous Types* to create types on the fly. The instance of the class *NewExpression* is used to represent an anonymous type in expression trees.

4.10 Expression Tree Normalization

While writing a LINQ query, more than one way may exist to perform the same operation. For example, in his blog[2] , Bart De Smet shows different ways of testing Strings for equality, which is reproduced in Listing 4.7. All the four lambda expressions lead to four different expression trees

Listing 4.7: *Different ways to compare Strings*

```
class S
{
    static void Main()
    {
        Test((s1, s2) => s1 == s2);
        Test((s1, s2) => s1.Equals(s2));
        Test((s1, s2) => String.Equals(s1, s2));
        Test((s1, s2) => String.Compare(s1, s2) == 0);
    }

    static void Test(Expression<Func<string, string, bool>> e)
    {
        Console.WriteLine(e);
        Delegate d = e.Compile();
        Console.WriteLine(d.DynamicInvoke("Bart", "John"));
        Console.WriteLine(d.DynamicInvoke("Bart", "Bart"));
    }
}
```

as shown in Listing 4.8. All are LambdaExpression instances, but their *Body* is fundamentally different. The first one is a BinaryExpression with NodeType set to ExpressionType.Equal while the second and the third are MethodCallExpressions and the last one is a mix of the two (the top node will be a BinaryExpression with the left-hand side being a MethodCallExpression to String.Compare and the right-hand side a ConstantExpression with Value set to 0). This gives rise to the idea of normalizing the different expression trees into one common form, as it might be useful when analyzing an expression tree to carry out some processing, like translating

[2] B# .NET Blog, http://bartdesmet.net/blogs/bart/archive/2008/08/13/expression-tree-normalization-how-many-ways-to-say-string-string.aspx

Listing 4.8: *Output of code in Listing 4.7*

```
(s1, s2) => (s1 = s2) False True
(s1, s2) => s1.Equals(s2)
False
True
(s1, s2) => Equals(s1, s2)
False
True
(s1, s2) => (Compare(s1, s2) = 0)
False
True
```

LINQ to a SQL query. Since expression trees are immutable this would imply using a visitor to create a new normalized expression tree.

4.11 Summary

In this chapter, an introduction to LINQ queries, Lambda expressions and Expression trees is provided, followed by a description of the *Expression Tree Visualiser*, a tool that is shipped as part of the Visual studio 2008 for visualizing the structure of an expression tree. An overview of building and visiting an expression tree with appropriate references to other resources was offered. Finally, different types of nodes that are common in an expression tree are described and the importance of expression tree normalization is emphasized. With this background, we will explore the techniques of translating a LINQ query to JPQL query which can be described as a LINQ provider for JPQL, in the next chapter.

Chapter 5

LINQ Query Translation

Developing applications that works with the database systems has become common in software development and hence the developer has to deal with two different worlds, object-oriented domain models and physical database systems. Different query languages exist to handle these different models. For example, JPQL and LINQ can be used to query the logical entity model and SQL can be used to query the relational database systems. Lot of efforts are being made to abstract the physical data model as much as possible, so that the developer can concentrate only on object domain model. At runtime, queries expressed over the object model can be translated into the native query language of the target data store by a component contributed by the DBMS vendor.

Between LINQ and JPQL, the former is more expressive as it adopts a functional-style approach. As LINQ is a Microsoft .NET Framework component and is widely used with .NET languages such as C# and Visual Basic, there is a need for more expressive query language like LINQ for the Java world. As part of this thesis work, we direct our efforts towards developing an integrated query language for Java. This idea is to build expression trees in Java and navigate these expression trees to construct a JPQL query, so that there is no need to modify the compiler.

In this chapter, we present the challenges involved in translating a LINQ query to an equivalent JPQL query and describe a prototype (developed in C# 3.0), that builds a JPQL query string by visiting a LINQ expression tree. The ultimate aim is to develop a LINQ to JPQL provider (that translates LINQ ASTs to JPQL ASTs), based on the model driven approach and to enumerate the pros and cons of such approach. This will involve developing an Ecore metamodel for LINQ and a translation algorithm to transform the instances of such metamodel.

5.1 Query Transformation Challenges

A LINQ to JPQL translation imposes many challenges, deriving from the mismatch between the functional-style of LINQ and the relational algebra roots of JPQL.

1. It should be noted that it is hard to provide support for translating, all the possible LINQ queries to JPQL equivalent. One of the limitations is imposed by the data sources on

which these queries operate. For example, LINQ standard query operators are defined against sequences (ordered collections, allowing duplicates), while the JPQL deals with multisets (unordered collections). Similar issue can also be seen with the LINQ to SQL provider, which does not provide support of all LINQ operators. So the first challenge is to identify the subset of LINQ operators that we plan to support for the translation to LINQ.

2. As we saw previously, JPQL has its own grammar that defines syntactically correct queries. This means that the job of translation algorithm is not only to generate a LINQ equivalent query but also to make sure that the generated query adheres to the rules of the JPQL grammar. This means we should be aware of the constraints imposed by the JPQL grammar while devising a translation algorithm. For example, as per the current Early Draft 1 on JPA 2.0 (JSR 317), subqueries are restricted to the WHERE and HAVING clauses and are not allowed in the FROM clause. Even within the WHERE and HAVING clauses, subqueries are not always allowed. For example, subqueries are not allowed in IS[NOT]NULL, IS[NOT]EMPTY, [NOT]MEMBER[OF] JPQL expressions that are used in WHERE and HAVING clauses. The translation algorithm should take this into account while performing the translation. One more important observation is the SELECT clause in LINQ queries is optional, while it is mandatory in JPQL query. This implies that even for LINQ queries without SELECT clause (having only GROUPBY), the translation has to generate relevant SELECT clause in JPQL.

3. In LINQ, nearly all the operators are overloaded. For instance, the *Where* operator has two overloaded methods. This means it is not sufficient to decide on the operators to support but also to decide on the overloaded versions of the operator to support. This may not be trivial. Consider the LINQ *OrderBy* operator whose overloaded methods are

Listing 5.1: *Overloaded OrderBy Operator*

```
public static IOrderedEnumerable<TSource> OrderBy<TSource, TKey>(
     this IEnumerable<TSource> source,
     Func<TSource, TKey> keySelector);

public static IOrderedEnumerable<TSource> OrderBy<TSource, TKey>(
     this IEnumerable<TSource> source,
     Func<TSource, TKey> keySelector, IComparer<TKey> comparer);
```

shown in Listing 5.1. The only difference between the two methods is that the second version takes a *comparer* as an additional argument. The *keySelector* extracts a key, which will be used by the *comparer* for ordering. The second method allows us to provide a custom comparer. This means that the users can write their own comparer (as a lambda expression). If no comparer is provided or the *comparer* argument is null, the default comparer is used. But when the user provides an instance of his own comparer, then the problem is to translate the semantics of this user defined comparer. If this problem is not solved, then the support given by translation is restricted to operators without custom comparer or custom comparer set to null.

