System Query Language (SyQL)
An Abstraction Layer for Querying Software Models

By: Dott. Ing. Mirco Bianco

Faculty Supervisor:
Prof. Werner Nutt

External Supervisor:
Prof. Ernesto Damiani

A thesis submitted for the degree of Doctor of Philosophy

Faculty of Computer Science
Free University of Bozen-Bolzano
April 2012

Dedico questo lavoro a chi per qualsiasi motivo, lo consulterà.

E mi auguro che, in queste pagine, riesca a trovare qualcosa di utile.

Table of Contents

1 Introduction

Improving the efficiency of the software development process is a goal of Software Engineering. In industry, delivering faultless software products within budget and on time is a key success factor. Controlling the software development process may result very difficult due to the presence of humans involved in the process, who are very often under-pressure to meet deadlines. It is well-know that humans (developers) under-stress make a lot of mistakes [56]. Therefore, the source code produced can contain more defects and the quality of the final product is affected negatively. Today, the software quality improvement is an important aspect, but other aspects, as software process auditing and certification, are also important. In this dissertation, we focus mainly on the software quality improvement.

The contribution of this dissertation is twofold: first, we provide an innovative way of querying a relational database; secondly, we provide an implementation of this model by means of a SQL-like language called System Query Language (SyQL). SyQL is an object oriented data manipulation language (DML), which can be used to give a better view of the software development process to the different *stakeholders* (hereinafter called *users*) like software developers and project managers.

Morasca *et al.* [93][25] show that is possible to use different kinds of metrics as quality indicators for software projects. Therefore, the possibility to compute complex metrics through declarative queries can enhance the implementation of such approach. DeMarco [42] stated in 1982: “you cannot control what you cannot measure”; this is true for all the branches of engineering. The objective of this thesis is to provide a method for abstracting data present into a database by using a specialized schema based on methods, object types and values, and a language for querying this schema.

The novelty, introduced by SyQL, is the possibility to abstract both types and values to a higher level by converting a generic underlying schema into another one and by providing evaluation facilities for measures like linguistic variables. The absolute novelty is represented by the combination of these two aspects with the real mapping between the underlying database schema and a preliminary schema (hereinafter called meta-model). This represents an innovation in the software engineering field, because the real mapping makes the query runnable against the preliminary meta-model,

contrariwise Bossi *et al.* [21] proposed a meta-model without the mapping, making the queries' execution impossible. The preliminary meta-model, implemented by using OO facilities, allows the user to access and to evaluate the data collected by PROM in an easier way than before.

In the software engineering domain, SyQL may help the engineers during reverse engineering tasks; it can improve the efficiency of the code inspections by evaluating the fault-proneness of classes and methods. The managers can prevent deadlocks by checking software quality indicators and effort data.

In SyQL, an user can run analyses by using linguistic variables [126]. Therefore, users, who have no idea of the order of magnitude of the measures needed to take a decision, can perform queries more easily and without using thresholds. Indeed, thresholds can lead to take a wrong decision, because a wrong choice of the cut-off values may hide important information [126]. For instance, the interpretation of metrics like the effort spent on a software artifact requires to take into the account several other metrics. For the effort, we have identified three different measures that are able to provide a context to support the evaluation of the effort measure: start date time, end date time, and number of persons involved in the activity. This context can also be enriched by other metrics like the developers experience, the difficulty level of tasks, etc. We have selected these three measures, because it is possible to collect them in a non-intrusive way. The non-intrusive measuring is important to avoid bias. The term “non-intrusive measuring” is very often assimilated to the one “automatic measuring”, but it is not completely correct. When, non-intrusive measuring is performed, it means that the software development process remains as it is. If there are already metrics collected on the process manually, we integrate these with the automatically collected ones.

The automatic metrics collection is implemented by tools like PROM [112] and Hackystat [73]. These tools can collect different types of measures (software metrics, effort, productivity, etc.).

In this field, an open issue is to provide a useful representation of all the collected measures. Since it is not feasible to have a unique solution, we tried to address this problem by creating a language. After that, the language has been validated through a couple of controlled experiments, in which a fair number of users tried to solve a set

of problems by using both SyQL and SQL/LINQ.

1.1 Problem Statement

We want to enable the different stakeholders involved in the software development process to benefit from the automatically collected data. Different users can access to the data, which are stored into a relational database, by using a very concise query language (SyQL).

By using SyQL, it is possible to perform queries on the data collected from the PROM system [112] by hiding the schema of the relational data warehouse through a specifically designed meta-model. We want to stress this point, because the direct access to this kind of relational data warehouse has several drawbacks:

- most of the collected data are explicitly dependent from time, hence storing temporal data implies that the size of the relational tables is continuously increasing. To query this kind of data structures, the user must know the indexing structures of the data warehouse very well. Otherwise, querying those tables require easily long time, because it is very easy to join two or more huge tables together without using indexes in the proper way. In addition, due to space reasons, a data warehouse cannot contain indexes for all the fields of big tables. Finally, it is not possible to optimize the structure of a data warehouse for all kinds of queries, because the questions to tackle are a priori unknown;
- some metrics, like the effort, are stored as events. Thus, these metrics require to be aggregated in a specified time interval before being evaluated by the user. This process of selection and aggregation requires to know the internal structure of the data warehouse and how the data are stored into it. In this situation, it is easy to incur in one or more mistakes due to the lack of knowledge about the data warehouse.

Summarizing, the direct access to the data warehouse through a query language, like SQL, requires high skills for different reasons:

- the queries become usually very long, thus they are complex to write and to modify;

- the user should be able to evaluate a query execution plan, otherwise the query may not terminate in a reasonable amount of time;
- the user must know how the data warehouse tables are filled by the PROM components. Otherwise, the selection and aggregation of the process data may be hard to do.

Moreover, when a user wants to take benefits by using these data, she/he makes implicitly an assumption: to be able to interpret a set of complex measures like the software process metrics. This is not trivial because most of these metrics are strictly interrelated. Therefore, if the user does not know perfectly the definitions of the measures, the analysis process may be fruitless. SyQL helps the users by giving the possibility to interpret data through fuzzy logic.

The research questions that try to address these problems are the following:

RQ1: How could SyQL help the user to write less error-prone queries?

RQ2: How could SyQL increase the efficiency of the software development process by helping the engineers to identify the defects?

The first research question is focused on the explanation of the effects due to the adoption of this query language, whereas the second one is focused on the side effects due to the adoption of this language in the software engineering domain.

1.2 Goals

The main goal of this dissertation is to propose a generic system that helps the users to gain information from a DBMS, and in particular from a DBMS containing data collected by an Automated In-process Software Engineering Measurement and Analysis Systems (AISEMA). The secondary goal is to explore the possibility offered from these automatically collected data for improving the efficiency of the development process by reducing the number of software faults.

The main goal is oriented to cross the barriers of adoption of an AISEMA system by making the collected information available to all the stakeholders. It is known that one of the steepest barriers to the adoption of an AISEMA system are the software developers: they “are not against AISEMA systems, but rather against being measured as passive entities, having the data used against them or having no benefit from the

measurement.” [32] Therefore, if the developers can browse the collected data in an easier way than before, we think that this barrier can be significantly reduced.
In this dissertation, we try to meet the first goal by proposing an implementation of a query language, next we validate it in a couple of controlled experiments.
During the development of this tool, we have given the priority to the developer needs, it means that this tool (at this stage) can answer to the developers' questions better than the other stakeholders' ones. The motivation is twofold:

- as said before, they are the steepest barrier to the adoption of these measuring systems;
- they develop the product, hence they may produce more interesting information useful for improving the process.

In summary, we want to create an abstraction layer (the language) between the users, and the data stored into the data warehouse. In addition, we want to enable users to take benefits from more complex tools like models created by using machine learning techniques. These models are trained to evaluate the fault-proneness of software artifacts.

1.3 Methodology

As main research methodology, this dissertation uses the action research approach [115][119]. This methodology allows us to improve our problem solution while simultaneously gathering on-field experience [41].
Action research contemplates an iterative process divided in: diagnosing problems, planning actions, performing the actions, evaluating the results and specifying the lessons learnt. At the end of each iteration, we had enough information to improve the SyQL language.
To ensure the correct development of the tool, we have adopted the unit tests as testing method. During the development of a new language, the regression testing is very important, because it is very easy to introduce defects during the fixing/enhancing process.
At the beginning of each iteration, when we start to implement new features, all the tests are run, in order to ensure the fulfilling of the requirements.
During the development of a query engine for SyQL, we have used two different

prototypes of the same query engine at two different stages of development. These prototypes have been tested during two case-studies, in which software process data were analyzed. Each case-study has been the check points of our work, i.e. the end of an iteration (evaluation of the results), from which we restart a new iteration. In this way, we have discovered qualitatively the lacks of our language. Moreover, the case studies produce additional knowledge useful to improve the efficiency of the development process, which is presented as a compendium of this Thesis.
At the end of these two iterations, we performed a couple of controlled experiments, in which the SyQL and SQL/LINQ [89] users' performances were measured by making possible a comparison between them.

1.4 Publications

1. Mirco Bianco. SyQL: Querying Software Process Data Through an Object-Oriented Metamodel. *ODBMS.ORG*, 2011.
2. Mirco Bianco. SyQL: Querying Software Process Data Through an Object-Oriented Metamodel. In *Proceeding of the International Symposium on Data-Driven Process Discovery and Analysis (SIMPDA 2011)*. Campione d'Italia, Italy, 2011.
3. Mirco Bianco and Ernesto Damiani. SyQL: A Controlled Experiment Concerning the Evaluation of its Benefits. In *Proceedings of the Annual Conference on Software Engineering (SE 2010)*. Phuket, Thailand, 2010. (**Best Research Paper Award**)
4. Mirco Bianco, Alberto Sillitti, and Giancarlo Succi. Fault-Proneness Estimation and Java Migration: A Preliminary Case Study. In *Proceedings of the First International Conference on Software, Services & Semantic Technologies (S3T'2009)*. Sofia, Bulgaria, pg.124-131, 2009.
5. Mirco Bianco. SyQL: A Tool for Analyzing the Software Development Process. In *Proceedings of GIIS 2009 Ph.D. Symposium (GIIS'2009)*. Salerno, Italy, pg. 137-152, 2009.
6. Mirco Bianco, Alberto Sillitti, and Giancarlo Succi. Analyzing the Software Development Process with SyQL and Lagrein. In *Proceedings of the 21st International Conference on Software Engineering Knowledge Engineering*

(SEKE'2009), Boston, Massachusetts, USA, pag. 682-687, 2009

7. Mirco Bianco, Alberto Sillitti, and Giancarlo Succi. Lagrein: Software Metrics Visualization and Process Mining Tool. EclipseCon 2009. Santa Clara, California, USA, 2009 *(Poster session)*.
8. Mirco Bianco, Alberto Sillitti, and Giancarlo Succi. SyQL: an object oriented, fuzzy, temporal query language for repositories of software artifacts. In *Companion To the 23rd ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*. Nashville, Tennessee, USA, pg. 715-716, 2008.
9. Mirco Bianco, Marco Scotto, and Giancarlo Succi. Extracting and analyzing software code metrics from C# source code. The Eighteenth ASTReNet Workshop. London, 2007 *(4 pages paper)*.

1.5 Thesis Organization

This thesis makes several contributions towards realizing domain specific language [80][90] that helps the software development process stakeholders to benefit of the data that are automatically collected from the PROM system. These contributions can be broadly categorized into *software process improvement*, *automatic metrics collection systems*, and *implementation*. Figure 1 shows the organization of the remainder of the thesis, according to the categorization.

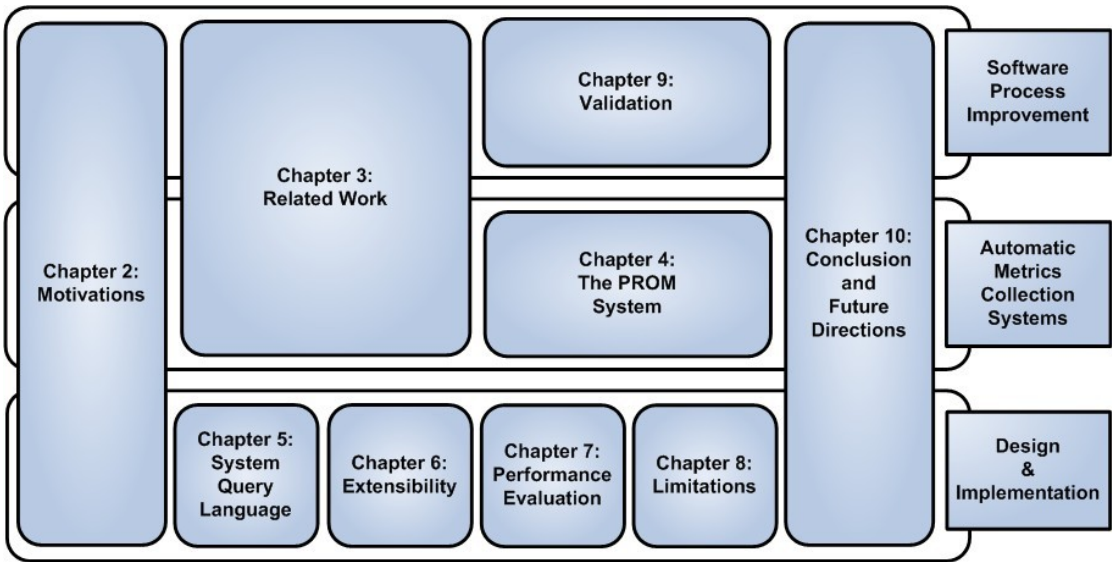

Figure 1: Thesis contributions and organization.

In the following, we detail the chapter-wise thesis contributions:

- **Chapter 2 : Motivations.** This chapter contributes by presenting five distinct motivations that justify the effort spent for developing SyQL. Each motivation is completed with an example that makes evident the limits of the actual approach based on SQL. We show how some Object Oriented facilities like inheritance and late binding may help the user to solve several information retrieval problems. The motivations of using fuzzy linguistic are shown with a practical example related to the software metrics domain. The possibility to write very concise queries is stressed by showing a real problem. Finally, the possibility to compute new properties on an existing database schema is motivated by an example, where data representing software structure are queried using an attribute computed on the fly.
- **Chapter 3 : Related work.** This chapter provides a review of the literature in which SyQL is directly or indirectly involved. The Chapter starts by presenting the Software Measurement Theory, which is the theoretical base of the Automated In-process Software Engineering Measurement and Analysis (AISEMA) systems. Then, the most relevant literature on the Domain Specific Languages is presented by showing the approaches used for implementing SyQL. After that, the areas of use of SyQL are introduced by presenting the major contributions in the Software Fault-Proneness Prediction area. Finally, a comparison with similar tools is provided by evidencing the benefits of our querying approach.
- **Chapter 4 : The PROM System.** This chapter provides a high-level description of the PROM AISEMA system starting from the overall architecture. After that, the source code metrics extractors are presented in detail by showing the technical issues encountered by the engineers during the development of these software probes for Java and C#. Finally, we presented the location of SyQL inside the PROM architecture by showing how it contributes to close the control loop over the software development process.
- **Chapter 5 : System Query Language – SyQL.** This chapter explains in detail how the SyQL query engine works. The structure, the data model, and the semantic of the language are presented and discussed by showing the benefits of the design. During the presentation of the internal architecture of the tool, the technical choices are motivated. As a proof of concept, the current implementation

of the abstraction layer (SyQL library) for PROM collected metrics is presented. After that, a description about the query execution work-flow is provided by showing how the double mechanism of type and value abstraction processes the data. Finally, the implemented optimization is presented and discussed.

- **Chapter 6 : Extensibility.** This chapter explains a methodology for implementing new concepts and new fuzzy evaluators, which provide respectively type abstraction and value abstraction to the SyQL user. By reading this Chapter, a user can extend the SyQL engine by himself/herself. This methodology is presented by an example, which shows operationally how to write the extension code.
- **Chapter 7 : Performance Evaluation.** This chapter presents the results of the performance evaluation conducted on the SyQL query engine running on a parallel machine. The current bottleneck is identified, and an alternative solution is presented. Finally, a roughly evaluation of the Capers Jones' Index [74] is provided to the reader by showing quantitatively the expressiveness of the SyQL language.
- **Chapter 8 : Limitations.** This chapter shows a couple of currently known limits of the SyQL query engine. The first limitation depends on the domain specificity of SyQL, which is a data manipulation language. The second limitation is caused by the lack of closure, which does not allow to reuse the query results. Finally, a set of solutions to overcome these limitations is proposed.
- **Chapter 9 : Validation.** This chapter presents a couple of controlled experiments, and two case studies in which SyQL was used. During the controlled experiments, we measured the gain given by SyQL in terms of user productivity by working on two real databases. While, in the two case studies, the language was used to support researchers to build the data sets necessary for training fault-proneness prediction models. Finally, as a compendium of this Thesis, the models' performances are presented, and it is shown a possible way to reuse the prediction models by embedding them into the SyQL libraries.
- **Chapter 10 : Conclusion and Future Directions.** The concluding chapter of the thesis summarizes the contributions and discusses future research directions for SyQL. Finally, it provides the answer to the research question raised in Section 1 .

2 Motivations

In this chapter, we discuss the motivations that led to implement SyQL. A set of examples may help the reader to understand better the issues that SyQL addresses. In these examples, we are going to compare, if possible, queries written in SyQL to the corresponding SQL ones. These comparisons show the different solutions devised to solve the same problem, making evident the benefits provided by the meta-model.

Controlling the software development process can be of paramount importance. Software production stops may be prevented by checking software quality indicators and effort data. SyQL enables a user to write queries that contain temporal and linguistic variables like TODAY / YESTERDAY / TOMORROW and HIGH / MEDIUM / LOW, in a very concise way. It benefits from Object Oriented facilities, like inheritance and late-binding. SQL does not provide these features completely. As the software development process evolves, the temporal aspect is important [59]. Linguistic variables [126] are very important, because they provide the necessary level of abstraction between users and values of metrics, necessary to enable the user to interpret the values correctly. This mechanism becomes useful when the software metrics are involved in filtering conditions, because the evaluation of a software metric requires experience that the user usually does not have.

In the following, we present five distinct motivations that put in evidence the relevance of our work. Language conception, object orientation, linguistic variables, conciseness, and uncertainty are introduced by a set of examples.

2.1 First Motivation – The language aspect

The large quantity of metrics collected by our automatic metrics collection system [109] is stored into a relational data warehouse (see details in chapter 4). This data warehouse contains software metrics, software structure (e.g. class hierarchy, method calls, namespace declaration, etc.) and effort data. For a person skilled in the art, it is easy to understand that mining these data requires proper tools like a data manipulation language (DML) designed with this scope in mind. As standard DML , almost all the relational data warehouse engines adopt an extended version of SQL that comprehends “roll-up” and “drill-down” clauses, which allow the users to

aggregate and disaggregate data along different dimensions. By using this extended SQL, it is possible to reduce the length of the queries (see Example 1). In addition to that, the adoption of these clauses makes possible to increase the level of optimization of the query execution by advantageously reducing the query execution time.

In Example 1, we show how the resulting queries are affected by the structure of the used language (in the specific case by the availability of the "roll-up" and "drill-down" clauses) to support the work disclosed in this dissertation.

Example 1:

In this example, we aggregate the columns of the relation *method_loc* presented below (Figure 2). This table contains the fully qualified signature of a method, the number of line of code (LOC), and the timestamp. We use as dimensions for the cube the following column: time, namespace, class, and method name concatenated with the method parameters if any. In the first query, we used the ROLLUP clause, whereas in the second one we avoid using it. Both queries are equivalent, and they produce the same result. However, from the point of view of the time performances the first one is better, because it accesses to the table *method_loc* once, whilst the second one accesses to it five times in total.

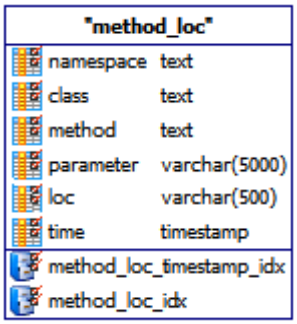

Figure 2: Schema of relation *method_loc*.

With Rollup:

```
[1]  SELECT  time, namespace, class,
[2]          method || '(' || parameter || ')', SUM(loc) AS lineOfCode
[3]  FROM method_loc
[4]  GROUP BY ROLLUP(time, namespace, class,
[5]                  method || '(' || parameter || ')');
```

Listing 1: SQL query with Rollup clause.

Without Rollup:

```
[1]  SELECT  time, namespace, class,
[2]  method || '(' || parameter || ')', SUM(loc) AS lineOfCode
[3]  FROM method_loc
[4]  GROUP BY time, namespace, class, method || '(' || parameter || ')'
[5]  UNION ALL
[6]  SELECT time, namespace, class, NULL, SUM(loc)
[7]  FROM method_loc
[8]  GROUP BY time, namespace, class
[9]  UNION ALL
[10] SELECT time, namespace, NULL, NULL, SUM(loc)
[11] FROM method_loc
[12] GROUP BY time, namespace
[13] UNION ALL
[14] SELECT time, NULL, NULL, NULL, SUM(loc) AS
[15] FROM method_loc
[16] GROUP BY time
[17] UNION ALL
[18] SELECT NULL, NULL, NULL, NULL, SUM(loc) AS
[19] FROM method_loc;
```

Listing 2: SQL query without Rollup clause.

Another interesting aspect useful to support this work regards the reusing of the user experience. Since both the standard SQL [39] and the extended SQL have a similar structure, the learning curve of an SQL user for learning the extended SQL is advantageously smooth by saving time. However, in the reusing of the users' programming skills, both the standard SQL and the extended SQL may not be flexible enough like an Object-Oriented environment. In order to overcome this limitation, new specifications for interoperability with an Object-Oriented languages like Java (SQL/JRT – ISO/IEC 9075, Part 13) have been defined.

By using these extensions, programmers can write database procedures reusing their own Object-Oriented programming skills. String and numeric conversions become the same as in Java and proprietary JDBC libraries assure an efficient data conversion. Therefore, data manipulation libraries defined by the users can be easily imported into SQL environments, an the procedures contained into these libraries can be invoked from standard SQL statements. In this way, it is then possible to integrate Java's code

into SQL statements.

The main drawback of this approach is that the SQL and the Object-Oriented environments are on the same level, hence a user cannot benefit of the abstraction provided by a language that has a similar structure, but resides at a higher level like SyQL.

Indeed, the approach taken with SyQL is completely different; a SyQL statement invokes an SQL statement/procedure only if it is needed. When it is necessary, it retrieves other data during the other stages of the query execution. These data (needed to complete the query execution) may accrue from different data sources (web services, files, data bases, etc.), and not necessarily from the data warehouse. In this way, SyQL wants to enable the user to manipulate all these data easily. Queries are significantly shorter than the corresponding SQL translations, and SQL users require only few hours to learn SyQL.

Specifically, the syntax of SyQL is similar to the ones adopted by two other "new" query languages: LINQ [89] and .QL [92] (a comparison between SyQL and the other similar query languages is provided in Section 3.5).

For choosing the proper way to implement the SyQL query engine, we have followed the guidelines for the implementation of the Domain Specific Languages (DSL) given from Mernik et al. in 2005 [90] (see Figure 3). By looking to them, we have decided to implement the SyQL query engine as a language interpreter for the following reasons:

- we do not need to perform Domain-specific analysis, verification, optimization, parallelization, and transformation (AVOPT) of a program;
- we do not have to follow a domain specific notation strictly;
- we have a potentially large user community.

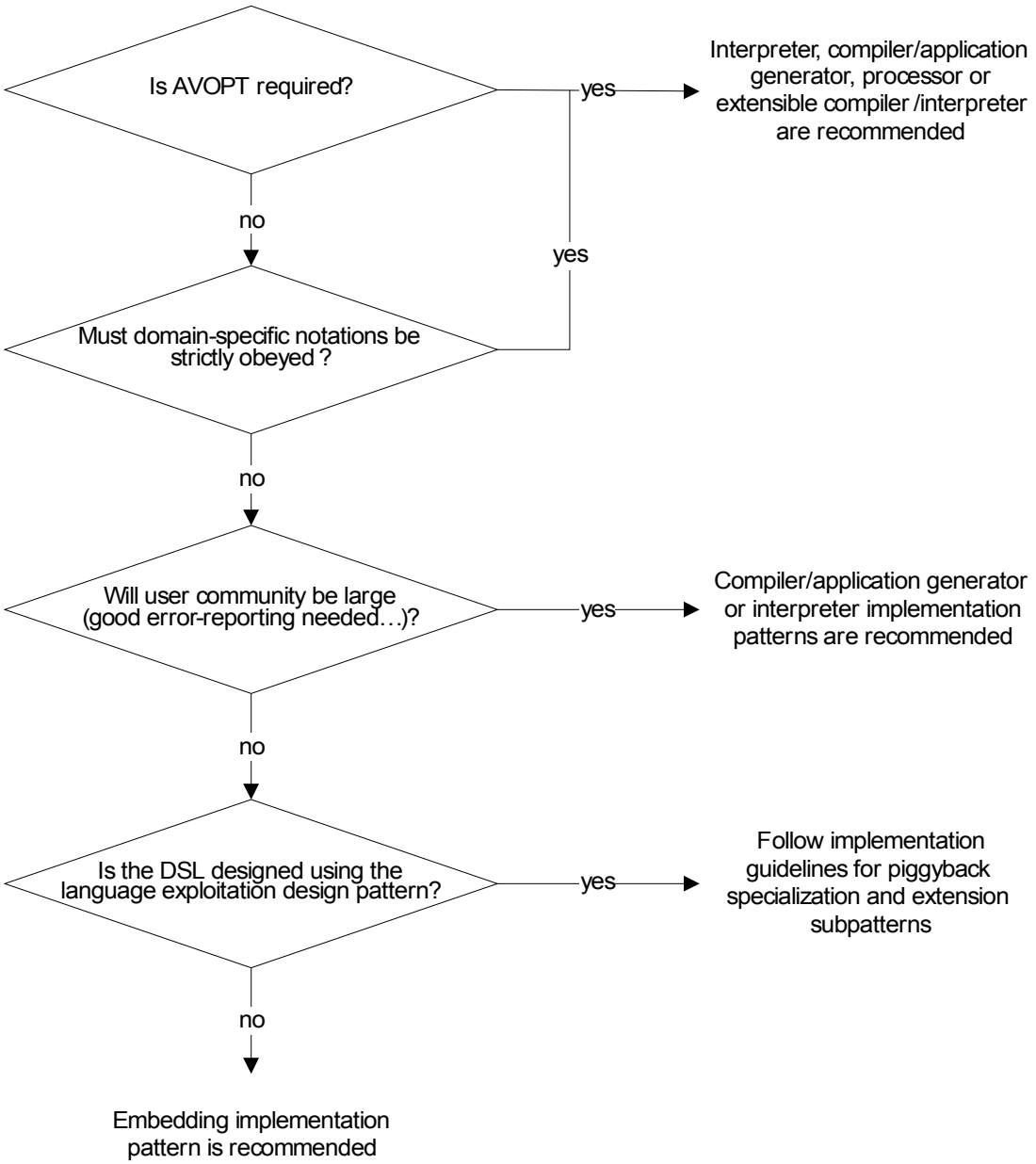

Figure 3: Implementation guidelines [90].

More details about the DSL(s) are provided in section 3.3 .

2.2 Second Motivation – Object-Oriented Facilities

Object-Oriented facilities like *class inheritance* and *method late binding* can be very useful in implementing libraries in an Object-Oriented environment.

SQL can provide inheritance between relations; there are different ways to design and implement a schema for that [2]. Different kinds of generalizations / specializations are widely discussed in literature [40]. All these approaches require to modify the

schema in some way.

SyQL object inheritance helps the user to create generalizations / specializations without modifying the data warehouse schema.

In Example 2, a possible way to implement the method late binding without modifying the underlying schema is shown. It is easy to see that generalization / specializations can be easily implemented using standard Java's classes, hence different implementations of a method (with the same signature) allow the SyQL concept developer to specify different behaviors (see section 5.4). This is possible because the SyQL query engine works over the DBMS of the data warehouse (see section 5.4).

Example 2:

In this example, we model the relationship between two SyQL concepts using the object inheritance. *TestMethod* class is a subclass of *Method* class.

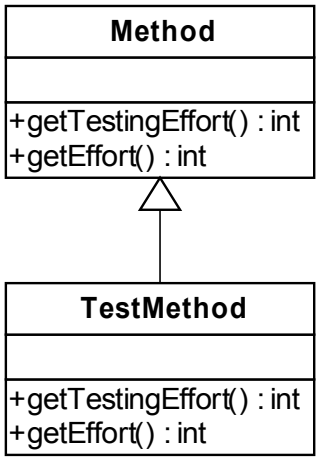

Figure 4: Inheritance diagram of concepts *Method* and *TestMethod*.

Assuming that each *TestMethod* tests exactly one *Method*. Late binding is used (in this specific case) to maintain the sum invariance property between the two set of objects. Specifically, if we sum together all the values returned by *Method.getTestingEffort(),* we get the same result of a sum between all the *TestMethod.getEffort()* returned

values. Two equivalent queries are shown below.

```
[1]   FROM Method m
[2]   WHERE 1 = 1
[3]   SELECT SUM(m.getTestingEffort());

[1]   FROM TestMethod tm
[2]   WHERE 1 = 1
[3]   SELECT SUM(tm.getEffort());
```

Listing 3: SyQL queries that returns the overall effort spent today on all the methods.

In the following, the three methods implemented.

```
[1]   package it.unibz.syql.concepts;
[2]   ...
[3]   public class Method extends AbstractConcept {
[4]     ...
[5]     @InternalCondition(cost = 50)
[6]     public int getEffort() {
[7]       Connection conn = null;
[8]       int returnEffort = -1;
[9]
[10]      try {
[11]        conn = ConnectionTools.getConnection(
[12]                Engine.getInstance().getDbName());
[13]        PreparedStatement ps = conn.prepareStatement(
[14]          "SELECT sum(cl.effort) AS effort" +
[15]          "FROM syql.class_effort AS cl " +
[16]          "WHERE cl.namespace = ? AND cl.class = ? ;");
[17]        ps.setString(1, this.classNamespace);
[18]        ps.setString(2, this.className);
[19]        ResultSet rs = ps.executeQuery();
[20]
[21]        if(rs.next())
[22]          returnEffort = rs.getInt("effort");
[23]        else
[24]          returnEffort = 0;
[25]
[26]        rs.close();
[27]        ps.close();
[28]      }
[29]      catch(Exception e) {
[30]        throw new RuntimeException("Problems in getEffort procedure", e);
[31]      }
```

```
[32]     finally {
[33]        try {
[34]           conn.close();
[35]        }
[36]        catch(Exception e) { }
[37]     }
[38]     return returnEffort;
[39]   }
[40]   ...
[41]   private LinkedList<Class> testClasses = new LinkedList<Class>();
[42]
[43]   public int getTestingEffort() {
[44]     int returnEffort = 0;
[45]     for(Class currTestClass : testClasses)
[46]        returnEffort += currTestClass.getEffort();
[47]     return returnEffort;
[48]   }
[49]   ...
[50] }
[51]
[52] public class TestMethod extends Method {
[53]
[54]   ...
[55]   @Override
[56]   public int getTestingEffort() {
[57]     return 0;
[58]   }
[59]   ...
[60] }
```

Listing 4: Method and TestMethod declarations (partial).

The method *TestMethod.getTestingEffort()* is correctly redefined. It returns always 0 because a test method cannot be tested by another test method.

To disambiguate between *Method* and *TestMethod*, we have adopted a set of techniques. These techniques are described in detail in Section 5.8.2 and may change depending on the project under analysis.

By executing SyQL queries in a fully Object Oriented environment (Java), the interoperability between the query engine and the other Java packages is possible, so that Weka [54] models can be easily integrated into the libraries (see Example 3 and

Section 9.2). By using this technique, we can avoid calling java methods from SQL, so that the risks of errors like wrong parameters passing are reduced.

Example 3:

In this example, we show a SyQL query that runs a Weka [54] model. The model takes software metrics as input and produces a boolean value as output. The returned boolean value is true if the method is categorized as faulty, otherwise it is false.

```
[1]   FROM Method m
[2]   WHERE m.isPotentiallyFaulty()
[3]   SELECT m;
```

Listing 5: SyQL query that returns potentially faulty method.

2.3 Third Motivation – Linguistic Variables

Providing users with useful information is very challenging, especially when the data warehouse is very large, and the users are unable to evaluate numerical measures. This fact becomes even more evident when the evaluation of a metric is related to another one (see Example 4). To face this problem, we introduce into the language the fuzzy logic equal operator (implemented with the keyword IS) and linguistic variables. Indeed, the Fuzzy logic is useful for performing qualitative analysis on large data sets, which could be more useful than quantitative analysis, because often the user cannot do an a priori estimation of the value of software metrics [128]. So there is the chance that the user can miss some important results if he/she uses a wrong threshold value. Therefore, value abstraction mechanism may help the user by encapsulating the “experience” to evaluate a particular metric into several fuzzy sets. Implementation details of this feature are presented and discussed in section 5.6 .

Example 4:

In this example, we show how SyQL simplifies the evaluation of the McCabe Cyclomatic Complexity (MCC or CC) [88]. The MCC is a very good predictor for maintenance productivity, if it is divided by the number of Source Lines of Code (SLOC or LOC) [60]. The resulting measure is a "complexity density" measure. Thus, we have to provide this facility to the user by defining the proper fuzzy evaluator. It should create the correct membership functions by using the ratio between MCC and SLOC.

Definition 1:

McCabe Cyclomatic Complexity (MCC) =

The cyclomatic number V(G) of a graph G with n vertices, e edges, and p connected components is

$$v(G)=e-n+p \qquad [88]$$

```
[1]   FROM Method m
[2]   WHERE m.getCC() IS high
[3]   SELECT m ;
```

Listing 6: SyQL query that returns method with high Cyclomatic Complexity.

Write an equivalent standard SQL query is impossible, due to the absence of the fuzzy equal operator. The code that evaluates the MCC metric is shown below.

```
[1]   package it.unibz.syql.concepts.supportypes.fuzzy;
[2]
[3]   import it.unibz.syql.concepts.Method;
[4]   import it.unibz.syql.fuzzy.IFuzzyEvaluator;
[5]
[6]   public class MethodCCFuzzyEvaluator implements IFuzzyEvaluator {
[7]
[8]     private String [] fuzzyLiterals = {"low", "medium", "high"};
```

```
[9]    private Method mthd;
[10]
[11]   public MethodCCFuzzyEvaluator(Method mthd) {
[12]    this.mthd = mthd;
[13]   }
[14]
[15]   @Override
[16]   public String[] getFuzzyLiteralList() {
[17]    return fuzzyLiterals;
[18]   }
[19]
[20]   @Override
[21]   public Double getFuzzyScore(String fuzzyLiteral, Object value)
[22]         throws IllegalArgumentException {
[23]    int loc = mthd.getLOC().getValue();
[24]    return UtilFuzzyEvaluator.getMinAvgMaxFuzzyScore(
[25]         fuzzyLiteral,
[26]         value,
[27]         (float) (loc * 0.16),
[28]         (float) (loc * 0.24),
[29]         (float) (loc * 0.20));
[30]   }
[31] }
```

Listing 7: Fuzzy Evaluator for the Cyclomatic Complexity software metric.

At lines 24 – 29, the method *getMinAvgMaxFuzzyScore(...)* is called. This method computes the value of the membership function of the linguistic variable *fuzzyLiteral* for the passed *value.* This function is parametrized by using three values (lines 27 - 29) that are computed using the LOC value. The details of the computed membership functions are presented in section 5.6 .

2.4 Fourth Motivation – Conciseness

SQL is not enough in term on conciseness for the user, since SQL queries become very long and difficult to read and reuse (see Example 5).

It is well-know that long queries are more prone to produce wrong code [83], and not all the mistakes are caught by the SQL interpreter. To address this situation, we show in the example below how SyQL can simplify the analysis by leveraging the knowledge of the engineers. This is of paramount importance for final users, since the engineers know where and how the data are stored.

Example 5:

In this example, we compare two equivalent queries. The former is written in SQL, the latter in SyQL. The queries return the classes modified today. For each modified class, the daily variation of the Source Line of Code (SLOC) is also computed.

SQL version:

```
[1]  CREATE TEMPORARY TABLE lastUploadID AS (
[2]     SELECT  MAX(id) AS ID
[3]     FROM    upload
[4]     WHERE   "user"=2 AND machine='METRICSSERVER' );
[5]
[6]  CREATE TEMPORARY TABLE newLOCperClass AS (
[7]     SELECT  entity, value,
[8]             fvm.new_file_versioning_hash_id
[9]     FROM    file_versioning_modification fvm,
[10]            entity_product_metrics epm1,
[11]            entity e1,
[12]            lastUploadID
[13]    WHERE   fvm.upload_id = lastUploadID.id
[14]       AND  fvm.new_file_versioning_hash_id = epm1.file_versioning_id
[15]       AND  epm1.property = 108
[16]       AND  epm1.entity = e1.id
[17]       AND  e1.name_method IS NULL );
[18]
[19] CREATE TEMPORARY TABLE oldLOCperClass AS (
[20]    SELECT  entity,
[21]            value,
[22]            fvm.previous_file_versioning_hash_id
[23]    FROM    file_versioning_modification AS fvm,
[24]            entity_product_metrics AS epm1,
[25]            entity AS e1,
[26]            lastUploadID
[27]    WHERE   fvm.upload_id = lastUploadID.id
[28]       AND  fvm.previous_file_versioning_hash_id=epm1.file_versioning_id
[29]       AND  epm1.property = 108
[30]       AND  epm1.entity = e1.id
[31]       AND  e1.name_method IS NULL );
[32]
[33] SELECT  classDeltaLOC.ClassName,
[34]         classDeltaLOC.AddedLOC,
[35]         ep.project_name AS ProjectName
```

```
[36] FROM (
[37]    SELECT DISTINCT
[38]       e.id,
[39]       e.name_class AS ClassName,
[40]    cast(newLOCperClass.value AS int)-cast(oldLOCperClass.value AS int)
[41]          AS AddedLOC
[42]    FROM newLOCperClass, oldLOCperClass, entity AS e
[43]    WHERE   newLOCperClass.entity = oldLOCperClass.entity
[44]       AND  e.id = newLOCperClass.entity
[45]       AND  e.name_namespace IS NULL
[46]    UNION
[47]    SELECT DISTINCT
[48]       e.id,
[49]       e.name_namespace || '.' || e.name_class AS ClassName,
[50]    cast(newLOCperClass.value AS int)-cast(oldLOCperClass.value AS int)
[51]          AS AddedLOC
[52]    FROM    newLOCperClass, oldLOCperClass, entity AS e
[53]    WHERE   newLOCperClass.entity = oldLOCperClass.entity
[54]       AND  e.id = newLOCperClass.entity
[55]       AND  e.name_namespace IS NOT NULL
[56] ) AS classDeltaLOC, entity_project ep
[57] WHERE   classDeltaLOC.addedloc <> 0
[58]    AND  ep.entity = classDeltaLOC.id;
```

Listing 8: SQL query for computing the daily variations of line of code of the modified classes.

SyQL version:

```
[1]  FROM    Class c
[2]  WHERE   c.hasBeenModified(today)
[3]  SELECT  c.getFullName(),
[4]          c.getLOC(today) - c.getLOC(yesterday);
```

Listing 9: SyQL query for computing the daily variations of line of code of the modified classes.

2.5 Fifth Motivation – Handling Uncertainty Without Affecting the Schema

Handling uncertainty is very important. SyQL can perform a priori unknown analyses, and it is designed to be flexible and extendable (see section 5.4). Several problems arise when the user wants to perform analyses using attributes that are not present in

the relational schema. By using SQL, creating a new useful attribute may result very hard also on a small and simple schema (see Example 6). Creating attributes over a pre-existing schema enables the user to perform analyses in a simpler way. Usually, the creation of a new dimension becomes necessary, since the database designer did not know the ultimate goal of the schema [78].

Example 6:

In this example, we want to perform the selection of all the classes with a value of Depth of Inheritance Tree (see Definition 2) equals to 2. Next, we want to print out the inheritance tree. Hence, the query result must have three columns: class name (class1), class name of the base class (class2), and class name of the base class of class2 (class3). See the graphical example below.

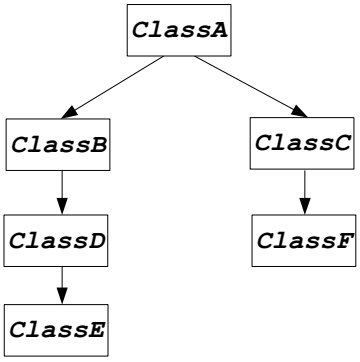

Figure 5: Inheritance diagram of example classes.

Definition 2:

Depth of Inheritance (DIT) =

depth of the class in the inheritance tree.

The depth of a node of a tree refers to the length of the maximal path from the node to the root of the tree. [31]

The returned result set should be:

Class1	Class2	Class3
ClassD	ClassB	ClassA
ClassF	ClassC	ClassA

In order to solve this problem, we propose two possible solutions: the first is written by using SQL, and the second one by using SyQL. Below, the UML diagram shows the definition of the schema.

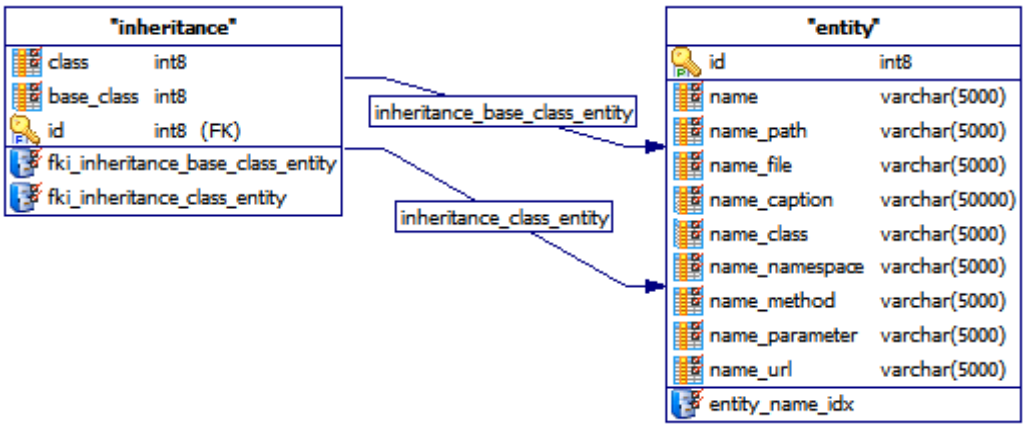

Figure 6: Schema of relations *inheritance* and *entity*.

Assuming that each class cannot have more than one base class (this is true in most of the popular programming languages used today such as Java and C#). We can see that the relation *inheritance* can capture only one tree branch per record (e.g. *{ClassF_id, ClassC_id}, {ClassC_id, ClassA_id}*, in this case we have DIT(*ClassF*) = 2, DIT(*ClassC*) = 1, and DIT(*ClassA*) = 0). Therefore, we need to compute the length of the inheritance tree of either the whole class set or a subset of it. Solving this problem by SQL is possible, but the solution may result tricky and difficult to devise especially for not experienced SQL users.

A possible solution written in SQL is presented below.

```
[1]  SELECT  e1.name_namespace || '.' || e1.name_class AS class1,
[2]    e2.name_namespace || '.' || e2.name_class AS class2,
[3]     e3.name_namespace || '.' || e3.name_class AS class3
[4]  FROM (
[5]    SELECT i1.class
[6]    FROM inheritance AS i1 INNER JOIN inheritance AS i2
[7]        ON i1.base_class = i2.class
[8]    EXCEPT
[9]    SELECT i1.class
[10]   FROM inheritance AS i1
[11]    INNER JOIN inheritance AS i2 ON i1.base_class = i2.class
[12]    INNER JOIN inheritance AS i3 ON i2.base_class = i3.class
[13] ) AS dit_depth_2
[14]   INNER JOIN inheritance AS inher_1
[15]    ON dit_depth_2.class = inher_1.class
```

```
[16]   INNER JOIN inheritance AS inher_2
[17]    ON inher_1.base_class = inher_2.class
[18]   INNER JOIN entity AS e1 ON inher_1.class = e1.id
[19]   INNER JOIN entity AS e2 ON inher_1.base_class = e2.id
[20]   INNER JOIN entity AS e3 ON inher_2.base_class = e3.id;
```

Listing 10: SQL query that returns the class inheritance tree of classes with DIT equals to two.

We do the difference between the two relations that contain the classes with $DIT \geq 2$, and $DIT \geq 3$ respectively, so that we get a relation that contains only the classes with $DIT = 2$, and we call it *dit_depth_2*. After that, we enriched the result by joining *dit_depth_2* with *inheritance* and *entity* relations two and three times respectively.

Now we solve the same problem using SyQL. The resulting query is the following:

```
[1]   FROM Class c
[2]   WHERE c.getDIT() = 2
[3]   SELECT c, c.getBaseClass(), c.getBaseClass().getBaseClass();
```

Listing 11: SyQL query that returns the class inheritance tree of classes with DIT equals to two.

The method *getDIT()* is implemented (for this example only) in Java as follows:

```
[1]   package it.unibz.syql.concepts;
[2]   ...
[3]   public class Class extends AbstractConcept {
[4]   ...
[5]    @InternalCondition(cost = 50)
[6]    public ClassDIT getDIT() {
[7]     Connection conn = null;
[8]     PreparedStatement ps = null;
[9]     int returnDIT = 0;
[10]    try {
[11]       conn = ConnectionTools.getConnection(
[12]          Engine.getInstance().getDbName());
[13]       ps = conn.prepareStatement(
[14]            "SELECT inher.base_class " +
[15]            "FROM syql.inheritance AS inher " +
[16]            "WHERE inher.\"class\" = ? ;");
[17]       int currEntity = this.entity_id;
[18]       do {
```

```
[19]          ps.setInt(1, currEntity);
[20]          ResultSet rs = ps.executeQuery();
[21]          if(rs.next()) {
[22]             currEntity = rs.getInt(1);
[23]             returnDIT++;
[24]          }
[25]          else
[26]             currEntity = -1;
[27]          rs.close();
[28]       } while(currEntity != -1);
[29]    }
[30]    catch(Exception e) {
[31]       throw new RuntimeException("Problems in getDIT procedure", e);
[32]    }
[33]    finally {
[34]       try {
[35]          ps.close();
[36]       } catch(Throwable t) { }
[37]
[38]       try {
[39]          conn.close();
[40]       } catch(Throwable t) { }
[41]    }
[42]    return new ClassDIT(this, returnDIT, null);
[43]   }
[44] ...
[45] }
```

Listing 12: Class Depth Inheritance Tree getter specification.

The SyQL query is more concise than the SQL corresponding query, but it also takes longer to execute. The most evident advantage of SyQL is the capability to hide the internal structure of the relational data warehouse. On the other hand, SyQL requires expert developers, who know the internal organization of the data warehouse, and how the storage happens. In Chapter 8 , we discuss this limitation and performances in detail.

After creating new attributes over a pre-existing schema, SyQL is able to perform inner join operations between the new and the old dimensions.

The mining of the information stored into a relational data warehouse requires to join data together in the proper way. Since the PROM data warehouse [110] stores data coming from multiple different data sources (source code analyzers, user effort

probes, bug tracking systems, and version control systems), the join of these heterogeneous data is a hard task. Relationships between data are formalized into the relational schema in an incomplete way. This issue may happen for different reasons, and some of them are listed below:

- the relational schema may become too complex to manage, then the designer simplifies the design;
- the relational schema may require additional constraints that slow down data acquisition, then constraints are relaxed;
- at the design-time, the relationships between data are totally unknown (see Example 7).

The last reason is the most common, especially when the data warehouse is adapted to store extra-information available in the production site, where the metrics collection system is deployed. As a proof of concept, a real example is provided below.

Example 7:

In this example, we show how it is possible to hide a complex join operation between two tables that have no foreign key constraints. The table *entity_property_ts_bug* contains part of the bug tracking information coming from the Microsoft Team System. This table has several fields that contain user names (changeby, activatedby, resolvedby, assignedto, createdby). These names are written in the following format: *Surname, Name* (e.g. Smith, John). Conversely, the field *name* of the table *user* stores the user information in the opposite format: Name Surname (e.g. John Smith).

This problem can be solved in SQL, but it requires user effort every time. With SyQL, we can implement a method that returns the id of the user who resolves the bug. Thus, the final user does not need to join *user* and *entity_property_ts_bug* anymore.

```
[1]   FROM Bug b
[2]   WHERE b.getResolverID() = 22
[3]   SELECT b.getTitle();
```

Listing 13: SyQL query that returns all the bugs titles resolved by user 22.

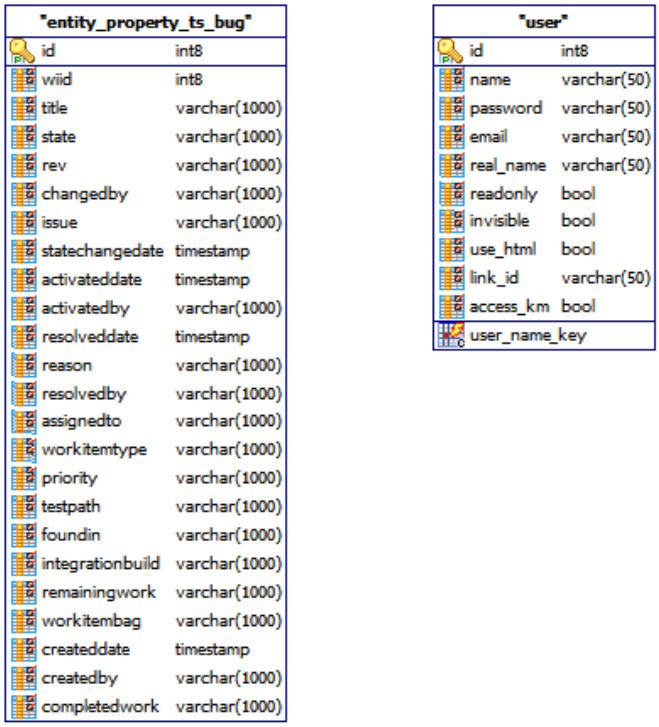

Figure 7: Schema of relations *entity_property_ts_bug* and *user*.

```
[1]   class Bug extends AbstractConcept {
[2]      @InternalCondition(cost = 20)
[3]      public Integer getResolverID() {
[4]
[5]         Integer resolverID = null;
[6]         Connection conn = null;
[7]         PreparedStatement ps = null;
[8]         ResultSet rs = null;
[9]
[10]        String [] splittedName = this.resolverRawName.split(",");
[11]        String formattedName =  splittedName[1].trim() +
[12]                  " " + splittedName[0].trim();
[13]        try {
[14]           conn = ConnectionTools.getConnection(
[15]                Engine.getInstance().getDbName());
[16]           ps = conn.prepareStatement(
[17]              "SELECT id FROM \"user\" WHERE name = ? ;");
[18]           ps.setString(1, formattedName);
[19]           rs = ps.executeQuery();
[20]           if(rs.next())
[21]              resolverID = rs.getInt(1);
```

```
[22]          rs.close();
[23]          ps.close();
[24]       }
[25]       catch(Exception e) {
[26]          throw new RuntimeException(
[27]             "Problems in getResolverID procedure", e);
[28]       }
[29]       finally {
[30]          try {
[31]             conn.close();
[32]          } catch(Exception e) { }
[33]       }
[34]       return resolverID;
[35]    }
[36] }
```

Listing 14: Definition of method *getResolverID()*.

3 Related work

In this section, we present the result of a review of the topics that relate to our research: *automatic metrics collection systems*, *domain specific languages*, and *software faulty proneness prediction models.*

The area of *automatic metrics collection systems* has been reviewed to put in evidence the peculiarities of the PROM [109][110] system; to show the basis of this system, a review of the *Software measurement theory* is also provided. The area of *domain specific languages (DSL)* has been explored to decide if the implementation of a domain specific language would have been worthy or not. Next, we analyzed the *software faulty proneness prediction* topic during our two case studies. This review helps us to use SyQL in the right way for improving the performance of the software development process by reducing the number of post-release bugs. Finally, a comparison between SyQL and other *similar tools* is presented by showing the points of strength of SyQL.

3.1 Software Measurement Theory

Galileo Galilei (1564 - 1642) thought that experimental sciences need to "Count which is countable, Measure what is measurable, and what is not measurable, make it measurable".

Software measurement makes aspects of software products and processes quantifiable by providing a better understanding of software development. Based on measurements, the outcome of projects can be predicted and influenced when needed. Thus, software measurement enables the managers to close the control loop on the development process by improving the performance of it [42].

Any software measurement tool is founded on software measurement theory.

Measurement is defined as the process by which numbers or symbols are assigned to attributes of entities in the real world, in such a way as to describe them according to clearly defined rules [48][51][42]. This definition involves two main concepts:

- *entities* – they can be objects such as parts of a program (packages, classes, methods, and attributes), events such as release milestones or actions such as testing a piece of software;

- *attributes* – they represent features or properties of entities. Thus, software measurement captures information regarding certain features or properties of objects, events or actions that are relevant to measure the software development.

The representation condition must be obeyed by any defined software metric, it means that empirical relations have to be preserved by the metric in the formal representation [48][66][51].

In software measurement, the typical entities measured are software products and software process entities. Typical attributes are, for instance, size, complexity or reliability of programs, time or effort to make a new release. Some of these attributes can be directly measured (i.e. Line of code or Cyclomatic Complexity [88]). For others, the final measure is in fact a combination of several direct measurements that take into account not only the entity itself but also its environment (i.e. reliability of programs). The attributes that are directly measured based only on the entity itself are called *internal attributes*. The attributes that are not directly measurable are called *external attributes* (there are no relations concerning names with the SyQL annotations).

To obtain a measure of an external attribute, we need to combine several measures of internal attributes. Not all mathematically possible operations for combination are meaningful in the real world. A meaningful operation has to preserve actual relationships between entities. This is known as the *representation condition* and is part of the representational theory of measurement [48]. In the representational theory, measurement is a mapping from the empirical world to a formal, relational world. A measure is a number or a symbol assigned to an entity by the mapping in order to characterize an attribute.

The operations that are actually meaningful for a measure depend on the characteristics of its measurement scale. There are five main categories of measurement scales that allow an increasingly larger set of meaningful mathematical operations: nominal, ordinal, interval, ratio, and absolute. The meaningful mathematical operators for each measurement scale are shown in Table 1.

- *Nominal scale* – It preserves the clear dissimilarity between entities by placing them in different classes, but there is no notion of ordering among the classes.

Any type of representation (i.e. numbers or symbols) of classes is valid, but there is no notion of magnitude associated with numbers or symbols. Organizing software entities into categories based on their faultiness (e.g. “faulty” or “non-faulty”) is an example of measurement on a nominal scale.

- *Ordinal scale* – It preserves dissimilarity and ordering between entities. The classes represent a ranking. Any type of representation that preserves the ordering is acceptable. There is no notion of the magnitude of the difference between two classes. Organizing software faults according to their criticality into “minor”, “major” and “critical” is an example of measurement on an ordinal scale.
- *Interval scale* – It preserves ordering and differences between entities. The classes are ordered and the difference between any two of the classes is known. An example is the timing of a release, measured in units such as days or months with respect to the beginning of the project. We can say, for instance, that release-A happened at three months after the beginning of the project, and release-B happened three months later (or at six months after the beginning of the project), but it is meaningless to say that release-B happened twice as late as the release-A.
- *Ratio scale* – It preserves ordering, the size of intervals and ratios between attributes. On a ratio scale, there is a zero element that corresponds to the lack of the attribute and the measurement increases at equal intervals known as units. An example is the length of software, measured in LOC (lines of code). The measurement starts from zero (no lines of code) and the unit is one line of code. It is meaningful to say that ClassA has twice as large as ClassB.
- *Absolute* – It preserves ordering, the size of intervals and ratios between attributes. These scales have only one possible way of measuring objects. Counting is the most common example of absolute scale. The only possible transformation applicable to those scales is the identity $t(x) = x$. Counting the developers, who participate at a Pair Programming Session or in a meeting, can be useful for monitoring the software development process.

Measurement scale	Meaningful mathematical operators
Nominal	=
Ordinal	=, <, >
Interval	=, <, >, +, -
Ratio	=, <, >, +, -, *, /
Absolute	=, <, >, +, -, *, /

Table 1: Measurement scales and their corresponding meaningful mathematical operators.

By using SyQL, it is possible to define and compute indirect measures as combinations of direct measures and others environment data. The system does not perform any explicit check to ensure that user-defined combinations are meaningful. It generally treats as ratio scale all numeric measures. Consequently, it allows users to perform addition, subtraction, multiplication and division of such direct measures.

3.2 Automated In-process Software Engineering Measurement and Analysis Systems

In this section, different implementations of Automated In-process Software Engineering Measurement and Analysis Systems (AISEMA) [33] are compared. The reviewed systems are PROM [112], Hackystat [73], Rally[1] , EPM [101], ECG [108], and SUMS [100].

Their common characteristic of these systems is the non-intrusiveness of the data collection process that is fully automated. Data collected from running software projects can be used for the understanding, steering and improving the development process of the same projects. They are specifically designed for software development, by supporting known software process improvement programs such as PSP.

