JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
Michael MORITZ, BSc

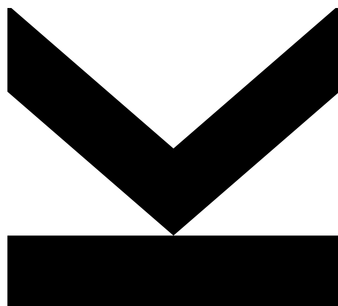
Submitted at
Institute of Business Informatics - Data & Knowledge Engineering

Supervisor
o. Univ.-Prof. Dipl.-Ing. Dr. techn. Michael Schrefl

Co-Supervisor
Ilko Kovacic, MSc

October 2020

Prototype of a Tool for Managing and Executing OLAP Patterns



Master Thesis

to obtain the academic degree of

Master of Science

in the Master's Program

Business Informatics

JOHANNES KEPLER UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at

Sworn Declaration

I hereby declare under oath that the submitted Master Thesis has been written solely by me without any third-party assistance, information other than provided sources or aids have not been used and those used have been fully documented. Sources for literal, paraphrased and cited quotes have been accurately credited.

The submitted document here present is identical to the electronically submitted text document.

Linz, 12.10.2020

Place, Date

Signature

Kurzfassung

Der OLAP-Patternansatz ermöglichen es generische Strategien zur Komposition von OLAP-Abfragen zur Befriedigung spezifischer Informationsbedarfstypen zu dokumentieren. Durch solche OLAP-Pattern können bewährte Praktiken dokumentieren werden, die innerhalb und über Domänen oder Organisationen hinweg genutzt und in zukünftigen Abfragekompositionen wiederverwendet werden können. Die notwendigen Schritte um ein OLAP-Pattern zu definieren als auch zu verwenden werden durch den OLAP-Patternansatz spezifiziert. Der Definitionsprozess beschreibt, wie ein neues Muster definiert werden kann, während der Nutzungsprozess beschreibt, wie ein OLAP-Pattern an eine spezifische Analysesituationen angepasst werden kann.

In dieser Arbeit wird ein Repository zur Unterstützung dieser Prozesse prototypisch umgesetzt. Dieses Repository für OLAP-Patterns soll als zentraler Zugriffspunkt innerhalb von Organisationen dienen und die Definition, Speicherung und Nutzung von OLAP-Patterns unterstützen. Die Kommunikation mit den Benutzer*innen erfolgt über die Eingabe von Befehlen, welche einer vordefinierten Sprache zur Definition und Verwendung von OLAP-Patterns folgen. Dabei unterstützt das Repository auch Definition von multidimensionalen Datenmodellen und den darauf aubauenden Geschäftsbegriffen, welche als Grundlage für den OLAP-Patternansatz dienen. Ein multidimensionales Datenmodell erlaubt es ein konkretes Data Warehouse einer Organisation konzeptuell abzubilden, während Geschäftsbegriffe das notwendige Geschäftsvokabular zur Beschreibung von Informationsbedarfen abbilden. Des Weiteren stellt das Repository Strukturierungs- und den Zugriffsmöglichkeiten bereit um einerseits OLAP-Patterns, Geschäftsbegriffen und multidimensionalen Modellen logisch zu strukturieren und andererseits nach ihnen zu suchen. Schließlich unterstützt das Repository die Anpassung von OLAP-Patterns an spezifische Informationsbedarfe und die Ausführung von vollständig angepassten OLAP-Patterns zum Zwecke der Generierung der benötigten OLAP Abfrage.

Abstract

The OLAP pattern approach allows to document generic strategies for composing OLAP queries to satisfy specific types of information needs. Such OLAP patterns can document best practices for the (re)use within and across domains or organizations in future query compositions. The necessary steps to define and use an OLAP pattern are specified by the OLAP pattern approach. The definition process describes how a new pattern can be defined, while the usage process describes how an OLAP pattern can be adapted to a specific analysis situation.

In this thesis a repository to support these processes is prototypically implemented. This repository for OLAP patterns should serve as a central access point within organizations and support the definition, storage and usage of OLAP patterns. Communication with users is done by entering commands that follow a predefined language for defining and using OLAP patterns. The repository also supports the definition of multidimensional data models and business terms based on them, which serve as the basis for the OLAP pattern approach. A multidimensional data model allows to conceptually map a concrete data warehouse of an organization, while business terms map the necessary business vocabulary to describe information needs. Furthermore, the repository provides structuring and access options to logically structure OLAP patterns, business terms and multidimensional models and to search for them. Finally, the repository supports the adaptation of OLAP patterns to specific information needs and the execution of fully customized OLAP patterns for the purpose of generating the required OLAP query.

Contents

1 Introduction **1**

1.1 Preface 1

1.2 Problem Statement 4

1.3 Running Example 5

1.4 Outline 5

2 Fundamentals **6**

2.1 Enriched Multidimensional Model 6

2.2 OLAP Patterns 8

2.3 Repositories 12

2.4 Language Processing 14

3 Analysis **17**

3.1 Task Description 17

3.2 Requirements 29

4 Design **45**

4.1 Architecture 45

4.2 Data Structure 46

4.3 Repository Application 50

4.4 Editor Application 70

5 Implementation **71**

5.1 Repository Application 72

5.2 Editor Application 82

6 Evaluation **83**

6.1 Repository Requirements 83

6.2 System Requirements 84

6.3 User Requirements 85

7 Conclusion **95**

Bibliography **97**

A OLAP pattern language 99

B Macro language 103

C JSON response 104

D Template Expression 107

E Generated OLAP Query 108

List of Figures

1.1 Pattern usage process following [1] 3
1.2 Use cases for an OLAP pattern repository 4

2.1 Fragment of the Austrian Milk Company's MDM 6
2.2 Fragment of the Austrian Milk Company's eMDM 8
2.3 Language processing [18, p. 10] 14
2.4 AST for mathematical expression 15

3.1 Use cases defining user requirements 30

4.1 Top-level architecture of the OLAP pattern repository 46
4.2 Class hierarchy as generated by the language processor 46
4.3 MDMElement class diagram 47
4.4 Context class diagram 48
4.5 Constraint class diagram 49
4.6 Class diagram for Context Descriptions 49
4.7 Repository application architecture 50
4.8 Generic repository architecture following Sommerville [20, p.160] 50
4.9 Communication interfaces of repository application components 51
4.10 PatternEvent class diagram 52
4.11 TermEvent class diagram 53
4.12 MDMEvent class diagram 53
4.13 OrganizationElementEvent class diagram 53
4.14 SearchEvent class diagram 54
4.15 Controller Interface 54
4.16 Procedure for processPattern method 56
4.17 Activity diagram for processPatternInstantiation method 57
4.18 Exemplary activity diagram for Catalogues in the processOrganizationElement method 58
4.19 Activity diagram for processShowStatement method 59
4.20 Language processor's arcitecture 59
4.21 language processor interface 60
4.22 Tokens of the "Jersey Breed" context (Listing 3.7) 61
4.23 AST of the "Jersey Breed" context (Listing 3.7) 62
4.24 Object representing the Jersey unary dimension predicate's context (Listing 3.7) . . . 62

4.25 Data storage interface 63
4.26 Class diagram of required and provided data storage interfaces 64
4.27 KBS interface 65
4.28 Template preprocessor interface 66
4.29 Procedure behind the executePattern method 67
4.30 AST created by the MacroParser 68
4.31 Procedure behind the expr method 69
4.32 Procedure behind the dimKey method 69
4.33 Interface provider's interface 70
4.34 Interface of the editor application 70

5.1 Deployment diagram 71
5.2 Repository application packages 73
5.3 Semantic analysis steps for the "Jersey Breed" business term 75
5.4 VariableBuilder class diagram 75
5.5 Process for persisting objects using Hibernate 78
5.6 Semantic analysis steps for the template expression in Figure 4.30 79
5.7 GUI of the editor application 82

6.1 Create "Austrian Milk Models" repository 85
6.2 Create "Dairy Precision Farming" MDM 85
6.3 Create "Time" dimension 86
6.4 Create "Feeding" cube 86
6.5 Create "Dairy Business Terms" glossary 87
6.6 Create "Jersey" business term context 87
6.7 Create "Jersey" business term template 87
6.8 Create "Jersey" business term description 88
6.9 Search for catalogues 88
6.10 Create new catalogue "Dairy OLAP Patterns" 89
6.11 Create new pattern "Breed-Specific Subset-Subset Comparison" 89
6.12 Add template to a pattern 90
6.13 Add description to a pattern 90
6.14 Instantiate "Breed-Specific Subset-Subset Comparison" pattern 91
6.15 Execute "Austrian Milk Custom Breed-Specific Subset-Subset Comparison" pattern . 91
6.16 Show "Dairy OLAP Patterns" catalogue 92
6.17 Search for patterns containing "Subset" in their names 92
6.18 Show "Breed-Specific Subset-Subset Comparison" patterns 93
6.19 Delete catalogue 93
6.20 Show deleted pattern 94

List of Tables

2.1 Pattern description 9
2.2 Pattern template 10
2.3 Requirements for content in repositories [16] 13
2.4 Requirements repositories 13

3.1 User- and System-Requirements 31

6.1 Evaluation of general repository requirements following Shahzad et al. [16] 83

Listings

2.1 Grammar for simple mathematical expressions . . . 15
3.1 Create statements regarding organizational structures . . . 18
3.2 Delete Statement for the catalogue "Dairy OLAP Patterns" . . . 19
3.3 Search statement for catalogues . . . 19
3.4 Show statement for a catalogue path . . . 19
3.5 Create “Time" dimension statement . . . 20
3.6 Create “Feeding" cube statement . . . 21
3.7 Business term context for "Jersey Breed" . . . 22
3.8 Business term description for "Jersey breed" . . . 22
3.9 Business term template for "Jersey Breed" . . . 23
3.10 Pattern context for “Breed-Specific Subset-Subset Comparison" taken from [1] with authors permission . . . 24
3.11 Pattern description for “Breed-Specific Subset-Subset Comparison" taken from [1] with authors permission . . . 25
3.12 Delete statement for "Breed-Specific Subset-Subset Comparison" . . . 26
3.13 Delete Statement for"Breed-Specific Subset-Subset Comparison"'s templates and descriptions . . . 26
3.14 String search for business terms . . . 27
3.15 String search for business terms with multiple search terms . . . 27
3.16 Show statement for a business term . . . 27
3.17 Instantiation of "Breed-Specific Subset-Subset Comparison" Pattern taken from [1] with authors permission . . . 28
3.18 Execute "Austrian Milk Custom Breed-Specific Subset-Subset Comparison" pattern 28
4.1 Uninstantiated template expression . . . 68
5.1 Excerpt of the pattern mapping . . . 77
5.2 JsonObjectBuilder usage . . . 80
5.3 AsyncResponse for responding to the editor . . . 80
5.4 RepositoryApplication class . . . 81
5.5 execute command . . . 81
A.1 Syntax of the OLAP pattern language . . . 99
B.1 Syntax of the macro language . . . 103
C.1 JSON representation of the Jersey business term . . . 104
D.1 Template Expression for Jersey business term . . . 107

E.1 Generated OLAP query 108

Chapter 1

Introduction

This section briefly introduces the basic concepts relevant for this thesis in the preface, the research gap is pointed out in the problem statement, followed by an outline of the structure of this thesis.

1.1 Preface

Strategic decision making is one of the most complex and challenging tasks in any organization and requires rational actions from the management, which is why there is a strong demand for decision support [2]. Therefore, managers answering business questions must be provided the necessary information in an appropriate level of detail [3]. Operational databases, as implemented in many companies, can hardly satisfy such a need for information for a number of reasons [4]: First of all, the data in large organizations must be located within a system of many independent databases which are widely distributed. This makes data gathering a time consuming task especially as operational databases do not perform well on operations joining many tables [5]. Secondly, the data must be integrated due to inconsistencies such as different semantics, encoding, and units of measurements. Thirdly, the data stored in operational databases is usually available on a too detailed level of granularity. Following Reichmann [3], a data summary however is more appropriate and user-friendly for decision making. Lastly, operational databases typically do not include historical data which does not allow to track changes in values over time [4]. For a strategic decision however, the trend is often more important than the current data.

Employing a data warehouse addresses the restrictions of operative databases as, according to Inmon [4], a data warehouse (DWH) *"is a subject-oriented, integrated, time-variant and non-volatile collection of data in support of management's decision making process"*. Thus, a data warehouse focuses on analytical operations performed on subjects of interest. Real world business events typically represent such subjects of interest, e.g., a sale in the retail domain. In contrast, operational databases are focused on supporting a company's applications with regard to business operations, e.g., recording a sale by storing the order along with the ordering positions for a specific customer. DWHs integrate data from multiple sources to form one globally consistent, physical data image, i.e., depending on the actual data this entails a process of converting, reformatting, summarizing, and transforming data before it can actually be stored. The time-variant aspect of a DWH is considered by associating each occurrence of a business event with the point in time it is recorded. Lastly, a DWH is non-volatile as the data stored is usually not updated, rather new data is added to capture additional occurrences of the business event.

A multidimensional data model (MDM) is used to conceptually represent a DWH [6]. An MDM represents business events in an n-dimensional data space, also denoted as a multidimensional cube. The business events represented in such a multidimensional cube, are called measures, facts, or data points. Each dimension in a MDM consists of multiple levels ordered in a granularity hierarchy. For example, for the time dimension, this may be the relationship from day to week and from week to month. Additionally, dimensions may have attributes that add descriptions to levels. Dimension can be used in multiple roles, called dimension roles, within one cube, e.g., to represent order- and shipping date with the same time dimension. These conceptual foundations allow decision makers, or more generally, every kind of domain expert, to view historical occurrences of business events and aggregate them along the dimensions for analyzing purposes.

DWH users follow the paradigm of online analytical processing (OLAP) to answer business questions; depending on the implementation as multidimensional or relational data warehouse, literature distinguishes between relational OLAP (ROLAP) and multidimensional OLAP (MOLAP) [5]. OLAP defines a set of operations that allow to analyze the data in a cube along its dimensions. These operations are roll-up (increase the aggregation level), drill-down (decrease the aggregation level), slice (restricting a single dimension), and dice (restricting multiple dimensions) [7].

For gathering and providing information to the management, reports are defined that are based on underlying OLAP queries [8]. These reports are often provided at regular intervals to monitor the effects of decisions [2]. However, reports cover only 60-80% of the information needed, which is why ad-hoc queries are additionally required [9]. The composition of ad-hoc OLAP queries, however, is a challenging task in general as it requires comprehensive knowledge. According to Allen & Parsons [10] the composition of queries from scratch consist of three steps. First, one needs to formulate the information demand in natural language. Second, the information demand needs to be adapted to fit the MDM to be considered. Third, the adapted information demand is translated into the desired query language. To perform these steps one requires beside expertise in the domain also necessary query language capabilities as well as knowledge about the underlying conceptual model of the data warehouse and it's implementation.

Working on ad-hoc OLAP queries in different domain specific projects, Schuetz et al. [11] and Kovacic et al. [8] recognized that repeating information demands can be identified and abstracted to types of information demands. Inspecting the queries composed to satisfy these information demands showed that in most cases very similar solutions, i.e., composed OLAP queries, were used. This insight led to the development of the OLAP pattern approach that provides a mean to document a strategy for OLAP query composition with regard to a specific type of information demand to be satisfied [8]. Using OLAP patterns prevents from writing similar queries from scratch over and over again. In addition, OLAP patterns could help to avoid faulty queries that result from adapting existing ones [10]. To this end, OLAP patterns record best practices, which enables them to be shared within and across organizations and domains.

Employing OLAP patterns requires business terms to be defined conceptually on top of MDMs [1]. OLAP queries – either composed ad-hoc or underlying a report – contain representations of business terms used in day-to-day interactions, i.e., the business terms are mapped to corresponding query expressions. For example, when analyzing low-income customers, a domain expert has a special group of customers with certain characteristics in mind, which can be represented by limiting the dimensions accordingly; the realization in a target query language may be a restriction of the

customers turnover. Multidimensional models that are enriched by conceptual representations of business terms are denoted as enriched multidimensional models (eMDM) [1].

Every OLAP pattern follows the same structure consisting of a descriptive and a formal part [1]. The descriptive part contains aliases, the type of information demand considered, the solution to be followed, related patterns, and exemplified applications [1]. The formal part contains pattern variables, i.e., pattern parameter and derived elements, in addition to constraints, templates, and local cubes of a local multidimensional model; pattern variables, constraints, and local cubes are also referred to as the pattern's context. For derived elements a corresponding derivation rule defines how the name to be bound can by determined, thus, avoiding the specification of additional pattern parameters. Pattern parameters and derived elements allowing for adaptation of constraints, templates, and local cubes. Constraints define conditions that need to be fulfilled in order to apply the pattern. Pattern templates represent the core OLAP query – following the solution described – as an incomplete implementation, where pattern variables and macros are interspersed [1]. Finally, local cubes allow for describing, e.g. the result of subqueries in templates, that is, local cubes are provided by the underlying templates. This is why patterns – especially domain-independent OLAP patterns – once defined can cover a large range of specific information demands.

A set of OLAP patterns, i.e., a pattern catalogue, can be defined to support future information demands [1]. Data warehouse experts are typically authors of such pattern catalogues as they provide a profound knowledge concerning the query language, the composition of queries, and the eMDM available. In contrast, domain experts use OLAP patterns in a specific analysis situation, i.e., a business context, when an information demand is occurring in the context of a decision-making process. To compose the desired OLAP query, domain experts select an appropriate pattern according to their information demand and generate the corresponding query by following the process defined by Kovacic et al. [1] consisting of pattern instantiation, pattern grounding, pattern execution, and OLAP query processing as in Figure 1.1.

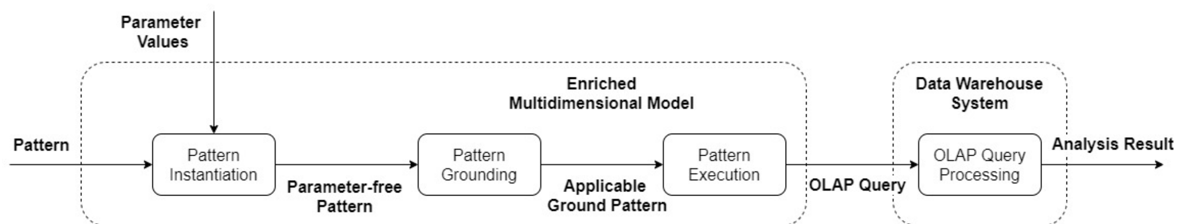


Figure 1.1: Pattern usage process following [1]

During *pattern instantiation*, values are bound to parameters in the pattern definition in order to either receive an partial-instantiated or a parameter-free pattern; the names bound correspond to elements of a domain-specific vocabulary, i.e., names of eMDM elements [1]. Derived elements are resolved during *pattern grounding* by considering the bound parameters and the associated eMDM along with existing local cubes yielding a ground pattern – derived elements are thus replaced in the pattern definition with the names determined [1]. The grounded pattern can be executed to obtain the OLAP query if it is applicable in the context of the eMDM; a ground pattern is applicable (valid) if eMDM elements or local cubes can be identified matching the bound names that further satisfy all constraints. During the execution of ground patterns the pattern's template(s) are processed by resolving macros taking into account the eMDM, resulting

in an executable OLAP query. Finally, this query can be processed by the DWH system just as any other query designed from scratch to determine analytical results from a data warehouse.

1.2 Problem Statement

The goal of this thesis is to develop a OLAP pattern repository application that provides means for storing and accessing OLAP patterns that are grouped to catalogues, cubes and dimensions that are grouped in MDMs, and business terms that are grouped in vocabularies. In addition, language statements should be processed allowing to define and use OLAP patterns as well as eMDM elements. The OLAP pattern repository application shall support the communication of both user groups, i.e., data warehouse experts that represent pattern authors and domain experts representing pattern users. A definition-centric view has to be provided to pattern authors – who currently miss a way to efficiently manage the definition and organization of patterns over language commands – allowing them to execute create, retrieve, update, and delete (CRUD) statements. A usage-centric view has to be provided to domain experts – who currently miss support to easily instantiate, ground, and execute patterns over language statements – allowing them to execute pattern usage statements. The communication of both user group views shall be based on existing language definitions (see details in the Appendix of Kovacic et al. [1]).

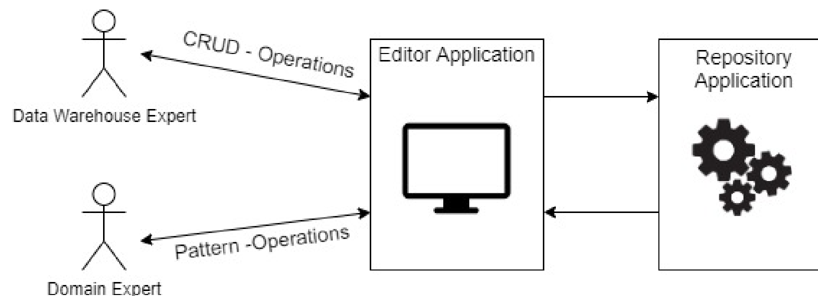


Figure 1.2: Use cases for an OLAP pattern repository

A simple command line tool, i.e., editor application, shall therefore be developed to enable the interaction between the users and the repository application (see Figure 1.2). This thesis covers the elicitation of requirements to develop a design that can finally be realized by a corresponding implementation yielding the following components:

- A repository application that is capable of processing and interpreting language statements that allows for the definition and usage of OLAP patterns and eMDM elements.
- An editor application that provides a simple interface to interact with the repository application.

The process of pattern definition and pattern usage as well as the definition of cubes, dimensions, and business terms has to follow the steps described in Kovacic et al. [1]. It is worth noting that both the implementation of pattern grounding as well as the check whether a pattern is applicable with respect to an (e)MDM are not part of this thesis, that is, they are treated as black boxes to be used.

1.3 Running Example

This thesis builds upon a running example from the precision dairy farming domain allowing to illustrate the concepts of the OLAP pattern approach as well as the functionality of the repository application. It should be noted that the example is only intended to make the explanations comprehensible but does not restrict the repository application's functionality to this domain. The fictional dairy company, following the example of Kovacic et al. [1], is called the *Austrian Milk Company* and produces milk for dairy products in various farms all over the country. To improve both efficiency and animal health the farm employs a wide range of sensors to monitor the herd. For example, the consumed feed is captured and integrated into the company's relational data warehouse; a *Feeding* cube stores the measured daily consumed feeding information, i.e., consumed (roughage in Kilogram), per cattle and farm. Built upon this data warehouse, multiple reports are regularly generated; however, these reports cannot satisfy all information demands that may arise in future analysis situations, that is, the composition of ad-hoc OLAP queries will be necessary. To minimize future efforts in composing ad-hoc OLAP queries the *Austrian Milk Company* want to employ the OLAP pattern approach.

1.4 Outline

The structure of this thesis is as follows: Section 2 introduces the theoretical foundations of this thesis. Section 3 covers a requirements analysis based on the given detailed task description. Section 4 describes the design of the overall architecture and its components, i.e., the editor, controller, language processor, repository component, template processor, and the knowledge based system. Section 5 details the implementation of these components, while Section 6 evaluates the design and implementation. Finally, a summary and a conclusion of the thesis is given.

Chapter **2**

Fundamentals

The fundamentals section presents the theoretical foundations of this thesis. First of all the enriched multidimensional model is described and exemplified using the running example introduced in section 1.3, followed by an insight into OLAP patterns. Subsequently, the state of the art for repositories and language processing is detailed. It should be noted that the description of the enriched multidimensional model and OLAP patterns is primarily based on the work of Kovacic et al. [1].

2.1 Enriched Multidimensional Model

Data warehouses (DWHs) are commonly used information sources in the strategic decision making process integrating data from distributed sources to provide a corporate view on data [4, p. 19]. The schema of a DWH is conceptually documented by a multidimensional model (MDM) depicting a multidimensional data space in which the data, i.e., the business events are entered. This multidimensional data spaces are referred to as cubes describing the occurrence of business events with measures and dimensions. Cubes can contain multiple dimensions which consist of a level hierarchy and allow to view the measures in detailed and aggregated form. In addition, attributes may be part of dimensions offering descriptions to specific levels. Dimensions may occupy different roles in one cube using different dimension roles. In general terms, cubes and dimensions are referred to as entities whereas dimension roles, measures, levels, and attributes are denoted as properties. Other properties have a value range which is defined for measures, levels, and attributes using value sets and for dimension roles using the dimension; value sets further define the values semantics [1]. Thus, only values which are part of the respective value set are valid values for measures, levels, and attributes [1].

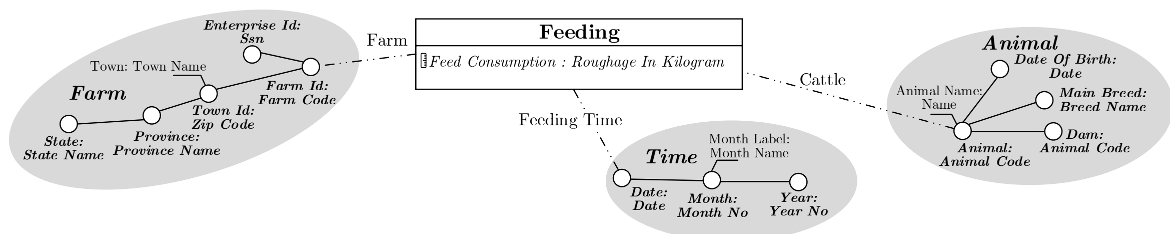


Figure 2.1: Fragment of the Austrian Milk Company's MDM

In Figure 2.1 a fragment from the MDM of the *Austrian Milk Company* is depicted. A *Feeding* cube represents daily feeding information per animal by the measure property *Feeding Consumption*. The value set *Roughage in Kilogram* determines the value type for this measure, i.e., the allowed values and their semantics. The cube further comprises a dimension role *Cattle* referencing the dimension *Animal*, as well as the dimension roles *Farm* and *Feeding Time* referencing the corresponding dimensions *Farm* respectively *Time*. Accordingly, a business event for this cube represents a feeding of a specific animal, on a specific farm at a certain point in time. These business events can be aggregated along the level hierarchies for each dimension; for example, one can aggregate feedings along the *Time* dimension from the most detailed *Date* level, over the *Month* to the *Year* level.

In addition, to multidimensional model elements business terms are defined. Business terms, are means to represent domain-specific concepts conceptually along with corresponding expressions required during the composition of OLAP queries [1]. According to the OLAP pattern approach, a business term has a business term type, reflecting the entity type it is applicable to, i.e., cubes or dimensions, the arity, i.e., the number of context parameters, and the functionality, i.e., predicate, grouping, ordering, or derived measure functionality. Business terms consists of implicitly defined context parameters (the business term type defines whether one or two cubes or dimensions are expected), constraints, descriptions, and templates [1]. Context parameters provide means to state entities to which the terms should be applied. Constraints restrict the applicability of terms by stating conditions that must hold true for the entities bound to the context parameters. Type constraints ensure that constantly defined entities of a certain type are available. A property constraint ensures that an entity referenced by the context parameter has a property of a given type. Domain constraints ensure, the entity referenced by a context parameter has a property with the given domain. It is worth noting that for business terms representing calculated measures a return type is specified.

For each business term descriptions define the semantics and provide general information on the business term and the concept it represents. A description consists of the description language, alias names of the term (if available), and the description text itself. Expression templates are representations of the business term in a specific query language and dialect. Thus, business term templates comprise language and dialect attributes, as well as an expression, which is the query snippet representing the term.

An MDM enriched by business terms is denoted as an enriched multidimensional model (eMDM). Upon the MDM fragment depicted in Figure 2.1, business terms for the dairy industry used by the *Austrian Milk Company* are defined. The business term named *Average Feed Consumption* represents a unary calculated measure that returns a measure with value from *Roughage in Kilogram*. It has one <ctx> parameter defining the cube to which it is applied. A property constraint restricts the cube to be bound to <ctx> parameter to one with a measure named *Feed Consumption*. A domain constraints restricts the cube to be bound to the <ctx> parameter to one with a measure property *Feed Consumption* of the type *Roughage In Kilogram*. As shown in the template, the term is represented by the average aggregation function applied to this measure. The other business terms in Figure 2.2 apply to dimensions, which is why each of them has a <ctx> parameter of the *Dimension* type. For the business term *Per Farm* a domain constraint restricts the applicability to dimensions with a level *Farm Id* which must be of the type *Farm Code*.