4. In Chapter 2, we have seen that the JPQL grammar alone does not capture all of the constraints of the query language. So generating a JPQL query adhering to the grammar alone (as in step 2) does not guarantee the correctness of the query. One example of such scenario where the translated JPQL query is not well-formed is shown in Listing 5.2. The query is not well-formed as it violates the WFR 13 of our list in Chapter 2 (section 2.4).

Listing 5.2: *Translated JPQL query - Not well-formed*

```
LINQ query is :
      var query = from e in empList.AsQueryable()
      where e.id > 2
      orderby e.salary
      select e.name;

Translated JPQL query is :
      SELECT e.name FROM employee AS e WHERE (e.id > 2) ORDER BY e.salary
```

The challenges for LINQ to JPQL translation can be summarized as follows: Identifying the subset of LINQ operators to support, deciding upon the overloaded methods to support for this subset, adhering to the grammar and well-formedness rules of JPQL and appropriately handling the scenarios for which the translation is not supported.

5.2 Prototype to Generate JPQL Query

This section describes the C# 3.0 program developed for translating a LINQ query to an equivalent JPQL query. This program uses a approach that is similar to that described by Matt Warren [War08] in his article on *Building an IQueryable Provider* that gives an example for LINQ provider for SQL. One major difference is with respect to the way the query is built. In the SQL provider, query is built by continuously appending strings to a stringbuilder when the nodes of an expression tree are visited. In our prototype (JPQL provider), the query is built in parts and finally concatenated to form a complete JPQL query.

The prototype works as follows: It has an abstract class `ExpressionVisitor` which is inherited by class `Linq2Jpql`. This inherited class has few methods that override the methods in the base class to construct the JPQL query string. Translation begins with the invocation of the method

Listing 5.3: *Building a JPQL query string*

```
1 internal string Translate(Expression expression)
2 {
3  this.sb = new StringBuilder();
4  this.Visit(expression);
5  if (sel == null)
6       sel = "Select * ";
7  return (sel + frm + whr + ordBy + grpBy);
8 }
```

`Translate` (Listing 5.3) in the class `Linq2Jpql` that takes the expression to be translated as a argument. This expression is recursively visited and based on the visited node type, the overridden methods in `Linq2Jpql` build the JPQL query in parts . After visiting the entire expression tree, all the JPQL query parts (here, *sel, frm, whr, ordBy, grpBy*) that are constructed so far are suitably concatenated (line 6 in Listing 5.3) to form a JPQL query. The entire code can be found in the Appendix C.

One of the overridden methods that is of interest is `VisitMethodCall` (Listing 5.4). It is invoked for the node type `MethodCallExpression`. Based on the method that is invoked, a

Listing 5.4: *Visiting node of type MethodCallExpression*

```
protected override Expression VisitMethodCall(MethodCallExpression m)
{
        // some code here

    if (m.Method.Name == "Where")
    {
     sb.Append(" WHERE ");
     LambdaExpression lambda = (LambdaExpression)StripQuotes(m.Arguments[1]);
     this.Visit(lambda.Body);

     whr = sb.ToString();
     ClearStringBuilder(sb);
     this.Visit(m.Arguments[0]);
     return m;
    }

        // some code here
}
```

query string is constructed by visiting the `Arguments` of the node type.

This program is not complete and limitations do exist (for example, a query with *SelectMany* operator is not supported; only one entity can be queried in a single query), but it gives an insight for better understanding the expression trees and handling them to generate queries for the target data source.

5.3 Realization of Standard Query Operators

This section describes the metamodel developed for enabling LINQ like capability in Java. It can be used to express queries (instances of metamodel) that are later translated to JPQL by AST-to-AST rewriting techniques. To start with a metamodel for a query language, we would first need a data model. When we developed a metamodel for JPQL in Chapter 2, the data model we used is described by the package *Schema*. For LINQ metamodel, we use the Ecore.ecore as a base data model. The motivation for using the Ecore.ecore comes from the OCL metamodel (OCL.ecore). A query which is an instance of LINQ metamodel is translated by the translation algorithm to an instance of JPQL metamodel.

The entire metamodel is organized in three packages: *types, basics* and *operators* as shown in Figure 5.1. Package *types* has a class `PrimitiveType` that inherits from the Ecore class `EDataType` to represent the basic data types. Class `Entity` inherits from `Eclass` and `Field` inherits from `EAttribute`. The package *basics* has classes whose instances will be used by the supported LINQ operators. The subset of LINQ operators and their overloads that are chosen for translating to JPQL is included in the package *operators*. The LINQ like metamodel described above, can be used to express queries over the entity model using method invocation syntax. The AST of such queries can be translated to JPQL using AST-AST rewriting methods, which is sent to the database engine for evaluation.

One of the most important contribution towards developing a integrated query language for

Java is made by the project JaQue [1]. JaQue provides an infrastructure for Microsoft LINQ like capabilities on Java platform. Using Java bytecode manipulation techniques, JaQue builds expression trees, which can be translated later to native query language of the database.

One of the applications for the in-memory realization of the standard query operators in Java can be for EMF persistence. The changes that are needed to the architecture of EMF persistence are not simple as the job of Object Relational Mapping (ORM) engine has also need to be considered. More details can be found in [GP08].

5.4 Summary

In this chapter, we presented the ideas around developing LINQ like query language for Java. The challenges involved in the LINQ to JPQL query translation are enumerated, followed by the prototype developed in C/ 3.0. Though the prototype has some limitations, it serves as a base to understand the query transformation process. We also developed an Ecore metamodel for realizing the LINQ standard query operators for Java.