The PROfessional Metrics system ([112]) has been developed by the Center for Applied Software Engineering under the supervision of Prof. Ing. Giancarlo Succi. This extensible metrics acquisition platform is able to collect metrics from different

[1]Rally Software Development, homepage http://www.rallydev.com/agile_products/lifecycle_management/

systems in different environments. It is fully developed in Java with evident benefits for portability. It is based on a client server architecture using the fat-server/thin-client paradigm. This is important to be unobtrusive in the development process. On the client side, effort data are collected and sent to the server. PROM provides plug-ins for Eclipse, JBuilder, Visual Studio, NetBeans, IntelliJ Idea, Microsoft Office, and OpenOffice; in addition the Trace tool support tracking of operating system calls (the traceability data).

On the server side, the data are collected and stored into staging tables, where a pool of threads are periodically executed, and the raw data are prepared to be stored in the long term storage tables. Reporting tools such as BIRT and SyQL are delegated to extract data from these tables. The reports are available in different formats: Web pages, HTML emails, and CSV[2] (to store query results in a textual format). The server also performs a daily checkout of the project source code (if any), then it executes the proper parser on the code, computes the software metrics and stores them into the data warehouse.

Hackystat [73] is for some aspects similar to PROM: it has a central server, it is able to gather metrics both from the process and source code. The main differences with PROM are the following: it is an Open Source project; it does not provide a Data Manipulation Language like SyQL.

Rally (previously 6th Sense) is marketed by the Rally Software Development, and it is specifically designed for the supervision of Agile teams. It is based on Hackystat, and it can be adapted to specific development environments.

EPM [101] collects and integrates data from issue tracking systems, code repositories, and email servers, and it does not collect data from developer machines. If it is compared with Hackystat, PROM, and Rally, EPM collects a restricted set of specific data.

ECG [108] aims at studying programmer behavior, it is focused to identify various types of episodes (such as copy-paste-change), and it collects specific data from development IDEs (for the moment, only Eclipse is supported). Up to now, this tool is a prototype.

SUMS [100] is also a prototype. It is specifically developed for collecting data

[2]Eclipse Fundation, Business Intelligence and Reporting Tools, homepage http://eclipse.org/birt/phoenix/

coming from the development process of high performance computing applications. It can collect both generic and specific data.

In Table 2, it is possible to find a summary of this review.

	Hackystat	Prom	Rally	EPM	ECG	SUMS
Supported version control systems	SVN, CVS	SVN, CVS, MS Team System, Visual Source Safe	SVN, CVS, Custom	CVS, SVN	-	-
Supported testing frameworks	Junit, CPPUnit	Team System Unit Testing framework	-	-	-	-
Build tools	YES	YES	-	-	-	-
Supported build tool	ANT,	Automake, MSBuild	-	-	-	-
Issue tracking	YES	YES	-	YES	-	-
Word processors	YES	YES	-	-	-	-
Email clients	-	YES	-	-	-	-
Code metrics	YES	YES	-	-	-	-
Supported Programming Languages for software metrics extraction	Java, C#, C++, Ada, Chapel, Eiffel, Fortran, Lisp, Matlab, Pascal, Perl, Tcl, ZPL, XML, HTML, JSP, SQL	C, C++, Java, C#	Same as Hackystat	-	-	-
Data Manipulation Language	-	YES	-	-	-	-
Other	-	-	-	-	-	YES

Table 2: An overview of the features present in the existing AISEMA systems.

3.3 Domain Specific Languages – DSL

In this section, we present a review of the literature about the domain specific languages.

"A domain-specific language (DSL) is a language designed to provide a notation tailored toward an application domain, and is based only on the relevant concepts and features of that domain."[80]

We want to begin our review by reporting the starting sentence of Mernik *et al.*, 2005: "Many computer languages are *domain specific* rather than general purpose"[90].

Every day we use some domain specific language during our activity, these DSL(s) are often hidden by a Graphical Front-end (Excel, HTML, etc.), but sometimes we use DSL(s) directly by writing code (LATEX, MATLAB, R, SQL, VHDL, etc.). We choose the appropriate language depending on the work we do. Where, the appropriate language is the one that helps us to reach our goal, without errors, more quickly than any other existing language.

In literature, computer scientists (and not only) refer to DSL(s) by using different names:

- *application-oriented languages* [106];
- *special purpose languages* [122];
- *specialized languages* ([13], p. 17);
- *task specific languages* [97];
- *fourth-generation languages* (4GLs) [86] [87];
- *little languages* [12].

The *little languages* usually do not implement the features found in general-purpose programming languages (GPLs).

The 4GLs are usually DSL implemented for database applications. In 1982, James Martin uses this term to identify non-procedural, high-level specification languages. Successful implementations of 4GLs are the MARK IV (1967) and the Sperry's MAPPER (1969 internal use, released in 1979). Both these languages were developed to support the activities related to the file system management. The "*fourth-generation languages"* name comes from the language level measure defined by Capers Jones [74]: a language of fourth generation requires 20 average source statements per function point (see Table 3). A more detailed version of the programming language table is freely available online [75].

Language	Level	Average Source Statements Per Function Point
1st Generation default	1	320
2nd Generation default	3	107
3rd Generation default	4	80
4th Generation default	16	20
…	…	…
Java	6	53
SQL	25	13

Table 3: Programming Languages Table (partial).[3]

We can see from Table 3 that the SQL level is higher than the Java one. It means that SQL on the same function point is more concise than Java in the average; obviously, the Function Point must be implementable for both languages. By using the facilities provided by these two languages, SyQL can achieve a level higher than SQL. Details about the comparison are provided in Section 7.2 . To achieve this result, we have combined part of SQL with a *domain-specific embedded language* [69] (application library) written in Java. This application library enables us to translate a SyQL in SQL, and to map the SQL results in Java's classes. In this dissertation, we refer to this library either with the terms “concept library” or “SyQL library”.

A DSL is defined by a grammar, which is usually written by using the Backus-Naur Form (BNF) [5]. The BNF is a meta-syntax used to define the context-free grammars. By using this syntax, it is possible to define the lexical and the syntactical rules of a language. There are different extensions of BNF: the EBNF (Extended BNF), and the ABNF (Augmented BNF). These formalisms allow to synthesize a Finite State Machine (lexer) and a Stack Automaton (parser), which are able to recognize properly a well-written program. Where, well-written means that the program text conforms with the language rules (written in BNF/EBNF/ABNF).

Several parser/lexer generators are freely available today:

- ANTLR [102] (http://www.antlr.org/);
- JavaCC [79] (https://javacc.dev.java.net/);
- Coco/R(http://www.ssw.uni-linz.ac.at/coco/);
- CUP [70] (http://www.cs.princeton.edu/~appel/modern/java/CUP/);

[3]SPR – Programming Language Table – http://www.spr.com/programming-languages-table.html

- GNU bison (http://www.gnu.org/software/bison/);
- Pre-cc (http://formalmethods.wikia.com/wiki/PRECC);
- SableCC [57] (http://sablecc.org/).

These parser generators speed up the implementation of a DSL front-end. To implement SyQL, we have chosen JavaCC for developing and maintaining the language parser. This generator accepts an EBNF grammar as input and produces the Java parser code as output. The main reason for this choice is due to the fact that we already had maintained parsers by using this generator. The experience helps to speed up the implementation and the modifications of the language. Having the possibility to change the grammar easily is very important, because the DSLs change more frequently than general-purposes programming languages [124]. To assist the editing of the JavaCC EBNF grammar file, we have adopted the JavaCC Eclipse plug-in.

As stated by Mernik *et al.* [90], the development of a DSL is hard, because it requires both language development and domain expertise, which are not easy to find in a single person. To reduce the implementation effort, we have reused part of the existing SQL grammar, and part of the syntax rules of the SQL expressions (see Section 5.1). This approach is pretty common in this field [19][22], since the definition of a completely new grammar may be difficult to do in a short time.

3.3.1 DSLs Implementation Patterns

To success in the development of a Domain-Specific Language, the right implementation pattern must be chosen. Korsar *et al.*, 2008 [80] have compared different approaches on the same languages, these implementation patterns have been already classified by Mernik *et al.*, 2005 [90] and by Spinellis, 2001 [114]. The differences between these patterns are sometimes very blurring, hence it is difficult to categorize a specific implementation of a DSL in one of them.

Below, a brief presentation of these implementation patterns is given:

- *Preprocessing* – DSL constructs are translated into base language constructs, and they are statically analyzed by using the base language processor. This architectural choice makes the error-messaging difficult to understand. There are different ways to put in place this mechanism.
  - *Macro processing* – The semantic of new language constructs is defined by

using other constructs in base language. This sub-pattern is easy to use if the base language uses a preprocessor (e.g. C/C++).

    - *Source to source transformation* – The DSL source code is translated into the source code of the base language. This approach is commonly used by the parser generators (e.g. JavaCC → Java, ANTLR → C#/Java, etc.)
    - *Pipeline* – The DSL processor recognizes the DSL new constructs by converting them into the base language. Base language constructs are merely copied in the output. The output stream of the DSL processor becomes the input stream of the base language processor.
    - *Lexical processing* – The architecture is the same of the *Pipeline*, but the translation is done using conversion rules matched by regular expressions. The syntax tree-based analysis is not used.

- *Embedding* – This approach reuses all the lexical/syntax of the host language (GPL). New operators/data types are defined. The basic form of embedding is represented by application libraries.
- *Compiler/interpreter* – In this approach, the language developer uses standard compiler/interpreter techniques to implement the DSL. In this way, a complete static analysis of DSL program can be done. The most evident limitation of this technique is the cost of the implementation. Because, the developers must have both language development and domain expertise [90].
- *Compiler generator* – This approach presents similarities with the previous one, except for the development of the language front-end (parser) that is done by using a parser generator (compiler-compilers). So, the language developer is a user of a DSL that allows to specify lexical/syntax rules by using BNF/EBNF/ABNF syntax.
- *Extensible compiler/interpreter* – This approach is probably one of the most interesting approaches that we have reviewed. A language is here extended by using domain-specific optimization rules and/or domain specific code generation rules. All this mechanism is usually implemented by using facilities like reflection and introspection. The most evident limitation of this approach is the fault-proneness of the extension phase. Because, it is easy to introduce buggy libraries that make the generated code buggy.

- *Commercial Off-The-Shelf (COTS)* – In this approach, an existing language is applied to a specific domain (e.g. XML). XML is used to solve a large quantity of problems, the same happens for Comma Separated Value (CSV), which is widely used to interchange data in the scientific field.

To implement SyQL, we have adopted an *hybrid* approach: we have used JavaCC *compiler generator* to generate an *extensible interpreter* by using Java's reflection.

3.4 Software Fault-Proneness Prediction

In this section, we provide a review of the main works on the software fault-proneness prediction, since the integration into the SyQL libraries of one or more fault prediction models may provide benefits to the final user by adding value to the entire AISEMA system. By having the possibility to predict defects in software, the software companies may reduce the fixing costs, therefore, the efficiency of the software development process can increase. The Boehm's Curve (see Figure 8) [18] shows that the cost of fixing a defect in a software increases over time. During the 1970's, when Boehm was compiling his research, the cost of a bug was probably higher (in the average) than today. This curve has been flattened by the increased number of personal computers and by the software development methodologies like *Agile* [64] and *Big Design Up Front (BDUF)* [4]. But the problem is still here, a famous sentence of Sturdy says: “The cheapest bug to fix is one that doesn't exist”. Unfortunately, a code without any bug is hard to get due to the undecidability of the Halting problem (over a Turing machine), which makes the code correctness impossible to prove in general.

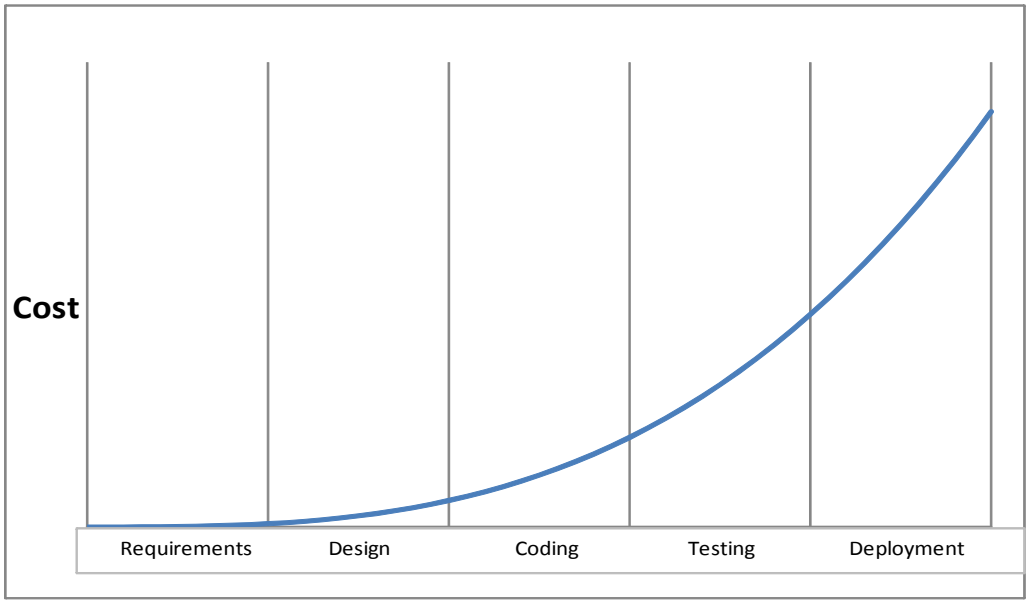

Figure 8: Boehm's Curve

A possible approach to reduce the number of defects in code is given by machine learning and other regression techniques. By using software metrics to train different kinds of models, it is possible to predict if a piece of code can be faulty or not. One of the main benefits is that the process of software metrics acquisition and fault-proneness evaluation can be fully automated, and it is what we have tried to do in Case Study A [14] and B [16], in which we have only used automatically collected data. So that, humans are involved only in the final part of the process, when the results are evaluated, and decisions are taken. These prediction models are useful, and they can make the code inspections more effective and less expensive in terms of time than in the past. They can also help to discover the software modules that require more testing effort [63]. Indeed, it is well-known that the distribution of faults is not uniform among the code, and it is influenced by the modifications [8].

One of the limitations of this metric-based approach is the complexity of the training phase: Fenton *et al.*, 2000 [50] have shown that neither size metrics (LOC, SLOC, etc.) nor complexity metrics (McCabe Cyclomatic complexity, and others complexity metrics [130]) are good predictors of fault-proneness. This evidence that the problem is more complex than expected, and it cannot be tackled by using only one family of metrics [49].

In terms of classification, Briand et al. [24] get the best results on OO software. They are able to classify if a class is faulty or not with the 68% of the correctness, and 73%

of the completeness (see Definition 3 on page 44). They have performed this analysis on Java's code. The set of the independent variables includes a subset of measures related to coupling [23], a set of polymorphism measures [11], and a subset of the Object Oriented Chidamber and Kemerer metrics [31]. A total number of 22 measures has been used to build the regression model.

The regression method adopted is the *Multivariate Adaptive Regression Splines* (MARSplines) [55]. This non-parametric regression procedure makes no assumption about the relationship between the dependent and independent variables. The *MARSplines* training algorithm constructs this relation from a set of coefficients and basis functions that are entirely "driven" from the regression data. By using "divide and conquer" strategy, the input space is partitioned into regions, each with its own regression equation. This makes *MARSplines* particularly suitable for problems with input dimensions greater than two, where the other techniques may be affected by the curse of dimensionality. As stated by the authors, this approach is viable from the economic standpoint (they have proposed a cost-benefit model), but the extracted model performances are not enough to make it usable in a software company in the real world. In the conclusion, they propose to enhance the performances by building more complex models with a higher number of features evaluated. The same stated by Fenton *et al.*, 1999 [49] and Nagappan *et al.* [96]:

- Fenton *et al.*, 1999 [49] propose a more complex prediction model based on Bayesian Belief Networks (BBN). In this critique to the defect prediction models, they claim the necessity to have more information about the development process. Indeed, all the reviewed papers use only software metrics.
- Nagappan *et al.* [96] claim that is necessary to work on more data (test coverage, usage profiles, change effort), with more automation, and on a large quantity of projects.

Definition 3:

Correctness (precision):

it is defined as the number of entities (packages, classes, packages) correctly classified as faulty, divided by the total number of methods classified as faulty. Low correctness means that a high percentage of the methods being classified as fault-prone do not actually contain a fault. We want correctness to be high, so the company can save resources needed for the code inspections.

Completeness (recall):

it is defined as the number of software entities (packages, classes, packages) correctly classified as faulty, divided by the total number of faulty methods in the system. It is a measure of the percentage of faults that would have been found if the prediction model was used in the stated manner. Low completeness indicates that many faults are not detected. These faults would then slip to subsequent development phases.

Study	Modelling Technique	Automatic data collection	Method Based	Programming Language	Project size
Basili *et al.* [9]	LR	NO	NO	C++	180 classes 8 modules
Briand *et al.* [24]	MARS	NO	NO	Java	2 projects 212 classes, 2,707 methods
Cartwright *et al.* [29]	Linear regression	NO	NO	C++	133,000 lines of code, 32 classes
Subramanyam *et al.* [116]	OLS	NO	NO	C++, Java	706 classes
Gyimóthy *et al.* [61]	LR+ML	NO	NO	C++	Mozilla project >1,000,000 LOC
Kanmani *et al.* [76]	NN	NO	NO	C++	200 similar systems, 1,185 classes
Tomaszewski *et al.* [121]	Linear regression	NO	NO	-	2 systems, 1800 classes, 1,100,000 LOC
Zhou *et al.* [129]	ML + SFR	NO	NO	C++	Subset of KC1 NASA project. 145 classes (about 40,000 LOC)
Nagappan *et al.* [96]	LR	NO	YES	C++/C#	5 projects 1,100,000 LOC

Morasca *et al.* [93]	LR + RS	NO	NO	BLISS	>100 modules
Case Study A [14] (Our Study)	ML	YES	YES	C#	99 modules 4,039 classes 11,083 methods 95,187 SLOC
Case Study B [16] (Our Study)	ML	NO	NO	Java	4,813 classes, 68,666 methods, ~80,000 SLOC

LR = Logistic regression, MARS = Multivariate Adaptive Regression Splines, OLS = Ordinary Least Square regression, NN = Neural Network, ML = Machine Learning, SFR = Severity Fault Ranking, RS = Rough Set

Table 4: Related work on fault proneness prediction.

In Table 4, the most relevant and recent works are listed. We have compared the following peculiarities:

- *Modelling technique* adopted – the regression algorithm adopted on the data set. The mostly used techniques in these studies are the following:
  - the Logistic Regression [68] (4 over 10 reviewed works have used it): [9] [61][96][93];
  - Neural Network [17] and Machine Learning: (3 over 10 reviewed works have used it) [61][76][129];
  - the Linear Regression (2 over 10 reviewed works have used it): [29][121];
- Presence of an *automatic effort and metric collection system* – no one of the reviewed works have used automatically collected data to build the initial data set by making the compilation of the researches more expensive in terms of time and in terms of money. The data colletion of following studies involves people directly:
  - Basili *et al.* [9] did the experiment with students;
  - Subramanyam *et al.* [116] map the defects with classes manually by using the defect resolution logs gathered from the configuration management system of the company;
  - Gyimóthy *et al.* [61] use the bug tracking data together with the source patch files to map the defects with classes, this task can be only partially automated;
  - Morasca *et al.* [93] use customer failure reports to map defects with software modules;
  - Kanmani *et al.* [76] did an experiment with students, and expert testers

find the defects;

- Zhou *et al.* [129] pre-process and rank manually the defects according to their severity. The mapping between classes and defects is done by using the error reports;

- *Method based* prediction – it is shown if the predictions of fault-proneness are performed to method level or not: only our case study A [14] and Nagappan *et al.* [96] predicts if a method is fault-prone or not. Method-level predictions can be interesting, because they can improve the efficiency of the code inspections by pointing the fault prone methods instead of classes;
- *Programming language* – most of the articles report analysis performed on C++ code, only two of the papers presented in this review are about Java code analyses; in case study B [16], we have also analyzed Java code by using an ad-hoc parser specifically designed for Java. Only part of the code base of Nagappan *et al.* [96] and our Case Study A [14] are constituted by C# code;
- *Project size* – the size metrics of the projects analyzed by the authors of the papers. We can see that the sizes of our case studies are aligned to the others. Not in all the papers the lines of code are reported; Tomaszewski *et al.* [121] have not told anything about the characteristics of the analyzed projects, due to the bond of confidentiality between the authors and the company, which made the code available.

In the last rows, our two case studies are included, and the details are presented in Chapter 9 . In these two case studies, SyQL has helped us to extract information from the PROM data warehouse [109] by speeding up the process of building the training data set. As stated by Nagappan et al. [96], the mapping process between faults and code locations is a key success factor for this kind of researches, because it enables the researchers to get additional knowledge by exploring large software repositories [37][52][113]. The methodologies adopted to carry out the mapping differ from each other. This makes difficult to compare the results coming from different studies in terms of completeness/correctness, because a different mapping methodology may lead to a different data set. Thus, the final results could be biased. With the PROM metrics collection system active, the data about the effort spent by the developers on a specific part of code, together with the software metrics are automatically collected

and stored into relational tables by making the mapping between defects and code possible. This system and other equivalent ones (see Section 3.2) can make researches in this field more affordable, and they can bring tangible benefits in the short time inside the companies that allow to instrument their software development chains. As stated by Cartwright *et al.* [29], these fault-proneness prediction models probably have only local significance, hence each team of developers shall have its own prediction models. The initial training of the models (feature selections, choosing the proper model) cannot be up to now fully automated. These models require to be retuned periodically to ensure the highest performances.

Gyimóthy et al. [61] have compared two different machine learning techniques (Decision Tree [104], Neural Network [17]) and the Logistic Regression. Summarizing their results, they have tried to predict if a class is fault-prone or not by using CK metrics. They get the best prediction performance by using all the metrics together. In Table 5, a summary (partial) of the results is shown. In terms of correctness, the Logistic Regression performs better than the other two techniques, but in terms of completeness the Decision Tree is the best. The Decision Tree is the one that produces models having value of correctness and completeness that are similar to each other. In our case studies, the Decision Tree, on our data set, outperforms any other technique, and we get much better performance than any other of the studies included in this review.

	Logistic Regression	**Decision Tree**	**Neural Network**
Correctness	72,57%	68,38%	68,94%
Completeness	65,24%	67,84%	64,76%

Table 5: Gyimóthy et al. 's summary of the results (partial). [61]

3.5 Similar Tools

In this section, we provide a comparison between SyQL and other similar tools. Different aspects are evaluated and discussed evidencing different advantages and disadvantages.

There are several works on languages that can be used to query repositories of software data. The features that appear to be the most relevant to consider are the following: the capability to perform temporal queries on product and process metrics,

the possibility to help the user to filter the results through a mechanism of value abstraction based on linguistic variables [126] (such as high, medium, low), and the possibility to be used in a general context. In addition, it is important to consider some other technical aspects such as: support of combined analysis (software metrics/effort), temporal management, fuzzy logic support (linguistic variables), supported programming languages (languages from which the tool can extract information for analysis tasks), and object orientation. We remark that *object orientation* is a very important feature for us, because, upon it, we have built the type abstraction mechanism by making the queries executable against the PROM meta model. In Table 6, we use these criteria to compare some of the most relevant existing work and SyQL.

Languages	**Support to combined analysis (software metrics / effort)**	**Temporal management**	**Fuzzy Logic**	**General Purpose language**	**Supported programming languages (for analysis task)**	**Object Orientation**
LINQ [89]	NO	NO	NO	YES	None	YES
FuzzySQL [36]	NO	NO	YES	YES	None	NO
SQLf [20]	NO	NO	YES	YES	None	NO
.QL [92]	NO	NO	NO	NO	Java	YES
DmFSQL	NO	NO	YES	YES	None	NO
SCQL [65]	NO	YES	NO	NO	None	NO
NDepend – CQL[4]	NO	NO	NO	NO	All .NET languages	YES
COOR – CQL [38]	NO	NO	YES	NO	C++	NO
SyQL	YES	YES	YES	NO	C/C++, Java, C#, VB.NET	YES

Table 6: Comparison between different query languages

[4] http://www.ndepend.com/

3.5.1 Language INtegrated Query – LINQ

Language Integrated Query – LINQ [89] is designed to be embedded into another programming language. Hence, queries can be performed with the same expressive power from a program written either in C# 3.5, VB 9.0, or any other .NET language.
A subset of the LINQ syntax is similar to the SyQL one, but it is designed to achieve different purposes. LINQ is more general and can perform queries on different data sources, while SyQL is tight with a specific data source (the PROM metrics data warehouse). By using LINQ, it is possible to query XML documents, databases, and other data sources; in fact, it is possible to create new data providers by implementing specific interfaces. In Listing 15, the program prints out the name of the customers who have placed an order with id greater than 199.

```
[1]  Northwnd nw = new Northwnd(@"northwnd.mdf");
[2]
[3]  var companyNameQuery =
[4]     from ord in nw.Orders
[5]     where ord.OrderID > 199
[6]     select ord.Customer.CustomerName;
[7]
[8]  foreach (var customer in companyNameQuery)
[9]  {
[10]    Console.WriteLine(customer);
[11] }
```

Listing 15: Sample LINQ code (written in C#).

SyQL can only read the data, whereas LINQ can also modify them. Both languages are fully object oriented. The mechanism adopted by LINQ for mapping relations into classes and viceversa is based on code annotations. Due to the fact that LINQ can also modify the data, the code annotations are richer than SyQL ones, and it is possible to specify extra information like relationships, typing, constrains, etc. In Listing 12, it is possible to see a declaration of a class named *Order* that is automatically serialized / deserialized from a database table named *Orders*. This table has two columns: OrderID, and CustomerID. The last one specifies a relationship with the entity of Customer.

```
[1]  [Table(Name="Orders")]
[2]  public class Order
```

```
[3]  {
[4]     [Column(Id=true)]
[5]     public int OrderID;
[6]     [Column]
[7]     public string CustomerID;
[8]     private EntityRef<Customer> _Customer;
[9]     [Association(Storage="_Customer", ThisKey="CustomerID")]
[10]    public Customer Customer {
[11]       get { return this._Customer.Entity; }
[12]       set { this._Customer.Entity = value; }
[13]    }
[14] }
```

Listing 16: Definition (written in C#) of a class that is automatically mapped to the corresponding DB table.[5]

SyQL allows the use of Fuzzy equal operator and temporal tokens, whereas LINQ does not. Summarizing, LINQ is too complex for our needs, and we have found difficulties to fit it into the PROM platform, which is completely developed in Java and not in .NET. A clone of LINQ for Java exists (Jaque[6]). However, we have found more worthy to develop a query engine from scratch, which can fulfill completely all the requirements by using specifically designed solutions. The main goal of LINQ is to make the persistency management of the objects transparent to the programmers. Whereas, the main goal to SyQL is to help the researchers and the software engineers to understand better the development process. It is easy to understand that it is hard reconciling these two distinct goals. Therefore, SyQL has been implemented from scratch.

3.5.2 fuzzySQL

FuzzySQL [36] is a commercial relational database front-end; it supports fuzzy conditions and it is designed to assist the user during the analysis tasks.

The main difference between SyQL and FuzzySQL is that FuzzySQL is a general-purpose relational database front-end, while SyQL is a specific tool to perform information retrieval tasks on the metrics data warehouse with additional features to handle temporal analysis of the software development process. A fuzzySQL query is wrapped into a standard SQL statement.

[5]Take from MSDN, http://msdn.microsoft.com/

[6]Java integrated QUEry library, Project Homepage, http://code.google.com/p/jaque/wiki/Version_One_dot_Five

In Listing 17, an example of fuzzySQL is provided. This query returns all the projects that have a high budget and short duration. The fuzzySQL query engine computes the truth values for the actual values of *duration* and *budget*. Finally, it returns a result of four columns: *project*, *duration*, *budget*, and *truth value*.

```
[1]   SELECT project, duration, budget
[2]   FROM Projects
[3]   WHERE budget is High
[4]     AND duration is Short;
```

Listing 17: Sample fuzzySQL query.

Summarizing, FuzzySQL has only one similarity compared to SyQL, it is the presence of the IS (fuzzy equal) operator. This operator (like in SyQL) computes the truth value for the actual record. If more than one IS-clause is used, the final truth value of the record is computed by using Zadeh's rules. This language is not object-oriented, and it does not support abstraction concepts like SyQL. Therefore, if we use this language instead of SyQL, the queries still remain long and difficult to read and reuse.

3.5.3 SQLf

SQLf [20] is an extension of the standard SQL that supports both vague predicates and vague quantifiers (linguistic variables). This tool is similar to FuzzySQL [36] and to dmFSQL [28]. Therefore, the differences between SQLf and SyQL are the same that exist between SyQL and the other two languages.

In Listing 18, an example of SQLf query is provided. This query returns all the departments where most of the employees are old. The equality *"='old'"* evaluates the *age* field using the membership function of the linguistic variable "old".

```
[1]   select #dep from Emp
[2]   group by #dep having most are age = 'old'
```

Listing 18: Sample SQLf query.

Summarizing, this language is designed to address the problem of interpreting values. The query processor provides a mechanism for abstracting values. With SyQL, a similar mechanism is available.

3.5.4 Semmle – .QL

.QL [92] is a commercial tool designed and commercialized by Semmle[7]. This tool enables engineers to perform code analysis tasks as reverse engineering and discovery of bad code smells.

.QL is the most similar tool to SyQL. Their goals are similar, but SyQL was designed to fulfill a large amount of requirements. In the following, we present a comparison between the two tools:

- SyQL can be used to perform code and process metrics analysis; on the contrary .QL can handle only code metrics;
- SyQL can perform tasks on different projects written in different programming languages, vice versa .QL can perform analysis only on Java projects;
- SyQL can handle temporal aspect by showing metrics evolutions, .QL does not;
- .QL data acquisition fully relies on Eclipse RCP components, whereas SyQL relies on PROM components, which are designed and implemented specifically to collect particular types of data;
- all the .QL queries must be translated to SQL before executing, hence, .QL is always wrapped into SQL. SyQL overcomes this limitation by using a more complex architecture that allows to build an abstraction layer between users and tables at run-time;
- both are extensible, but SyQL has a better architecture that distinguishes concepts by the other components. In .QL, this distinction is not so clear. (source: .QL project manager @ OOPSLA 2008). Therefore, the extension process is easier and quicker than the .QL one;
- it is possible to extend .QL by using .QL itself, on the contrary, with SyQL, we have to use Java;
- SyQL supports the fuzzy logic conditions, .QL does not. It can make the difference on usability, when the user does not know the order of magnitude of the metrics involved in the query.

In Listing 19, the query returns all the types that declare a child type without implementing the method “visitChildren”. This kind of queries are useful to check if

[7]Semmle, Semmle Code: Flexible Code Analysis, http://semmle.com/semmlecode/

the coding standards are violated or not.

```
[1]  from ASTType t
[2]  where not(t.declaresMethod(" visitChildren ")
[3]  select t.getPackage (), t , t.getAChild()
```

Listing 19: Sample .QL query.

Summarizing, .QL is designed to solve a subset of the problems addressed by SyQL. SyQL can be used also by people who are not directly involved into the development process, but who know the domain of software engineering. Usually these users are not strictly interested in code stuff, but they are interested in other process aspects, like effort spent, time to completion, number of bug fixed, etc.

3.5.5 dmFSQL

dmFSQL [28] is an extension of the previously presented fuzzySQL [36]. The language developers added general-purpose data-mining oriented constructs. This enables language users to perform basic data mining operations (like clustering, classification, etc.) directly from the query.

The query provider is implemented as an Oracle front-end. If compared with SyQL, dmFSQL has the same differences that we have already listed for fuzzySQL before. dmFSQL is designed to be used by data mining experts, whereas SyQL does not; SyQL uses data mining technology to give a better understanding of the development process to the final user. However, the complexity is always hidden into the concept libraries written in Java.

SyQL is very different from dmFSQL, and the only evident similarity between them is the fuzzy logic support, because the purposes of these two languages are different.

In Listing 20, an example taken from Carrasco *et al.* [28] is shown.

```
[1]  CREATE_MINING PROJECT p_direct_marketing
[2]  ON TABLE example_clients
[3]  WITH COLUMNS FOR
[4]  CLUSTERING (
[5]     payroll WEIGHT_CLUSTERING 0.4
[6]             FCOMP_CLUSTERING FEQ
[7]             ABSTRACTION_LEVEL_CENTROID LABEL,
[8]     balance WEIGHT_CLUSTERING 0.2
```

```
[9]             FCOMP_CLUSTERING FEQ
[10]            ABSTRACTION_LEVEL_CENTROID AVG,
[11]    area    WEIGHT_CLUSTERING 0.4
[12]            FCOMP_CLUSTERING FEQ
[13]            ABSTRACTION_LEVEL_CENTROID LABEL )
[14] CLASSIFICATION (
[15]    payroll WEIGHT_CLASSIFICATION 0.4
[16]            FCOMP_ CLASSIFICATION FEQ,
[17]    balance WEIGHT_ CLASSIFICATION 0.2
[18]            FCOMP_ CLASSIFICATION FEQ,
[19]    area    WEIGHT_ CLASSIFICATION 0.4
[20]            FCOMP_ CLASSIFICATION FEQ);
[21]
[22] SELECT_MINING CLUSTERING p_direct_marketing
[23] INTO TABLE_CLUSTERING clients_clu,
[24] TABLE_CENTROIDS clients_cen
[25] OBTAINING OPTIMAL_H3 CLUSTERS;
[26]
[27] SELECT_MINING CLASSIFICATION p_direct_marketing
[28] FROM clients TO clients_cla
[29] ACCORDING_TO
[30] TABLE_CENTROIDS clients_cen
[31] WHERE CLUSTER_ID IS 2 THOLD 0.7;
```

Listing 20: Sample dmFSQL query.

From the first statement, it is possible to see that dmFSQL is not only DML, but it is also Data Definition Language (DDL). The *CREATE_MINING* statement defines the analysis project *"p_direct_marketing"* over the *"example_clients"* table; it also specifies information for clustering, and the specified attributes *"payroll*, *balance, area"* participate to the clusters' definition with the following parameters: weights of 0.4, 0.2, 0.4 respectively (lines [5], [8], [11]), Fuzzy Equal Comparator to obtain the distance matrix for all of them (lines [6], [9], [12]). Finally, the clusters are characterized by the contents of the columns "*payroll"* and *"area"* (lines [7], [13]), and by the average of the *balance* values contained in each cluster (line [10]). The same happens for the classification process, where comparison operators and weights are defined (lines [14]-[20]). In the first SELECT_MINING clause, the clusters and the centroids are created, and they are stored into the tables "clients_clu", and "clients_cen" respectively.

In the last SELECT_MINING CLASSIFICATION statement (line [27]), a problem of

focusing a marketing campaign is solved by selecting all the clients belonging to the cluster 2 with the minimum degree of 0.7 (line [31]). The centroids stored in table *"clients_cen"* are reused to perform the classification process (line [30]). Then, the final result is stored in the new table "clients_cla" (line [28]).

Summarizing, this language is designed to be used by data-mining experts on problems arising from different domains. On the contrary, SyQL is designed to be easy, and to enable all the software process stakeholder to access easily to the process data.

3.5.6 SCQL

SCQL [65] is developed by a group of researchers of the Department of Computer Science of the University of Victoria.

SCQL is a domain-specific temporal query language used to retrieve information from a relational database containing information gathered from a source control system.

Both, SCQL and SyQL, have keywords to manage temporal data. The main difference is that SyQL is designed to be extended to handle different aspects of the development process (effort, code metrics, requirements, etc.), while SCQL is designed only to perform information retrieval tasks on software repository data. SCQL has not fuzzy logic support.

In Listing 21, an example taken from Hindle *et at* [65] is shown. This example implements the Formula 1. The query returns an answer if existing an author *a* who only modifies the files that another author *b* has already modified.

$$\exists a, b \in Author\ s.t.\ a \neq b \wedge \forall\ r\ Revision\ s.t.\ isAuthor(a, r) \Rightarrow$$
$$\exists r_b \in Revision\ s.t.\ before(r_b, r) \wedge isAuthor(b, r_b) \wedge r.file = r_{b,file}$$

Formula 1: Logical description of the query shown in Listing 21.

```
[1]  E(a, Author) {
[2]     E(b, Author) {
[3]        a!=b &&
[4]        A(r, a.revisions) {
[5]           A(f, r.file) {
[6]              Ebefore( r2, f.revisions, r) {
[7]                 isAuthorOf( b, r2)
[8]              }
```

```
[9]            }
[10]        }
[11]     }
[12]  }
```

Listing 21: Sample SCQL query.

The query in Listing 21 first finds all the couples of distinct authors (lines [1]-[3]). Then all the revisions produced by author *a* (line [4]) are iterated. So that, the language interpreter checks (per each revision) if the file of the current revision has another previous revision that belongs to author b (lines [6]-[7]). The expression *a.revisions* retrieves all the revisions related to the author *a* (line [6]); while *isAuthorOf(b,r2)* returns true if *b* is the author of the revision of the file *f*, otherwise false (line [7]).

Summarizing, SCQL is a query language that allows to explore the evolutions of software repositories by using powerful first-order logic constructs. Temporal aspect is also contemplated. Unfortunately, the quantity of available functions is limited to repository mining. By using SyQL, it is possible to overcome this limitation by enabling the user to access to a wider set of process metrics.

3.5.7 NDepend – CQL

NDepend[8] is an application developed by SMACCHIA.COM S.A.R.L[9] a privately-held software company specializing in the creation of software tools for supporting the development process. This tool has been adopted by the major software houses. The user can interact with the graphical user interface by using CQL (Code Query Language) for extracting information from .NET projects. By this language, the user can extract different kinds of information from the source code base such as static source code metrics, test coverage data, etc.

Comparing NDepend (CQL) and SyQL, it is necessary to remark that these two languages are designed to achieve different goals. With NDepend, it is easier keep under control a set of .NET projects, because the tool is highly integrated with the .NET environment. SyQL is more platform independent (it supports also C/C++ and Java), and it aims to help the users to control different aspects of the software development process. By using SyQL, it is possible to compare the values of a

[8]NDepend – Homepage – http://www.ndepend.com/

[9]SMACCHIA.COM – Homepage – http://smacchia.com/

specific metric inside a specified time interval (e.g., showing the size evolution, in terms of lines of code, of a certain class in the last six months). Conversely, NDepend makes possible only comparing two different versions of the code by showing the changes.

SyQL makes possible running effort analyses on source code (e.g., compute the total effort spent by the developers on a specific package/namespace); the user can track the bug fixing process by pointing at the methods that have been modified during a specific fixing task. With NDepend, this kind of queries is impossible to do, because process and product data are not collected by a metrics collection system like PROM.

CQL is now at revision 1.8 (2009/10/23), the language is not designed to be extended by the user like SyQL. The CQL language specification[10] defines and documents the information available to NDepend users. Most of them are specific of the .NET platform.

Both CQL and SyQL have not language closure. However, SyQL allows the user to manipulate results by using the set operators (UNION, EXCEPT, INTERSECT).

In Listing 22, the query returns all the methods that invoke and are invoked statically by the method *MyNamespace.Foo.method1()* . This query can be useful to identify possible stack overflows.

```
[1]  SELECT METHODS
[2]  WHERE   IsUsing  "MyNamespace.Foo.method1()"
[3]    AND   IsUsedBy "MyNamespace.Foo.method1()"
```

Listing 22: Sample NDepend – CQL query.

Summarizing, NDepend is designed to deal only .NET code, whereas SyQL is designed an extensible tool, which can deal with different aspects of the software development process (effort, bug tracking, etc.). NDepend is created to be used only by developers, whereas SyQL can be used by all the process stakeholders.

3.5.8 COOR – CQL

The COOR system [38] is a repository to promote the Object Oriented code reuse. This repository collects and classifies Object Oriented software artifacts in a

[10]NDepend – Code Query Language 1.8 Specifications – http://www.ndepend.com/CQL.htm

semiautomatic way. The retrieving process is assisted by an ad hoc fuzzy query language called Component Query Language (CQL). CQL supports different fuzzy operators, which are specifically designed to evaluate object descriptions.

Compared with CQL, SyQL can do the same operations by providing an additional level of abstraction for types. SyQL can be configured to work on several meta-models, whereas CQL does not provide this flexibility.

In Listing 23, we provide a CQL query that returns all the software components with the following features:

- manage visual menus (see line [1]);
- support function of text searching (less important) (see line [2]).

```
[1]  and(a_good_extent_of manage menu,
[2]  a_low find text)
```

Listing 23: Sample COOR – CQL query.

Summarizing, CQL is a domain specific language designed to assist code reusing tasks, and it supports various fuzzy operators. The CQL queries are executed against a relational database that implements a meta-model, specifically designed to address this problem.

4 The PROM System

In this Chapter, we present the entire PROM [109][110][112] system. This is a high level overview on the whole system to help the reader to understand better where the work presented in this dissertation is used.

The structure of the Chapter is the following: Section 4.1 presents the architecture of the PROM System; Section 4.2 describes the two software metrics extractors implemented by us and used in the two case studies described in Chapter 9 ; finally, Section 4.3 positions SyQL inside the global architecture.

4.1 The Architecture

PROM is a distributed non-intrusive system for collecting software and process metrics [109][112]. Figure 9 shows the role of SyQL and Lagrein [72] in the system. The metric collection system is distributed: the application plug-ins are installed on the clients, and they are able to trace the user activities inside the most common IDE(s) (Microsoft Visual Studio, Eclipse, Emacs, etc.); the Source Code Analysis component (hereafter promPM) runs on a standalone machine that takes daily snapshots of the source code from the File Control System. These components send the collected data to the Metric Server using Apache XML-RPC protocol implementation. Then the Metrics Server organizes these data and stores them inside the relational data warehouse, which is a PostgreSQL[11] instance. Finally, the extracted information are delivered to the managers and to the developers in two possible ways, either by an automatic statically generated report (using Eclipse BIRT) or by Lagrein in a "dynamic/visual" way. Lagrein can also launch SyQL queries by visualizing the results.

Most of this process is fully automated. The PROM administrators have only to supervise the system by inspecting the quality of the data acquisition through automatic tools [32] sporadically.

The architecture of PROM is designed to make possible the integration of other data sources already present in the process. Because, in totality of the case studies [32], we had to integrate other information already collected (effort estimations, criticality

[11]PostgreSQL – Project Homepage – http://www.postgresql.org/

estimations, bug reports, etc.). The format of these data is not homogeneous, hence the possibility to develop quickly ad hoc software probes is of paramount importance.

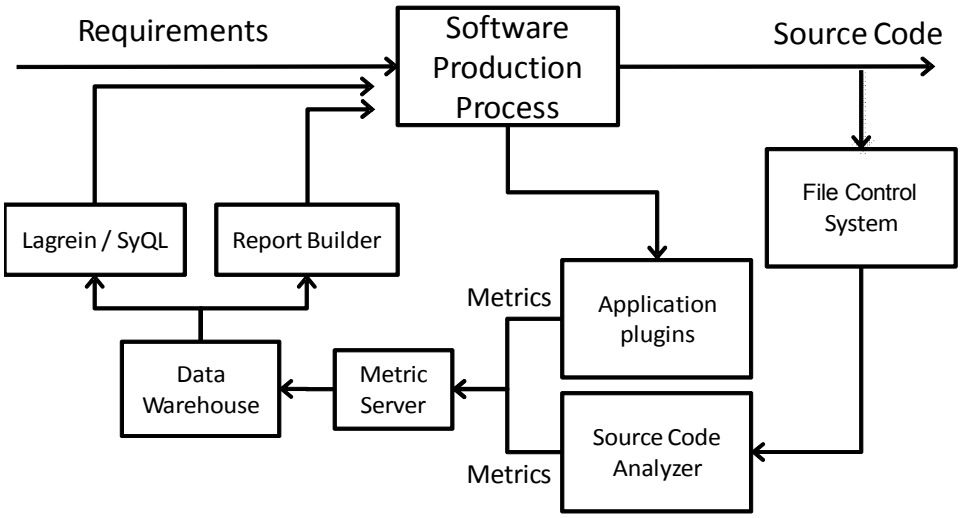

Figure 9: The PROM architecture.

4.2 Software Metrics Extractors

In this section, the source code metrics extractors used during the case studies described in Chapter 9 are presented. Both have been developed by us using the *Test Driven Development* (TDD) approach [10]. We concentrate our attention on these two components because they have been very complex to implement, test, and deploy; they are integrated with the *promPM* [109] suite, which provides facilities to the source code parsers for computing software metrics by using a common code meta-model. This meta-model is populated by the parsers, after that *Source Code Analyzer* computes the final software metrics to send to the PROM *Data Warehouse*.

4.2.1 Code Meta-Model

The choice of the abstraction meta-model for representing source code entities is crucial, because a wrong choice in the representation could make impossible to meet the goals of the application. The goals of the proposed abstract representation are the following:

1. perform reverse engineering operations (e.g., building a class diagram of an application written in an object oriented programming language);
2. be able to evaluate the size of the program;
3. calculate software CK [31] metrics.

To achieve these goals our representation stores the following information:

1. class inheritance relations;
2. the attributes (i.e. instance and class variables) of each class;
3. all the method calls and attribute accesses;
4. the software metrics directly computable of each class/method (LOC, CC, Halstead Volume [62], etc.).

We decided to implement our representation by using the relational model; this lets us to use a relational DBMS for the data management. There are different meta-models for modeling source code: Dagstuhl middle model [82], Faamos meta model [43], JavaML[6], GXL [67], and Datrix [81]. We chose to derive our model from the Famoos [43] meta model, because it fits better into a relational schema. We have discarded the other meta-models because they are XML based, and fast data accessibility (that is difficult to achieve with XML documents) is requisite for being compatible with our framework. Figure 10 (on page 63) shows the E-R diagram of the relational representation. For clarity, we call tables the E-R entity because we already used the entity name for an entity.

In this schema, the two main tables are the *entity* table and the *attribute* table, then we have two other tables: *metrics* and *file*. Each tuple of the table *entity* represents an entity recognized by the application (i.e. a class, an interface, or a method). To distinguish the different types of entities, there is the mandatory field *type*, this field may assume the following values: 1) 'C', if the entity is either a class or a struct or an enumerative type; 'M', if the entity is a method or a function; 'I', if the entity is an interface.

The field *name* contains the fully qualified name (defined in part 10.8.2 of [26]) of the entity converted in the format defined by Succi *et al.* [117].

Each tuple of the table *attribute* represents either a field of a class or a field of a structure. The data field *name* contains the name of the attribute, and the data field *type* contains the fully qualified name (defined in part 10.8.2 of [26]) of the type of the attribute.

The relations *implements*, *interf_inher* and *class_inher* are used to keep track of the inheritance class tree. These relations are mandatory if we want to perform reverse engineering tasks using this tool, for this purpose other four relations are necessary,

nested relations keeps track of the inner class definitions. The relation *has_namespace* keeps track of the definitions inside the namespaces, and it is redundant because all types are fully qualified. The relations *has_attr* and *has_method* keep track of the attributes and methods ownership. The relations *has_file* knows where an entity is defined, today it is not trivial because the C# 2.0 ECMA specification introduces the partial type declarations (defined in part 8.7.13 of [26]). This feature allows the developer to split the definitions of a type in different files. For collecting product metrics, our representation uses the table *metric* for defining the metrics we collect: Lines of Code, McCabe Cyclomatic Complexity [88], Halstead Volume [62], Fan-in, Fan-out (see Table 7). These metrics are defined in Section 5.8.1 and 5.8.2 .

id	**name**	**description**
1	'LOC'	'Lines of code'
2	'C'	'McCabe Cyclomatic Complexity'
3	'V'	'Halstead Volume'
4	'FI'	'Fan-in'
5	'FO'	'Fan-out'

Table 7: Content of 'metric' table.

We state that this representation is adequate for modeling the source code written in every existing object-oriented and procedural language, we use this representation also for analyzing C, C++, Java, VB.NET source code.

After this relational meta-model is filled in, the data are extracted and other software metrics computed by the Source Code Analyzer (Figure 1). These software metrics include number of attributes (NOA), number of method calls for a method (NMC), number of invocations for a method (NMI), and CK metrics [31]. The split of the computations of metrics in two phases allows us to speed up the process of metrics extraction and to cross the project's border, making possible inter project analyses.

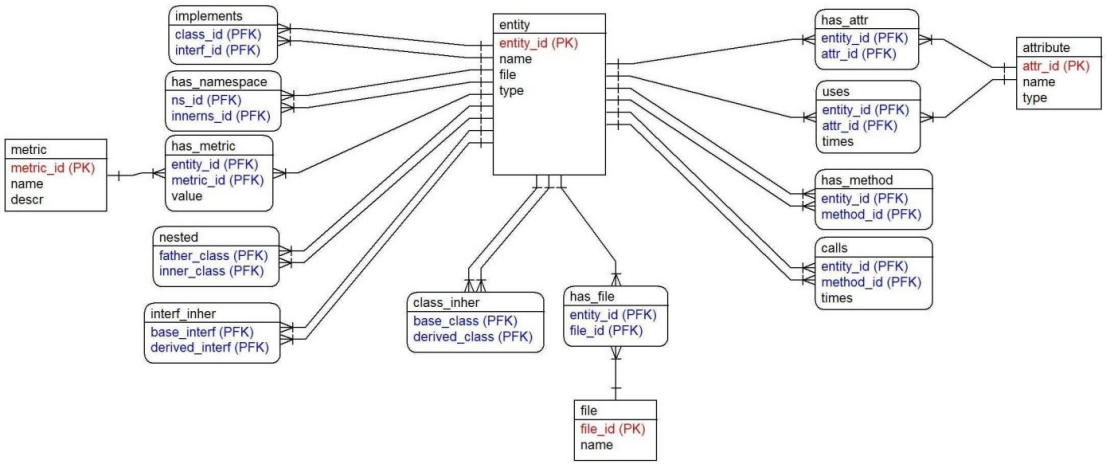

Figure 10: The code meta-model.

4.2.2 C# Metrics Extractor

The two main modules are the C# metrics extractor (*CSharpRelationGenerator*) and the preprocessor (*IfExpressionEvaluator*).

For the development of these two components, we have used MinosseCC[12]: an automatic LL(k) parser generator. This tool generates, as output, C# source files compatible with the .NET and MONO platforms. A very important feature of this tool is the capability to handle JavaCC grammar specifications [35]. JavaCC is a well-known tool for automatic generation of LL(k) parser written in Java language (also used for the SyQL front-end). MinosseCC allows us to use an existing grammar file written for JavaCC, so we have improved this grammar by adding all the language constructs present in C# 2.0. A list of added language constructs can be found here (http://msdn.microsoft.com/en-us/library/7cz8t42e(VS.80).aspx).

This metrics extractor can perform preprocessing and parsing at the same time, and this is possible because C# language specifications do not include macros like C or C++, so processing everything in one pass is possible. The three components involved in the preprocessing are the parser lexer, the preprocessor lexer and the preprocessor parser. They handle all the preprocessing directives defined in part 9.5 of [26]. When the parser lexical analyzer finds a preprocessing expression (defined in part 9.5.2 of [26]), it switches from the DEFAULT to the FIND_IF_EXPRESSION (see Figure 11) lexical states (defined in Definition 4), where it invokes the axiom, this method evaluates the preprocessing expressions (defined in part 9.5.2 of [26]). If the

[12]MinosseCC Parser Generator – Project Homepage – http://forge.novell.com/modules/xfmod/project/?minossecc

evaluation result is true it gives back the control to the parser lexer by switching back to the DEFAULT lexical state, otherwise, if the evaluation result is false, the parser lexer provides to skip the code block by switching the parser lexer from the DEFAULT to the IGNORE_IF lexical states. Figure 11 shows the other lexical state transitions that occur when the parser lexer encounters the other preprocessing directives.

After the parsing phase, we perform the visiting of the AST by using a visitor pattern [58], during this phase the tool fills in the tables of the relational representation.

Definition 4:	
Lexical State =	In JavaCC/MinosseCC, a lexical state is an ordered list of regular expressions that exclusively participate to the token matching phase. [35]

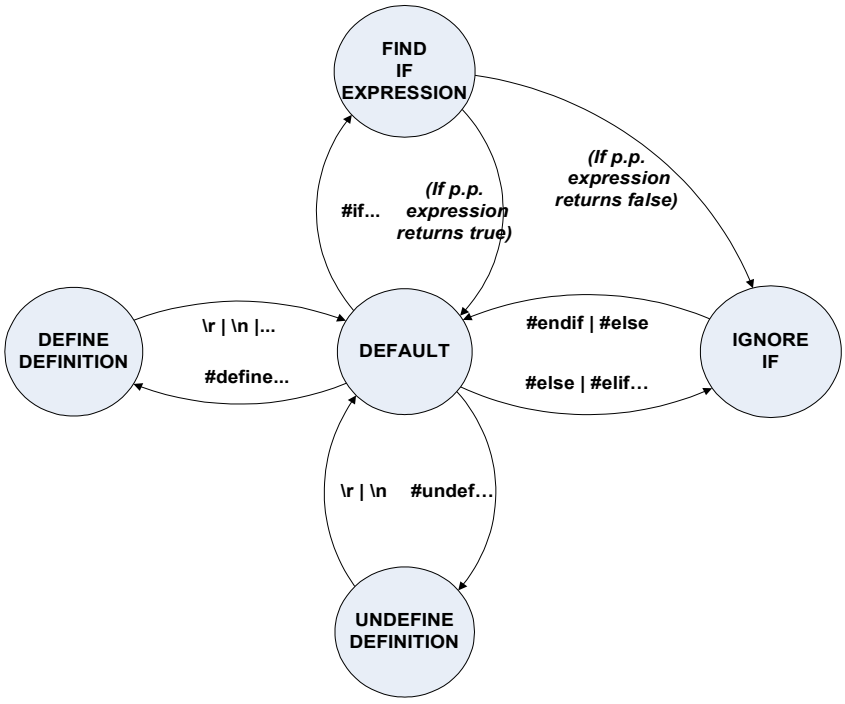

Figure 11: Lexical states of the C# parser lexer.

4.2.2.1 Preprocessor

The preprocessor is the component that evaluates preprocessing expressions, and it is implemented as a stack automaton. The grammar used by this expression evaluator is the following:

```
[1]  evaluate   ::=  or-expr
[2]  or-expr    ::=  and-expr ( "||" and-expr )*
[3]  and-expr   ::=  eq-expr ( "&&" eq-expr )*
[4]  eq-expr    ::=  unary ( ("==" | "!=") unary )*
[5]  unary      ::=  "!" unary
[6]                | primary
[7]  primary    ::=  "(" or-expr ")"
[8]                | <KEYWORD>
[9]                | <IDENTIFIER>
```

Listing 24: C# preprocessor production rules written in EBNF notation.

These production rules are a subset of rules defined in appendix A.1.10 of the ECMA-334 specification [26], because other rules are implemented in the parser lexer. This architecture allows the C# parser to work over the preprocessing process without taking care of it.

The development and testing of the preprocessor took two weeks/man starting from the C/C++ preprocessor developed by Succi *et al.* [118], which has been adapted according to the C# specification [26] (section 9.5.1).

4.2.2.2 Parser

The C# parser is the most complex component of the *promPM* suite. It implements all the production rules defined in part A.2 of the C# specification [26], and all the semantic of the production rules; we define as semantic all the actions taken to be able to perform correct code parsing, such as *Namespace and type names* resolution (part 10.8 of C# specification [26]), *Method invocations* (part 14.5.5.1 of C# specification [26]), etc. Then, to perform the calculation of the software metrics shown in Table 7 on page 62, we added extra semantic.

To achieve a fine grained of metrics measurement, we chose to implement all the production rules defined in the specifications, because the calculations of the McCabe

Cyclomatic Complexity, Halstead Volume, Fan-in and Fan-out metrics are not possible without having a "complete" parse tree of the source file. As complete parse tree, we intend to know all the fully qualified type names (defined in part 10.8.2 of C# specification [26]) of the used variables and where possible what methods are invoked. To achieve this result, it is not possible to implement the parser by using a light weight approach such as island grammar [44][91]. It became evident when we implemented part 10.8 of the specification [26], which defines the *Namespace and type names* resolution. To perform it correctly, the system must be able to know all the other defined classes inside the application under analysis and into the referenced assemblies. An assembly is a software library that contains a configured set of loadable code modules and other resources that together implement a unit of functionality (defined in part 6 of the Common Language Infrastructure specification [34]). To implement the resolution mechanism correctly, we load all the assemblies (application and referenced libraries) in reflection only context inside the application domain of the parser. An application domain is a mechanism "to isolate applications running in the same operating system process from one another. Types loaded into one application domain are distinct from the same type loaded into another application domain, and instances of objects shall not be directly shared from one application domain to another."[34] The reflection only context allows us to load cross compiled assemblies for another platform without being able to instantiate the types defined into them. This makes possible to load either an X64 assembly or an assembly compiled for an x86 machine that runs Windows CE.

For setting up the metrics acquisition, the parser automatically collects all the necessary information from the Visual Studio project files. Before starting the parsing, the source code is compiled by using NANT 0.85 (http://nant.sourceforge.net/) and MSBuild respectively for projects written with Visual Studio 1.1 and 2.0. Then, all the necessary assemblies file paths are collected and the assemblies loaded in reflection only context. Finally, the parsing starts and the metrics are stored into the relational meta-model.