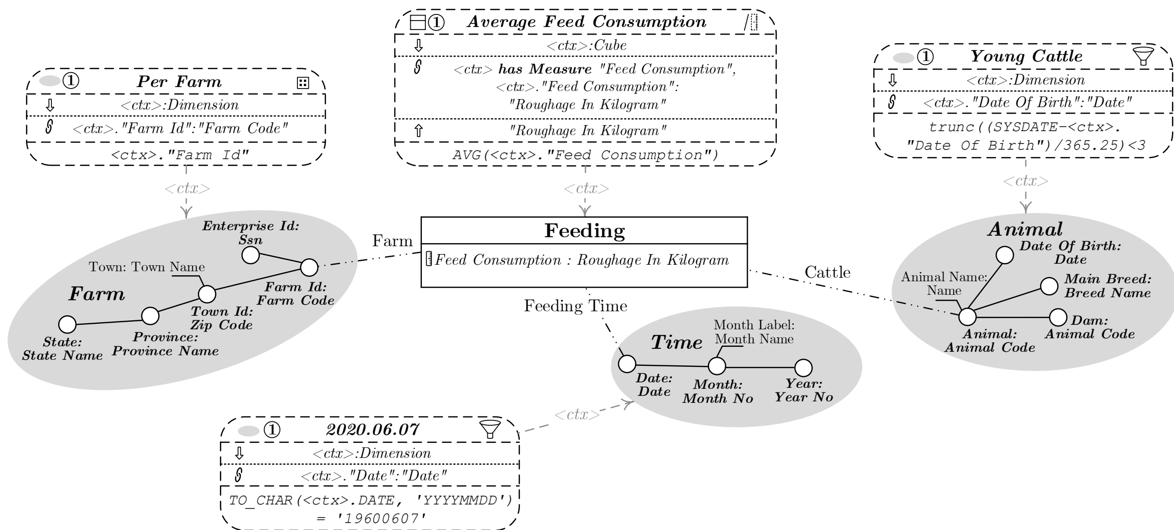


Figure 2.2: Fragment of the Austrian Milk Company's eMDM

The *Young Cattle* term is applicable to every dimension with a level *Date Of Birth* of type *Date*. Lastly, the *2020.06.07* business term, representing a specific day, applies to all dimensions with a *Date* level of the *Date* type.

2.2 OLAP Patterns

The idea of defining patterns for a domain can be traced back to the work of Alexander [12] who first introduced patterns for construction processes. He describes a pattern as a general solution to a problem frequently occurring in our environment [12, p. X]. Hence, patterns can be seen as reusable templates that allow to solve similar problems in different contexts based on existing knowledge. In order for Alexander's patterns to be share- and comparable he introduced a uniform structure for describing and documenting patterns. This structure consists of five elements, namely a unique pattern name, a context in which it can be applied, a description of the problem it solves, the generic solution to the problem and references to similar patterns [12, p. XI]. Alexander finally published a set of patterns that are applicable to various problems during the design phase of cities and buildings [12].

The notion of patterns, however, is not restricted to the architectural domain, but rather can be applied to other domain's problems as well. There are numerous works introducing patterns in various domains such as the software engineering domain. Gamma et al. [13], also referred to as the Gang of Four, introduced the notion of patterns in the context of object-oriented programming to provide solutions to frequently occurring conceptual problems to software developers. These patterns are referred to as *software design patterns* and can be divided into creational [13, p. 94], structural [13, p. 155], and behavioral [13, p. 249] patterns. Creational software design patterns focus on solutions on how to create objects independently of their representations, whereas structural patterns provide solutions for depicting relations between objects. Lastly, behavioral software design patterns provide solutions for complex control flows. As a software design pattern only describes general solution strategies, each of these patterns can be applied in any software engineering project independently of the used programming language, as long as it is an object oriented.

The idea of defining patterns for the composition of OLAP queries is shaped by Kovacic et al. [1], [8] and Schuetz et al. [11], [14]. In the context of data warehouses the use of reports is widespread to extract information and satisfy information demands. Still these reports only cover 60-80% of the actually demanded information, which is why the remaining demand must be covered using ad-hoc queries [9]. For ad-hoc query composition there are two predominant methods; either queries are composed from scratch or by adapting existing queries from previous projects. Apparently, neither is satisfying, as composing from scratch is very time consuming, whereas adapting existing queries is error prone and results in significantly worse results [10]. So, in the context of OLAP a possibility to effectively reuse existing knowledge and best practices was missing.

Schuetz et al. [14] were facing similar issues in a project concerning the dairy industry. Even though they had worked on numerous OLAP projects they were missing means to reuse the gained knowledge. Additionally, domain experts where uncertain which queries needed to be implemented; a tight schedule however did not allow to postpone query composition to the end of the project. Thus, they came up with the idea to formulate query composition strategies, i.e., OLAP patterns, that provide generic solutions for different types of information demands which were identified. Once a generic solution for a type of information demand is formulated, one can apply these solutions to specific analysis problems. This allowed to implement domain-independent OLAP queries which could be later easily adapted to specific analysis situations. Therefore, Kovacic et al. [8] define OLAP patterns as *" an instruction on how to compose an OLAP query that satisfies the information need in a specific analysis situation"*. The following sections will further discuss how such OLAP patterns are defined and used.

2.2.1 Pattern definition

A pattern is defined considering an associated eMDM, that is, the available entities, properties, business terms, and corresponding types, represent the vocabulary to be used. The definition of patterns is twofold comprising a system-independent and a system-specific part. The system-independent part consists of a pattern context and one or more textual definitions called pattern descriptions. A *pattern description* consists of a predefined structure allowing users to compare patterns and decide on the applicability to their specific information demand (see Table 2.1). This structure consists of a list of possible alternative pattern names (aliases), a problem description, the solution idea behind the pattern, an example as well as other patterns that are related [1].

#	Element Name	Comment
1	Name	A distinctive name within the pattern
2	Language	Description's language like English, German etc.
3	Alias	Other names by which the pattern is known
4	Problem	Information demand the pattern aims to satisfy
5	Solution	How the information demand is satisfied
6	Related	Other patterns this is related to
7	Example	Exemplary usage situation

Table 2.1: Pattern description

A *pattern context* consists of a unique pattern name as well as definitions of pattern parameters, derived elements, constraints, and local cubes [1]. Parameters allow to adapt patterns to specific information demands. Derived elements follow a derivation rule and are constructed from parameters regarding a specific eMDM; the derivation rule is either based on the domain of an entity's property or the return type of a business term. It is not necessary to include derived elements in a pattern context, however, they simplify the usage as less parameters need to explicitly specified. Constraints are means to define relationships between parameters, derived elements, and constant values that must hold true for an eMDM or the local cubes in order for the pattern to be applicable [1]. Besides type, property, and domain constraints, return and applicable-to constraints can be defined in a pattern, where return constraints define the expected return type of a calculated measure and applicable-to constraints define business terms to be applicable to certain entities [1]. Local cubes of a pattern's allow to represent the result of subqueries provided by the template(s), i.e., these elements are not actually part of the eMDM but are available solely within the pattern [1]. This allows, for example, to define the application of parameters that refer to business terms to the interim result obtained by those subqueries.

#	Element Name	Comment
1	Name	A distinctive name within the pattern
2	Language	The expression's query language, e.g., SQL
3	Dialect	The language dialect, e.g., PostgreSQL
4	Expression	The basic query structure of the pattern

Table 2.2: Pattern template

The pattern context, as well as the pattern description, is independent of any system or query language and thus, makes the approach, just like software design patterns, applicable in different contexts [1]. In contrast to software design patterns, OLAP patterns, however, additionally offer a system-specific part, namely the *pattern templates*. Templates are incomplete implementations of OLAP queries representing the realization of the solution strategy while considering a specific target language, system, and data model; pattern templates can be seen as blueprints for writing actually applicable queries [1]. A template definitions includes the query language, language dialect, and an expression [1] (see Table 2.2). Expressions are the actual representation of the pattern in the specific language and dialect, containing placeholders for parameters and derived elements as well as macros making them adaptable to related information demands. Macros in the context of OLAP patterns are small functions used in the template expressions that are called during the execution to gain and substitute a code snippet [1]. Multiple templates allow a pattern to be used for either different languages, modeling paradigms such as star- or snowflake-realizations, data models such as MOLAP or ROLAP, or database management systems.

2.2.2 OLAP Pattern Usage

A pattern, once defined, is a documentation of a solution strategy to compose OLAP queries that satisfy a certain type of information demand. To apply an OLAP pattern to a specific analysis situation, however, additional steps are required. For this reason the pattern usage process is defined allowing for the adaptation to specific analysis situations (Figure 1.1). Accordingly, to use an OLAP pattern in a specific analysis situation the steps of instantiation, grounding, execution,

and query processing must be followed [1].

During the *instantiation* of an OLAP pattern, names are bound to the pattern's parameters [1]. These names represent the names of eMDM elements of an associated eMDM. For example, if the value bound to a parameter of type cube is **"Feeding"**, then the eMDM must provide a cube named **"Feeding"**.

The result of the instantiation of a pattern is a new more specific pattern, that is, a pattern with less unbound parameters. Pattern instantiations do not require to bind names to all parameters; if names are only bound to a subset of parameters, a partially instantiated pattern is obtained [1]. In such a partially instantiated pattern only a subset of parameters is left for the user to be specified, thus the pattern is more specific and can only be adapted to a smaller number of information demands [1]. This allows to create specialization hierarchies of patterns reaching from abstract to specific as further describes in subsection 2.2.3. However, to continue the pattern usage, the pattern must be fully instantiated, meaning that no parameters are left unbound.

Grounding a parameter-free pattern is an optional step, as it is required only if derived elements are defined in the pattern context [1]. It refers to the process of evaluating derivation rules considering the associated eMDM along with the pattern's local cubes in order to obtain the names to be bound to the derived elements [1]. If derivation rules cannot be evaluated in the context the associated eMDM and the pattern's local cubes, the pattern cannot be applied.

OLAP patterns where names are bound to all parameters and derived elements are denoted as ground patterns that can be *executed* [1]. The execution converts one or all of the pattern's templates to an executable OLAP query. The pattern execution starts with performing a validation that ensures that the pattern can actually be applied to the eMDM while considering the pattern's local cubes; if the pattern is not applicable, the process ends at this point. A ground pattern is considered applicable (valid), if the names in constraints can be matched to eMDM elements or the pattern's local cubes that fulfill all pattern constraints. The macro calls within the template(s) are then being processed, aiming to get an expression snippet corresponding to the language and dialect used in the pattern template. The expression snippet obtained is adapted considering the parameters provided, and substitutes the corresponding macro call in the template.

The two macros that can be distinguished are the *$dimKey* and *$expr* macro. The *$dimKey* takes a dimension's name as parameter and returns the corresponding base level; a level which no other level rolls up to. The *$expr* macro on the other hand obtains the expression of the business term referred to by the first parameter, and replaces the context parameters in the expression according to the values bound in the *$expr* macro call; hence, it has multiple parameters including the business term name, and the values for its parameters [1]. As a result the macro returns the term's expression that is free of parameters and thus adapted to the analysis situation in the matching language for the template. If the business term does not include an expression template in the same language and dialect as the pattern template to be executed, the execution fails.

2.2.3 Pattern Organization

OLAP patterns are generic instructions to compose OLAP queries satisfying a certain information demand. Nevertheless, by instantiating the patterns it is possible to further specify them and construct pattern hierarchies from generic to specific. Hence, patterns can reach from domain-

independent, which is the most generic form, to organisation-specific. The border between the hierarchy levels however is not a clear cut but rather a fluent transition [1].

A generic or domain-independent pattern is characterized by a wide range of applicability [1]. That is, a domain-independent pattern provides solutions for information demands that frequently occur across domains. The idea behind OLAP patterns is, that such generic solutions can be adapted to solve similar domain- or organisation-specific problems in the future. As an example a pattern comparing two subsets (**"Subset-Subset Comparison Pattern"**) is something that can be applied in any domain on any two subsets to be compared [1].

In contrast to domain-independent patterns, domain-specific patterns are either instantiated or defined in such a way, that makes them only applicable for a certain domain [1]. Within this domain the patterns are however still flexible enough to be applied in different organisations. Just as described in subsection 2.2.1 for the definition, the eMDM also determines the vocabulary to be used in the instantiation. That is, through a targeted selection of the eMDM's terms, one can shape a generally applicable pattern to be restricted to a certain domain. Consider the *Austrian Milk Company* using the **"Subset-Subset Comparison Pattern"** in the form of a **"Feeding Subset-Subset Comparison"**. If the pattern is restricted to analyze a *Feeding* cube, its applicability is restricted to the domain of animal owners; i.e., it can also be applied to a zoo. In addition, if the subsets are restricted to *Cattle*, the pattern is still domain specific, as it can be applied to all cow holders, but more restrictive as it is not applicable to zoos anymore.

The most specific patterns are represented by organization-specific patterns [1]. A pattern can be deemed organization-specific if it is (fully) instantiated using the elements contained in an organization's specific MDM, however, the pattern can still be used for different applications within this organization. Considering an instantiation of the **"Feeding Subset-Subset Comparison Pattern"** that compares the set of Jersey cattle with the set of Highland cattle, one can imagine, that such a specific pattern will be applicable to hardly any other company than the *Austrian Milk Company*. Within the *Austrian Milk Company* the pattern can however be use in one application to compare the breeds regarding feed consumption and in another regarding milk yield.

2.3 Repositories

In order to create and persist patterns and their hierarchies a central point of storage is needed, that allows to define, access, and use OLAP patterns. This section thus describes the concept of a repository as a generic approach for storing and accessing arbitrary data objects.

In the field of business process modelling a repository is a state of the art approach to support the reuse and administration of process models [15], [16]. Repositories are defined on a very abstract level as a *"...central component for storing, managing and changing process knowledge"* [15]. In order to gain maximum advantage from using a repository, Shahzad et al. [16] introduce generally applicable requirements that need to be fulfilled. These requirements concern on the one hand the repository itself and on the other hand the content it stores. For any data object it needs to be clarified at first whether or not the criteria formulated for content shown in Table 2.3 are fulfilled.

Table 2.3: Requirements for content in repositories [16]

#	Requirement	Description
1	Reusable	Content is reusable if it can be used for multiple purposes either with or without modifying and adapting it.
2	Language independent	The content stored in the repository must be independent of the target languages it should be used for. In the context of business processes this means that process models must be independent of modelling languages.
3	Domain and organisation independent	This requirement describes the general usability of repositories for content coming from various application fields.

Re-usability states that users must be able to retrieve the repository's content and make it applicable to similar contexts by modification and adaption [16]. Language independence is more specific to the initial focus on process models, however it can also be interpreted the necessity for technology- or system-independent content [16]. Domain- and organisation-independent states that the repository must not restrict the usability of its content, e.g., as process models are generally applicable, a repository must not be restricted to a specific domain's process models [16].

Besides the requirements for the content to be stored in an repository, Shahzad et al. [16] introduce six requirements for repositories. By exploring the functionality of numerous existing repositories the generally applicable requirements stated in Table 2.3 could be identified.

Table 2.4: Requirements repositories

#	Requirement	Description
1	Extensible	A repository must allow to add and modify content to include new knowledge.
2	Flexible	Within the repository multiple versions of content must be allowed. This refers to the need of adapting and storing content which apply to certain contexts.
3	Openness	Being open means that the repository should be unrestrictedly available to any possible user.
4	Accepting	A repository should allow to structure its content according to business needs following different schemes.
5	Usable	The access of the repository's content should be allowed via a easy-to-use, usually graphical, interface.
6	Navigable	It should be possible to browse and search for content.

Extensibility means to provide methods to add content, i.e., repositories must consider that future situations may require to add new knowledge. As the content itself must be reusable, the repository must also allow to persist these adapted or modified versions, which is what the authors define as flexible. Openness, is a feature that allows users to perform Create-/Update-/Delete- (CRUD) operations without any need for permission. Accepting for repositories states that one should be free to organize the content in a way that best suits the users. Usability means that using the repository must be considered easy by the user. Hence, this requirement includes a certain subjectivity, as it does not determine the term "easy". Finally, a navigable repository must enable the user to search the content using search criteria.

2.4 Language Processing

A repository for OLAP patterns shall be maintained using textual commands following a predefined OLAP pattern language (see section 1.2). Languages are defined by a grammar, which is a set of rules that determines how compliant statements must look like. Processing language statements regularly consists of three steps, i.e., lexical, syntactical, and semantic analysis [17]. These steps are typically applied in compilers, where language processing is an intermediate step in the transformation of program code to an executable program [18]. For this thesis, however, the process depicted in Figure 2.3 is sufficient, taking the statement and the grammar as input to generate an abstract syntax tree (AST) as output.

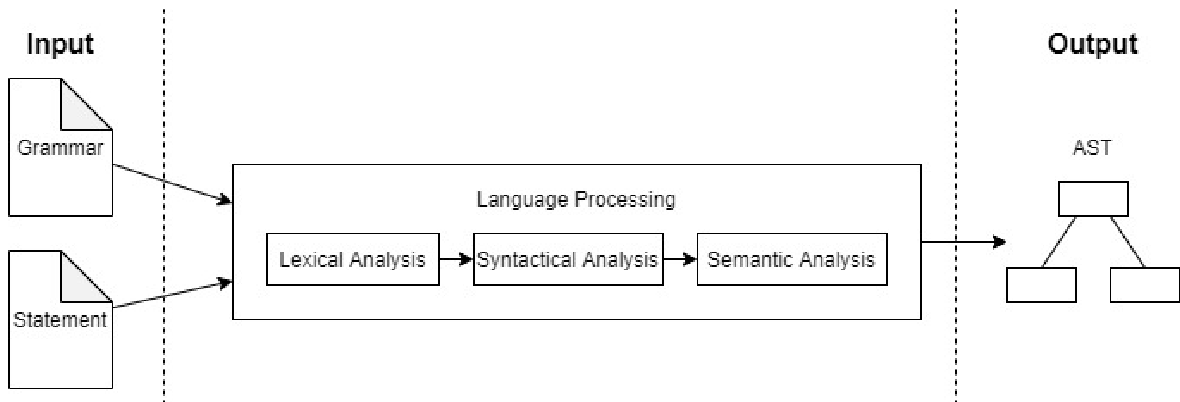


Figure 2.3: Language processing [18, p. 10]

To make this process comprehensible the Listing 2.1 depicts an example grammar for simple mathematical expressions containing only one-digit numbers and trigonometric functions. The rule in line 1 states, that an expression consists of numerous terms connected by a **"+"** or **"-"** operator. A term, as stated in line 6, aggregates any number of Factors connected by a **"*"** or **"/"** operator. Factors (line 11) are either one-digit numbers, parenthesized sub-expressions or function names with parenthesized sub-expressions. The function rule (line 17) depicts which function names are valid for the language, i.e. **"sin"**, **"cos"** and **"tan"**.

Processing, for example, the mathematical expression $5 + 8 * 9$ requires the consideration of the steps depicted in Figure 2.3. During the *lexical analysis* the input statement, which is only a sequence of character symbols at this time, is aggregated in form of tokens. *"A lexical token is a sequence of characters that can be treated as a unit in [...] the language"* [18, p. 24]. That is, anything under quotation marks in Listing 2.1 is a token in the example grammar. The aim during the lexical analysis task however, is not to find any matching token, but the longest possible. Using this approach avoids unmatched character sequences at the end of the process, i.e., if any character in the input statement could not be matched, it is not a valid statement. For this example the following tokens are recognized: **"5"**, **"+"**, **"8"**, **"*"**, **"9"**, i.e., the statement is valid and consists of five tokens with one character. If the statement included, e.g., a logical operator like **"&"**, which is not a valid sign in a mathematical expression, the result of the lexical analysis would be that the whole statement is invalid.

```
1  Expression:
2    Expression "+" Term |
3    Expression "-" Term |
4    Term;
5
6  Term:
7    Term "*" Factor
8    Term "/" Factor
9    Factor;
10
11 Factor:
12   "0" | "1" | "2" | "3" | "4" |
13   "5" | "6" | "7" | "8" | "9" |
14   Function "(" Expression ")" |
15   "(" Expression ")";
16
17 Function:
18   "sin" | "cos" | "tan";
```

Listing 2.1: Grammar for simple mathematical expressions

In a second step the lexically correct and tokenized input sequence is further syntactically analysed; this process is also referred to as *parsing*. Therefore, the token sequences are analysed to check whether expressions can be formed that correspond to grammar rules, i.e., to form valid expressions the tokens must appear in the same order as defined in a grammar rule. Corresponding to the rules in the grammar, the matched token sequences form a hierarchy that is regularly visualized as an abstract syntax tree (AST). Every leaf in such an AST refers to one token generated in the previous lexical analysis step; every other node refers to a rule name respectively.

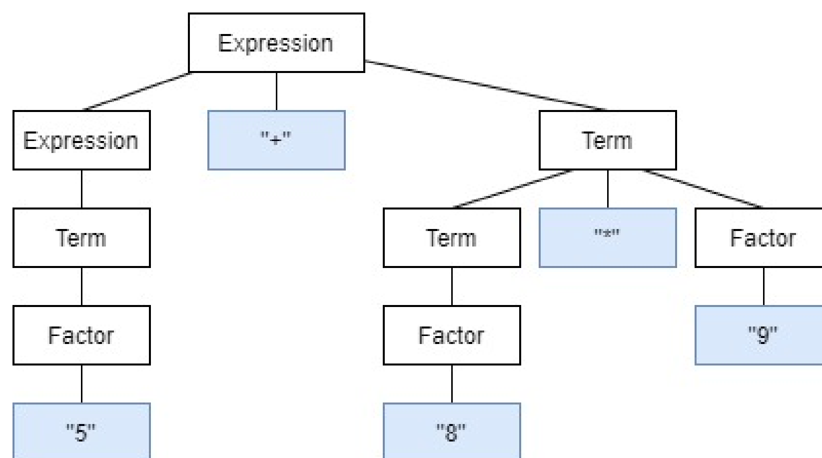


Figure 2.4: AST for mathematical expression

An example AST can be found in Figure 2.4 visualizing the mathematical example expression. There is only one mathematical expression included matching the rule in line 2, i.e., an addition. Regarding the left side of the addition, the expression rule in line 4 is matched, i.e., it just consists of a term. Further, the term is matched to the rule in line 9, hence the expression consists only of one factor, which is **"5"**. Equivalently the right side of the addition is matched with the

corresponding rules. In the context of parsing a distinction between contextual and context-free grammars is necessary; in contextual grammars tokens surrounding a certain token sequence determine the rule to be applied, whereas for context free grammars the surrounding tokens are not considered. For this work, however, the focus is on context-free grammars only, as the OLAP pattern grammar can be classified as such.

Finally, a *semantic analysis* is performed on the AST by iterating over the nodes, which generate the output of the process (Figure 2.3), i.e., an attributed AST. In this step attributes, further describing the nodes, are generated and added to the tree like, e.g., a data type [19, p. 153]. However, it not only adds attributes but also checks for correct usage. In the context of mathematical expressions semantic analysis checks, e.g., that there is no division by zero. Other than Waite & Goos [19], Appel [18, p. 94] describes that the output of a semantic analysis need not be an AST again but rather a simpler structure suitable for further processing.

Irrespective of whether the output is an AST or not, the language processing task provides a syntactically and semantically correct structure representing the statement allowing for further processing. The processing steps however strongly depend on the task and cannot be generalized. For a mathematical statement it could be, e.g, the conversion of the expression into post-fix notation, but also to solve the expression like in a calculator.

Chapter 3

Analysis

This section analyses which functionality a repository application must provide, considering the fundamentals (chapter 2) and the problem statement (section 1.2). Therefore, the task description (section 3.1) analyses the OLAP pattern language (Appendix A) and the necessary statements the repository must process. Further, the requirements in section 3.2 depict scenarios describing how the repository application must support the users and system requirements stating what is necessary to provide this functionality; the requirements mirror the functionality described by Kovacic et al. [1]. It is worth noting that the functionality of the editor is not detailed, as it should represent a plain user interface without any functionality.

3.1 Task Description

The goal of this thesis is to develop a repository application that supports both domain and data warehouse experts in the management and usage of OLAP patterns and eMDMs (business terms as well as MDMs). Thus, it is intended to be a central point of storing and accessing patterns within an organization. Such a repository application must consist of both organization elements and content elements. Organization elements should provide means to structure the content and are represented by repositories containing catalogues, glossaries, and MDMs. Content elements are represented by OLAP pattern definitions, business terms definitions, cube and dimension definitions (the actual information necessary to compose OLAP queries). A pattern definition consists of a pattern context representing pattern parameters, derived elements, constraints and local cubes, one or more pattern templates representing the generic query structure and one or more pattern description providing useful information. Consequently, a business term consists of a business term context (defining the context parameters and constraints), templates and descriptions. A cube definition comprises the specification of cube properties, i.e. dimension roles which reference dimensions and measures that are restricted to a type using value sets. Dimension definitions contain the specification of levels and attributes as well as the relationships between these elements, i.e. "roll-up" relations between two levels and "described-by" relations between levels and attributes. The tasks concerning operations for organization elements are described in subsection 3.1.1. How the repository content should be created, retrieved, updated, deleted, and used is described in subsection 3.1.2. Since the communication with the repository application should be via OLAP pattern language commands (section 1.2) each subsection depicts corresponding example statements the repository application must be able to process.

3.1.1 Organization Elements: Repositories, Catalogues, Glossaries, and MDMs

The repository application developed in this work must be able to group content elements inside repositories that comprise MDMs, glossaries and catalogues. That is, the repository should provide a structure for each type of content, i.e. cubes and dimensions, business terms and patterns. Cubes and dimensions should be grouped by the repository, using *multidimensional models*; they depict the conceptualization of an actual data warehouses used by an organization. The groupings for business terms that must be supported are called *glossaries* and should represent the wording used in a specific domain or organization. Further, a repository should allow to organize patterns in *catalogues*, which are sets of patterns that are frequently used, for example, by a certain department. The names inside an organization elements must be unique, as all elements should be referenced solely by name. The according tasks for the repository application, namely creating, retrieving and deleting (section 3.1.1) organization elements are specified in the subsequent sections.

Defining Organization Elements

The definition of organization elements should be supported using the corresponding language statements determined by the OLAP pattern language (Appendix A). Each statement starts with the `CREATE` keyword, followed by the structure element's designation to be created, i.e., the type of the structure element, and a path expression. For the repository – the top level structure element – the path only consists of its name. For all other organization elements the path consists of the repository name, a slash and the actual name for the element to be created. To allow names containing white spaces the whole language is based on quoted identifiers. It is worth noting, that no means for restructuring are intended, i.e., once a catalogue, glossary, or MDM is created inside a repository, it cannot be moved to another repository.

```
1 CREATE REPOSITORY "Austrian Milk Models" ;
2 CREATE CATALOGUE "Austrian Milk Models"/"Dairy OLAP Patterns";
3 CREATE GLOSSARY "Austrian Milk Models"/"Dairy Business Terms";
4 CREATE MULTIDIMENSIONAL_MODEL "Austrian Milk Models"/"Dairy Precision Farming";
```

Listing 3.1: Create statements regarding organizational structures

Listing 3.1 depicts the statements used to create a repository structure for the *Austrian Milk Company*. That is, the task to be fulfilled by the repository application is to create a repository structure named **"Austrian Milk Models"** containing an empty catalogue called **"Dairy OLAP Patterns"**, an empty glossary **"Dairy Business Terms"**, and an empty MDM **"Dairy Precision Farming"**.

Deleting Organization Elements

Deleting organization elements describes the task for the repository application to support the OLAP pattern language (Appendix A) statements for removing organization elements. These statements are characterized by the `DELETE` keyword at the beginning; subsequently, the organization element designation and its path can be stated. Thus, the task for the repository is to check whether an element specified by the path exists and if so, to delete it and all the contained content elements.

```
1 DELETE CATALOGUE "Austrian Milk Models"/"Dairy OLAP Patterns";
```

Listing 3.2: Delete Statement for the catalogue "Dairy OLAP Patterns"

In Listing 3.2, the catalogue with the name **"Dairy OLAP Patterns"** in the repository **"Austrian Milk Models"** should be deleted. Further, the task includes to delete all patterns, i.e. their contexts, descriptions and templates, comprised in the catalogue.

Retrieving Organization Elements

The task for retrieving organization elements is to support the corresponding OLAP pattern language statements namely search and show statements. Search statements should allow the user to perform a simple string search, i.e., find all organization elements (search target) containing the search string (search term) in their name (search scope). Hence, a search statement starts with SEARCH keyword and allows the user to subsequently state a search target, an optional search space, i.e., a path where to search in, a search term and the search scope. An exemplary statement can be seen in Listing 3.3

```
1 SEARCH CATALOGUE IN "Austrian Milk Models" CONTAIN "Dairy" IN NAME;
```

Listing 3.3: Search statement for catalogues

The statement in Listing 3.3 should provide the user with all catalogues (search target) in the **"Austrian Milk Models"** repository (search space) containing the word **"Dairy"** (search term) in their names (search target).