[1] JaQue, `http://code.google.com/p/jaque/`

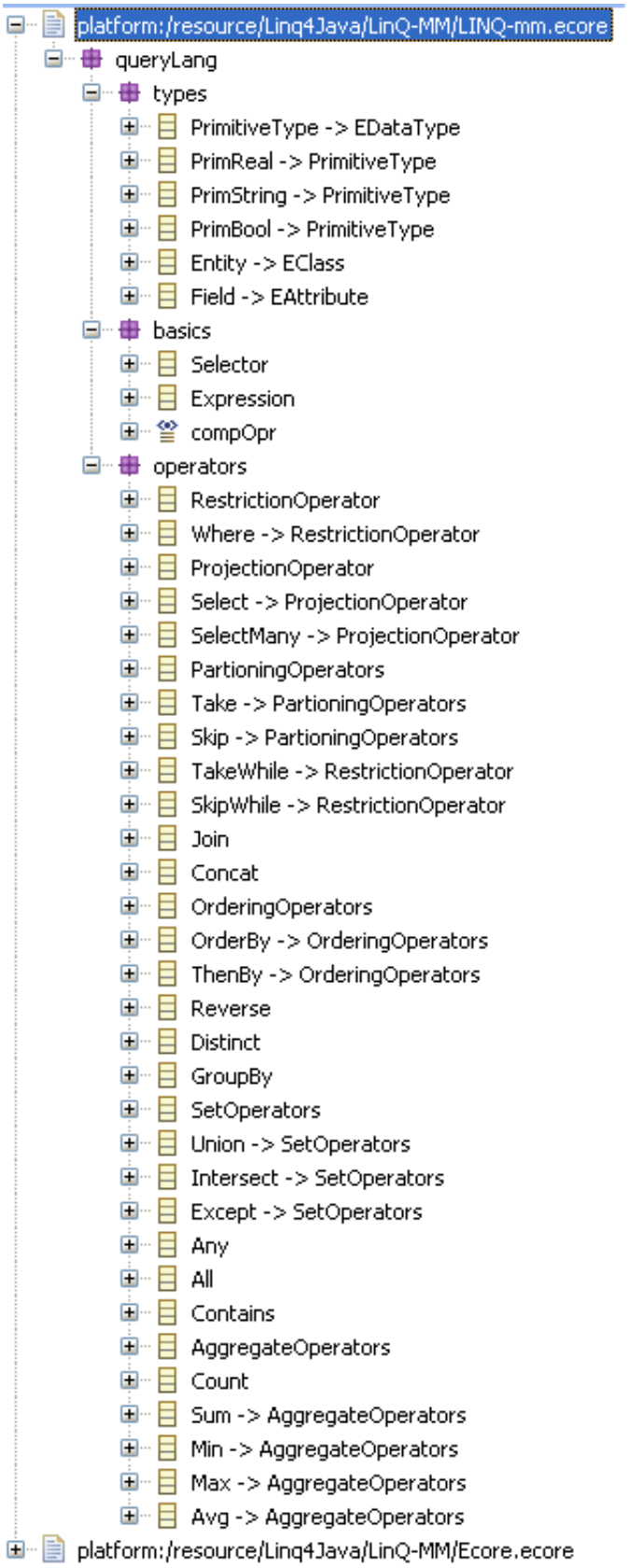

Figure 5.1: *LINQ metamodel*

Chapter 6

Conclusions and Future Works

In this chapter, we give the conclusions and then we present ideas that evolved during the course of this thesis work and that are probable candidates for future work.

6.1 Conclusions

The first objective of this master thesis was to embed JPQL in a typesafe manner in Java adopting the Model Driven approach. Aligned with this goal, we introduced the concepts revolving around DSL in Chapter 1. This chapter also introduced the second objective of developing an integrated query language for Java, similar to LINQ for Microsoft .NET framework languages like C# and VB.

String-based queries can cause negative impact on programmer's productivity, as these queries are not accessible to development environment features like compile-time type checking, autocompletion, and refactoring. This was the main motivation behind our efforts to implement JPQL as an internal DSL. We relied on the most favored approach for embedding a DSL, the fluent interface or method chaining approach and developed an Ecore metamodel in Chapter 2. The code generation capability of Eclipse Modeling Framework(EMF) combined with the DSL2JDT tool provided the necessary APIs required for method chaining. The approach we followed is not limited to a query language and can be used to embed any DSL in general purpose programming languages like Java. This can be achieved by building an Ecore metamodel from the DSL grammar. Though we have done this manually for JPQL embedding, frameworks like Gymnast [DG08b] can be used to automatically create an Ecore metamodel from the grammar. Apart from automatically generating Java classes and methods from metamodel by using EMF and DSL2JDT, another advantage is the specification of constraints in the Ecore model itself as Annotations (for example, as OCL constraints). In Chapter 3, we showed the usage of Java Generics to implement the type checking at compile time for embedded queries and to generate problem markers for syntactically invalid queries.

We identified few shortcomings of typesafe embedding of JPQL and one of them is verbose query embedding, but this additional noise in the host language can be reduced by the proposed techniques like using intermediate classes. Also in JPQL, a query may either be dynamically specified at runtime (as string) or configured in metadata (annotation or XML) and referenced by

name (Named Queries). Our present approach of JPQL embedding considers only the dynamic queries and in this aspect it lags behind the support given by Dali JPA Tools Project [Tea08] for named queries. ORM products like Hibernate and Oracle TopLink are widely used for Java persistence and they have their own proprietary APIs that support additional features that are not part of JPA 2.0 and hence JPQL embedding scores less against the query language like HQL (Hibernate Query Language). With a number of significant changes are expected for JPA 2.0[1] and hence for JPQL, embedding of JPQL (based on JSR 317 which is an Early Draft Review as of now) is less attractive. All these limitations of JPQL embedding, paved way for the second part of this thesis work, realizing a functional query language for Java, similar to LINQ. As a step towards this, in Chapter 4, we analyzed the LINQ language and provided an overview of building and visiting an Expression tree.

Developing an integrated query language differs from embedding a query language in one important aspect. In JPQL embedding, we had the language grammar from which metamodel is constructed and methods for fluent interfaces are subsequently generated. But this is only one part of the effort needed for implementing an integrated query language. For the other part, we also need to consider the task of ORM mapping. For example, in LINQ to SQL, details for ORM mapping is specified in the code by means of metadata decorated to the class definitions. As doing such O/R mapping by ourselves demands large efforts, we limited ourselves to LINQ → JPQL translation in our first prototype. Chapter 5 explains the challenges and the techniques for developing such a prototype. In order to fully realize the query translation and optimization techniques that functional query languages call for, we see the need for a contribution by the providers of ORM engines in future.

6.2 Future Works

6.2.1 Benchmarking of Query Processor for Model Repositories

Model repositories are Object Oriented DBMSs that are used to persist EMOF (Essential Meta Object Facility). The Ecore that has been defined in the Eclipse Modeling Framework(EMF) is more or less aligned to EMOF. EMF Objects can be stored and retrieved using queries (expressed in LINQ or OCL) in Teneo, which is a database persistency solution for EMF using Hibernate or JPOX/JDO 2.0. The queries can be evaluated on client side or they can be moved to server side for evaluation (query shipping) for large repositories. In this case, LINQ or OCL are translated into the query language of the database engine. In order to avoid the additional work of query translation, one possibility would be to choose a query language that is as expressive as LINQ or OCL and at the same time can be processed on the database engine without the need for translation. One such query language is XQuery which can be processed by a query engine called *Pathfinder*[2]. As a future work, the performance of pathfinder query processor can be evaluated when used a model repository for large EMF models.