4.2.3 Java Metrics Extractor

The Java source code metrics extractor has been implemented by us after the development of the C# parser. The C# experience helped us to implement the equivalent Java component quickly by discarding useless parts of the language. The resulting implementation is smaller and easier to maintain than the C# parser. Compared with C#, Java has no preprocessor, and this also makes simpler the overall architecture. For reducing the complexity, we have combined two components: a lightweight parser implemented by using the Island Grammar technique [44][91]; and the CKJM[13] java bytecode metrics extractor. The Island Grammar parser extracts the Line Of Code and the McCabe Cyclomatic Complexity [88] by parsing the source code, whereas the CKJM computes the remaining information needed to fill the code meta-model by parsing the Java bytecode.

The Island Grammars are not complete grammars that are able to produce a partial Abstract Syntax Tree of a program. “Not complete grammars” means that not all the syntactical and the lexical rules are implemented. The name Island Grammars comes exactly from that. The *Islands* are the implemented rules, and the *Water* (between the Islands) are the unimplemented ones.

To implement an Island Grammar, a developer can choose between two approaches: bottom-up, or top-down. In bottom-up, the developer starts from an “ocean of water”, hence he/she makes “emerge the Islands” by implementing the rules of interest. Contrariwise, in top-down, the developer starts from the Islands, after that he/she implements the water rules to make possible the parsing of source files.

Depending on the problem to solve, the developer has to choose the right approach. The bottom-up approach is useful to implement analyzers that look for specific patterns into the code (e.g. to check if all the pthread_mutex_lock(...) invocations have corresponding pthread_mutex_unlock(...) in order to avoid deadlocks).

The top-down is used when all the leafs of the AST are unnecessary (e.g. to count the number of Lines of Code of a method, the needed information are the following: the name-space, the defining class, the method signature. Whereas, the variables declarations and usages are unnecessary).

The Java parser used in this case study has been implemented by using the top-down

[13]CKJM – Chidamber and Kemerer Java Metrics – http://www.spinellis.gr/sw/ckjm/

approach. We have defined five distinct types of *Water*(s), which are implemented in five lexical states:

1. all the code after the package declaration (if any) to the type declarator (class, enum, interface, annotation) is discarded, and this is useful to get the package name discarding all the import declarations;
2. all the code before the semicolon is discarded. This is useful to reach the end of the statements, so it is possible to count the lines of code easily.
3. all the code inside round parentheses is discarded. This is useful to skip all the method calls and the selection statements conditions;
4. all the code inside curly parentheses is discarded. This is useful to skip the code inside the array initialization blocks;
5. all the code inside the angle parentheses is discarded. This is useful to discard type parameters, which do not contribute to the class signature.

These five *Water* lexical states allow us to implement a small subset of the Java grammar, mainly the method declarations and the selection statements (if, for, do, while, case). We want to remark that the McCabe Cyclomatic Complexity (MCC) computation process does not discard the conditional statements. Therefore, all the types of *Water*(s) define the question mark ('?') as exit character. In this way, it is possible to perform an exact computation of the MCC without implementing all the production rules of the expressions. Listing 25 shows all the types of *Water*(s) inside a Java code snippet. The uncolored parts of code are matched in the default lexical state. Names of packages, classes, and methods are collected in this state, because they allow to map the extracted metrics with the corresponding implementations inside the bytecode, which is analyzed by using the CKJM metrics extractor.

```
WATER_1      WATER_3      WATER_5
WATER_2      WATER_4
[1]  package it.unibz.prom.metrics.parsers.javaLineCounter.test;
[2]
[3]  import it.unibz.prom.metrics.parsers.javaLineCounter.parser.ast.Scope;
[4]  import java.io.File;
[5]  import java.io.Serializable;
[6]  import java.lang.reflect.Method;
[7]  import java.util.HashMap;
[8]  import java.util.LinkedList;
[9]  import java.util.Stack;
[10] import java.util.Vector;
[11] import org.junit.runner.RunWith;
[12] import org.junit.runners.Suite;
[13]
[14] class Test1_bis {
[15]    int a = 0;
[16]    void method1() {
[17]       a++;
[18]       if(a > 0 && a < 100)
[19]          a++;
[20]    }
[21] }
[22]
[23] @RunWith(Suite.class)
[24] @SuppressWarnings("serial")
[25] public class Test1 < M ,
[26]                      T extends LinkedList<Scope>,
[27]                      X extends Serializable,
[28]                      W extends Scope & Serializable>
[29]       extends LinkedList<Scope> implements Serializable, Scope {
[30]
[31]    Stack<Scope[]> stackScope = null;
[32]    int [] myArray = {0,1,2,3};
[33]    @SuppressWarnings({ "unused", "hiding" })
[34]    private <T> void testPrivateUnusedMethod(){
[35]       int i = (stackScope == null ? 0 : 1);
[36]    }
[37] }
```

Listing 25: Java snippet containing all the types of Waters.

We decided to implement this parser from scratch, because we want to have the control on the whole measurement process of Lines of Code and McCabe Cyclomatic

Complexity without having any bias coming from an external component, which may compute the metrics in a different way. The other information can be effectively collected from the bytecode without any bias.

4.3 Where SyQL is placed

SyQL is placed over the PROM data warehouse by fetching data only from this data source. According to the Three-Tier Architecture [46], we can state that SyQL resides in the Logic (Application) tier, where data are fetched from the Data tier, processed and sent to the Presentation tier. From the point of view of the System Administrator, SyQL is nothing more than an application that queries a predefined database. By using Java Database Connectivity API (JDBC), SyQL can fetch transparently data across the network. To send data to the Presentation tier an open implementation of the XML-RPC protocol is used. Therefore, clients cannot have any problem of network filtering, because the protocol uses HTTP as the transport and XML as the encoding.
In the next Chapters, we are going to describe SyQL in details by showing its benefits and its limitations.

5 System Query Language – SyQL

In this chapter, we are going to present and discuss the most important technical decisions taken during the design and the implementation process of SyQL. We start by providing a definition of SyQL. Then, we introduce possible point of views of the different actors involved in the software development process.

A general definition for SyQL can be the following: SyQL is a domain specific language that helps the non-experts to simplify the query writing process.

Starting from this definition, we can infer that SyQL aims to be applied to a wide range of scenarios, and not only to Software Engineering domain, in which SyQL may help users to benefit of the automatically collected data coming from the development process.

This level of generality can only be obtained by taking into account flexibility from the earlier phases of the design.

In more general terms, we can say that we try to address the problem of the Object-Relational Impedence Mismatch [98] by integrating the relational concepts into the languages. Indeed, SyQL can be integrated into a language like Java as described in Section 8.1 .

Potential SyQL users may pursue different objectives, which lead to solve different sets of tasks. In the Software Engineering domain, potential users may have to solve the problem listed below:

- a *manager* can use SyQL to check the actual status of all running projects;
- a *project manager* (PM) could use SyQL to show the effort spent by the developers on specific parts of a project. SyQL may also help to identify code parts that are deteriorating by using software metrics;
- a *developer* can be interested at using SyQL in reverse engineering tasks along the time line by showing the evolution of a specific project part;
- a *tester* can write a SyQL query that helps to identify the not very well tested parts of a project by showing test coverage data;
- a *researcher* can use SyQL in the data set preparation phase to speed up the entire analysis process.

The structure of this chapter is the following: In Section 5.1 , we present the syntax of

the language by putting in evidence its flexibility in expressing otherwise complex queries. In Section 5.2 , we introduce the data model on which SyQL is based. In Section 5.3 , we describe part of SyQL semantic by using natural language and second-order logic. In Section 5.4 , we present the general architecture of a query engine for evaluating SyQL queries. In Section 5.5 , we illustrate the execution of SyQL queries by using examples. In Section 5.6 , we discuss the linguistic variables supported by SyQL query engine. In Section 5.7 , we present the optimizations that we implemented after the initial sessions, in which performance has been evaluated. Finally, Section 5.8 concludes the chapter by presenting the currently available schema designed and implemented to retrieve data from the PROM database.

5.1 Language Structure

In this section, we present the most relevant parts of the SyQL syntax, whereas the complete BNF grammar [5] for SyQL non-terminal and terminal symbols is provided in Appendix A.

We start this description by introducing the structure of the language by an example. The query below (Listing 26) returns a collection of class names, the related effort spent by the developers since yesterday, and the number of methods for each class. This query uses two concepts: Class and Method. A SyQL concept is a self-defining entity that can expose some methods, which can be used into SyQL queries. A concept is the minimum part of a query: if a user wants to query some data, these data must be part of a concept. In Section 6.1 , we propose a methodology for the creation of new concepts.

```
[1]  FROM Class c, Method m
[2]  WHERE   c.getFullName() = m.getDefClassFullName()
[3]    AND   c.getEffort(YESTERDAY) IS High
[4]  SELECT  c.getFullName(),
[5]          c.getEffort(TODAY - 1 'day'),
[6]          COUNT(m)
[7]  GROUP BY c.getFullName(), c.getEffort(TODAY - 1 'day');
```

Listing 26: Sample SyQL Query.

The query shows the most important elements of the language that we are going to describe formally. The main production rules of the language are introduced by using the Extended-BNF (EBNF) formalism.

The lexical structure of SyQL is mostly similar to SQL. The axiom of the language is shown below:

```
[1]  Query_Node_Axiom ::=
[2]     FromClause WhereClause SelectClause ( GroupByClause )?
[3]     ( SetOperators FromClause WhereClause SelectClause
[4]     ( GroupByClause )? )* ( ";" )? <EOF>
```

The first row introduces the *FromClause* (see definition at line [5]), which could contain one or more *FromElement*(s). Each of them is composed by two literals: the former identifies the concept type, and the latter declares the concept name (like in SQL) also called range variable.

```
[5]  FromClause ::=
[6]     <FROM> FromElement FromElementName
[7]    ( "," FromElement FromElementName )*
[8]  FromElement ::=
[9]     Identifier
[10] FromElementName ::=
[11]    Identifier
```

The second and third rows introduce the *WhereClause* (see definition at line [12]). In the example, there are two conditions: an equal join condition and a fuzzy condition.

```
[12] WhereClause ::=
[13]    <WHERE> SyQLExpression
[14] SyQLExpression ::=
[15]    SyQLAndExpression ( <OR> SyQLAndExpression )*
[16] SyQLAndExpression ::=
[17]    SyQLUnaryLogicalExpression ( <AND> SyQLUnaryLogicalExpression )*
[18] SyQLUnaryLogicalExpression ::=
[19]    (  FuzzyExpression |
[20]       ( <NOT> )? ( SyQLRelationalExpression | "(" SyQLExpression ")" ) )
[21] ...
[22] SyQLSimpleExpression ::=
[23]    SyQLMultiplicativeExpression
[24]    ( ( "+" | "-" ) SyQLMultiplicativeExpression )*
[25] SyQLMultiplicativeExpression ::=
```

```
[26]    SyQLUnaryExpression ( ( "*" | "/" ) SyQLUnaryExpression )*
[27] ...
[28] SyQLTemporalExpression ::=
[29]    ( ( <TODAY> | <TOMORROW> | <YESTERDAY> ) ) ( SyQLTemporalPostfix )?
[30] ...
```

The fuzzy condition (line [3] of Listing 26) evaluates the effort spent yesterday by the developers. The method *c.getEffort(...)* is a Java method that returns an instance implementing the *IFuzzableType* interface (see the class diagram in Figure 22 on page 117). In the fourth, fifth, and sixth rows, the *SelectClause* is shown (see definition at line [39]). This is a non-empty collection of *MethodCall*(s) (definition at line [31]) and/or aggregation functions (definition at line [43]).

```
[31] MethodCall ::=
[32]    Identifier ( "." Identifier "(" ( Arguments )? ")" )*
[33] Arguments ::=
[34]    Argument ( "," Argument )*
[35] Argument ::=
[36]    SyQLPrimaryExpression
[37] Identifier ::=
[38]    <IDENTIFIER>
[39] SelectClause ::=
[40]    <SELECT> SelectElement ( "," SelectElement )*
[41] SelectElement ::=
[42]    SyQLSimpleExpression
[43] AggregationFunction ::=
[44]    ( <AVERAGE> | <MINIMUM> | <MAXIMUM> | <SUM> | <COUNT> | <MEDIAN> )
[45]   "(" SelectElement ")"
```

In the last row, we declare the *GroupByClause* (see definition at line [46]), which has the same meaning of the SQL *Group By*.

```
[46] GroupByClause ::=
[47]    <GROUP_BY> GroupByElement ( "," GroupByElement )*
[48] GroupByElement ::=
[49]    MethodCall | Identifier
```

We show part of the language structure by using the EBNF grammar formalism. In

Figure 12 (on page 77), the related Abstract Syntax Tree (AST) classes are shown. The AST produced by the parser is a custom parse tree, because we have implemented manually the logic of the tree creation.

From the definition of the *FromClause,* we can see that the language has no closure, hence it is impossible to reuse the result of a query like in SQL. SQL allows the user to reuse temporary relations generated during the query execution. We have not implemented the closure, because it would have added unnecessary complexity to the query engine during the development of the first release of the query engine. After the experimentation on a couple of case studies, we noticed that the closure can be useful to write more complex and detailed queries, which are necessary to conduct more deeper analysis. This limitation is discussed in details in section 8.2 , where a solution to this lack is also proposed.

If we compare the syntax of a SQL query with the corresponding one in SyQL, the most evident difference is the order of *SelectClause*, *FromClause*, and *WhereClause* (see line [1]). This is necessary because it enables us to perform code auto-completion. This choice has been adopted (for the same reason) also by two other query languages: .QL [92], and LINQ [89]. The "auto-completion/code completion" is a very important feature, Robbes *et al.* 2008 [105] and Murphy *et al.* 2006 [94] show its importance in a production environment. In their empirical study [94], Murphy *et al.* monitored 41 developers who use Eclipse as IDE. They found that the 6.7% of the number of executed commands are for auto-completion. The auto-completion came sixth among the top command executed.

Sometimes it is impossible to answer a particular question in a single query, so it is necessary to write more than one query. As in SQL, SyQL allows the user to make use of set operators. These operators are very important, because they enable the user to perform basic set operations. We have implemented the *UNION*, the *EXCEPT*, and the *INTERSECT* operators (see line [50]).

```
[50] SetOperators ::=
[51]    <UNION> | <EXCEPT> | <INTERSECT>
```

All these operators are already present in SQL, and we have kept the same meaning for them, in order to provide users with a smoother learning curve.

The same happens for the expressions. To avoid learning problems, the structure of

the expression is similar to the corresponding SQL expression, except for the presence of the *FuzzyExpression* non-terminal symbol (see line [52]).

```
[52] FuzzyExpression ::=
[53]    MethodCall <IS> FuzzyLabel
[54] FuzzyLabel ::=
[55]    Identifier
```

Fuzzy expressions allow the user to express filtering conditions by using the fuzzy equal operator (see line [53]) and linguistic variables (see line [54]). We can see the definition of the non-terminal *FuzzyLabel*: it is simply defined as an *Identifier* (see line [54]). We have done this choice to make possible defining new linguistic variables during the entire life cycle of SyQL (see section 5.6).

We want to emphasize that a *FromElement* (see line [8]) is also expanded as an *Identifier*. The reason is the same as before: to fulfill the extensibility requirement, we have used the java reflection to load the concept type at runtime. Therefore, at parse-time the *FromElement* is parsed as an *Identifier*.

To avoid capitalization problems during the query writing process, the terminal symbols defined by us are case insensitive (e.g. '**from**', '**From**', '**fRom**', and '**FROM**' are all matched as <*FROM*> token). This makes easier to write queries.

Implementing all these lexical and syntactical rules in a single stack automaton has been possible with a limited effort by using an automatic parser generator that accepts the EBNF grammar format as input. To implement the SyQL front-end (parser), we have used the Java Compiler Compiler (JavaCC) parser generator [35]. The JavaCC allows the developer to easily generate the java code of the LL(k) [1] stack automaton. We have chosen this parser generator because LL grammars are more intuitive than LR grammars.

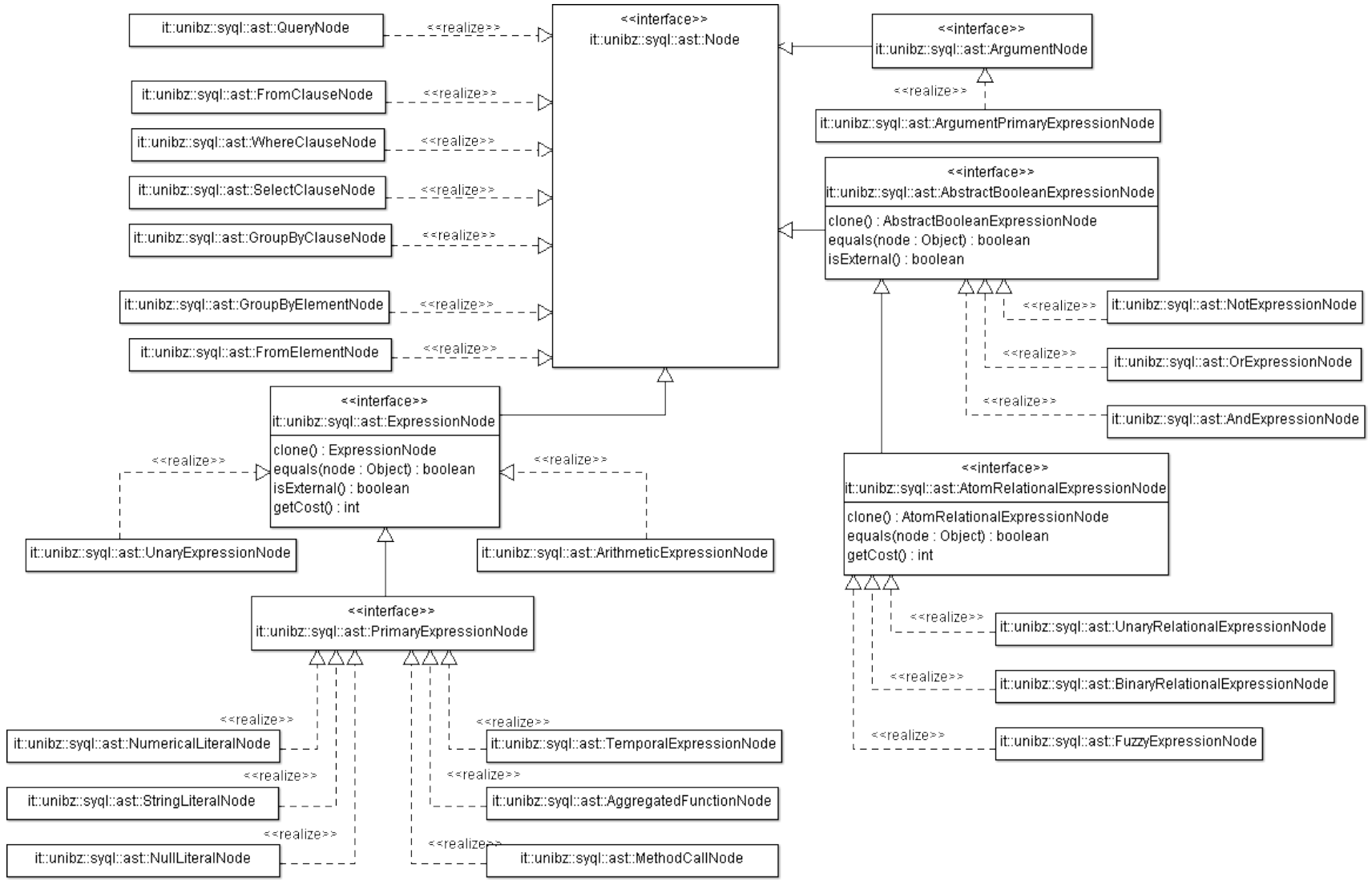

Figure 12: Class Diagram of the AST classes (partial).

5.2 Data Model

In this section, we are going to introduce the data model on which SyQL is based. In order to make it easier to understand, we define it by way of natural language, and then we show a sample schema based on this data model.

The data model is based on object types, classes, concepts, methods and values entities (see Figure 13 below), which can be used through SyQL query language.

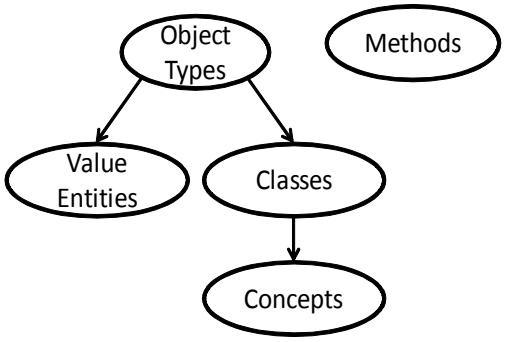

Figure 13: Hierarchy of Data Model entities.

Therefore, a schema based on these entities can be easily implemented by using an object-oriented programming language like Java. By doing this, the schema can be also executed in an object-oriented runtime like a Java Virtual Machine.

In the following, we show a sample schema based on this data model:

- **Class**{ getEffort(start, end) ClassEffort,
  hasBeenModified(start, end) boolean,
  getFullName() string, … }
- **ClassEffort**{ isSpentBy(string) boolean,
  getValue() int, … }
- **ClosedBug** { getDescription() string,
  getModifiedClasses() Class[], … }
- **Method** { getDefiningClass() Class, … }

It is important to say that the object types contain the classes, and the classes contain concepts. The concepts are responsible for the mapping between the above-defined schema and the AISEMA relational tables. In the defined schema, the object types *Class*, *ClosedBug*, and *Method* are concepts, whereas the *ClassEffort* type is a class, which makes possible the manipulation of measures that require a certain context to

be correctly interpreted just like the effort. *ClassEffort* represents the effort spent on a specific *Class* by one or more users

In order manipulate the measures, both classes and concepts contain the methods that provide the necessary logic for that. By way of example, the method *isSpentBy(string)* defined in *ClassEffort* class makes possible to determine who have spent effort on a particular instance of *Class* concept. In the same way, the method *getValue()* also defined in *ClassEffort* class returns an integer value, which is a value entity, that represents the effort in seconds spent by a developer on a particular instance of *Class* concept.

We want to underline that we have also defined the values entities as object types, in order to simplify the implementation by using a programming language like Java. Indeed, Java type-system defines numbers as objects like in SyQL by making the implementation of this part of the SyQL data model straightforward.

Both classes and concepts are extension instances of SyQL, whereas the methods are functions. In Section 5.3 , we are going to describe the semantic of these two elements available in SyQL.

An instance of concept *Class* or *Method* represents a class or a method of a generic software project, and an instance of object type *ClosedBug* represents a bug reported in a bug tracking system that is now resolved.

By using the data-model defined above, the PM can write the following query:

```
[1]  FROM    Class c
[2]  WHERE   c.getEffort(TODAY - 1'week',TODAY).isSpentBy("Mario Rossi")
[3]    AND   c.getEffort(TODAY - 1'week',TODAY).isSpentBy("John Doe")
[4]    AND   c.hasBeenModified(TODAY - 1 'week', TODAY)
[5]  SELECT  c.getFullName();
```

Listing 27: SyQL query that selects cross-modified classes.

The query shown in Listing 27 returns a result that is a list of strings containing the class names. In line 1, the range variable *c* is declared as a set of instances of concept *Class* representing the classes constituting the software project. In lines 2-3, the range variable *c* is used for getting an instance of object type *ClassEffort* for each class through the *getEffort(…,…)* method by specifying the last week time interval. The *ClassEffort* instance contains the data about the effort spent on a specific class in the

last week, so that it is possible to retrieve the name of the developer who had spent this effort by using the method *isSpentBy(String)*. In line 4, it is specified a condition for checking if the class of the generic software project has been modified in the last week. Finally, in line 5, the *getFullName()* method is invoked by printing out all the names of the classes that satisfy the criteria specified in lines 2-4 described above.
In the next section, we describe the semantic of SyQL queries by also using logical description.

5.3 Language Semantics

In this section, we are going to describe the semantic of the SyQL language by using natural language and logic. This logical description is focused on the most relevant entities coming out from the processing steps undertaken during the query processing of a generic SyQL query, i.e. we start from a generic relational instance, and we conclude by describing how the final query result set is computed.
In order to explain how the SyQL engine works, we will follow the description paths as shown in Figure 14.

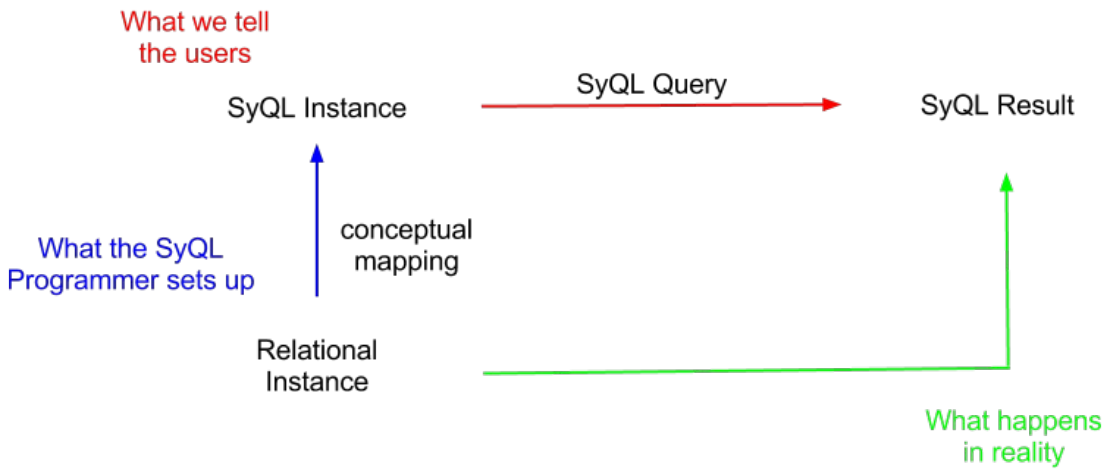

Figure 14: Description Path.

Firstly, we describe the blue path, in order to show how a developer can set up the conceptual mapping between the DBMS and the object oriented environment by defining properly the extension instances; after that we describe the red path, in which all the processing steps can be easily understood by the user, and then we describe how the implementation computes the SyQL result by following the green path.

The “Relational Instance” corner represents the DBMS, in which the data are stored. These data are mapped (“conceptual mapping” arrow) into a SyQL schema instance (“SyQL Instance” corner), which is based on a meta model defined by a SyQL developer in a way that we are going to explain. This schema provides a set of objects for every type. Finally, the “SyQL Query” arrow represents a well-formed query that produces results (“SyQL result” corner).

Now we are going to define the different elements that either a developer or a user have to specify, in order to enable the SyQL query engine to process the data contained into a generic DBMS.

In Formula 2, we can see that the SyQL schema comprises a set of *AvailableConcepts*, which contains *C* different concepts declarations ($ConceptDecl_j$) like *Class*, *ClassEffort*, *ClosedBug*, and *Method*. These concepts declarations are comprised in the extension instances defined by the developer. The concepts declarations can be arbitrary created, in order to extend the number of entities that can be queried by the users. We want to say that the design process of the extension instances is strictly related to the underlying database schema (see Section 6.1), whereas the implementation process is similar to the one adopted for a generic object-oriented software.

$$AvailableConcepts ::= \{ConceptDecl_1, \ldots, ConceptDecl_C\}$$

where

$$C ::= \text{Total number of concepts definitions available}$$

Formula 2: SyQL concepts.

By referring to Formula 3, the *j-th* concept declaration comprises a set of methods $f_{1,j}(AvailableConcepts), \ldots, f_{Q_j,j}(AvailableConcepts)$ like *getEffort(...)*, *hasBeenModified(...)*, etc. These methods allow to enhance usability of SyQL by implementing the necessary logic to manipulate data. We want to underline that no restrictions are present in the definition of a method. This makes possible to manipulate data in a very effective way without limitations, since well-known object-oriented methodologies can be applied during the design and the implementation of the methods. In particular, methods can return class instances by modeling many-to-many relationship in a very easy way. In the next subsection, some implementations

of these methods are described in the details.

$$ConceptDecl_j ::= \{ f_{1,j}(AvailableConcepts), \ldots, f_{Q_j,j}(AvailableConcepts) \}$$

where

$Q_j ::=$ Total number of attributes of the j^{th} concept
$f_{x,j}(AvailableConcepts) = x^{th}$ function of the j^{th} *concept*

Formula 3: Concepts definitions.

Now we are going to introduce the elements that the user has to specify, in order to execute a SyQL query.

As shown in Formula 4, a generic SyQL query (*SyQLQuery*) comprises a number of concepts instances $(Concept_1, \ldots, Concept_F)$ specified in the *FromClause* equals to *F*, a number of conditions $(Condition_1, \ldots, Condition_W)$ specified in the *WhereClause* equals to *W,* a number of selected attributes $(Select_1, \ldots, Select_S)$ specified in the *SelectClause* equals to *S*, and a number of grouping attributes $(Group_1, \ldots, Group_G)$ specified in the *GroupByClause* equals to *G*.

In the SyQL query (*SyQLQuery*), the concepts instances $(Concept_1, \ldots, Concept_F)$ must be defined in the *AvailableConcepts* set, the *l-th* condition $(Condition_l)$ is a Boolean function taking the concepts instantiations $(Concept_1, \ldots, Concept_F)$ as arguments, the *k-th* selected attribute $(Select_k)$ is a generic function taking $(Concept_1, \ldots, Concept_F)$ as arguments, and the grouping attributes set $\{Group_1, \ldots, Group_G\}$ is a subset of the selected attributes set $\{Select_1, \ldots, Select_S\}$.

$$
\begin{aligned}
&SyQLQuery ::= \{ \\
&\quad Concept_1, \ldots, Concept_F, \\
&\quad Condition_1, \ldots, Condition_W, \\
&\quad Select_1, \ldots, Select_S, \\
&\quad Group_1, \ldots, Group_G \quad | \\
&\quad \forall m \in \mathbb{Z}^+ (m \leq F \rightarrow Concept_m \in AvailableConcepts) \\
&\quad \wedge \\
&\quad \forall l \in \mathbb{Z}^+ (l \leq W \quad \rightarrow \\
&\qquad\qquad \exists ! x \in \mathbb{Z}^+ (x \leq BF \rightarrow \\
&\qquad\qquad Condition_l = bf_x (Concept_1, \ldots, Concept_F) \\
&\quad \wedge \\
&\quad \forall k \in \mathbb{Z}^+ (k \leq S \quad \rightarrow \\
&\qquad\qquad \exists ! y \in \mathbb{Z}^+ (y \leq GF \rightarrow \\
&\qquad\qquad Select_k = gf_y (Concept_1, \ldots, Concept_F) \\
&\quad \wedge \\
&\quad G > 0 \rightarrow \forall x \in \{Group_1, \ldots, Group_G\} (x \subset \{Select_1, \ldots, Select_S\}) \; \}
\end{aligned}
$$

where

$F ::=$ Total number of range variables declared in the From clause of the SyQL Query
$W ::=$ Total number of conditions specified in the Where clause of the SyQL Query
$S ::=$ Total number of selected methods specified in the Select clause of the SyQL Query
$G ::=$ Total number of grop attributes specified in the Group By clause of the SyQL Query
$GF ::=$ Total number of generic functions
$gf_i(\ldots) ::= i^{th}$ generic function
$BF ::=$ Total number of boolean functions
$bf_j(\ldots) ::= j^{th}$ generic boolean function

Formula 4: SyQL query.

Formula 5 defines how the intermediate result of the SyQL query (*SyQLQuery*) is computed: if the number of concept's instances (*F*) is greater than 1, the intermediate result (*SyQLIntermResult*) of the SyQL query (*SyQLQuery*) is a Cartesian product between the concept's instantiations which satisfy the conditions $(Condition_1(\ldots), \ldots, Condition_W(\ldots))$, otherwise (*F = 1*) the intermediate result (*SyQLIntermResult*) of the SyQL query contains the instances of the concept instantiation that satisfy said conditions.

$$
\begin{aligned}
&SyQLIntermResult(SyQLQuery) ::= \\
&\quad \langle ir_1, \ldots, ir_F \rangle | \\
&\quad F = 1 \rightarrow \langle ir_1, \ldots, ir_F \rangle \in Concept_1 \\
&\quad \wedge \\
&\quad F > 1 \rightarrow \langle ir_1, \ldots, ir_F \rangle \in Concept_1 \times \ldots \times Concept_F \\
&\quad \wedge \\
&\quad \forall k \in \mathbb{Z}^+ (k \leq W \rightarrow Condition_k (ir_1, \ldots, ir_F))
\end{aligned}
$$

Formula 5: Intermediate result.

Finally, the computation of the final result $(SyQLQueryResult(SyQLQuery))$ of the SyQL query is shown in Formula 6. This final result is a relation having a number of columns equals to the number of the selected attributes *S*, and a number of records equals to the number of records of the intermediate result.

$$
\begin{aligned}
&SyQLQueryResult(SyQLQuery) ::= \langle so_1, \ldots, so_S \rangle \mid \\
&\quad S = 1 \rightarrow \langle so_1, \ldots, so_S \rangle = \quad Select_1(SyQLIntermResult(SyQLQuery)) \\
&\quad \wedge \\
&\quad S > 1 \rightarrow \langle so_1, \ldots, so_S \rangle = \quad Select_1(SyQLIntermResult(SyQLQuery)) \\
&\qquad\qquad\qquad\qquad\quad \cup \ldots \cup \\
&\qquad\qquad\qquad\qquad\quad Select_S(SyQLIntermResult(SyQLQuery))
\end{aligned}
$$

Formula 6: Final result.

In the next subsection, we are going to show by way of example how translate a relational database schema into a SyQL schema like the one shown at the beginning of Section 5.2 . This process can be carried on by means of a generic methodology, which can be applied to a generic database schema.

5.3.1 Mapping of a relational database schema into a SyQL schema

A generic relational database schema is solely based on two types of elements: the tables and the constraints. These elements are typically generated from an entity-relationship schema, which is based on a larger number of elements, which are the entity, the relations and the cardinalities. It is well-known that it is always possible to convert an entity-relationship schema into a relational one, whereas the opposite process requires a reverse engineering step that a developer can usually perform without difficulties. Therefore, we can assume that the two types of schemas are interchangeable.

An object-oriented schema like the one used by SyQL is richer in terms of defining elements, since there are classes and methods, which considerably extend the possibilities to model the reality by using a computerized representation. This augmented expressiveness of the object-oriented schema makes the mapping process against a relational database, and viceversa, nontrivial.

Now we are going to describe a methodology for mapping an entity-relationship

schema (or a relational schema) into an object-oriented schema, which can be queried by using SyQL.

This methodology comprises the following steps:

a) defining at least one concept for each entity of the entity-relationship schema;
b) defining at least one class for each relationship of the entity-relationship schema;
c) defining for the concepts of interest at least one method for instantiating one or more classes defined at step (b) and/or one or more concepts defined at step (a);
d) defining methods of interest, e.g. accessing attributes of the entity/relationship, implementing business rules, analysis techniques, etc.

Of course the above-mentioned methodology can be restricted only to part of the entity-relationship schema, which is particularly interesting for the user.

In order to show how the above-defined methodology has to be applied on a real relational schema, we are going to describe a guiding example by using a small part of the PROM database schema.

We will start from the relational schema shown below, and we are going to show all the necessary steps to get to an object-oriented schema similar to the one shown at the beginning of Section 5.2 .

The relational schema of the DBMS is the following:

entity(ID, methodName, className, namespaceName, type)
metric(ID, name)
users(ID, firstname, surname)
bug(ID, description, openTime, closeTime)
entityProductMetrics(entityID, metricID, dateTime, value)
entityEffort(entityID, userID, startTime, duration)

The *entity* table contains all the information concerning software artifacts like methods and classes, the *metric* table stores all the information about the collected metrics (e.g. LOC, Effort, etc.). The *users* table contains personal data of the developers monitored by PROM system, and the *bug* table stores all the information collected by PROM system from the bug tracking system.

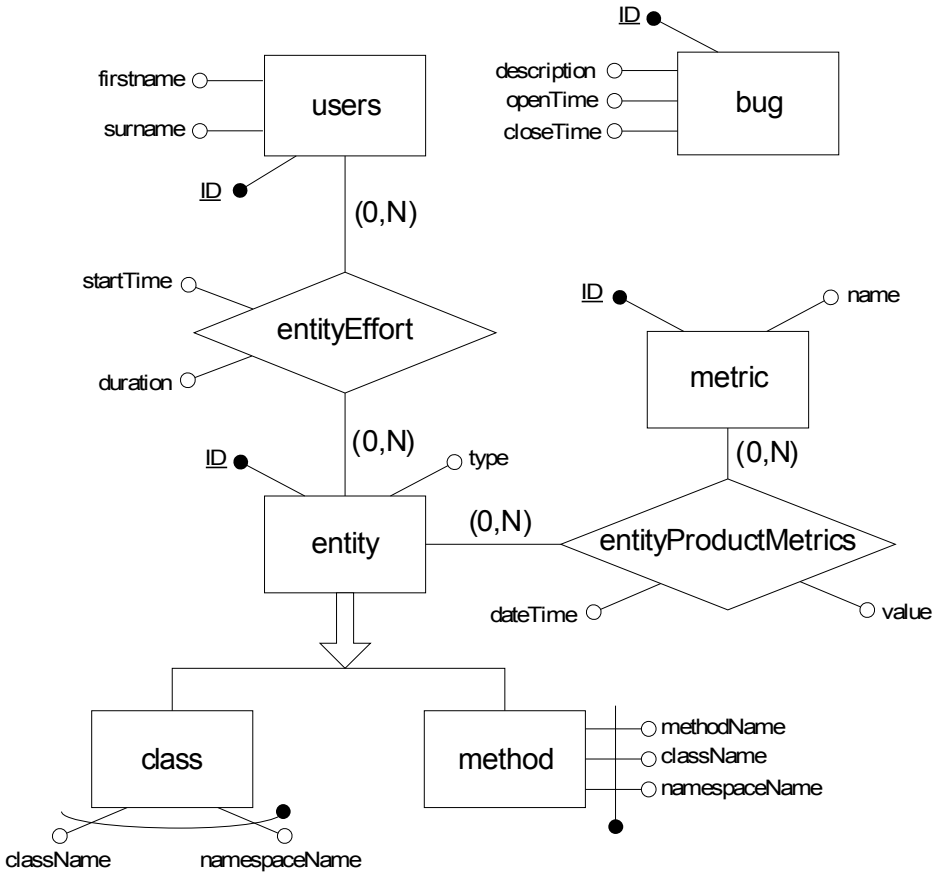

Figure 15: Entity-Relationship diagram of part of the PROM database.

We want to underline that the *type* attribute of the *entity* entity implements the partial generalization that involves *class* and *method* entities. In the relational schema, the attribute *type* is a char field that is equal to 'C' when the record is a class, to 'M' when the record is a method, and to null when the record is neither a class nor a method.

In order to perform the first step (a) of the methodology, we can see in Figure 15 that the (real) entities are the following: *class*, *method*, *users*, *metric* and *bug*.

For each of the entities defined in the entity-relationship schema is possible to define at least one concept by specifying an SQL query able to return the information associated to the specific entity, i.e. the query should map the tables of the relational schema, in a way that the query results have the same schema of the specific entity. Now we are going to explain this step by way of examples.

For the entity class, we can define a concept called *Class* by specifying the following query:

```
[1]  SELECT ID, className, namespaceName
[2]  FROM  entity AS e
[3]  WHERE e.type = 'C';
```

Listing 28: SQL query that defines *Class* concept.

The execution of this query returns all the information necessary to create the *class* entity: it filters out all the unnecessary records by a filtering condition (see line 3 of the above query), and it selects only the columns that define the key of the *class* entity.

The same happens for the *Method* concept (see Listing 29 below).

```
[1]  SELECT ID, methodName, className, namespaceName
[2]  FROM  entity AS e
[3]  WHERE e.type = 'M';
```

Listing 29: SQL query that defines *Method* concept.

For the remaining three entities (*users*, *bug* and *metric*), we also report the SQL queries necessary to define the *User*, *ClosedBug* and *Metric* concepts.

```
[1]  SELECT ID, firstname, surname
[2]  FROM  users;
```

Listing 30: SQL query that defines *User* concept.

```
[1]  SELECT  ID, name
[2]  FROM  metric;
```

Listing 31: SQL query that defines *Metric* concept.

```
[1]  SELECT  ID, description, openTime, closeTime
[2]  FROM  bug AS b
[3]  WHERE b.closeTime IS NOT NULL;
```

Listing 32: SQL query that defines *ClosedBug* concept.

We want to underline that the *ClosedBug* concept relates only to the bugs that are already closed, and not to all the bug records contained in the *bug* table. For this reason, we have specified the filtering condition in the where-clause of the above-defined query.

In the following of this Chapter and in the next one, we are going to explain how to implement code runnable by a generic SyQL query by using these queries.
In order to perform the second step (b) of the methodology, we can see in Figure 15 that the relationships specified are the following:

- *entityProductMetrics*, which is a many-to-many relationship that links *entity* and *metric* entities;
- *entityEffort*, which is a many-to-many relationship that links *entity* (*class* and *method*) and *users* tables.

For the *entityProductMetrics* relationship, we have decided to create one class called *ClassLOC*, which models the value of the Line of Code metric of a Class in a specific moment; for implementing this class, we have to define a constructor that takes, as input, a *Class* concept instance, a date-time, and a value of the LOC metric.
Conversely, for the *entityEffort* relationship, we have decided to create two classes: *ClassEffort* and *MethodEffort*. These classes model, respectively, an effort of a certain duration spent, in a particular moment, by an user on a particular class or method. The constructors of both classes take, as input, an instance of *Class/Method* concept, a duration and the start time of the effort, and optionally an instance of *User* concept.
The third step (c) of the methodology requires to define the methods that instantiate the classes and/or the concepts defined during the first two steps. Since the implementation issues are treated in detail in Chapter 6 , we are going to show only the queries that are executed by the methods, in order to retrieve the data necessary for the computation of the result. The design methodology of these method cannot be formalized, since said methods have to be defined according to the requirements that the object-oriented schema should fulfill. The methods implemented by us have been designed to fulfill the requirements coming from the PROM usage experience.
Both in *Class* and *Method* concepts, we have defined the *getEffort(start_time, end_time)* methods. These methods compute the effort spent by the developers, in the interval specified by the arguments, on the current class or method by running the query shown below (see Listing 33).

```
[1]   SELECT  SUM(ee.duration)
[2]   FROM    entityEffort ee
[3]   WHERE   ee.entityID = <ID> AND
[4]           ee.startTime <= <END> AND
[5]           ee.startTime + duration >= <START>;
```

Listing 33: SQL query executed by the *getEffort(...,...)* method.

This query returns an integer number, which is used to create the proper *ClassEffort* or *MethodEffort* instance. The ID parameter is provided by the *Class* or *Method* instance on which this method has been invoked, whereas END and START parameters come from the arguments specified by the SyQL user.

In the *ClosedBug* concept, we have defined the *getModifiedClasses()* method, which returns an array of *Class* instances that represents the classes modified during the bug fixing process by the developers. In order to compute this array of *Class* instances, this method executes the query shown in Listing 34.

```
[1]   SELECT DISTINCT e.ID, e.className, e.namespaceName
[2]   FROM    entity e, bug b, entityEffort ee
[3]   WHERE   b.closeTime IS NOT NULL AND
[4]           b.ID = <BUG_ID> AND
[5]           e.type = 'C' AND
[6]           e.ID = ee.entityID AND
[7]           ee.startTime >= b.openTime AND
[8]           ee.startTime + ee.duration <= b.closeTime;
```

Listing 34: SQL query executed by the *getModifiedClasses()* method.

This query tries to map the classes (see the condition at line [5]) and the bugs by comparing the effort time-frame with the opening/closing date time of a bug (see lines [6]-[8]). We want to stress that this is a possible strategy to map bugs with code-elements. However, it is not the only possible way to do it. For this reason, this method is not presented into the PROM library presented in Section 5.8 .

In order to perform the fourth step (d) of the methodology, we have implemented the methods that we are going to describe below.

In the *Class* and *Method* concepts, we have implemented the *hasBeenModified(start_time, end_time)* method. The method returns the Boolean value true if the class or the method has been modified in the interval specified by the

arguments, otherwise it returns false.

In order to to compute the method's result, the method execute the query shown in Listing 35 below.

```
[1]   SELECT  *
[2]   FROM    entityProductMetrics epmi, entityProductMetrics epml
[3]   WHERE   epmi.value <> epml.value AND
[4]           epmi.entityID = epml.entityID AND
[5]           epml.entityID = <ID> AND
[6]           epml.metricID = epmi.metricID AND
[7]           epml.date = (SELECT  MAX(epmp.date)
[8]                        FROM    entityProductMetrics epmp
[9]                        WHERE   epmp.date <= <END_DATE> AND
[10]                               epmp.metricID = epmi.metricID) AND
[11]          epmi.date >= <START_DATE> AND
[12]          epmi.date < epml.date;
```

Listing 35: SQL query executed by the *hasBeenModified(...,...)* method.

If the result set of this query contains one or more records, the class or the method has been modified, otherwise it is not.

The query shown in Listing 36 is executed by the *isSpentBy(User)* method of the *ClassEffort* and *MethodEffort* concepts. We want to stress that the parameters like *ID, END* and *START* have the same meaning of the *ClassEffort*'s ones, since a *Class* instance creates a *ClassEffort* instance through *getEffort(...,...)* method.

```
[1]   SELECT  DISTINCT u.ID, u.firstname, u.surname
[2]   FROM    entityEffort ee, users u
[3]   WHERE   ee.userID = u.ID
[4]           ee.entityID = <CLASS_ID> AND
[5]           ee.startTime <= <END> AND
[6]           ee.startTime + duration >= <START>;
```

Listing 36: SQL query executed by the *isSpentBy(...)* method.

Each instance of the *ClassEffort* class can use the resulting relation of this query for determining the ownership of this effort spent on a specific class.

It is well-know that a method can be defined inside a class, for this purpose a possible way to retrieve information about the defining class of a method is shown in Listing 37. The *getDefiningClass()* method of the *Method* concept executes the below query,

in order to retrieve the necessary information for instantiating a new *Class* instance.

```
[1]   SELECT  e.ID, e.className, e.namespaceName
[2]   FROM    entity e, entity mthd
[3]   WHERE   e.type = 'C' AND
[4]           mthd.ID = <ID> AND
[5]           e.className = mthd.className AND
[6]           e.namespaceName = mthd.namespaceName;
```

Listing 37: SQL query executed by the *getDefiningClass()* method.

The above SQL query returns the necessary data to create a *Class* instance by providing the ID of the current *Method* instance.

In the following sections, we are going to explain how we have implemented the aspects, which has been introduced in this section, in an Object Oriented environment like Java. In particular, we describe how the query results are computed along the green path shown in Figure 14 on page 80.

5.4 Architecture

In this section, we present the internal architecture of the SyQL query engine.

Previously, we showed how we have designed the language to be flexible. Here, we illustrate how the internal structure is designed to achieve this goal.

Figure 16 shows the general scenario in which SyQL operates: the SyQL query engine sits on top of the AISEMA DBMS, and runs on a Java Virtual Machine (JVM).

We have decided to keep the Data and the Database Management System (DBMS) decoupled from the SyQL Engine by using the Java Database Connectivity API (JDBC). The Java Database Connectivity (JDBC) API is the industry standard for database-independent connectivity between the Java programming language and a wide range of databases.[14] Therefore, we can run the SyQL query engine on the top of different DBMS(s). During the testing, we have used SyQL on the top of PostgreSQL relational databases.

[14] http://java.sun.com/javase/technologies/database/

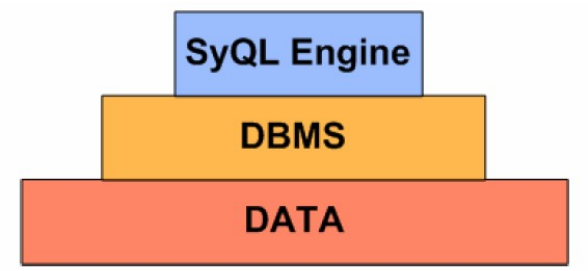

Figure 16: The Data Layers.

SyQL uses a data model consisting of classes and methods, which are contained in a SyQL library. In this domain (Software Engineering domain), the SyQL library contains the proper classes and methods for providing a clear view of the software development process by mapping the data from the PROM database.

This architecture makes possible to implement a query engine without developing a sophisticated query planner/executor and to develop methods that cannot be evaluated inside the database. Indeed, we say a method is external if can be evaluated both inside the database and inside the JVM, and a method is internal if can be evaluated only inside the JVM.

The main idea at the base of this implementation was to push as many conditions as possible into the query engine of the underlying DBMS. In this way, the implementation of a query planner and executor becomes very simple and compact.

A subset of the classes contained into the SyQL library are defined as concepts. A SyQL concept (previously defined as $ConceptDecl_j$) is a self-defining entity that can expose some methods, which can be used in SyQL queries. Each concept is characterized by comprehending a defining SQL query (previously defined as $SQLQuery_j(d)$). The relation defined by this SQL query contains the records from which the concept instances are created.

In a SyQL query, a concept is the minimum part of the query, i.e. a user can query some data if and only if these data are part of a concept.

In the current implementation, a concept is a Java class comprehending all the method definitions that can be called during the execution of a SyQL query.

These classes contained in the SyQL library are regular Java classes: they can reference external libraries; they can perform I/O operations; they can use the Java Native Interface (JNI), etc. Concept classes are loaded and instantiated by using Java

Reflection. We can then implement a new concept class without modifying the classes of the SyQL query engine. In Section 6.1 , we describe how to implement new concepts from their definition to their implementation.

Now we are going to describe the basic steps performed by the SyQL query engine to map the condition from the SyQL query to the relational DBMS.

The main step for converting the conditions present in the *Where-Clause* is the conversion of the *SyQLExpression* in Conjunctive Normal Form (CNF) by using Double negative elimination, distributive property of AND, and De Morgan's laws.

The CNF notation is very helpful, because a clause (conjuncts) of the CNF formula can be processed by the underlying DBMS only if all the conditions (literals) inside the clause are evaluated as external, otherwise the clause must be evaluated by the SyQL query engine. A condition is evaluated as an external one if all the predicates (the method calls) of the condition are external, otherwise a condition is evaluated internally. If all the conditions of a query are evaluated as external, we have a complete translation of the SyQL query into a SQL query with no specific engine processing.

For this purpose, all the methods of a concept class that can appear in the SyQL query are annotated in two different ways: external or internal. We say a method is "external" if the returned value is present in one column of the defining SQL query, otherwise, it is "internal". If a method is "external", it is necessary to specify in the annotation the field name that identifies the column of the defining SQL query.

In the current implementation, the method of the concept classes are invoked through Java reflection. The mapping between a *MethodCall* (see line [31] of the grammar definition) and its implementation is performed by Java Multi Method Framework (JMMF) [53]. This library provides a method resolution algorithm that implements the multiple-polymorphism. Multiple-polymorphism is the essence of the Object Oriented polymorphism, which is one of the most important features of the Object Oriented paradigm. In Java, all the method calls use late binding, unless a method has been declared final. The binding between a method call and method body is performed by using the types of the passed arguments, and everything happens at run-time. Unfortunately, the Java reflection implementation does not provide multiple-polymorphism natively.

Method binding is the beginning of the query execution process, Figure 17 illustrates the different modules involved while processing SyQL queries. These modules implement the visitor pattern [58], which is particularly effective for parse tree manipulation. We have preferred to implement separate visitors instead a single one to avoid unnecessary complexity. This provides a better separation of concerns between the tasks of the modules without affecting the performances: the size of parsing trees is usually limited since SyQL is conceived to write short queries. The details of the query execution are described in Section 5.5.2 .

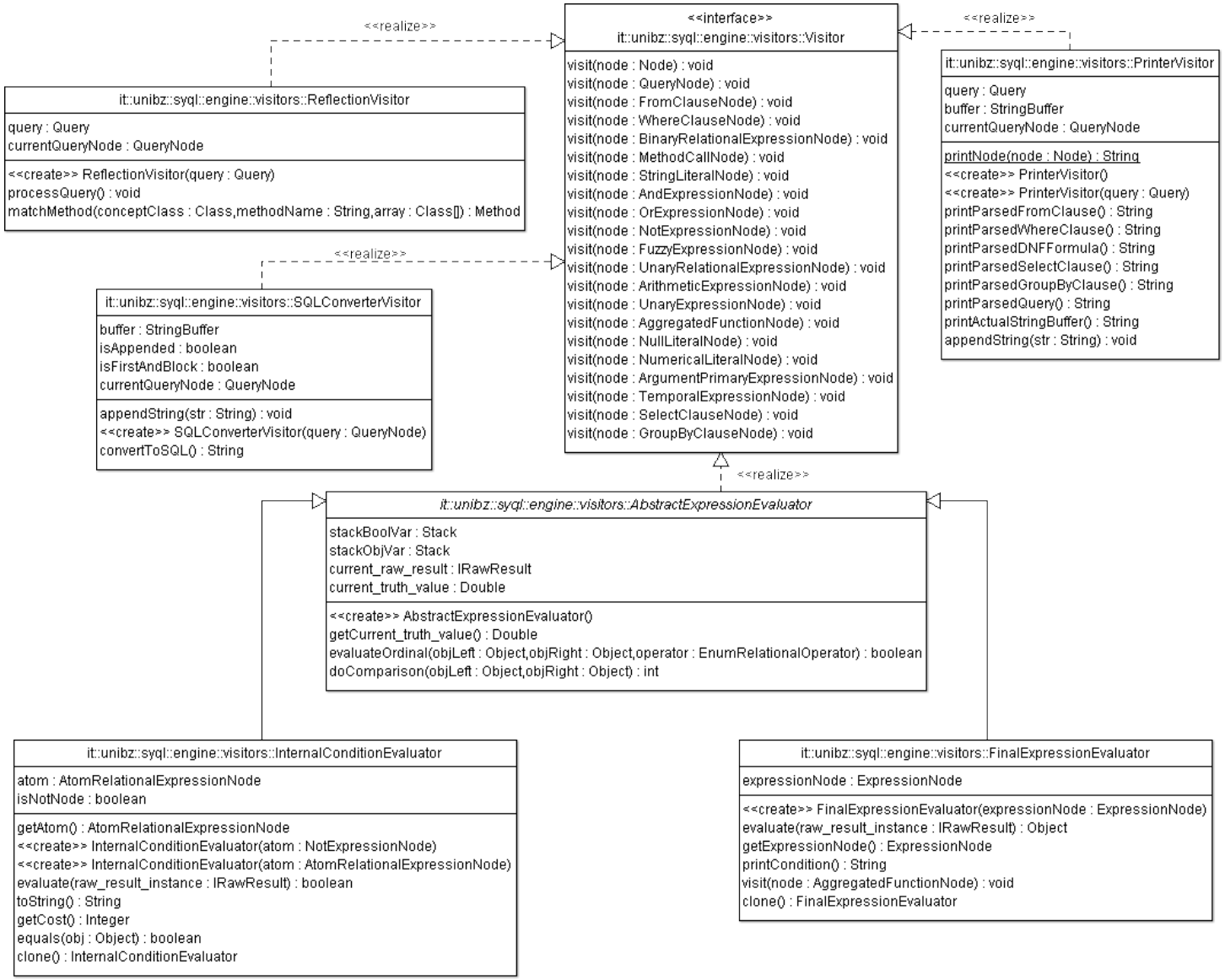

Figure 17: Class Diagram of the SyQL Engine Visitor classes (partial).

5.5 SyQL Query Planning and Execution

In this section, we are going to show how a SyQL query is processed. A detailed description of the techniques adopted is given; examples are provided.

5.5.1 The Main Approach

As we have said before (section 5.4), the SyQL query engine works on top of the DBMS (Figure 16). It has been implemented by reusing the underlying query planner and executor. The idea is to push as many conditions as possible into the query engine of the underlying DBMS.

The main steps is to convert the conditions present in the *SyQLExpression* of the *WhereClause* in a Conjunctive Normal Form (CNF) by using Double negative elimination, distributive property of AND, and De Morgan's laws (see Formula 7).

Double negative elimination:

$$\neg\neg A = A$$

Distributive property of AND:

$$A \wedge (B \vee C) = (A \wedge B) \vee (A \wedge C)$$

De Morgan's laws:

$$\neg(A \vee B) = (\neg A) \wedge (\neg B)$$
$$\neg(A \wedge B) = (\neg A) \vee (\neg B)$$

Formula 7: DNF conversion logical equivalences.

"A CNF formula is a conjunction of clauses, each of which is a disjunction of literals. A literal is either a variable or its negation. For example, CNF formula $\psi = (A + \neg B) \cdot (B + \neg C + D)$ contains two clauses, $(A + \neg B)$ and $(B + \neg C + D)$, and five literals. A clause is *satisfied* if at least one of its literals assumes value 1, *unsatisfied* if all its literals assume value 0, *unit* if all but one literal assume value 0, and *unresolved* otherwise. Literals with no assigned Boolean value are *free literals*. A formula is satisfied if all its clauses are satisfied, and unsatisfied if at least one clause is unsatisfied. It is often simpler to refer to clauses as sets of literals, and to the CNF formula as a set of clauses." [85]

The CNF notation is very helpful, because a clause (conjuncts) of the CNF formula

can be processed by the underlying DBMS only if all the conditions (literals) inside the clause are evaluated as external, otherwise the clause must be evaluated by the SyQL query engine. A condition is evaluated as an external one if all the predicates (the method calls) of the condition are external, otherwise a condition is evaluated internally. If all the conditions of a query are evaluated as external, we have a complete translation of the SyQL query into a SQL query with no specific engine processing.