Besides search operations for concrete element types, it might be useful to just gain an overview of the elements included in a path. This means, the user should be able to just state a path of interest in form of SHOW statements as defined in the OLAP pattern language, to gain a list of comprised elements.

```
1 SHOW  "Austrian Milk Models"/"Dairy OLAP Patterns";
```

Listing 3.4: Show statement for a catalogue path

In Listing 3.4 a statement to explore a catalogue is depicted. For this example, the task for the repository application is to find any pattern included in this path and provide a list with their names. Accordingly, if the target is an MDM, all cubes and dimensions are to be listed, whereas the business terms must be listed for glossaries.

3.1.2 Content Elements: OLAP Patterns, Business Terms, Cubes, and Dimensions

Within the repository application four types of content elements are distinguished, namely OLAP patterns, business terms, cubes and dimensions. These elements provide the actual information that should enable the repository application to automatically generate OLAP queries that satisfy the users information demands. This means, users must be able to create, retrieve, update, and

delete content elements following the definition process in subsection 2.2.1. Further, it must be possible to instantiate and execute both business terms and patterns following the steps of the usage process (subsection 2.2.2).

Defining Content Elements

Inserting content is a central task for an OLAP pattern repository application, which must enable the user to add business terms, patterns, cubes and dimensions to the corresponding organization elements. The repository application therefore must support commands allowing to state the definition of these content elements following the OLAP pattern language, i.e. CREATE commands.

The structure for all these commands consist of a starting and an end line, which are independent of the content element, and a midsection, which is content-element specific. Basically, the starting line of a CREATE statement consists of the CREATE keyword, the content element designation, and a path. The element designation in fact represents the available content element PATTERN, CUBE, DIMENSION, and for business term the business term types UNARY_DERIVED_MEASURE, BINARY_DERIVED_MEASURE, CUBE_ORDERING, UNARY_CUBE_PREDICATE, BINARY_CUBE_PREDICATE, UNARY_DIMENSION_PREDICATE, BINARY_DIMENSION_PREDICATE, DIMENSION_GROUPING, and DIMENSION_ORDERING [1]. The structure of the path consists of the repository name, the name of a MDM, glossary or, catalogue, and the name of the actual content element to be created. The end line of each statement consists of the END keyword and the element designation and indicates that the definition is over. The middle section of the statements and thus the task is element dependent and is thus described in detail for each type of content element in the following.

```
1 CREATE OR REPLACE DIMENSION "Austrian Milk Models"/"Dairy Precision
    ↪ Farming"/"Time" WITH
2   LEVEL PROPERTIES
3     "Date":"Date";
4     "Month":"Month No";
5     "Year":"Year No";
6   END LEVEL PROPERTIES;
7
8   ATTRIBUTE PROPERTIES
9     "Month Label":"Month Name";
10  END ATTRIBUTE PROPERTIES;
11
12  CONSTRAINTS
13    "Date" ROLLS_UP_TO "Month";
14    "Month" ROLLS_UP_TO "Year";
15    "Month" DESCRIBED_BY "Month Label";
16  END CONSTRAINTS;
17 END DIMENSION;
```

Listing 3.5: Create "Time" dimension statement

Firstly, for dimensions one should be able to define any number of dimension properties, i.e., levels and attributes, and their relationships. The relations to be distinguished are "roll-up" relations (a level can roll up to another level) and "described-by" relations (an attribute describes a level). Further, for any level and attribute it should be possible to associate a value set, i.e., define the

type of its values. The task for the repository application is to read the information contained in the statement and add the dimension to the MDM stated by the user.

In Listing 3.5 a statement to define an example **"Time"** dimension inside the **"Dairy Precision Farming"** MDM of the **"Austrian Milk Models"** repository is depicted. The levels and the corresponding value sets separated by a **":"** are stated in the lines 3, 4 and 5. For example, the level **"Year"** is restricted to the **"Year No"** value set. Equivalently, attributes and their ranges as in line 9 must be processed. Lastly, the "roll-up" and "described-by" relationships from the lines 13, 14 and 15 relate levels and attributes. Levels may be related to other levels to form "roll-up" hierarchies using the ROLLS_UP_TO keyword; the *Date* rolls up to the *Month* level. In addition, levels may be related to attributes describing them using the DESCRIBED_BY keyword; the *Month* level is described by the attribute *Month Label*.

Secondly, users should be enabled to create cubes containing both measure and dimension role properties. For measures it should be possible to restrict their values by associating them with value sets inside the MEASURE PROPERTIES block. For dimension roles, one should be able to associate the name of a previously defined dimension inside the DIMENSION_ROLE PROPERTIES block. That is, if no dimension with the name already exists in the MDM, the repository must not create the cube. Otherwise, the repository should persist the cube within the MDM stated by the user.

```
1  CREATE OR REPLACE CUBE "Austrian Milk Models"/"Dairy Precision
     ↪ Farming"/"Feeding" WITH
2    MEASURE PROPERTIES
3      "Feed Consumption":"Roughage In Kilogram";
4    END MEASURE PROPERTIES;
5
6    DIMENSION_ROLE PROPERTIES
7      "Farm":"Farm";
8      "Feeding Time":"Time";
9      "Cattle":"Animal";
10   END DIMENSION_ROLE PROPERTIES;
11 END CUBE;
```

Listing 3.6: Create “Feeding" cube statement

In Listing 3.6 the CREATE statement corresponding to the **"Feeding"** cube for the *Austrian Milk Company* introduced in section 2.1 is depicted. The repository must therefore create a cube named **"Feeding"** in the **"Dairy Precision Farming"** MDM comprised by the **"Austrian Milk Models"** repository (1). In the MEASURE PROPERTIES block (lines 2, 3 and 4) the **"Feed Consumption"** measure is defined, which is restricted to **"Roughage In Kilogram"** values. Inside the DIMENSION_ROLE PROPERTIES block from lines 6 to 10 three dimension roles are defined, i.e. the **"Farm"**, **"Feeding Time"** and **"Cattle"** dimension roles are defined, referencing the **"Farm"**, **"Time"** and **"Animal"** dimensions.

Thirdly, the user should be enabled to create business terms by consecutively creating a term context, description, and a template. Although context, description, and template conceptually occur only in combination, the repository should allow to define them separately to enable users to define definitions in various languages and templates for different systems. Thus, the OLAP

pattern language grammar defines statements for the three components of the pattern definition, which the repository needs to support.

To define a business term context, a corresponding CREATE statement must be supported, where parameters and constraints can be defined. Users should be able to state the type of the business term and various constraints. Thus, the statements include a CONSTRAINTS block, where domain- and property constraints are formulated to restrict the context parameter values.

```
1 CREATE UNARY_DIMENSION_PREDICATE "Austrian Milk Models"/"Dairy Business
    ↪ Terms"/"Jersey" WITH
2   CONSTRAINTS
3     <ctx>."Main Breed":"Breed Name";
4   END CONSTRAINTS;
5 END UNARY_DIMENSION_PREDICATE;
```

Listing 3.7: Business term context for "Jersey Breed"

In Listing 3.7 a create statement for the **"Jersey"** breed is provided. In contrast to other statements the repository must recognize the type of the business term from line 1, i.e., the UNARY_DIMENSION_PREDICATE designation represents a business term of the respective type. Further, an implicit context parameter (<ctx>) must be created according to the business term type; e.g. a UNARY_DIMENSION_PREDICATE indicates a <ctx> parameter restricted to the **"DIMENSION"** value set. In the CONSTRAINTS block a domain constraint in line 3 restricts the **"Main Breed"** level of <ctx> to be of the type **"Breed Name"**.

It should further be possible to add one or more business term descriptions to an existing business term context, hence, the corresponding OLAP pattern language statements need to be supported as well. As the description only includes text, the task for the repository is simple, that is, to persist the included textual attributes.

```
1 CREATE TERM DESCRIPTION FOR "Austrian Milk Models"/"Dairy Business
    ↪ Terms"/"Jersey" WITH
2   LANGUAGE = "English";
3   ALIAS = "Cattle Breed Jersey";
4   DESCRIPTION = "Restriction of result to cattle of main breed Jersey";
5 END TERM DESCRIPTION;
```

Listing 3.8: Business term description for "Jersey breed"

In Listing 3.8 the corresponding command to the English description for the Jersey business term states the term's alias name is **"Cattle Breed Jersey"** and provides a description text for the users.

The creation of business term templates must also be possible using OLAP pattern language commands. Generally, the task for templates is the same as for descriptions, as only textual attributes are included, that is, language, dialect, and expressions are strings the repository should persist. The task for the repository however does not include to check the expression in any way as the correctness of the query snippet must be ensured by the user. Note that the code of the target language must be wrapped using "*{}*", in contrast, macro calls are outside these wrapping

structures. By defining the grammar this way, the expression can include strings of arbitrary target languages without incorporating their languages' grammars.

```
1 CREATE TERM TEMPLATE FOR "Austrian Milk Models"/"Dairy Business
    ↪ Terms"/"Jersey" WITH
2   LANGUAGE = "SQL" ;
3   DIALECT = "ORACLEv11" ;
4   EXPRESSION = "*{ <ctx>.""Main Breed"" = ""Jersey"" }*";
5 END TERM TEMPLATE;
```

Listing 3.9: Business term template for "Jersey Breed"

In Listing 3.9 the representation of the SQL statement in the Oracle version 11 dialect is stated in the expression.

In addition to cubes, dimensions, and business terms, users should be able to define patterns just like terms by defining context, description, and template separately. Basically, a pattern definition is similar to the definition of a business term, except that the attributes of the definitions differ and the context also includes derived elements, additional constraints, and local cubes. Thus, the following will focus on tasks originating from these elements particularly. The creation of templates is not detailed again, as it mirrors the business term template creation, except that the data model (relational or multidimensional) and implementation variant (star or snowflake schema) are captured.

In a pattern context, the repository must additionally be able to process the `DERIVED ELEMENTS` block containing further variables and their derivation rules. It is worth noting, that for this prototype the derived variables need not be evaluated in the usage process (subsection 2.2.2), thus type checks for the rules are not part of the task as this is performed by the pattern grounding component. In the context of patterns, the repository does have to support three additional constraints, namely return constraints as well as unary and binary applicable-to constraints. Further, patterns also offer local cubes of a pattern's local multidimensional model provided by the template (subsection 2.2.1), which is represented in the same forms as constraints concerning multidimensional model elements, i.e., type, property, and domain local cubes. Hence, the task for extracting information from the local cubes stated in the `LOCAL CUBES` block is equivalent to the constraints of the same name.

An example statement can be seen in Listing 3.10, which is the **"Breed-Specific Subset-Subset Comparison"** that can be used in the context of the Austrian Milk Company's eMDM. In the `DERIVED ELEMENTS` block, one can see the formulation of derivation rules starting from line 17 to line 19. The meaning behind derivation rules in e.g. line 17 is, that the name of the dimension <compDim> is the name of the dimension referenced by the <compDimRole> dimension role of the <sourceCube> parameter, whereas line 19 defines that the name of the number value set <cubeMeasureDom> is the name of the return type of the <cubeMeasure> unary derived measure. Further, in the `CONSTRAINTS` block, e.g. in the line 42 a unary applicability constraint restricts the values of the <animalBreedSlice> parameter, to be a business term which must be applicable to the MDM element **"Animal"**.

```
1 CREATE PATTERN "Austrian Milk Models"/"Dairy OLAP Patterns"/"Breed-Specific
     ↪ Subset-Subset Comparison" WITH
2
3   PARAMETERS
4       <sourceCube>:CUBE;
5       <baseCubeSlice>:UNARY_CUBE_PREDICATE;
6       <animalBreedSlice>:UNARY_DIMENSION_PREDICATE;
7       <compDimRole>:DIMENSION_ROLE;
8       <iDimSlice>:UNARY_DIMENSION_PREDICATE;
9       <cDimSlice>:UNARY_DIMENSION_PREDICATE;
10      <joinDimRole>:DIMENSION_ROLE;
11      <groupCond>:DIMENSION_GROUPING;
12      <cubeMeasure>:UNARY_DERIVED_MEASURE;
13      <compMeasure>:BINARY_DERIVED_MEASURE;
14  END PARAMETERS;
15
16  DERIVED ELEMENTS
17      <compDim>:DIMENSION <= <sourceCube>.<compDimRole>;
18      <joinDim>:DIMENSION <= <sourceCube>.<joinDimRole>;
19        <cubeMeasureDom>:NUMBER_VALUE_SET <= <cubeMeasure>.RETURNS;
20  END DERIVED ELEMENTS;
21
22  FRAGMENTS
23    "interestCube":CUBE;
24    "interestCube" HAS MEASURE <cubeMeasure>;
25    "interestCube".<cubeMeasure>:<cubeMeasureDom>;
26    "comparisonCube":CUBE;
27    "comparisonCube" HAS MEASURE <cubeMeasure>;
28    "comparisonCube".<cubeMeasure>:<cubeMeasureDom>;
29  END FRAGMENTS;
30
31  CONSTRAINTS
32      <sourceCube> HAS DIMENSION_ROLE "Cattle";
33      <sourceCube> HAS DIMENSION_ROLE <compDimRole>;
34      <sourceCube>."Cattle":"Animal";
35      "Animal" HAS LEVEL "Main Breed";
36      "Animal"."Main Breed":"Breed Name";
37
38      <sourceCube>.<compDimRole>:<compDim>;
39      <sourceCube>.<joinDimRole>:<joinDim>;
40      <cubeMeasure> RETURNS <cubeMeasureDom>;
41
42      <animalBreedSlice> IS_APPLICABLE_TO "Animal";
43      <baseCubeSlice> IS_APPLICABLE_TO <sourceCube>;
44      <iDimSlice> IS_APPLICABLE_TO <compDim>;
45      <cDimSlice> IS_APPLICABLE_TO <compDim>;
46      <groupCond> IS_APPLICABLE_TO <joinDim>;
47      <cubeMeasure> IS_APPLICABLE_TO <sourceCube>;
48      <compMeasure> IS_APPLICABLE_TO ("interestCube","comparisonCube");
49  END CONSTRAINTS;
50 END PATTERN;
```

Listing 3.10: Pattern context for “Breed-Specific Subset-Subset Comparison" taken from [1] with authors permission

Regarding pattern descriptions, the task for the repository is to persist the corresponding string attributes to be able to provide information to the users. These attributes differ from the ones used in business term descriptions and include, language, alias name(s), problem description, solution idea, an example and related pattern name(s).

```
1 CREATE PATTERN DESCRIPTION FOR "Austrian Milk Models"/"Dairy OLAP
    ↪ Patterns"/"Breed-Specific Subset-Subset Comparison" WITH
2   LANGUAGE = "English";
3   ALIAS = "Breed-Specific Comparison";
4   PROBLEM = "Aggregated measure values for two specified groups of facts
      ↪ relating to a
5       specific breed from a single source cube should be compared in a
          ↪ meaningful way.";
6   SOLUTION = "...";
7   RELATED = "Breed-Specific Subset-Subset Side-By-Side Comparison";
8   EXAMPLE = "....";
9 END PATTERN DESCRIPTION;
```

Listing 3.11: Pattern description for "Breed-Specific Subset-Subset Comparison" taken from [1] with authors permission

The exemplified information to the **"Breed-Specific Subset-Subset Comparison"** pattern can be taken from Listing 3.11. Accordingly, it is an English description, which states the alias name, i.e. **"Breed-Specific Comparison"** and describes the analysis situations to which it can be applied. For the sake of brevity the solution as well as the example text are not detailed. However, based on the existing information, it can be deemed similar to the **"Breed-Specific Subset-Subset Side-By-Side Comparison"**.

Updating Content Elements

The OLAP pattern language grammar defines the updating of patterns or eMDM elements to be a `CREATE OR REPLACE` statement. Hence, update statements are just the same as create statements with the exception that the `CREATE` at the beginning is replaced by the `CREATE OR REPLACE` keyword. Thus, updating content elements for the repository application is to be implemented as delete and re-insert process. Accordingly, it is not the task of the repository application to support more specific update like renaming variables or reformulating constraints for business term- or pattern contexts.

Deleting Content Elements

For deleting content elements simple delete statements are predefined by the OLAP pattern language which are similar to `DELETE` statements for organization elements. Thus, structure of all delete statements follows the same schema: the `DELETE` keyword is followed by the element designation, followed by the path of the element to be deleted. The task for the repository is to identify the corresponding element and perform a cascading delete, e.g., if a pattern should be deleted, all associated descriptions and templates must be deleted as well. Delete statements in the context of this work however regard elements only, not their components; so it is not possible to formulate statements to delete e.g. a specific constraint from a pattern. Consequently, such statements are not part of the task for this work either.

```
1 DELETE PATTERN "Austrian Milk Models"/"Dairy OLAP Patterns"/"Breed-Specific
    ↪ Subset-Subset Comparison";
```

Listing 3.12: Delete statement for "Breed-Specific Subset-Subset Comparison"

In Listing 3.12 the **"Breed-Specific Subset-Subset Comparison"** pattern, as defined in Listing 3.10 should be deleted. That is, the repository application must remove the pattern from the **"Dairy OLAP Patterns"** catalogue in the **"Austrian Milk Models"** repository. Further, the description added to the **"Breed-Specific Subset-Subset Comparison"** pattern in Listing 3.11 must also be removed.

```
1 DELETE PATTERN TEMPLATE FOR "Austrian Milk Models"/"Dairy OLAP
    ↪ Patterns"/"Breed-Specific Subset-Subset Comparison" WITH
    ↪ DATA_MODEL="Relational", VARIANT="Star-Schema", LANGUAGE="SQL",
    ↪ DIALECT="Oraclev11";
2
3 DELETE PATTERN DESCRIPTION FOR "Austrian Milk Models"/"Dairy OLAP
    ↪ Patterns"/"Breed-Specific Subset-Subset Comparison" WITH
    ↪ LANGUAGE="English";
```

Listing 3.13: Delete Statement for"Breed-Specific Subset-Subset Comparison"'s templates and descriptions

The deletion of templates and descriptions of business terms require further arguments for the delete operation. In Listing 3.13 the **"Breed-Specific Subset-Subset Comparison"** pattern's templates are deleted that were defined for the relational data model, following a star schema realization considering SQL with an **"Oraclev11"** dialect. If arguments are omitted completely, then all templates are deleted, while omitting some arguments leads to the deletion of all templates matching those arguments. Similarly, descriptions require additional arguments for the delete operation, i.e., the language.

Retrieving Content Elements

To actually use the content, both eMDM elements and patterns must be retrievable. Hence, the repository must offer means enabling users to search for certain content elements that potentially satisfy their needs. The OLAP pattern language defines two types of SEARCH statements that allow for the user to search for content fulfilling certain criteria. The basic search statements aim at string searches, i.e., the statements are equivalent to the search statements for organization elements except that the search scope can be specified to section of a description (if available otherwise only the name) (section 3.1.1). Accordingly, the structure consists of the keyword SEARCH and subsequently contains a search target (pattern, business term, cube or dimension), an optional search space (the path to search in), a search term (the string to search for) and a search scope (name and sections available in descriptions), hence, possible search scopes for a pattern or term are all attributes included in the description, i.e. the language, solution, alias names etc, while for cube and dimension searches, the name is the only semantically reasonable scope.

```
1 SEARCH TERM IN "Austrian Milk Models"/"Dairy Business Terms" CONTAIN "Subset"
    ↪ IN NAME, ALIAS;
```

Listing 3.14: String search for business terms

The Listing 3.14 depicts an example on how to search for business terms. The task for the repository is to extract the case-insensitive search string, i.e., **"Subset"**, and find all business terms containing it in their names and alias names. Additionally, the business terms must be part of the **"Dairy Business Terms"** glossary, to be part of the search result.

More advanced search statements also allow to have multiple combinations of search terms and search scopes.

```
1 SEARCH TERM IN "Austrian Milk Models"/"Dairy Business Terms" CONTAIN "Subset"
    ↪ IN NAME, ALIAS CONTAIN "English" in LANGUAGE;
```

Listing 3.15: String search for business terms with multiple search terms

The Listing 3.15 is an extended version of the Listing 3.14 statements, that further restricts the search results to business terms having an English description.

Further, the repository application must support to retrieve content elements using SHOW statements. In contrast to organization elements, a show statement for a content element must provide the user with the initial definition statement. Accordingly, the repository application must be capable of creating OLAP pattern language definitions for content elements. The regarding structure consists of two parts beginning with the SHOW keyword, followed by a content element's path.

```
1 SHOW TERM IN "Austrian  Milk  Models"/"Dairy  Business  Terms"/"Jersey";
```

Listing 3.16: Show statement for a business term

For the statement in Listing 3.16 the repository therefore must provide the user with the definition of the Jersey business term (Listing 3.7).

Using Content Elements

Corresponding to the usage process (subsection 2.2.2), the last part of the task for an OLAP pattern repository is to support users in the practical application of patterns. Therefore, in a first step the repository must support the instantiation of any previously defined patterns. Following the OLAP pattern language an INSTANTIATE PATTERN statement consists of a path that defines the pattern to instantiate, a path of for the new instantiated pattern and one or more binding expressions. Therefore, the BINDINGS block allows to assign values to any parameter defined in the pattern context by using the "=" operator. Thus, the task for the repository is to create and persist a new pattern, which is a copy of the initial pattern, and replace the instantiated variables with the corresponding values.

```
1 INSTANTIATE PATTERN "Austrian Milk Models"/"Dairy OLAP
    ↪ Patterns"/"Breed-Specific Subset-Subset Comparison" AS
2      "Austrian Milk Models"/"Dairy OLAP Patterns"/"Austrian Milk Custom
         ↪ Breed-Specific Subset-Subset Comparison" WITH
3   BINDINGS
4      <sourceCube> = "Feeding",
5      <animalBreedSlice> = "Jersey",
6      <baseCubeSlice> = "Low Daily Feed Consumption",
7      <compDimRole> = "Cattle",
8      <iDimSlice> = "Young Cattle",
9      <cDimSlice> = "Old Cattle",
10     <joinDimRole> = "Farm",
11     <groupCond> = "Per Farm",
12     <cubeMeasure> = "Average Feed Consumption",
13     <compMeasure> = "Average Feed Consumption Ratio"
14  END BINDINGS;
```

Listing 3.17: Instantiation of "Breed-Specific Subset-Subset Comparison" Pattern taken from [1] with authors permission

In Listing 3.17 the pattern defined in Listing 3.10 is instantiated. The *Breed-Specific Subset-Subset Comparison* is the one to be instantiated, whereas the *Austrian Milk Custom Breed-Specific Subset-Subset Comparison* is the name for the new, more specific pattern. The newly returned pattern is only a more specific version of the existing one, except that the pattern parameters removed and replaced by the name bound in the derivation rules, constraints and local cubes.

Fully instantiated patterns, where all parameters are bound to parameters, shall be able to be executed to get an executable OLAP query. The task for the repository application, hence is to process EXECUTE commands. The execution command, in its simplest form, only consists of one line, if the user intends to execute all available templates of a pattern. Specific templates are selected by specifying the sections of a pattern template, that is, DATA_MODEL, VARIANT, LANGUAGE, and DIALECT. The basic structure of the EXECUTE statements for executing all pattern templates includes the path to a pattern, followed by the paths to the MDM and the glossary. For executing specific templates, the statement is extended by the template sections to be considered.

```
1 EXECUTE PATTERN "Austrian Milk Models"/"Dairy OLAP Patterns"/"Austrian Milk
    ↪ Custom Breed-Specific Subset-Subset Comparison" FOR "Austrian Milk
    ↪ Models"/"Dairy Precision Farming" USING "Austrian Milk Models"/"Dairy
    ↪ Business Terms";
2
3 EXECUTE PATTERN "Austrian Milk Models"/"Dairy OLAP Patterns"/"Austrian Milk
    ↪ Custom Breed-Specific Subset-Subset Comparison" FOR "Austrian Milk
    ↪ Models"/"Dairy Precision Farming" USING "Austrian Milk Models"/"Dairy
    ↪ Business Terms" WITH TEMPLATE DATA_MODEL="Relational", VARIANT="Star
    ↪ Schema", LANGUAGE="SQL", DIALECT="ORACLEv11";
```

Listing 3.18: Execute "Austrian Milk Custom Breed-Specific Subset-Subset Comparison" pattern

The task for the repository, considering Listing 3.18 line 1, is to execute all templates of the **"Austrian Milk Custom Subset-Subset Comparison"** pattern in the context of the **"Dairy**

Precision Farming" MDM enriched by the **"Dairy Business Terms"**. In contrast, in line 3 the task is only to execute the specific template, i.e., the template for a relational data model in the star schema in the **"ORACLEv11"** dialect of the SQL language. Either way, the goal is to get an executable query, generated from the template(s), adapted to the eMDM.

3.2 Requirements

The requirements in this section define how the OLAP pattern repository application must be implemented in order to be useful in practice. Therefore, it is split into two parts following the idea of Sommerville [20, p. 94], describing the requirements resulting from the section 3.1 as a combination of user and system requirements. The user requirements describe, in the form of user scenarios following [20, p. 105], how the repository application is intended to be used in practice. The aim is to formulate user requirements in a way that they are understandable for someone without technical knowledge [20, p. 94]. In a second step the corresponding system requirements are derived. Following Sommerville [20, p. 94] system requirements are expanded user requirements, describing how the system should provide the functionality and thus can be considered the starting point for the design. The system requirements for this work deviate from this definition by avoiding redundancy and only focus on how functionality should be provided. It is noteworthy, that also the general requirements for repositories as depicted in section 2.3 are considered important for this work.

3.2.1 User requirements

User requirements formulate, in a non-technical way, how the OLAP pattern repository is intended to be used in practice. To make these requirements comprehensible, they are be formulated as scenarios following the schema of Sommerville [20, p. 105]. Accordingly, scenarios *"are descriptions of example interaction sessions"* [20, p. 105] and thus for this work are built around the language statement depicted in the task (section 3.1). Each scenario should include an initial assumption, a regular flow, exceptions and a goal state [20, p. 106]. Based on the problem statement groups of user requirements can be formed, i.e., requirements regarding create, retrieve, update, delete, and execute operations. The user requirements can further be classified according the OLAP pattern approach in the ones regarding the definition, building a definition-centric view, and the ones regarding the usage process, building the usage-centric view. Each group of requirement can additionally be associated with one or both of the user groups. In Figure 3.1 an overview of requirement groups is provided, where each is classified and associated with a user role.

Table 3.1 provides an overview of the specific requirements for this work, where a classification concerning the element type is made. Hence, there are requirements for organization and for content elements. The focus of the user requirements clearly lies on create-, instantiate-, and execute-operations, which are separately defined for each type of organization and content element. The reason for this is that they differ significantly depending on which element is involved, whereas update-, delete-, search-, and show-requirements are basically identical for every content or organization element, respectively.

In the following, all scenarios regarding patterns, eMDM elements, and organization elements are described (see Table 3.1). At first, requirements for organization elements are defined in

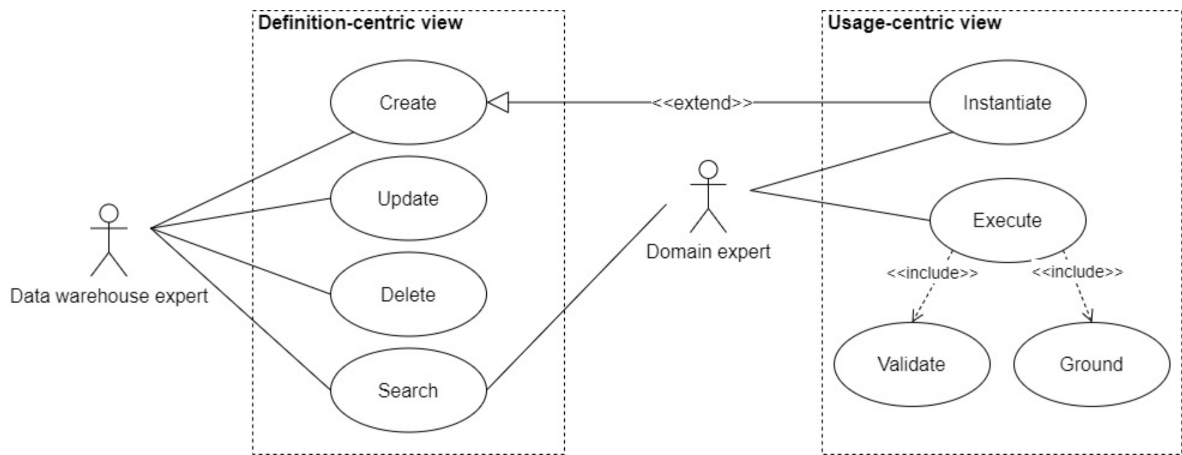


Figure 3.1: Use cases defining user requirements

section 3.2.1, followed by four sections on content element requirements. That is, in section two requirements for patterns are stated, whereas section three covers business-term-specific and section four cube- and dimension-specific requirements. Lastly, section five defines general content specific requirements, i.e., concerning update-, delete-, search-, and show-operations.