[1] Looking Forward to JPA 2.0, `http://java.dzone.com/articles/looking-forward-jpa-20`
[2] Pathfinder, `http://www.pathfinder-xquery.org/`

6.2.2 Development of Query Optimizer

In the previous chapter, we described the translation from LINQ to JPQL as one of the ways to provide integrated query functionality to Java and did not provide any optimizations for such queries. Optimization is the key component of native queries (Figure 6.1). Users should be able to write native query expressions and the database should execute them with performance on par with the string-based queries. The inspiration for developing a query optimizer as part of future work comes from the University of Texas project, titled *lambda-DB*[3]. This project aims at developing frameworks and prototype systems that address the query optimization challenges for OODBs.

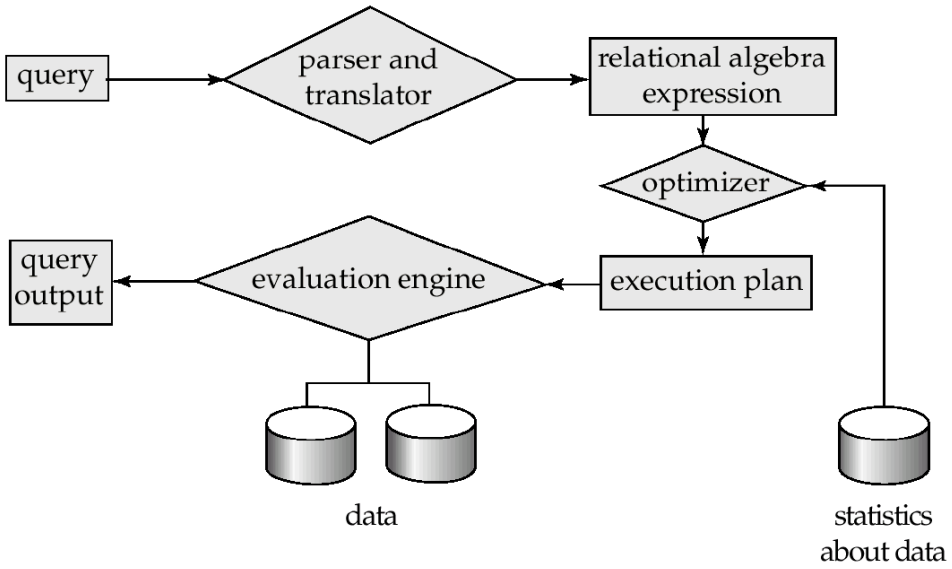

Figure 6.1: *Basic steps of query processing*

[3]lambda-DB: An ODMG-Based Object-Oriented DBMS, `http://lambda.uta.edu/lambda-DB/manual/overview.html`

Appendix A

BNF Grammar for JPQL

QL_statement ::= select_statement | update_statement | delete_statement

select_statement ::= select_clause from_clause [where_clause] [groupby_clause] [having_clause] [orderby_clause]

update_statement ::= update_clause [where_clause]

delete_statement ::= delete_clause [where_clause]

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

identification_variable_declaration ::= range_variable_declaration { join | fetch_join }*

range_variable_declaration ::= abstract_schema_name [AS] identification_variable

join ::= join_spec join_association_path_expression [AS] identification_variable

fetch_join ::= join_spec FETCH join_association_path_expression

association_path_expression ::= collection_valued_path_expression | single_valued_association_path_expression

join_spec::= [LEFT [OUTER] | INNER] JOIN

join_association_path_expression ::= join_collection_valued_path_expression | join_single_valued_association_path_expression

join_collection_valued_path_expression::= identification_variable.collection_valued_association_field

join_single_valued_association_path_expression::= identification_variable.single_valued_association_field

collection_member_declaration ::= IN (collection_valued_path_expression) [AS] identification_variable

single_valued_path_expression ::=
state_field_path_expression | single_valued_association_path_expression

state_field_path_expression ::= {identification_variable |
single_valued_association_path_expression}.state_field

single_valued_association_path_expression ::= identification_variable.
{single_valued_association_field.}* single_valued_association_field

collection_valued_path_expression ::= identification_variable.
{single_valued_association_field.}* collection_valued_association_field

state_field ::= {embedded_class_state_field.}* simple_state_field

update_clause ::= UPDATE abstract_schema_name [[AS] identification_variable]
SET update_item {, update_item}*

update_item ::= [identification_variable.]{state_field | single_valued_association_field} =
new_value

new_value ::= simple_arithmetic_expression | string_primary | datetime_primary |
boolean_primary | enum_primary simple_entity_expression | NULL

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

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

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

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

constructor_item ::= single_valued_path_expression | aggregate_expression

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

where_clause ::= WHERE conditional_expression

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

groupby_item ::= single_valued_path_expression | identification_variable

having_clause ::= HAVING conditional_expression

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

orderby_item ::= state_field_path_expression [ASC | DESC]

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

subquery_from_clause ::= FROM subselect_identification_variable_declaration
{, subselect_identification_variable_declaration}*

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

simple_select_clause ::= SELECT [DISTINCT] simple_select_expression

simple_select_expression::= single_valued_path_expression | aggregate_expression | identification_variable

conditional_expression ::= conditional_term | conditional_expression OR conditional_term

conditional_term ::= conditional_factor | conditional_term AND conditional_factor

conditional_factor ::= [NOT] conditional_primary

conditional_primary ::= simple_cond_expression | (conditional_expression)

simple_cond_expression ::= comparison_expression | between_expression | like_expression
| in_expression | null_comparison_expression | empty_collection_comparison_expression |
collection_member_expression | exists_expression

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

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

in_item ::= literal | input_parameter

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

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

empty_collection_comparison_expression ::= collection_valued_path_expression IS [NOT]
EMPTY

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

exists_expression::= [NOT] EXISTS (subquery)

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

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

all_or_any_expression} | arithmetic_expression comparison_operator {arithmetic_expression | all_or_any_expression}

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

arithmetic_expression ::= simple_arithmetic_expression | (subquery)