In Appendix C, the DNF/CNF conversion algorithm is described in details.

5.5.2 Query Data Flow

In Figure 18, the query execution workflow is shown.

After the user has sent the query to the SyQL engine by using the XML-RPC protocol, the query text is parsed by the JavaCC generated parser. Next, the parse tree is processed by the *ReflectionVisitor*. This class only performs dynamic-binding for methods by using JMMF as shown below.

```
[1]  private Method matchMethod(  Class<?> conceptClass, String methodName,
[2]                               Class<?>[] args_type_array)
[3]          throws NoSuchMethodException, MultipleMethodsException {
[4]     // Here we use JMMF.
[5]     MultiMethod mm =
[6]             MultiMethod.create(  conceptClass, methodName,
[7]                                  args_type_array.length);
[8]     Method rtnMthd = mm.getMethod(conceptClass, array);
[9]     return rtnMthd;
[10]
[11]  }
```

Listing 38: JMMF method resolver invocation.

The above method is invoked by the method *visit(MethodCallNode)*. Another functionality offered from this visitor is the conversion of the *SyQLTemporalExpression*(s) (see line [28] of the grammar specification) into *SimpleDateTime* class instances. This class adds functionalities to the *java.util.Date* class by using the Decorator pattern [58].

After that, the *SQLConverterVisitor* (SQL translator) visits the AST, so that the SyQL query is translated in SQL (Section 5.5.3 provides a complete example). This SQL query is executed against the SQL underlying database, hence the returning result set

is a super set of the final one. The query engine fetches these records into the Java runtime by using JDBC driver.

The third visitor is the *InternalConditionEvaluator*, which evaluates and discards the records that do not satisfy the internal CNF clauses of the *Where-Clause* expression, and if there is at least one fuzzy condition it also computes the truth value by applying Zadeh's rules [126].

Finally, *FinalExpressionEvaluator* is the last visitor invoked, and visits the elements in the *Select-Clause,* and it computes the final query results. If aggregate functions are specified, the engine applies these operators to the groups defined in the *Group-By-Clause* (as in SQL). The grouping is performed by following the order of the elements specified in the *Group-By-Clause.* All the elements present in the *Group-By-Clause* must be present in the *Select-Clause* (see line [41] of the grammar specification). We want to stress that by design of the SyQL engine, the query result coming out from the execution of a SyQL query is a collection of Java objects, which can be simply printed out or easily reused inside the Java runtime environment.

For printing the query results, the *PrinterVisitor* can be used. This class is not directly involved in the query execution process, and we have mainly implemented this class for debugging and testing purposes.

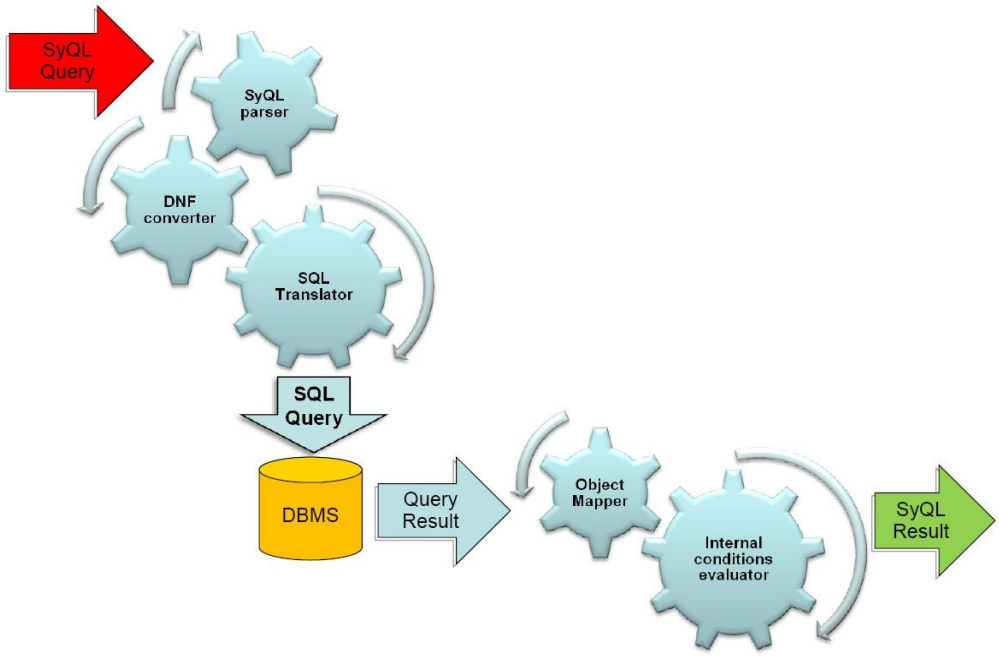

Figure 18: The SyQL Query Workflow.

Example 8:

In this example, for each method with the same name defined in the same class, we compute the total number of parameters.

```
[1]   FROM Method m
[2]   WHERE 1=1
[3]   SELECT  m.getNamespace(), m.getClassName, m.getMethodName(),
[4]           SUM(m.getParameters().size())
[5]   GROUP BY m.getNamespace(), m.getClassName, m.getMethodName();
```

Listing 39: SyQL query that returns the sum of all the parameters of methods with the same name defined in the same class.

The *FinalExpressionEvaluator* may produce an output like this:

m.getNamespace()	m.getClassName()	m.getMethodName()	m.getParameters().size()
it.unibz.namespc1	Class2	TheMethod	0
it.unibz.namespc2	Class1	Method_Foo	4
it.unibz.namespc1	Class3	TheMethod	6
	Class2	MyMethod	3
	Class1	Method_Foo	1
	Class2	MyMethod	5

These are the groups before applying the aggregate function SUM:

m.getNamespace()	m.getClassName()	m.getMethodName()	m.getParameters().size()
	Class1	Method_Foo	1
	Class2	MyMethod	5
	Class2	MyMethod	3
it.unibz.namespc1	Class3	TheMethod	6
it.unibz.namespc1	Class2	TheMethod	0
it.unibz.namespc2	Class1	Method_Foo	4

This is the final result:

m.getNamespace()	m.getClassName()	m.getMethodName()	Sum(m.getParameters().size())
	Class1	Method_Foo	1
	Class2	MyMethod	8
it.unibz.namespc1	Class3	TheMethod	6
it.unibz.namespc1	Class2	TheMethod	0
it.unibz.namespc2	Class1	Method_Foo	4

5.5.3 SQL Translation

In this sub-section, we show in practice how a SyQL query is translated into SQL and executed. The definitions of the concepts involved in this example are provided in Listing 40, and the example of query conversion is provided in Listing 41.

In this query, the *SyQLExpression* (lines 02-04) of the original version of the query is transformed into an equivalent CNF expression (lines 02-21 of the CNF version). Finally, the formula converter simplifies the formula by deleting redundant conditions (lines 12-13, 16, 17-18 of the CNF version), and the last query is then executed. In the simplified CNF version, the clauses at line 02, 06-07, 16 are considered external, therefore they are converted into SQL and executed against the DBMS. The remaining conditions are evaluated by the internal condition evaluator. By using the CNF, the number of the conditions grows up, but it is not a limitation for the performance, because we use caching mechanism for evaluating the conditions.

The internal condition evaluator is the most critical component for the performance, section 5.7 presents and discusses the implemented optimizations.

```
[1]  public class Class extends AbstractConcept {
[2]  ...
[3]     @ExternalCondition(columnName="class")
[4]     public String getClassName() {
[5]        return this.className;
[6]     }
[7]     @ExternalCondition(columnName="namespace")
[8]     public String getClassNamespace() {
[9]        return this.classNamespace;
[10]    }
[11]    @InternalCondition(cost = 20)
[12]    public ClassLOC getLOC() {
[13]       ...
[14]    }
[15]    ...
[16] }

[17] public class Method extends AbstractConcept {
[18] ...
[19]    @ExternalCondition(columnName="defclass")
[20]    public String getDefiningClassName() {
[21]       return this.definingClassName;
[22]    }
[23]    @ExternalCondition(columnName="methodname")
[24]    public String getMethodName() {
[25]       return this.methodName;
[26]    }
[27]    @InternalCondition(cost = 50)
[28]    public MethodLOC getLOC() {
[29]       ...
[30]    }
[31]    @InternalCondition(cost = 50)
[32]    public MethodNOA getNOA() {
[33]       ...
[34]    }
[35]    ...
[36] }
```

Listing 40: Definition of concepts Class and Method (partial).

Original version:

```
[1]   FROM Class c, Method m
[2]   WHERE   m.getDefiningClass() = c.getClassName() AND
[3]           (  m.getLOC() > 5 OR m.getLOC() < 150) AND
[4]              NOT (m.getNOA() > 3 OR m.getMethodName() LIKE 'test%')
[5]   SELECT m.getFullName();
```

CNF version:

```
[1]   FROM Class c, Method m
[2]   WHERE  ( m.getDefiningClass() = c.getClassName()) AND
[3]      (  m.getDefiningClass() = c.getClassName() OR m.getLOC() < 150) AND
[4]      (  m.getDefiningClass() = c.getClassName() OR
[5]         NOT m.getNOA() > 3) AND
[6]      (  m.getDefiningClass() = c.getClassName() OR
[7]         NOT m.getMethodName() LIKE 'test%') AND
[8]      (  m.getLOC() > 5 OR  m.getDefiningClass() = c.getClassName()) AND
[9]      (  m.getLOC() > 5 OR m.getLOC() < 150) AND
[10]     (  m.getLOC() > 5 OR NOT m.getNOA() > 3) AND
[11]     (  m.getLOC() > 5 OR NOT m.getMethodName() LIKE 'test%') AND
[12]     (  NOT m.getNOA() > 3 OR
[13]        m.getDefiningClass() = c.getClassName()) AND
[14]     (  NOT m.getNOA() > 3 OR m.getLOC() < 150) AND
[15]     (  NOT m.getNOA() > 3) AND
[16]     (  NOT m.getNOA() > 3 OR NOT m.getMethodName() LIKE 'test%') AND
[17]     (  NOT m.getMethodName() LIKE 'test%' OR
[18]        m.getDefiningClass() = c.getClassName()) AND
[19]     (  NOT m.getMethodName() LIKE 'test%' OR m.getLOC() < 150) AND
[20]     (  NOT m.getMethodName() LIKE 'test%' OR NOT m.getNOA() > 3) AND
[21]     (  NOT m.getMethodName() LIKE 'test%')
[22]  SELECT m.getFullName();
```

CNF simplified version:

```
[1]   FROM Class c, Method m
[2]   WHERE  ( m.getDefiningClass() = c.getClassName()) AND
[3]      (  m.getDefiningClass() = c.getClassName() OR m.getLOC() < 150) AND
[4]      (  m.getDefiningClass() = c.getClassName() OR
[5]         NOT m.getNOA() > 3) AND
[6]      (  m.getDefiningClass() = c.getClassName() OR
[7]            NOT m.getMethodName() LIKE 'test%') AND
[8]      (  m.getLOC() > 5 OR  m.getDefiningClass() = c.getClassName()) AND
[9]      (  m.getLOC() > 5 OR m.getLOC() < 150) AND
```

```
[10]     ( m.getLOC() > 5 OR NOT m.getNOA() > 3) AND
[11]     ( m.getLOC() > 5 OR NOT m.getMethodName() LIKE 'test%') AND
[12]     ( NOT m.getNOA() > 3 OR m.getLOC() < 150) AND
[13]     ( NOT m.getNOA() > 3) AND
[14]     ( NOT m.getNOA() > 3 OR NOT m.getMethodName() LIKE 'test%') AND
[15]     ( NOT m.getMethodName() LIKE 'test%' OR m.getLOC() < 150) AND
[16]     ( NOT m.getMethodName() LIKE 'test%')
[17] SELECT m.getFullName();
```

Listing 41: Example of CNF conversion written in SyQL.

By looking at the SQL translation in Listing 41, it is possible to see that the SQL From-clause contains the defining SQL query of the *Class* concept, and the SQL Where-clause contains the translation of the external clauses. We say every occurrence of concept is translated into SQL by appending in the from-clause of the translated SQL query its defining SQL query as an inline view, and every external clause is translated into SQL by appending in the where-clause of the translated SQL query its SQL conversion performed by looking at the method annotations.

By looking at the defining SQL query of concept *Class*, it is possible to see that the data necessary to instantiate a *Class* object are fetched from *entity* table of the AISEMA DBMS. This table contains also the data for instantiating *Method* objects.

The *getEffort(..., ...)* method uses the data fetched from *entity* table to retrieve and aggregate the effort data from another table of the AISEMA database called *entity_property,* which has a column of *datatime* type.

5.6 Support for Linguistic Variables

Linguistic variables support is a feature of SyQL, which enables users to specify conditions without knowing neither the range nor the order of magnitude of a certain property returned by a concept's method. Linguistic variables allow a user to filter the query results by abstracting values. Indeed, variables like 'high', 'medium', and 'low' can be used without missing any important results. These variables have to be used together with fuzzy logic constructs. One of the most important fuzzy logic constructs is the fuzzy equal operator, which is identified in SyQL by the “IS” keyword.

The term "fuzzy logic" [126] was defined after the development of the theory of fuzzy

sets [127] by Lotfi Zadeh. This logic is derived from fuzzy set theory, and allows dealing with reasoning that is approximate rather than precise. In contrast with the binary logic that uses the binary sets, also known as crisp logic, the fuzzy logic variables may have a membership value that ranges (inclusively) between 0 and 1. Furthermore, also the set membership values can range (inclusively) between 0 and 1 as in fuzzy set theory. In fuzzy logic, the degree of truth of a statement can assume a real value between 0 and 1, and it is not constrained to the two truth values {true (1), false (0)} as in classic crisp logic. [99] When linguistic variables are used together with crisp logic operators, the computation of a single final truth value requires to use specific functions like Zadeh logical operators (see Table 8).

Operator	Truth value computation
NOT	`truth(NOT x) = (1 - truth(x))`
AND	`truth(x AND y) = min(truth(x), truth(y))`
OR	`truth(x OR y) = max(truth(x), truth(y))`

Table 8: Zadeh logical operators.

In the next subsection, we are going to describe how we have implemented the linguistic variable support inside the SyQL query engine by evidencing the aspect of the extensibility, which allows us to define new fuzzy variables without modifying the SyQL query engine.

5.6.1 Implementation

The SyQL parser parses a linguistic variable as an *Identifier* (see the definition of the *FuzzyLabel* non-terminal symbol at line [54] of the grammar specification). We made this choice in order to improve the flexibility of the query engine. Like the concepts, also the fuzzy set identifiers are loaded at runtime. It means that it is possible to specify new fuzzy sets without modifying the internal structure of the query engine. All the method calls inside the *WhereClause* can be part of a *FuzzyExpression,* if and only if the returning type of the method implements the interface *IFuzzableType*. This interface enables to get two information: 1) the value of the measure through the method *getPropertyValue()*; 2) the instance of the class that performs the evaluations using the fuzzy sets through the method *getFuzzyEvaluatorInstance()*. This last class

instance must implement the interface *IFuzzyEvaluator*.

The *IFuzzyEvaluator* interface is shown in Figure 21. This interface defines two methods:

- *getFuzzyLiteralList()*: it returns the list of String(s) that identify the different fuzzy sets;
- *getFuzzyScore(String, Object)*: it computes the truth value for the measure (second parameter) in the specified fuzzy set (first parameter). To avoid assumptions about the type of handled measure, we define the measure parameter as a general *Object*.

It is possible to see that the fuzzy measures have been kept separated from the fuzzy evaluators. In the next subsection, we are going to describe the reasons of this architectural choice by using several examples related to software metrics domain.

5.6.2 Linguistic Variables and Software Metrics

As mentioned before, some metrics cannot be evaluated correctly by using only one measure; the Cyclomatic Complexity alone is not very useful to predict some qualitative aspects of the code like the complexity of maintenance (see Example 4 on page 20). Conversely, the ratio between Cyclomatic Complexity and Lines Of Code can be more helpful. Gill *et al.* in 1991 [60] have shown that the density of complexity can predict the complexity of maintenance more effectively than the Cyclomatic Complexity alone. The same happens for coding effort. In our metric collection system, we store the developer effort as a positive long number. Evaluating the effort as a number without any context is hard. If we want to know whether the effort spent on a particular code artifact (class/method) is 'high', we have to contextualize better our query. Because, the effort is evaluable only if the observational time interval (defined from the start/end time) is known: if a developer has spent today about six hours of effort on a specific class, this is a more intense effort if it is compared with another developer who has spent the same effort on the same class in the last month.

All these context information are passed to the evaluator class by constructors (see classes on the right side of Figure 21). The classes *MethodEffortFuzzyEvaluator* and

ClassEffortFuzzyEvaluator take the start/end time as constructor parameters. Also the *ClassDeltaLOCFuzzyEvaluator* takes these time boundaries as constructor parameters. This is because, evaluating the growth of a class without knowing the interval of interest is impossible.

In order to enable users to benefit rapidly of the collected software metrics, for each software metric evaluable by fuzzy logic, we have defined a specific evaluator (see Figure 21).

Creating fuzzy sets for software metrics evaluation is not trivial. We have done a tentative by creating three different fuzzy sets only for metrics collected from 100 Open Source Java projects (see Appendix B). Unfortunately, only a subset of metrics has been collected (see Table 9).

Metric	MIN	AVG	MAX
CC/LOC	0.16	0.20	0.24
NOM/Class	4	7	10
Class/Package	6	17	26
DIT	2	3	5
WMC	14	87	184
LOC/Class	25	90	200
CBO	2	8	18
LCOM	0	19	30

Table 9: Descriptive statistics of the collected software metrics of 100 projects.

Therefore, the following metrics can be evaluated only by using crisp logic conditions:

- Method LOC;
- Method Fan-in;
- Method Fan-out;
- Method Halstead Volume;
- Method NMI;
- Method NMC;
- Class NOC;
- Class NOA.

For each metrics in Table 9, we have created three fuzzy sets identified by the linguistic variables 'High', 'Medium', and 'Low' (see Figure 19). In the future, when project data availability will increase (more case studies), others fuzzy clustering techniques will be explored [36].

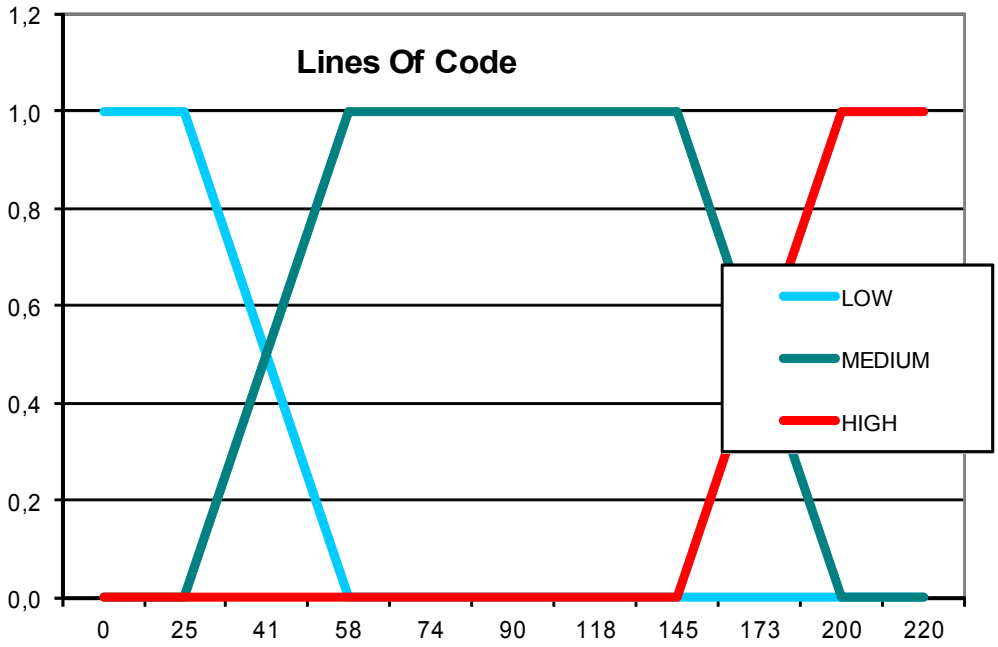

Figure 19: Membership functions of the Line Of Code metric.

The membership functions of the three fuzzy sets are defined by using the maximum, the average, and the minimum; the *value* of the metric is the independent variable, the truth value is the dependent one. The general implemented formula is the following:

$$LOW = \begin{cases} 1 & 0 \le value \le MIN \\ \dfrac{2 \cdot value}{MIN - AVG} - \dfrac{MIN + AVG}{MIN - AVG} & MIN < value \le \dfrac{(MIN + AVG)}{2} \\ 0 & value > \dfrac{(MIN + AVG)}{2} \end{cases}$$

$$MEDIUM = \begin{cases} 0 & 0 \le value \le MIN \\ \dfrac{2 \cdot value}{AVG - MIN} - \dfrac{2 \cdot MIN}{AVG - MIN} & MIN < value < \dfrac{MIN + AVG}{2} \\ 1 & \dfrac{MIN + AVG}{2} \le value \le \dfrac{AVG + MAX}{2} \\ \dfrac{2 \cdot value}{AVG - MAX} - \dfrac{2 \cdot MAX}{AVG - MAX} & \dfrac{AVG + MAX}{2} < value < MAX \\ 0 & value \ge MAX \end{cases}$$

$$HIGH = \begin{cases} 0 & value \le \dfrac{AVG + MAX}{2} \\ -\dfrac{2 \cdot value}{AVG - MAX} + \dfrac{AVG + MAX}{AVG - MAX} & \dfrac{AVG + MAX}{2} < value < MAX \\ 1 & value \ge MAX \end{cases}$$

Formula 8: General definition of the software metrics membership functions.

About the other "time span dependent" metrics (class effort, method effort, and class delta LOC), we have used the same formula (Formula 8), and we have computed the three parameters by doing statistical analysis on a case study lasting two years (see details in section 9.2). In these two years, software metrics and effort have been both collected.

Metric	MIN	AVG	MAX
Method Effort/Working day	1sec	2min 14sec	5h 31min 43sec
Class Effort/Working day	1sec	7min 29sec	11h 45min 36sec
Class Delta LOC/Working day	0	1,89	145

Table 10: Descriptive statistics of the effort data and delta LOC coming from the first case study.

In Table 10, the descriptive statistics of effort and delta LOC are presented. We want to highlight that the efforts cumulative values per working day are computed over the

whole development team; they are not related to each developer separately (e.g. If the *developer_A* has spent 20 seconds of effort on the class *Foo*, and the *developer_B* has spent 34 seconds on the same class during the same day, then, the overall effort per this day for this class is 54 seconds).

Regarding the variation of lines of code for a class (delta LOC), we want to point out that the statistical summary is computed over absolute values: positive and negative variations of class LOC are both evaluated.

We conclude this section by saying that in this particular domain, the architecture of the SyQL query engine allows the user to stretch the proper fuzzy membership function along the temporal line transparently. This enables the final user of the language to evaluate time-dependent metrics.

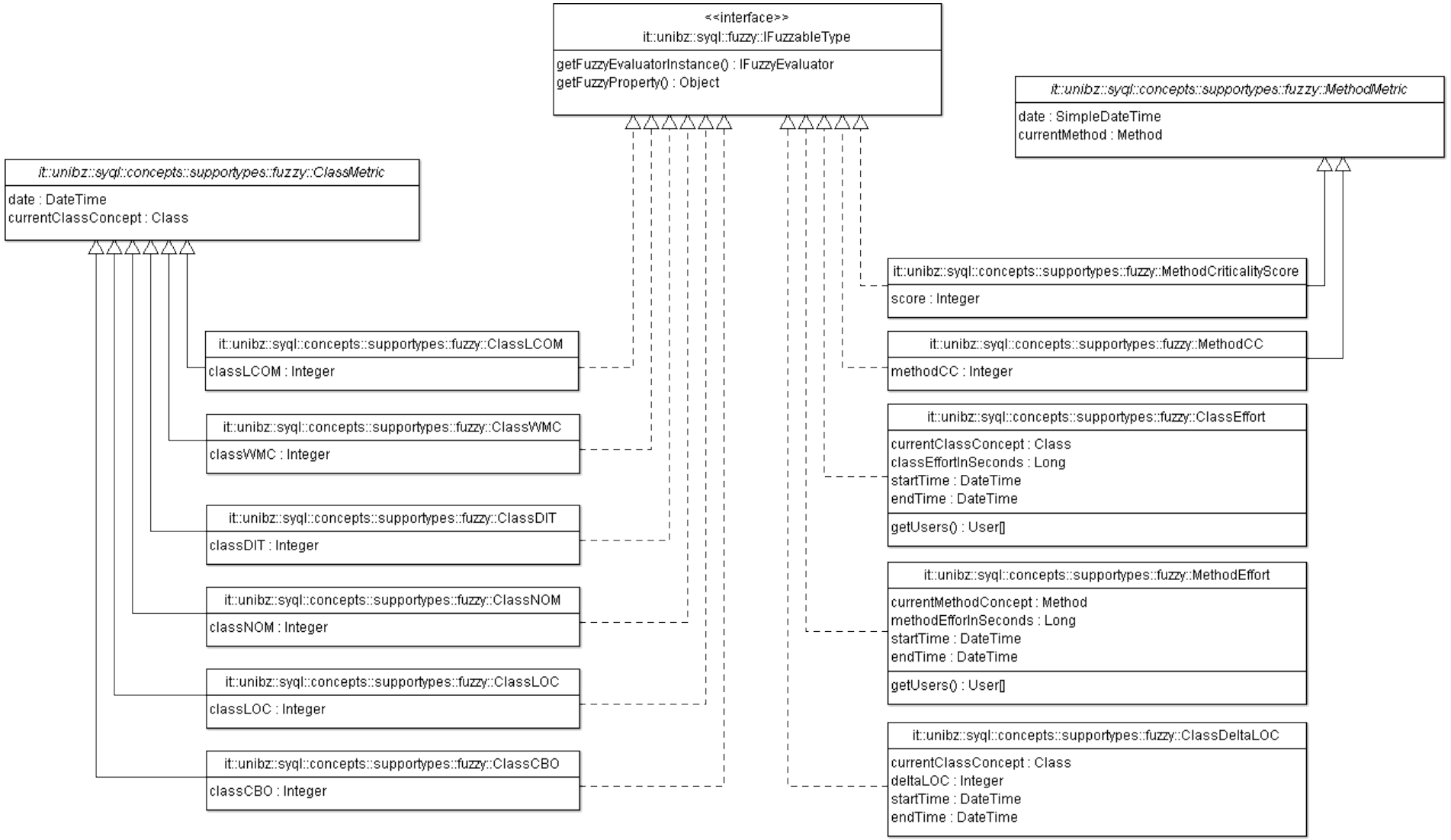

Figure 20: : Class Diagram of the metrics classes evaluable by fuzzy logic (partial).

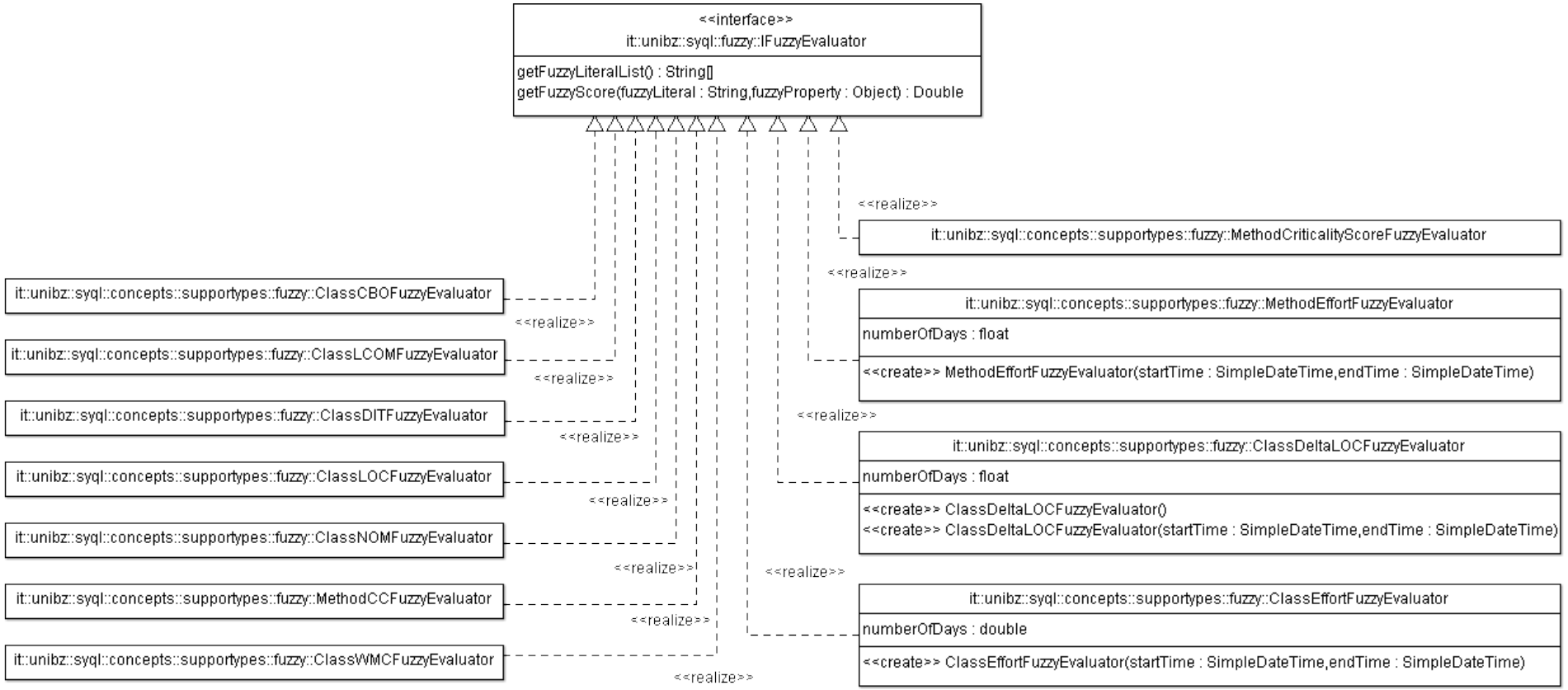

Figure 21: : Class Diagram of the software metric fuzzy evaluators (partial).

5.7 Optimizations

In this section, we discuss the optimizations implemented to make a SyQL query executable on an affordable machine in a reasonable amount of time.

Usually, internal conditions require a lot of computations. This is because most of them need to fetch data from the database several times. To address this problem, we adopted two solutions: 1) sorting these conditions according to their cost, wherein the cost is estimated by the developer of the SyQL libraries during the implementation (see line [11] of the concept class definition in Listing 40); 2) evaluating these conditions in parallel taking advantage of the modern parallel/multicore hardware architectures.

5.7.1 Conditions Sorting

As in SQL, the evaluation order of the conditions is cost-based. This cost is helpful to speed up the execution of a query by giving extra information to the query planner. We use these costs to sort the CNF clauses and the conditions inside a clause. This simple optimization allows us to take benefit from the CNF notation.

The costs are computed by invoking the method *getCost()* specified by the *AtomRelationalExpressionNode* interface. This method visits the underlying branch of the parsing tree, and it sums all the costs associated to the leafs.

The possible leafs with the associated costs are:

- *NumericalLiteralNode* – cost = 0;
- *NullLiteralNode* – cost = 0;
- *NumericalLiteralNode* – cost = 0;
- *StringLiteralNode* – cost = 0;
- *TemporalExpressionNode* – cost = 1
- *MethodCallNode* – cost = defined by the library developers:

  The cost is associated to a specific method by using the *cost* parameter of the *InternalCondition* annotation (see line [11] of the concept class definition in Listing 40).

  A condition may be composed by N different method calls:

  - *m.isATestMethod()* – 1 method call;

- *m.getLOC() > getLOC(yesterday)* – 2 method calls;
- *m.getLOC(u,getDate('2010-01-03')) < m.getLOC((u,getDate('2010-01-05'))* – 4 method calls
- ...
- … – N method class

The cost of the j-th condition in the i-th CNF clause is defined as follows:

$$C_{ij} = \sum_{k=0}^{N} C_{ijk}$$

Where
C_{ijk} = cost of the k-th leaf of the j-th condition in the i-th CNF clause.

Formula 9: Cost of the j-th leaf in the i-th CNF clause.

This definition allows to sort all the conditions into the i-th CNF clause according to the costs:

$$\{..., C_{i2}, C_{i1}, C_{i3}, ...\} \; iff \; C_{i2} \le C_{i1} \wedge C_{i1} \le C_{i3} \wedge C_{i2} \le C_{i3}$$

Formula 10: Conditions sorting.

The cost of the i-th CNF clause is the sum of the costs of all the M conditions contained into it:

$$C_i = \sum_{j=1}^{M} C_{ij}$$

Formula 11: Cost of the i-th CNF clause.

The clauses are sorted according to the costs:

$$\{..., C_3, C_1, C_{2,} ...\} \; iff \; C_3 \le C_1 \wedge C_1 \le C_2 \wedge C_3 \le C_2$$

Formula 12: Clauses sorting.

To avoid extra computation, the duplicates of the clauses are deleted; the same happens inside clauses for conditions. After this pruning, we use the Bubble Sort Algorithm to sort all the remaining conditions.

5.7.2 Parallel Execution

Another optimization that we have implemented, it is the parallel evaluation of the internal conditions. Today multi core/multiprocessor machines are affordable, and machines with tens of processors are relatively cheap. We have taken benefits by using this architecture: Java multi threading APIs enabled us to speed up the query execution process with an interesting acceleration factor.

In the current SyQL release, the internal condition evaluator processes the partial result set (materialized from the underlying database) in parallel: the component creates N distinct "worker" threads (where by default N is the number of logical CPU available in the running system), each of them processes a single record per time. By using this Master/Worker threading schema, the thread scheduler can have the maximum freedom in order to achieve the maximum acceleration factor.

5.7.3 Concept Tuning

In order to speed up the execution of concept methods, it is possible to implement local caches. This increases the memory space needed to execute the SyQL engine, but the execution time of a method that accesses to the database (like *Class.getLOC(SimpleDateTime)*) may be reduced up to two orders of magnitude. We have decided to use local caches instead of global ones, because problems may arise when late binding takes place, so the global cache requires complex policies to be kept consistent by adding unnecessary complexity to the system.

In Listing 42, an example of local cache is shown. We have marked the field *_cache_GetLOC* as transient in order to keep the caching information inside the SyQL engine without sending useless data to the client.

The creation of a second meta model for the data can be very useful to speed up data analysis. Sen *et al.* in 2007 [111] did a review on 30 different data warehousing methodologies. To make possible the implementation of one of them, we have specified two extra methods in the *AbstractConcept* abstract class: *conceptSetup(), conceptDispose()*. These two methods are invoked before and after the query execution respectively. Only the methods' implementations of the concepts involved in the query are executed (details are provided in Section 6).

These two methods make possible to transform the data automatically and

transparently for the user. It is also possible to inhibit the invocation of the *conceptDispose()* by specifying the value of the *keepSessionAlive* flag of the query engine as true. In this way, the created meta model will not be disposed, and it is possible to save time for subsequent analyses.

```
[1]  transient private HashMap<SimpleDateTime,ClassLOC> _cache_GetLOC =
[2]       new HashMap<SimpleDateTime,ClassLOC> ();
[3]  ...
[4]  @InternalCondition(cost = 50)
[5]  public ClassLOC getLOC(SimpleDateTime temporalArg) {
[6]     int returnLOC = -1;
[7]     ClassLOC rtnLOC = null;
[8]     Connection conn = null;
[9]     ResultSet rs = null;
[10]    SimpleDateTime date = null;
[11]
[12]    rtnLOC = this._cache_GetLOC.get(temporalArg);
[13]    if(rtnLOC != null)
[14]       return rtnLOC;
[15]    ...
[16]    ...
[17]    rtnLOC = new ClassLOC(this, returnLOC, date);
[18]    rtnLOC = storeInCache(this._cache_GetLOC, temporalArg, rtnLOC);
[19]    return rtnLOC;
[20] }
```

Listing 42: Implementation of a local cache for a concept method.

5.8 Available Concepts

In this section, we present the available schema, which contains the concept classes. These classes are implemented to allow the user to access the PROM database [109]. At present, only the concepts that have been considered to be helpful for SyQL users have been implemented. Future developments of SyQL will include more concepts. As an example it could be useful to make available as a concept the *"WorkingDay"* class that allows a manager to inspect the today activities by comparing them with the past ones. In the current implementation, the eight concepts available are the following:

- **Class**: it represents the classes parsed by the source code analyzer;
- **Method**: it represents the methods parsed by the source code analyzer;
- **TestMethod**: it is a specialization of Methods, and it only returns test methods, the disambiguation between the two concepts is done reading the NUnit (http://www.nunit.org/) code annotations;
- **Bug**: it represents all the bugs present into the bug tracking system;
- **ClosedBug**: it is a specialization of Bug, and it represents all the bugs closed into the bug tracking system;
- **User:** it represents the users who are monitored by the metrics collection system;
- **Chronon**: it represents all the dates in which the metrics collection system was active;
- **Util**: it defines a single object, so as the cardinality of the query results is not affected. It is particularly useful for manual specification of the dates (dates are not a terminal symbol of the language).

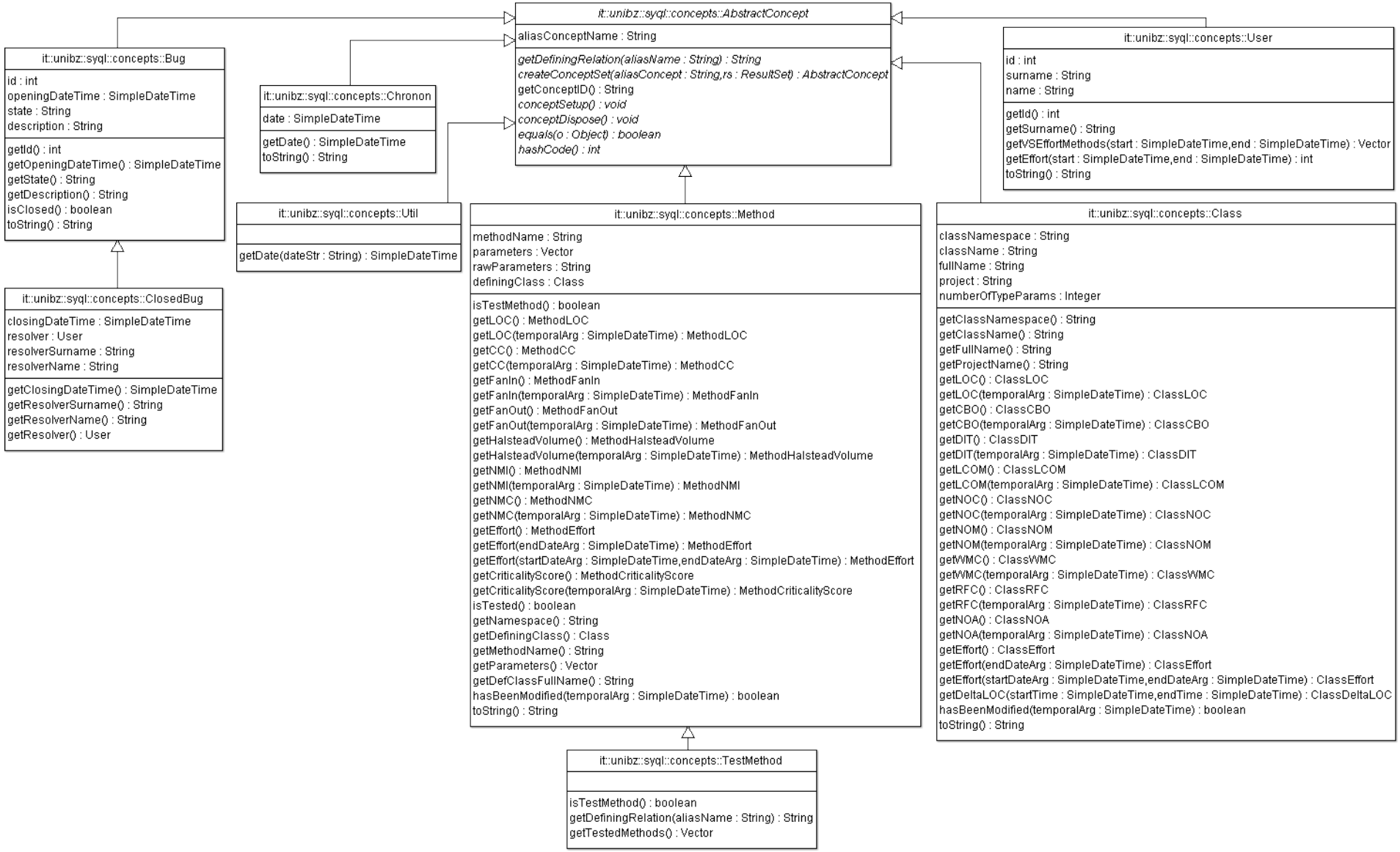

Figure 22: Class Diagram of the available SyQL concepts (partial).

5.8.1 Class

The C*lass* concept allows the user to manipulate the code classes recognized by the Source Code Metrics Extractors [109] (see Chapter 4). This class provides methods to retrieve the following information: unique class identifier, software metrics, and effort metrics.

The *getFullName()* method concatenates the namespace name of the class if any with the class name. Parametrized types are not handled. For each type of software metric extracted by the code analyzer, the class provides two distinct methods: the former without parameters (e.g. *getLOC()*) that retrieves the last metric values, the latter with a temporal parameter that retrieves the value of the metrics at the specified time (e.g. *getLOC(SimpleDateTime)*). We also implemented a specific method (*getDeltaLOC(SimpleDateTime startTime, SimpleDateTime endTime)*) that returns the variation of lines of codes within a specified interval.

The software metrics available in this concept are the CK metrics, and the line of code; the definitions are provided below:

- LOC = Lines Of Code of a class. This measure is the sum of all the lines of code that are written into the class scope (field declarations, and methods) except for the inner classes.
- CBO = Coupling Between Objects classes. “CBO for a class is a count of the number of other classes to which it is coupled.” [31]
- DIT = Depth of Inheritance Tree. “Depth of inheritance of the class is the DIT metric for the class. In cases involving multiple inheritance, the DIT will be the maximum length from the node to the root of the tree.” [31]
- LCOM = Lack of Cohesion in Methods.

  “Consider a Class C_1 with n methods $M_1, M_2, \ldots, M_n$. Let {I,} = set of instance variables used by method M_i. There are n such sets $\{I_1\}, \ldots, \{I_n\}$. Let P = $\{(I_i, I_j) | I_i \cap I_j = \varnothing\}$ and Q = $\{(I_i, I_j) | I_i \cap I_j = \varnothing\}$. If all n sets $\{I_1\}, \ldots, \{I_n\}$ are $\varnothing$ then let P = $\varnothing$. ” [31]

$$LCOM = \begin{cases} |P| - |Q|, \text{if } |P| > |Q| \\ 0, \text{otherwise} \end{cases}$$

- NOC = Number of children. “Number of immediate subclasses subordinated to a class in the class hierarchy.” [31]
- NOM = Number of method of a class.
- WMC = Weighted Methods per Class. “Consider a Class C_1 with methods M_1…, M_n that are defined in the class. Let c_1..., c_n be the” cyclomatic “complexity of the methods. Then: ” [31]

$$WMC = \sum_{i=1}^{n} c_i$$

- RFC = Response for a class. It is defined as “|RS| where RS is the response set for the class.” [31]

$$RS = \{M\} \cup_{all\ i} \{R_i\}$$

Where
$\{R_i\}$ = set of methods called by method i
$\{M\}$ = set of all methods in the class

- NOA = Number of attributes of a class.

For what concerns the effort, it is computed by using the following formula:

$$Effort_C(t_{start}, t_{end}) = \sum_{i=1}^{N} \sum_{t \geq t_{start}}^{t \leq t_{end}} \sum_{m \in Methods_C} EffortEvent_{m,i}(t) + \sum_{i=1}^{N} \sum_{t \geq t_{start}}^{t \leq t_{end}} EffortEvent_C(t)$$

N := Number of developers
C := Specific class
t_{start} := Beginning of the observation interval
t_{end} := End of the observation interval
$Methods_C$ = Set of methods defined into the scope of class C
$EffortEvent_{m,i}(t)$ = Quantity of effort (in seconds) spent in the timeframe t on the method m by the developer i-th
$EffortEvent_C(t)$ = Quantity of effort (in seconds) spent in the timeframe t inside the general scope of the class C

Formula 13: Effort definition for classes.

To make easier effort retrieval, we have implemented three different methods:

- *getEffort()*: it returns the effort spent on the class from its creation to now (t_{start} = 0, t_{end} = now);

- *getEffort(SimpleDateTime endTime)*: it returns the effort spent on the class from its creation to the specified temporal parameter;
  (t_{start} = 0, t_{end} = *endTime*);
- *getEffort(SimpleDateTime startTime, SimpleDateTime endTime)*: it returns the effort spent on the class within the specified interval (t_{start} = *startTime*, t_{end} = *endTime*);

With these methods, we can easily navigate along the temporal line by showing software metrics and effort evolution.

Finally, we have defined the method *hasBeenModified(SimpleDateTime)* to manage class modifications. This method returns true if the class has been modified during the day specified as argument, otherwise it returns false.

5.8.2 Method and TestMethod

The concepts *Method* and *TestMethod* allow the user to manipulate the methods of the classes recognized by the Souce Code Metrics Extractor. These classes provide methods to retrieve the same information of the concept *Class:* there are two methods for each software metric, and three methods for the effort. The metrics available here are different from *Class* metrics. The collected software metrics for methods are the following:

- LOC = Lines Of Code. In our metrics collection system, we collect the number of statements that terminate with a semicolon (in Java, C/C++, and C# code);
- CC = McCabe Cyclomatic Complexity [88]. This number represents the minimum number of test cases that are required to achieve full code-coverage of the method (another definition is provided in Definition 1);
- Fan-In = Number of input of the procedure;
- Fan-out = Number of output of the procedure;

- Halstead Volume [62] = This measure gives an estimation of size of the compiled procedure. The Volume (V) is defined as follows:

$$\underline{V = (N1 + N2) \cdot \log(n1 + n2)}$$

$$N1 = total\ number\ of\ operators$$
$$N2 = total\ number\ of\ operands$$

$$n1 = number\ of\ distinct\ operators$$
$$n2 = number\ of\ distinct\ operands$$

Formula 14: Halstead's Volume definition.

- NMI = Number of Invocations for a method
- NMC = Number of Method Calls for a method

The semantic of the effort methods (*getEffort(...)*) is the same but the effort is defined in a different way. In the following, we provide the effort definition for the *Method* and *TestMethod* concepts:

$$Effort_M(t_{start}, t_{end}) = \sum_{i=1}^{N} \sum_{t \geq t_{start}}^{t \leq t_{end}} EffortEvent_{M,i}(t)$$

Where
N := Number of developers
M := Specific method
t_{start} := Beginning of the observation interval
t_{end} := End of the observation interval
$EffortEvent_{C,i}(t)$ = Quantity of effort (in seconds) spent in the timeframe t on the method M by the developer i-th

Formula 15: Effort definition for methods.

The distinction between *Method* and *TestMethod* is done by reading the code annotations; the source code analyzer reads the annotations from the code, and it stores them into the appropriate tables of the data warehouse. These tables are read by the implementation of the method *createConceptSet(String, ResultSet)* of class *Method* (for space reasons, we have not reported this method on the UML diagram in Figure 22). This method may return either a *Method* or a *TestMethod* instance.

In the *TestMethod* class, this method has been further specialized, but we have not implemented the dispatching between the two SyQL concepts, due to the fact that *TestMethod* is a subclass (a specialization) of the *Method* concept class. Obviously,

the SQL statements that define the two concepts are different.

The *TestMethod* class allows the user to see which methods are tested by using the method *getTestedMethods()*. This method returns a *Vector* that contains all the methods invoked by the current *TestMethod* instance. These two concepts are very useful to check the quality of the code (see Example 9).

Example 9:

In this example, we want to assess the current status of testing of the project. Therefore, for each tested method, we retrieve the McCabe Cyclomatic Complexity and the number of tests. This can be useful to identify a part of code that may not meet quality requirements.

```
[1]   FROM Method m, TestMethod t
[2]   WHERE   NOT m.isTestMethod() AND m.isTested AND
[3]           t.getTestedMethods().contains(m)
[4]   SELECT m, m.getCC(), COUNT(t)
[5]   GROUP BY m, m.getCC();
```

The returned result set could be:

m	m.getCC()	COUNT(t)
it.foo.Class1.getValue()	2	1
it.foo.HelperClass2.getInstance(int, int)	3	1
it.foo.package1.Class3.getIndex()	3	1
it.foo.package2.Class4.toString()	2	4
...	...	

Finally, as well as for *Class*, we have defined the method *hasBeenModified(SimpleDateTime)*. This method returns true when the current instance was modified during the day specified as argument, otherwise it returns false.

5.8.3 Bug and ClosedBug

The user can query data coming from the bug tracking system by using these two concept classes. The *Bug* class implements the same mechanism of the *Method* class

described before: the method *createConceptSet(String, ResultSet)* contains the logic to decide which class should be instantiated.

The concept *Bug* exposes methods that return typical bug information: the unique identifier, the opening DateTime, the current state of bug (unassigned, assigned, resolved, invalid, etc.).

The concept *ClosedBug* adds the information coming from the bug fixing process, which are the resolver name/surname and the closing DateTime.

5.8.4 User

The concept *User* represents the entire set of users who are present into the metrics collection system. For each of them, the system collects automatically the effort spent on different artifacts: documents, web-pages, source code files, methods, etc. The actual implementation of this concept provides two important methods:

- *getEffort(SimpleDateTime start, SimpleDateTime end)* returns the value of the effort spent by the current user on code artifacts in the specified time interval;
- *getEffortMethod(SimpleDateTime start, SimpleDateTime end)* returns a *Vector* that contains the name of the artifacts, on which, the current user has spent his/her effort in the specified time interval.

The other methods return the user's unique identifier and the user's surname.

5.8.5 Chronon

The Chronon concept implements the definition given by Jensen *et al.* in 1992:

> “ A chronon is the shortest duration of time supported by a temporal DBMS – it is a non decomposable unit of time. A particular chronon is a sub-interval of fixed duration on time-line.
>
> …
>
> ” [71]

In our implementation, the *Chronon* concept defines a set of *SimpleDateTime* objects. Each of the elements represents an upload of software metrics into the data warehouse. Usually, our software metrics extractor analyzes the modified/added parts of the code once a day. This concept is very useful, especially for showing the metrics evolution (see Example 10).

Example 10:

In this example, we show how it is possible to visualize the growth (in terms of lines of code) of a specific namespace (it.foo.MyNamespace).

```
[1]   FROM Class cl, Chronon chr
[2]   WHERE cl.getClassNamespace() = 'it.foo.MyNamespace'
[3]   SELECT SUM(cl.getLOC(chr.getDate()).getValue()), chr.getDate()
[4]   GROUP BY chr.getDate();
```

The method call *cl.getLOC().getValue()* is necessary, because the SUM aggregation function handles only integer values.

The returned result set could be:

SUM(cl.getLOC().getValue())	chr.getDate()
1102	2010-02-03
1158	2010-02-04
1286	2010-02-05
1312	2010-02-06
...	...

5.8.6 Util

The *Util* concept is a unary concept: it means that it has only one possible instance. The defining SQL statement is "SELECT 1". This concept does not affect the cardinality (number of records) of the query result. It only provides a function (*getDate(String)*) that converts *String* objects into *SimpleDateTime* objects. We can see this concept as a dummy concept, which does not fetch data from the database.

In conclusion, we have defined the currently available SyQL functionalities. In the future the number of the available concepts will be increased, and the already existing functionalities will be enhanced.

6 Extensibility

In this Chapter, we propose a methodology to extend SyQL by implementing new concepts and new fuzzy evaluators. This methodology has to be applied ex-post to the one described in Section 5.3.1 . By reading this Chapter, it is possible to get the necessary notions to create a new abstraction layer or to modify an existing one by using Java programming language.

We introduce the methodology by an example. In this example, we create a new abstraction layer for an existing database, where an imaginary company (hereinafter called *MyFooCompany*) stores the information about its employees, salaries, and teams.

After introducing the database schema, we describe in details the necessary steps to put in place the abstraction on types (by implementing new concepts) and on values (by implementing new fuzzy evaluators).

MyFooCompany database schema description:

MyFooCompany provides two types of employees: regular employees and managers. Each manager has the leadership of no more than one team, and each team is composed by at least one employee, who can be a manager, and one manager. The teams can be created and dissolved. An active team cannot receive a new member after its creation. The salary of each employee is calculated monthly by taking into the account of different factors not collected by the system.

Figure 23 shows a possible database schema produced by the engineers who designed the system. We can see that the *person* relation contains of all the employees of the company (both managers and other employees). The distinction between the two categories is possible by using the *manager* field. This is a typical solution to implement a partial generalization into a relational schema; in this particular case, the managers are a subset of the employees. The *team* relation contains all the information of the present and past working teams, and the *team_member* contains the composition of these teams. The *team_member* relation implements a many-to-many relationship that links *team* and *person* relations. Finally, the *montly_salary* relation contains all the retributions of the employees without distinction between manager

and regular employees.

In the next two sections, we apply the methodology for developing concepts classes and Fuzzy evaluators. These two types of software artifacts enable the SyQL query engine executing queries by using abstraction both on types and values.

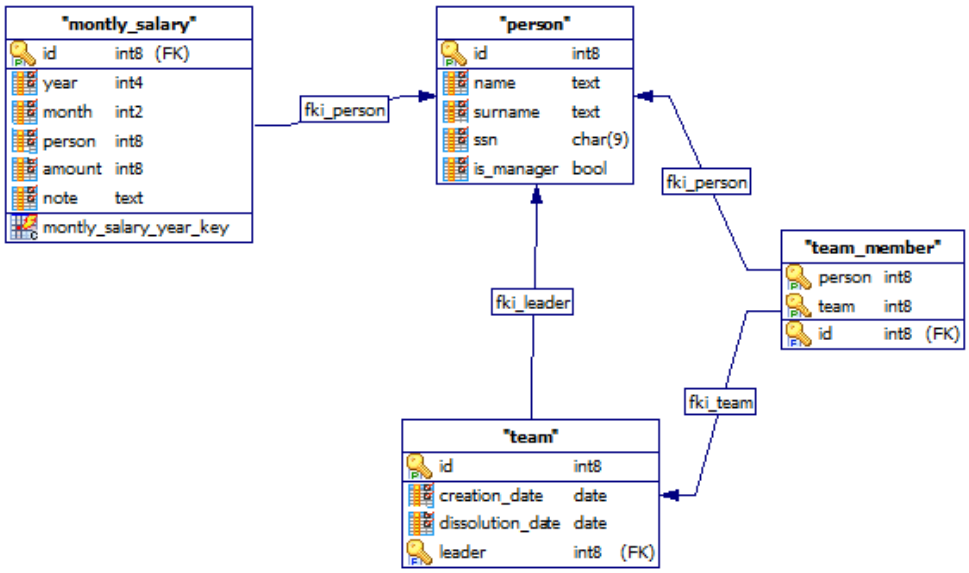

Figure 23: *MyFooCompany* database schema.

6.1 How to Implement New Concepts

In this Section, we show how it is possible to implement four new concepts starting from the schema shown in Figure 23.

After examining the database schema, we have identified several concepts that can simplify the usage of database data. These concepts are the following:

- *Employee* – this concept models all the persons who work inside the company *MyFooCompany;*
- *Manager* – this concept models all the manager of *MyFooCompany*;
- *Team* – this concept models all the working groups active and non active inside *MyFooCompany*;
- *ActiveTeam* – this concept models all the active working group inside the company.

We want to underline that the relations implementing a many-to-many relationship like *team_members* should not to be modeled as concepts. The information present

into them can be retrieved through methods defined into the concepts (e.g. Employee, Manager, Team, ActiveTeam), since a concept can provide a more precise context than a relational schema. In this way, the query writing process can be simplified by the possibility to use this "many-to-many translation pattern".

The implementation of concepts classes must fulfill few requirements. A new concept class must be created in the *it.unibz.syql.concepts* package by extending (directly or indirectly) the *it.unibz.syql.concepts.AbstractClass* abstract class specifying a dummy default constructor. *AbstractClass* requires the implementation of the following methods.

- `public String getDefiningRelation(String aliasName);`
- `public AbstractConcept createConceptSet(String aliasConcept, ResultSet rs);`
- `public int hashCode();`
- `public boolean equals(Object o);`
- `public void conceptDispose() throws Exception;`
- `public void conceptSetup() throws Exception.`

We recommend that all the concept classes specify a unique *serialVersionUID* static final long field. In this way, the class instances can be properly serialized.