User Requirements for Organization Elements

This subsection introduces requirements originating from organization elements and thus build upon the regarding language statement described in the task (section 3.1).

Create Repository Requirement (CRR): Creating a repository is a scenario that refers to setting up a new organization structure within the repository application. This requirement applies to situations where a DWH expert wants to define a new space to maintain content elements for, e.g., an independent unit in an organization like a subsidiary.

- **Initial state:** A new organization structure needs to be set up, however, there are no pre-conditions concerning the repository or its content.
- **Regular flow:** A unique name for the repository must be defined. Therefore, the user may want to follow the SOER to search for existing repositories having the same or similar names. In case that there is no existing repository with the exact same name, the corresponding `CREATE` statement (see Listing 3.1) is formulated and executed.
- **Exception cases:** (i) The command to execute includes syntactical errors; a detailed error message should help the user to correct the statement. (ii) The given repository name already exists; an error message must prompt the user to find a new name.
- **Goal state:** The repository element is added and the success message of the procedure is provided to the user.

User Requirements for Organization Elements
Create Repository Requirement (CRR) Create Catalogue Requirement (CCR) Create Glossary Requirement (CGR) Create MDM Requirement (CMR)
Delete Organization Element Requirement (DOER)
Search Organization Element Requirement (SOER)
Show Organization Element Requirement (SOLR)
User Requirements for Content Elements: Patterns
Create Pattern Requirement (CPR) Create Pattern Context Requirement (CPCR) Create Pattern Template Requirement (CPTR) Create Pattern Description Requirement (CPDR)
Instantiate Pattern Requirement (IPR)
Execute Pattern Requirement (EPR)
User Requirements for Content Elements: Business Terms
Create Business Term Requirement (CBR) Create Business Term Context Requirement (CBCR) Create Business Term Template Requirement (CBTR) Create Business Term Description Requirement (CBDR)
User Requirements for Content Elements: Cubes and Dimensions
Create Cube Requirement (CCR) Create Dimension Requirement (CDR)
User Requirements for Content Elements
Update Content Element Requirement (UCER)
Delete Content Element Requirement (DCER)
Search Content Element Requirement (SCER)
Show Content Element Requirement (SCDR)
System Requirements
Language Statement Processing Requirement (LSPR)
Extensibility Requirement (EXTR)
Object Representation Requirement (ORR)
Interchangeability Requirement (ICR)

Table 3.1: User- and System-Requirements

Create Catalogue Requirement (CCR): The CCR is a scenario of the definition-centric view that corresponds to the need for adding a new catalogue to an existing repository. That is, the DHW expert wants to create a new set of patterns, e.g., for a department of an organization.

- **Initial state:** The repository application contains at least one repository element, which must be extended by a new catalogue.
- **Regular flow:** A unique name for the catalogue within the repository must be defined. To find a unique name the user may want to follow the SOER once he/she already has a name in mind, to find out if there is an existing catalogue with that name. In case that there is no existing catalogue with the intended name, the corresponding `CREATE` statement (see Listing 3.1) is formulated and executed.
- **Exception cases:** (i) The command to be execute includes syntactical errors; a detailed error message should help the user to correct the statement. (ii) The given catalogue name already exists; an error message must prompt the user to find a new name. (iii) The repository does not exist; the user must be informed to specify an existing repository.
- **Goal state:** The catalogue element is added to the repository and the user is confirmed of success of the procedure.

Create Glossary Requirement (CGR): Creating a glossary applies to situations where a new glossary of business terms should be set up in the repository application. Hence a new glossary must be added to an existing repository organization element.

- **Initial state:** The repository application contains at least one repository element, which must be extended by a new glossary.
- **Regular flow:** A unique name for the glossary within the repository must be defined. To find a unique name the user may want to follow the SOER once he/she already has a name in mind, to find out if there is an existing glossary with that name. In case that there is no existing glossary with the intended name, the corresponding `CREATE` statement (see Listing 3.1) is formulated and executed.
- **Exception cases:** (i) The command to be execute includes syntactical errors; a detailed error message should help the user to correct the statement. (ii) The given glossary name already exists; an error message must prompt the user to find a new name. (iii) The repository does not exist; the user must be informed to specify an existing repository.
- **Goal state:** The glossary element is added to the repository and the success of the procedure is confirmed to the user.

Create MDM Requirement (CMR): Creating a MDM refers to the need of depicting a new organizational or reference MDM within the repository application. Hence a new MDM must be added to an existing repository organization element.

- **Initial state:** The repository application contains at least one repository element, which must be extended by a new MDM.

- **Regular flow:** A unique name for the MDM within the repository must be defined. To find a unique name the user may want to follow the SOER once he/she already has a name in mind, to find out if there is an existing MDM with that name. In case that there is no existing MDM with the intended name, the corresponding `CREATE` statement (see Listing 3.1) is formulated and executed.
- **Exception cases:** (i) The command to be executed includes syntactical errors; a detailed error message should help the user to correct the statement. (ii) The given MDM name already exists; an error message must prompt the user to find a new name. (iii) The repository does not exist; the user must be informed to specify an existing repository.
- **Goal state:** The MDM element is added to the repository and the success of the procedure is confirmed to the user.

Delete Organization Element Requirement (DOER): Deleting organization elements is a requirement of data warehouse experts wanting to clean up the repository application. It might be necessary to delete organization elements if the contained content is not needed any longer. For all organization elements all contained content elements must be removed as well, i.e., a cascading delete is to be performed. The reason, why there is only a generic delete requirement for all organization elements is, that the scenarios and language statements (see section 3.1) are identical for every element. However, deleting elements is to be treated with caution as for this version no backups are intended.

- **Initial state:** An organization element which should be removed is persisted in the repository.
- **Regular flow:** The element to be deleted needs to be identified (e.g., by following SCER or SCDR), that is, the name and location must be known. A `DELETE` statement (see Listing 3.2) is formulated and executed.
- **Exception cases:** (i) The command to be executed includes syntactical errors; a detailed error message should help the user to correct the statement. (ii) The user may provides either an element name or a path that does not actually exist; an error message should state that the path is incorrect.
- **Goal state:** The organization element and its content have been removed from the repository, leaving it in a consistent state. The user receives a success message.

Search Organization Element Requirement (SOER): Searching for organization elements containing a specific string is a scenario, that can be connected to both user roles and is classified as part of the definition-centric view. Both user groups may need to retrieve an organization element in course of another scenario, or just want to inspect the current organization structure. The goal of a search statement is to construct and return a machine interpretable representation of the organization element and the included content, allowing for a reasonable visualization.

- **Initial state:** One has a certain type of organization element in mind, that should be found, e.g., a catalogue. Additionally, a search string the user knows or assumes to occur in the organization element's name is known.

- **Regular flow:** Initially, a path may be chosen, to narrowed the search further, e.g. for a catalogue one can state the repository it must be contained in. Therefore, the user may want to discover possible paths following the SOLR. Finally the corresponding `SEARCH` statement (see Listing 3.3) is formulated and executed.
- **Exception cases:** (i) The command to be execute includes syntactical errors; a detailed error message should help the user to correct the statement.
- **Goal state:** The set of organization elements meeting the search criteria is returned to the user.

Show Organization Element Requirement (SOLR): Showing an organization element is a scenario that allows the user to view a path of interest to explore which elements are included. That is, for a repository, one can, for example, find the names of the comprised catalogues, MDMs and, glossaries.

- **Initial state:** The repository application includes at least one organization structure that can be discovered.
- **Regular flow:** The path to be shown is stated in a `SHOW` (see Listing 3.4) statement which is further executed. Thus, this scenario may include an iteration, that is, one can start from the root and discover the structure all the way to, e.g., a pattern (SCDR).
- **Exception cases:** (i) The command to be execute includes syntactical errors; a detailed error message should help the user to correct the statement. (ii) The path provided in the statement does not exist; the user must be informed by an error message.
- **Goal state:** A textual representation of the elements within the path is offered.

User Requirements for Content Elements: Patterns

This subsection describes the user scenarios that concern patterns, their contexts, templates, and descriptions. That is, this section will focus on peculiarities of patterns regarding creation, instantiation, and execution.

Create Pattern Requirement (CPR): The create pattern scenario is a part of the definition-centric view and describes the need for automatizing creation of new patterns. The scenario includes the identification, design, and creation of a pattern corresponding to a recurring information demand. Hence, it is an aggregation of the CPCR, CPTR and CPDR scenario that are subsequently described.

- **Initial state:** The data warehouse expert has identified a new recurring information demand, that is not covered by any existing pattern. Thus, the goal is to formulate a best practice solution that can be reused for future occurrences of that type of information demand.
- **Regular flow:** The DWH expert starts by defining the pattern context, i.e., following the CPCR scenario. Once the context is persisted in the repository, it can be continued to formulate a corresponding template as described in the CPTR scenario. Subsequently, a description as in CPDR must be provided, to complete the pattern definition.

- **Exception cases:** (i) Every error that occurs during one of the three sub-scenarios.
- **Goal state:** Pattern context, descriptions and templates are persisted in the repository keeping it in a consistent state. The UI confirms the success to the user.

Create Pattern Context Requirement (CPCR): This scenario is the first sub-scenario of the CPR to be executed. It depicts the process that is necessary for a DWH expert to add a pattern's context to the repository.

- **Initial state:** The initial state as for CPR.
- **Regular flow:** Depending on the flexibility to be provided (domain-independent, domain-specific, or organization-specific) pattern parameters can be declared with respect to an associated eMDM. Derived elements can be defined by formulating derivation rules upon these parameters, constants, and other derived elements. Upon the parameters and derived elements, constraints can be defined that restrict the applicability to the intended situations. Lastly, local cubes can be defined specifying fragments of the local multidimensional model that is provided by the pattern's templates. Additionally, a unique pattern name must be chosen; therefore the users may want to inspect the catalogue he/she wants to add the pattern to. That is, one can follow the steps described in the SCDR, to see which names are already taken. The `CREATE PATTERN` command (see Listing 3.10) is formulated that depicts the pattern context. Finally, the command is sent to the repository which must persist the provided information.
- **Exception cases:** (i) The user may provides either a repository or a catalogue name that does not actually exist; an error message should state that the path is incorrect. (ii) The command to be execute includes syntax errors; a detailed error message should help the user to correct the statement. (iii) The pattern is semantically incorrect, e.g., parameters in constraints are used but never defined; the user must be informed in an error message.
- **Goal state:** The new pattern and its context are persisted in the repository. The user does receive a confirmation.

Create Pattern Template Requirement (CPTR): Creating a template is the second step to define a pattern, describing the need of domain experts to add a generic query structure to a pattern that is interspersed with placeholders for variables and macro calls. It also applies to fully defined patterns, which need templates in additional query languages, dialect, data model or variant.

- **Initial state:** The repository contains at least one pattern context that needs a template to be fully instantiated or that needs to be extended by further templates.
- **Regular flow:** In a first step, the data model, variant, query language, and dialect for the template needs to be specified. The query structure representing the pattern in the specific query language is constructed, including placeholders for variable values and necessary macro calls. The pattern context to be extended needs to be identified (e.g. by following SCER to find specific patterns or SOLR to find all patterns in a certain catalogue), that is, the

pattern's name and location must be known. In the next step a `CREATE TEMPLATE` statement (see Listing 3.10) needs to be executed adding the template to the specified pattern context.

- **Exception cases:** (i) The user may provides a path that does not actually exist; an error message should state that the path is incorrect. (ii) The command to be execute includes syntax errors; a detailed error message should help the user to correct the statement. (iii) Placeholders or macros that not actually exist may be added to the expression, however, it is not required to identify this kind of error.
- **Goal state:** The template is persisted in the repository and was added to the respective pattern. The user does receive a confirmation.

Create Pattern Description Requirement (CPDR): This requirement refers to the third step in the pattern definition and describes the scenario of adding a description to a pattern. Note that the process is the same, if a description. in another language should be added to a fully defined pattern.

- **Initial state:** The repository contains at least one pattern that needs to be extended by descriptions. That is, a pattern context and template already exist in the repository, but need to be described.
- **Regular flow:** At first, the language needs to be specified, i.e., a pattern can have numerous different descriptions in different languages. Further, meaningful textual descriptions need to be found for the description attributes. The pattern context to be extended needs to be identified (e.g. by following SCER to find specific patterns or SOLR to find all patterns in a certain catalogue), that is, the pattern's name and location must be known. In the next step a `CREATE DESCRIPTION` statement (see Listing 3.11) needs to be executed adding the template to the specified pattern context.
- **Exception cases:** (i) The user may provides a path that does not actually exist; an error message should state that the path is incorrect. (ii) The command to be execute includes errors; a detailed error message should help the user to correct the statement.
- **Goal state:** The description is persisted in the repository and is added to the respective pattern. The user does receive a confirmation.

Instantiate Pattern Requirement (IPR): Instantiating a pattern is a more specific form of a creation (CPR), i.e., it also adds a pattern to a repository. However, the instantiation follows the process introduced in subsection 2.2.2, that is, the new pattern is a copy of an existing one with some or all variables replaced by values. Thus, it allows to generate a specific pattern from a generic one.

- **Initial State:** The repository contains a nonempty set of generic patterns, i.e., all or at least some variables are unbound. Further, there is a need to adapt one of these patterns to a more specific situation.

- **Regular flow:** The domain expert searches for the name and location of the pattern to be instantiated (e.g. by following the SCER) and explores the context to find the variables that need to be instantiated (e.g. following the SCDR). The values for the variables must be determined in such a way, that they depict the analysis situation, while considering the assocaited eMDM. The `INSTANTIATE PATTERN` statement (see Listing 3.17) is formulated and executed.
- **Exceptions:** (i) The user may provides a path that does not actually exist; an error message should state that the path is incorrect. (ii) The command to be execute includes syntactical errors; a detailed error message should help the user to correct the statement. (iii) One or more of the parameters in the statement are not defined for the chosen pattern; an error message should inform the user.
- **Goal State:** A new pattern is created in the repository which includes all derived elements with corresponding derivation rules, local cubes, and constraints as well as templates from the initial pattern and additionally associates some/all variables with the bound values. The user receives a confirmation message.

Execute Pattern Requirement (EPR): An execution scenario refers to the last two steps (grounding with validity checking and the actual execution) in the pattern usage process, i.e., it includes all necessary steps to generate an executable query from a fully instantiated pattern. The users have the possibility to state the context for both the grounding and the execution by stating an MDM and a glossary path following the usage process in subsection 2.2.2. Further users can choose to execute all templates associated with a pattern, or just the ones meeting specific criteria as depicted in the task section 3.1;

- **Initial State:** The repository contains a nonempty set of fully instantiated patterns, i.e., parameter-free patterns, as well as an associated eMDM.
- **Regular flow:** The domain expert searches for the name and location of the pattern to be executed, e.g., by following the SCER scenario. Optionally, the data model, variant, language, and dialect can be stated. Further, the location of an MDM and a glossary must be explored, the pattern should be executed for (e.g. by using the SCER again). An `EXECUTE PATTERN` command (see Listing 3.18) is formulated and executed for the MDM using the glossary.
- **Exceptions:** (i) The user may provides either a pattern template name or a path that does not actually exist; an error message should ask the user to correct the input. (ii) The command to be execute includes syntactical errors; a detailed error message should help the user to correct the statement. (iii) The pattern is not applicable to the eMDM; an error message should inform the user.
- **Goal State:** The user receives the desired query, satisfying the information demand.

User Requirements for Content Elements: Business Terms

This subsection describes the user scenarios concerning business terms, their contexts, templates, and descriptions. Even though these requirements are very similar to the ones for patterns, there are

some important differences, that demand autonomous requirements. However, the requirements in this section regard the same operations as for patterns, i.e., updating, deleting, searching, and showing business terms.

Create Business Term Requirement (CBR): This scenario is a part of the definition-centric view and describes the need for automatizing creation of new business terms. The scenario includes the identification, design, and creation of a business term that represents a regularly used term in a specific domain. Hence, it is an aggregation of the CBCR, CBTR, and CBDR scenarios that are subsequently described.

- **Initial state:** The data warehouse expert has identified a term used in a specific domain which is not depicted in the glossary. Thus, the goal is to formulate a business term that can be used to provide a corresponding query snippet for future query composition.
- **Regular flow:** The DWH expert starts by defining the business term context, i.e., following the CBCR scenario. Once the context is persisted in the repository, it can be continued to formulate a corresponding template as described in the CBTR scenario. Subsequently, a description as in CBDR must be provided, to complete the pattern definition
- **Exception cases:** (i) Every error that occurs during one of the three sub-scenarios.
- **Goal state:** Business term context, descriptions, and templates are persisted in the repository keeping it in a consistent state. The UI confirms the success to the user.

Create Business Term Context Requirement (CBCR): This scenario is the first sub-scenario of the CBR to be executed. Hence, the process that is necessary for a DWH expert to add a business term's context to the repository is depicted. In contrast to the CPCR, the CBCR does not include steps to add derived elements or local cubes, as both are not include in the conceptual definition of business terms.

- **Initial state:** The initial state as for CBR.
- **Regular flow:** Firstly, the user must determine which type of business term is needed; the business term type implicitly determines the context parameters and their types, that allow for an application of the business term to specific entities. Constraints can be defined that restrict the applicability to certain MDM entities. Additionally, a unique name must be chosen; therefore the users may want to inspect the glossary he/she wants to add the business term to. That is, one can follow the steps described in the SCDR, to see which names are already taken. The `CREATE` command (see Listing 3.7) is formulated that depicts the term context. Finally, the command is sent to the repository application which must persist the provided information.
- **Exception cases:** (i) The user may provides a path that does not actually exist; an error message should state that the path is incorrect. (ii) The command to be execute includes syntax errors; a detailed error message should help the user to correct the statement. (iii) The business term is semantically incorrect, e.g., parameters in constraints are used but never defined; the user must be informed in an error message.

- **Goal state:** The new business term and its context are persisted in the repository. The user does receive a confirmation.

Create Business Term Template Requirement (CBTR): Creating a template is the second step in defining a business term, describing the need of domain experts to add a generic query snippet to a business term that includes placeholders for variables. It also applies to fully defined business terms, which need templates in additional query languages.

- **Initial state:** The repository contains at least one business term context that needs a template to be fully instantiated or that needs to be extended by further templates.
- **Regular flow:** In a first step, the query language and dialect for the template needs to be specified, e.g., following the DWH system used in an organization. The query snippet representing the business term in the specific query language, i.e. the expression, is constructed, including placeholders for variable values. The business term context to be extended needs to be identified (e.g. by following SCER to find specific business terms or SOLR to find all business terms in a certain glossary), that is, the name and location must be known. In the next step a `CREATE TEMPLATE` statement (see Listing 3.9) needs to be execute adding the template to the specified business term context.
- **Exception cases:** (i) The user may provides either a path that does not actually exist; an error message should state that the path is incorrect. (ii) The command to be execute includes syntax errors; a detailed error message should help the user to correct the statement. (iii) Placeholders or macros that not actually exist may be added to the expression, however, it is not required to identify this kind of error.
- **Goal state:** The template is persisted in the repository and was added to the respective business term. The user does receive a confirmation.

Create Business Term Description Requirement (CBDR): This requirement refers to the third step in the definition, i.e. the CBR and describes the scenario of adding a description to a business term. Note that the process is the same, if a description in another language should be added to a fully defined business term.

- **Initial state:** The repository contains at least one business term that needs to be extended by further descriptions. That is, a business term context and a template already exist in the repository, but need to be described.
- **Regular flow:** At first, the language needs to be specified, i.e., a business term can have numerous descriptions in different languages. Further, meaningful textual descriptions need to be found for the description attributes. The business term context to be extended needs to be identified (e.g., by following SCER to find specific business terms or SOLR to find all business terms in a certain catalogue), that is, the name and location must be known. In the next step a `CREATE DESCRIPTION` statement (see Listing 3.8) needs to be execute adding the template to the specified business term context.

- **Exception cases:** (i) The user may provides a path that does not actually exist; an error message should state that the path is incorrect. (ii) The command to be executed includes errors; a detailed error message should help the user to correct the statement.
- **Goal state:** The description is persisted in the repository and is added to the respective business term. The user does receive a confirmation.

User Requirements for Content Elements: Cubes and Dimensions

This subsection focuses on the requirements that originate from the creation of cubes and dimensions.

Create Cube Requirement (CCR): Creating a cube is a scenario that is connected to the role of the DWH expert who wants to depict a cube, for an organization's MDM or a reference MDM in the repository application.

- **Initial state:** The repository application contains at least one MDM that must be extended by a new cube.
- **Regular flow:** Firstly, the user must determine of which measures and dimension roles the cube must consist. Step two is to define the value sets for the measures and the dimensions for the dimension roles. Additionally, a unique name (within the MDM) must be chosen; therefore the users may want to inspect the MDM, to see which names are already taken. That is, one can follow the steps described in the SCDR. Subsequently, the `CREATE` command (see Listing 3.6) is formulated that depicts the cube. Finally, the command is sent to the repository application which must persist the provided information.
- **Exception cases:** (i) The user may provides a MDM name that does not actually exist; an error message should state that the path is incorrect. (ii) The command to be executed includes syntax errors; a detailed error message should help the user to correct the statement. (iii) The given cube name is already taken; a message must ask the user to provide a unique name.
- **Goal state:** The new cube is persisted in the MDM. The user does receive a confirmation.

Create Dimension Requirement (CDR): Creating a cube is a scenario that is connected to the role of the DWH expert who wants to depict a dimension, for an organization's MDM or a reference MDM in the repository application.

- **Initial state:** The repository application contains at least one MDM that must be extended by a new dimension.
- **Regular flow:** Firstly, the user must determine of which levels and attributes the dimension must consist. Step two is to define the relationship between these elements, i.e., define "roll-up" relations between different levels and "described-by" relations between levels and attributes. Additionally, a unique name (within the MDM) must be chosen; therefore the users may want to inspect the MDM, to see which names are already taken. That is, one can

follow the steps described in the SCDR. Subsequently, the `CREATE` command (see Listing 3.5) is formulated that depicts the dimension. Finally, the command is sent to the repository which must persist the provided information.

- **Exception cases:** (i) The user may provides a MDM name that does not actually exist; an error message should state that the path is incorrect. (ii) The command to be executed includes syntax errors; a detailed error message should help the user to correct the statement. (iii) The given dimension name is already taken; a message must ask the user to provide a unique name.
- **Goal state:** The new dimension is persisted in the MDM. The user does receive a confirmation.

User Requirements for Content Elements

This subsection defines requirements for scenarios which are almost identical for all kinds of content elements, i.e, business terms, patterns, cubes, and dimensions. Thus, the following covers requirements originating from update-, delete-, search-, and show-operations for these four types of content elements. Furthermore, some of these requirements apply to templates and descriptions as well.

Update Content Element Requirement (UCER): Updating content elements refers to the need of adapting to new situations by changing or extending the currently persisted element, like adding new parameters for patterns or business terms.

- **Initial state:** The repository application includes a nonempty set of content elements, where one of the elements must be updated.
- **Regular flow:** The data warehouse expert identifies necessary changes and provides an alternative structure for the respective content element. Hence, for updating a content element, the same procedure as for creating it must be applied. The only difference in any case is, that instead of a `CREATE`, a `CREATE OR REPLACE` keyword is used at the beginning of the statement.
- **Exception cases:** (i) The language statement is syntactically incorrect or is incomplete; the command needs to be corrected and re-executed. (ii) Changes should be applied to a not existing element; in this case the statement is interpreted as a regular `CREATE` command. (iii) Potentially, the update can be applied to a wrong element; this work is not intended to correct such errors, that is, there is no backup. (iv) The last potential error is a not existing path which should be identified and result in an error message.
- **Goal state:** The changes have been persisted within the repository and are confirmed to the user.

Delete Content Element Requirement (DCER): Deleting content elements is a requirement of data warehouse experts wanting to clean the repository application. It might be necessary to delete content elements if there are other, similar elements or it is not needed any longer.

The reason, why there is only a generic delete requirement is, that the scenarios and language statements (see section 3.1) are identical for all content element (expect for the deletion of descriptions and templates that require additional arguments). However, deleting elements is to be treated with caution as for this version no backups are intended.

- **Initial state:** Content elements that are unnecessary, redundant or faulty are persisted in a repository structure and need to be removed. For each type of content element, all associated elements (templates and descriptions) must be removed as well, i.e., a cascading delete is to be performed.
- **Regular flow:** The element to be deleted needs to be identified (e.g. by following SCER), that is, the name and location must be known. A DELETE statement (see Listing 3.12) is formulated and executed (see also arguments for deleting descriptions and templates Listing 3.13).
- **Exception cases:** (i) The user may provides a path that does not actually exist; an error message should state that the path is incorrect. (ii) The command to be execute includes syntactical errors; a detailed error message should help the user to correct the statement.
- **Goal state:** The elements have been removed from the repository structure, leaving it in a consistent state. The process is confirmed to the user.

Search Content Element Requirement (SCER): The SCER corresponds to the need of DWH and domain experts to search for patterns, business terms, cubes, and dimension containing a certain string in, e.g., the name. Searching for a specific or a group of content elements is often part of the regular flow of other scenarios, e.g. to decide which patterns/business terms to execute. The goal of a search statement is to construct and return a machine interpretable representation of the content element.

- **Initial state:** One has a certain type of content element in mind, that should be found. Additionally, one ore more search string the user know or assumes to occur in one of the elements attributes must be determined.
- **Regular flow:** Initially, a path may be chosen, to narrowed the search further, otherwise all organization structures within the repository application are searched. Therefore, the user may want to discover possible paths following the SOLR. Finally the corresponding SEARCH statement (see Listing 3.14) is formulated and executed.
- **Exception cases:** (i) The command to be execute includes syntactical errors; a detailed error message should help the user to correct the statement.
- **Goal state:** The set of content elements meeting the search criteria is returned to the user.

Show Content Element Requirement (SCDR): Showing a content element is a scenario that allows the user to view its structure. That is, the user is provided with the initial create statement corresponding to the content element.

- **Initial state:** The repository application includes at least one content element that can be discovered.
- **Regular flow:** Initially, one needs to identify the content element to be shown, by either following the SOER or by a show scenario for a organization element (SOLR). The path to be shown is stated in a `SHOW` statement (equivalent to Listing 3.4) which is further executed.
- **Exception cases:** (i) The command to be execute includes syntactical errors; a detailed error message should help the user to correct the statement. (ii) The path provided in the statement does not exist; the user must be informed by an error message;
- **Goal state:** A `CREATE` statement for the elements within the path is offered.

3.2.2 System Requirements

Examining the user requirements one can derive system requirements for the repository application, in order to fulfill expectations. System requirements for this work are a mean to transfer the functional user requirements into high level implementation needs. The following system requirements can be discovered:

Language Statement Processing Requirement (LSPR): Both the task in section 3.1 and the user requirements in subsection 3.2.1 depict how statements following the OLAP pattern language (Appendix A) are intended to be used as a mean to communicate with the repository application. To extract the information for further processing steps, the repository application must implement a language processing functionality following section 2.4. Hence, the repository application must include a unit that provides the corresponding features, i.e., take the statement as an input and generate a reasonable output (e.g. a semantically correct AST). That is, the repository application must implement a lexer, parser and semantic analyser capable of processing OLAP pattern language statements.

Extensibility Requirement (EXTR): The repository application to be developed should further be extensible to support extensions of the OLAP pattern approach as described by Kovacic et al. in their conclusion [1]. For example, collection and optional variables should be considered in the design.

Object Representation Requirement (ORR): The data concerning the OLAP pattern approach must centrally be stored in the repository application in any kind of database structure. However, a repository application needs an internal, more efficient way to be able to process the content (see IPR, EPR). This means, that in order to perform execution of a pattern in the context of a glossary and an MDM, it is deemed more efficient to iterate an internal object representation of the elements rather than having a large amount of database accesses. Further, in context of search requirements (SOER and SCER) the aim is to create a machine readable representation of the resulting elements, i.e., an object. So, the repository application must implement a mean to build an object structure depicting the actually persisted elements for processing and must also be able to persist newly generated objects in the database.

Interchangeability Requirement (ICR): The OLAP pattern repository application is a prototype and hence should be able to be flexible to try different technologies in further research. For example, it should be able to test different storage methods on their applicability for the task. Thus, the repository application must be a composition of different highly individual components, that do not effect each other. Accordingly, if one component is replaced through another component, doing the same work in a different way, the overall functionality should not be affected.