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

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

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

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

string_expression ::= string_primary | (subquery)

string_primary ::= state_field_path_expression | string_literal | input_parameter | functions_returning_strings | aggregate_expression

datetime_expression ::= datetime_primary | (subquery)
datetime_primary ::= state_field_path_expression | input_parameter | functions_returning_datetime | aggregate_expression

boolean_expression ::= boolean_primary | (subquery)
boolean_primary ::= state_field_path_expression | boolean_literal | input_parameter |

enum_expression ::= enum_primary | (subquery)
enum_primary ::= state_field_path_expression | enum_literal | input_parameter |

entity_expression ::= single_valued_association_path_expression | simple_entity_expression

simple_entity_expression ::= identification_variable | input_parameter

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

functions_returning_datetime ::= CURRENT_DATE| CURRENT_TIME | CURRENT_TIMESTAMP

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

trim_specification ::= LEADING | TRAILING | BOTH

Appendix B

JPQL Metamodel with Generics

```
@namespace(uri="http://de.tuhh.sts/jpql", prefix="jpql")
package jpql;
@namespace(uri="schemaU", prefix="schemaP")
package schema {
  @Ecore(constraints="PersistentUnitTest")
  class PersistentUnit {
    val Entity[*]#ownerUnit entities;
  }
  class Entity {
    attr String entityName;
    attr String dbName;
    val SimpleStateField<?>[*]#owner simpleStateFields;
    val EmbeddedClassField[*]#owner embeddedClassFields;
    ref PersistentUnit#entities ownerUnit;
  }
  abstract class SimpleOrEmbeddedField {
    attr String[1] name;
  }
  class SimpleStateField <T extends myDataTypes.MyTypes> extends SimpleOrEmbeddedField {
    ref Entity#simpleStateFields owner;
  }
  class EmbeddedClassField extends SimpleOrEmbeddedField {
    ref EmbeddedClass classType;
    ref Entity#embeddedClassFields owner;
  }
  class EmbeddedClass extends Entity {
  }
  class AbstractSchema extends Entity {
    val LOBField[*]#owner largeObjs;
    val AssocField[*]#owner assocFields;
  }
  class LOBField extends SimpleOrEmbeddedField {
    ref AbstractSchema#largeObjs owner;
  }
  class AssocField {
    attr String name;
    attr Multiplicity multiplicity;
    ref AbstractSchema type;
    ref AbstractSchema#assocFields owner;
  }
  class SingleValuedAssocField extends AssocField {
  }
  class CollValuedAssocField extends AssocField {
```

```
  }
  class SupportedJavaType {
    attr String name;
  }
  enum Multiplicity {
    GT1 = 0;
    LE1 = 0;
  }
}

@namespace(uri="pathExprU", prefix="pathExprP")
package pathExpr {
  @Ecore(constraints="pathExprTest")
  abstract class PathExp extends primaries.IdVar {
    attr String[1] idVar;
    val schema.SingleValuedAssocField[*] singleValuedAssoField;
  }
  class StateFieldPathExpr <T extends myDataTypes.MyTypes> extends PathExp {
    val schema.EmbeddedClassField[*] embeddedClsField;
    val schema.SimpleStateField<T>[1] simpleStateField;
  }
  class SingleValuedAssoPathExpr extends PathExp {
    val schema.SingleValuedAssocField[1] singleValAssoField;
  }
  class SingleValPathExp {
    val StateFieldPathExpr<?> stFldPthExpr;
    val SingleValuedAssoPathExpr singleValuedAssoPathExpr;
  }
  class CollValuedPathExpr extends PathExp {
    val schema.CollValuedAssocField[1] collValuedAsso;
  }
}

@namespace(uri="expressionsU", prefix="expressionsP")
package expressions {
  class AllAnyExpr {
    attr AllAny allAnySome;
    val selectSubQ.SubQuery subQuery;
  }
  class AggExp {
    attr Aggregate Aggr;
    attr String idVar;
    val pathExpr.StateFieldPathExpr<?> stFldPathExpr;
    val pathExpr.SingleValuedAssoPathExpr singleValAssoPathExpr;
  }
  class ExprPrimary <T extends myDataTypes.MyTypes> {
    op jpqlType computeType();
    val pathExpr.StateFieldPathExpr<T> stFldPathExpr;
    val functions.NumRetFn numFunc;
    val functions.StrRetFn strFunc;
    attr DateTimeFn dateTimefn;
    val ExprPrimary<T> simpleArithExpr;
    val AggExp aggExp;
    val primaries.InputParam inputParam;
    attr String idVar;
    attr int integerInput;
    attr String stringInput;
    attr ArithOperators[*] Opr;
  }
  class Expr <T extends myDataTypes.MyTypes> {
    val ExprPrimary<T> ExprPrim;
    val selectSubQ.SubQuery subQuery;
```

```
  }
  @Ecore(constraints="sameTypeExpr")
  class BetExpr <T extends myDataTypes.MyTypes> {
    val Expr<T>[3] betExpr;
  }
  class CondExpr <T extends myDataTypes.MyTypes> {
    val BetExpr<T> betExpr;
    attr BoolOperators[*] BoolOpr;
  }
  enum ArithOperators {
    Plus = 0;
    Minus = 1;
    Multiply = 2;
    Divide = 3;
  }
  enum BoolOperators {
    And = 0;
    Or = 1;
    Not = 2;
  }
  enum DateTimeFn {
    CtDate = 0;
    CtTime = 1;
    CtTimeStamp = 2;
  }
  enum AllAny {
    All = 0;
    Any = 1;
    Some = 2;
  }
  enum Aggregate {
    Avg = 0;
    Max = 1;
    Min = 2;
    Sum = 3;
    Count = 4;
  }
  enum jpqlType {
    String = 0;
    Numeric = 1;
    Date = 2;
    Boolean = 3;
    Enum = 4;
    Int = 5;
    Long = 6;
    Double = 7;
    Char = 8;
    Dummy = 9;
  }
}

@namespace(uri="myDataTypesU", prefix="myDataTypesP")
package myDataTypes {
  class MyTypes {
  }
  class MyInt extends MyTypes {
  }
  class MyString extends MyTypes {
  }
  class MyDate extends MyTypes {
  }
}
```

Appendix C

LINQ to JPQL translation - C# Prototype

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;
using System.Text;
using System.Linq.Expressions;
using System.Collections.ObjectModel;

namespace JPQLquery
{
 public abstract class ExpressionVisitor
 {
  protected ExpressionVisitor()
  {
  }

  protected virtual Expression Visit(Expression exp)
  {
    if (exp == null)
        return exp;
    switch (exp.NodeType)
    {
      case ExpressionType.Negate:
      case ExpressionType.NegateChecked:
      case ExpressionType.Not:
      case ExpressionType.Convert:
      case ExpressionType.ConvertChecked:
      case ExpressionType.ArrayLength:
      case ExpressionType.Quote:
      case ExpressionType.TypeAs:
           return this.VisitUnary((UnaryExpression)exp);

      case ExpressionType.Add:
      case ExpressionType.AddChecked:
      case ExpressionType.Subtract:
      case ExpressionType.SubtractChecked:
      case ExpressionType.Multiply:
      case ExpressionType.MultiplyChecked:
      case ExpressionType.Divide:
      case ExpressionType.Modulo:
```