6.1.1 Step 1 – SQL Definition

After identifying the necessary concepts by following the methodology proposed in Section 5.3.1 , it is necessary to write a defining SQL query for each of them. The query result must contain all the instances that the concept wants to represent. Each instance must be contained in a separate record, and the fields must at least identify the instance uniquely. It means that the SQL select clause must contain at least one key.

Table 11 lists the SQL definitions of the concepts under development.

Employee	**SELECT** id, name, surname, ssn, is_manager **FROM** person;
Manager	**SELECT** id, name, surname, ssn **FROM** person **WHERE** is_manager = true;
Team	**SELECT** t.id, t.creation_date, t.dissolution_date, l.id, l.name, l.surname, l.ssn **FROM** team AS t INNER JOIN person AS l ON t.leader = l.id
ActiveTeam	**SELECT** t.id, t.creation_date, l.id, l.name, l.surname, l.ssn

	FROM team AS t INNER JOIN person AS l ON t.leader = l.id **WHERE** dissolution_date IS NULL;

Table 11: Definitions of *MyFooCompany* concepts by using SQL.

We want to stress that the relations implementing a many-to-many relationship (e.g. *team_members*) shall not appear in the SQL definitions, since they can affect the cardinality of the SQL definitions by making the concepts meaningless.

When all the SQL queries are defined, it is possible to start implementing the skeletons of new concepts by implementing the *getDefiningRelation* method. For brevity sake, we show the implementations of this method only for *Employee* (Listing 43) and *Manager* (Listing 44) classes. We want to underline that *Manager* is implemented as a subclass of *Employee.* In this way, we can reuse part of the logic of the *Employee* class. The same is for *Team*, which extends *ActiveTeam.*

```
[1]  public class Employee extends AbstractConcept {
[2]     ...
[3]     @Override
[4]     public String getDefiningRelation(String aliasName) {
[5]        return  "SELECT id AS " + aliasName + "_id," +
[6]                " name AS " + aliasName + "_name," +
[7]                " surname AS " + aliasName + "_surname," +
[8]                " ssn AS " + aliasName + "_ssn," +
[9]                " is_manager AS " + aliasName + "_is_manager" +
[10]               "FROM person";
[11]    }
[12]    ...
[13] }
```

Listing 43: Definition of the *getDefiningRelation* method of the *Employee* concept.

```
[1]  public class Manager extends Employee {
[2]     ...
[3]     @Override
[4]     public String getDefiningRelation(String aliasName) {
[5]        return  "SELECT id AS " + aliasName + "_id," +
[6]                   " name AS " + aliasName + "_name," +
[7]                   " surname AS " + aliasName + "_surname," +
[8]                   " ssn AS " + aliasName + "_ssn" +
[9]                "FROM person " +
[10]               "WHERE is_manager = true";
[11]    }
```

```
[12]    ...
[13] }
```

Listing 44: Definition of the *getDefiningRelation* method of the *Manager* concept.

The *getDefiningRelation* method provides a string parameter, because it is necessary to use this parameter as prefixes for the fields names declared in SQL select clause. In this way, the concept is safe from naming conflicts during the execution of the SQL code.

6.1.2 Step 2 – Definition of Attributes and Constructors

Usually, we define two constructors per concept: a dummy one, and a complete one. Listing 45 shows the parameters and the constructors of *Employee*, which stores the same information present in the SQL select clause.

```
[1]  public class Employee extends AbstractConcept {
[2]     ...
[3]     protected int id;
[4]     protected String name;
[5]     protected String surname;
[6]     protected String ssn;
[7]
[8]     public Employee() {
[9]        this.id = -1;
[10]       this.name = null;
[11]       this.surname = null;
[12]       this.ssn = null;
[13]    }
[14]    public Employee(int id, String name, String surname, String ssn) {
[15]       this.id = id;
[16]       this.name = name;
[17]       this.surname = surname;
[18]       this.ssn = ssn;
[19]    }
[20]    ...
[21] }
```

Listing 45: Constructors of the *Employee* class.

Listing 46 shows the constructors of the *Manager* class. They completely reused the logic of the constructors defined in the *Employee* superclass.

```
[1]  public class Manager extends Employee {
[2]     ...
[3]     public Manager() {
[4]        super();
[5]     }
[6]     public Manager(int id, String name, String surname, String ssn) {
[7]        super(id, name, surname, ssn);
[8]     }
[9]     ...
[10] }
```

Listing 46: Constructors of the *Manager* class.

Listing 47 shows the attributes and the constructors of the *ActiveTeam* class. At line [5], it is possible to see the definition of the *leader* field, which is instantiated by the second constructor at lines [19]-[20]. In this way, it is possible to access to all the methods exposed by the *Manager* class.

```
[1]  public class ActiveTeam extends AbstractConcept {
[2]     ...
[3]     protected int teamID;
[4]     protected SimpleDateTime creationDate;
[5]     protected Manager leader;
[6]
[7]     public ActiveTeam() {
[8]        this.teamID = -1;
[9]        this.creationDate = null;
[10]       this.leader = null;
[11]    }
[12]
[13]    public ActiveTeam(
[14]          int teamID, SimpleDateTime creationDate,
[15]          int leader_id, String leader_name,
[16]          String leader_surname, String leader_ssn) {
[17]       this.teamID = teamID;
[18]       this.creationDate = creationDate;
[19]       this.leader = new Manager(leader_id, leader_name,
[20]                                 leader_surname, leader_ssn);
[21]   }
[22]    ...
```

```
[23] }
```

Listing 47: Constructors of the *ActiveTeam* class.

Listing 48 shows the partial definition of the *Team* class by evidencing the addition of the *dissolutionDate* field (at line [3]) necessary to model dissolved teams. The class reused part of the logic of its superclass (see lines [15]-[16]).

```
[1]  public class Team extends ActiveTeam {
[2]     ...
[3]    protected SimpleDateTime dissolutionDate;
[4]
[5]     public Team() {
[6]        super();
[7]        this.dissolutionDate = null;
[8]     }
[9]
[10]    public Team (
[11]          int teamID, SimpleDateTime creationDate,
[12]          SimpleDateTime dissolutionDate, int leader_id,
[13]          String leader_name,
[14]          String leader_surname, String leader_ssn) {
[15]       super(  teamID, creationDate, leader_id,
[16]               leader_name, leader_surname, leader_ssn);
[17]       this.dissolutionDate = dissolutionDate;
[18]    }
[19]    ...
[20] }
```

Listing 48: Constructors of the *Team* class.

6.1.3 Step 3 – Definition of the Materialization Logic

After defining all the constructors, it is possible to define the creation logic for each concept. Operationally, we implement the method *createConceptSet* for collecting data from the passed result set. This operation could be not trivial. For instance, the *Employee* and *Team* concepts require an extra logic for instantiating objects at runtime in the proper way.

To prevent implementation issues, we want to warn the reader about the execution context of the *createConceptSet.* This method is invoked on a dummy instance of the

concept created by using the dummy constructor. Therefore, any instance modifications will not be influential on the final result.

Listing 49 shows the implementation of the materialization method for the concept *Employee*. It is possible to see at line [12] the presence of a choice that makes possible to instantiate the proper type. The corresponding implementation of the Manager class does not have this choice (see Listing 50).

```
[1]  public class Employee extends AbstractConcept {
[2]     ...
[3]     @Override
[4]     public AbstractConcept createConceptSet(
[5]           String aliasName, ResultSet rs) throws SQLException {
[6]        Employee rtnEmpl = null;
[7]        int id = rs.getInt(aliasName + "id");
[8]        String name = rs.getString(aliasName + "_name");
[9]        String surname = rs.getString(aliasName + "_surname");
[10]       String ssn = rs.getString(aliasName + "_ssn");
[11]
[12]       if(rs.getBoolean(aliasName + "_is_manager"))
[13]          rtnEmpl = new Manager(id, name, surname, ssn);
[14]       else
[15]          rtnEmpl = new Employee(id, name, surname, ssn);
[16]
[17]       return rtnEmpl;
[18]    }
[19]     ...
[20] }
```

Listing 49: Implementation of the *createConceptSet* method of the *Employee* class.

```
[1]  public class Manager extends Employee {
[2]     ...
[3]     @Override
[4]     public AbstractConcept createConceptSet(
[5]           String aliasName, ResultSet rs) throws SQLException {
[6]        int id = rs.getInt(aliasName + "id");
[7]        String name = rs.getString(aliasName + "_name");
[8]        String surname = rs.getString(aliasName + "_surname");
[9]        String ssn = rs.getString(aliasName + "_ssn");
[10]       return new Manager(id, name, surname, ssn);
[11]    }
[12]     ...
```

```
[13] }
```

Listing 50: Implementation of the *createConceptSet* method of the *Manager* class.

The two remaining concepts are in the same situation: the materialization method of concept *Team* decides which classes should be instantiated by checking the presence of the *dissolutionDate* attribute, whereas the *createConceptSet* method of the concept *ActiveTeam* does not need to check any attribute. For brevity sake, we have voluntarily omitted the code of these two methods.

Finally, we want to underline that the implementation of this method is not influenced by the position in the hierarchy tree of the concept, whilst it is conditioned by the defining SQL query.

6.1.4 Step 4 – Definition of the Equivalence and Hashing Properties

After defining the materialization logic, it is possible to define if two concepts are equivalent or not. We provide a briefly explanation on that, because there are no differences between this step and the normal programming practices that show how to implement the *equals* and *hashCode* methods. The latter method is used for data caching during the query execution phase.

To define these two methods easily, we need to identify the fields that uniquely identify the concepts' instances into the system. To do that, we should check at the corresponding SQL query together with the database schema.

After a brief inspection, we identify the attribute *person.id* for the concepts *Employee* and *Manager*, and the attribute team.id for the concepts *Team* and *ActiveTeam*. Therefore, the implementation of the *equals* and *hashCode* methods will use only the fields that contain the values of these two attributes. For conciseness, the methods can be implemented in the *Employee* (Listing 51) and *ActiveProject* (Listing 52) classes without writing extra code in the other two subclasses.

```
[1]  public class Employee extends AbstractConcept {
[2]     ...
[3]     @Override
[4]     public boolean equals(Object nextObj) {
[5]        if(nextObj == this)
[6]           return true;
[7]        if(nextObj == null || ! (nextObj instanceof Employee))
[8]           return false;
[9]        Employee nextEmployee = (Employee)nextObj;
[10]       return this.id == nextEmployee.id;
[11]    }
[12]
[13]    @Override
[14]    public int hashCode() {
[15]       int hash = 5;
[16]       hash = 31 * hash + this.id;
[17]       return hash;
[18]    }
[19]    ...
[20] }
```

Listing 51: Implementation of the *equals* method of the *Employee* class.

```
[1]  public class ActiveTeam extends AbstractConcept {
[2]     ...
[3]     public boolean equals(Object nextObj) {
[4]        if(nextObj == this)
[5]           return true;
[6]        if(nextObj == null || ! (nextObj instanceof ActiveTeam))
[7]           return false;
[8]        ActiveTeam nextTeam = (ActiveTeam)nextObj;
[9]        return this.teamID == nextTeam.teamID;
[10]    }
[11]    @Override
[12]    public int hashCode() {
[13]       int hash = 7;
[14]       hash = 31 * hash + this.teamID;
[15]       return hash;
[16]    }
[17]    ...
[18] }
```

Listing 52: Implementation of the *equals* method of the *ActiveProject* class.

6.1.5 Step 5 – Definition of the External Methods

The external methods are getter methods that provide access to the data fetched directly from the database during the execution of the *createConceptSet* method.

Listing 53 shows the external methods of the *Employee* (and *Manager*) class. Each of these methods is annotated with the *ExternalCondition* annotation, where the corresponding database attribute name is specified without the prefix used in the *createConceptSet* method (see Section 6.1.3). For brevity sake, we have voluntarily omitted the definitions of the external methods of the classes *ActiveTeam* and *Team,* because they are very similar to the ones presented below.

```
[1]  public class Employee extends AbstractConcept {
[2]     ...
[3]     @ExternalCondition(columnName="id")
[4]     public int getId() {
[5]        return this.id;
[6]     }
[7]
[8]     @ExternalCondition(columnName="name")
[9]     public String getName() {
[10]       return this.name;
[11]    }
[12]
[13]    @ExternalCondition(columnName="surname")
[14]    public String getSurname() {
[15]       return this.surname;
[16]    }
[17]
[18]    @ExternalCondition(columnName="ssn")
[19]    public String getSsn() {
[20]       return this.ssn;
[21]    }
[22]    ...
[23] }
```

Listing 53: External methods of the *Employee* class.

6.1.6 Step 6 – Definition of the Internal Methods

After defining external methods, it is possible to define internal ones. These methods are out from the database mapping process. Therefore, it is possible working without any constraints in the fully object oriented environment. These methods can be useful to retrieve data from the many-to-many relationship at different aggregation levels. During the implementation, the developer should only annotate the method with the *InternalCondition* annotation by specifying the corresponding estimation of the execution cost. This cost can be estimated by using an arbitrary scale (estimated execution time, amount of memory used, etc.).

As a proof of concept, we implement a couple of simple methods for providing essential information about *groups* and *employees* to the SyQL user.

In Listing 54, the *getTotalEmployees* method returns the total number of members of the working group by collecting data from the *team_member* relation (many-to-many relationship). To do that, the method fetches information from the underlying database in a transparent way for the language user. To speedup the database access, we create an ad hoc data structure inside the database, which can quickly provide the total number of the employees per group for all the possible instances of the *Employee* class by skipping repetitive and expensive aggregations. This support data structure is created by the *conceptSetup* method (lines [12]-[22]). At the end of each query execution, this structure is destroyed by the *conceptDispose* (lines [25]-[36]) method.

To avoid runtime issues, we want to warn the reader that these two methods are invoked on a dummy instance of the concept created by using the dummy constructor. Therefore, any modification of the current instance will not be influential.

```
[1]  public class ActiveTeam extends AbstractConcept {
[2]     ...
[3]     protected static String teamMembersNumberTable =
[4]        "DROP TABLE IF EXISTS syql.team_members_number; " +
[5]        "CREATE TABLE syql.team_members_number as ( " +
[6]           "SELECT t.team AS team_id, COUNT(t.person) AS members_number" +
[7]           "FROM team_member AS t GROUP BY t.team"); " +
[8]        "CREATE INDEX team_members_number_idx " +
[9]           "ON syql.team_members_number USING btree(team_id); ";
[10]
```

```
[11]    @Override
[12]    public void conceptSetup() throws Exception {
[13]       Connection conn = null;
[14]       try {
[15]          conn = ConnectionTools.getConnection(
[16]                Engine.getInstance().getDbName());
[17]          conn.createStatement().execute(teamMembersNumberTable);
[18]       }
[19]       finally {
[20]          try { conn.close(); } catch(Exception e) { }
[21]       }
[22]    }
[23]
[24]    @Override
[25]    public void conceptDispose() throws Exception {
[26]       Connection conn = null;
[27]       try {
[28]          conn = ConnectionTools.getConnection(
[29]                Engine.getInstance().getDbName());
[30]          conn.createStatement().execute(
[31]                "DROP TABLE syql.team_members_number;");
[32]       }
[33]       finally {
[34]          try { conn.close(); } catch(Exception e) { }
[35]       }
[36]    }
[37]
[38]    @InternalCondition(cost = 10)
[39]    public TeamMembersNumber getTotalEmployees() {
[40]       int employeesNumber = -1;
[41]       Connection conn = null;
[42]       ResultSet rs = null;
[43]       try {
[44]          conn = ConnectionTools.getConnection(
[45]                  Engine.getInstance().getDbName());
[46]          PreparedStatement ps = conn.prepareStatement(
[47]             "SELECT tn.members_number " +
[48]             "FROM syql.team_members_number as tn " +
[49]             "WHERE tn.team_id = ? ;");
[50]          ps.setInt(1, this.teamID);
[51]          rs = ps.executeQuery();
[52]          if(rs.next())
[53]             employeesNumber = rs.getInt(1);
[54]          rs.close();
```

```
[55]          ps.close();
[56]       }
[57]       catch(Exception e) {
[58]          throw new RuntimeException(
[59]             "Problems in getTotalEmployees procedure", e);
[60]       }
[61]       finally {
[62]          try { conn.close(); } catch(Exception e) { }
[63]       }
[64]       return new TeamMembersNumber(employeesNumber);
[65]    }
[66]    ...
[67] }
```

Listing 54: Definition of the *getTotalEmployees*, *conceptSetup*, and *conceptDispose* methods of the *ActualTeam* class.

In Listing 55, the *getSalary* method returns the salary of the employee in the month specified through the method parameter. The parameter is truncated to the month. We did not implement the *conceptSetup* and *conceptDispose* methods, because the fields, on which the selection operation is performed, are already indexed by the *unique* constraint.

```
[1]  public class Employee extends AbstractConcept {
[2]     ...
[3]     @InternalCondition(cost = 10)
[4]     public EmployeeSalary getSalary(SimpleDateTime sdt) {
[5]        int employeeSalary = -1;
[6]        Connection conn = null;
[7]        ResultSet rs = null;
[8]        try {
[9]           conn = ConnectionTools.getConnection(
[10]                  Engine.getInstance().getDbName());
[11]          PreparedStatement ps = conn.prepareStatement(
[12]             "SELECT amount " +
[13]             "FROM montly_salary " +
[14]             "WHERE person = ? AND year = ? AND month = ? ;");
[15]          ps.setInt(1, this.id);
[16]          ps.setInt(2, sdt.getYear());
[17]          ps.setInt(3, sdt.getMonth());
[18]          rs = ps.executeQuery();
[19]          if(rs.next())
[20]             employeeSalary = rs.getInt(1);
[21]          rs.close();
```

```
[22]          ps.close();
[23]       }
[24]       catch(Exception e) {
[25]          throw new RuntimeException(
[26]             "Problems in getTotalEmployees procedure", e);
[27]       }
[28]       finally { try { conn.close(); } catch(Exception e) { } }
[29]       return new EmployeeSalary(employeeSalary, this);
[30]    }
[31]    ...
[32] }
```

Listing 55: Definition of the *getSalary* method of the *ActualTeam* class.

Both the internal methods return complex types (*TeamMembersNumber*, *EmployeeSalary*). In the next section, it is shown how to enable the query engine to perform the fuzzy evaluation on these measures.

6.2 How to Implement New Fuzzy Evaluators

In this section, we provide the necessary notions to implement Java types that are evaluable by fuzzy logic. In the previous section, we have shown how an internal method can return a complex type useful for further evaluations. Now, we show operationally the necessary steps to implement two classes: *TeamMembersNumber* and *EmployeeSalary*.

The types evaluable by fuzzy logic must implement the interface *it.unibz.syql.fuzzy.IFuzzableType*. This interface defines the following two methods:

- `public Object getFuzzyProperty();`
- `public IFuzzyEvaluator getFuzzyEvaluatorInstance().`

The former returns the measure to evaluate, whereas the latter returns fuzzy evaluator that will be used to evaluate the measure.

6.2.1 Step 1 – Definition of the Measure and of the Evaluation Logic

The first step requires to define two aspects:

- the measure evaluated by the fuzzy evaluator;
- the fuzzy evaluator used for evaluating such a measure.

For the class *EmployeeSalary* (see Listing 56), the measure is the *employeeSalary* field (see line [12]), whereas the fuzzy evaluator is chosen according to the type of employee (see lines [17]-[20]). Indeed, a salary of a manager needs to be evaluated in a different way compared to a salary of a regular employee.

```
[1]  public class EmployeeSalary implements IFuzzableType {
[2]     int employeeSalary;
[3]     protected Employee employee;
[4]
[5]     public EmployeeSalary(int employeeSalary, Employee employee) {
[6]        this.employeeSalary = employeeSalary;
[7]        this.employee = employee;
[8]     }
[9]
[10]    @Override
[11]    public Object getFuzzyProperty() {
[12]       return this.employeeSalary;
[13]    }
[14]
[15]    @Override
[16]    public IFuzzyEvaluator getFuzzyEvaluatorInstance() {
[17]       if(this.employee instanceof Manager)
[18]          return new ManagerSalaryFuzzyEvaluator();
[19]       else
[20]          return new EmployeeSalaryFuzzyEvaluator();
[21]    }
[22] }
```

Listing 56: Definition of the *EmployeeSalary* class.

In Listing 57, the *TeamMembersNumber* class defines the *employeesNumber* field as a measure (see line [9]), and the *TeamMembersNumberFuzzyEvaluator* class as a single fuzzy evaluator (see line [14]). In this particular case, we used a single fuzzy evaluator because, in both cases (active and dissolved), there are no differences in the evaluation of the number of people of a team.

```
[1]  public class TeamMembersNumber implements IFuzzableType{
[2]     protected int employeesNumber;
[3]     public TeamMembersNumber(int employeesNumber) {
[4]        this.employeesNumber = employeesNumber;
[5]     }
[6]
[7]     @Override
[8]     public Object getFuzzyProperty() {
[9]        return this.employeesNumber;
[10]    }
```

```
[11]
[12]    @Override
[13]    public IFuzzyEvaluator getFuzzyEvaluatorInstance() {
[14]       return new TeamMembersNumberFuzzyEvaluator();
[15]    }
[16] }
```

Listing 57: Definition of the *TeamMembersNumber* class.

6.2.2 Step 2 – Definition of the Fuzzy Evaluator

After defining the measures and the evaluation logics, it is possible to define the fuzzy evaluators. To define a fuzzy evaluator, we need to know the following information:

- the linguistic variables that we want to adopt;
- the membership function for each linguistic variable.

A fuzzy evaluator must implement the interface *it.unibz.syql.fuzzy.IFuzzableType*, this class defines the following two methods:

- ***public*** *String[] getFuzzyLiteralList();*
- ***public*** *Double getFuzzyScore(String fuzzyLiteral,*
  Object fuzzyProperty).

The former returns a collection containing the linguistic variables. The latter returns the value of the membership function that is identified by the first parameter (*fuzzyLiteral*) for the measure specified by the second parameter (*fuzzyProperty*).

In order to complete our example, we have to define three fuzzy evaluators: *ManagerSalaryFuzzyEvaluator*, *EmployeeSalaryFuzzyEvaluator*, and *TeamMembersNumberFuzzyEvaluator*.

For brevity sake, we implemented only the first one (see Listing 58), because the other two are very similar. For all the evaluators, we defined three linguistic variables: *high, medium* and *low* (see lines [2] and [6]). As membership functions, we used the same functions defined in Formula 8 on page 108. These functions have been parametrized with the following values (see lines [11]-[12]):

- minimum value = 2000€;
- maximum value = 9000€;
- average value = 5000€.

```
[1]  public class ManagerSalaryFuzzyEvaluator implements IFuzzyEvaluator {
[2]     private String [] fuzzyLiterals = {"low", "medium", "high"};
[3]
[4]     @Override
[5]     public String[] getFuzzyLiteralList() {
[6]        return fuzzyLiterals;
[7]     }
[8]     @Override
[9]     public Double getFuzzyScore(String fuzzyLiteral,
[10]          Object fuzzyProperty) throws IllegalArgumentException {
[11]       return UtilFuzzyEvaluator.getMinAvgMaxFuzzyScore(
[12]              fuzzyLiteral, fuzzyProperty, 2000, 9000, 5000);
[13]    }
[14] }
```

Listing 58: Definition of the *ManagerSalaryFuzzyEvaluator* class.

By customizing the *getFuzzyScore* method, a user can specify the proper shapes for the membership functions. This is useful to interpret different measure types.
Finally, after this final step. It is possible to use the SyQL query engine to execute query like the one in Listing 59. The query below returns the data of all the manager that are leading a small team having a high salary in the last month.

```
[1]  FROM ActiveTeam t, Manager m
[2]  WHERE   t.getLeader() = m AND
[3]          m.getSalary() is HIGH AND
[4]          t.getEmployeesNumber(TODAY) is LOW
[5]  SELECT m.getName(), m.getSurname(), m.getSSN();
```

Listing 59: SyQL query using the *MyFooCompany* database.

7 Performance Evaluation

In this Chapter, we present the performance evaluation of SyQL by showing the benefits and the limits of the current approach.

In the first part of the Chapter, we focus on the time performance of the query engine by executing ad hoc queries in a controlled environment. In this way, we show how the system scales up to guarantee an acceptable level of performance necessary to deliver to the final user an adequate data browsing service. During this evaluation, we show the drawbacks of this approach, hence we also present a solution to overcome the current limits, which are emerged during the case studies presented in Section 9 .

In the second part, we present the evaluation of the programming productivity of SyQL by estimating the Capers Jones' Index. This index estimates the number of lines of code needed to implement a function point.

7.1 Abstraction Layer Assessment

In this section, we start providing a description of the methodology and of the infrastructure that we have adopted to provide an objective evaluation of the performance of the query engine. Then, we present the result of the two assessments. In the first assessment, the number of concepts of a query was progressively increased. In the second one, we have observed the effects varying the total number of CPU cores available by keeping the same query. Finally, we provide a discussion, where we propose a solution to the current performance limitations that affect the SyQL Engine under certain conditions.

7.1.1 Testing Methodology and Infrastructure

The testing methodology, that we have adopted, it is oriented to measure the time spent by the different components of the query engine in completing their tasks. To measure the times, we made the difference between the values returned by the Java Runtime method *System.currentTimeMillis()* before and after the task execution. During the query execution, we have measured the time spent by the components to complete the following tasks:

- query parsing;
- executing methods reflection;
- converting the user *where* formula to CNF;
- executing the SQL query against the DBMS;
- evaluating the internal conditions;
- executing selection and group operations for computing the final result set.

We want to evidence that we discard the time necessary to send the query result back to the client, because this measure is susceptible to several factors that are not under our control, like network traffic and connection latency.

In addition, we have also measured the maximum memory usage of the query engine during the query execution by varying the number of concepts. To measure the space occupancy, we have done the difference between the numbers of bytes returned by the methods *Runtime.totalMemory()* and *Runtime.freeMemory()*. We measured the memory occupancy at the end of each of the phases listed above, and we noticed that the peak was always at the end of the last one.

To limit the effects of the measuring noise, each measure was repeated five times, hence the final value of each one was computed as the average of the repetitions discarding the maximum and the minimum.

As testing infrastructure, we used a Sun Fire X4600 M2 as a testing machine. The configuration of this machine was the following:

- CPU(s): 8 x Dual-Core AMD Opteron(tm) Processor 8220 (clock 2.8Ghz, 128KB of L1 cache, 2MB of L2 cache);
- Memory: 2 x banks of 16GB DIMM Synchronous 333 MHz (32GB in total);
- Operating System: Ubunt 8.04.3 LTS, kernel 2.6.24-17-server;
- Hard Drive(s): 2 x SCSI Disk Seagate ST914602SSUN146G (Mirroring mode);
- Java Virtual Machine:

  java version 1.6.0_06
  Java(TM) SE Runtime Environment (build 1.6.0_06-b02)
  Java HotSpot(TM) 64-Bit Server VM (build 10.0-b22, mixed mode)
- DBMS: version string = 'PostgreSQL 8.3.8 on x86_64-pc-linux-gnu'.

To avoid network delays, we have installed the SyQL engine and the PostgreSQL instance on the same machine. We run the experiments on the data set used in the first case study (in Section 9.2 on page 204). In this way, the performance assessment is done on a real world database, and this makes stronger the value of our experimental results.

This medium size machine makes possible to assess the performance of our engine without the effects of the memory paging on a real multiprocessor machine. During the experiments, the machine was completely free from other processes except for system ones, and the processes' priorities have not been modified. We launched the set of queries by a shell script, and we printed the measures on a text file.

7.1.2 Number of Concepts

In this experiment, we observe the times and the memory usage (dependent variables) varying the number of concepts declared into a SyQL query (independent variable).

The objective of this experiment is to show how the performances are affected, when more complex queries are sent to the query engine. During the experiments, the query engine used all the 16 cores available, and the JVM was initialized with the Xmx parameter to 10,000 megabytes.

The SyQL queries that we have repetitively executed on the system are listed in the following:

```
[1]  FROM    Class c1
[2]  WHERE   c1.getLOC() <> -1
[3]  SELECT  c1.getFullName(), c1.getLOC();
[4]
[5]  FROM    Class c1, Class c2
[6]  WHERE   c1.getLOC() <> -1 AND c2.getLOC() <> -1 AND
[7]          c1.getFullName() = c2.getFullName()
[8]  SELECT  c1.getFullName(), c1.getLOC(), c2.getFullName(), c2.getLOC();
[9]  ...
```

```
[10] ...
[11] FROM    Class c1, Class c2, Class c3, ... , Class c7
[12] WHERE   c1.getLOC() <> -1 AND c2.getLOC() <> -1 AND
[13]         c3.getLOC() <> -1 AND ... AND c7.getLOC() <> -1
[14]         c1.getFullName() = c2.getFullName() AND
[15]         c2.getFullName() = c3.getFullName() AND
[16]         ... AND
[17]         c6.getFullName() = c7.getFullName()
[18] SELECT  c1.getFullName(), c1.getLOC(), c2.getFullName(), c2.getLOC();
```

Listing 60: Testing queries for performance assessment.

This set of queries has the property to increase proportionally with the number of declared concepts the number of the following elements:

- the length of query text;
- the quantity of data fetched from the data warehouse without increasing the number of fetched records;
- the quantity of external conditions evaluated by the PostgreSQL instance (e.g. *c1.getFullName() = c2.getFullName()*);
- the quantity of internal conditions evaluated by the SyQL query engine (e.g. *c1.getLOC() <> -1*);
- the quantity of expressions specified in *SelectClause.*

Table 12 presents the result of this experiment. It is possible to see that the CNF Conversion Time is not affected by the number of concepts. This happens because all the *user formula*(s) are already in CNF format. Therefore, the converter is not proportionally tested. This can be a lack of the experiment, but in general, the fraction of time spent on this phase is so short that this component does not affect the overall execution time in a significant way. The same happens for the parser and for the reflection module.

	Number of Concepts						
	1	**2**	**3**	**4**	**5**	**6**	**7**
Parse Time [sec]	0.001	0.001	0.001	0.001	0.001	0.002	0.001
Reflection Time [sec]	0.002	0.003	0.005	0.005	0.006	0.007	0.008
CNF Conversion Time [sec]	0.001	0.001	0.001	0.001	0.001	0.001	0.001
DBMS Query Execution Time [sec]	3.079	6.904	10.050	13.447	16.168	19.164	22.418
Internal Conditions Evaluation Time [sec]	0.665	1.290	1.851	2.106	2.560	2.973	3.404
Selection and Group Execution Time [sec]	0.013	0.044	0.047	0.061	0.051	0.059	0.074
Overall Execution Time [sec]	**3.767**	**8.296**	**12.051**	**15.599**	**18.840**	**22.352**	**25.899**
SQL Query Execution Time [sec]	2.757	5.782	8.696	11.613	14.551	17.505	20.376
Maximum Memory Usage [MB]	63.225	112.478	173.824	238.809	205.182	219.869	293.276
Fetched Database Records	6978	6978	6978	6978	6978	6978	6978

Table 12: Query processing times versus number of concepts.

In Figure 24, we plot the result of this experiment. The graph shows that the DBMS Query Execution and the Internal Conditions Evaluation take longer than the other phases (more than 99% of the entire execution time). Therefore, these are the most influential on the overall performance, whereas the contribution of the others is insignificant. The relation between these two times and the number of concepts is proportional, and this is good a thing, because it means that the system can scale pretty well if the number of database record to fetch does not become huge. When it happens the query engine may have problems to compute the query result due to the space overhead given by the Java runtime, and it can run out of memory. We will discuss this problem in details in Section 7.1.4 .

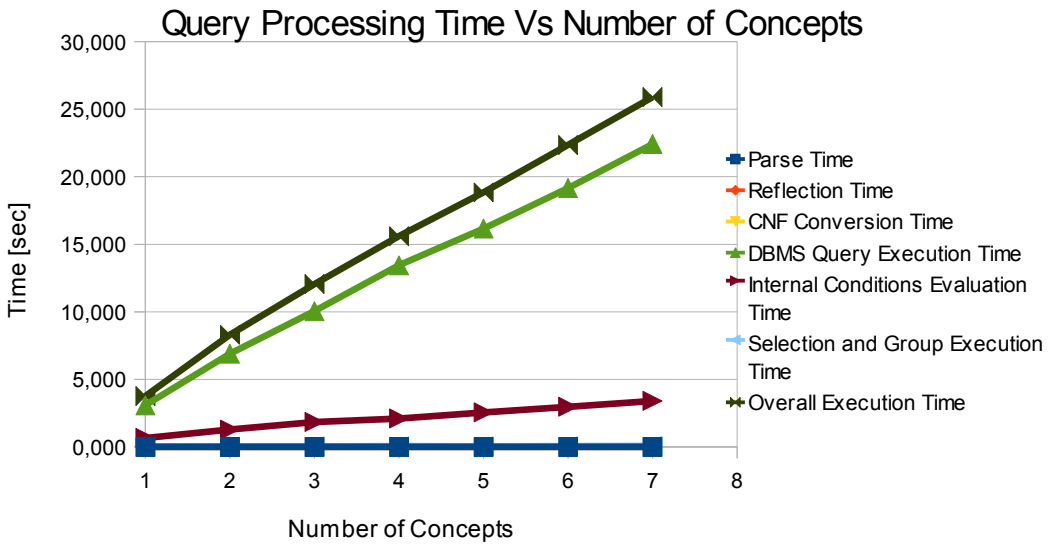

Figure 24: Query processing times versus number of concepts (global).

Figure 25 is a zoom of Figure 24, and we can see that the only notable contribution is given by the *Selection and Group* phase, where the final result is computed. We can see that the time and the number of concepts are not perfectly proportional due to the measuring noise coming from the non-determinism of the operating system scheduler. This noise becomes evident, when we try to measure times that have the same order of magnitude of the operating system tick.

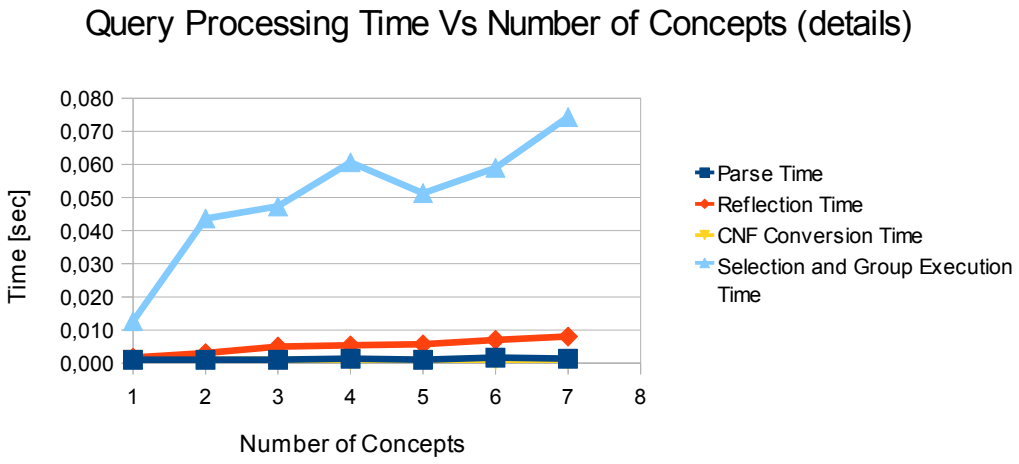

Figure 25: Query processing times versus number of concepts (detailed).

By using paralleling programming, the *Interval Conditions Evaluation* phase is easy to speed up, contrariwise the *DBMS Query Execution* phase is more difficult to make run faster, because it is not completely dependent from us. To prove this, we measured the time necessary to execute the SQL code generated by SyQL against our PostgreSQL instance. In this way, we can measure if the bottleneck is generated from either the SyQL engine or the PostgreSQL instance. In Figure 26, we have plotted the two execution times and the difference of them, the SyQL average overhead is equal to 11.56%. We consider this value of overhead acceptable for materializing relational data into collections of objects. To remove this bottleneck, we suggest to use standard optimization techniques for the SQL statements that define the concepts. In particular, we suggest to index all the fields that are mapped with concept methods, because they are usually used as part of join conditions. In this experiment, all the SQL joins are performed by using the merge join algorithm.

We want to stress that the adoption of the SQL optimization techniques is not the main objective of this dissertation.

Figure 26: Comparison between the SyQL DBMS Execution Time and the (plain) SQL execution time.

We conclude the discussion of this experiment by showing the maximum memory usage reached during the query execution versus the number of concepts (see Figure 27). We can see that there is a pseudo-proportionality relation between the memory usage and the number of concepts. The jump between x-values 4 and 5 is caused by the increasing number of calls to the garbage collector, which are caused by the increasing request of memory heap space.

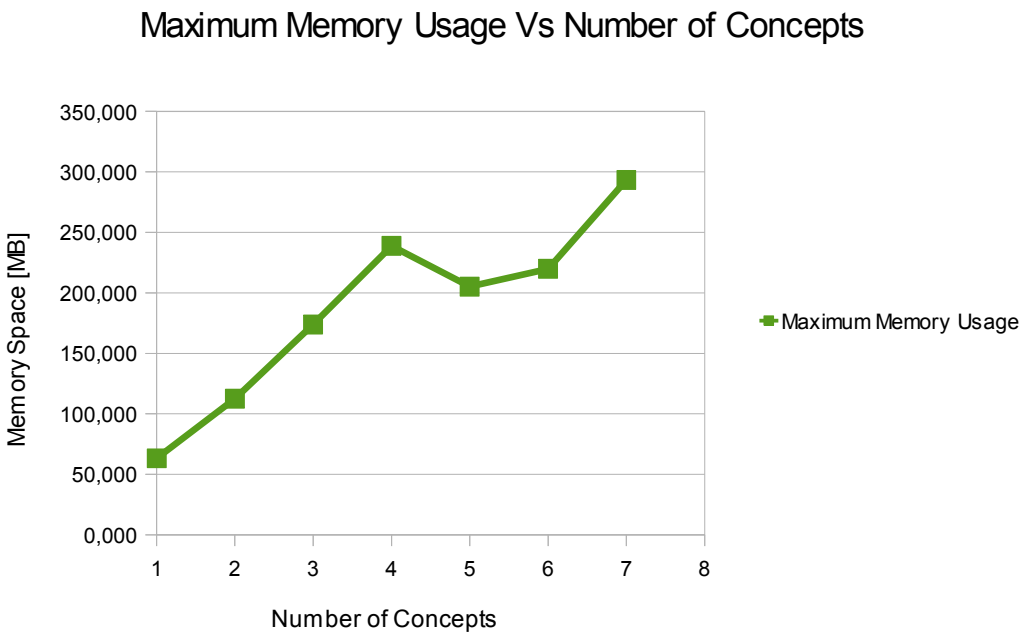

Figure 27: Maximum memory usage versus number of concepts.

7.1.3 Number of Cores

The objective of this experiment is to show the system scaling across multiple CPU cores. In this experiment, we observe the execution times (dependent variables) varying the number of available core (independent variable). The testing SyQL query adopted is the last one defined in Listing 60. This query declares seven concepts; this is important because it generates a pretty large workload that can make the randomness of the Operating System scheduler less influent on the timing measures. Table 13 reports the result of the experiment. For each measure, we also computed the speedup. The speedup is defined as the ratio between the execution time of the

sequential program (number of cores equal to 1) and the execution time of the parallel version of the same program (number of cores greater than 1).

		Number of Cores				
		1	**2**	**4**	**8**	**16**
Parse Time	**[sec]**	0.003	0.002	0.002	0.003	0.001
	Speedup	**1.000**	**1.143**	**1.333**	**1.000**	**2.000**
Reflection Time	**[sec]**	0.012	0.011	0.012	0.011	0.008
	Speedup	**1.000**	**1.091**	**1.029**	**1.059**	**1.500**
CNF Conversion Time	**[sec]**	0.001	0.001	0.001	0.001	0.001
	Speedup	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**
DBMS Query Execution Time	**[sec]**	24.520	24.835	24.719	24.821	22.418
	Speedup	**1.000**	**0.987**	**0.992**	**0.988**	**1.094**
Internal Conditions Evaluation Time	**[sec]**	15.103	8.634	5.861	4.707	3.404
	Speedup	**1.000**	**1.749**	**2.577**	**3.209**	**4.437**
Selection and Group Execution Time	**[sec]**	0.108	0.117	0.099	0.100	0.074
	Speedup	**1.000**	**0.926**	**1.091**	**1.087**	**1.457**
Overall Execution Time	**[sec]**	39.912	33.807	30.436	29.642	25.899
	Speedup	**1.000**	**1.181**	**1.311**	**1.346**	**1.541**

Table 13: Query processing times versus number of cores used.

In Figure 28 on page 154, timing data are plotted, whereas Figure 29 shows the speedups. As in previous experiment, the two longest phases are the *DBMS Query Execution* and the *Internal Condition Evaluation*. Therefore, they are the most interesting for studying the speedup. Indeed, Figure 29 omits the other phases.

The *Internal Condition Evaluation* phase has been explicitly implemented for taking benefits from multithread programming (the 82.6% of the code runs in parallel). In fact, it shows an effective speedup, which helps to reduce the overall execution time of a query. We can see that the speedup is stronger at the beginning (number of core equals to 1-2) than at the end. This is caused by the inter-processor communication overheads, which are lower inside the same CPU (we used dual core CPUs) than on the bus. The result obtained is coherent with the Amdahl's Law (see Formula 16).

$$Speedup = \frac{1}{(1-P)+\frac{P}{N}}$$

$$P = (1-\frac{1}{Speedup})\cdot(\frac{N-1}{N})$$

$$Speedup_{max} = \lim_{N\to\infty} \frac{1}{(1-P)+\frac{P}{N}} = \frac{1}{(1-P)}$$

Where
P = percentage of parallel code
N = number of processors

Formula 16: Amdahl's Law

In Table 14, we show the results of the application of the Amdahl's Law for computing the percentage of parallel code executed (P), and the maximum theoretical speedup achievable with an infinite number of CPU. We can see that the *Overall Execution Time* is close to the maximum theoretical speedup computed by using the Amdahl's law. It means that if we want to run faster, we have to increase the percentage of parallel code by modifying the *DBMS Query Execution* phase.

		Number of Cores 16
DBMS Query Execution Time	**[sec]**	22.418
	Speedup	1.094
	%Parallel Code	9.1%
	Maximum theoretical speedup	1.101
Internal Conditions Evaluation Time	**[sec]**	3.404
	Speedup	4.437
	%Parallel Code	82.6%
	Maximum theoretical speedup	5.757
Overall Execution Time	**[sec]**	25.899
	Speedup	1.541
	%Parallel Code	37.4%
	Maximum theoretical speedup	1.599

Table 14: Amdahl's law application.

Indeed, the *DBMS Query Execution* does not show a significant speedup, for a twofold reason: firstly, this phase is implemented without using multithread programming (only 9.10% of the code runs in parallel), secondly PostgreSQL does not support multiprocessing. In section Section 7.1.4 , we try to address this problem by proposing an alternative execution strategy for the SQL queries by taking benefits

from multiprocessing. This strategy will improve the scaling and the overall performances in time and space of the query engine.

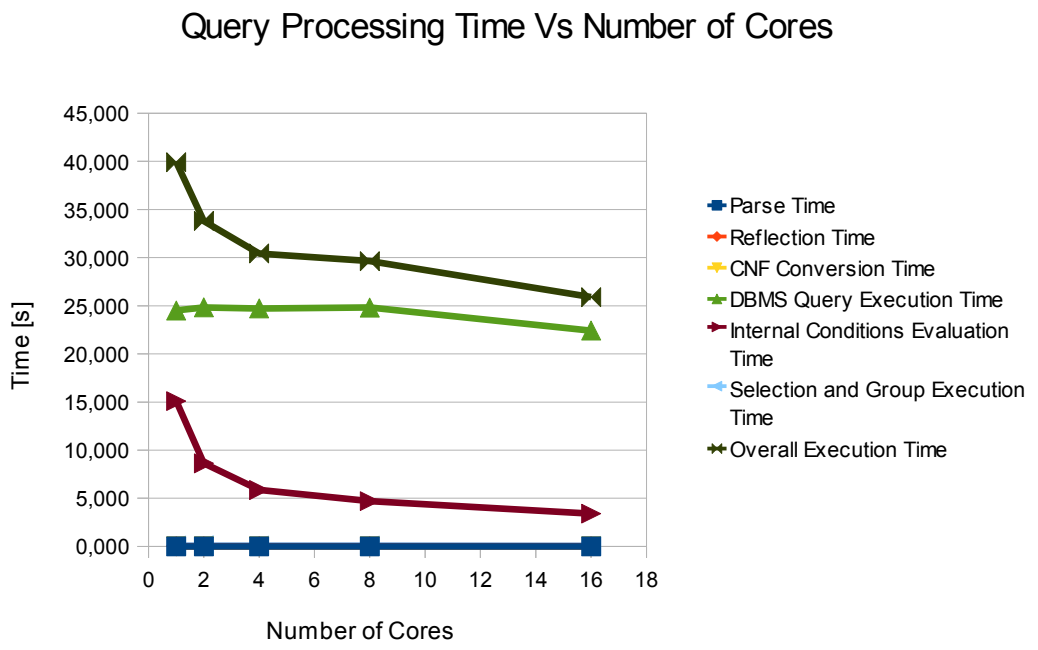

Figure 28: Query processing times versus number of cores used.

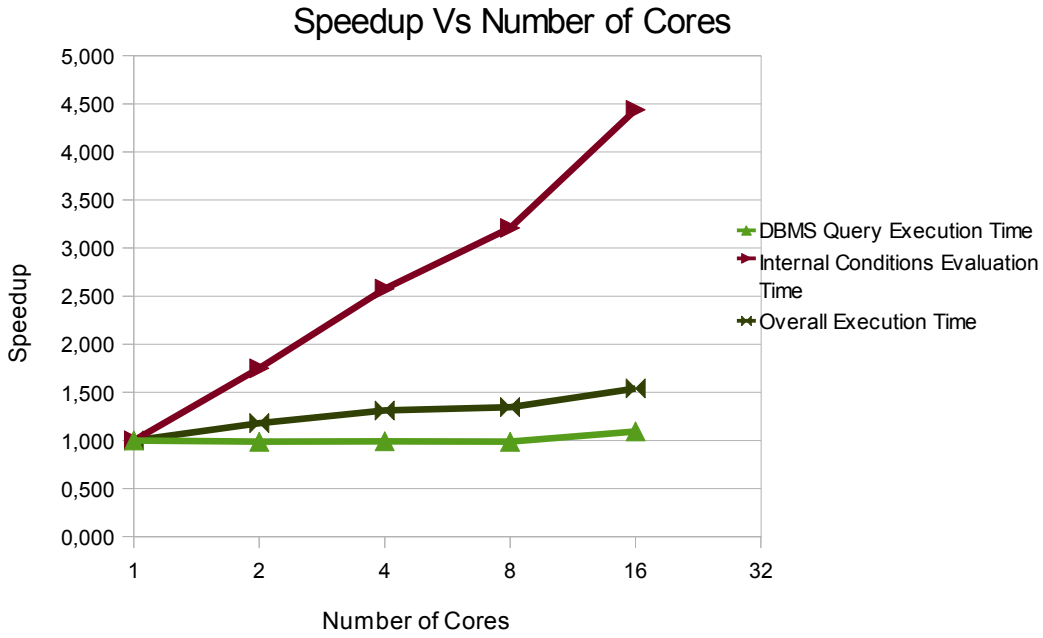

Figure 29: Speedup versus number of cores used.

7.1.4 Discussion

In previous Sections, we identify that the current bottleneck is the *DBMS Query Execution* phase, when the SQL query is executed, and the data are fetched into the query engine. Figure 26 shows that the 88.44% of the time is spent for executing the SQL query, and the time is proportional to the quantity of data fetched. This can be a serious issue when we need to perform Cartesian products between SQL tables. This happens when there are no external join conditions. In Listing 61, we perform a join by using the latest LOC value, which is an internal attribute, to get all pairs of classes with the identical LOC value. This query fetches 6978^2 records from the PostgreSQL instance, and this number of records can make the execution not affordable on a desktop machine due to insufficient memory. On our machine, it takes one hour, 12 minutes and 37 seconds to complete with a maximum memory usage of 22.2 GB.

```
[1]   FROM Class c1, Class c2
[2]   WHERE c1.getLOC() = c2.getLOC()
[3]   SELECT c1.getFullName(), c1.getLOC(), c2.getFullName(), c2.getLOC();
```

Listing 61: SyQL query without external join conditions.

Summarizing, our solution should achieve two goals:

- make the *DBMS Query Execution* phase as parallel as possible;
- minimize the number of record fetched from the underlying DBMS.

The idea that we propose is to separate the SQL query in a set of multiple SQL queries by following certain rules.

The general rule is that every SyQL query concept is translated into a SQL query by taking its conditions that appear into one or more external CNF clauses (of the *WhereClause*). In this way, we can spread the computation across multiple threads, hence they open distinct sessions against the PostgreSQL database. In this way, we can overcome the PostgreSQL limit that bounds the available CPU(s) per session to one. Each thread will populate a distinct collection to maximize the parallelism.

We provide another rule, which is called *splitted query execution*, that is an exception to the general one. This rule is triggered when there is one or more join conditions into an external CNF clause. When it happens, two or more SyQL concepts are

translated into a single SQL query by taking their join conditions and the eventual others conditions that appear into one or more external CNF clauses. In this way, the indexing structure of the underlying database is reused, and fewer records are fetched by the query engine. This last rule will be probably enhanced during its implementation by taking into account additional information like the concept size, etc.

When all the data are fetched from the PostgreSQL database, it is possible to start evaluating the internal conditions by performing an inner join between the different collections of fetched data. In this way, the quantity of parallel code should increase and the required memory space should decrease.

The proposed solution has been designed to overcome one limits discovered during the beta testing of SyQL on real world case studies, which are presented in Section 9 .

7.2 Language Assessment

In this section, we try to estimate the value of the Capers Jones' Index [74] (already introduced in Section 3.3 on page 36). We want to warn the reader that this estimation is conducted in an informal way, because the estimation requires to know the average number of lines of code of the language under evaluation necessary to implement a function point. To produce a reliable average number of line of code, we should have a large set of SyQL queries that implement a wide set of function points, and we have not. The only purpose of this assessment is to quantify approximately the SyQL language level by comparing with SQL.

Example 5 on page 22 is taken as reference, because it is a real world example solved by using both languages. By counting the lines of code as they are written, the SQL code in Listing 8 is 55 lines (discarding blank lines), whereas the SyQL code in Listing 9 is four lines. We can identify inside the query four function points:

- determining of the last upload_id;
- retrieving of today values of lines of code per class;
- retrieving of yesterday values of lines of code per class;
- building of final result-set by selecting only the classes that present a variation of lines of code different from 0.

The average value of lines of code per function point for the SQL query is equal to

13.75, which is very close to the value reported by Capers Jones (13) in Table 3 on page 38. It means that our example can be reasonably assumed as the representative of the entire set of SQL queries, hence we can take the index value of SyQL as an acceptable estimation. Therefore, the average value of lines of code per function point is equal to 1.33 (4/3).

By using the Capers Jones' formula (see Formula 17), we get a language level of 252.625.

$$language\ level = \frac{337.5}{Avg\ LOC / FP} - 0.5$$

Formula 17: Capers Jones' index definition.

Compared with the other level values, this value is dramatically high. However, it is not a surprise, because SyQL is a very domain specific language that uses a library that provides type and value abstraction. Finally, in the computation of the language level, we voluntarily omitted the lines of code of the Java functions used in the query, because the user does not implement any of them.

8 Limitations

In this Chapter, we present and discuss the currently known limits of the SyQL query engine. For each of them, we propose a possible solution to overcome the limit. The design of these solutions is taken as an opportunity to add extra features that make SyQL more usable by the final users.

We present a couple of limitations: the first one is caused by the domain specificity of SyQL; the second one depends by the design of the language, which does not allow to reuse the query results.

Summarizing, SyQL is only a domain specific – data manipulation language that has not the constructs of a general programming language. In addition, SyQL does not support the language closure, which allows the user to define a new concept at runtime by using the already existing concepts.

8.1 Only Domain Specific – Data Manipulation Language

The excessive specificity of a domain specific language has been already stated by Canfora *et al.* [27]. In 1998, they implemented a query language to automatize source code analysis by limiting the human intervention. To make it interoperable with the existing application, they developed a foreign language interface. This interface allows the developer to use this system directly from C code.

The solution, that we are going to show, is similar to the one described by Canfora *et al.* [27]: we have designed an extension of the Java language by using a code preprocessor before compiling the source code. In this way, the java compiler does not require any modification.

In addition, by implementing this feature, the user can run SyQL queries either on the client-side or server-side transparently. To do that, we need to implement the *splitted query execution* before (described in Section 7.1.4 on page 155), because it stores the results in separated collections before joining them together. This can be useful to split the load between client and server.

In Listing 62, we present a piece of code where the user specifies a query that is partially run on the server, whereas the final result is computed on the client. In this listing, we declare a java method that accepts as a parameter a list of classes. This

collection becomes a query concept (line [15]) that is joined with the collection containing the yesterday existing methods, which has been determined by checking the LOC value of yesterday (line [17]).

```
[1]  import java.util.ArrayList;
[2]  import java.util.List;
[3]  import it.unibz.syql.concepts.Class;
[4]  import it.unibz.syql.concepts.Method;
[5]
[6]  public class Foo {
[7]    /**
[8]     * This method returns the yesterday total number
[9]     * of existing methods for the specified classes.
[10]    * @param classes Classes of interest.
[11]    * @return The total number of yesterday methods.
[12]    */
[13]    public static int getYesterdayClassNOM(List<Class> classes) {
[14]       List<List<Integer>> result =
[15]          <?   FROM $classes c, Method m
[16]               WHERE   m.getDefiningClass() = c AND
[17]                       m.getLOC(YESTERDAY) <> -1
[18]               SELECT COUNT(m);
[19]          ?>;
[20]       return result.get(0).get(0);
[21]    }
[22] }
```

Listing 62: Sample code written by using our extended-Java (theoretically).

The preprocessing translates the previous piece of code into pure Java code shown in Listing 63. The query specified between the two escape sequences ('<?', '?>') is converted into a string, and the identifier of the collection of classes is converted into a number followed by a prefix (line [16]), then the reference of the external collection is passed as argument (line [19]). We want to stress that this conversion is always possible, because neither variables nor import declarations are added to the produced Java code. Moreover, this translation can be done on the fly by using a finite state machine, because it is not necessary to check if an expression is balanced or not. Therefore, the process is very fast.

During the translation process, it is possible to check if the query is syntactically correct or not by using the SyQL parser on the extracted query before producing the

Java code. This can prevent runtime errors by providing a useful support to the developer. The compilation performance will not be affected by this check, because the query text to parse is usually very small, and the SyQL parser performance are good (see Table 13 on page 152).

```
[1]  import java.util.ArrayList;
[2]  import java.util.List;
[3]  import it.unibz.syql.concepts.Class;
[4]  import it.unibz.syql.concepts.Method;
[5]
[6]  public class Foo {
[7]    /**
[8]     * This method returns the yesterday total
[9]     * number of methods for the specified classes.
[10]    * @param classes Classes of interest.
[11]    * @return The total number of yesterday methods.
[12]    */
[13]    public static int getYesterdayClassNOM(List<Class> classes) {
[14]       List<List<Integer>> result =
[15]          it.unibz.syql.engine.Engine.getInstance().executeQuery(
[16]             "FROM $0 c, Method m " +
[17]             "WHERE m.getDefiningClass() = c " +
[18]                "AND m.getLOC(YESTERDAY) <> -1 " +
[19]             "SELECT COUNT(m) ;", classes);
[20]       return result.get(0).get(0);
[21]    }
[22] }
```

Listing 63: Translation into pure Java code of Listing 62 (theoretically).

To execute this query, the local query engine splits the query into two sub-queries: the first one (Listing 64) will be executed on the server-side, the second one (Listing 65) will be executed locally by using the result of the first query as a concept.

```
[1]  FROM    Method m
[2]  WHERE   m.getLOC(YESTERDAY) <> -1
[3]  SELECT  m;
```

Listing 64: SyQL query executed on server-side (theoretically).

To split the conditions across queries, we use the same mechanism described in Section 5.5.3 on page 100, but instead of producing an SQL query, here we produce a SyQL query, which is sent to the server to be processed.

```
[1]   FROM    $classes c, $serverResult m
[2]   WHERE   m.getDefiningClass() = c
[3]   SELECT  count(m);
```

Listing 65: SyQL query executed on client-side (theoretically).

In conclusion, we want to remark that we design this solution to avoid sending client data to the server, since this can be a privacy issue, and it can create an excessive load on the central SyQL query engine. However, if it is necessary, the system can work in fat-server mode by processing also the client data.

8.2 No Closure

The possibility to reuse the result of a query can be useful to avoid implementing new concept methods. We treat this problem in Section 9.4.2 on page 222 by showing with a real world example how the closure can help the user to write complex SyQL query. By using the *closure*, a user can break down an information retrieval problem in smaller pieces by helping the query engine to speedup the entire execution process.
The implementation of the closure, subordinated to the implementation of the *splitted query execution* feature (described in Section 7.1.4 on page 155), can lead the query engine to implement an additional abstraction layer over the existing concepts. Therefore, the type abstraction can be enhanced with the introduction of the new *superconcept*(s). A *superconcept* is a concept that is defined from a SyQL query instead of a SQL one. The *closure* and the *superconcept*(s) are similar: the former is the runtime definition of the latter. The *closure* allows the user to define a new concept during the query writing, whereas the *superconcept* is statically defined. By defining a *superconcept*, it is possible to define additional methods that can help the user to get data difficult to retrieve and to compute. These additional methods can return values that are evaluable by using fuzzy logic. In this way, the abstraction is again on types and on values. On the contrary, by using the *closure*, the abstraction is only on types.