Chapter 4

Design

This section introduces a design for an OLAP pattern repository that solves the issues discovered in the problem statement (section 1.2), while considering general repository requirements from section 2.3 and specific requirements originating from the OLAP pattern approach (chapter 3). The proposed design is documented using the Unified Modeling Language (UML)[1], a very commonly used notation, which allows to depict the application architecture in an implementation independent way. In section 4.1 an overview of the architecture is provided based on two main components, i.e., the editor and the repository application. Section 4.2 introduces a common data structure shared by both, editor and repository application. Further, section 4.3 introduces the repository application, while section 4.4 goes into detail on the editor application.

4.1 Architecture

The architecture for an OLAP pattern repository is widely determined by the problem statement in section 1.2 and therefore consists of two top-level components, i.e., an editor application and a repository application. The repository application is designed to be the central point for storing and accessing the content elements within an organisation structure, whereas the client application allows users to access the repository via OLAP pattern language statements. Since in modern organisations possible users may be locally distributed, it can be assumed, that the communication between repository application and editor application regularly takes place over the network. Hence, the architecture follows the client-server architecture as introduced by Sommerville [20, p. 488], i.e., a client running on a local computer accesses a server on a remote computer. In the OLAP pattern repository architecture the client is named editor application and the server repository application respectively. The editor sends OLAP pattern language commands to the repository application over local network or the internet to receive the server's response as depicted in Figure 4.1. A response may either be a success/error message or an object following the data structure (section 4.2).

The editor therefore provides a textual input field to formulate a command – that follows the predefined grammar – a send button and a response field. It thus contains no logic to process the statement in any kind, that is, the editor application is a thin client [20, p. 492] It follows that the business logic to provide to implement the requirements must be implemented by the repository application, allowing the user to issue any command introduced in the task (section 3.1).

[1] http://uml.org/

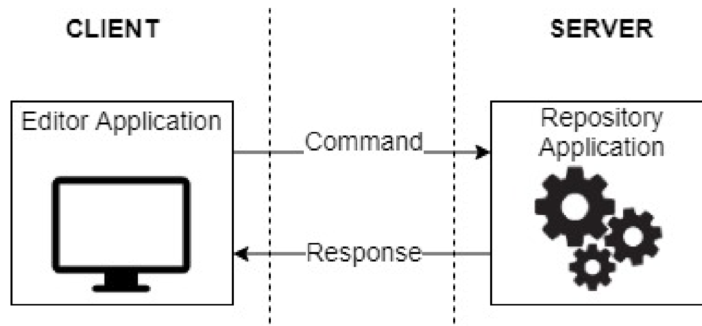


Figure 4.1: Top-level architecture of the OLAP pattern repository

4.2 Data Structure

The data structure that is presented in this section represents the organization and content elements described in the task (section 3.1) on a conceptual level (see also ORR) using UML class diagrams. Such a class diagram allows to depict classes, their comprised properties and relations to other classes. Basically, the data structure consists of four organization and four content elements, that form composition relationships. As per convention in many programming languages the names for all classes start with a capital letter. The classes for the organization elements are *Repository*, *MDM*, *Vocabulary*, and *Catalogue*. These organization elements comprise the corresponding four content elements, namely the *Cube*, *Dimension*, *Term*. and *Pattern* classes. Figure 4.2 provides an overview of these classes and their relationships.

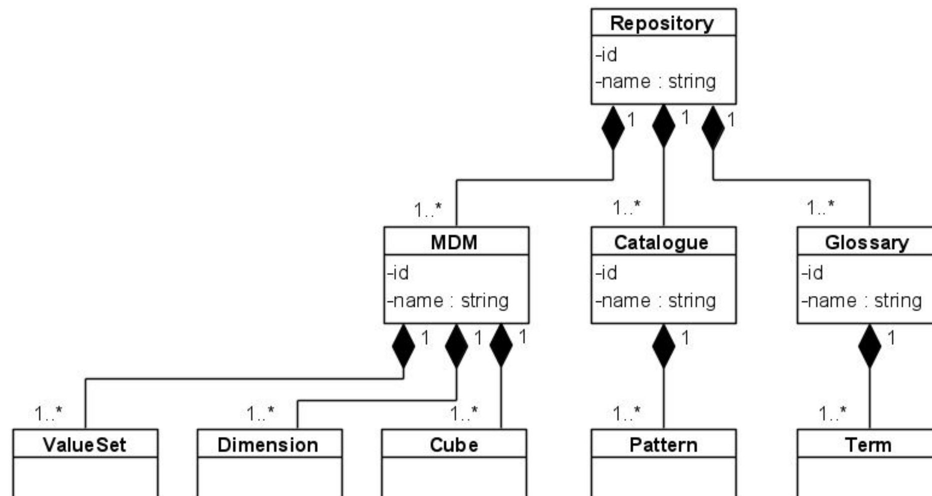


Figure 4.2: Class hierarchy as generated by the language processor

The *Repository* class has a composition relationship to the classes *MDM*, *Glossary*, and *Catalogue*. A *Repository* object must therefore contain one or more *MDM*, *Glossary*, and *Catalogue* objects. Each of these *MDM*, *Glossary*, and *Catalogue* objects on the other hand can only be part of one *Repository* object. Further, any organization element consists of two properties, namely a *name* and an *id* property. The class that comprises *Patterns* is the *Catalogue* class, which means its objects contain a non-empty set of *Pattern* objects. The same applies to *Glossary* objects which contain a non-empty set of *Term* objects. *MDM* objects in contrast contain three non-empty sets, i.e., one that comprises *Cube* objects, one that comprises *Dimension* objects, and a third one that comprises *ValueSet* objects. It is worth noting, that all these relationships are compositions, hence, an object may only exist as long as its containing object exists. A *Cube* object, for example,

cannot exist without an associated *MDM* object.

Considering the structure of an MDM in section 2.1, two types of elements, i.e., entities and properties, can be distinguished (see Figure 4.3). Hence, the class hierarchy consists of the common super class *MDMElement*, which has two specializations, namely the classes *MDMProperty* and *MDMEntity*. The *Cube* and *Dimension* classes are subsumed under the class *MDMEntity*, while the *CubeProperty* and the *DimensionProperty* are subsumed under the *MDMProperty* class.

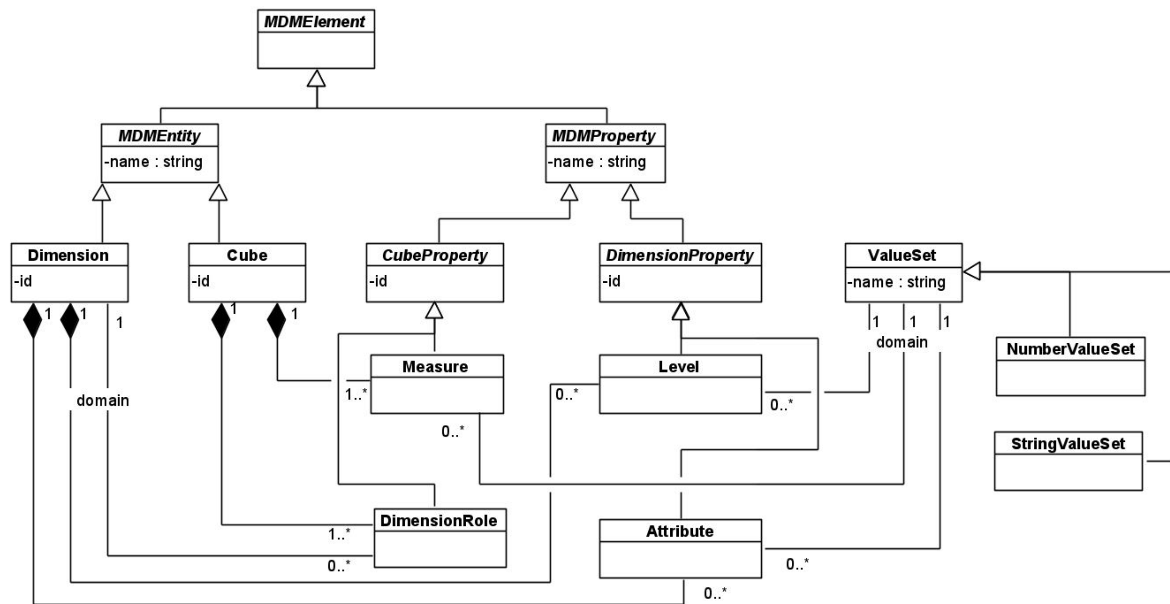


Figure 4.3: MDMElement class diagram

The subclasses of the *CubeProperty* class are the *Measure* and the *DimensionRole* class, while the *DimensionProperty* class subsumes both the *Level* and *Attribute* class. According to the *MDM* structure in section 2.1, a *Cube* object is related to *Measure* and *DimensionRole* objects, while a *Dimension* object contains multiple *Level* and *Attribute* objects as shown in Figure 4.3. Following the task (section 3.1) both *MDMEntity* and *MDMProperty* classes contain a *name* property, which is used to explicitly identify their objects. The relationships between *Cube* and *Measure* objects and respectively between *Cube* and *DimensionRole* object represent compositions, that is, neither a *Measure* nor a *DimensionRole* object can exist outside a *Cube* object. The same applies for *Level* and *Attribute* objects within a *Dimension* object. The *ValueSet* super class allows to define value types, i.e., *NumberValueSet* and *StringValueSet* objects, that can be associated to the *Measure*, *Level*, and *Attribute* object over the *domain* relationship. For *DimensionRole* objects the domain is represented by a *Dimension* object.

Besides MDM elements, the eMDM (section 2.1) consists of business terms, which are represented by the *Term* class (see Figure 4.3). As terms and patterns share a widely similar structure, one can benefit from a common super class in the class hierarchy. Therefore, both the *Term* class representing business terms and the *Pattern* class representing OLAP patterns are specializations of the generic *Context* class. A *Context* class (see Figure 4.4) has a *name* and an *id* property, and comprises classes representing variables, constraints, local cubes etc. This design allows to easily extend a business term to the same functionality as a pattern (if needed in the future) and thereby takes into account the extensibility of the application as formulated in the EXTR.

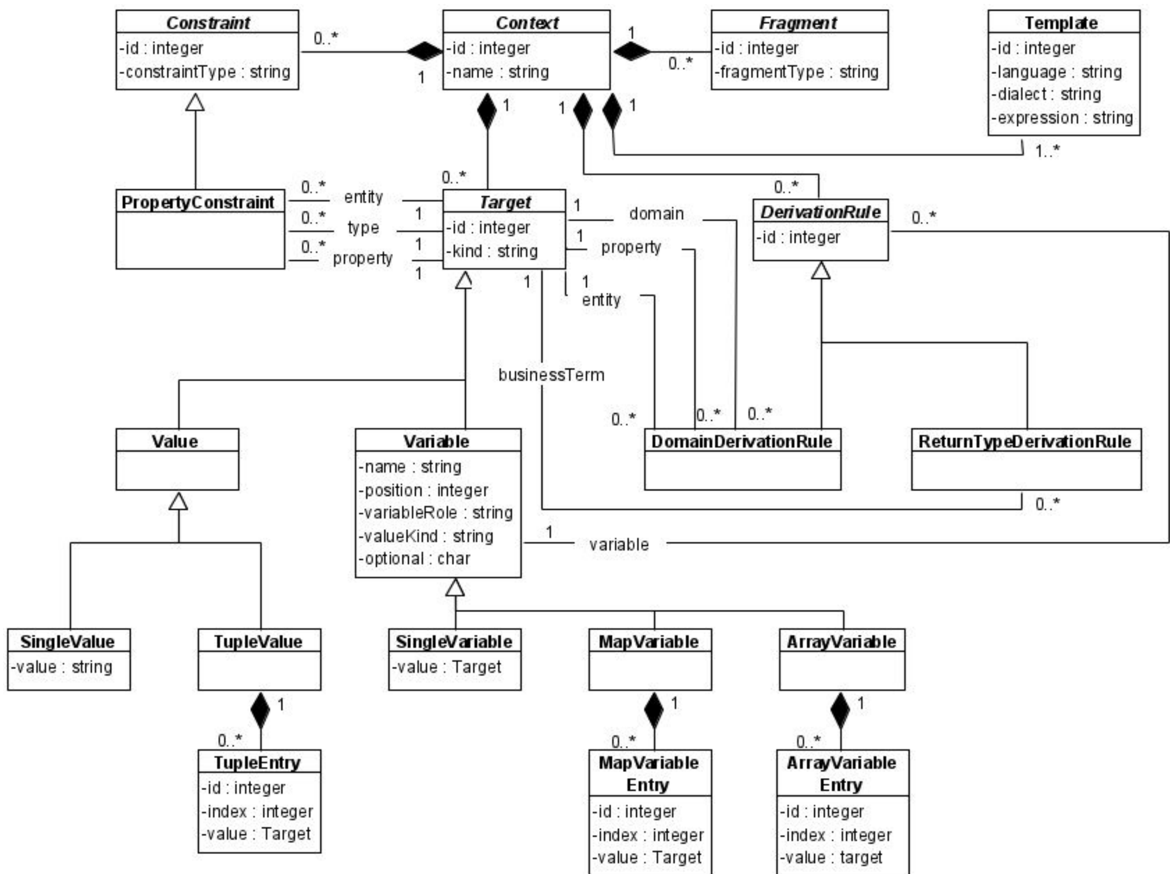


Figure 4.4: Context class diagram

The *Context* is composed of a set of *Targets*; a *Target* represents either a *Variable* or a *Value*; basically, a *Target* might occur as part of a *Constraint* or *LocalCube*. Each *Target* is identified by an *id* property and contains a *kind* property (discriminator attribute) stating whether it represents a "VARIABLE" or "VALUE". A *Variable* represents a parameter or derived element in a pattern or term context, whereas a *Value* may be part of a constraint or is assigned to a *Variable*. *Variables* consist of a name, the position in the context (-1 for a calculated), a variable role which is either "PARAMETER" or "CALCULATED", a value kind and an optional property. The value kind and optional property refers to extensions of the basic pattern approach described by Kovacic et al. [1]; value kind allows to distinguish variables representing a single value and those representing multiple values, i.e., Tuples, whereas the optional property allows to label parameters that need not be instantiated. Further extensions from Kovacic et al. [1] that have been considered in the design phase are collection variables, represented by *ArrayVariables* and a *MapVariables*, besides the regular *SingleVariables*. Consequently, *MapVariableEntries* and *ArrayVariableEntries* represent values assigned to a *MapVariable* or an *ArrayVariable* respectively; both collection variables may have multiple values assigned. Another extension that was considered in the design is the distinction of *SingleValues* and *TupleValues*, whereby a *SingleValues* represents only one value and *TupleValue* can be composed of one or more values.

Constraints depict an abstract constraint class, which consists only of an *id* and a *constraintType* property. Following section 2.2, constraints can be distingusihed into *TypeConstraints*, *DomainConstraints*, *PropertyConstraints*, *UnaryApplicableToConstraints*, *BinaryApplicableToConstraints*, and *ReturnConstraints*. Each *Constraint* sub-class references one or more *Target* objects as depicted

for the *PropertyConstraint* class in Figure 4.4. In Figure 4.5 the *Constraint* structure is depicted in more detail focusing on the *Target* references.

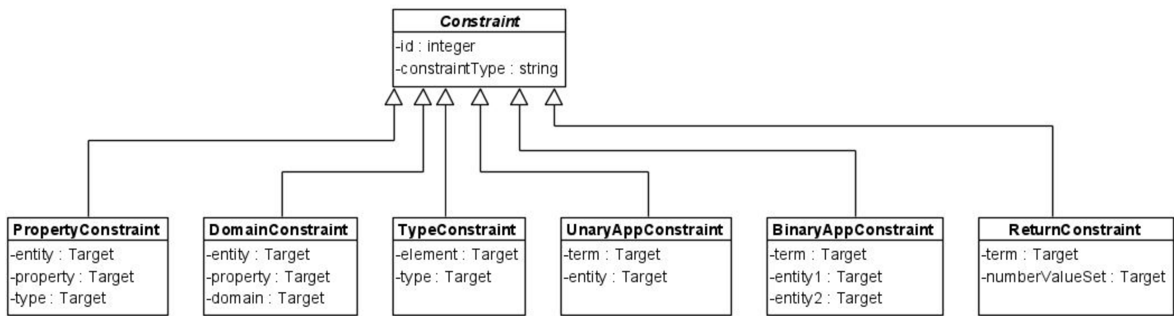


Figure 4.5: Constraint class diagram

A *PropertyConstraint* references three *Targets* referred to as entity, (property) type, and property (Figure 4.5). *DomainConstraints* also include three *Target* references, i.e., entity, property, and domain, whereas a *TypeConstraint* only references two *Target* objects, i.e., element and type. *UnaryApplicableToConstraints* include two *Target* references, i.e., the term and the entity to be applied to, while *BinaryApplicableToConstraints* include three *Target* references, i.e., the term and the two entities to be applied to. Lastly, *ReturnConstraints* include two *Target* references, i.e., the term and the number value set returned.

LocalCubes specifying local cubes mirror the structure of *Constraints* and thus have an *id* and a *localCubeType* property (Figure 4.3). Also the sub-classes are alike, that is, there is a *TypeLocalCube*, *PropertyLocalCube*, and a *DomainLocalCube*, which also have identical *Target* references.

The last class, which is related to the *Context* is the *Template* class, representing templates as described in section 2.2. As the template for both, patterns, and business terms includes an expression, language and dialect property, it is reasonable to relate it to the common super class.

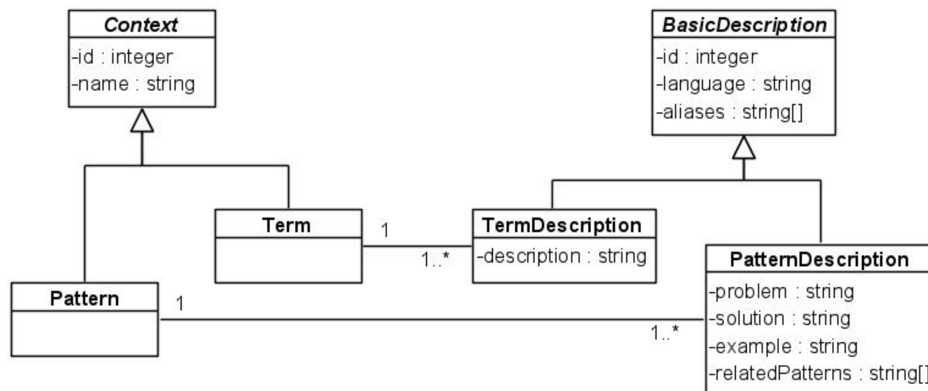


Figure 4.6: Class diagram for Context Descriptions

Even though a business term or pattern comprises already most components by specializing the *Context*, there is one that it is missing – the description. Descriptions however are not identical (see section 2.2), which is why a *TermDescription* and a *PatternDescription* are distinguished and related to the *Term* and *Pattern* class respectively (Figure 4.6). Nevertheless, they share common properties, i.e., *id*, *language*, and *aliases*, which can be grouped in the super class *BasicDescription*. The *TermDescription* class extends this basic version of a description only by the *description*

property, while the *PatternDescription* class extends it by four additional properties, namely the *problem*, *solution*, *example*, and the *related patterns*.

4.3 Repository Application

The architecture of the repository application follows the component-based approach as described in Sommerville [20, p. 452]; that is, a software is a composition of independent components that offer interfaces for communication (see also ICR). Considering the task description (section 3.1) an architecture containing six main components is proposed by this work, namely a *interface provider*, *language processor*, *data storage*, *knowledge based system*, *template processor*, and a *controller*. The UML component diagram in Figure 4.7 depicts how these components build up the OLAP pattern repository. Component diagrams are contained in the UML specifications and allow to determine the components comprised by an architecture.

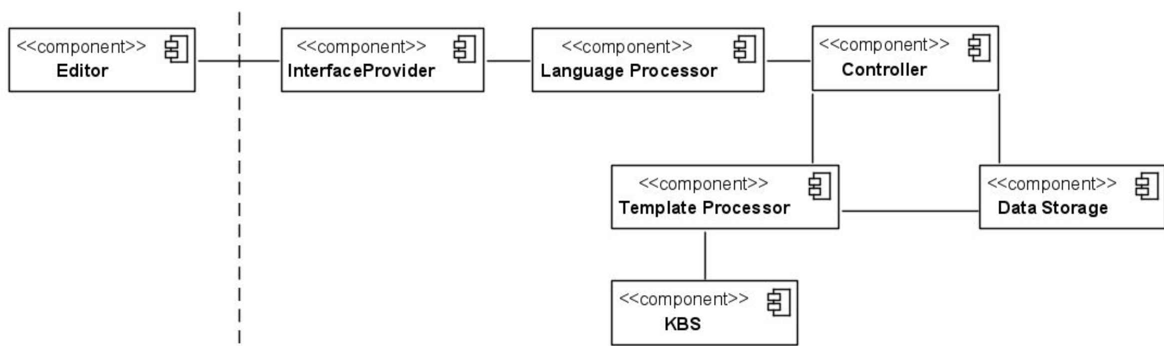


Figure 4.7: Repository application architecture

The *interface provider* offers an interface to the editor, receives OLAP pattern language commands and hands them over to the *language processor*. A *language processor* is needed to extract the information from the language statements and provide it in form of an internal object representation (see also LSPR). The *data storage* provides an interface to a database allowing to persist, delete, and retrieve organization and content elements. Checking whether or not a pattern is applicable to an eMDM or whether a term is applicable to a MDM or to a pattern's local cube respectively is the task for the *knowledge based system (KBS)*. To execute a pattern, the *template processor* performs the language processing steps (Figure 2.3) on the template's expression to identify and substitute the macro calls with the respective code snippets. The *controller* contains the business logic to decide how to process an object representation of a statement which is generated by the language processor, that is, it delegates tasks to the respective component. In addition, the controller generates a response that is sent to the editor, which is either a success or error message, a machine readable search result, or an executable query.

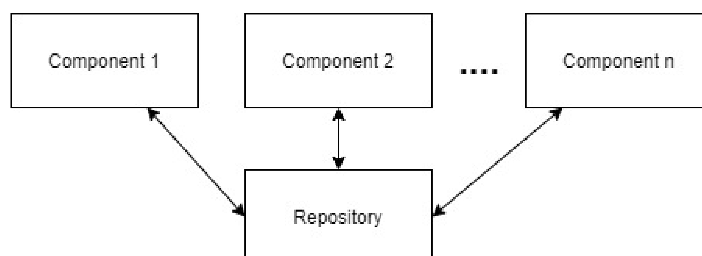


Figure 4.8: Generic repository architecture following Sommerville [20, p.160]

In terms of communication between the components, Sommerville [20, p. 159] introduces a repository reference architecture, which is applicable to any software, where large data needs to be centrally stored and accessed by various components. According to that, each component implements a business functionality and reads and/or writes to a central storage, called repository, independently of all other components. That is, the communication between components takes place only via the repository as depicted in Figure 4.8. For the repository application to be designed, however, this is not deemed suitable, as method calls are deemed necessary between various components e.g. between template processor executing patterns and the KBS validating them. For that reason, the communication within the repository application is handled using interfaces leading to the detailed repository application architecture as depicted in Figure 4.9.

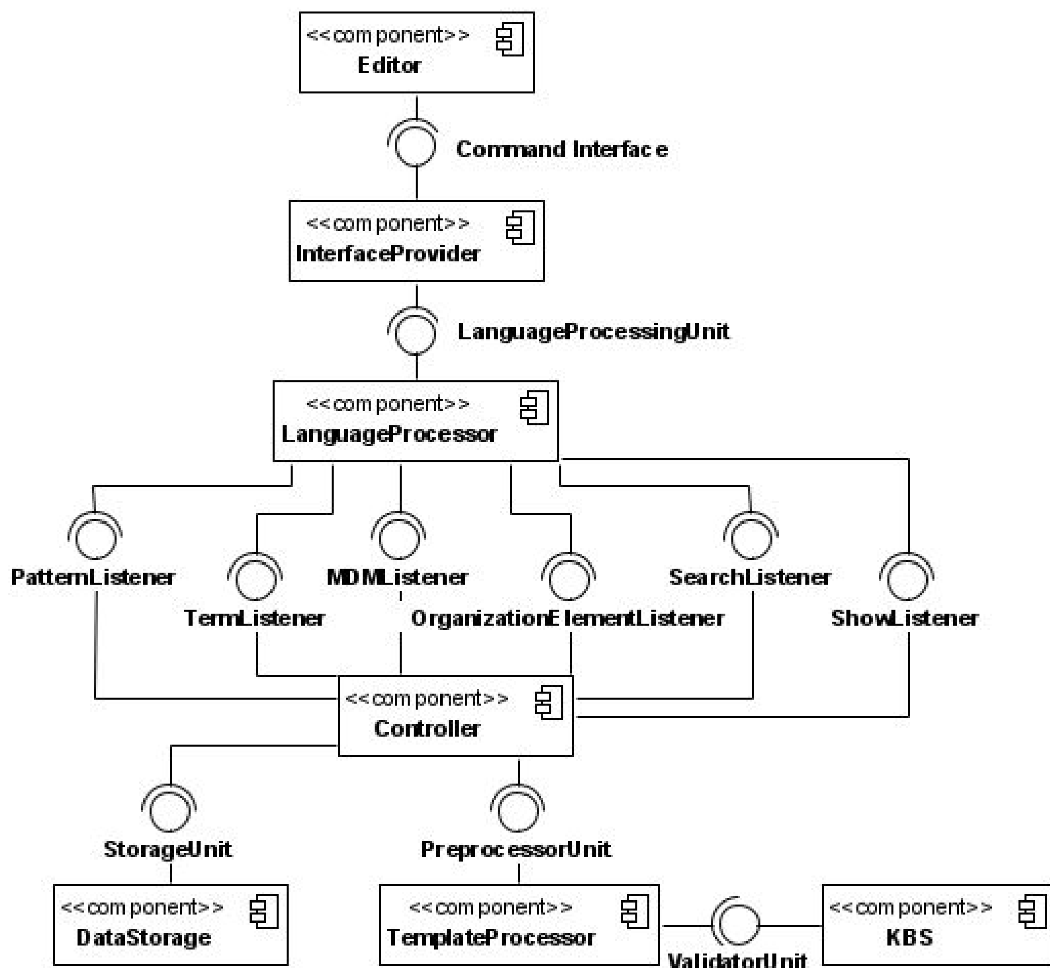


Figure 4.9: Communication interfaces of repository application components

The interface of the components are split into required and provided interfaces as specified by the UML component diagram. A required interface (semicircle icon) describes functionality that is needed by the component and must thus be provided by another component in the repository architecture. Provided interfaces (lollipop icon) describe functionality offered to other components, i.e., interfaces the component implements. In the following, the components are detailed by describing the necessary data structure, the interfaces they require and provide, and the behaviour implemented behind the provided interfaces. Initially, the controller design is detailed in subsection 4.3.1. Following the language processing (subsection 4.3.2) is introduced; the data storage is described in subsection 4.3.3. Finally, subsection 4.3.5 covers the template processor.

4.3.1 Controller

The controller encapsulates the business logic in the repository application, that is, the controller takes object representations generated by the language processor (event objects) and orchestrates the methods made available by required interfaces to ensure that both content and organization elements can be created as determined by the task (section 3.1). Consequently, the controller also orchestrates the functionality necessary to delete, search and show both types of elements in the same way. For patterns, the controller also includes the logic allowing to instantiate them. The grounding and execution logic, as contained in the pattern usage process (Figure 1.1), however, is the only behavior that is implemented in the template processor (subsection 4.3.5). Even though the controller also receives the object representations regarding executions, the actual logic to generate OLAP queries is contained in the template processor – the execution task therefore is completely delegated to the template processor. The following gives an overview of the event objects the controller is able to process in the structure section, describes specific interfaces provided by the controller and the behavior to implement the comprised methods.

Structure

This section focuses on the event objects the controller receives from the language processor and hence must be further processed. In general, six types of events, regarding pattern-, term-, mdm-, organisation-element-, search- and show-statements can be distinguished. Accordingly, a *PatternEvent* (Figure 4.10) is generated by the language processor in terms of create, delete, instantiate, and execute commands regarding pattern contexts, templates, and descriptions.