```
    case ExpressionType.And:
    case ExpressionType.AndAlso:
    case ExpressionType.Or:
    case ExpressionType.OrElse:
    case ExpressionType.LessThan:
    case ExpressionType.LessThanOrEqual:
    case ExpressionType.GreaterThan:
    case ExpressionType.GreaterThanOrEqual:
    case ExpressionType.Equal:
    case ExpressionType.NotEqual:
    case ExpressionType.Coalesce:
    case ExpressionType.ArrayIndex:
    case ExpressionType.RightShift:
    case ExpressionType.LeftShift:
    case ExpressionType.ExclusiveOr:
        return this.VisitBinary((BinaryExpression)exp);

    case ExpressionType.TypeIs:
        return this.VisitTypeIs((TypeBinaryExpression)exp);

    case ExpressionType.Conditional:
        return this.VisitConditional((ConditionalExpression)exp);

    case ExpressionType.Constant:
        return this.VisitConstant((ConstantExpression)exp);

    case ExpressionType.Parameter:
        return this.VisitParameter((ParameterExpression)exp);

    case ExpressionType.MemberAccess:
        return this.VisitMemberAccess((MemberExpression)exp);

    case ExpressionType.Call:
        return this.VisitMethodCall((MethodCallExpression)exp);

    case ExpressionType.Lambda:
        return this.VisitLambda((LambdaExpression)exp);

    case ExpressionType.New:
        return this.VisitNew((NewExpression)exp);

    case ExpressionType.NewArrayInit:
    case ExpressionType.NewArrayBounds:
        return this.VisitNewArray((NewArrayExpression)exp);

    case ExpressionType.Invoke:
        return this.VisitInvocation((InvocationExpression)exp);

    case ExpressionType.MemberInit:
        return this.VisitMemberInit((MemberInitExpression)exp);

    case ExpressionType.ListInit:
        return this.VisitListInit((ListInitExpression)exp);

    default:
        throw new Exception(string.Format("Unhandled expression type: '{0}'",
            exp.NodeType));
  }
}

protected virtual MemberBinding VisitBinding(MemberBinding binding)
{
```

```
    switch (binding.BindingType)
    {
     case MemberBindingType.Assignment:
          return this.VisitMemberAssignment((MemberAssignment)binding);

     case MemberBindingType.MemberBinding:
          return this.VisitMemberMemberBinding((MemberMemberBinding)binding);

     case MemberBindingType.ListBinding:
           return this.VisitMemberListBinding((MemberListBinding)binding);

     default:
          throw new Exception(string.Format("Unhandled binding type '{0}'",
                binding.BindingType));
    }
  }

protected virtual ElementInit VisitElementInitializer(ElementInit initializer)
{
   ReadOnlyCollection<Expression> arguments =
                       this.VisitExpressionList(initializer.Arguments);
   if (arguments != initializer.Arguments)
            return Expression.ElementInit(initializer.AddMethod, arguments);

   return initializer;
}

protected virtual Expression VisitUnary(UnaryExpression u)
{
   Expression operand = this.Visit(u.Operand);
   if (operand != u.Operand)
     return Expression.MakeUnary(u.NodeType, operand, u.Type, u.Method);

   return u;
}

protected virtual Expression VisitBinary(BinaryExpression b)
{
   Expression left = this.Visit(b.Left);
   Expression right = this.Visit(b.Right);
   Expression conversion = this.Visit(b.Conversion);

   if (left != b.Left || right != b.Right || conversion != b.Conversion)
   {
      if (b.NodeType == ExpressionType.Coalesce && b.Conversion != null)
         return Expression.Coalesce(left, right, conversion as LambdaExpression);
      else
         return Expression.MakeBinary(b.NodeType, left, right,
                                      b.IsLiftedToNull, b.Method);
   }
   return b;
  }

protected virtual Expression VisitTypeIs(TypeBinaryExpression b)
{
   Expression expr = this.Visit(b.Expression);
   if (expr != b.Expression)
      return Expression.TypeIs(expr, b.TypeOperand);

   return b;
}
```

```
protected virtual Expression VisitConstant(ConstantExpression c)
{
   return c;
}

protected virtual Expression VisitConditional(ConditionalExpression c)
{
   Expression test = this.Visit(c.Test);
   Expression ifTrue = this.Visit(c.IfTrue);
   Expression ifFalse = this.Visit(c.IfFalse);
   if (test != c.Test || ifTrue != c.IfTrue || ifFalse != c.IfFalse)
      return Expression.Condition(test, ifTrue, ifFalse);

   return c;
 }

protected virtual Expression VisitParameter(ParameterExpression p)
{
    return p;
}

protected virtual Expression VisitMemberAccess(MemberExpression m)
 {
    Expression exp = this.Visit(m.Expression);
    if (exp != m.Expression)
        return Expression.MakeMemberAccess(exp, m.Member);

    return m;
 }

protected virtual Expression VisitMethodCall(MethodCallExpression m)
 {
    Expression obj = this.Visit(m.Object);
    IEnumerable<Expression> args = this.VisitExpressionList(m.Arguments);
    if (obj != m.Object || args != m.Arguments)
       return Expression.Call(obj, m.Method, args);

    return m;
  }

 protected virtual ReadOnlyCollection<Expression> VisitExpressionList(
                        ReadOnlyCollection<Expression> original)
  {
     List<Expression> list = null;

     for (int i = 0, n = original.Count; i < n; i++)
     {
         Expression p = this.Visit(original[i]);
         if (list != null)
         {
           list.Add(p);
          }
         else if (p != original[i])
         {
            list = new List<Expression>(n);
            for (int j = 0; j < i; j++)
            {
               list.Add(original[j]);
            }
            list.Add(p);
          }
       }
```

```
        if (list != null)
            return list.AsReadOnly();

        return original;
        }

protected virtual MemberAssignment VisitMemberAssignment(MemberAssignment assignment)
  {
      Expression e = this.Visit(assignment.Expression);

      if (e != assignment.Expression)
                return Expression.Bind(assignment.Member, e);

      return assignment;
}

protected virtual MemberMemberBinding VisitMemberMemberBinding(MemberMemberBinding bindin
{
    IEnumerable<MemberBinding> bindings = this.VisitBindingList(binding.Bindings);

    if (bindings != binding.Bindings)
      return Expression.MemberBind(binding.Member, bindings);

    return binding;
}

protected virtual MemberListBinding VisitMemberListBinding(MemberListBinding binding)
{
    IEnumerable<ElementInit> initializers =
                              this.VisitElementInitializerList(binding.Initializers);

    if (initializers != binding.Initializers)
    {
        return Expression.ListBind(binding.Member, initializers);
    }
    return binding;
}

protected virtual IEnumerable<MemberBinding> VisitBindingList(
                                    ReadOnlyCollection<MemberBinding> original)
{
    List<MemberBinding> list = null;
    for (int i = 0, n = original.Count; i < n; i++)
    {
       MemberBinding b = this.VisitBinding(original[i]);
       if (list != null)
       {
          list.Add(b);
       }
       else if (b != original[i])
       {
          list = new List<MemberBinding>(n);
              for (int j = 0; j < i; j++)
              {
                 list.Add(original[j]);
              }
              list.Add(b);
        }
      }
      if (list != null)
           return list;
```