The *splitted query execution* feature together with the closure can increase the range of application of SyQL by providing a new dimension of abstraction.
In Listing 66, we provide a query that prints the variation of lines of code per day, of the methods defined by the classes that have been created during the last week and are still existing. This query is written without using the closure.

```
[1]   FROM Class c, Method m, Chronon chr
[2]   WHERE   c.getLOC(TODAY - 1 'WEEK') = -1 AND
[3]           c.getLOC(TODAY) <> -1
[4]           m.getDefiningClass() = c AND
[5]           m.getLOC(chr.getDate()) <> -1 AND
[6]           chr.getDate() > TODAY - 1 'WEEK' + 1 'DAYS'
[7]   SELECT  m.getFullName(), chr.getDate(),
[8]           m.getLOC(chr.getDate()).getValue() -
[9]             m.getLOC(chr.getDate() - 1 'DAY').getValue();
[10]
[11]
```

Listing 66: SyQL query written without using the closure.

Listing 67 shows an equivalent query to the one written in Listing 66 by using the closure. It is possible to see that all the logic necessary to get the classes added during the last week is aggregated into the nested query (lines [1]-[4]).

```
[1]   FROM    (  FROM Class c,
[2]              WHERE   c.getLOC(TODAY - 1 'WEEK') = -1 AND
[3]                      c.getLOC(TODAY) <> -1
[4]              SELECT  c ) AS lastWeekCreatedClasses,
[5]           Method m, Chronon chr
[6]   WHERE   m.getDefiningClass() = lastWeekCreatedClasses.c AND
[7]           m.getLOC(chr.getDate()) <> -1 AND
[8]           chr.getDate() > TODAY - 1 'WEEK' + 1 'DAYS'
[9]   SELECT  m.getFullName(), chr.getDate(),
[10]          m.getLOC(chr.getDate()).getValue() -
[11]            m.getLOC(chr.getDate() - 1 'DAY').getValue();
```

Listing 67: SyQL query written by using the closure (theoretically).

In Listing 68, we have rewritten the query above by using the *superconcept* *LastWeekCreatedClasses*, which is defined by using the SyQL query at lines [1]-[4] of Listing 67.

```
[1]  FROM    LastWeekCreatedClasses lwcc, Method m, Chronon chr
[2]  WHERE   m.getDefiningClass() = lwcc.c AND
[3]          m.getLOC(chr.getDate()) <> -1 AND
[4]          chr.getDate() > TODAY - 1 'WEEK' + 1 'DAYS'
[5]  SELECT  m.getFullName(), chr.getDate(),
[6]          m.getLOC(chr.getDate()).getValue() -
[7]             m.getLOC(chr.getDate() - 1 'DAY').getValue();
[8]
[9]
```

Listing 68: SyQL query written using the superconcept (theoretically).

In conclusion, the *closure* and the *superconcept*(s) are two important features that can enable advanced users to write high level queries by using the same set of approaches adopted for SQL. This can reduce the necessity to implement continuously new library features by reducing the number of defect introduced into the SyQL libraries.

8.3 Query Planning

Another limitation of the SyQL engine is the query planning. Query planning is directly related to the query execution performances, because a query can be executed in a multitude of ways, and the task of query planner is to find the optimal execution way.

In the current implementation of the SyQL query engine, we have tried to simplify the query planning process by translating the external condition into SQL, so that the SyQL query engine can rely on the underlying DBMS. This approach requires to translate the condition specified in the where-clause of the SyQL query into a CNF formula, and this conversion can be a limitation. In order to improve the performances of the query engine, an alternative way of executing the query is to use also the DNF conversion instead of the CNF only. In this way, the average execution time of the query engine will be reduced by increasing the usability of SyQL query engine.

Finally, we want to conclude this Chapter by saying that in the future the separation

between the SyQL query engine and the DBMS could be not so sharp as now. Since, the layering could be “broken”, in order to achieve better performances.

9 Validation

In this chapter, we present the validation of our work. This validation has been made in order to answer to the following question: can SyQL increase user productivity?

In order to answer this question, we tried to quantify the performance gain given by the language in terms of productivity and correctness by performing a couple of controlled experiments [15]. In these experiments, undergraduate students have solved a set of tasks by using both SyQL and SQL/LINQ. These experiments enabled us to measure the effort (minute men) spent by the users and the size of the code necessary to complete the tasks. The experiments put in evidence the conciseness of SyQL, which allows to reduce dramatically the number of wrong queries and the length of queries. Therefore, a user can complete more tasks correctly by increasing the overall productivity/efficiency.

Moreover, we used two real case studies as instances of the validation methodology. During these activities, SyQL has been used to support the data set preparation process of two scientific works. On these occasions, SyQL users have solved more difficult tasks due to the higher complexity of the AISEMA database. In order to have a fair comparison, the effort spent to prepare the data set using standard tools has been also collected. This is important, since it is common in this type of researches to write ad-hoc programs for extracting information from a database. By using SyQL, this is not necessary any more. If we need to develop some new concepts/methods, the SyQL architecture allows the user to reuse them in the future. In this way, a code-base that captures the experience is built [27]. In terms of overall time, SyQL outperforms the ad-hoc approach.

9.1 Controlled Experiments

In many different scientific areas, controlled experiments are widely conducted, because this research method is useful for validating theories. Software engineering is no exception, and the empirical software engineering area takes advantages by this research method [7].

To quantify the performance gain given by our language, we performed a couple of controlled experiments, where undergraduate students were required to solve a set of

data processing tasks by using both SyQL and SQL/LINQ.

These experiments enable us to do considerations about user productivity: the final results can help us to produce better estimations about the productivity of the SyQL users. Due to the lack of historical data (SyQL is a new language), we cannot have yet estimations by using effectively the FP/LOC ratio, whereas in SQL is possible (see Table 3 on page 38).

Therefore, the experiments have to be designed to compare objectively the two languages by preventing the measuring interference coming from the users who have taken part in the experiments. This point will be discussed in details in the next section. Moreover, to make possible comparing the measured variables, we have not used all the features present in SyQL: the fuzzy logic and the temporal expressions have been not covered.

9.1.1 Experimental Design

To the best of our knowledge, there are no scientific works that compare standard SQL and LINQ to another object-oriented query language through an experiment. To design the comparison between SyQL, SQL, and LINQ properly, we reviewed several other works [3][95][125], where the authors comparatively analyzed different software development methodologies, e.g. comparing pair programming to solo programming. The cited experiments are the closest works to our approach.

Our experiments were designed as a pair to compare the two couples of languages (SyQL Vs SQL and SyQL Vs LINQ) preventing interference coming from users taking part to both SQL (or LINQ) and SyQL query design.

The two experiments have a similar design, and the differences are evidenced in the following description.

We have chosen SQL and LINQ because both have large communities of users [47] that support their integration in multiple systems. This makes SQL and LINQ widely available in terms of working implementations by enabling us to set up easily an environment for conducting the experiment.

Let us start by stating the final goal of the experiment pair: we want to compare the effort (minute/person) and the size of the code (LOC) necessary to complete a predefined set of process data extraction tasks by using both SyQL and SQL (or

LINQ), taking into account the correctness of the resulting queries. The experiments variables are the following:

- dependent variables: task execution time (minute men), query length (LOC), query correctness (Boolean variable);
- controlled variables: two sets of tasks (on two distinct databases);
- independent variables: query languages (SyQL and SQL).

The tasks' execution times and the queries' LOC are two directly measured dependent variables. Tasks' execution time can be easily measured by collecting for each task the start and the end times. Lines of Code (LOC) can be counted offline (i.e. after the end of the experiment) by using a common predefined criterion for both languages.

During the experiments, the queries' texts and start/end times were both collected by using electronic Acrobat FDF forms. These forms were directly filled by the subjects, and they contained the descriptions of the tasks. After collecting all these data automatically via Adobe® Acrobat®, we were able to compute for each query the indirect dependent variables (query correctness). Each of them is Boolean, and its value is true if and only if the corresponding query is correct. Correctness was evaluated by manually checking the queries' texts; a query was considered correct if and only if it completely fulfilled the requirements of the corresponding task. It is important to remark that we did not look only at the result-set produced, because it is not an error-proof method: a query can be wrong even producing the same result-set of a correct one.

As subjects for the first experiment, we took twenty-one (21) undergraduate students attending an undergraduate course in Software Engineering Project (SEP) management. For the second experiment, we took nine (9) graduate students, attending a graduate course in Service Oriented Architecture (SOA).

During both experiments, the students were split into two groups, called “control” and “experimental”.

Each experiment consisted of two sessions. During each session, all subjects had to complete the same task list: the control group did it by using SQL (or LINQ), whereas the experimental group used SyQL. Hence, at the end of both sessions, each subject had taken part both to the control and to the experimental group, trying to complete

two distinct lists of tasks. We want to stress this point, because this is an important choice of experimental design we took in order to avoid interference, i.e. the experience gathered by a subject when querying the database. This experience includes different information: field names, relations/classes names, result-sets, tasks breakdowns, etc. For this reason, we also changed database at the end of each session.
At the beginning of the first session of the first experiment, all 21 subjects filled in a questionnaire to assess their SQL skills. Answers showed that all the subjects had already some experience with SQL, which was not a surprise because SQL is one of the main topics of a first year undergraduate course. In addition, 11 of the 21 students had already used SQL to solve real world problems. On average, the subjects' SQL skills were good, and almost all our technical questions about SQL syntax and semantics were correctly answered (4.8% of wrong answers). On the contrary, in the second experiment, all nine students had no previous experience in LINQ.
During each session, before starting the experiment, subjects belonging to the experimental group received a 20-minute presentation about SyQL, in which the language was introduced by means of examples. In the second experiments, students also received a 20-minute presentation about LINQ.
At the end of both sessions, subjects who were part of the experimental group (i.e., the SyQL users) were requested to fill in a second questionnaire about their "SyQL experience". On average, the experience of using SyQL was considered "Good" by the students who took part to the two experiments (the scale was Excellent, Good, Fair, Poor). In addition, no bugs were reported by the subjects that took part in the experiments.

9.1.1.1 Differences between the two experiments

The two experiments mainly differ from the query languages adopted for solving the tasks: in the first experiment, SyQL and SQL were used for the comparison; in the second experiment, SyQL and LINQ were compared. The differences between LINQ and SQL make necessary to build an ad-hoc client for LINQ, which allows the user to query interactively the databases, since LINQ is a language originally designed to be embedded into a programming language like C# or VB.NET. Both the SQL and the LINQ clients do not have the auto-completion, in order to avoid bias coming from this

mechanism.

In the second experiment, we have added the SQL views to the databases. These views have been defined by using the defining queries of the SyQL concepts. This addition has been requested by the PhD Thesis Committee on 14/01/2011, in order to increase the fairness of the comparison between SyQL and LINQ. Finally, we propose a set of possible SQL solutions to the experimental tasks by using SQL views. In this way, we want to show why SyQL can still produce more elegant queries than SQL.

Each of the experiments involves a different number of subjects, but we do not consider it an issue, since we did considerations only within the same experimental set. In this way, the conclusions cannot be influenced by the different statistical support.

9.1.2 The Databases

The two databases we used for this experiment are DELLSTORE[15] and USDA[16]. Both are freely available online. DELLSTORE contains data of an online video store, whereas USDA contains all the foods available in the United States together with the nutrient data (it is also possible to query the database at http://www.nal.usda.gov/fnic/foodcomp/search/index.html), and it is maintained by the United States Department of Agriculture. The reasons that bring us to use these two databases are described below.

Firstly, the effects coming from the knowledge of software engineering domain are limited, since no data related to this domain are contained in these databases.

Secondly, these databases allow us to generate tasks that can be completed in an amount of time compatible with experiments' schedule. Indeed, information retrieval tasks from AISEMA databases, in our experience, require a lot of time by making the execution in a controlled environment impassable. For this reason, we have decided to employ two databases that are not related to the Software Engineering domain since the very beginning of the experiments' design. In addition to that, we think that a more generic experiment can provide us more significant data, since SyQL is designed,

[15]DELLSTORE database – PgFoundry: Sample Databases – http://pgfoundry.org/frs/download.php/543/dellstore2-normal-1.0.tar.gz

[16]USDA database – PgFoundry: Sample Databases – http://pgfoundry.org/frs/download.php/555/usda-r18-1.0.tar.gz

from the very beginning, to query any kind of databases, and is not limited to AISEMA databases. This choice makes possible to reach conclusions that can be extended in other domains not limited to Software Engineering.

By looking at the size of the two databases, it is possible to appreciate that they have a similar number of relations: eight for DELLSTORE (see Figure 30 on page 172) and 10 for USDA (see Figure 31 on page 173). This similarity enables us to generate two similar sets of tasks and the necessary SyQL libraries for them.

Therefore, we claim that these databases are good benchmarks for measuring the benefits provided by SyQL.

In the next sub-section, we show the possible solutions of the tasks, so as the reader can appreciate the similarities between the two set of tasks. For each database, we created three SyQL concepts (see Figures 32 and 33 on pages 174 and 175) by following the guidelines listed in Section 6.1 on page 126. To make possible to use LINQ, we generated the necessary classes by using Microsoft SqlMetal[17], a specific tool for mapping the LINQ code to SQL code. During this process, we have also renamed some of the fields of the classes for leveling the situation between the two groups of users (see Figure 34 on page 176 and Figure 35 on page 177). In addition to that, we have created SQL views into the databases, and we have defined them by using the defining queries of the SyQL concepts (see Listing 69 and Listing 70).

```
[1]  CREATE VIEW customer_view AS
[2]     SELECT  customerid, firstname, lastname,
[3]             country, username, password, age, income, gender
[4]     FROM dbo.customers;
[5]
[6]  CREATE VIEW order_view AS
[7]     SELECT  orderid, orderdate, customerid, netamount,
[8]             tax, totalamount
[9]     FROM dbo.orders;
[10]
[11] CREATE VIEW product_view AS
[12]    SELECT  dbo.products.prod_id, dbo.products.title,
[13]            dbo.products.actor, dbo.products.price,
[14]            dbo.products.common_prod_id, dbo.categories.categoryname,
[15]            dbo.inventory.quan_in_stock AS quantity_in_stock,
```

[17]SQLMetal – SQLMetal.exe (Code Generation Tool) – http://msdn.microsoft.com/en-us/library/bb386987.aspx

```
[16]            dbo.inventory.sales
[17]    FROM dbo.products INNER JOIN dbo.categories
[18]            ON dbo.products.category = dbo.categories.category
[19]                 INNER JOIN dbo.inventory
[20]                    ON dbo.products.prod_id = dbo.inventory.prod_id;
```

Listing 69: SQL views of DELLSTORE database.

```
[1]  CREATE VIEW food_view AS
[2]     SELECT  dbo.food_des.ndb_no AS food_id,
[3]             dbo.fd_group.fddrp_desc AS food_group,
[4]             dbo.food_des.long_desc AS long_description,
[5]             dbo.food_des.shrt_desc AS short_description,
[6]             dbo.food_des.comname AS common_name,
[7]             dbo.food_des.manufacname AS manufacturer_name,
[8]             dbo.food_des.ref_desc AS reference_description,
[9]             dbo.food_des.sciname AS scientific_name,
[10]            dbo.food_des.n_factor AS nutrition_factor,
[11]            dbo.food_des.pro_factor AS protein_factor,
[12]            dbo.food_des.fat_factor,
[13]            dbo.food_des.cho_factor AS cholesterol_factor
[14]    FROM dbo.food_des INNER JOIN dbo.fd_group
[15]                ON dbo.food_des.fdgrp_cd = dbo.fd_group.fdgrp_cd;
[16]
[17] CREATE VIEW journal_view AS
[18]    SELECT  datasrc_id AS journal_article_id,
[19]            authors AS authors_list,
[20]            title, year, journal AS journal_name,
[21]            vol_city AS volume, issue_state AS issue,
[22]            start_page, end_page
[23]    FROM dbo.data_src;
[24]
[25] CREATE VIEW nutrient_view AS
[26]    SELECT  nutr_no AS nutrient_id,
[27]            units AS measurement_unit,
[28]            tagname AS tag_name,
[29]            nutrdesc AS nutrient_description
[30]    FROM dbo.nutr_def;
```

Listing 70: SQL views of USDA database

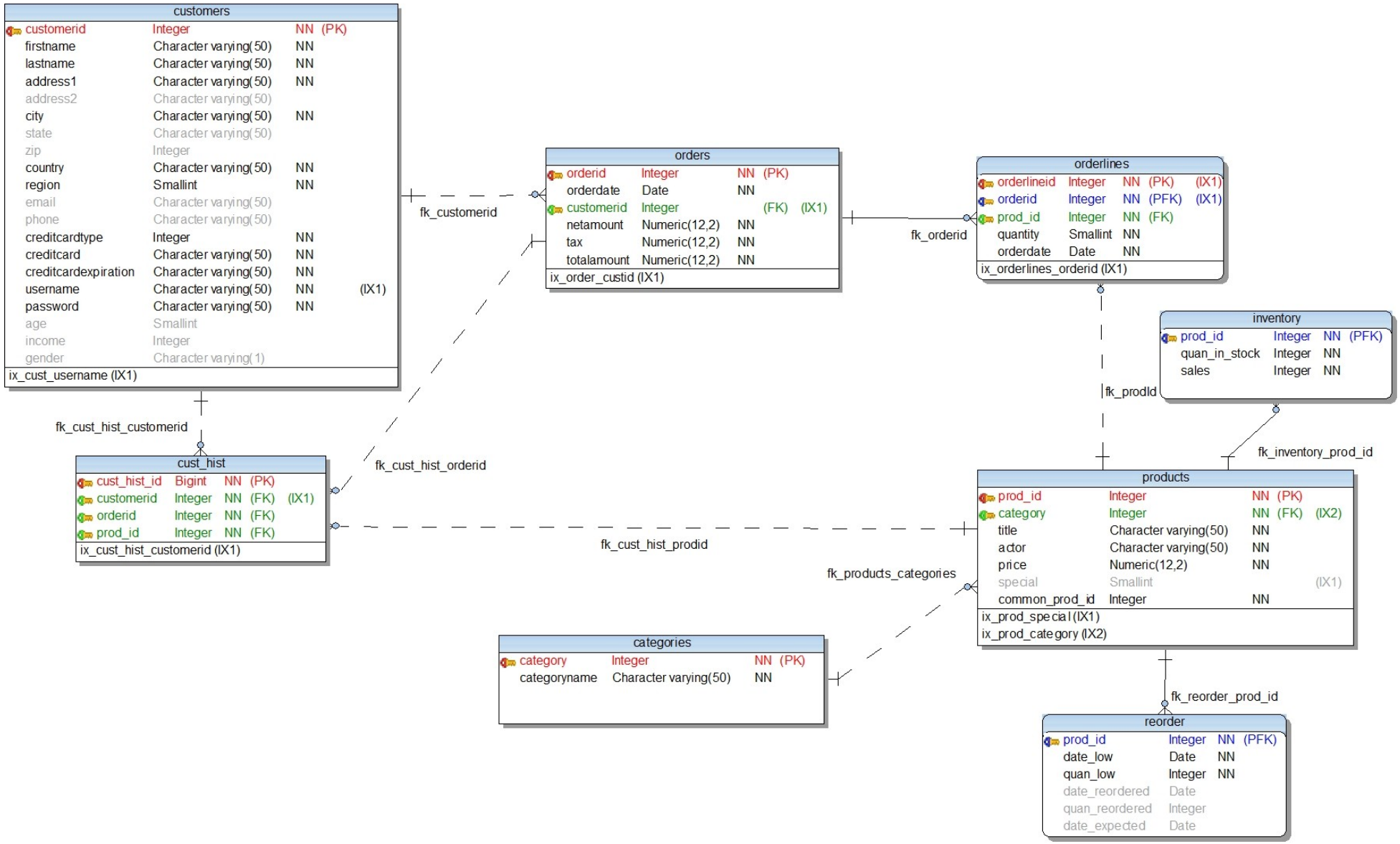

Figure 30: Entity relational diagram of the *DELLSTORE* database.

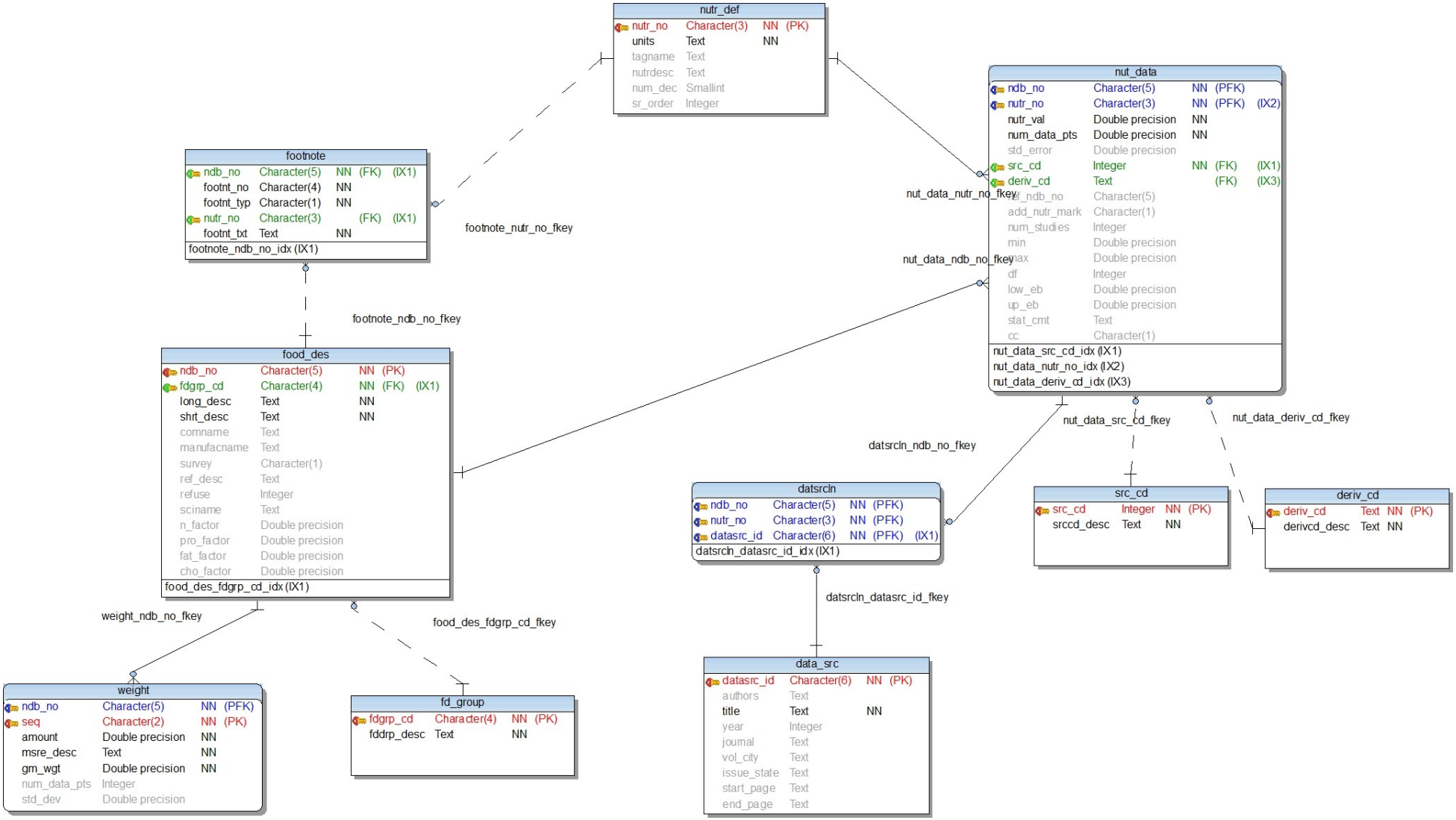

Figure 31: Entity relational diagram of the *USDA* database.

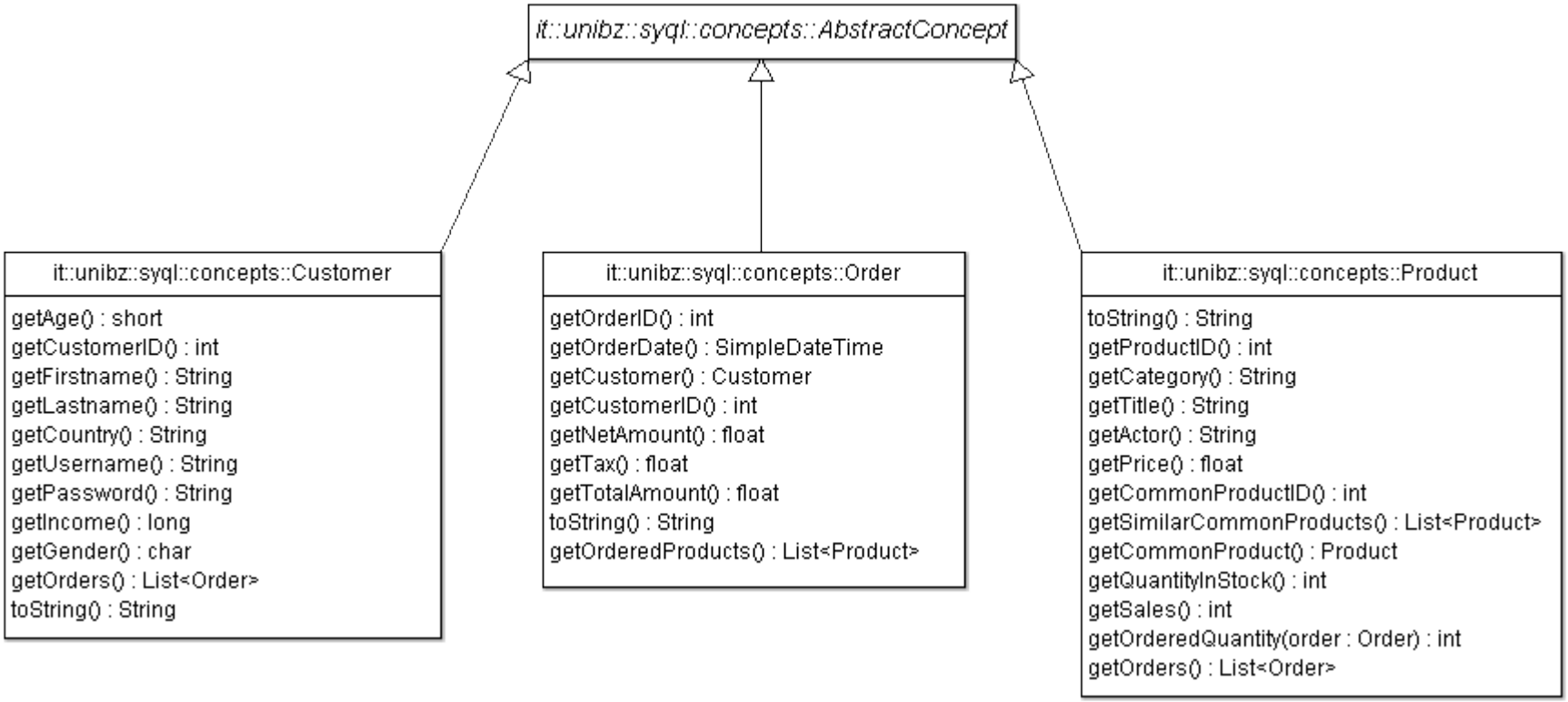

Figure 32: Concepts **c**lass diagram for the *DELLSTORE* database.

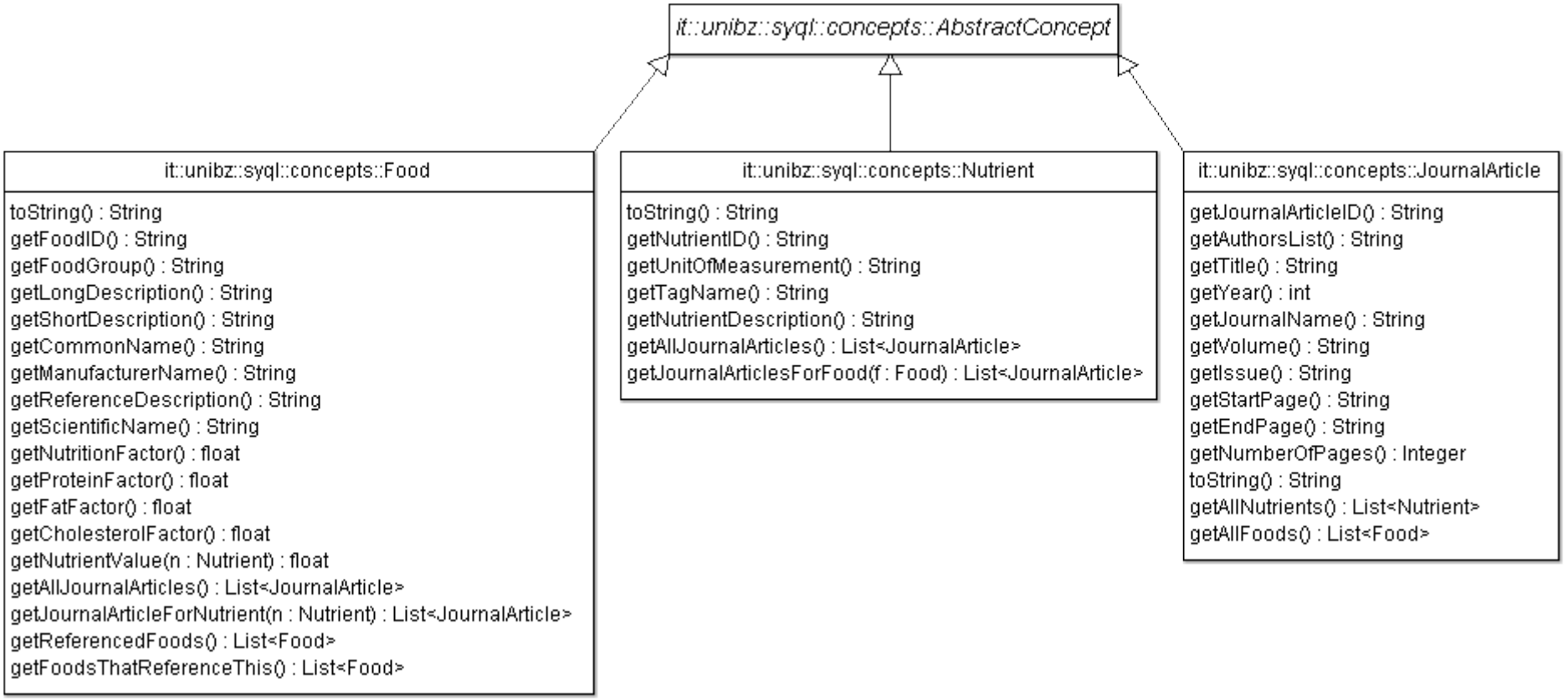

Figure 33: Concepts **c**lass diagram for the *USDA* database.

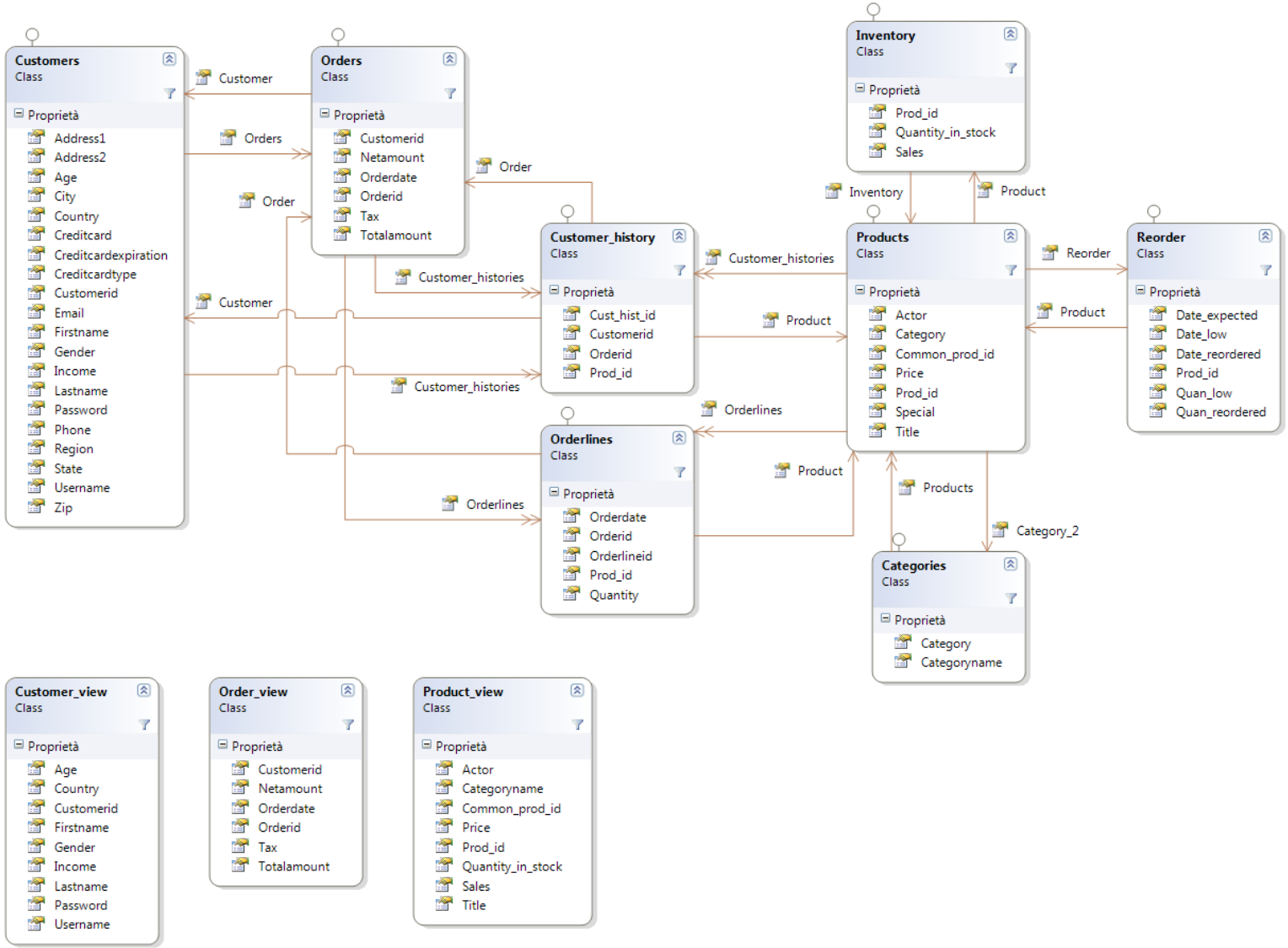

Figure 34: LINQ class diagram for the *DELLSTORE* database.

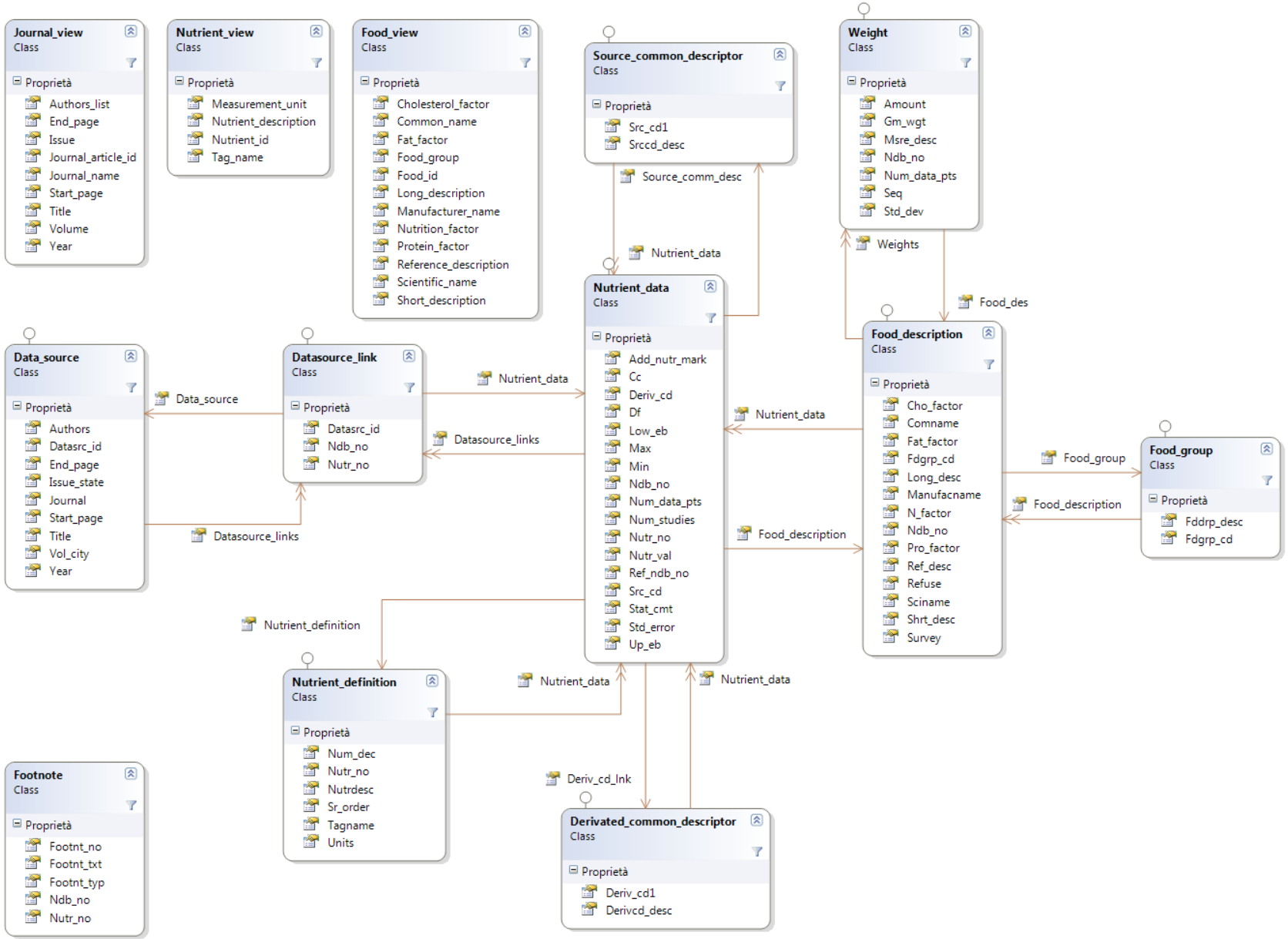

Figure 35: LINQ class diagram for the USDA database .

9.1.3 The Tasks

We designed each task-list to contain four distinct tasks, and each task had to be completed by using a predefined set of SQL features. In this way, we were able to generate a proper list of tasks ordered in ascending order by the required effort: each task requires progressively more effort than the previous one if tackled by SQL, whilst it requires a uniform effort if tackled by SyQL. Moreover, each task can be solved by using a single SQL/LINQ/SyQL query.

In the following, we list the SQL skills needed to solve each task:

- **task 1**: filtering;
- **task 2**: filtering + join with other tables;
- **task 3**: filtering + join with other tables + aggregate functions;
- **task 4**: filtering + join with other tables + aggregate functions + set operators.

As filtering, we mean one or more conditions applied to a single relation for discarding the undesired records.

The first tasks together with both SQL, SyQL and LINQ possible solutions are the following:

- ***DELLSTORE***: select all the customers (firstname, lastname, and id) who are older than 35 years, and younger than 55 years;
  - **SQL:**

```
[1] SELECT firstname, lastname, customerid
[2] FROM customers
[3] WHERE age > 35 AND age < 55;
```

  - **SyQL:**

```
[1] FROM Customer c
[2] WHERE c.getAge() > 35 AND c.getAge() < 55
[3] SELECT c.getFirstname(), c.getLastname(), c.getCustomerID();
```

– **LINQ:**

```
[1] from c in d.Customers
[2] where c.Age > 35 && c.Age < 55
[3] select new {c.Firstname, c.Lastname, c.Customerid};
```

- ***USDA***: select all the foods (id and short description) produced by the manufacturer *Burger King*.

  – **SQL:**

```
[1] SELECT food_des.ndb_no, food_des.shrt_desc
[2] FROM food_des
[3] WHERE lower(food_des.manufacname) LIKE 'burger%king%';
```

  – **SyQL:**

```
[1] FROM    Food f
[2] WHERE   f.getManufacturerName().toLowerCase() LIKE 'burger%king%'
[3] SELECT  f.getFoodID(), f.getShortDescription();
```

  – **LINQ:**

```
[1] from f in u.Food_description
[2] where SqlMethods.Like (f.Manufacname.ToLower(), "burger%king%")
[3] select new { f.Ndb_no, f.Shrt_desc };
```

In the second group of tasks, we added the difficulty of joining one or more relations by using an inner join operation. The USDA task required more effort by the subjects, because it requires to join more tables together. This implies a better understanding of the database structure.

Tasks descriptions are as follows :

- ***DELLSTORE***: select all the products (id, and title) that have a quantity in stock greater than 499 units.

  – **SQL:**

```
[1] SELECT p.prod_id, p.title
[2] FROM products p, inventory i
[3] WHERE p.prod_id = i.prod_id AND i.quan_in_stock > 499;
```

  – **SyQL:**

```
[1] FROM Product p
[2] WHERE p.getQuantityInStock() > 499
[3] SELECT p.getProductID(), p.getTitle();
```

– **LINQ:**

```
[1] from p in d.Products
[2] where p.Inventory.Quantity_in_stock > 499
[3] select new {p.Prod_id, p.Title};
```

- ***USDA***: select all the foods (id, and long description) that are part of the 'Beverages' group and contain more than 2 grams of 'Fructose'.

  – **SQL:**

```
[1] SELECT
[2]    food_des.ndb_no,
[3]    food_des.long_desc
[4] FROM
[5]    public.food_des,
[6]    public.nut_data,
[7]    public.nutr_def,
[8]    public.fd_group
[9] WHERE
[10]      food_des.ndb_no = nut_data.ndb_no AND
[11]      food_des.fdgrp_cd = fd_group.fdgrp_cd AND
[12]      nut_data.nutr_no = nutr_def.nutr_no AND
[13]      nutr_def.nutrdesc = 'Fructose' AND
[14]      nut_data.nutr_val > 2 AND
[15]      fd_group.fddrp_desc = 'Beverages';
```

  – **SyQL**:

```
[1] FROM Food f, Nutrient n
[2] WHERE   f.getFoodGroup() = 'Beverages' AND
[3]       n.getNutrientDescription() = 'Fructose' AND
[4]       f.getNutrientValue(n) > 2
[5] SELECT f.getFoodID(), f.getLongDescription();
```

  – **LINQ:**

```
[1] from nut in u.Nutrient_definition join data
[2]       in u.Nutrient_data on nut equals data.Nutrient_definition
[3] where nut.Nutrdesc == "Fructose" && data.Nutr_val > 2 &&
[4]       data.Food_description.Food_group.Fddrp_desc == "Beverages"
[5] group data by data.Food_description into result
[6] select new {result.Key.Ndb_no, result.Key.Long_desc};
```

To complete the third group of tasks, the subjects in the control group (SQL users) had to count the database records by using aggregate functions. Contrariwise, the SyQL

users relied on the method *size()* specified by the *java.util.Collection* interface. Tasks descriptions are as follows:

- ***DELLSTORE***: select all the customer (firstname, lastname, and id) that have placed exactly six orders.
  - **SQL:**

```
[1] SELECT c.firstname, c.lastname, c.customerid
[2] FROM customers c, (
[3]    SELECT customerid, count(orderid) AS numberOfOrders
[4]    FROM orders
[5]    GROUP BY customerid
[6]    HAVING COUNT(orderid) = 6
[7]    ) AS o
[8] WHERE o.customerid = c.customerid;
```

  - **SyQL:**

```
[1] FROM Customer c
[2] WHERE c.getOrders().size() = 6
[3] SELECT c.getFirstname(), c.getLastname(), c.getCustomerID();
```

  - **LINQ:**

```
[1] from c in d.Customers
[2] where c.Orders.Count == 6
[3] select new {c.Firstname, c.Lastname, c.Customerid};
```

- ***USDA***: select all the Journal Articles (id, and title), in which exactly seven foods are discussed.
  - **SQL:**

```
[1] SELECT  data_src.datasrc_id, data_src.title
[2] FROM    data_src,
[3]    (  SELECT distinct datasrc_id, ndb_no
[4]    FROM public.datsrcln ) AS foo
[5] WHERE data_src.datasrc_id = foo.datasrc_id
[6] GROUP BY data_src.datasrc_id, data_src.title
[7] HAVING COUNT(*) = 7;
```

  - **SyQL:**

```
[1] FROM    JournalArticle ja
[2] WHERE   ja.getAllFoods().size() = 7
[3] SELECT  ja.getJournalArticleID(), ja.getTitle();
```

– **LINQ:**

```
[1] from ds in u.Data_source join
[2] par_res in (
[3]       from dslk in u.Datasource_link
[4]       group dslk by
[5]          new {sourceid = dslk.Datasrc_id, ndb_no = dslk.Ndb_no}
[6]             into partial_res
[7]       select partial_res.Key
[8] ) on ds.Datasrc_id equals par_res.sourceid
[9] group ds by new {srcID = ds.Datasrc_id, title = ds.Title}
[10]           into result
[11]   where result.Count() == 7
[12]   select result.Key;
```

The fourth group of tasks was the most difficult ones. To tackle these tasks, SQL users had to use SQL set operators. In both sessions, SQL users failed to produce proper solutions, whereas the 63.6% and the 70.0% of SyQL users successfully solved the DELLSTORE and USDA fourth tasks respectively.

The fourth tasks together with both SQL, SyQL, and LINQ possible solutions are as follows:

- ***DELLSTORE***: select the orders (id, and customer's id) containing three units of a product that has no common and similar products (it means that a product defines itself as a common product, and no other products define it as a common product).

  – **SQL:**

```
[1] SELECT o.orderid, o.customerid
[2] FROM (
[3]       SELECT ol.prod_id, ol.orderid, o.customerid
[4]       FROM orderlines ol, orders o
[5]       WHERE ol.orderid = o.orderid
[6]       GROUP BY ol.prod_id, ol.orderid, o.customerid
[7]       HAVING SUM(ol.quantity) = 3
[8]    ) AS o,
[9]    (
[10]         SELECT prod_id
[11]         FROM products
[12]         WHERE prod_id = common_prod_id
```

```
[13]
[14]        EXCEPT
[15]
[16]        SELECT p2.prod_id
[17]        FROM products p1, products p2
[18]        WHERE   p1.prod_id <> p2.prod_id AND
[19]                p1.common_prod_id = p2.prod_id AND
[20]                p2.prod_id = p2.common_prod_id
[21]     ) AS p
[22]  WHERE o.prod_id = p.prod_id;
```

– **SyQL:**

```
[1] FROM Product p, Order o
[2] WHERE   p.getSimilarCommonProducts().size() = 0 AND
[3]           p.getCommonProductID() = p.getProductID() AND
[4]           p.getOrderedQuantity(o) = 3
[5] SELECT o.getOrderID(), o.getCustomerID() ;
```

– **LINQ:**

```
[1] from ol in d.Orderlines
[2] group ol by new { product = ol.Product, order = ol.Order }
[3]         into inter_result
[4] where   inter_result.Sum(i => i.Quantity) == 3 &&
[5]         inter_result.Key.product.Prod_id ==
[6]                 inter_result.Key.product.Common_prod_id &&
[7]         !( from p1 in d.Products join p2 in d.Products
[8]                 on p1.Common_prod_id equals p2.Prod_id
[9]           where p2.Common_prod_id == p2.Prod_id && p1 != p2
[10]          select p2).Contains(inter_result.Key.product)
[11]  select new {
[12]        order = inter_result.Key.order.Orderid,
[13]        customerid = inter_result.Key.order.Customer.Customerid};
[14]
```

- ***USDA***: select all the foods (id, and long description) that are not referenced by other foods and reference three other foods.

– **SQL:**

```
[1] DROP TABLE IF EXISTS ref;
[2] CREATE TEMPORARY TABLE ref AS (
[3]   SELECT DISTINCT ndb_no, ref_ndb_no
[4]   FROM nut_data
[5]   WHERE ndb_no <> ref_ndb_no AND ref_ndb_no <> ''
[6] );
[7]
```

```
[8] SELECT f.ndb_no, f.long_desc
[9] FROM food_des AS f, (
[10]      SELECT ndb_no
[11]      FROM ref
[12]      GROUP BY ndb_no
[13]      HAVING COUNT(*) = 3
[14]   ) AS ref_food, (
[15]      SELECT ndb_no
[16]      FROM food_des
[17]
[18]      EXCEPT
[19]
[20]      SELECT r2.ndb_no
[21]      FROM ref AS r1, ref AS r2
[22]      WHERE r1.ref_ndb_no = r2.ndb_no
[23]   ) AS back_ref_food
[24]   WHERE   f.ndb_no = ref_food.ndb_no AND
[25]              back_ref_food.ndb_no = f.ndb_no;
```

- **SyQL:**

```
[1] FROM Food f
[2] WHERE   f.getReferencedFoods().size() = 3 AND
[3]            f.getFoodsThatReferenceThis().size() = 0
[4] SELECT  f.getFoodID(), f.getLongDescription();
```

- **LINQ:**

```
[1]    from tripleRef in
[2]    (  from ThreeRef in
[3]       (  from n in u.Nutrient_data
[4]          group n by new { refer =n.Ref_ndb_no,
[5]                food = n.Food_description } into inter_res
[6]          where   inter_res.Key.food.Ndb_no !=
[7]                                  inter_res.Key.refer &&
[8]                  inter_res.Key.refer != ""
[9]          select inter_res.Key
[10]     )
[11]        group ThreeRef by ThreeRef.food into inter_res_2
[12]     where inter_res_2.Count() == 3
[13]     select inter_res_2.Key
[14]    )
[15]   where !(   from nut in u.Nutrient_data
[16]              select nut.Ref_ndb_no).Contains(tripleRef.Ndb_no)
[17]   select new{ tripleRef.Ndb_no, tripleRef.Long_desc};
```

9.1.4 Measurement

Comparing queries across languages and environments is known to be a difficult problem [84]. To measure queries' length, we defined a specific Line-of-Code (LOC) size metrics suitable for SyQL, SQL, and LINQ queries. In this way, we could compare query length without being affected by the user coding style.

When defining our metrics, we tried to address the well-known issue of bias by counting the Select-Clause as a single line, because, for the same task, it contains always the same number of elements both in SQL and in SyQL. The same happens for the Group-By-Clause: we always counted it as a single line without considering its elements, because it is directly related to the Select-Clause.

On the contrary, the From-Clause and the Where-Clause are measured in a different way. The former is equal to one plus the number of the specified join-conditions (e.g. … FROM Customer c JOIN Sales s ON c.customer_id = s.customer_id …), whereas the latter is computed by counting the number of conditions specified in the clause. In this way, one or more natural-joins are counted as a single line. To our purposes, this count is the correct one, because this construct adds no join conditions to the Where-Clause.

As far as the Having-Clause is concerned, we considered one line per each condition specified.

In the next example, all rules are used to compute the length of the following SQL query:

```
[1]  SELECT c.firstname, c.lastname, c.customerid
[2]  FROM customers c NATURAL JOIN orders o
[3]  WHERE c.lastname = 'Smith'
[4]  GROUP BY c.customerid, c.firstname, c.lastname
[5]  HAVING count(DISTINCT orderid) = 6
```

Listing 71: Computing the length of a generic SQL query.

According to our metrics, the size of the query above is 5, computed as follows:

- *Select-Clause* = 1 (always);
- *From-Clause* = 1 (1 natural join);
- *Where-Clause* = 1 (1 filtering condition);
- *Group-By-Clause* = 1 (always);
- *Having-Clause* = 1 (1 condition).

Our results show a correlation between query length and the average time necessary to write it by using SyQL, SQL, and LINQ: in general, the longer the query according to our metrics, the larger the effort (see Figure 36). Therefore, we claim that the above-defined criterion is correct to estimate user effort.

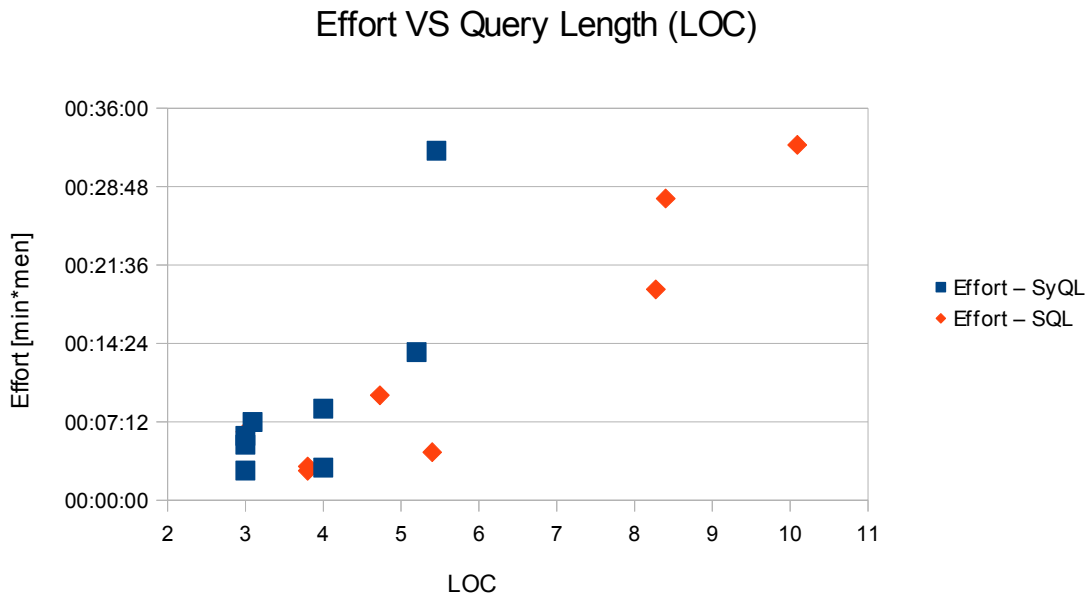

Figure 36: Comparison between Effort [minutes*men] and query length.

9.1.5 Evaluation Plan

In this section, we introduce the main contribution of Subsection 9.1 , showing that SyQL queries are less error-prone than SQL and LINQ ones. In addition to that, we can also say that the effort needed to start using SyQL is less than the one required to learn using LINQ.

In the SyQL vs. SQL experiment, we decided to analyze only correctness data for several reasons; the first one relates to the learning process: users have more experience by using SQL than SyQL, hence comparing the efforts between the two group maybe considered invalid; the second reason comes from the nature of the measure. Once a correctness criterion was defined, data about errors in queries are Boolean values, and therefore, are less affected by uncertainty than effort data.

In the SyQL vs. LINQ experiment, the previous consideration about the learning process is no more valid, because the users have no experience both in SyQL and in

LINQ. Therefore, the comparison of the efforts can be more significant than in the effort measured in the first experiment.

9.1.5.1 Tasks Execution Times (the Efforts)

To produce statistically significant results about effort data it is necessary to compare the efforts collected during both sessions by pooling the two data sets into a single one. This is necessary for two reasons: first, it makes possible to compute the individual acceleration factor of each user; second, it augments the statistical support by increasing the number of observations of the phenomena.

Analysis of Variance (ANOVA) tests has been conducted on the data coming from the two sessions of both experiments with significance set to 5%.

First Experiment (SyQL Vs SQL)

Indeed, effort data coming from the first experiment can be used for comparison only within the same experimental set, because, the execution of the ANOVA test on effort data coming from both experiments sessions rejected the null-hypothesis, which states that the distribution of the efforts spent on the different questions are similar. Table 15 shows the result of the ANOVA analysis on the effort data.

Variable	ANOVA Probability
Elapsed Time Query_1 (SQL)	0.032
Elapsed Time Query_1 (SyQL)	0.005
Elapsed Time Query_2 (SQL)	0.000
Elapsed Time Query_2 (SyQL)	0.000
Elapsed Time Query_3 (SQL)	0.002
Elapsed Time Query_3 (SyQL)	0.527
Elapsed Time Query_4 (SQL)	0.471
Elapsed Time Query_4 (SyQL)	0.001

Table 15: Result of the ANOVA tests (5% of significance) conducted on the effort variables (first experiment).

The results of the analysis show that almost all the effort variables have a different distribution except the *"Elapsed Time Query_3 (SyQL)"* and the *"Elapsed Time Query_4 (SQL)"* variables. However, the other two respective variables *"Elapsed Time Query_3 (SQL)"* and *"Elapsed Time Query_4 (SyQL)"* have different distributions, so they cannot be pooled together in a single data set. Therefore, a

statistically significant comparison between the efforts of the two data sets showing a difference in the effort due to the choice of the query language cannot be carried out at this stage. In other words, it is not possible to compute the acceleration factor of a user by using first experiment data.