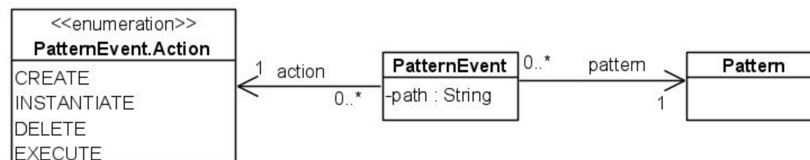


Figure 4.10: PatternEvent class diagram

PatternEvents therefore comprise a *Pattern* object, a path, and an action. The action enumeration indicates the operation to be performed, including CREATE, DELETE, INSTANTIATE and EXECUTE; the path determines the *Catalogue* holding the *Pattern*. The *Pattern* object follows the data structure in section 4.2 and comprises the necessary information on the pattern context, description, or template. It consist of the name, parameters, derived elements, constraints and local cubes, if a pattern context should be created; if the pattern should be deleted, it solely contains the name. In terms of a parsed create-statement for pattern templates and descriptions, the *Pattern* object contains a name and a *PatternDescription* or *Template* object comprising the corresponding text properties. A *PatternEvent* informing observers about an instantiation, however, contains a pattern object consisting solely of *Parameters*, which are associated with the bound values. If a *PatternEvent* is created for an execution, the contained *Pattern* only includes the name of a fully instantiated pattern to be executed and optionally contains a *Template* object if one or more specific template should be executed.

The *TermEvent* offers the same information as the *PatternEvent* with the distinction that it is related to business-term-specific statements; it notifies the controller that an OLAP pattern language statement containing a business term context, template or description is parsed. Consequently,

the class diagram is similar as well (see Figure 4.11), with one main distinction, i.e., it does not include INSTANTIATE or EXECUTE actions.

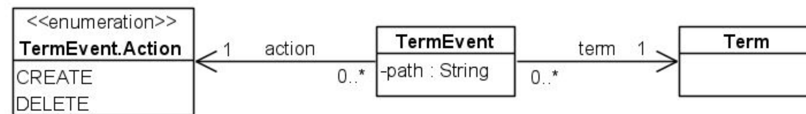


Figure 4.11: TermEvent class diagram

A *TermEvent* therefore consists of a *Term* object with a path to the containing *Glossary* and and action enumeration indicating whether it represents a create- or delete-statement. The *TermEvent*'s related *Term* comprises the term name, *Parameter*, and *Constraint* objects if a term context is to be created. In case of a description/template to be created, the *Term* object contains a name and a corresponding *TermDescription*/*Template* object. For delete-statements regarding the entire term, the *Term* object consists of the term's name only, whereas for descriptions/templates a *TermDescription/Template* is included determining the object(s) to be deleted.

MDMEvents are generated by the language processor once a *Cube* or a *Dimension* is parsed. It also comprises the action to be performed, the *MDMEntity* to be created or deleted and the path to the *MDM* (see Figure 4.12).

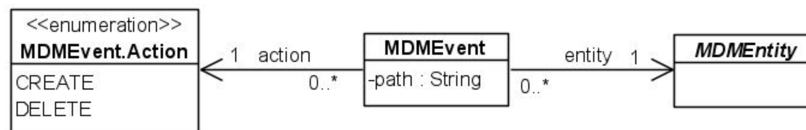


Figure 4.12: MDMEvent class diagram

For a parsed create statement regarding a cube the *MDMEntity* to be related is a *Cube* object that comprises *Measures* and *DimensionRole* objects. Correspondingly, for a create dimension statement, an *MDMEvent*, related to a *Dimension* object, is created by the language processor; this *Dimension* object contains *Level* and *Attribute* objects and the corresponding *RollUpRelations* and *DescribedByRelations*.

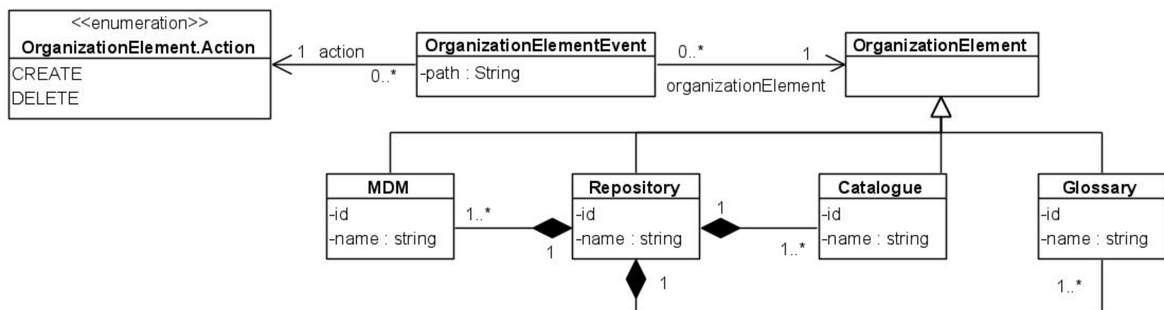


Figure 4.13: OrganizationElementEvent class diagram

An *OrganizationEvent* (Figure 4.13) is an object corresponding to a create- or delete statement for organization elements, namely repositories, catalogues, glossaries and MDMs. The *Action* enumeration therefore contains two states, CREATE and DELETE. For repositories the path property is empty, as they represent the root elements of organization hierarchies; for catalogues, glossaries and MDMs the path specifies the containing repository. The OrganizationElement

included is an object representing the element to be created/deleted and in both cases only contains the name.

Figure 4.14: SearchEvent class diagram

A *SearchEvent* is an object providing all parameters included in a search statement. Accordingly, there is a *searchTarget* representing the content or organisation element type to search for (e.g. **"PATTERN"**), the optional search space determined by the *searchSpace* parameter, and map (*searchStrings*) connecting search terms with the search scopes defined in the statement.

Lastly, a ShowEvent depicts a regarding show statement which only contains the string of the path to be shown. Hence, the ShowEvent also contains only one property, i.e. the path property, containing this path.

Interface

The controller's interface consists of six provided and two required interfaces. Specifically, the controller implements all observer interfaces that are required by the language processor (subsection 4.3.2) as depicted in Figure 4.15. That is, the language processor can notify the controller once any language statement is parsed, to further process the contained information.

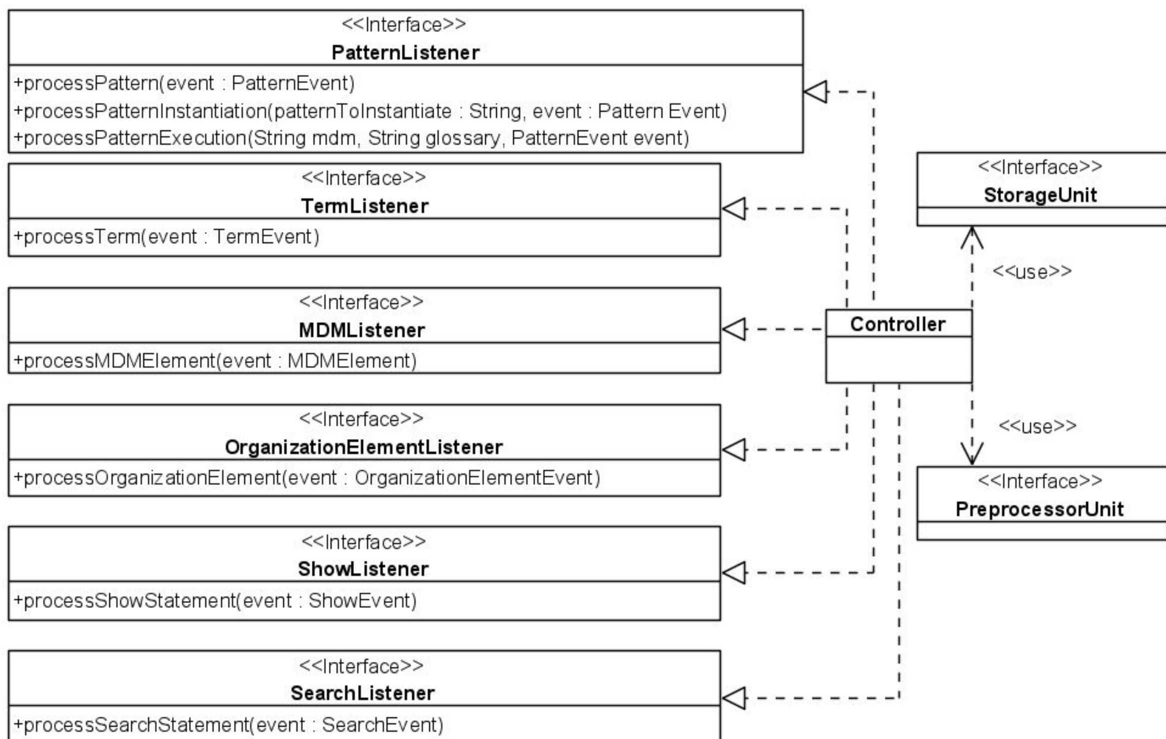


Figure 4.15: Controller Interface

The *PatternListener* interface allows for other components (in particular for the language processor in this work) to pass the information necessary to create, delete, instantiate or execute

a pattern. A *PatternListener* must therefore implement three methods, namely *processPattern*, processPatternInstantiation and *processPatternExecution* method. The language processor calls the *processPattern* methods to signal, that a pattern must be persisted or deleted; this pattern is depicted by the *PatternEvent* parameter. Respectively, the *processPatternInstantiation* method indicates, that a new instantiation of a pattern must be created, providing the path of the pattern to be instantiated (*toInstantiate*) and the bindings (PatternEvent). The *processPatternExecution* method signals that an OLAP query must be generated from a pattern, taking three parameters, namely the MDM and glossary paths as well as a *PatternEvent* representing the pattern.

Consequently, the *TermListener*, *MDMListener* and *OrganizationElementListener* allow to pass *TermEvents*, *MDMEvents* and *OrganizationEvents* that represent create or delete statements. The *TermListener* interface defines only one method, i.e., the *processTerm* method, allowing the language processor to inform about a term that must be created or deleted. The same holds true for the *processMDMElement* and the *processOrganizationElement* methods within the *MDMListener* and the *OrganizationElementListener*, i.e., both methods are used by the language processor to inform about an element that must be created or deleted.

Search statements are passed to the controller via the *processShowStatement* method within the *SearchListener* interface; show statements via the *processSearchStatement* method of the *ShowListener* respectively. Both methods take a corresponding event object as the only parameter.

In terms of required interfaces, the controller needs access to a *StorageUnit* and a *PreprocessorUnit* interface. That is, to process CREATE, DELETE, INSTANTIATE, SEARCH and SHOW statements for content and organization elements the controller uses the regarding methods provided by a *StorageUnit*. On the other hand, tasks concerning EXECUTE statements require the functionality as provided by a *PreprocessorUnit*. The specific methods for these interfaces can be taken from subsection 4.3.3 for the *StorageUnit* or subsection 4.3.5 for the *PreprocessorUnit*.

Behavior

The behavior describes the basic logic behind the provided interfaces, i.e. it introduces how event objects representing a language statement are processed. Therefore, the methods comprised by each interface as depicted in Figure 4.15 are implemented by the controller

Regarding the *SearchListener* interface, one method, namely *processSearchStatement*, must be implemented. The behavior behind a search operation is however database specific and thus provided by the storage unit, that is, the controller simply delegates search tasks.

Considering the *PatternListener* interface the controller implements the *processPattern*, *processPatternInstantiation* and *processPatternExecution* methods. As the logic regarding executions is determined by the template processor, the controller does only delegate the *PatternEvent* within the *processPatternExecution*. The *processPattern* method is designed to process *PatternEvents* comprising information to create or delete *Patterns*, *PatternDescriptions*, and *PatternTemplates*. The specific steps can be taken from the procedure in Figure 4.16. It is worth noting, that this process is basically the same for the *processTerm* method contained in the *TermListener* interface and for the *processMDMElement* in the *MDMListener* interface, since all these methods aim to create or delete a content element within an organization element.

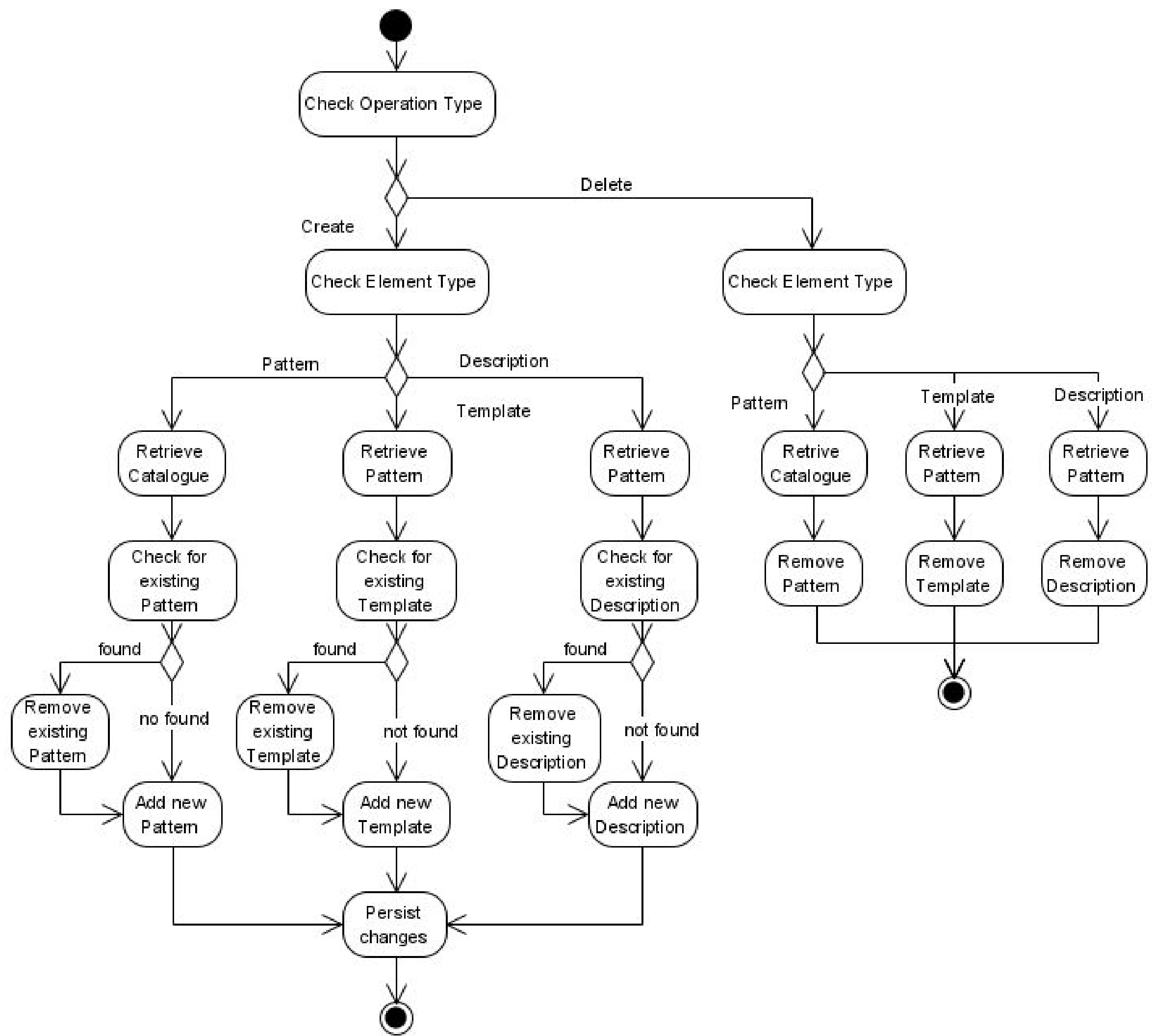


Figure 4.16: Procedure for processPattern method

The *processPattern* method checks in a first step, whether an element should be created or deleted. In case of a create, a subsequent check determines if a *Pattern*, *Template* or *PatternDescription* is the object to be added to the storage. For a pattern, the next step is to retrieve the catalogue it should be persisted in, by using the *getCatalogue* method provided by the *StorageUnit* interface. Further it must be assess if the catalogue already contains an equivalent pattern, that is, the *getPattern* method of the *StorageUnit* is called. It is worth noting, that a pattern is deemed equivalent, if it has the same name. If no existing pattern was found, the new pattern can be added to the catalogue's set of patterns; otherwise the existing pattern must be removed first (using the *deletePattern* method) to subsequently add the new version of the pattern. Finally, the changes must be persisted by calling the *persistCatalogue* method.

This process is in principle the same for the *Template* and *PatternDescription* objects as well. Instead of retrieving a catalogue, one however needs to retrieve a pattern (*getPattern*) to check if the comprised set of *Templates/PatternDescriptions* contains an identical element. If not, the *Template/PatternDescription* can be added, otherwise the existing one needs to be removed first (*deletePatternTemplate* or *deletePatternDescription*). Finally, the pattern with its updated set of *Template/PatternDescription* objects is persisted in the database using the *persistPattern* method.

If the check for the operation identified a delete-operation, the subsequent action to be performed

is a check for the element type es well. Again, if a pattern must be deleted, step two is to retrieve the containing catalogue (*getCatlogue*). Consecutively, the pattern must be removed from the catalogue and deleted from the storage using the *deletePattern* method provided by the *StorageUnit*.

To delete a *Template* or *PatternDescription* object, the procedure differs, as the delete-statements for these elements may demand to remove multiple objects from the data storage. However, the *StorageUnit* offers the *deletePatternTemplates* respectively *deletePatternDescriptions* methods that search and delete the intended objects.

For processing pattern instantiations, the controller implements the *processPatternInstantiation* method included in the *PatternListener* interface. This method is called by the language processor in case an `INSTANTIATE PATTERN` statement was parsed and has two parameters, i.e. the path of the pattern to be instantiated (in the following referred to as the generic pattern) and a *PatternEvent*, containing the *Pattern* object with all the variables and their values (referred to as specific pattern). To instantiate a pattern and to persist the resulting more specific pattern the controller implements the process in Figure 4.17.

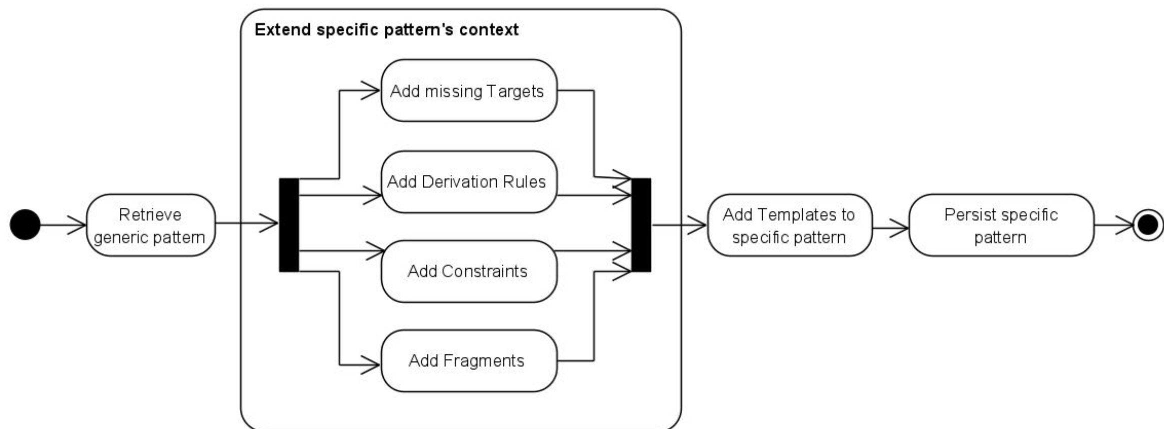


Figure 4.17: Activity diagram for processPatternInstantiation method

Firstly, the pattern to be instantiated, which is determined by the *toInstantiate* parameter, must be retrieved (i) using the *getPattern* method provided by the *StorageUnit*. Subsequently, the specific pattern (ii) must be extended by all targets, constraints, derived elements and local cubes which are contained in the pattern to be instantiated and not already included in the specific pattern. This refers to the usage process (subsection 2.2.2), according to which a specific pattern is equivalent to the generic one but with values bound to the variables. Hence, the "add derivation rules" action refers to a loop that generates a copy of every *DerivationRule* comprised by the pattern being instantiated and adds it to the specific pattern. The same applies for the "add targets" "add constraints" and "add local cubes" actions for *Target*, *Constraint* respectively *LocalCube* objects. As there is no natural order to these four steps, they are visualized as parallel in Figure 4.17. In a following step, the templates (iii) from the pattern being instantiated are also copied and added to the specific pattern, as, due to the equivalent variables, these are applicable to both. The descriptions of the pattern being implemented, however, cannot be applied to the specific pattern, considering that by instantiating a pattern e.g. the applicability changes. Finally, the newly generated, specific pattern must be persisted in the data storage.

The *processOrganizationElement* method is implemented by the controller as it provides the

OrganisationListener interface; this method is called by the language processor whenever a statement concerning a repository, MDM, catalogue or glossary was parsed. Even though the `processOrganizationElement` method basically performs similar operations (i.e. creating and deleting objects) as the *processPattern* method, the steps are slightly different (see Figure 4.18).

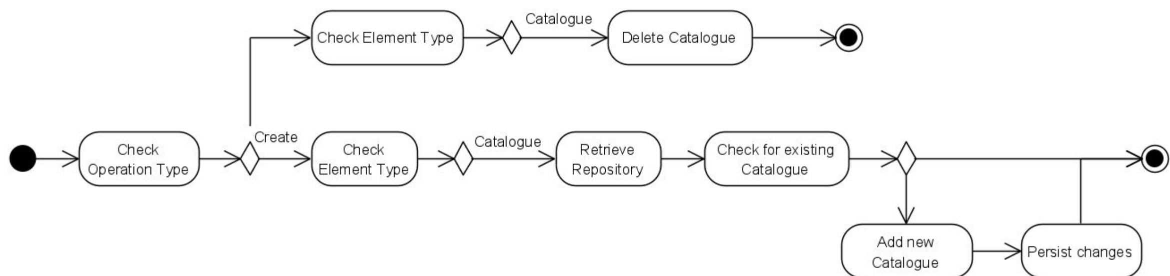


Figure 4.18: Exemplary activity diagram for Catalogues in the processOrganizationElement method

Initially, the action comprised by the *OrganizationEvent* parameter is checked, to distinguish between create and delete operations. Afterwards, a type check is performed on the *OrganizationElement* included in the *OrganizationEvent*; the example in Figure 4.18 visualizes the process for a *Catalogue* only, since the sequence is similar for *Repository*, *MDM* or *Glossary* objects. If it was determined, that a Catalogue must be added to the repository, the controller retrieves the *Repository* defined by the *OrganizationEvent*'s path parameter. Thereafter, the *Repository*'s catalogues are browsed, to find out whether there is already a *Catalogue* included, having the same name. In case such a *Catalogue* already exists, the process ends at this point, as re-creating organisation elements is pointless, otherwise, the new *Catalogue* is added to the *Repository* which is then persisted. If a Catalogue must be deleted instead, it is sufficient to simply call the *deleteCatalogue* method as provided by the *StorageUnit*.

The provision of the *ShowListener* interface requires the controller to implement the *processShowStatement* method, which is used by the language processor to inform about a parsed show statement. For this purpose the process in Figure 4.19 is implemented.

In a first step, the *Repository*, determined by the first part in the path, is retrieved; if the path contains only one part, the *Repository* representation is subsequently returned. As a show statement does not include an element designation, the process basically includes a try and error procedure. That is, whether the second part of the path is a MDM, glossary or catalogue can only be decided by trying to retrieve all types of elements. If the path is an MDM path and consists only of two parts the corresponding representation is returned, otherwise, the next step is to determine if the third part is a cube or dimension name. Again, it must be tried to retrieve both, to decide which representation is to be returned. If the second part of the path states a catalogue name instead, the subsequent step to be performed for a path containing two parts is to return the catalogue representation. Otherwise, the third part specifies a pattern name, which is retrieved to return its definition. It is worth noting, that the procedure for showing catalogues and glossaries respectively patterns and business terms are identical. Hence, the procedure in Figure 4.19 is a simplified version, not including the redundant steps for glossaries and business terms.

Figure 4.19: Activity diagram for processShowStatement method

4.3.2 Language Processor

The language processor encapsulates the functionality which is necessary for processing OLAP pattern language statements and represent them in form of event objects (see subsection 4.3.1). It provides the functionality of language processing as described in section 2.4, thus, it consists of the three sub-components lexer, parser, and semantic analyzer. The architecture of the language processor is as depicted in Figure 4.20.

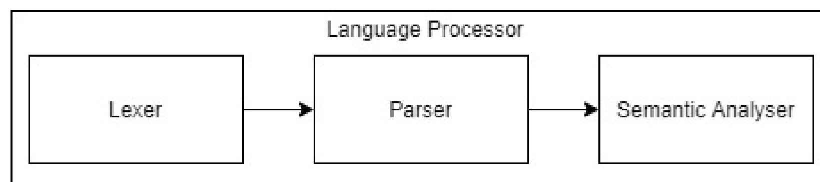


Figure 4.20: Language processor's arcitecture

Considering, that the aim is to create an object representation, the language processor follows the

approach of Appel [18, p. 94] and does not create another AST in the semantic analysis. The language processor takes a language statement following the task in section 3.1, generates an object representation (section 4.2), and passes the result on to interested components. Therefore, the language processor is designed to be a subject as defined in the observer pattern (also known as listener pattern) following Gamma et al. [13, p. 326]. A subject in this context is considered to be an object, which notifies a list of observers (listeners) about any state changes via specific events. The following describes the interfaces to provide these events, and the behavior behind the interface.

Interface

The language processor provides only one interface, namely the `LanguageProcessingUnit` interface, which determines necessary methods for processing OLAP pattern language statements. To provide this interface, the language processor must implement all contained methods depicted in Figure 4.21.

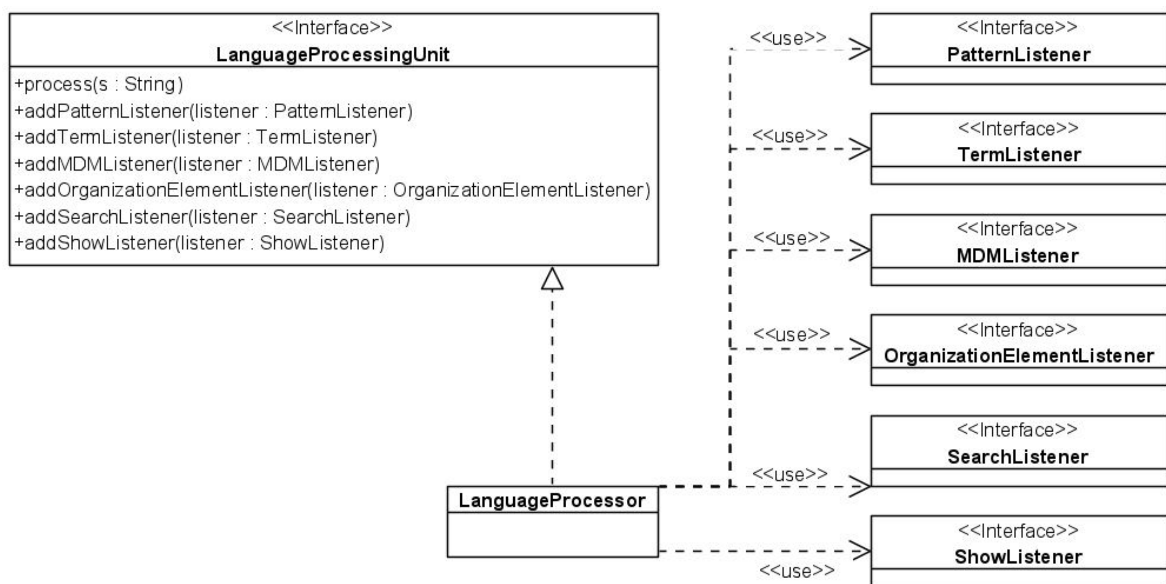


Figure 4.21: language processor interface

In the *LanguageProcessingUnit* seven methods are comprised, six of which allow to register a listener component. This means, that e.g. *addPatternListener* takes a *PatternListener*, which is informed about any parsed statement regarding patterns from that moment on. The key function for the *LanguageProcessingUnit* however is the *process* method, which takes a string containing a OLAP pattern language statement, generates an event object and informs the registered listeners.

From other components, i.e., the observers (listeners), the language processor requires six interfaces, to be implemented. It is worth noting, that from the language processor's perspective, it is not important if there is one component providing all seven interfaces (like the controller) or if there are seven components providing one interface each. However, the language processor requires to access one *PatternListener*, *TermListener*, *MDMListener*, *OrganizationElementListener*, *SearchListener* and *ShowListener* interface. The interfaces correspond to the event objects introduced for the controller (see also subsection 4.3.1) and hence allow for the language processor to inform the controller about a parsed statement. The behavior implemented by the observers (the controller), is not important to the language processor, as it is not further involved into processing the object

representations. Generally, a component that provides the *PatternListener* interface is able to process *PatternEvent* objects, whereas *TermListeners* are able to process *TermEvent* objects. *MDMListeners* offer logic to process *Cube* and *Dimension* withing *MDMEvent* objects, while *OrganizationElementListeners* provide functionality to process *Repository*, *Catalogue*, *Glossary*, and *MDM* objects comprised by *OrganizationEvents*. The functionality to retrieve any kind of element is provided by *SearchListeners* and *ShowListeners* respectively.