```
        return original;
    }

    protected virtual IEnumerable<ElementInit> VisitElementInitializerList(
                                    ReadOnlyCollection<ElementInit> original)
    {
        List<ElementInit> list = null;
        for (int i = 0, n = original.Count; i < n; i++)
        {
            ElementInit init = this.VisitElementInitializer(original[i]);

            if (list != null)
            {
               list.Add(init);
            }
            else if (init != original[i])
            {
               list = new List<ElementInit>(n);
               for (int j = 0; j < i; j++)
               {
                   list.Add(original[j]);
               }
               list.Add(init);
             }
          }
          if (list != null)
             return list;

          return original;
    }

    protected virtual Expression VisitLambda(LambdaExpression lambda)
    {
       Expression body = this.Visit(lambda.Body);
       if (body != lambda.Body)
          return Expression.Lambda(lambda.Type, body, lambda.Parameters);

       return lambda;
    }

   protected virtual NewExpression VisitNew(NewExpression nex)
   {
       IEnumerable<Expression> args = this.VisitExpressionList(nex.Arguments);

       if (args != nex.Arguments)
       {
           if (nex.Members != null)
                return Expression.New(nex.Constructor, args, nex.Members);
           else
                return Expression.New(nex.Constructor, args);
       }
       return nex;
    }

    protected virtual Expression VisitMemberInit(MemberInitExpression init)
    {
       NewExpression n = this.VisitNew(init.NewExpression);
       IEnumerable<MemberBinding> bindings = this.VisitBindingList(init.Bindings);

       if (n != init.NewExpression || bindings != init.Bindings)
           return Expression.MemberInit(n, bindings);
```

```
      return init;
    }

  protected virtual Expression VisitListInit(ListInitExpression init)
  {
      NewExpression n = this.VisitNew(init.NewExpression);
      IEnumerable<ElementInit> initializers =
                            this.VisitElementInitializerList(init.Initializers);

      if (n != init.NewExpression || initializers != init.Initializers)
          return Expression.ListInit(n, initializers);

      return init;
    }

  protected virtual Expression VisitNewArray(NewArrayExpression na)
  {
       IEnumerable<Expression> exprs = this.VisitExpressionList(na.Expressions);
       if (exprs != na.Expressions)
       {
          if (na.NodeType == ExpressionType.NewArrayInit)
          {
              return Expression.NewArrayInit(na.Type.GetElementType(), exprs);
          }
          else
          {
              return Expression.NewArrayBounds(na.Type.GetElementType(), exprs);
          }
        }
        return na;
  }

  protected virtual Expression VisitInvocation(InvocationExpression iv)
  {
     IEnumerable<Expression> args = this.VisitExpressionList(iv.Arguments);
     Expression expr = this.Visit(iv.Expression);

     if (args != iv.Arguments || expr != iv.Expression)
         return Expression.Invoke(expr, args);

     return iv;
    }
}

class Linq2Jpql : ExpressionVisitor
{
   StringBuilder sb;
   String sel, frm, whr, ordBy, grpBy, param, idVar;
   int sb_len, moreThanOneParam;

   internal Linq2Jpql()
   {
   }

   internal string Translate(Expression expression)
   {
       this.sb = new StringBuilder();
       this.Visit(expression);
       if (sel == null)
             sel = "Select * ";
       return (sel + frm + whr + ordBy + grpBy);
    }
```

```
private void ClearStringBuilder(StringBuilder strBldr)
{
    sb_len = strBldr.Length;
    strBldr.Remove(0, sb_len);
}

private static Expression StripQuotes(Expression e)
{
    while (e.NodeType == ExpressionType.Quote)
        e = ((UnaryExpression)e).Operand;

    return e;
}

protected override Expression VisitMethodCall(MethodCallExpression m)
{
   if (m.Method.Name == "Select")
   {
      moreThanOneParam = 1;
      sb.Append("SELECT ");
      LambdaExpression lambda = (LambdaExpression)StripQuotes(m.Arguments[1]);
      this.Visit(lambda.Body);

      sel = sb.ToString();
      ClearStringBuilder(sb);
      this.Visit(m.Arguments[0]);
      return m;
    }
    if (m.Method.Name == "Where")
    {
       sb.Append(" WHERE ");
       LambdaExpression lambda = (LambdaExpression)StripQuotes(m.Arguments[1]);
       this.Visit(lambda.Body);

       whr = sb.ToString();
       ClearStringBuilder(sb);
       this.Visit(m.Arguments[0]);
       return m;
     }
     if (m.Method.Name == "OrderBy")
     {
        sb.Append(" ORDER BY ");
        LambdaExpression lambda = (LambdaExpression)StripQuotes(m.Arguments[1]);
        this.Visit(lambda.Body);

        ordBy = sb.ToString();
        ClearStringBuilder(sb);
        this.Visit(m.Arguments[0]);
        return m;
      }
      if (m.Method.Name == "GroupBy")
      {
         LambdaExpression lambda = (LambdaExpression)StripQuotes(m.Arguments[1]);
         this.Visit(lambda.Body);
         param = sb.ToString();

         grpBy = " GROUP BY " + param;
         sel = "SELECT " + param;
         ClearStringBuilder(sb);
         this.Visit(m.Arguments[0]);
         return m;
```

```
            }

            throw new NotSupportedException(string.Format(
                                "The method '{0}' is not supported", m.Method.Name));

    }

    protected override Expression VisitBinary(BinaryExpression b)
    {
        sb.Append("(");
        this.Visit(b.Left);
        switch (b.NodeType)
        {
          case ExpressionType.And:
          case ExpressionType.AndAlso:
                sb.Append(" AND ");
                break;

          case ExpressionType.Or:
          case ExpressionType.OrElse:
                sb.Append(" OR ");
                break;

          case ExpressionType.Equal:
                sb.Append(" = ");
                break;

          case ExpressionType.NotEqual:
                sb.Append(" <> ");
                break;

          case ExpressionType.LessThan:
                sb.Append(" < ");
                break;

          case ExpressionType.LessThanOrEqual:
                sb.Append(" <= ");
                break;

          case ExpressionType.GreaterThan:
                sb.Append(" > ");
                break;

          case ExpressionType.GreaterThanOrEqual:
                sb.Append(" >= ");
                break;

          case ExpressionType.Add:
                sb.Append(" + ");
                break;

          case ExpressionType.Subtract:
                sb.Append(" - ");
                break;

          case ExpressionType.Multiply:
                sb.Append(" * ");
                break;

          case ExpressionType.Divide:
                sb.Append(" % ");
                break;
```