Second Experiment (SyQL Vs LINQ)

The effort data coming from the second experiment shows enough similarity to be compared over the two experimental sets (DELLSTORE, USDA). As shown by ANOVA test (see the results in Table 16), the effort variables related to the first tasks (the easiest) have a similar distribution, making possible the pooling of the two variables.

Variable	ANOVA Probability
Elapsed Time Query_1 (LINQ)	0.463
Elapsed Time Query_1 (SyQL)	0.641
Elapsed Time Query_2 (LINQ)	0.003
Elapsed Time Query_2 (SyQL)	0.002
Elapsed Time Query_3 (LINQ)	0.786
Elapsed Time Query_3 (SyQL)	0.006
Elapsed Time Query_4 (LINQ)	-
Elapsed Time Query_4 (SyQL)	0.005

Table 16: Result of the ANOVA tests (5% of significance) conducted on the effort variables (second experiment).

Therefore, a statistically significant comparison between the efforts of the two simplest tasks can be carried out, so that the acceleration factor of a user can be estimated with a good support.

9.1.5.2 Lines Of Code (LOC)

First Experiment (SyQL Vs SQL)

The lines of code (LOC) variables show the same behavior as effort variables of the first experiment. Inside each group of tasks, the ANOVA probability values shown in Table 17 are too low by making it difficult to start analyses in order to come to well-supported conclusions.

Variable	ANOVA Probability
LOC Query_1 (SQL)	0.000
LOC Query_1 (SyQL)	1.000
LOC Query_2 (SQL)	0.000
LOC Query_2 (SyQL)	0.000
LOC Query_3 (SQL)	0.087
LOC Query_3 (SyQL)	0.361
LOC Query_4 (SQL)	0.103
LOC Query_4 (SyQL)	0.000

Table 17: Result of the ANOVA tests (5% of significance) conducted on the lines of code variables (first experiment).

Second Experiment (SyQL Vs LINQ)

Conversely, some lines of code (LOC) variables measured during the second experiment show similar distributions (see Table 18). Therefore, we explored the size of SyQL and LINQ queries in order to reach statistically supported conclusions.

Variable	ANOVA Probability
LOC Query_1 (LINQ)	1.000
LOC Query_1 (SyQL)	1.000
LOC Query_2 (LINQ)	0.005
LOC Query_2 (SyQL)	0.011
LOC Query_3 (LINQ)	0.311
LOC Query_3 (SyQL)	1.000
LOC Query_4 (LINQ)	-
LOC Query_4 (SyQL)	1.000

Table 18: Result of the ANOVA tests (5% of significance) conducted on the lines of code variables (second experiment).

9.1.5.3 The Queries Correctness

First Experiment (SyQL Vs SQL)

More results that are interesting came from the ANOVA test performed on the query correctness data as shown by Table 19. All groups of tasks present relatively high values of ANOVA probability, except for the variable *"Query_3 Correctness (SQL)"*. For this reason, subsequent analyses have been conducted only on query correctness variables of the groups of tasks 1, 2, and 4, by discarding group of tasks 3.

Variable	ANOVA Probability
Query_1 Correctness (SQL)	0.506
Query_1 Correctness (SyQL)	0.361
Query_2 Correctness (SQL)	0.346
Query_2 Correctness (SyQL)	1.000
Query_3 Correctness (SQL)	0.001
Query_3 Correctness (SyQL)	0.361
Query_4 Correctness (SQL)	1.000
Query_4 Correctness (SyQL)	0.774

Table 19: Result of the ANOVA tests (5% of significance) conducted on query correctness variables (first experiment).

As a proof of concept, it is possible to appreciate these differences/equalities about the distributions simply looking at the statistical summary of the collected data in Table 30 on page 195.

Second Experiment (SyQL Vs LINQ)

A similar behavior of the correctness variables is observed during the second experiment (see Table 20), but only group of tasks 3 and 4 show relatively high values of ANOVA probability.

Variable	ANOVA Probability
Query_1 Correctness (LINQ)	0.011
Query_1 Correctness (SyQL)	0.397
Query_2 Correctness (LINQ)	0.010
Query_2 Correctness (SyQL)	0.101
Query_3 Correctness (LINQ)	0.397
Query_3 Correctness (SyQL)	0.879
Query_4 Correctness (LINQ)	1.000
Query_4 Correctness (SyQL)	0.411

Table 20: Result of the ANOVA tests (5% of significance) conducted on query correctness variables (second experiment).

Therefore, we are not able to produce well-supported conclusions about the correctness of the first tasks.

9.1.6 Analysis

First Experiment (SyQL Vs SQL)

Our statistical analysis on the correctness data coming from the groups of tasks 1, 2, and 4 clearly shows that the overall number of correct queries written by using SyQL is higher than the SQL ones.

By looking at Table 21, it is possible to see the overall results computed over the three groups of tasks. For each language, 63 queries were included within the three groups of tasks. SyQL users wrote 55 correct queries in total, which is the 87.30% of the total number of queries. Compared with the SQL users, the advantages of SyQL are obvious, because SQL users wrote only 35 correct queries in total, which is the 55.56% of the whole set of queries.

Hence, the correctness performance of the users of the experimental groups improved of 31.74%.

SQL Correct Queries	**SyQL Correct Queries**
35 (63)	55 (63)
55.56%	***87.30%***

Table 21: Overall correct queries (groups of tasks 1, 2, and 4 – first experiment).

Next, the results of a finer grained analysis are presented. For each user, we have performed a comparison between the results achieved by using both query languages in the same group of tasks. Therefore, for each group of tasks, we created four disjoint sets that contain:

- users who have failed to complete the tasks by using both languages;
- users who have successfully completed only a single task by using SQL;
- users who have successfully completed only a single task by using SyQL;
- users who have successfully completed both tasks by using both languages.

In Table 22, the results of the first group of tasks are shown. It is possible to see that no user has written any wrong queries. Only a single user has successfully completed one task by using SQL, and three users completed one task by using SyQL. Finally, the 80.85% of the users (17) have successfully completed both tasks by using both

languages. An explanation can be found in the excessive simplicity of the tasks of this group, which can be easily completed by using both languages. In fact, the tasks only require to specify filtering conditions.

Group of Tasks 1			
Both Incorrect	**Only SQL Correct**	**Only SyQL Correct**	**Both Correct**
0	1	3	17
0.00%	***4.76%***	***14.29%***	***80.95%***

Table 22: Results about the queries of the group of tasks 1 (first experiment).

Almost the same results come from the second group of tasks (see Table 23). However, this time there are no users who have completed successfully only one task by using SQL. To complete these tasks, it is necessary to specify filtering conditions and to join one or more relations by using inner join. Therefore, the increased level of difficulty makes the writing SQL queries more difficult.

Group of Tasks 2			
Both Incorrect	**Only SQL Correct**	**Only SyQL Correct**	**Both Correct**
0	0	4	17
0.00%	***0.00%***	***19.05%***	***80.95%***

Table 23: Results about the queries of the group of tasks 2 (first experiment).

In the fourth group of tasks, results are completely different from the two groups of tasks described before.. Table 24 shows that no one of the users has solved both tasks by using both languages. Moreover, 66.67% of users have completed a single task by using SyQL, and no tasks at all by using SQL. We can explain this result by referring to the higher difficulty of the tasks that require considerable SQL skills (already listed in section 9.1.3). In addition to that, SQL users had to deal with a database schema they were not familiar with.

Group of Tasks 4			
Both Incorrect	**Only SQL Correct**	**Only SyQL Correct**	**Both Correct**
7	0	14	0
33.33%	***0.00%***	***66.67%***	***0.00%***

Table 24: Results about the queries of the group of tasks 4 (first experiment).

Second Experiment (SyQL Vs LINQ)

In the second experiment, correctness results (see Table 25 and Table 26) confirm the previous ones: the probability that a SyQL user will complete successfully a difficult task is higher.

Group of Tasks 3			
Both Incorrect	**Only SQL Correct**	**Only SyQL Correct**	**Both Correct**
2	0	6	1
22.22%	***0.00%***	***66.67%***	***11.11%***

Table 25: Results about the queries of the group of tasks 3 (second experiment).

From the table above, it is possible to see that six users have successfully completed one of the third level tasks only by using SyQL, and only a single user has correctly completed these tasks by using both languages. This effect becomes more evident in the fourth group of tasks (see Table 26), in which three users reached the result by using only SyQL.

Group of Tasks 4			
Both Incorrect	**Only SQL Correct**	**Only SyQL Correct**	**Both Correct**
6	0	3	0
66.67%	***0.00%***	***33.33%***	***0.00%***

Table 26: Results about the queries of the group of tasks 4 (second experiment).

Therefore, the data we collected show the same trend of the data of the first experiment, if we take into account the different number of subjects involved.

Effort statistics show a reduction of about 23% on the easier tasks (the first group of tasks).

Group of Tasks 1	
Average Effort LINQ [min*men]	13.25
Std Dev Effort LINQ [min*men]	2.96
Average Effort SyQL [min*men]	10.22
Std Dev Effort SyQL [min*men]	5.07
Ratio Effort LINQ/SyQL	1.3
Effort reduction %	**-22.85%**

Table 27: Results about the effort spent to complete the tasks 1 (second experiment).

This result is very interesting because it allows us to appreciate the fact that SyQL can make the user's learning curve less steep. In other words, users have fewer issues

when they start using SyQL than LINQ. This is also confirmed by the correctness of the queries produced by users for solving the first tasks: eight subjects (88.8%) completed correctly all tasks by using SyQL, whereas only three users completed the tasks by using SQL (see the statistical summary in Table 31 for details).

The data about the queries' LOC can be summarized in the following sentence: higher the task complexity, higher is the reduction in terms of lines of code (see Table 28 and Table 29).

On the easiest set of tasks, the SyQL queries have almost the same length of the LINQ ones, whereas on the third set of tasks, SyQL shows the benefits of its abstraction layer by reducing the queries' length of more than 50%.

Group of Tasks 1	
Average LOC LINQ	3.44
Std Dev LOC LINQ	0.53
Average LOC SyQL	3.56
Std Dev LOC SyQL	0.53
Ratio LOC LINQ/SyQL	0.97
LOC reduction %	**3.23%**

Table 28: Results about the queries' length of the group of tasks 1 (second experiment).

LOC reduction has also positive effects on query correctness (see Table 25).

Group of Tasks 3	
Average LOC LINQ	6.33
Std Dev LOC LINQ	1.75
Average LOC SyQL	3.00
Std Dev LOC SyQL	0.00
Ratio LOC LINQ/SyQL	2.11
LOC reduction %	**-52.63%**

Table 29: Results about the queries' length of the group of tasks 3 (second experiment).

The results of this second experiment confirm the increment of correctness observed during the first experiment by adding conclusions about the effort and query size. These conclusions are strengthened by the fact that users have no previous experience both in LINQ and in SyQL. We also noticed that LINQ users used the SQL views (through the auto-generated classes) in only four occasions, and in 75% of the cases the resulting queries were wrong.

	Language	Independent Variable	Average	Std Dev	Min	P25	Median	P75	Max	% Correct Answers
Dellstore	***SyQL***	**LOC Query_1**	4	0,00	4	4	4	4	4	90,91%
	SQL	**LOC Query_1**	3,8	0,42	3	4	4	4	4	80,00%
	SyQL	**Elapsed Time Query_1**	3	1,33	1	2	3	4	5	90,91%
	SQL	**Elapsed Time Query_1**	2,7	0,95	2	2	2,5	3	5	80,00%
	SyQL	**LOC Query_2**	3	0,00	3	3	3	3	3	100,00%
	SQL	**LOC Query_2**	3,8	0,42	3	4	4	4	4	90,00%
	SyQL	**Elapsed Time Query_2**	2,73	1,02	1	2	3	3,5	4	100,00%
	SQL	**Elapsed Time Query_2**	3,1	1,1	2	2,25	3	3	5	90,00%
	SyQL	**LOC Query_3**	3,09	0,30	3	3	3	3	4	90,91%
	SQL	**LOC Query_3**	5,4	0,97	4	5	5	5,75	7	100,00%
	SyQL	**Elapsed Time Query_3**	7,18	5,52	3	3,5	4	8	19	90,91%
	SQL	**Elapsed Time Query_3**	4,4	1,65	3	3	4	5	8	100,00%
	SyQL	**LOC Query_4**	5,45	0,82	5	5	5	5,5	7	63,64%
	SQL	**LOC Query_4**	8,4	2,50	4	6,25	9	10,75	11	0,00%
	SyQL	**Elapsed Time Query_4**	32,08	18,28	10	16,5	31	41	7	63,64%
	SQL	**Elapsed Time Query_4**	27,7	13,62	14	18	22,5	39	51	0,00%
	SyQL	**Total Elapsed Time**	45	18,08	22	30	43	58,5	75	
	SQL	**Total Elapsed Time**	37,9	14	22	28,5	33	49,25	59	
USDA	***SyQL***	**LOC Query_1**	3	0,00	3	3	3	3	3	100,00%
	SQL	**LOC Query_1**	3	0,00	3	3	3	3	3	90,91%
	SyQL	**Elapsed Time Query_1**	5,1	1,67	2	4,25	5	5,75	8	100,00%
	SQL	**Elapsed Time Query_1**	6,27	4,67	2	3	4	8,5	15	90,91%
	SyQL	**LOC Query_2**	5,2	0,42	5	5	5	5	6	100,00%
	SQL	**LOC Query_2**	8,27	0,65	8	8	8	8	10	72,73%
	SyQL	**Elapsed Time Query_2**	13,6	4,5	7	11,25	12,5	17,5	20	100,00%
	SQL	**Elapsed Time Query_2**	19,37	8,37	8	13,5	18	26	32	72,73%
	SyQL	**LOC Query_3**	3	0,00	3	3	3	3	3	100,00%
	SQL	**LOC Query_3**	4,73	0,65	4	4	5	5	6	36,36%
	SyQL	**Elapsed Time Query_3**	5,9	3	3	4	5	7,25	13	100,00%
	SQL	**Elapsed Time Query_3**	9,63	4,42	4	7	9	11	20	36,36%
	SyQL	**LOC Query_4**	4	0,00	4	4	4	4	4	70,00%
	SQL	**LOC Query_4**	10,09	1,76	7	9	10	10,5	13	0,00%
	SyQL	**Elapsed Time Query_4**	8,4	6,28	2	6	7	9	24	70,00%
	SQL	**Elapsed Time Query_4**	32,63	16,4	6	20,5	31	48	53	0,00%
	SyQL	**Total Elapsed Time**	33	7,97	19	28	32,5	40,5	43	
	SQL	**Total Elapsed Time**	67,92	26,22	26	48	67	90,5	106	

Table 30: Statistical summary of the first experiment data.

	Language	Independent Variable	Average	Std Dev	Min	P25	Median	P75	Max	% Correct Answers
Dellstore	*SyQL*	LOC Query_1	4	0,00	4	4	4	4	4	100,00%
	LINQ	LOC Query_1	4	0,00	4	4	4	4	4	75,00%
	SyQL	Elapsed Time Query_1	11	4,18	6	9	10	13	17	100,00%
	LINQ	Elapsed Time Query_1	13,5	1,91	12	12	13	14,5	16	75,00%
	SyQL	LOC Query_2	3	0,00	3	3	3	3	3	100,00%
	LINQ	LOC Query_2	4,5	1,73	3	3	4,5	6	6	100,00%
	SyQL	Elapsed Time Query_2	3,2	1,3	2	2	3	4	5	100,00%
	LINQ	Elapsed Time Query_2	3,75	2,06	2	2	3,5	5,25	6	100,00%
	SyQL	LOC Query_3	3	0,00	3	3	3	3	3	80,00%
	LINQ	LOC Query_3	6,75	2,06	5	5	6,5	8,25	9	25,00%
	SyQL	Elapsed Time Query_3	17,4	5,77	12	14	17	17	27	80,00%
	LINQ	Elapsed Time Query_3	31	7	26	27	28	33,5	39	25,00%
	SyQL	LOC Query_4	5	0,00	5	5	5	5	5	20,00%
	LINQ	LOC Query_4	N.D.	N.D.	N.D.	N.D.	N.D.	N.D.	N.D.	0,00%
	SyQL	Elapsed Time Query_4	19,5	10,61	12	15,75	19,5	23,25	27	20,00%
	LINQ	Elapsed Time Query_4	N.D.	N.D.	N.D.	N.D.	N.D.	N.D.	N.D.	0,00%
	SyQL	Total Elapsed Time	39,4	10,97	25	32	41	47	52	
	LINQ	Total Elapsed Time	40,5	15,42	18	38,25	45,5	47,75	53	
USDA	*SyQL*	LOC Query_1	3	0,00	3	3	3	3	3	75,00%
	LINQ	LOC Query_1	3	0,00	3	3	3	3	3	0,00%
	SyQL	Elapsed Time Query_1	9,25	6,55	5	5,75	6,5	10	19	75,00%
	LINQ	Elapsed Time Query_1	10,4	6,8	0	7	14	15	16	0,00%
	SyQL	LOC Query_2	4,5	1,00	3	4,5	5	5	5	50,00%
	LINQ	LOC Query_2	7,75	0,50	7	7,75	8	8	8	20,00%
	SyQL	Elapsed Time Query_2	31	13,17	17	21,5	31	40,5	45	50,00%
	LINQ	Elapsed Time Query_2	36,8	14,31	21	22	42	49	50	20,00%
	SyQL	LOC Query_3	3	0,00	3	3	3	3	3	75,00%
	LINQ	LOC Query_3	5,5	0,71	5	5,25	5,5	5,75	6	0,00%
	SyQL	Elapsed Time Query_3	6	1	5	5,5	6	6,5	7	75,00%
	LINQ	Elapsed Time Query_3	28,5	16,26	17	22,75	28,5	34,25	40	0,00%
	SyQL	LOC Query_4	4	0,00	4	4	4	4	4	50,00%
	LINQ	LOC Query_4	N.D.	N.D.	N.D.	N.D.	N.D.	N.D.	N.D.	0,00%
	SyQL	Elapsed Time Query_4	5	2,83	3	4	5	6	7	50,00%
	LINQ	Elapsed Time Query_4	2	-	2	2	2	2	2	0,00%
	SyQL	Total Elapsed Time	47,25	12,42	36	36,75	47,5	58	58	
	LINQ	Total Elapsed Time	59	16,19	36	49	66	68	76	

Table 31: Statistical summary of the second experiment data.

9.1.7 Threats to validity

9.1.7.1 Internal threats

Some perils threat the internal validity of both experiments. In the first experiment, the subjects already know SQL but not SyQL. This makes possible to speed up the execution of the SQL tasks by underestimating the acceleration factor provided by SyQL.

Another threat related to both experiments concerns the differences between the two tasks sets. This threat exists because the two database schemas are different, and a subject may require different amounts of effort to understand them. To eliminate this threat, all the subjects should know both schemas before the experiments. However, this is impossible in a controlled environment, because it takes times, and it means to lose the control of the subjects. This situation is very risky, since external interferences may invalidate the results.

Another peril originates from the subjects' language knowledge. This peril exists because in almost all the cases, except in the first tasks of the second experiment, were impossible to compare the effort data coming from the two tasks, since in both experiments the situation was anomalous: in the first experiment, the student already knew SQL but not SyQL; in the second experiment, neither LINQ nor SyQL was known, before the experiment, by the subjects. A more common situation should be when both languages are known. To eliminate this peril, all the subjects should know both languages used in the experiment.

9.1.7.2 External threats

Several threats may have an impact on the generalizability of the study.

First of all, participants of the experiments were students who subscribed voluntarily to the experiment. This makes possible they have a skill above the average. To eliminate this threat in further studies a more neutral group of subjects should be used.

Another threat related only to the first experiment concerns subjects' SQL experience. This threat exists because the subjects already known SQL prior to the experiment. However, they have not the skills and the speed that can be expected from a senior SQL developer. Therefore, to eliminate this threat senior SQL developers should be

used as subjects.

Another first experiment threat originates from the DELLSTORE and USDA sets of tasks. First, the thirds and fourths tasks are more complex but require less effort than every-day querying tasks. And second, the short duration of the tasks as compared to every-day querying tasks might favor SQL users due to the experience factor.

Another peril originates from the use of tasks that do not need for their completion the concepts used in Software Engineering (e.g. Class, User, Method, etc.). This peril exists because SyQL is mainly employed to query database of AISEMA systems, even though SyQL is designed, from the very beginning, to query any kind of databases, and is not limited to AISEMA databases. In our experience, information retrieval tasks from AISEMA databases require a lot of time by making the execution in a controlled environment impassable. For this reason, we have decided to employ two databases that are not related to the Software Engineering domain since the very beginning of the experiments' design.

9.1.8 Results

We can say that SyQL improves the quality of the queries written by the users. The correctness gain measured by us ranges from 9.5% to 66.67% depending from the complexity of the task, and the reduction of the lines of code is up to 52.63%. In addition, the users require less effort to start using SyQL than LINQ, and they obtain better results in terms of correctness.

However, looking at the queries written by the SQL users, we have noticed that the subjects produced no similar errors between them, also during the attempts to solve the tasks 3 and 4 (the most difficult) in both experiments. Therefore, this supports the fact that we have not introduced any research bias during the preparation and the execution of the experiments.

In terms of total elapsed time (see Table 30 and Table 31), the SyQL users have spent a similar amount of time on the two sets of tasks in both experiments: the average times are 45 (±18.08) vs 39.4 (±10.97) minutes in the DELLSTORE sessions, and 33 (±7.97) vs 47.25 (±12.42) minutes in the USDA sessions. These differences can be due to the different subjects involved in the experiments, and to the different environments where the experiments took place.

We want to warn the reader that during the second experiment, some subjects were unable to complete the third and the fourth tasks, therefore the average values of the total elapsed times are not the sum of the average times of the single tasks.

By comparing the averages of the total elapsed times in the DELLSTORE session of the SQL users (37.9 ± 14 minutes) and of the LINQ users (40.5 ± 15.42 minutes), it is possible to notice that there are no relevant differences. The same happens for the USDA session, in which the average total elapsed time (67.92 ± 26.22 minutes) of the SQL users is similar to the one of the LINQ user (59 ± 16.19 minutes). Therefore, we can say that no additional conclusions can be drawn by looking at the total elapsed times computed for distinct sessions.

By looking at the averages of the language total times (see Table 32), which are computed by pooling together the total elapsed times of the two experiments, it is possible to see that the SyQL total time is 24% lower than the SQL total time, and 20% lower than the LINQ total time.

Language Total Times		
Language	**Average**	**Std. Dev.**
SyQL	40.37	14.09
SQL	53.62	25.84
LINQ	50.78	17.75

Table 32: Statistical summary of the language total times.

9.1.8.1 SQL/LINQ Views

As mentioned before, SQL views (accessible through LINQ) have been used by LINQ users only in a few occasions. In our opinion, it depends from the classes generated by Microsoft SqlMetal. This tool generates a class for each table/view, but for the tables also generates methods for traversing foreign key constraints (see Figure 34 and Figure 35). For this reason, the users probably found easier writing queries by using tables than using views. Moreover, this evidences that both the languages provide methods for speeding up the query writing by reusing database information like the foreign keys. And this last consideration increases the significance of this comparison. In the next Section, we are going to show why SyQL can lead the user to write more concise and more elegant queries than SQL, although SQL views are used.

9.1.9 SyQL vs SQL Views

To compare the two languages, we reference some of the SyQL solutions proposed in Section 9.1.3 . For a brevity sake, we assume that the following SQL queries are run against a SQL database containing the views defined in Section 9.1.2 on page 169 (see Listing 69 and Listing 70).

The first significant example is given by the second task of the USDA session. By using SQL, it is possible to create an additional view (*FoodHasNutrientValue*), as shown in Listing 72 at lines [1]-[5], that increases the clarity of the relation *nut_data*. This relation implements the many-to-many relationship by linking together the relations on which *food_view* and *nutrient_view* views have been created.

At lines [11]-[13], two join conditions and one filtering condition have been specified, in order to join the three views specified in the from-clause together (see line [8]) by filtering out the records not compliant with the task requirements.

```
[1]  CREATE VIEWS FoodHasNutrientValue AS
[2]     SELECT  ndb_no AS food_id,
[3]             nutr_no AS nutrient_id,
[4]             nutrient_value AS nutr_val
[5]     FROM    nut_data;
[6]
[7]  SELECT  f.food_id, f.long_description
[8]  FROM    food_view f, nutrient_view n, FoodHasNutrientValue fhnv
[9]  WHERE   f.food_group = 'Beverage' AND
[10]         n.nutrient_description = 'Fructose' AND
[11]         fhnv.food_id = f.food_id AND
[12]         fhnv.nutrient_id = n.nutrient_id AND
[13]         fhnv.nutrient_value > 2;
```

Listing 72: SQL solution of the second task (USDA session).

By using SyQL, it is possible to reduce the *FoodHasNutrientValue* view into a single method call having one parameter (see Listing 73). In this way, the three conditions specified in the SQL solution can be reduced in only one condition (see line [4]), and only the *Food* and the *Nutrient* concepts have to be specified in the from-clause.

```
[1]  FROM Food f, Nutrient n
[2]  WHERE   f.getFoodGroup() = 'Beverages' AND
[3]     n.getNutrientDescription() = 'Fructose' AND
```

```
[4]     f.getNutrientValue(n) > 2
[5]   SELECT f.getFoodID(), f.getLongDescription();
```

Listing 73: SyQL solution of the second task (USDA session).

By comparing the two solutions, it is possible to see that the benefits in terms of clarity and conciseness are widely in favor of SyQL.

The following examples are more difficult than the one that we have just described, and we use them to explain why SyQL can produce better solutions than SQL.

In Listing 74, an SQL solution to the third task of the DELLSTORE session is proposed. To solve this task, we have not used views, since the naming used in the SQL schema is enough clear. The relations involved are *customers* and *orders*, which are linked by a one-to-many relationship. These two relations are joint together by a join condition (see line [8]).

In this SQL query, we have used a having-clause (see line [6]) for filtering out the records aggregated through the *count* aggregate function (see line [3]).

```
[1]   SELECT c.firstname, c.lastname, c.customerid
[2]   FROM customers c, (
[3]     SELECT customerid, count(orderid) AS numberOfOrders
[4]     FROM orders
[5]     GROUP BY customerid
[6]     HAVING COUNT(orderid) = 6
[7]     ) AS o
[8]   WHERE o.customerid = c.customerid;
```

Listing 74: SQL solution of the third task (DELLSTORE session).

By using SyQL (see Listing 75), it is possible to reduce all the above-mentioned elements into a single filtering condition (see line [2]), which contains two nested method calls and a comparison. The first method call (*getOrders()*) allows to skip the specification of several elements: the join condition and the *orders* relation in the from-clause. The second method call (*size()*) allows to skip the aggregation elements: *count* aggregate function and the group-by-clause (see line [5] of Listing 74). Finally, the numerical comparison (= 6) allows to filter the objects without using the having-clause.

```
[1]   FROM Customer c
```

```
[2]  WHERE c.getOrders().size() = 6
[3]  SELECT c.getFirstname(), c.getLastname(), c.getCustomerID();
```

Listing 75: SyQL solution of the third task (DELLSTORE session).

The comparison of the two solutions is again in favor of SyQL due to its higher conciseness.

To solve the third task of the USDA session, we propose an SQL query shown in Listing 76. In this query, we create the *JournalHasFood* views (see lines [1]-[3]) in order to create a link between the *data_src* and *food_des* relations (the former is hidden by the *journal_view* view). These relations are linked by a many-to-many relationship, hence the solution requires to join together the *journal_view* and *JournalHasFood* views (see lines [6]-[7]). In order to complete the task, it is necessary to aggregate the records by using the group-by-clause (see line [8]). Next, the resulting records have to be filtered by using the having-clause (see line [9]).

```
[1]  CREATE VIEWS JournalHasFood AS
[2]     SELECT distinct datasrc_id AS journal_id, ndb_no AS food_id
[3]     FROM public.datsrcln;
[4]
[5]  SELECT journal_article_id, title
[6]  FROM journal_view, JournalHasFood
[7]  WHERE journal_article_id = journal_id
[8]  GROUP BY journal_article_id, title
[9]  HAVING COUNT(*) = 7;
```

Listing 76: SQL solution of the third task (USDA session).

By using SyQL (see Listing 77), neither a join nor a group-by-clause nor a having-clause has to be used, since it is possible to reduce them into a single condition, which contains two nested methods calls and one numerical comparison like the solution shown in Listing 75.

```
[1]  FROM JournalArticle ja
[2]  WHERE ja.getAllFoods().size() = 7
[3]  SELECT ja.getJournalArticleID(), ja.getTitle();
```

Listing 77: SyQL solution of the third task (USDA session).

The SyQL solution is again far better than the SQL one, since it is possible to translate the many-to-many relationships and the grouping operations into method calls by

simplifying a lot the resulting SyQL query.

Another example is shown in Listing 78, where we provide an SQL solution to the fourth task of the USDA session. In order to simplify the solution of this task, we have defined a view called *food_ref* (see lines [1]-[3]). This view implements an N-to-many relationship, wherein N is the total number of nutrient contained into the *nutr_def* relation, by linking the *food_def* relation to itself (a food can reference up to N different foods, and a food can be referenced by many foods). The solution requires the following steps: a) joining the *food_view* and the *food_ref* views together (see line [6]); b) filtering out all the self-referenced foods (see line [7]); c) filtering out all the referenced foods (see lines [8]-[10]) by using the *NOT EXISTS* quantifier; d) grouping and filtering the foods by using a group-by-clause and a having-clause.

```
[1]  CREATE VIEWS food_ref AS
[2]     SELECT DISTINCT ndb_no AS id, ref_ndb_no AS refid
[3]     FROM nut_data;
[4]
[5]  SELECT  f.food_id , f.long_description
[6]  FROM    food_view f join food_ref fr on f.food_id = fr.id
[7]  WHERE   f.food_id <> fr.refid AND
[8]          NOT EXISTS ( SELECT *
[9]                       FROM    food_ref fr0
[10]                      WHERE   fr0.refid = f.id)
[11] GROUP BY   f.food_id , f.long_description
[12] HAVING COUNT(*) = 3;
```

Listing 78: SQL solution of the fourth task (USDA session).

SyQL (see Listing 79) allows to translate the steps (a), (b), and (d) into a single condition (see line [2]), and the step (c) into another condition (see line [3]). Both the conditions contain two nested method calls and one numerical comparison like the previous SyQL solution.

```
[1]  FROM Food f
[2]  WHERE   f.getReferencedFoods().size() = 3 AND
[3]          f.getFoodsThatReferenceThis().size() = 0
[4]  SELECT  f.getFoodID(), f.getLongDescription();
```

Listing 79: SyQL solution of the fourth task (USDA session).

This SyQL solution shows again that the possibilities offered by an object-oriented

environment allow to simplify the work of the final user by translating an N-to-many relationship into a couple of methods (*getReferencedFoods()* and *getFoodsThatReferenceThis()*). Moreover, the SyQL queries look more elegant and easier to read/write than the SQL ones.

9.1.10 Conclusion

In this section of the dissertation, we have discussed a couple of controlled experiments for quantifying the gain provided by SyQL with respect to general-purpose SQL. We can conclude by saying that SyQL improves the quality of the queries by reducing the query length. This reduction is mainly due to the possibility offered by SyQL to translate one-to-many, many-to-many, and N-to-many relationships into more concise method calls. In addition to that, we can say that SyQL can reduce the amount of time necessary to write a set of queries by 20% .

9.2 Case Study A

In this case study, we have experienced the benefit coming from the mapping of the meta-model used by PROM into a simpler one, which enables the user to retrieve the data easily by making possible complex analyses.

The *critical task* was the mapping of the bugs with the source code by using automatic collected data. These data have been gathered during an industrial experience lasting two years, where software metrics and process data were collected by using the PROM system (details are provided in Chapter 4). By using these data, a model for method fault-proneness prediction has been extracted and validated. After that, it has been embedded into the SyQL library. In this way, the user can easily predict which methods are faulty or not, making the reuse of the experience possible. By using the linguistic variables, we have smoothed the response of these prediction models, otherwise the predictions become useless due to the presence of false positive after the defect resolution. To make it possible, we also collected the criticality score for methods from the developers by using an interactive graphical user interface. A new method has the highest criticality score by default (5), when a developer decides that this method is more “mature”, he/she can gradually decrease. This score plays an important role for sharing the knowledge between the developers about the stability of

the entire system.

We want to underline that the model extraction and validation are not part of the language validation, we report these result as a compendium of our work.

In this section, we start presenting the metrics collected on the data collection site, then we show the contribution of SyQL to prepare the data set (setup of language-supported task execution phase), then we present the elaboration of data, and we discuss the performance of the different models; several machine learning and one regression techniques have been explored. Finally, we compared the performance (in terms of effort required) to write a query by using SyQL (setup of language-supported task execution step) and without using SyQL (setup of manual task execution step).

9.2.1 Data Collection Site

In this experimentation, both the project developers and the source code were involved: the development team counts 20 developers (C# certified); the source code under analysis was a part of the code base maintained by these developers (about 40%).

The code base was constituted by 99 different modules (assemblies). Every night, the C# source code metrics extractor took a snapshot of the modified/added parts of these projects.

We focus our analysis on the last six months of the experimentation (we will refer to this period with the acronym POI – “period of interest”). Before the beginning of the POI, a lot of nightly builds were broken, because the code had received major refactoring, thus, the temporal series of code metrics were discontinuous (see Figure 37).

One of the most difficult problems, which we have encountered, was reproducing the environment to success in building projects [103]. In the POI, the size of the code base was increased of 38,300 SLOC (Source Lines Of Code), at the beginning it was 56,887 SLOC, at the end it became 95,187 SLOC. At the beginning of the POI, the total number of methods was 6,750 and 11,083 at the end. The total number of classes increased from 2,594 to 4,039.

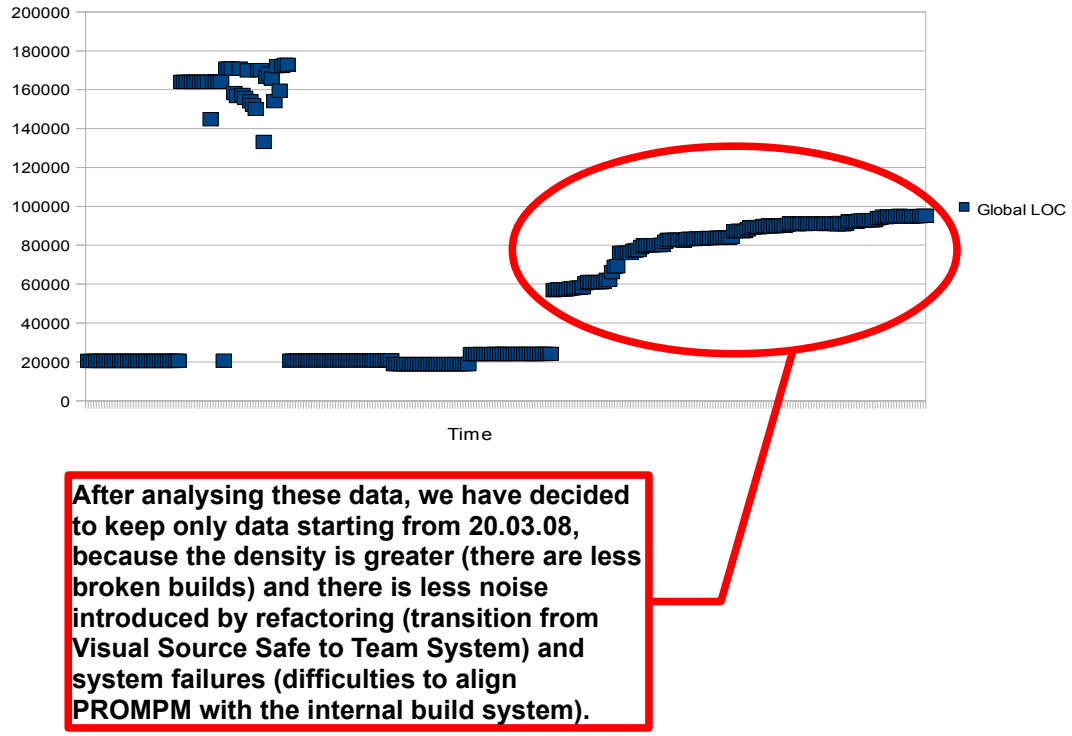

Figure 37: Trend of the global lines of code.

In this study, we have considered only the bug entries activated in the POI, which involve source code changes, and had been not reassigned. In total, we get 56 bug entries. Not reassigned means that the bug is activated (taken over) and resolved by the same person. Using not reassigned bug is very important, because our non-invasive effort measurement system can provide us the effort that each developer spent on a particular method in a specified period of time. So if a bug is switched from a developer to another, or if there are two or more developers on a single bug, then we are not able to map a bug with the corresponding portions of code.

9.2.2 Data Set Preparation

To prepare the data set, we simply run the SyQL query in Listing 80. This query works only on the bug activated starting from March 20th, 2008 (line [9]), this is the beginning of the POI. The query realizes the mapping bugs/code at lines [3]-[4], where the methods *getVSEffortMethods* is invoked. This method returns the collection of methods edited by the developer *u* during the fixing period of the bug *b*; at line [6], there is the join condition between methods and classes. I filtered out the test methods using the condition at line [10].

```
[1]  FROM    ClosedBug b, User u, Method m, Class c, Util u
[2]  WHERE   b.getResolverSurname() = u.getSurname() AND
[3]       u.getVSEffortMethods(   b.getOpeningDateTime(),
[4]                               b.getClosingDateTime()).contains(m) AND
[5]       m.getDefiningClass() <> null AND
[6]          m.getDefClassFullName() = c.getFullName() AND
[7]          (  c.getClassNamespace() <> null OR
[8]          c.getClassNamespace() <> '') AND
[9]          b.getOpeningDateTime () >= util.getDate('2008-03-20') AND
[10]            NOT m.isTestMethod() AND
[11]            (  m.getLOC(b.getOpeningDateTime()) <> -1 OR
[12]               m.getLOC(b.getClosingDateTime()) <> -1 )
[13] SELECT    m.toString(), m.getEffort().getValue(),
[14]           m.getLOC().getValue(),  m.getCC().getValue(),
[15]           m.getFanIn().getValue(), m.getFanOut().getValue(),
[16]           m.getHalsteadVolume().toString(), m.getNMI().getValue(),
[17]           m.getNMC().getValue(),  c.getLOC().getValue(),
[18]           c.getCBO().getValue(),  c.getRFC().getValue(),
[19]           c.getLCOM().getValue(), c.getDIT().getValue(),
[20]           c.getWMC().getValue(),  c.getNOA().getValue(),
[21]           c.getNOC().getValue(),  c.getNOM().getValue(),
[22]           count(b.getId())
[23] GROUP BY  m.toString(), m.getEffort().getValue(),
[24]           m.getLOC().getValue(),  m.getCC().getValue(),
[25]           m.getFanIn().getValue(), m.getFanOut().getValue(),
[26]           m.getHalsteadVolume().toString(), m.getNMI().getValue(),
[27]           m.getNMC().getValue(),  c.getLOC().getValue(),
[28]           c.getCBO().getValue(),  c.getRFC().getValue(),
[29]           c.getLCOM().getValue(), c.getDIT().getValue(),
[30]           c.getWMC().getValue(),  c.getNOA().getValue(),
[31]           c.getNOC().getValue(),  c.getNOM().getValue();
```

Listing 80: SyQL query to compute the number of bugs per method (case study A).

This query returns for each faulty method the method signature, the effort spent on this method starting from its creation, the lines of code of the class, the McCabe Cyclomatic Complexity [88], the CK metrics [31] of the defining class, etc. (the statistics of all the independent variables are provided in Table 33); at line [22] the SyQL engine computes the number of bug for the current method *m*. In total, we get 17 different independent variables. We want to warn the reader about the statistics of the class metrics, because the CK metrics statistics are higher than the ones reported by Basili *et al.* [9]. This happens due to repeated observations of the CK metrics of

the classes that define the methods.

Independent Variable	AVG	Std. dev.	Min	P25	Median	P75	Max
Effort Spent On Method	803.778	1,592.1	91	189	335	783	28,516
Method LOC	7.782	12.499	0	2	4	10	295
Method MCC	2.451	2.494	1	1	2	3	33
Method FanIn	13.945	23.405	0	4	8	16	516
Method FanOut	0.847	1.553	0	0	0	1	22
Method Halstead Vol.	361.587	659.287	2.000	62.270	167.370	430.00	13,932.6
Method NMI	1.833	4.949	0	0	1	2	91
Method NMC	5.452	6.900	0	1	3	7	78
Class LOC	237.032	344.675	1	35	95	309	1,916
Class CBO	43.612	75.565	1	10	20	47	521
Class RFC	138.390	173.528	2	28	68	195	1,165
Class LCOM	1,979.0	4,830.8	0	45	276	1,830	45,647
Class DIT	1.353	1.467	1	1	1	1	9
Class WMC	65.070	75.472	1	15	36	87	506
Class NOA	25.921	36.059	0	4	13	33	181
Class NOC	0.156	2.372	0	0	0	0	46
Class NOM	43.047	47.017	1	10	24	61	303

Table 33: Statistics of the independent variables

For extracting non-faulty methods, we did the difference between the methods present into the system at the end of the POI and the faulty methods (computed using the previous query) using the set operator <*EXCEPT*>. After that, we attach these two data sets together, obviously the non-faulty methods have 0 number of bugs. Now, SyQL cannot evaluate set operators hierarchically, so we compute the faulty and non-faulty data sets separately using two different SyQL queries, then we append the first data set to the second one.

After the data-extraction, we have computed the boolean dependent variable named "hasBug", which is true when the method is involved in at least one bug fixing procedure and the overall effort spent on this method is greater than 90 seconds. We choose to compute the dependent variable using this threshold because the initial data set contains faulty methods with only few seconds of efforts. We have checked this fact together with the developers involved in the fixing process, and we have agreed that these methods were not very relevant (most of them are accessor methods) and non-faulty. Therefore, the effort variable became a filtering attribute. Before the filtering, the data set had in total 22,082 methods (260 faulty and 21,822 not-faulty), after the filtering, the data set was reduced to 1,521 methods (175 faulty and 1346 not-faulty). In this way, the faulty and non-faulty sets have been balanced, and the ratio

between faulty and non-faulty methods has been increased. In general, it can be good for starting to extract a classification model.

9.2.3 Data Pre-processing

Before starting the model extraction, we have preprocessed the data set by using the Principal Component Analysis (PCA) [45].

This technique is widely used [96][24][76][120]. Preprocessing software metrics is necessary, because they "are often highly correlated with one another, because the attributes of software components are often related. Unfortunately, if the independent variables of a model are highly correlated, estimates of the model parameters may not be stable. In other words, insignificant variations in the data may result in drastically different parameter estimates."[76]

Independent variable	PC1	PC2	PC3	PC4
Effort Spent on Method	-0.001	-0.989	-0.150	-0.003
Method LOC	-0.000	-0.003	0.018	0.001
Method MCC	0.000	-0.000	0.002	-0.000
Method FanIn	-0.000	-0.004	0.033	0.001
Method FanOut	0.000	-0.000	0.001	-0.001
Method Halstead Volume	0.000	-0.150	0.987	-0.030
Method NMI	-0.000	0.000	0.000	-0.002
Method NMC	0.000	-0.002	0.007	0.000
Class LOC	0.055	-0.007	0.028	0.930
Class CBO	0.002	0.001	0.003	0.062
Class RFC	0.031	-0.004	0.006	0.333
Class LCOM	0.998	-0.000	-0.002	-0.063
Class DIT	-0.000	0.000	-0.000	-0.000
Class WMC	0.014	-0.002	0.007	0.079
Class NOA	0.006	-0.000	-0.001	0.077
Class NOC	-0.000	0.000	-0.000	-0.000
Class NOM	0.009	-0.001	0.000	0.051
Eigenvalue	4,842.655	1,608.128	621.532	235.525
Proportion of Variance	0.885	0.098	0.015	0.002
Cumulative Variance	0.885	0.983	0.998	1.000

Table 34: Principal components (variance threshold = 0.95)

We have used the PCA with the varimax rotation (see Table 34); the variance threshold parameter was set to 0.95, it means that the algorithm captured the 95% of the data set variance. This preliminary analysis shows that most of the variance is concentrated in five independent variables: Class Lack Cohesion Of Methods-LCOM, Effort Spent on a Method, Method Halstead Volume, Source Line of Codes-LOC of

the defining class, and Class RFC.

After a brief evaluation of the results, we have accepted these variables, because they take into the account several of the measured aspects: the effort, the coupling (Class LCOM and Class RFC), and the size (Method Halstead Volume and Class LOC). We want to evidence that the McCabe Cyclomatic Complexity [88] was not evidenced as a relevant component due to its relatively low variance.

9.2.4 Extracted Models

We have explored three different techniques: Logistic Regression, Decision Tree, PsoSVM [77]. We evaluate the prediction models in terms of correctness and completeness (see Definition 3, page 44). The model's output is a binomial variable, which can take the following values: faulty, non-faulty.

To make our analysis more robust, we perform stratified sampling on the data set with a ratio of 66 percent. With these data, we perform 10 fold cross-validation [107] for training the model. The training data consist of 120 faulty and 891 non-faulty methods. For each algorithm, we have built a model by using different groups of independent variables: all the independent variables, only the five variables selected by the PCA, and only the five variables without the effort (four independent variables). We have chosen to remove the effort, because we tried to train the models removing one of the five variables per time, and we get the best results when the effort variable was out of the training set.

According to the data used, Logistic Regression is the worst technique: the correctness of the generated model is always under 12%, probably because the problem is too complex.

PsoSVM reaches a good value of completeness (100%) with all the variables and with only five variables, but in both cases it does not have good values of correctness (54.18%), it means that these two models generate too many false positives. The situation overturns when we remove the effort, the correctness goes up (96.63%) and the completeness goes down (49.14%), probably because this algorithm is inadequate for this particular problem (see the performance summary in Table 35).

We get the best result by using the Decision Tree, the decision tree reaches 84.57% of completeness and 87.57% of the correctness with only four variables (see Table 36),

probably because the effort data are too noisy, so we think that this problem could be solved by building clusters of effort data.

			Logistic Regression	Decision Tree	PsoSVM
All variables	**Faulty**	**Correctness**	11.02%	87.33%	54.18%
		Completeness	74.29%	74.86%	100.00%
	Non-Faulty	**Correctness**	86.80%	96.79%	100.00%
		Completeness	21.99%	98.59%	89.00%
5 variables	**Faulty**	**Correctness**	11.41%	85.26%	54.18%
		Completeness	41.14%	76.00%	100.00%
	Non-Faulty	**Correctness**	88.43%	96.92%	100.00%
		Completeness	58.47%	98.29%	89.00%
4 variables (no effort)	**Faulty**	**Correctness**	11.63%	87.57%	96.63%
		Completeness	99.43%	84.57%	49.14%
	Non-Faulty	**Correctness**	96.00%	98.00%	93.78%
		Completeness	1.78%	98.44%	99.78%

Table 35: Summary of the models performances.

	true Faulty	**true Not-Faulty**	**Correctness**
pred. Faulty	148	21	87.57%
pred. Non-Faulty	27	1325	98.00%
Completeness	84.57%	98.44%	

Table 36: Contingency matrix of the best model.

9.2.5 The Model into SyQL

Finally, we have embedded the Decision Tree model in the SyQL library, now the user can simply list the potentially faulty method by writing a query like the one shown in Listing 81. The query returns all the methods that have been predicted as faulty by the model without having a low criticality score.

```
[1]  FROM Method m
[2]  WHERE   m.isPotentiallyFaulty() AND
[3]          NOT m.getCriticalityScore() IS LOW
[4]  SELECT m;
```

Listing 81: SyQL query used to predict faulty methods by using prediction models and criticality scores.

9.2.6 Standard Approach vs SyQL

After the execution of the analysis, we have asked to a developer with high skills in Java and SQL programming to reproduce our study without using SyQL (setup of manual task execution). To measure the gain given by SyQL, we have done an evaluation by measuring the time in working days necessary for building the data set. The developer could ask to an oracle information about the data warehouse. This was necessary to level the knowledge, because the SyQL user already knows the internal structure of the database. The SyQL user (us) also implemented a method during the data extraction. This can also add some bias to the evaluation result by making the acceleration factor given by SyQL lower than expected (underestimated).

By using SyQL, we spent five working days (40 working hours) to complete the task. By using a standard methodology the developer took 15 days (120 working hours) to achieve a partial result, which was close to ours but without retrieving software metrics. The developer was able only to perform the mapping between defects and methods (the critical task).

For achieving this partial result, the developer extracted the data from the PROM data warehouse and she/he inserted the data into another database, which adopts another meta-model. After that, she/he was able to produce the final data set. To do the conversion between the two meta-models, she/he used an ad-hoc program written in Java.

This made evident the need of an abstraction layer between the user and the PROM data warehouse for saving the effort of a manual/semiautomatic conversion of the data between the two meta-models (PROM meta-model and the one used for the analysis). By using SyQL, the user can skip this step. Therefore, it is possible to save time and effort during this kind of analyses. The benefits are magnified when we need to repeat the analyses on other case studies, because the production process of a particular data set is defined by a SyQL query. Moreover, we want to say that the query shown in Listing 80 (page 207) takes two hours to be executed. In Section 7.1.4 , we propose a possible solution for speeding up the execution of this query.

Summarizing, we get an acceleration factor of about 3 by using SyQL during the preparation phase of the data set. Considering, it was a first test for SyQL, we can conclude that it is a positive result.

9.3 Case Study B

In this case study, we replicate the study performed during the first industrial experience. So, the critical task was the same: map bugs with the corresponding portions of code. Compared with the case study A, this study exhibits the following differences:

- the source code is written in Java instead of C#;
- automatic effort and criticality score collection does not take place;
- the resolving bug data entries are accompanied by source code information. Therefore, the mapping between code and bugs is easier to carry out;
- the bugs are mapped with classes instead of with methods. Thus, the resulting mapping is more grainy than the previous one;
- the data collection of code metrics and bug data is performed on a longer period (3-year vs. six months).

The data collection is performed automatically by using promPM [109]. This tool allows us to process a huge quantity of source code/bytecode in a fully automated way. Again in this case study, the extraction and the validation of the prediction model is a compendium of our work.

In this section, we start introducing the data collection site, after that, we briefly present the data collection infrastructure that we adopted in this case study. Next, we present the analysis of data, and how SyQL has been used by researchers (setup of language-supported task execution step). Then, we present the results by showing the performances of the extrapolated model that classifies classes as faulty or non-faulty by using software metrics as independent variables. Finally, we compare, in terms of effort spent, the SyQL approach with the SQL one.

9.3.1 Data Collection Site

The data collection site is inside the borders of the companies that developed the project. The contributors to this project are Pro Data[18] and ASA Software House[19], which started to operate and to gain experience in the South Tyrol area in 1988 and 1989 respectively.

[18]Pro Data – Homepage – http://www.prodata.it/

[19]ASA Software House – Homepage – http://www.asaon.com/

Since the mid-1990s they formed a strategic alliance, and nowadays they provide e-commerce desktop applications for different sectors such as service trade, agro-industries and tourism, sharing the same main code base.

9.3.2 Data Collection Infrastructure

Figure 38 shows the components used to collect data. The Source Code Analysis components ran on a standalone machine. The source code was checked-out from the File control System taking the latest version of each working day.

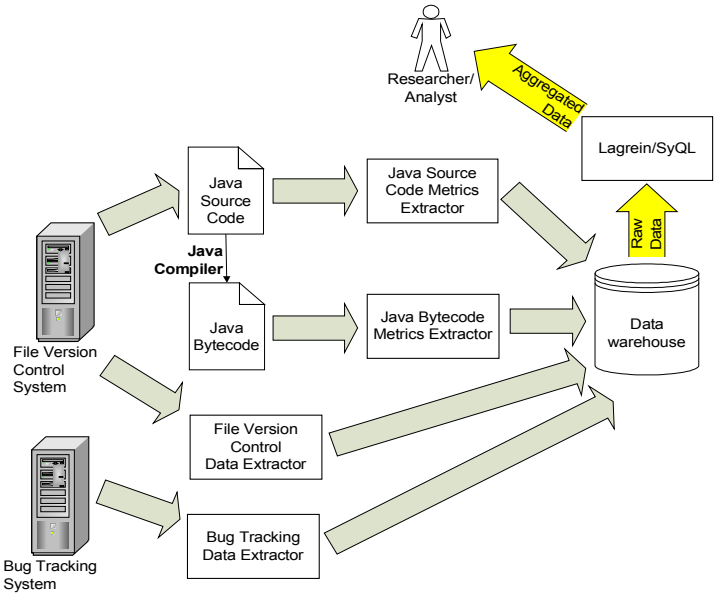

Figure 38: The System Architecture used in case study B.

The Java source code metrics extractor is based on an Island Grammar [44][91] parser; the Bytecode metrics extractor uses CKJM[20] to parse the Java bytecode.

The code analyzers store the collected data into the relational data warehouse. After that, the raw data is filtered and aggregated by using SyQL. Information contained into the File Version Control System (files addition, deletion, and modification) is collected by using an ad hoc data extractor. This component is specifically designed to extract information from the SVN repository, and it uses the log dump feature available in the most SVN command line clients. Another ad hoc software probe collects bug tracking information from the proprietary bug tracking system.

Figure 39 shows the collected LOC metric from the bytecode, we experience that the migration from Java 1.4 to Java 1.5 produces a drastically reduction of the computed Lines of Code (the reader can observe this step at the end of 2007). After a more

[20]CKJM – Chidamber and Kemerer Java Metrics – http://www.spinellis.gr/sw/ckjm/

thorough analysis of the bytecode, we have discovered that the cause was the Java 1.4 compiler, because it generates synthetic fields into the class scope more often than the version 1.5. Therefore, we decided to overcome definitively the problem by extracting the Source Lines of Code and the McCabe's Cyclomatic Complexity directly from the source code. To do that, we have used a parser written by us, which is presented in Section 4.2.3 .

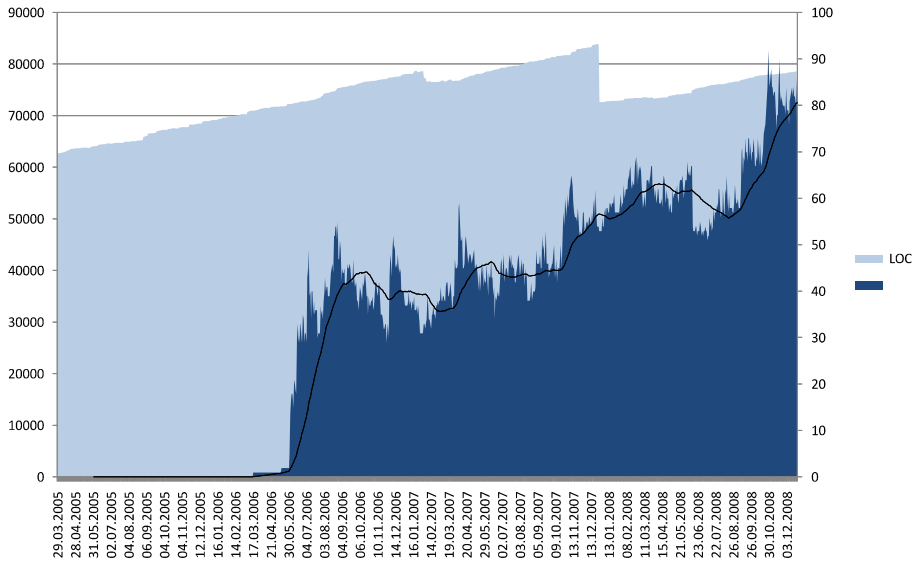

Figure 39: Preliminary Data Collected from Java bytecode

9.3.3 The Analysis

In this section, we show the results of our analysis on fault-proneness estimation. The analysis is performed over a 3-year period (2006.06.16 – 2009.02.05). We started from the mid of 2006 because the bug data start to be collected in the second quarter of the same year (see Figure 39). In this analysis, we tried to predict if a class is faulty or not. We consider the extracted software metrics as independent variables, and the faulty flag as the dependent variable. To build the training data set, we have combined the following data sources: software metrics, SVN logs, and bugs tracking system. In Listing 82, the SyQL query that generates the whole data set.

```
[1]  FROM     Class c, Bug b
[2]  WHERE    b.getModifiedClasses().contains(c)
[3]  SELECT   c.getFullName(), c.getWMC().getValue(),
[4]           c.getDIT().getValue(), c.getNOC().getValue(),
[5]           c.getCBO().getValue(), c.getRFC().getValue(),
[6]           c.getLCOM().getValue(), c.getCa().getValue(),
[7]           c.getNPM().getValue(), c.getSLOC().getValue(),
[8]           c.getCC().getValue(), c.getCalls().getValue(),
[9]           c.getUses().getValue(), count(b.getId())
[10] GROUP BY   c.getFullName(), c.getWMC().getValue(),
[11]            c.getDIT().getValue(), c.getNOC().getValue(),
[12]            c.getCBO().getValue(), c.getRFC().getValue(),
[13]            c.getLCOM().getValue(), c.getCa().getValue(),
[14]            c.getNPM().getValue(), c.getSLOC().getValue(),
[15]            c.getCC().getValue(), c.getCalls().getValue(),
[16]            c.getUses().getValue();
```

Listing 82: SyQL query to compute the number of bugs per class (case study B).