Behaviour

The behavior of the language processor is determined by the language processing steps introduced section 2.4. That is, (i) the *Lexer* is responsible for tokenizing the input statements according to the OLAP pattern language (Appendix A). Tokens that are defined by the grammar are basically the keywords described in the task (section 3.1) as well as strings, numbers, or special symbols ("/", "=", ";", etc.). Basically, tokens subsume keywords for the operation names, such as `CREATE`, `DELETE`, and `INSTANTIATE`, keywords for organisation and content elements, such as `REPOSITORY`, `CUBE`, and `PATTERN`, as well as keywords indicating blocks within a statement, such as `PARAMETERS`, `LEVELS`, and `MEASURES`. Strings, as used for element and variable names, are quoted identifiers and can contain white-spaces, numbers that comprise one or more digits, and special symbols, such as "=", "<", ">", and "/".

Figure 4.22: Tokens of the "Jersey Breed" context (Listing 3.7)

Accordingly, line 1 in Listing 3.7 is tokenized as depicted in Figure 4.22 by the blue squares. The three keywords in capital letters in Figure 4.22 are `CREATE`, `UNARY_DIMENSION_PREDICATE`, and `WITH`. Further, the line contains a path consisting of the three strings "Austrian Milk Models", "Dairy Business Terms", and "Jersey", which are separated by two special symbols, i.e. the "/".

Once a statement is tokenized, (ii) the *Parser* tries to match token sequences to the grammar rules – defined by the OLAP pattern language grammar – and generates a corresponding AST.

An incomplete tree for the tokens detected in Figure 4.22 is visualized in Figure 4.23. It is worth noting that the recognized tokens are highlighted in blue, in contrast to the white squares containing the matched rule names. At the top of the AST the base rule (`emdm_stmt`) is matched, which is the case for any valid OLAP pattern language statement. For this example, a create statement (`c_stmt`) is recognized which always consists of the `CREATE` keyword, and an element specific create-rule, in this case the one for a business term creation is `ct_stmt`. Further, the `ct_stmt` rule is matched to be a term definition (`t_def`), which is abbreviated for this section and therefore only consists of a term type (`t_type`), a term name (`t_name`), and the key word `WITH`. The type is matched to the `UNARY_DIMENSION_PREDICATE` keyword, the name however corresponds to the last name of a path (`path_exp`). Any path in context of the OLAP pattern language consists of multiple element names (`elem_name`), i.e., strings, which are separated by "/" symbols. The `path_exp` in Figure 4.23 comprises three strings and the two special symbols recognized by the *Lexer*. In the context of a term, the first string **"Austrian Milk Models"** is the repository name, string two **"Dairy Business Terms"** is the glossary name and the third string **"Jersey"** is the term name.

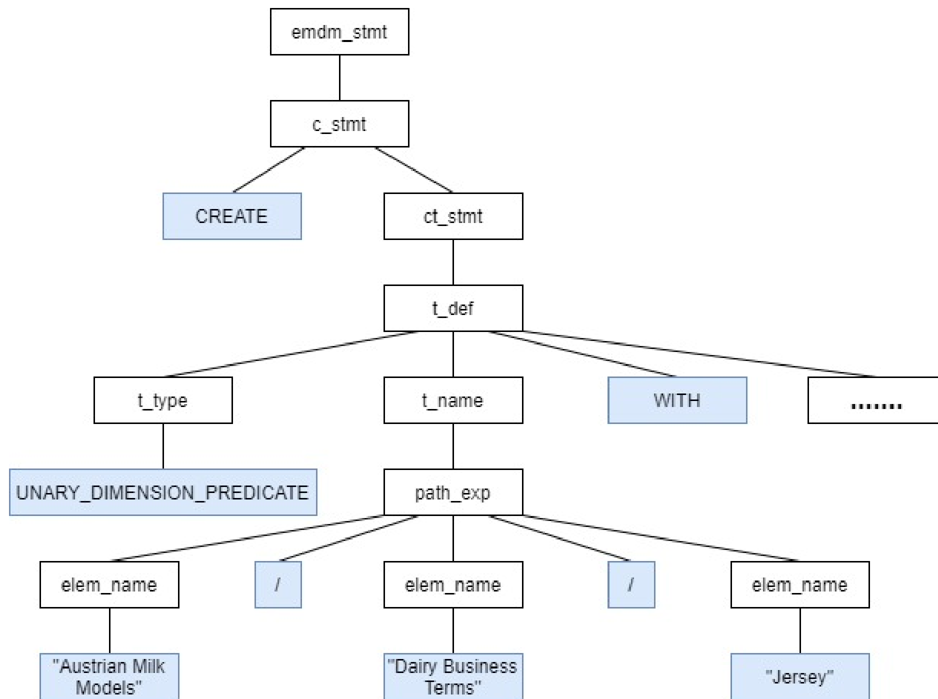


Figure 4.23: AST of the "Jersey Breed" context (Listing 3.7)

During the semantic analysis, (iii) the language processor must transform the AST into an object representation corresponding to the data structure introduced in section 4.2. Therefore, the assignment of the tokens to the rules are analysed to generate the corresponding objects, set the correct values for the properties and depict the relationships between the objects. Further more, logical constraints need to be checked, such as checking whether the constraints only contain parameters available in the context, or where a path expression contains the right number of steps to identify a location that is valid for the content element to be created.

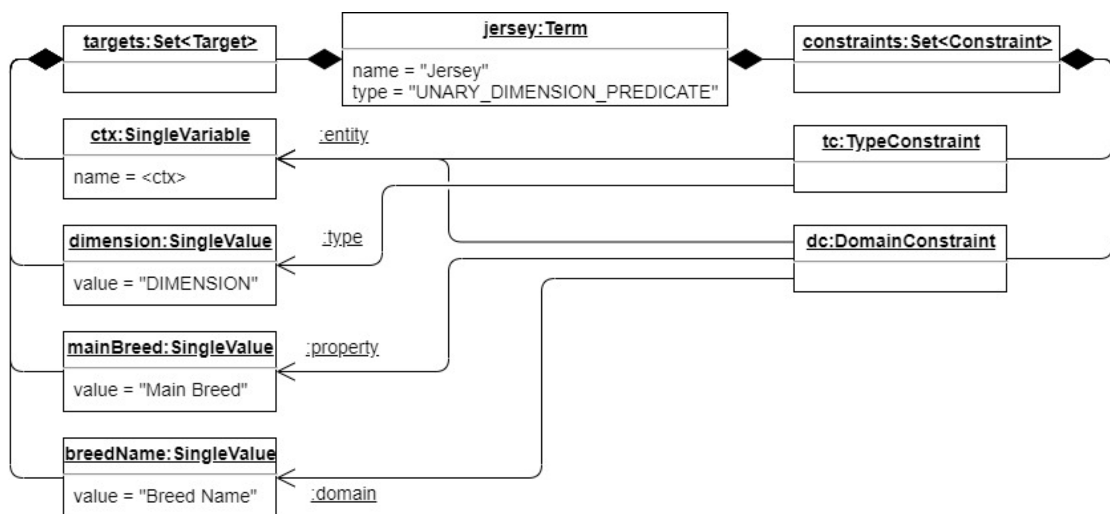


Figure 4.24: Object representing the Jersey unary dimension predicate's context (Listing 3.7)

In Figure 4.24 the object representing the "Jersery" business term's context is visualized. The parser needs to create an empty *Term* object first, to subsequently add its type and name. The type also determines the implicit term parameter(s) and type constraint(s) that must be added; in this case one <ctx> *Variable* restricted to a "DIMENSION" *Value*. Further *Constraint* objects are added while iterating over the rules regarding the `CONSTRAINTS` block in the AST; the parser

creates new *Value* and *Constraint* objects and adds them to the respective sets. The context also consists of a domain constraint, that is, a *DomainConstraint* object needs to be created, that references the <ctx> variable as its entity, a *SingleValue* (**"Main Breed"**) as property, and another *SingleValue* (**"Breed Name"**) as its domain. Lastly, the language processor must inform all observers, and provide them with the final object representation. Therefore, a *TermEvent* is created comprising the *Term* object from Figure 4.24, the path ("Austrian Milk Models"/"Dairy Business Term") and the CREATE action. This *TermEvent* is passed to the observers by calling the proccessPattern method of the *PatternListener* interface.

4.3.3 Data Storage

The data storage provides, database functionality allowing to create, delete, and retrieve both organization and content elements. It encapsulates a database in the background and provides an interface that allows for other components to only work with objects following the data structure in section 4.2. For that reason the design does not determine which type of database must be used, that is, the data storage enables an application design which is decoupled from the implementated database technology (see also ICR). Hence, there is no predefined behavior for this component, as CRUD operations on databases highly differ based on the implemented technology; therefore this section only describes the provided interface.

Interface

The interface of the data storage consists of only one provided interface, namely the `StorageUnit` interface. There are no required interfaces, as the data storage should be open to adapt to any kind of database. In Figure 4.25 there is an unspecified interface between the database and data storage, visualizing that the data storage must work with any interface provided by the database.

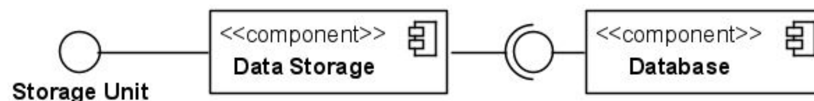


Figure 4.25: Data storage interface

The `StorageUnit` interface on the other side, clearly defines which methods the data storage offers for other components to access the persisted content and organization elements. Basically, there are methods to create, delete, and retrieve any type of element; update functions are not included, as following the UCER an update is to be performed as a re-creation.

Figure 4.26 depicts the detailed methods comprised by the `StorageUnit` interface. With the `openSession` method, components can start a regular database session, which opens a database connection and enables to access the database content. Respectively, the `closeSession` method is intended to be called by a component, once the database actions are completed, to close the database connection. That is, the following methods are intended to be used while a session is open.

To create content and organization elements within the database, the `StorageUnit` interface provides six persist methods, i.e. `persistRepository`, `persistCatalogue`, `persistGlossary`, `persistMDM`, `persistPattern`, and `persistTerm`. Therefore, each method has got one parameter which is an object to be persisted. However, it should be noted, that persist-methods only store

Figure 4.26: Class diagram of required and provided data storage interfaces

the given object in the database, that is, the data storage does not take care of inserting the objects into the organizational structure. To insert a pattern into the organization structure, one

would first want to retrieve a catalogue, add the pattern to this catalogue if possible and than call the persistCatalogue to persist the changes.

In contrast, the delete-methods allow to remove objects from the database. Specifically, the interface offers delete-methods for organization elements, i.e., deleteRepository, deleteCatalogue, deleteMDM and deleteGlossary, for patterns, i.e. deletePattern, deletePatternDescription, and deletePatternTemplate, for business terms, i.e., deleteTerm, deleteTermDescription, and deleteTermTemplate, and for MDM entities, i.e. deleteCube and deleteDimension. For each delete-method there are two different versions, one that allows pass the object to be deleted, and one that allows to pass the path. In context of a pattern, for example, the path to be passed to the deletePattern method consists of a repository, catalogue, and pattern name.

For retrieving elements the StorageUnit interface provides get-methods allowing for other components to access specific elements within the database. Therefore, one must pass the path and the unique element name to the StorageUnit and in return gets the intended object. Get methods are available for every type of element, which is why the following specific methods are comprised: getRepository, getCatalogue, getGlossary, getMDM regarding organization elements, getCube, getDimension, getTerm, getPattern. Additionally, the getType-method allows to retrieve the type of a term. For objects without unique name – *PatternTemplates* and *TermTemplates* – one must provide a path and a *Template* object that determines the database entry to be returned.

Lastly, the search-methods allow to retrieve elements matching the search criteria corresponding to search statements as introduced in section 3.1. Hence, the searchEvent parameter allows to provide the search target, the search space, and the search terms with the regarding search scopes. That is, the data storage is able to process search events as created by the language processor.

4.3.4 Knowledge-based System (KBS)

The Knowledge Based System (KBS) implements the necessary functionality to decide whether a pattern is applicable to a eMDM or not. Therefore it checks if the values bound to the parameters do represent business terms or MDM entities that are actually part of the eMDM, and if the constraints defined for each parameter hold true for the instantiated pattern. However, the KBS is not implemented in the course of this work, which is why no behavior is introduced either. Instead, this work defines an interface it must implement, so that it can be considered in the design of other components.

Interface

A ValidatorUnit provides the necessary functionality to decide whether a pattern is applicable for a MDM using a specific glossary. Thus, it consists of one method only, namely the validate method as shown in Figure 4.27.

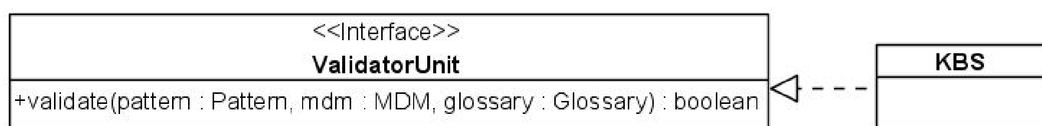


Figure 4.27: KBS interface

This method allows to validate a pattern in the context of a specific MDM enriched by a certain glossary. A boolean return value indicates if an execution with these parameters is possible or not. As the parameters are object representations of the pattern, MDM and glossary, the KBS can directly perform all necessary checks and hence does not require any other interface to implement the intended knowledge.

4.3.5 Template Processor

The template processor implements the grounding and execution steps as comprised by the usage process and therefore creates executable OLAP queries from pattern templates. In the course of this, the template processor contains the functionality to resolve derivation rules (pattern grounding) and to resolve macro calls within template expressions. For template expressions, an island grammar (Macro Grammar, Appendix B) defines rules to distinguish between query code and macro calls and further allows to identify the macro's parameters. That is, the template processor must implement the language processing functionality as described in section 2.4 and therefore consist of similar sub-components as the language processor (subsection 4.3.2), namely a *macro lexer*, *macro parser* and a *semantic macro analyzer*. It further communicates with a `ValidatorUnit`, as implemented by the `KBS`, in order to carry out an applicability check. The following sections describe the interface of the template processor the behavior needed to provide this interface.

Interface

The interface of the template processor subsumes two provided and two required interfaces (Figure 4.28).

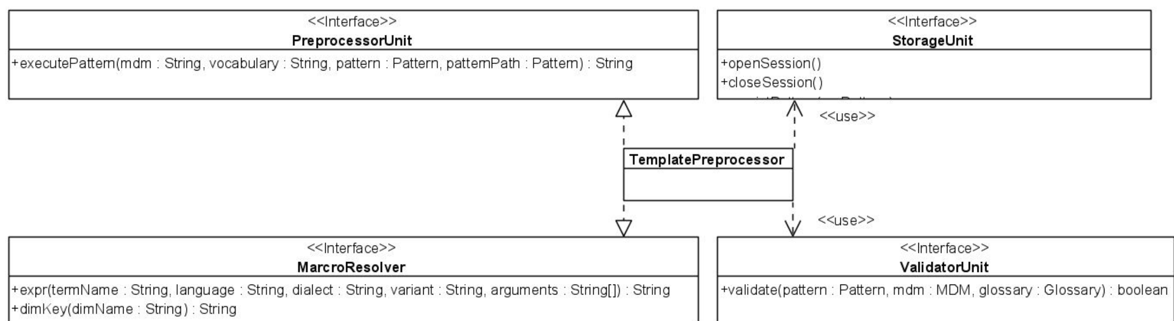


Figure 4.28: Template preprocessor interface

Firstly, the `PreprocessorUnit` interface is provided, comprising the *executePattern* method to execute an instantiated pattern following the subsection 2.2.2. It expects a MDM path, a glossary path, a *Pattern* object, and the pattern path as parameters in order to return an executable query. If every pattern's template must be executed, the pattern object should only contain the pattern's name, however, if specific templates should be executed, a *Template* object must be included that determines the templates to be executed by specifying the attributes, i.e., language, dialect, data model, and variant.

The second interface provided is the `MacroResolver` interface which contains the necessary functionality for the execution of macro calls (`$dimKey`- and `$expr`- macro calls), that is, an `expr`- and a `dimKey`-method. The `dimKey`-method, accordingly takes a dimension name (*dimName*) as a

parameter and returns the name of its base level. The expr-method allows to pass a term name as well as the language and dialect of the template to be executed, and the arguments to be bound to term's parameters in the term's template. With these parameters, the aim is to gain the query snippet representing the term's template in the demanded query language and dialect by substituting the <ctx> placeholders with the passed arguments.

Required interfaces on the other hand are the `StorageUnit` and the `ValidatorUnit`. The `StorageUnit` is required to gain access to the necessary eMDM elements and patterns for both the execution and the resolving of macros. Therefore the template processor only requires a subset of the provides `StorageUnit` methods, i.e., the get-methods. This means, the template processor does neither create nor delete elements from the storage.

A `ValidatorUnit` is needed to determine whether a pattern can be executed in the context of a certain eMDM. Accordingly, the interface requires to pass three parameters to the `validate`-method, i.e. the *Pattern*, *MDM*, and the *Glossary* object. The returned boolean value allows to decide whether an execution can be performed or has to be aborted by returning an error message.

Behavior

The template processor implements a separate behavior for each method it provides through the *PreprocessorUnit* and *MacroResolver* interface. The `executePattern` method allows for other components to pass a MDM and a glossary path providing the context and a *Pattern* object and a pattern path to specify which template(s) must be executed. By following the procedure depicted in Figure 4.29, the template processor implements the necessary functionality to transform template expressions of fully instantiated patterns into executable OLAP queries.

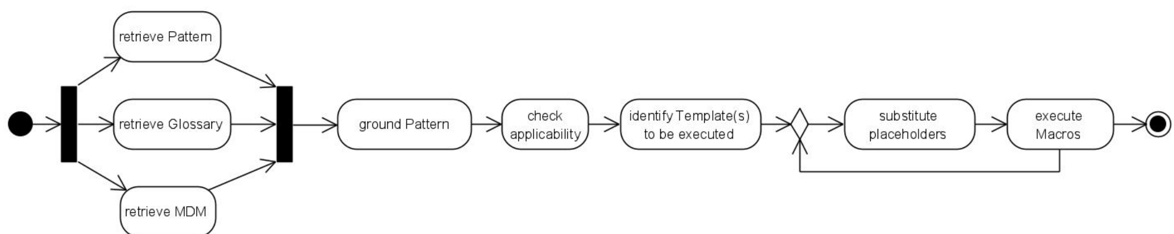


Figure 4.29: Procedure behind the executePattern method

First of all, (i) the *MDM*, *Glossary* and *Pattern* must be retrieved from the *StorageUnit*. It is worth noting, that even though one can pass a *Pattern* object, this is not the actual *Pattern* to be processed, but rather an object that was created by the language processor (Figure 4.20) to represent the execute statement. As there is no particular order in which the *MDM*, *Glossary* and *Pattern* should be retrieved these actions can be executed in parallel. Subsequently, (ii) one needs to ground the *Pattern* to get the values for the derived elements. The grounding procedure, however, is not part of this work and thus only a call of an internal method of the template processor. Once the *Pattern* is grounded, (iii) an applicability check is performed to assure it can be applied in the context of the *MDM* and *Glossary*. That is, the *validate* method of the *ValidatorUnit* (implemented by the *KBS*) is called, returning true if it is applicable and false if not. In the next step (iv) templates to be executed need to be identified, which are determined by the *Pattern* object. The *Pattern* object may contain a *Template* which includes the information about the templates the user wants to execute, i.e. the language, dialect, data model, and variant.

If no *Template* is contained, all available templates of the corresponding *Pattern* are executed. For every template (v) the first step is to replace the variable-placeholders in its expression with the values assigned to them during the instantiation, or for derived elements, with the values derived during the grounding process.

```
*{SELECT *
FROM <sourceCube> s
WHERE}* $expr(<cubeSlice>, s)
```

Listing 4.1: Uninstantiated template expression

Considering the example expression in Listing 4.1, the placeholder substitution consists of replacing the **"<sourceCube>"** as well as the **"<cubeSlice>"** placeholders by the bound values. After this step, the expression only consists of query code and macro calls. Hence, the last step (vi) is to execute the macros calls and substituted them by their results to gain the executable query. To do so, the template processor performs language processing on the template's expression by applying the macro grammar (see Appendix B). Following the language processing (section 2.4) steps, the *MacroLexer* identifies tokens, which the *MacroParser* puts into a tree structure, the AST. For the sake of brevity the lexical and syntactical analysis for an expression are not demonstrated at this point, as both follow the explanations of section 2.4. To demonstrate the semantic analysis, the placeholders in the expression from Listing 4.1 are substituted; specifically, the <sourceCube> placeholder with the value **"Feeding"** and the <cubeSlice> placeholder with **"Jersey"**. Lexical and semantic analysis of the thus instantiated template result in the AST depicted in Figure 4.30.

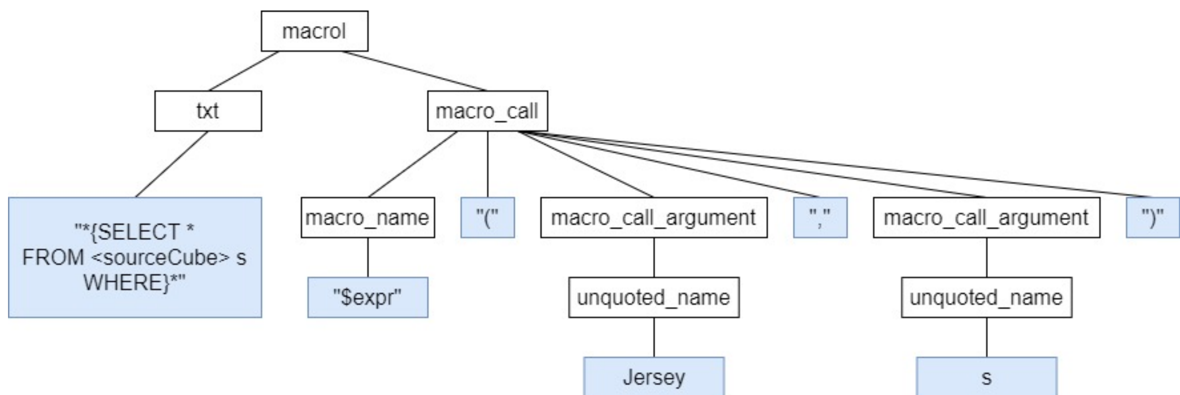


Figure 4.30: AST created by the MacroParser

There is one query text part (`txt`) and one macro call (`macro_call`) contained in the AST. The `txt` node contains the query code, in this example SQL code, surrounded by `"*{"` and `"}*"`. The macro call consists of the macro to be executed (`macro_name`) and its arguments (`macro_call_arguments`). Based on this AST, the *SemanticMacroAnalyzer*, is responsible for generating an actual query from the expression. In contrast to the language processor (subsection 4.3.2), the goal for the semantic analysis is not to generate an object representation, but rather to generate a new, modified version of the initial statement. Accordingly, the *SemanticMacroAnalyzer* removes the surrounding `"*{"` and `"}*"` of `txt` nodes, and replaces the macro calls by their results to generate a query code that is free of any OLAP pattern specific symbols or function calls. For the macro call in the example expression, the *SemanticMacroAnalyzer* identifies an *$expr* macro, that should return the query snippet representing the "Jersey" business term with

the <ctx> parameter bound to the value "s". Hence, the *expr* method of the *MacroResolver* interface is called with **"Jersey"** as the term name, the language and dialect identical to the pattern template (in this example "SQL" language and "Oraclev11" dialect) and "s" as the only argument. As a result, the **"$expr(Jersey, s)"** call in the initial template is replaced by the instantiated Jersey snippet, i.e. **"s."Main Breed" = "Jersey""**.

Besides the behavior regarding the execution of patterns, the template processor also implements functionality regarding the *MacroResolver* interface; it implements a method to execute a *$expr* and *$dimKey* macro. The *expr* function, corresponding to the *$expr* macro, allows to pass a term name for which the query snippet determined by the language and dialect parameters is executed by binding the arguments parameters to the <ctx> variable. This process is detailed in Figure 4.31.

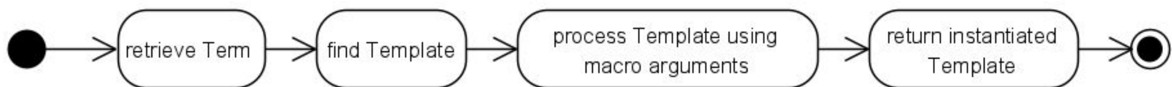


Figure 4.31: Procedure behind the expr method

First of all, (i) one needs to retrieve the *Term* object to be executed, by calling the corresponding method offered by the *StorageUnit*. As the *expr* method is only intended to be called in terms of an execution process, the term path is determined by the execution *Glossary*. Consequently, the *PreprocessorUnit* and the *MacroResolverUnit* interfaces must be implemented by the same component and are only split for explanation purposes. From the retrieved term, (ii) the associated template in the given language and dialect needs to be found; if there is no such template available, the *expr* method returns an error. Step three (iii) is to preprocess the template by replacing the <ctx> placeholders with the corresponding values of the arguments parameters; if <ctx> is a collection variable, the first argument is substituted for <ctx>[1], the second for <ctx>[2] respectively. Finally, the preprocessed template (iv) is returned to the calling component.

In terms of the *dimKey* method, corresponding to the *$dimKey* macro, the name of a *Dimension* can be passed, to get the name of the base level in return. To get the base level of a dimension, the procedure in Figure 4.32 must be followed.

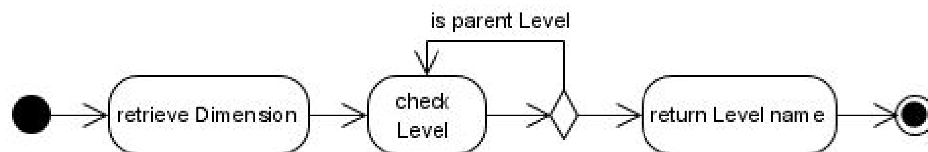


Figure 4.32: Procedure behind the dimKey method

Accordingly, the first action (i) is to retrieve the *Dimension* object specified by the *dimName* parameter. As for the *expr* method, the path for the dimension is determined by the execution context, in this case by the MDM. After retrieving the dimension (ii) a loop over all contained *Level* objects is performed to check whether it is a base level or not. That is, the *RollUpRelations* objects are inspected to check for each *Level* if they occur as a parent in any *RollUpRelation*. Lastly, if a *Level* is found (iii), which not a parent in any *RollUpRelation* its name is returned to the calling component.

4.3.6 Interface Provider

The *interface provider* supplies external components with an interface to use the functionality of the repository application. At the same time it allows, in the course of the ICR, to change between different interface implementations or technologies without influencing any other component.

The interface consists of one provided (*Command Interface*) and one required interface (*Language-ProcessingUnit*) as in Figure 4.33.

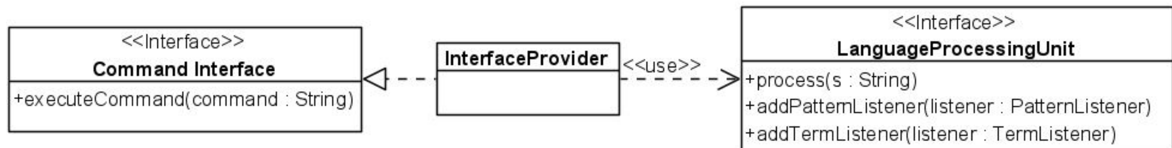


Figure 4.33: Interface provider's interface

The *Command Interface* consists of only one method (*executeCommand*) and allows for other components to pass a OLAP pattern language statement. The interface provider takes this command and passes it to language processor to analyze the statement using the *process* method; further, it provides the controller with the information about the sender, so that it can create and send an appropriate response messages.

4.4 Editor Application

The *editor application* provides a graphical user interface (GUI) to access the repository application and thus is the access point for DWH and domain experts to the OLAP pattern repository. According to the problem statement (section 1.2) it enables the user to write OLAP pattern language statements (Appendix A) to define and use OLAP patterns. However, this work does primarily focus on the repository application, which is why the editor is designed to be only a very basic text editor. That is, the user is provided with a simple text input field allowing to write statements, which does not offer any kind of support regarding the formulation of these statements. Further, the user is provided with a send button allowing to transfer the statement to the repository application for execution. Lastly, the the GUI provides a result field showing success and error messages as well as search and show results (object representations). Hence, beside the graphical UI the editor application provides, it also requires an interface to transmit statements to the repository application (see Figure 4.34).