```
      default:
           throw new NotSupportedException(string.Format(
                    "The binary operator '{0}' is not supported", b.NodeType));
    }
    this.Visit(b.Right);
    sb.Append(")");
    return b;
}

protected override Expression VisitConstant(ConstantExpression c)
{
     IQueryable q = c.Value as IQueryable;
     if (q != null)
     {
        sb.Append(" FROM ");
        sb.Append(q.ElementType.Name);
        sb.Append(" AS ");
        sb.Append(idVar);

        frm = sb.ToString();
        ClearStringBuilder(sb);
      }
      else if (c.Value == null)
      {
        sb.Append("NULL");
      }
      else
      {
        switch (Type.GetTypeCode(c.Value.GetType()))
        {
           case TypeCode.Boolean:
                sb.Append(((bool)c.Value) ? 1 : 0);
                break;

           case TypeCode.String:
                sb.Append("\"");
                sb.Append(c.Value);
                sb.Append("\"");
                break;

           case TypeCode.Object:
                throw new NotSupportedException(string.Format(
                          "The constant for '{0}' is not supported", c.Value));

           default:
                sb.Append(c.Value);
                break;
           }
        }
        return c;
}

protected override Expression VisitMemberAccess(MemberExpression m)
{
    if (m.Expression != null && m.Expression.NodeType == ExpressionType.Parameter)
    {
        idVar = m.Expression.ToString();

        if (moreThanOneParam > 1 && sel == null)
           sb.Append(", ");
```

```
            moreThanOneParam++;
            sb.Append(idVar + ".");
            sb.Append(m.Member.Name);
            return m;
        }
        throw new NotSupportedException(string.Format("The member '{0}' is not supported",
                                    m.Member.Name));
    }
}

public class employee
{
    public int id;
    public String name;
    public float salary;
}

class App
{
    static void Main()
    {
          employee[] empList = new employee[]  {
       new employee {id=1, name="john", salary=1200},
       new employee {id=2, name="tom", salary=1500},
       new employee {id=3, name="harry", salary=1200}
    };

    var query = empList.AsQueryable().Where(e => e.salary == 1200);

    //var query = from e in empList.AsQueryable()
    //            where e.id > 2
    //            orderby e.name
    //            //group e by e.salary;
    //            select new { e.name, e.salary };

    String queryString = new Linq2Jpql().Translate(query.Expression);
    Console.WriteLine(" LINQ query is : {0}", query.Expression.ToString());
    Console.WriteLine(" ");
    Console.WriteLine(" Translated JPQL query is : {0}", queryString);
    }
 }
}
```

Bibliography

[Dam06] Christian W. Damus. Implementing Model Integrity in EMF with MDT OCL, Eclipse Technical Article, 2006. http://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html.

[DG08a] Chris Daly and Miguel Garcia. Emfatic Language for EMF, 2008. http://www.alphaworks.ibm.com/tech/emfatic.

[DG08b] Chris Daly and Miguel Garcia. Gymnast, 2008. http://wiki.eclipse.org/Gymnast.

[Dye08] Wes Dyer. Yet Another Language Geek, 2008. http://blogs.msdn.com/wesdyer/archive/2006/12/22/thus-quoth-the-humble-programmer.aspx.

[EK07] Andrew D. Eisenberg and Gregor Kiczales. Expressive programs through presentation extension. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 73–84, New York, NY, USA, 2007. ACM.

[Fow08] Martin Fowler. Domain Specific Languages, 2008. http://martinfowler.com/dslwip/index.html.

[Gar06] Miguel Garcia. Formalizing the well-formedness rules of EJB3QL in UML + OCL. In T. Kühne, editor, *Reports and Revised Selected Papers, Workshops and Symposia at MoDELS 2006, Genoa, Italy*, LNCS 4364, pages 66–75. Springer-Verlag, 2006. http://www.sts.tu-harburg.de/~mi.garcia/pubs/atem06/JPQLMM.pdf.

[Gar08] Miguel Garcia. Automating the embedding of Domain Specific Languages in Eclipse JDT, Eclipse Technical Article, 2008. http://www.eclipse.org/articles/Article-AutomatingDSLEmbeddings/index.html.

[GP08] Miguel Garcia and Rakesh Prithiviraj. Rethinking the Architecture of O/R Mapping for EMF in terms of LINQ. In *Eclipse Modeling Symposium at Eclipse Summit Europe 2008, Stuttgart, Germany*, 2008. http://www.sts.tu-harburg.de/~mi.garcia/pubs/2008/ese/linq4emf.pdf.

[Gro06] Object Management Group. OMG OCL Specification v2.0, 2006. http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf.

[Gro08] Java Persistence 2.0 Expert Group. JSR 317: Java Persistence API, version 2.0, 2008. http://jcp.org/en/jsr/detail?id=317.

[Hun08] Michael Hunger. Java Embedded QUEry Language, 2008. http://jequel.de/.

[KR08] Jevgeni Kabanov and Rein Raudjärv. Embedded typesafe domain specific languages for java. In *PPPJ '08: Proceedings of the 6th international symposium on Principles and practice of programming in Java*, pages 189–197, New York, NY, USA, 2008. ACM.

[KT06] Thomas Kuhn and Olivier Thomann. Abstract Syntax Tree, Eclipse Technical Article, 2006. http://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html.

[LIN07] Language Integrated Query (LINQ), 2007. http://msdn.microsoft.com/en-us/library/bb397926.aspx.

[Mar08] Luke Marshall. Deriving Miss Daisy, 2008. http://mathgeekcoder.blogspot.com/2008/01/deriving-miss-daisy.html.

[NM08] Anders Norås and Thomas Mueller. Quaere, 2008. http://quaere.codehaus.org/.

[Pia08] Paolo Pialorsi. *Programming Microsoft®LINQ*. Microsoft Press, Redmond, 2008.

[PR07] Paolo Pialorsi and Marco Russo. *Introducing Microsoft®LINQ*. Microsoft Press, Redmond, WA, USA, 2007.

[Ric08] Jeffrey Ricker. Generics in Eclipse Modelling Framework, 2008. http://eclipse.dzone.com/articles/generics-eclipse-modelling-fra.

[Sil08] Minnesota Extensible Language Tools, 2008. http://melt.cs.umn.edu/index.php.

[Tea08] Dali Team. Dali plugin for configuring O/R Mapping in JPA, 2008. http://www.eclipse.org/webtools/dali/main.php.

[War08] Matt Warren. The Wayward Weblog, 2008. http://blogs.msdn.com/mattwar/archive/2007/07/30/linq-building-an-iqueryable-provider-part-i.aspx.

[Yao06] Liu Yao. Support for DSL in Eclipse. Master's thesis, Software Systems Institute (STS), Technische Universität Hamburg-Harburg, Germany, Sep 2006. http://www.sts.tu-harburg.de/pw-and-m-theses/2006/liuy06.pdf.