Since the location of Java source code reflects the fully qualified name of each class (e.g. "./src/com/package1/class1.java" contains the definition of the type "com.package1.class1"), we used file system locations to join software metrics with SVN logs data sources. To combine the SVN logs data source and the Bug tracking data source, we used the SVN revision number generated every time a single bug was fixed. The developers insert this revision number manually into the bug tracking system at the end of each bug fix.

The mapping process is performed by the method *Bug.getModifiedClasses()* invoked at line [2] of Listing 82. This method returns an array that contains the classes modified during the fixing process. The *contains(...)* method (see line [2] of Listing 82) is used to join classes with bugs. The method *Bug.getModifiedClasses()* is not declared in Section 5.8.3 and in Figure 22, because it has been specifically implemented for this case study. We have avoided putting it into general library, because the data about fixed classes are rarely available in the bug tracking systems of companies.

By using all these information, we have categorized the faulty and the non-faulty classes. In the next subsection, we introduce the data set. After that, we present how we have processed the data, and we discuss the obtained results.

9.3.3.1 The Data Set

In this study, we have used the CKJM and the island grammar parser. These components extract 12 different metrics for each class:

- WMC: Weighted Method per Class;
- DIT: Depth of Inheritance Tree;
- NOC: Number of Children;
- CBO: Coupling Between Object classes;
- RFC: Response for a Class;
- LCOM: Lack of cohesion in methods;
- Ca: Afferent couplings;
- NPM: Number of public methods;
- SLOC: Source Line of Code;
- CC: cumulative McCabe Cyclomatic Complexity;
- Calls: the number of methods calls into the class;
- Uses: the number of attribute accesses into the class.

Table 37 presents the descriptive statistics of the independent variables computed on the latest version of our project. The data set contains 1,516 bugs that are mapped over 2,069 faulty classes. The non-faulty classes are 2,744, in total we have 4,813 classes and not 10,716, because we consider only the latest version of the software.

Independent Variable	AVG	Std. Dev.	Min	P25	Median	P75	Max
WMC	66.80	152.95	1	12	27	67	3,415
DIT	1.78	1.15	1	1	1	2	8
NOC	0.82	4.98	0	0	0	0	147
CBO	17.08	18.08	0	7	12	21	354
RFC	70.58	87.86	1	24	46	83	14,378
LCOM	570.65	5,394.85	0	2	23	150	214,714
Ca	10.25	66.25	0	0	1	4	2,474
NPM	13.49	25.26	0	2	5	14	571
SLOC	16.28	27.96	1	4	8	18	581
CC	15.56	27.05	1	3	8	17	516
Calls	78.43	145.23	0	19	40	83	3,142
Uses	25.33	49.77	0	6	12	27	1,575

Table 37: descriptive statistics of the collected software metrics.

We examine the correlation between the variables using the Pearson Correlation Matrix (Table 38). After that, we keep only five independent variables, they are RFC, LCOM, Ca, CC, Uses. We choose these variables because the other ones (WMC, CBO, NPM, SLOC, Calls) are highly correlated with the first set (see bold values in Table 38). In addition, we have discarded DIT and NOC, because they made unstable the estimates of the model parameters, due to their very low variance (see Table 37).

	WMC	DIT	NOC	CBO	RFC	LCOM	Ca	NPM	LOC	CC	Calls	Uses
WMC	1.000											
DIT	-0.066	1.000										
NOC	0.032	-0.015	1.000									
CBO	0.572	-0.011	0.006	1.000								
RFC	**0.846**	-0.045	0.030	**0.844**	1.000							
LCOM	0.617	-0.016	0.012	0.321	0.572	1.000						
Ca	0.238	-0.008	0.153	0.105	0.212	0.197	1.000					
NPM	0.663	-0.058	0.026	0.431	**0.710**	**0.756**	0.284	1.000				
LOC	0.656	-0.014	-0.015	0.578	**0.783**	**0.660**	0.132	0.734	1.000			
CC	0.733	-0.062	0.001	0.521	**0.666**	0.258	0.090	0.335	0.515	1.000		
Calls	**0.935**	-0.047	0.028	0.726	**0.948**	0.666	0.227	0.710	0.722	0.666	1.000	
Uses	0.709	-0.085	0.006	0.510	0.721	0.519	0.177	0.640	**0.692**	0.438	0.710	1.000

Table 38: Correlation matrix of the collected software metrics.

9.3.3.2 Elaboration and Result

To improve the stability of the model, we did a PCA transformation of the selected independent variables. After that, We have trained the model using 10-folds cross-validation by using only 67% of the available examples. From the original data set, the stratified sampling algorithm extracts a subset of 3,231 classes: 1,371 faulty and 1,860 non-faulty.

	True Not-Faulty	True Faulty	Correctness
Pred. Not-Faulty	2364	369	86.50%
Pred. Faulty	380	1700	**81.73%**
Completeness	86.15%	**82.17%**	

Table 39: Decision Tree (ID3Numerical) contingency matrix.

By adopting this training schema, we can test our model on a “fresh” subset of data. This can give us a better estimation of the model performances in the real world.

Indeed, the contingency matrix (Table 39) is computed over the entire data set (4,813 examples).

Compared to the first case study, it is possible to see that the difference between the percentage values of correctness/completeness of the two sets (faulty/not-faulty) is smaller than the previous one. This probably happens due to the size of the two sets; in this case-study, it is more balanced (2,744 vs 2,069) than the first one (1,346 vs 175).

Low correctness means that a high percentage of the classes being classified as fault-prone do not actually contain a fault; low completeness means that many faults are missed. Briand *et al.* [24] obtained on Java source code 68% correctness, and 73% completeness. Our model has better performances, probably because we have performed this analysis on a longer period of time and on a larger code base. Finally, we have embedded the decision tree model into the SyQL library as described in section 9.2.5 on page 211.

9.3.4 Case Study Discussion

In the previous sections, we presented a case study, from where we have collected process metrics and product metrics. We have extracted software metrics from 995 daily-versions of an industrial Java project using promPM [109] infrastructure. The file version control information and the bug tracking data are extracted using ad hoc software probes. All the data are stored into a relational data warehouse (a PostgreSQL instance). The mining process of this quite large amount of semi-structured data requires a specific tool like SyQL to get the data in a usable format. This case study represents a good opportunity to enhance the performance of the software fault-proneness metric-based prediction models available today. Another good opportunity is the presence of the Java 1.4-1.5 migration, because it makes this case study unique. The source code changes can allow us to do size estimation of other code-bases after migration.

9.3.5 Standard Approach vs SyQL

As in the previous case study, we gave the same task to an SQL expert with the possibility to ask information to an Oracle (us). Then, we have evaluated (in terms of effort) the cost necessary to solve the same problem that we have tackled with the

support of SyQL. Indeed, by using SyQL, we have spent four working hours to execute the critical task. In this period of time, we have also modified the SyQL library by adding the *Bug.getModifiedClasses()* method. The resulting query is shown in Listing 82 on page 216; whereas, the SQL expert took about 12 working hours to complete the task. She/he started by understanding the database structure and how the data are stored, after that she/he was able to complete the critical task by writing the SQL query (see Listing 83).

```
[1]  SELECT * FROM (
[2]     SELECT smd.*, buggyClasses.numberofbugs
[3]     FROM sw_metrics_data smd LEFT JOIN (
[4]        SELECT  replace(replace( replace(sd.filename, '/trunk/src/',
[5]                        ''), '.java', ''), '/', '.' ) AS classname,
[6]                COUNT(*) AS numberofbugs
[7]        FROM bug_data AS bd, svn_data AS sd
[8]        WHERE   bd.svn <> 0 AND abg > 0 AND sd.action <> 'A'
[9]                AND sd.filename LIKE '%.java'
[10]               AND sd.filename LIKE '/trunk/src/%' AND bd.svn = sd.rev_id
[11]       GROUP BY sd.filename
[12]   ) AS buggyClasses ON buggyClasses.classname = smd.class
[13] WHERE buggyClasses.numberofbugs IS NOT NULL
[14] UNION
[15] SELECT smd.*, 0
[16] FROM sw_metrics_data smd LEFT JOIN (
[17]    SELECT  replace(replace( replace(sd.filename, '/trunk/src/', ''),
[18]          '.java', ''), '/', '.' ) AS classname,
[19]          COUNT(*) AS numberofbugs
[20]    FROM bug_data AS bd, svn_data AS sd
[21]    WHERE bd.svn <> 0 AND abg > 0
[22]       AND sd.action <> 'A' AND sd.filename LIKE '%.java'
[23]       AND sd.filename LIKE '/trunk/src/%' AND bd.svn = sd.rev_id
[24]    GROUP BY sd.filename
[25] ) AS buggyClasses ON buggyClasses.classname = smd.class
[26] WHERE buggyClasses.numberofbugs IS NULL
[27] ) AS foo
```

Listing 83: SQL query to compute the number of bugs per class (case study B).

This difference is probably related to the type abstraction mechanism that helps the engineers to work over a sort of "object oriented views".

We can conclude that SyQL has again increased the execution speed of a critical task. The acceleration factor that we have observed is about 3 (similar to the previous one).

By enhancing SyQL, we expect to increase this factor during the execution of critical tasks like the two already discussed in this section.

9.4 Lessons Learnt

In these case studies, we have learnt several lessons on different aspects and not only about SyQL. The first lesson originates from the steep barrier that we have encountered in the first industry experimentation (case study A). The second lesson comes from the research experience gained during the analysis of the collected data coming from both case studies.

9.4.1 First Lesson

The first lessons concerns the build system. As already stated by Porter *et al.* [103], the correct creation of the building environment is hard to put in practice. Because, the authors of the building scripts rely very often on the environment variables of the Operating System, which are modified during the building execution. These variables need to be correctly initialized at the beginning and disposed at the end of the building process. We want to underline that building the project is necessary to implement the software metrics extraction. Because, the software metrics extractors require the compiled code and referenced libraries to apply the language type-system rules (specific of the language) correctly. All those information must be extracted from the adopted build engine automatically.

In the first weeks of the first case study (not shown in Figure 37), we have experienced such a problem here described. The whole project was built every night, and the tests run on the continuous integration machine (the building engine adopted was MSBuild[21]). To avoid building the whole project at developing time, the developers references a set of pre-compiled libraries by specifying part of the file-paths via environment variables. Therefore, to make the build success, the continuous integration machine required a proper configuration. Maintaining this configuration requires effort, and very often they got broken builds for this reason. So, we advised them to use relative paths instead of environment variables. Thus, in less than one week, the situation improved and then the builds failed rarely.

In the second case study, we have not experienced this problem. The reason is

[21]MSDN – MSBuild Overview – http://msdn.microsoft.com/en-us/library/ms171452.aspx

twofold:

- we did not perform a daily metrics acquisition, but we have collected historical data from the SVN repository;
- the team adopted an open source build engine (ANT), hence, it has been easier for us acquiring data automatically.

Finally, we have learnt that the success of a software metrics collection project in industry depends, for the largest part, by the possibility to gather information automatically from the adopted build system.

9.4.2 Second Lesson

The second lesson concerns the SyQL language. During the analysis phase of both case studies, we have found sometimes difficult to express some types of SyQL queries. Especially during the bugs' assessment of the first case study, we have found the query writing hard to do. The difficulty depends upon the lack of the closure in SyQL. This lack makes necessary to implement very specific methods that are useless in other contexts. An example is provided for Listing 80 on page 207, where the method *User.getVSEffortMethod(...)* has been specifically implemented to make possible the mapping between methods and bugs. By using the closure, it would be possible to skip the implementation of this method. In Listing 84, we show how can be possible to solve theoretically the mapping problem by using the closure. We can see that the concept *User* has been removed from both the *FromClause*(s), because the ownership of the effort is provided by the *Effort* object (returned from *Method.getEffort(...)*), hence the concept *User* becomes useless in this query.

The result set returned by the query in Listing 80 on page 207 is the same of the query shown in Listing 84.

In Section 8.2 , the details about the SyQL closure are discussed, the benefits and the other features coming from the architecture changes are presented.

Another lack is the possibility to evaluate the set operators hierarchically (as already said in section 9.2.2). Writing multiple SyQL queries, for which the results need to be merged together, can be complex for the user, and it can be a source of problems. Because, it is easy to make mistakes when the process is not completely automatic. As soon as the data coming from other case studies data become available, the support for

hierarchies of set operators could speed up the data sets creation.

```
[1]  FROM Class c, (
[2]       FROM Method m, ClosedBug b, Util u
[3]       WHERE   m.getEffort( b.getOpeningDateTime(),
[4]                            b.getClosingDateTime()) > 0 AND
[5]               m.getEffort( b.getOpeningDateTime(),
[6]                            b.getClosingDateTime()).getUsers().
[7]                               contains(b.getResolver()) AND
[8]               m.getEffort( b.getOpeningDateTime(),
[9]                            b.getClosingDateTime()).getUsers().
[10]                              contains(b.getResolver()) AND
[11]              b.getOpeningDateTime () >= u.getDate('2008-03-20') AND
[12]              m.getDefiningClass() <> null
[13]      SELECT m, b
[14]   ) AS mapMethodBug
[15] WHERE   mapMethodBug.m.getDefClassFullName() = c.getFullName()
[16] SELECT  mapMethodBug.m.toString(),
[17]         mapMethodBug.m.getEffort().getValue(),
[18]         mapMethodBug.m.getLOC().getValue(),
[19]         ...
[20]         c.getLOC().getValue(), c.getCBO().getValue(),
[21]         c.getRFC().getValue(), c.getLCOM().getValue(),
[22]         ...
[23]         count(mapMethodBug.b.getId())
[24]  GROUP BY  mapMethodBug.m.toString(),
[25]            mapMethodBug.m.getEffort().getValue(),
[26]            mapMethodBug.m.getLOC().getValue(),
[27]            ...
[28]            c.getLOC().getValue(), c.getCBO().getValue(),
[29]            c.getRFC().getValue(), c.getLCOM().getValue(),
[30]            ...;
```

Listing 84: Partial SyQL query to compute the number of bugs per method by using the closure (theoretically).

10 Conclusion and Future Directions

This dissertation presented a query language (SyQL) for retrieving and interpreting the information stored into a relational data warehouse of software artifacts (populated by using the PROM System).

This query language is the first tool to be able to provide an abstraction for types and values at the same time. The type abstraction mechanism allows the developers to define language concepts by using a standard object oriented programming language like Java. These concepts provide the necessary information to map the underlying data warehouse meta-model into a new one that provides a higher level of abstraction. In the same way, it is possible to implement the value abstraction by providing the membership functions for the quantities of interest.

SyQL has been used in two industrial case-studies. Type abstraction helped the researchers to build data sets, whereas the value abstraction supported managers to use the prediction models in an industrial environment by taking into the account the evolution of the software development process.

During the analysis phase of the first case-study, we have come to appreciate the speedup given by SyQL at the beginning of the research process, when building a valid data set is an issue. We showed that this problem can be addressed by mapping the PROM data warehouse meta-model into a simpler one by using typical Object Oriented constructs like inheritance and late binding. The object orientation allows the developers to build an abstraction layer for the final users. In this way, SyQL users can write shorter queries than SQL ones by improving the query correctness.

As a proof of concept, we have performed a couple of controlled experiments for quantifying the gain provided by SyQL (see Section 9.1 for details). SyQL improves the quality of the queries written by the users. The correctness improvement ranges from 9.5% to 66.67% depending from the complexity of the task, and the reduction of the lines of code is up to 52.63%. In addition, the users require less effort to start using SyQL than LINQ, and they obtain better results in terms of correctness. Moreover, SyQL can reduce the amount of time necessary to write a set of queries by 20%.

Table 40 summarizes the main findings of this dissertation as the answers to the initial

two research questions.

Research question	Findings
RQ1 (Correctness)	• SyQL can download the researcher of the responsibility to understand the structure of the underlying data warehouse. • SyQL uses library concepts to implement an abstraction layer for types by hiding the internal complexity of a database. • SyQL performs all the operation in a object oriented environment. This makes possible the usage of the object oriented facilities.
RQ2 (Software Engineering)	• SyQL can help the users to interpret sets of complex metrics by using the fuzzy equal operator. • SyQL allows to reuse the past experience stored in fault prediction models. • SyQL can provide a real time situation of the software development process by helping the manager to highlight dangerous situations through software metrics and effort data.

Table 40: Main findings of thesis (answers to the initial research questions) .

In the future, we plan to use SyQL not only for process improvements, but also for process auditing and software process certification in different type of development environments like waterfall and Agile.

Moreover, we want to use the facilities provided by SyQL to support analysts of different areas during the browsing of large databases. In particular, we are looking at two specific domains: telecommunications and quality. In both these domains, it is necessary to look at metrics that are indicative of the quality of a specific service. Those metrics are often indirect metrics, so they require specific procedures for the computation that retrieve data from multiple data-sources. In addition, those procedures can be time-dependent (e.g. server up-time), and their evaluation may require experience (e.g. network latency). Therefore, the features provided by SyQL can be useful for supporting analysts who also operate outside the domain of Software Engineering.

Today, the management and the improvement of a large telecommunication system requires high-skilled persons who necessitate a good tool for planning their actions and for supporting their decisions. By using both the value and type abstraction of SyQL, an analyst can easily monitor the quality of the services provided by the system.

Further experiments, not only in the field of the Software Engineering, will help us to produce a more general-purpose query language like the one created by Comai et al.

[30] for the XML.

11 References

[1] Aho, A.V., Ullman, J.D.: The Theory of Parsing, Translation, and Compiling. Chapter 5 - One-pass no backtrack parsing. Prentice-Hall, Inc. (1972)

[2] Ambler, S.W.: Building Object Applications That Work. SIGS Books/Cambridge University Press (1998)

[3] Arisholm, E., Gallis, H., Dyba, T., Sjoberg, D.I.K.: Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise. IEEE TSE 33(2), 65--86 (2007)

[4] Auer, K., Miller, R.: XP applied. Reading, Massachusetts: Addison Wesley Professional (2001)

[5] Backus, J.W.: The syntax and semantics of the proposed International Algebraic Language of the Zurich ACM-GAMM conference. In: Proceedings of the International Conference on Information Processing, UNESCO, pp. 125--132 (1960)

[6] Badros, G.J.: JavaML: a markup language for Java source code. Computer Networks 33(1-6), 159--177 (2000)

[7] Basili, V.R.: The Role of Controlled Experiments in Software Engineering Research. Empirical Software Engineering Issues, LNCS 4336, 33--37 (2007)

[8] Basili, V.R., Perricone, B.T.: Software errors and complexity: an empirical investigation. Communications of the ACM 27(1), 42--52 (1984)

[9] Basili, V.R., Briand, L., Melo, W.L.: A validation of Object-Oriented design metrics as quality indicators. IEEE TSE 22(10), 267--271 (1996)

[10] Beck, K.: Test-driven development: by example. Addison-Wesley Professional (2003)

[11] Benlarbi, S., Melo, W.: Polymorphism Measures for Early Risk Prediction, In: Proceedings of the 21st International Conference on Software Engineering (ICSE '99), pp. 334--344 (1999)

[12] Bentley, J.L.: Programming pearls: little languages. Communications of the ACM 29(8), 711--721 (1986)

[13] Bergin, T.J., Gibson, R.G.: History of Programming languages II. ACM Press (1996)

[14] Bianco, M.: SyQL: A Tool for Analyzing the Software Development Process. In: Proceedings of GIIS 2009 Ph.D. Symposium (GIIS'2009), pp. 137--152 (2009)

[15] Bianco, M., Damiani, E.: SyQL: A Controlled Experiment Concerning the Evaluation of its Benefits. In: Proceedings of the Annual Conference on Software Engineering (SE 2010), pp. (2010)

[16] Bianco, M., Sillitti, A., Succi, G.: Fault-Proneness Estimation and Java Migration: A Preliminary Case Study. In: Proceedings of the Software Services Semantic Technologies Conference (S3T'2009), pp. 124--131 (2009)

[17] Bishop, C.M.: Neural Networks for Pattern Recognition. Clarendon Press (1995)

[18] Boehm, B.W.: Software Engineering Economics. Prentice-Hall (1981)

[19] Bonachea, D., Fisher, K., Rogers, A., Smith, F.: Hancock: a language for processing very large-scale

data. In: Proceedings of the 2nd Conference on Domain-Specific Languages. pp. 163--176 (1999)

[20] Bosc, P., Pivert, O.: SQLf A Relational Database Language for Fuzzy Querying. IEEE Transactions on Fuzzy Systems 3(1), 1--17 (1995)

[21] Bossi, A., Ghezzi, C.: Using FP as a query language for relational data-bases. Computer Languages 9(1), 25--37 (1984)

[22] Bravenboer, M., Visser, E.: Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '04), pp. 365--383 (2004)

[23] Briand, L.C., Melo, W.L., Devanbu, P.: An Investigation into Coupling Measures for C++. In: Proceedings of the 19th International Conference on Software Engineering (ICSE '97), pp. 412--421 (1997)

[24] Briand, L.C., Melo, W.L., Wust, J.: Assessing the applicability of fault-proneness models across object-oriented software projects. IEEE TSE 28(7), 706--720 (2002)

[25] Briand, L.C., Morasca, S., Basili, V.R.: Defining and Validating Measures for Object-Based High-Level Design. IEEE TSE 25(5), 722—743 (1999)

[26] C# Language Specification. Standard ECMA–334, 4th Edition. http://www.ecma-international.org (2006)

[27] Canfora, G., Cimitile, A., De Carlini, U., De Lucia, A.: An extensible system for source code analysis. IEEE TSE 24(9), 721--740 (1998)

[28] Carrasco, R.A., Vila, M.A., Araque, F., Salguero, A., Aguilar, M.A.: dmFSQL: a Server for Data Mining. In: Proceedings of the 2007 IEEE 23rd International Conference on Data Engineering Workshop (ICDEW'07), pp. 888--895 (2007)

[29] Cartwright, M., Shepperd, M.: An empirical investigation of an object-oriented software system. IEEE TSE 26(8), 786--796 (2000)

[30] Ceri, S., Comai, S., Damiani E., Fraternali, P., Tanca, L.: Complex queries in XML-GL. In: Proceedings of the 2000 ACM symposium on Applied computing - Volume 2 (SAC '00), pp. 888--893 (2000)

[31] Chidamber, S., Kemerer, C.: A metrics suite for object oriented design. IEEE TSE 20(6), 476--493, (1994)

[32] Coman, I.D.: Adoption and Usage of Automated In-process Software Engineering Measurement and Analysis Systems. PhD Thesis (2008)

[33] Coman, I.D., Sillitti, A., Succi, G.: A case-study on using an Automated In-process Software Engineering Measurement and Analysis system in an industrial environment. In: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering (ICSE'09), pp. 89--99 (2009)

[34] Common Language Infrastructure. Standard ECMA–335, 4th Edition. http://www.ecma-international.org (2006)

[35] Copeland, T.: Generating parsers with JavaCC. Centennial Books (2007)

[36] Cox, E.: Fuzzy Modeling and Genetic Algorithms for Data Mining and Exploration. Elsevier Inc. (2005)

[37] Čubranić, D., Murphy, G.C.: Hipikat: recommending pertinent software development artifacts. In: Proceedings of the 25th international Conference on Software Engineering (ICSE 2003), pp. 408--418 (2003)

[38] Damiani, E., Fugini, M.G., Fusaschi, E.: A Descriptor-Based Approach to OO Code Reuse. Computer 30(10), 73--80 (1997)

[39] Date, C.J.: A Guide to the SQL Standard (2nd Ed.). Addison-Wesley Longman Publishing Co., Inc. (1989)

[40] Date, C.J.: An introduction to database systems. Addison-Wesley Longman Publishing Co., Inc. (1986)

[41] Davison, M.R., Martinsons, M.G., Kock, N.: Principles of canonical action research. Journal of Information Systems 14, 65--86 (2004)

[42] DeMarco, T.: Controlling Software Projects. Yourdon Press, NY (1982)

[43] Demeyer, S., Tichelaar, S., Steyaert, P.: FAMIX 2.0: The FAMOOS Information Exchange Model. http://www.iam.unibe.ch/~famoos/FAMIX/Famix20/Html/famix20.html (1999)

[44] van Deursen, A., Kuipers, T.: Building Documentation Generators. In: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'99), pp. 40--49 (1999)

[45] Dunteman, G.: Principal Component Analysis. SAGE Publications (1989)

[46] Eckerson, W.W.: Three tier client/server architecture: Achieving scalability, performance, and efficiency in client server applications. Open Information Systems 10(3), 1--20 (1995)

[47] Eini, O.: The Pain of Implementing LINQ Providers. Queue 9(7), 1--13 (2011)

[48] Fenton, N.E.: Software Measurement: A Necessary Scientific Basis. IEEE TSE 20(3), 199--206 (1994)

[49] Fenton, N.E., Neil, M.: A critique of software defect prediction models, IEEE TSE 25(5), 675--689 (1999)

[50] Fenton, N.E., Ohlsson, N.: Quantitative Analysis of Faults and Failures in a Complex Software System. IEEE TSE 26(8), 797--814 (2000)

[51] Fenton, N.E., Pfleeger, S.L.: Software Metrics, 2nd ed. PWS Publishing Company (2000)

[52] Fischer, M., Pinzger, M., Gall, H.: Populating a Release History Database from Version Control and Bug Tracking Systems. In: Proceedings of the 19th International Conference on Software Maintenance (ICSM'03), pp. 23--32 (2003)

[53] Forax, R., Duris, E., Roussel, G.: A Reflective Implementation of Java Multi-Methods. IEEE TSE 30(12), 1055--1071 (2004)

[54] Witten, I.H., Frank, E.: Weka: Data Mining: Practical Machine Learning Tools and Techniques, 2nd edition. Morgan Kaufmann Series in Data Management Systems (2005)

[55] Friedman, J.: Multivariate Adaptive Regression Splines. The Annals of Statistics 19, 1--141 (1991)

[56] Fujigaki, Y.: Stress analysis: A new perspective on peopleware. American Programmer 6(7), 33--38, (1993)

[57] Gagnon, E.M., Hendren, L.J.: SableCC, an object-oriented compiler framework. In: Proceedings of the International Conference on Technology of Object-Oriented Languages, pp. 140--154 (1998)

[58] Gamma, E., Helm, R., Johnson, R., and Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., Inc. (1995)

[59] Gamper, J., Böhlen, M.H., Jensen, C.S.: Temporal Aggregation. Encyclopedia of Database Systems, pp. 2924--2929 (2009)

[60] Gill, G.K., Kemerer C.F.: Cyclomatic complexity density and software maintenance productivity. IEEE TSE 17(12), 1284--1288 (1991)

[61] Gyimóthy, T., Ference, R., Siket, L.: Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction. IEEE TSE 31(10), 897--910 (2005)

[62] Halstead, M.H.: Elements of Software Science. Operating and programming systems series 7. Elsevier Science Inc. (1977)

[63] Harrison, W.: Using software metrics to allocate testing resources. Journal of Management Information Systems 4(4), 93--105 (1988)

[64] Highsmith, J.,: Agile Software Development Ecosystems. The Agile Software Development Series, ed. A. Cockburn and J. Highsmith. Addison-Wesley (2002)

[65] Hindle, A., German, D.M.: SCQL: a formal model and a query language for source control repositories. SIGSOFT Software Engineering Notes 30(4), 1--5 (2005)

[66] Hitz, M., Montazeri, B.: Chidamber and Kemerer's Metrics Suite: A Measurement Theory Perspective. IEEE TSE 22(4), 267--271 (1996)

[67] Holt, R.C., Winter, A., Schurr, A.: GXL: Toward a Standard Exchange Format. In: Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00), pp. 162--171 (2000)

[68] Hosmer, D.W., Lemeshow, S.: Applied logistic regression. Wiley (2000)

[69] Hudak, P.: Building domain-specific embedded languages. ACM Computing Surveys 28(4) (1996)

[70] Hudson, S.E.: Cup parser generator for java. Technical report, GVU Center, Georgia Tech (1999)

[71] Jensen, C.S., Clifford, J., Gadia, S.K., Segev, A., Snodgrass, R.T.: A glossary of temporal database concepts. SIGMOD Rec. 21(3), 35--43 (1992)

[72] Jermakovics, A., Moser, R., Sillitti, A., Succi, G.: Visualizing software evolution with lagrein. In: OOPSLA Companion, pp. 749--750 (2008)

[73] Johnson, P.M., Kou, H., Agustin, J., Chan, C., Moore, C., Miglani, J., Zhen, S., Doane, W.E.: Beyond the Personal Software Process: metrics collection and analysis for the differently disciplined. In: Proceedings of the 25th International Conference on Software Engineering (ICSE'03), pp. 641--646 (2003)

[74] Jones, C.: Programming Productivity. Mcgraw-Hill (1986)

[75] Jones, C.: Spr programming languages table. release 8.2. http://www.mindspring.com/~dway/smalltalk/docs/0langtbl.pdf (1996)

[76] Kanmani, S., Rhymend Uthariaraj, V., Sankaranarayanan, V., Thambidurai, P.: Object-oriented software

fault prediction using neural networks. Information and Software Technology 49, 483--492 (2007)

[77] Kennedy, J., Eberhart, R.C.: Particle swarm optimization. In: Proceedings of the International Conference on Neural Networks, pp. 1942--1948 (1995)

[78] Kimball, R., Ross, M.: The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling (Second Edition). Wiley-India (2002)

[79] Kodaganallur, V.: Incorporating Language Processing into Java Applications: A JavaCC Tutorial. IEEE Software, 21(4), 70--77 (2004)

[80] Kosar, T., Martınez López, P.E., Barrientos, P.A., Mernik, M.: A preliminary study on various implementation approaches of domain-specific language. Information and Software Technology 50(5), 390--405 (2008)

[81] Lapierre, S., Laguë, B., Leduc, C.: Datrix™ source code model and its interchange format: lessons learned and considerations for future work. ACM SIGSOFT Software Engineering Notes 26(1), 53--56 (2001)

[82] Lethbridge, T.C., Tichelaar, S., Ploedereder, E.: The Dagstuhl middle metamodel: A schema for reverse engineering. Electronic Notes in Theoretical Computer Science 94, 7--18 (2004)

[83] Lipow, M.: Number of Faults per Line of Code. IEEE TSE SE-8(4), 437--439 (1982)

[84] Losee, R.M., Lee, A., Paris, H.: Measuring Search Engine Quality and Query Difficulty: Ranking with Target and Freestyle. Journal of the American Society for Information Science 50(10), 882--889 (1999)

[85] Marques-Silva, J., Guerra e Silva, L.: Solving Satisfiability in Combinational Circuits. IEEE Design and Test 20(04), 16--21 (2003)

[86] Martin, J.: Application Development Without Programmers. Prentice Hall PTR (1982)

[87] Martin, J.: Fourth-Generation Languages. Vol. I: Principles, Vol II: Representative 4GLs. Prentice-Hall. (1985)

[88] McCabe, T.J.: A Complexity Measure. IEEE TSE vol.SE-2(4), 308--320 (1976)

[89] Meijer, E., Beckman, B., Bierman, G.: LINQ: reconciling object, relations and XML in the .NET framework. In: Proceedings of the 2006 ACM SIGMOD international conference on Management of data, pp. 706--706 (2006)

[90] Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Computing Surveys 37(4), 316--344 (2005)

[91] Moonen, L.: Generating Robust Parsers Using Island Grammars. In: Proceedings of Eighth Working Conference On Reverse Engineering (WCRE'01), pp. 13--22 (2001)

[92] Moor, O. d., Verbaere, M., Hajiyev, E., Avgustinov, P., Ekman, T., Ongkingco, N., Sereni, D., Tibble, J.: Keynote Address: .QL for source code analysis. In: Proceedings of the Seventh IEEE international Working Conference on Source Code Analysis and Manipulation, pp. 3--16 (2007)

[93] Morasca, S., Ruhe, G.: A hybrid approach to analyze empirical software engineering data and its application to predict module fault-proneness in maintenance. Journal of System and Software 53(3), 225--237 (2000)

[94] Murphy, G.C., Kersten, M., Findlater, L.: How Are Java Software Developers Using the Eclipse IDE?. IEEE Software 23(4), 76--83 (2006)

[95] Müller, M.M.: Two controlled experiments concerning the comparison of pair programming to peer review. Journal of Systems and Software 78(2), 166--179 (2005)

[96] Nagappan, N., Ball, T., Zeller, A.: Mining metrics to predict component failures. In: Proceedings of the 28th international Conference on Software Engineering, pp. 452--461 (2006)

[97] Nardi, B.A.: A Small Matter of Programming: Perspectives on End User Computing. MIT Press (1993)

[98] Neward, T: The Vietnam of Computer Science. ODBMS.ORG (2006)

[99] Novák, V., Perfilieva, I., Močkoř, J.: Mathematical principles of fuzzy logic. Kluwer Academic, Dodrecht (1999)

[100] Nystrom, N.A., Urbanic, J., Savinell, C.: Understanding Productivity through Non-intrusive Instrumentation and Statistical Learning. In: Proceedings of the Workshop on Productivity and Performance in High-End Computing (P-PHEC), pp. 12--18 (2005)

[101] Ohira, M., Yokomori, R., Sakai, M., Matsumoto, K., Inoue, K., Torii, K.: Empirical Project Monitor: A Tool for Mining Multiple Project Data. In: Proceedings of the Workshop on Mining Software Repositories (MSR'04) (2004)

[102] Parr, T.J., Quong, R.W.: ANTLR: A Predicated-LL(k) Parser Generator. Software, Practice and Experience, 25(7), 789--810 (1995)

[103] Porter, A., Yilmaz, C., Memon, A.M., Schmidt, D.C., Natarajan, B.: Skoll: A Process and Infrastructure for Distributed Continuous Quality Assurance. IEEE TSE 33(8), 510--525 (2007)

[104] Quinlan, J.R.: C4.5: Programs for Machine Learning. Morgan Kaufmann (1993)

[105] Robbes, R., Lanza, M.: How Program History Can Improve Code Completion. In: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering, pp. 317--326 (2008)

[106] Sammet, J.E.: Programming Languages: History and Fundamentals. Prentice-Hall. (1969)

[107] Schaffer, C.: Selecting a classification method by cross-validation. Machine Learning, 13(1), 135--143 (2005)

[108] Schlesinger, F., Jekutsch, S.: ElectroCodeoGram: An Environment for Studying Programming. In: Proceedings of the Workshop on Ethnographies of Code (2006)

[109] Scotto, M., Sillitti, A., Succi, G., Vernazza, T.. A non-invasive approach to product metrics collection. Journal of System Architecture 52(11), 668--675 (2006)

[110] Scotto, M., Sillitti, A., Succi, G., Vernazza, T.: Non-invasive collection of software metrics: some issues and experiences. In: Sharing experiences on agile methodologies in open source software development, pp. 31--38. Polimetrica Publisher, Italy (2006)

[111] Sen, A., Sinha, A.P.: Toward Developing Data Warehousing Process Standards: An Ontology-Based Review of Existing Methodologies. IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews, 37(1), 17--31 (2007)

[112] Sillitti, A., Janes, A., Succi, G., Vernazza, T.: Collecting, Integrating and Analyzing Software Metrics and Personal Software Process Data. In: Proceedings of the 29th Euromicro Conference (EUROMICRO'03), pp. 336--342 (2003)

[113] Śliwerski, J., Zimmermann, T., Zeller, A.: When do changes induce fixes?. In: Proceedings of the 2005 international Workshop on Mining Software Repositories (MSR '05). pp. 1--5 (2005)

[114] Spinellis, D.: Notable design patterns for domain-specific languages. Journal of Systems and Software, 56(1), 91--99 (2001)

[115] Stringer, E.T.: Action Research. Second Edition. SAGE Publications (1999)

[116] Subramanyam, R., Krishnan, M.S.: Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects. IEEE TSE, 29(4), 297--310 (2003)

[117] Succi, G., Liu, E.: A relations-based approach for simplifying metrics extraction. ACM SIGAPP Applied Computing Review 7(3), 27--32 (1999)

[118] Succi, G., Wong, R.W.: The application of JavaCC to develop a C/C++ preprocessor. ACM SIGAPP Applied Computing Review 7(3), 11--18 (1999)

[119] Susman, G.I., Evered, R.D.: An Assessment of the Scientific Merits of Action Research. Administrative Science Quarterly, 23, 582--603 (1978)

[120] Tabachnick, G.B., Linda, S.F.: Using Multivariate Statistics, fourth edition. Boston (2001)

[121] Tomaszewski, P., Håkansson, J., Grahn, H., Lundberg, L.: Statistical models vs. expert estimation for fault prediction in modified code – an industrial case study. Journal of Systems and Software 80, 1227--1238 (2007)

[122] Wexelblat, R.L.: History of Programming Languages. Academic Press (1981)

[123] Wile, D.S.: Lessons learned from real DSL experiments. Science of Computer Programming 51(3), 265--290 (2004)

[124] Wile, D.S.: Supporting the DSL Spectrum. Journal of Computing and Information Technology 9(4), 263--287 (2001)

[125] Williams, L., Kessler, R., Cunningham, W., Jeffries, R.: Strengthening the case for pair-programming. IEEE Software 7(8), 19–25 (2000)

[126] Zadeh, L.A.: The Concept of a Linguistic Variable and its Application to Approximate Reasoning. Information Science 8, 199--249 (1975)

[127] Zadeh, L.A.: Fuzzy Sets. Information and Control (8), 338--353 (1965)

[128] Zhang, S., Lu, J., Zhang, C.: A fuzzy logic based method to acquire user threshold of minimum-support for mining association rules. Information Sciences 164(1-4), 1--16 (2004)

[129] Zhou Y., Leung, H.: Empirical Analysis of Object-Oriented Design Metrics for Predicting High and Low Severity Faults. IEEE TSE 32(10), 771--789 (2006)

[130] Zuse, H.: Software Complexity: Measures and Methods. Berlin: De Gruyter (1991)

Acknowledgments

Firstly, I would like to thank my Supervisors, Prof. Ernesto Damiani and Prof. Werner Nutt, for supporting me during these difficult years.
Secondly, I would like to thank Prof. Roberto Zicari and Prof. Dragan Gašević who have respectively reviewed Section 5 and Section 9 , and the Free University of Bolzano/Bozen staff that allows me to complete my PhD.
Finally, I would like to thank my parents that have encouraged and advised me during these years.

Appendix A

The SyQL BNF grammar

Non-Terminal Symbols:

```
[1]  Query_Node_Axiom ::=
[2]     FromClause WhereClause SelectClause ( GroupByClause )?
[3]     ( SetOperators FromClause WhereClause SelectClause
[4]     ( GroupByClause )? )* ( ";" )? <EOF>
[5]  FromClause ::=
[6]     <FROM> FromElement FromElementName
[7]     ( "," FromElement FromElementName )*
[8]  FromElement ::=
[9]     Identifier
[10] FromElementName ::=
[11]    Identifier
[12] WhereClause ::=
[13]    <WHERE> SyQLExpression
[14] SyQLExpression ::=
[15]    SyQLAndExpression ( <OR> SyQLAndExpression )*
[16] FuzzyExpression ::=
[17]    MethodCall <IS> FuzzyLabel
[18] FuzzyLabel ::=
[19]    Identifier
```

```
[20] SyQLAndExpression ::=
[21]    SyQLUnaryLogicalExpression ( <AND> SyQLUnaryLogicalExpression )*
[22] SyQLUnaryLogicalExpression ::=
[23]    (  FuzzyExpression |
[24]       ( <NOT> )? ( SyQLRelationalExpression | "(" SyQLExpression ")" ) )
[25] SyQLRelationalExpression ::=
[26]    SyQLSimpleExpression
[27]    (  (  <EQUAL> | <DIFFERENT> | <LESS_THAN_EQ> |
[28]          <GREATER_THAN_EQ> | <LESS_THAN> | <GREATER_THAN> | <LIKE>
[29]       )
[30]       SyQLSimpleExpression
[31]    )?
[32] SyQLSimpleExpression ::=
[33]    SyQLMultiplicativeExpression
[34]    ( ( "+" | "-" ) SyQLMultiplicativeExpression )*
[35] SyQLMultiplicativeExpression ::=
[36]    SyQLUnaryExpression ( ( "*" | "/" ) SyQLUnaryExpression )*
[37] SyQLUnaryExpression ::=
[38]    ( "+" | "-" )?
[39]    ( SyQLPrimaryExpression | "(" SyQLSimpleExpression ")" )
[40]    ( SyQLTemporalPostfix )?
[41] SyQLPrimaryExpression ::=
[42]    <S_NUMBER> | <S_CHAR_LITERAL> | <NULL> |
[43]    MethodCall | AggregationFunction | SyQLTemporalExpression
[44] MethodCall ::=
[45]    Identifier ( "." Identifier "(" ( Arguments )? ")" )*
[46] Arguments ::=
[47]    Argument ( "," Argument )*
[48] Argument ::=
[49]    SyQLPrimaryExpression
[50] Identifier ::=
[51]    <IDENTIFIER>
[52] SelectClause ::=
[53]    <SELECT> SelectElement ( "," SelectElement )*
[54] SelectElement ::=
[55]    SyQLSimpleExpression
[56] SetOperators ::=
```

```
[57]    <UNION> | <EXCEPT> | <INTERSECT>
[58] AggregationFunction ::=
[59]    ( <AVERAGE> | <MINIMUM> | <MAXIMUM> | <SUM> | <COUNT> | <MEDIAN> )
[60]   "(" SelectElement ")"
[61] SyQLTemporalExpression ::=
[62]    ( ( <TODAY> | <TOMORROW> | <YESTERDAY> ) ) ( SyQLTemporalPostfix )?
[63] SyQLTemporalPostfix ::=
[64]    ( ( <PLUS> | <MINUS> ) <S_NUMBER> <S_CHAR_LITERAL> )+
[65] GroupByClause ::=
[66]    <GROUP_BY> GroupByElement ( "," GroupByElement )*
[67] GroupByElement ::=
[68]    MethodCall | Identifier
```

Terminal symbols:

```
[69] SKIP {
[70]    " " | "\r" | "\t" | "\n"
[71] }
[72] <FROM> ::=
[73]    ("f" | "F") ("r" | "R") ("o" | "O") ("m" | "M")
[74] <WHERE> ::=
[75]    ("w" | "W") ("h" | "H") ("e" | "E") ("r" | "R") ("e" | "E")
[76] <SELECT> ::=
[77]    ("s" | "S") ("e" | "E") ("l" | "L")
[78]    ("e" | "E") ("c" | "C") ("t" | "T")
[79] <NULL> ::=
[80]    ("n" | "N") ("u" | "U") ("l" | "L") ("l" | "L")
[81] <GROUP_BY> ::=
[82]    ("g" | "G") ("r" | "R") ("o" | "O") ("u" | "U") ("p" | "P") (" ")+
[83]    ("b" | "B") ("y" | "Y")
[84] <TODAY> ::=
[85]    ("t" | "T") ("o" | "O") ("d" | "D") ("a" | "A") ("y" | "Y")
[86] <TOMORROW> ::=
[87]    ( "t" | "T") ("o" | "O") ("m" | "M") ("o" | "O") ("r" | "R")
[88]    ("r" | "R") ("o" | "O") ("w" | "W")
[89] <YESTERDAY> ::=
```

```
[90]    ("y" | "Y") ("e" | "E") ("s" | "S") ("t" | "T")
[91]    ("e" | "E") ("r" | "R") ("d" | "D") ("a" | "A") ("y" | "Y")
[92] <PLUS> ::= "+"
[93]
[94] <MINUS> ::= "-"
[95]
[96] <MULTIPLY> ::= "*"
[97]
[98] <DIVIDE> ::= "/"
[99]
[100]<EQUAL> ::= "="
[101]
[102]<DIFFERENT> ::= "<>" | "!="
[103]
[104]<LESS_THAN_EQ> ::= "<="
[105]
[106]<GREATER_THAN_EQ> ::= ">="
[107]
[108]<LESS_THAN> ::= "<"
[109]
[110]<GREATER_THAN> ::= ">"
[111]
[112]<LIKE> ::=
[113]   ("l" | "L") ("i" | "I") ("k" | "K") ("e" | "E")
[114]<AND> ::=
[115]     ("a" | "A") ("n" | "N") ("d" | "D")
[116]<OR> ::=
[117]   ("o" | "O") ("r" | "R")
[118]<NOT> ::=
[119]   ("n" | "N") ("o" | "O") ("t" | "T")
[120]<XOR> ::=
[121]   ("x" | "X") ("o" | "O") ("r" | "R")
[122]<IS> ::=
[123]   ("i" | "I") ("s" | "S")
[124]<UNION> ::=
[125]   ("u" | "U") ("n" | "N") ("i" | "I") ("o" | "O") ("n" | "N")
[126]<EXCEPT> ::=
```

```
[127]   ("e" | "E") ("x" | "X") ("c" | "C") ("e" | "E")
[128]   ("p" | "P") ("t" | "T")
[129]<INTERSECT> ::=
[130]   ("i" | "I") ("n" | "N") ("t" | "T") ("e" | "E") ("r" | "R")
[131]   ("s" | "S") ("e" | "E") ("c" | "C") ("t" | "T")
[132]<AVERAGE> ::=
[133]   ("a" | "A") ("v" | "V") ("g" | "G")
[134]<MINIMUM> ::=
[135]   ("m" | "M") ("i" | "I") ("n" | "N")
[136]<MAXIMUM> ::=
[137]   ("m" | "M") ("a" | "A") ("x" | "X")
[138]<SUM> ::=
[139]   ("s" | "S") ("u" | "U") ("m" | "M")
[140]<COUNT> ::=
[141]   ("c" | "C") ("o" | "O") ("u" | "U") ("n" | "N") ("t" | "T")
[142]<MEDIAN> ::=
[143]   ("m" | "M") ("e" | "E") ("d" | "D")
[144]<IDENTIFIER> ::=
[145]   (<LETTER>)+ (<DIGIT> | <LETTER> | <SPECIAL_CHARS>)* |
[146]   <S_QUOTED_IDENTIFIER>
[147]<LETTER> ::=
[148]   ["a"-"z","A"-"Z"]
[149]<SPECIAL_CHARS> ::= "$" | "_" | "#"
[150]
[151]<S_CHAR_LITERAL> ::=
[152]   "\'" (~["\'"])* "\'" ("\'" (~["\'"])* "\'")*
[153]<S_QUOTED_IDENTIFIER> ::=
[154]   "\"" (~["\n","\r","\""])* "\""
[155]<S_NUMBER> ::=
[156]   <FLOAT> | <FLOAT> (["e","E"] (["-","+"])? <FLOAT>)?
[157]<FLOAT>::=
[158]   <INTEGER> |
[159]   <INTEGER> ("." <INTEGER>)? |
[160]   "." <INTEGER>
[161]<INTEGER> ::= (<DIGIT>)+
[162]
[163]<DIGIT> ::=     ["0"-"9"]
```

Appendix B

Java Test Projects

Project ID	Project Name	LOC
1	freecol	22497
2	freemind	6247
3	EIRC	4242
4	gfx	172
5	misc5	391
6	testregex	63
7	webcam-servlet	617
8	ftpserver	1408
9	hardfilesystem	505
10	intellipack	190
11	webcam	110
12	gateway	140
13	datavision	11472
14	outliner	16502
15	iSQLViewer	20642
16	itext	41751
17	collection	1548
18	jdom	3552
19	jruby	40071
20	maverick	1030
21	jpos	10700
22	jsecurity	1653
23	mapyrus	10326
24	millstone	6643
25	jedit	41119
26	jgraphictools	269
27	antichess	5579
28	cache4j	252
29	hsqldb	35135
30	infosapient	7502
31	hibernate	36877
32	HTTPClient	5502
33	DynamicAspects	12355
34	lumbermill	3274
35	freecs	8403
36	bottomline	152
37	jdiff	4846
38	jgrapht	2036
39	Feedzeo	2027
40	htmlcleaner	1211
41	MozillaHtmlParser	556

42	hansel	1662
43	TraceLog	523
44	PDFClown	2141
45	scriptella	3161
46	Elvyx	2781
47	promTest	23
48	JIBSsource	9220
49	JExp	9091
50	JImageView	2437
51	rssowl	31658
52	jbidwatcher	13038
53	jcash	3057
54	openmap	66863
55	weka	108516
56	javaimagealbum	3886
57	currentcms	6104
58	infoglue	48201
59	ChatEverywhereClient	1019
60	log4j	6459
61	jlo	1235
62	javacc	20060
63	jakarta	901
64	checkstyle	11523
65	freenet	38175
66	jete	382
67	junit	2913
68	jython	35270
69	qform	5377
70	rmijdbc	2441
71	jorganizer	2493
72	jacorb	139177
73	ziga	31095
74	jfintrade	3351
75	jext	22181
76	jat	2954
77	jcfd	729
78	jcommon	10682
79	je	26303
80	jfreechart	52191
81	jostraca	30822
82	jpedal	36549
83	jsmooth	5408
84	jts	11393
85	juxy	1504
86	mysql-connector-java	24905
87	ngenes	4235
88	prodba	2008
89	proguard	10931
90	sitemesh	2851

91	swixat	3939
92	trove	1461
93	xdoclet	2738
94	xmlunit	1489
95	jga	5700
96	xbeans	3157
97	jodd	8718
98	pcgen	94323
99	sha4j	534
100	jfin	3056
	TOTAL LOC	**1354561**

Appendix C

DNF/CNF Conversion Algorithm

In this Appendix, we present the DNF conversion algorithm used by the query engine. The formulas converted by the algorithm are always written by humans, so they are usually not very long; it means that a sophisticated parallel implementation of the algorithm is not needed. The implementation of the algorithm is shown in Listing 85. In the Example 11, the DNF conversion algorithm is shown step-by-step.

```
[1]  PROCEDURE convert_to_DNF(Node &node)
[2]   IF(node is AndNode)
[3]       IF(node.getLeftChildNode() is OrNode)
[4]          CALL apply_distributive_property(
[5]               node,
[6]               node.getLeftChildNode(),
[7]               node.getRightChildNode())
[8]       ELSE IF(node.getRightChildNode() is OrNode)
[9]          CALL apply_distributive_property(
[10]            node,
[11]            node.getRightChildNode(),
[12]            node.getLeftChildNode())
[13]      END IF
[14]   ELSE IF(node is NotNode)
[15]      Node nextNode
[16]      nextNode = node.getChildNode()
[17]      IF(nextNode is AndNode OR nextNode is OrNode)
[18]         CALL apply_DeMorgan_theorem_recursively(node)
[19]      ELSE IF(nextNode is NotNode)
[20]         node = nextNode.getChildNode()
[21]      END IF
[22]   END IF
[23]  IF(node is AndNode OR node is OrNode)
[24]      Node leftChild, rightChild
[25]      leftChild = node.getLeftChildNode()
[26]      rightChild = node.getRightChildNode()
[27]      CALL convert_to_DNF(leftChild)
[28]      CALL convert_to_DNF(rightChild)
[29]   ELSE IF (node is NotNode)
[30]      Node nextNode
[31]      nextNode = node.getChildNode()
```

```
[32]       CALL convert_to_DNF(nextNode)
[33]    END IF
[34]  END PROCEDURE
[35]  PROCEDURE  apply_distributive_property(
[36]                Node &node, OrNode orNode, Node otherNode)
[37]     AndNode firstAndNode (orNode.getLeftChildNode(), otherNode)
[38]     AndNode secondAndNode (orNode.getRightChildNode(), otherNode)
[39]     OrNode rtnOrNode (firstAndNode, secondAndNode)
[40]     node = rtnOrNode
[41]  END PROCEDURE
[42]  PROCEDURE apply_DeMorganTheorem_recursively(Node &node)
[43]    IF(node is NotNode)
[44]     Node childNode
[45]     childNode = node.getChildNode()
[46]     IF(childNode is AndNode)
[47]        Node leftNode, rightNode
[48]        leftNode = NotNode(childNode.getLeftNode());
[49]        rightNode = NotNode(childNode.getRightNode());
[50]        node = OrNode(leftNode, rightNode)
[51]     ELSE IF(childNode is OrNode)
[52]        Node leftNode, rightNode
[53]        leftNode = NotNode(childNode.getLeftNode());
[54]        rightNode = NotNode(childNode.getRightNode());
[55]        node = AndNode(leftNode, rightNode)
[56]     END IF
[57]    END IF
[58]    IF(node is AndNode OR node is OrNode)
[59]     CALL apply_DeMorganTheorem_recursively(node.getLeftNode())
[60]     CALL apply_DeMorganTheorem_recursively(node.getRightNode())
[61]    END IF
END PROCEDURE
```

Listing 85: The DNF conversion algorithm (pseudo-code).

Example 11:

In this step-by-step example, we show how our simple implementation of the DNF conversion algorithm works. We replaced all the conditions specified into the *WhereClause* with letters from A to E.

The Boolean formula parsed by the SyQL front end is the following:

$$A \wedge \neg(B \vee \neg C) \vee \neg(D \wedge \neg E)$$

Formula 18: Formula at step 0.

The parse tree related to the above formula is the following:

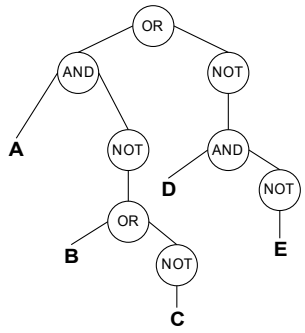

Figure 40: Parse tree at step 0.

The conditions A, D, and E are external. However, the conditions B, and C are internal. It is easy to understand that the above formula cannot be evaluated by the underlying DBMS, since it contains internal conditions. Then, we have to split it using our algorithm.

In the following figures, the trees before the transformation are displayed on the left; on the contrary, the transformed trees are displayed on the right. The nodes referenced by the *node* parameter (see Listing 85 line [1]) is highlighted by a dark circle. Only the modifying steps of the algorithm are shown.

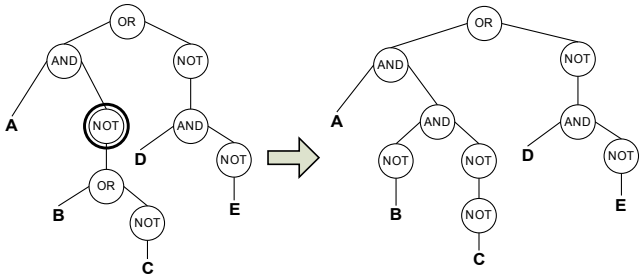

Figure 41: Parse tree at step 4.

At step 4, the algorithm applies the De Morgan's theorem (see Listing 85 line [18]), transforming the former tree (on the left) in the latter one (on the right). The resulting formula is the following:

$$A \wedge (\neg B \wedge \neg\neg C) \vee \neg (D \wedge \neg E)$$

Formula 19: Formula at step 4.

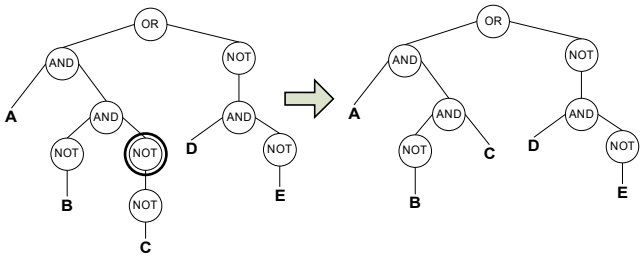

Figure 42: Parse tree at step 7.

At step 7, the algorithm detects and deletes double negations (see Listing 85 lines [19]-[20]). The resulting formula is the following:

$$A \wedge (\neg B \wedge C) \vee \neg (D \wedge \neg E)$$

Formula 20: Formula at step 7.

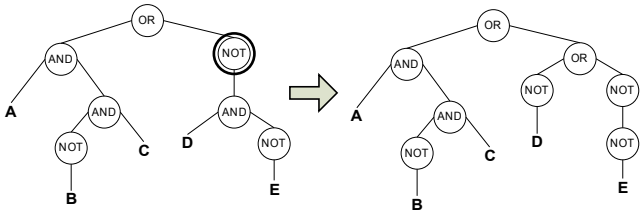

Figure 43: Parse tree at step 8.

At step 8, the algorithm applies the De Morgan's theorem (see Listing 85 line [18]). The resulting formula is the following:

$$A \wedge (\neg B \wedge C) \vee (\neg D \vee \neg\neg E)$$

Formula 21: Formula at step 8.

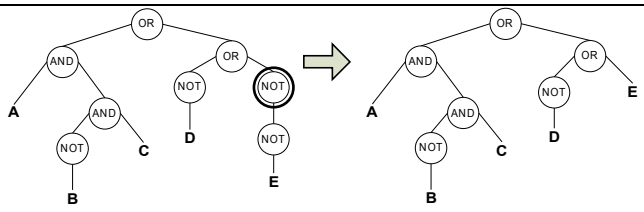

Figure 44: Parse tree at step 11.

At step 11, the algorithm deletes double negations (line 21). The resulting formula is the following:

$$(A \wedge \neg B \wedge C) \vee (\neg D) \vee (E)$$

Formula 22: Formula at step 11.

The formula above is now written in DNF notation. To convert it into CNF notation, we perform the Cartesian product between all the condition contained into the clauses of the DNF formula. At the end, the resulting formula is the following:

$$(A \vee \neg D \vee E) \wedge (\neg B \vee \neg D \vee E) \wedge (C \vee \neg D \vee E)$$

Formula 23: CNF final formula.