Figure 4.34: Interface of the editor application

In this course, the editor uses the *Command Interface* which offers the *executeCommand* method, taking an OLAP pattern statement as a parameter and returns a response (success message, error message or object representations). The *Graphical UI* is however not determined by this work and can be implemented in any way so that possible users can benefit from.

Chapter 5

Implementation

This section introduces a prototypical implementation of the design (chapter 4), covering both the implementation of the editor and the repository application. It is accessible on both, the enclosed CD or the public repository which is available at BitBucket[1]. As detailed in the design section a client-server architecture is followed, that is, the repository application is implemented as a *Java* server application (version 1.8.0_231) that is accessed via the editor, which is implemented as an *Angular* web application (version 8.3.21). In Figure 5.1 an overview of execution environments is provided showing which components run in which environment. The following will give a brief introduction of the overall implementation technologies used before the choice and the implementation details are further discussed in separate sections per component.

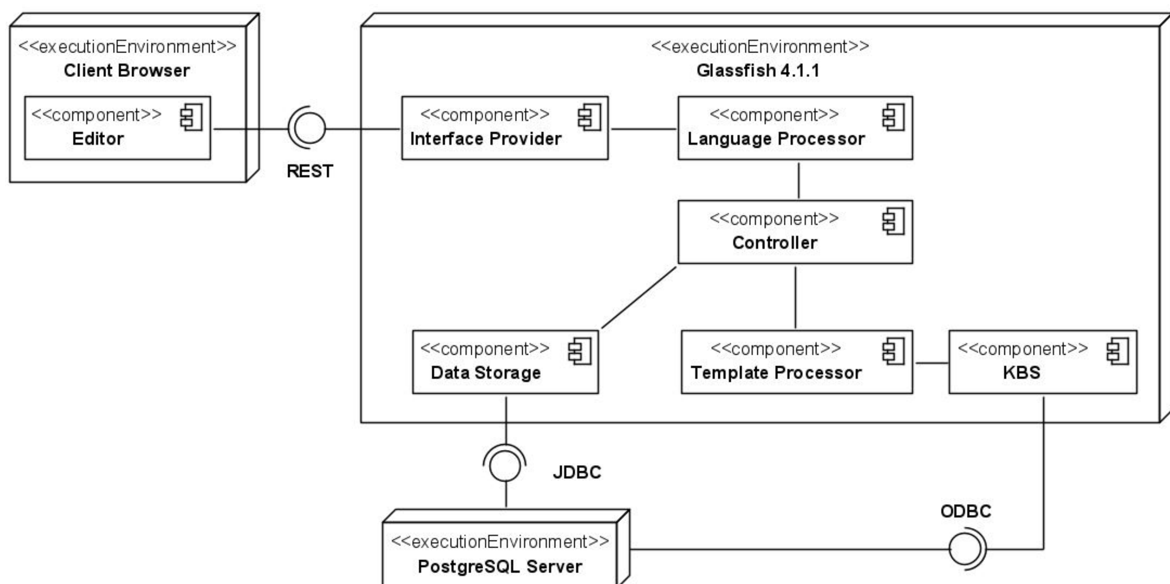


Figure 5.1: Deployment diagram

The server side is represented by two different execution environments, namely the *Application Server* and the *PostgeSQL Server*. The *Application Server* basically comprises the components as introduced in section 4.3 regarding the repository application (Figure 5.1); the description of the interfaces which are provided and consumed by the components are not shown here (details can be found in section 4.3). The *PostgreSQL Server* hosts a database allowing to persist both organization and content elements.

[1]https://bitbucket.org/mimo16/olappatternrepository/src/master/

To implement the logic for components on the application server – a *Glassfish 4.1.1*[2] server – the regular functionality of the Java programming language is extended by several software libraries. First of all, one needs a data format to transfer information between server and client; for this purpose the *JavaScript Object Notation* (JSON)[3], which is a common format to transfer data over a network, was chosen. Using the *Jackson Core* library (version 2.9.8)[4] allows to easily convert a regular Java object into the JSON notation and create a new object by specifying properties and their values. However, it should be pointed out that the data format to transfer information between the client and the server does not influence the overall functionality, which is why any other format would be equally appropriate. In order to process OLAP pattern language statements, the *ANTLR4*[5] parser generator library allows to generate both lexer and parser from an existing grammar. Accordingly, both the language processor and the template processor must only implement the semantic analysis. The last library to be added for the components in the application server is the *Hibernate* library (version 5.4.11)[6] allowing to map Java objects to relational database tables. That is, a relational database appears as if it contained objects rather than tuples.

The client-side is represented by the *Client Browser* execution environment, in which the Angular web application is running. In this context it is not important whether the editor is loaded from a server or locally stored on the client device. Regarding functionality, the standard framework provided by Angular is extenden by one library, namely the *ngx-json-viewer* (version 2.4.0)[7]. This library provides a component for graphical user interfaces, that allow for users to inspect JSON objects in form of drop-down lists.

The following sections go into detail on how these libraries are used to implement the components as intended by the design; the implementation of the components' behavior using standard java syntax is not discussed here. Section 5.1 describes implementation details for the repository application on the server-side, followed by section 5.2 introducing the editor implementation on the client-side.

5.1 Repository Application

The repository application is the central point for storing, maintaining and using OLAP patterns within an organization. It is implemented as a Java server application running on a *Glassfish 4.1.1* application server which can be accessed from distributed clients. Note that every other object oriented programming language allowing to implement server application is equally reasonable as well, however, Java provides a big community with numerous helpful and maintained libraries.

The *at.dke.olappattern* package contains the implementation of the repository application and is further divided into the four main packages *data*, *language*, *macro* and *repository_app* (Figure 5.2). That is, only the last package is specific to the repository application, whereas the previous three packages – regarding the data structure and classes for processing both grammars – are general purpose and can also be used for other implementations. The *data* package contains all classes

[2]https://www.oracle.com/middleware/technologies/glassfish-server.html
[3]https://www.json.org/json-en.html
[4]https://github.com/FasterXML/jackson
[5]https://www.antlr.org/
[6]https://hibernate.org/
[7]https://www.npmjs.com/package/ngx-json-viewer

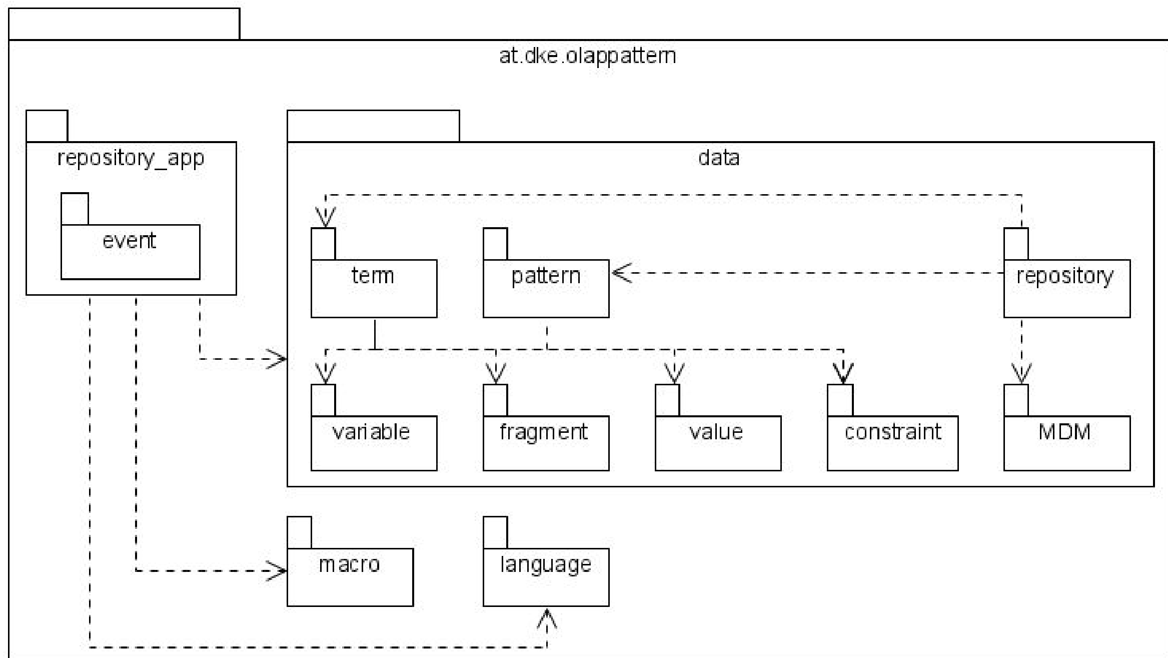


Figure 5.2: Repository application packages

that correspond to the data structure (section 4.2) and thus be used in other applications regarding OLAP patterns as well. In eight sub-packages the numerous classes are further organized; accordingly the*constraint* package contains the *Constraint* class and all available specializations. The *localcube*, *variable* and *value* packages contain the *LocalCube*, *Variable* and *Value* class hierarchies respectively and cover the remaining elements contained in patterns or terms. Both pattern and terms however are comprised by individual packages also including their descriptions, i.e., the *pattern* and *term* package, and are in dependency relationships with the variable, constraint, localcube and value packages. Besides that, the *MDM* package contains both the *MDMEntity* and *MDMProperty* classes. The repository sub-package groups organization elements and is consequently related to the *pattern*, *term* and *MDM* package.

The second main package included is the *language* package which comprises the necessary classes to implement language processing for OLAP pattern statements. That is, classes regarding lexer and parser, as well as the semantic analyzer for generating an object representation are included. It is worth noting, that the lexer and parser can also serve as basis for further semantic analyzers. The same holds true for the *macro* package which also includes lexer, parser and semantic analyzer, that provide functionality for parsing the macro island grammar. Finally, the *repository_app* package, contains all classes corresponding to the repository application's components as well as the interfaces they implement (introduced in section 4.3).

5.1.1 Language Processor

The language processor is responsible for analysing any OLAP pattern language statement which is sent to the repository application and thus provides the *LanguageProcessingUnit* interface. The *process* method contained therein, takes a string (the statement), generates a object representation, packs it into an event and notifies the registered observers (listeners). Therefore, the behavior described in subsection 4.3.2 must be implemented, i.e. the statement must be processes using a lexer, parser and semantic analyzer.

Even though manually implementing all three components is the most flexible way, it still is a time consuming task. Common parser generator on the other hand provide almost as flexible implementation scopes and at the same time enable a significantly faster development [21]. Accordingly, the *ANTLR4* (Another tool for language recognition) parser generator is used as a base for the language processor implementation. The choice for a parser generator is determined by the OLAP pattern language's format, the ANTLR4 specific *.g4* format and ANTLR4's popularity, gained through the use in well-known organizations like Twitter or in projects like Appache Hadoop. From a technological perspective any other parser generator can be used equivalently, which potentially requires a translation of the grammar into another format. To prevent potential errors and pitfalls during a transformation process, it is therefore most reasonable to stay with ANTLR4.

Based on a *.g4*-grammar-file, the ANTLR4 library allows to generate a number of classes and interfaces, which basically provide all the functionality for lexical and syntactic analysis for a corresponding language statement. The provided classes and interface are named after the grammar name as declared in the grammar file, which is *emdml* for the OLAP pattern language. Hence, the lexer and parser generated for the language processor are called *EMDMLLexer* and *EMDMLParser* respectively. To enable a semantic analysis of an AST generated by the *EMDMLParser*, a listener ([13, p. 326]) as well as a visitor pattern ([13, p. 366]) can be applied. Both provide methods, that are called whenever the respective rule is found during an iteration over the AST; the main difference between listener and visitor is basically, that the visitor methods are called only when a sub-tree is entered, whereas for the listener enter and exit methods inform when a sub-tree is entered and left respectively. For both purposes, the ANTLR4 parser generator provides an interface, namely *EMDMLListener* and *EMDMLVisitor*, as well as classes providing empty implementations for both interfaces called *EMDMLBaseListener* and *EMDMLBaseVisitor*. That is, one can decide to implement an interface (and therefore all its methods) or extend the respective base class (and therefore override only the methods that are actually needed) in the semantic analyzer. Considering the EXTR the listener is chosen over the visitor, as the combination of enter and exit methods provide more flexibility concerning future extensions. Further, the *SemanticAnalyzer* class overrides the EMDMLBaseListener so that only required methods must be implemented.

Based on the listener methods for the OLAP pattern language (Appendix A) the following introduces how the language processor is implemented to convert an example AST into an object representation, realizing the behavior introduced in the design (subsection 4.3.2). Therefore, the context of the Jersey business term is taken as example, however other elements are transformed similarly. Initially the *EMDMLLexer* and *EMDMLParser* automatically generate an AST as depicted in Figure 4.23, which is subsequently run through in a loop to visit all nodes. In Figure 5.3 the steps, i.e. the implemented methods, to semantically analyze the Jersey business term context and generate a corresponding *TermEvent* are visualized in form of a non-deterministic finite automaton (NEA).

Starting from the initial state q0, the first method to be implemented is the *enterT_def* listener method; it resets all internal variables of the *SemanticAnalyzer*, so that no previous states are carried forward and creates a new *Term* object. The result state q1 represents an empty *Term* containing no information. The *enterT_type* method is called once the corresponding node is visited in the AST, that is, it initiates the transition to state q2. The method is implemented to add a *Type* object to the term and to create the corresponding context variable(s) with

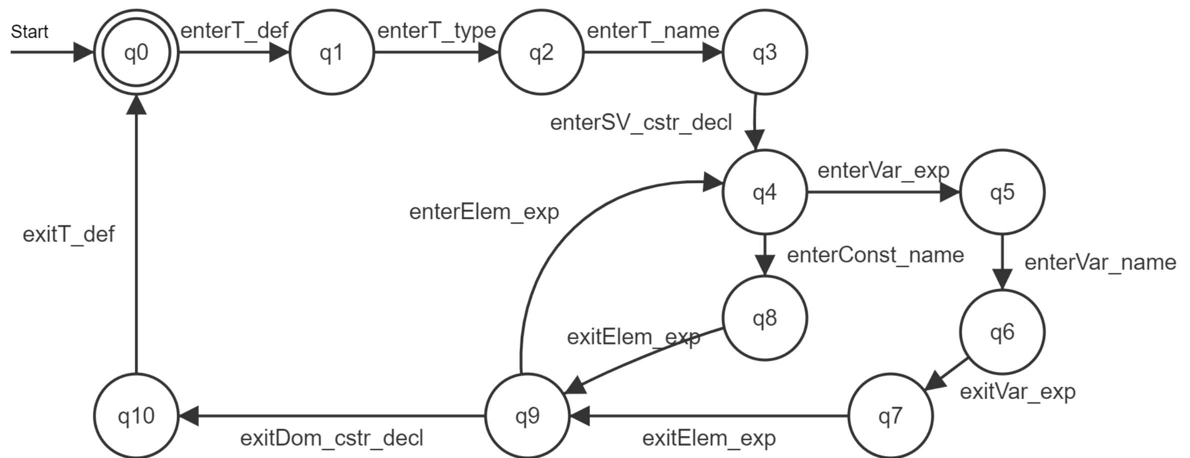


Figure 5.3: Semantic analysis steps for the "Jersey Breed" business term

their type declaration (type constraints). For the Jersey business term a *SingleVariable* named <ctx> and a *TypeConstraint* restricting it to **"DIMENSION"** are added. Subsequently, the *enterT_name* method allows to react to the term's name node in the AST, containing the glossary path and the actual term name. In the case of the Jersey term the path is split up into the actual term name (**"Jersey"**), which is added to the *Term*, and the glossary path (**"Austrian Milk Models"/"Dairy Business Terms"**) which is stored as path of the *TermEvent* respectively. The next methods processes constraint definitions. In the case of the Jersey term statement (Listing 3.7) a domain constraint, restricting the <ctx> parameter has to be considered. Following the grammar, a domain constraint consists of three element expressions (*elem_expr*), which are either variables (*var_exp*) or constants (const_name); in terms of the data structure (section 4.2) an element expression refers to a *Target*, which is also either a *Value* or *Variable*. As this structure is similar for all other constraints as well, the implementation of the *enterSv_cstr_decl* creates an empty list of *Targets* – *tempTargets* – allowing to collect the information regarding one constraint. Accordingly, from state q4 both a *enterVar_exp* or an *enterConst_name* call are valid to move on to state q5 or q6, respectively. In case of a constant, the *enterConst_name* method creates a new *SingleValue* and stores it internally in order to add it to the *Term*'s targets and to the *tempTargets* in the *exitElem_expr* to transit to q9. In case of a variable, the *enterVar_exp* method creates a new instance of the *VariableBuilder* class (see Figure 5.4), i.e., a class that allows to aggregate the information of a variable to finally create an appropriate *Variable* object.

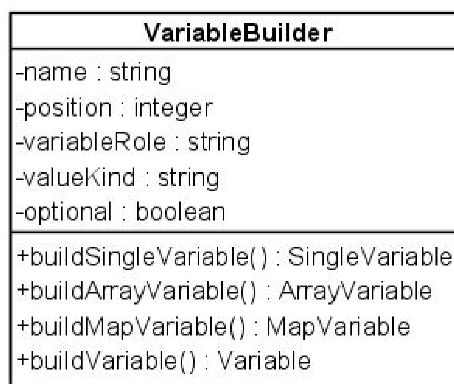


Figure 5.4: VariableBuilder class diagram

All listener methods called between *enterVar_exp* and *exitVar_exp* add information to the *VariableBuilder*, as for example *enterVar_name*. The reason for using a *VariableBuilder* is that the type of a collection variable should be determined by a symbol like **"*"** or **"[ ]"** at the end of the variable expression, i.e., indicating maps and arrays. As a result, it is not possible to create a *Variable* object before the end of the definition, which raises a demand to aggregate the information meanwhile. Once the variable expression is analyzed (in q6) the subsequent *exitElem_expr* call creates a *Variable* object and adds it to the *tempTargets* list. Considering the Jersey business term, the *tempTargets* therefore contain three *Targets*, a SingleVariable (<ctx>), and two *SingleValues* (**"Main Breed"** and **"Breed Name"**), before the *exitDom_cstr_decl* is called (in q9),. The last step of processing a domain constraint statement is the call of the *exitDom_cstr_decl* method, which creates a *DomainConstraint* object. Therefore, it is checked if the *Variable* included in the *tempTargets* list actually occur in the *Term*; that is, only previously defined variables can be used in constraints. If the check concludes that all variables are valid, a *DomainConstraint* is created, where the first target is set to be the entity, the second to be the property and the third to be the domain. State q10 therefore represents the *Term* object as depicted in Figure 4.24, i.e. the final representation of the statement. The *SemanticAnalyzer* gets back into the initial state after the *exitT_def* listener method was called. In this final method, the listeners are informed about the result, that is, a new *TermEvent* is created, comprising the *Term*, the path and an *Action* set to CREATE. This *TermEvent* is than passed to the *TermListeners* by calling their *processTerm* methods.

5.1.2 Data Storage

The data storage is the component that allows to save OLAP pattern related data in a database. It is part of the repository application which is hosted on the application server execution environment and provides the *StorageUnit* interface which allows for other components to work with object representations only. That is, the data storage communicates with the database, but hides that communication from other components. The database behind the data storage, however, is not part of the repository application instead it is located on a stand-alone *PostgreSQL* server. Hence, the implementation is based on a relational data model provided by a *PostgreSQL* database. Even though the design would allow to use any kind of database, the sophisticated, free and open-source tool support for relational databases makes it a reasonable choice for a first prototype. There are many relational databases available, that would all be equally sufficient for what is implemented in here. However, for the implementation of the KBS, even though not provided by this thesis, Datalog is deemed to be used in future extensions. Given these considerations, *PostgreSQL* databases enable the easiest integration of datalog for the future development of the repository application.

Independent of the database technology used, the data storage component is responsible for encapsulating the actual database accesses and enable the other components to work with objects instead. This means e.g. whenever a *getPattern* method is called, the data storage must query the database to retrieve the contained information regarding context, templates and descriptions, generate an object representation and return it to the caller. Consequently, for every persist call, the data storage needs to write the information contained in the objects into the respective tables and columns and, in terms of e.g. a pattern, reference the catalogue it should be contained in. For

these purposes the concept of object-relational mapping (ORM) was introduced, i.e. a database schema is mapped to a class schema [22]. The first implementation of ORM was introduced with the Java library *Hibernate*, which is available on an open source licence [22]. For that reason and because of its comprehensive documentation, *Hibernate* (in version 5.4.11) is the selected ORM library for the implementation of this thesis.

Basically, Hibernate allows two different mapping forms, i.e. XML mapping or mapping by annotations [22]. Whereas with annotations, the mapping is included in the respective class files, XML mapping allows to separate the mapping information into an XML file. For that reason, and for the fact that XML mapping is slightly more flexible, it is chosen over the maybe more prominent annotations mapping. Further, one can argue that using annotations violates the ICR requirement, as it entangles the data structure with the database in the background. An excerpt of the XML mapping regarding patterns can be found in Listing 5.1.

```
1  <hibernate-mapping>
2      <class name="at.dke.olappattern.data.term.Term" table="term">
3          <id name="id" type="integer" column="id">
4              <generator class="identity"></generator>
5          </id>
6
7          <property name="name" type="string" column="name"/>
8          <property name="returnType" type="string" column="return_type"/>
9
10         <many-to-one name="type"
               class="at.dke.olappattern.data.term.Type" column="type"
               cascade="save-update" not-null="true"
11                       fetch="select" lazy="false"/>
12
13         <set name="templates" table="term_template" cascade="all"
               lazy="false" fetch="select">
14             <key column="term" not-null="true"/>
15             <one-to-many entity-name="TermTemplate"/>
16         </set>
17
18   </class>
19
20 </hibernate-mapping>
```

Listing 5.1: Excerpt of the pattern mapping

Mappings are provided within the *<hibernate-mapping>* tag, which further contains the classes identified by the *<class>* tag. For every class, the name of the java class and the table it should be mapped to are stated; this means the *Term* class is mapped to the *term* table in this example. Thereafter, the properties comprised can be mapped by stating the name (name of the java property), a column (name of the database column) and the data type. That is, the java property identified by the name is persisted in the defined column. The id for the database table is mapped using the *<id>* tag and stating both name and column; additionally, a generator can be provided, enabling Hibernate to automatically create id values. Besides simple properties, many to one relationships can be mapped using the equally named tag. Therefore, name and column are stated,

with the data type set to the class which is related; for this example a *Term* is associated with a type. Furthermore, one can define additional properties to specify the relationship; in the example the type is not-null (terms must have a type), with cascade set to "save-update" (there is no cascading delete). Defining fetch type "select" and lazy as false, ensures that the type is always loaded from the database once the *Term* is loaded.

For the mapped classes, the Hibernate framework provides methods to persist and load objects. Therefore, it allows to open sessions to communicate with the database and provides means to handle transactions as depicted in Figure 5.5.

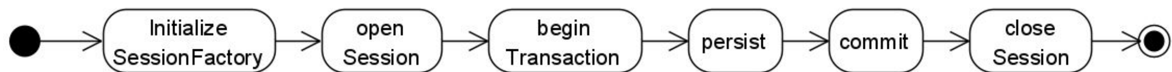


Figure 5.5: Process for persisting objects using Hibernate

Accordingly, before an object can be persisted, (i) a *SessionFactory* must be initialized, which means that the mappings must be loaded. The *SessionFactory* object provides an *openSession* method (ii) that must be called to open a database session, represented by a *Session* object returned. Once a session is open, (iii) a new transaction can be started using the *Session*'s *beginTransaction* method; in case of an error, these transactions provide a rollback function to automatically reverse any changes. Within the transaction (represented by a *Transaction* object), one has the choice between multiple operations provided by the *Session* object, that allow to persist an object in the database. For this work the *saveOrUpdate* method (iv) is deemed sufficient as it automatically decides whether an insert or update operation must be performed on the database. This means, by just calling this method and passing the object to be persisted, Hibernate does create all necessary SQL statements. Subsequently, (v) the changes can be committed using the *commit* method provided by the *Transaction*. Lastly, (vi) the session must be closed again.

The same procedure applies to all delete methods as well, except that Hibernate's *delete* method is called instead. There is however one exception where the automatically created delete statements cause problems, namely for context classes (patterns and terms). As both variables and values are included in a context's targets, Hibernate does not recognize, that some variables reference values. Due to that fact the values can only be delete once they are not referenced by variable any more; values and variables must be deleted manually to solve these issues.

Lastly, for retrieving objects, hibernate allows to formulate SQL-like queries; these queries contain the names of the mapped classes and their properties rather than table names and comprised columns. Hibernate queries are however just as powerful as regular SQL queries and allow to use SQL functions, like *avg*, *max* etc., as well.

5.1.3 Template Processor

The template processor, responsible for the transformation of templates to executable OLAP queries, is part of the java components hosted on the application server. It provides the *PreprocessingUnit* interface comprising the *executePattern* method which executes one or more pattern templates. Therefore, the template processor loads the MDM and glossary providing the context, as well as the pattern, from the data storage to subsequently process the pattern's template(s). For analysing the templates language processing based on lexical, syntactical and semantic analysis must be

performed. In terms of the implementation it therefore builds upon the ANTLR4 library (like the language processor) to process the templates underlying the island grammar (Appendix B). Hence, given the grammar notation in the *".g4"* format, a lexer and a parser are automatically generated. The *SemanticMacroAnalyzer* builds upon these components and overrides the corresponding *MacroBaseListener* to implement all methods necessary to substitute macro calls during the iteration of an AST.

Before macros can be substituted the Pattern, MDM and glossary are loaded, the stub methods for grounding and validation are called, and the templates to be executed are determined using the *getTemplates* method of the data storage. Subsequently, an internal method, i.e. *processTemplates*, iterates over all templates and prepares them for the language processing steps. Therefore it passes every template to another internal method (*substituteTemplatePlaceholders*) that substitutes the variable placeholders. In the next step the *processTemplates* method initiates the language processing steps for the templates. Considering the example AST in Figure 4.30, the *SemanticMacroAnalyzer* implements the processing steps in Figure 5.6.

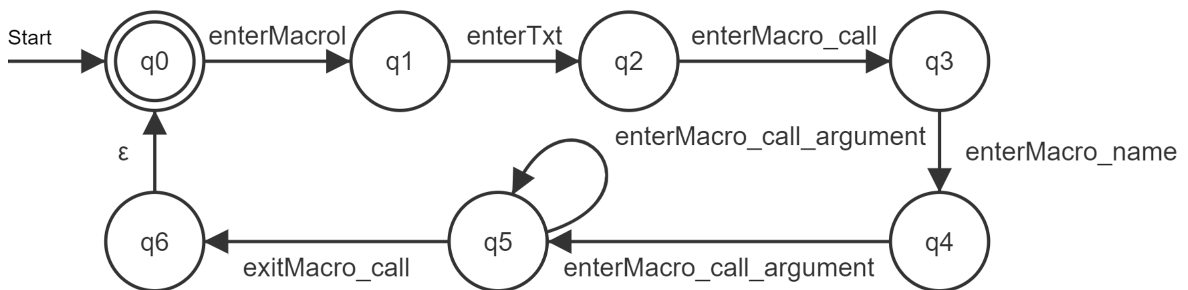


Figure 5.6: Semantic analysis steps for the template expression in Figure 4.30

Initially the *enterMacrol* listener method is implemented, which sets the only public property of the *SemanticMacroAnalyzer*, namely the *query* property, to be an empty string. To get from state q1 to q2, the *enterTxt* method is implemented, which removes the surrounding "*{" and "}*" symbols from the query text and adds it to the *query* string. Subsequently, for the transition to q3 the *enterMacro_call* method creates a new list (*macroArguments*) allowing to store the parameters of the macro call. The following *enterMacro_name* sets an internal enumeration allowing to store the macro to be called; i.e. the macro enumeration is set to *EXPR*. In every of the following enterMacro_argument calls, the string argument is added to *macroArguments* list; for the example expression the arguments are **"Jersey"** and **s**. Lastly, in the *exitMacro_call* method, the respective macro method as provided by the *MacroResolver* interface is executed. That is, for the example expression, the *expr* method for the Jersey term is called with **s** as value for the <ctx> parameter; the result of the *expr* method is consequently added to the *query* parameter. Accordingly, in q6 the *query* parameter includes the executable OLAP query.

If more than one template is to be executed, the template processor creates aggregates the results of the *SemanticMacroAnalyzer* in form of a single string with empty lines between the queries. Finally, the string containing the query/queries is returned to the controller, which generates and sends a response to the editor.

5.1.4 Controller

The controller, also running within the application server environment, is a java component that orchestrates the functionality of the data storage and template processor and thus depicts the business logic. In terms of implementation it therefore realizes the methods and the underlying logic as introduced in subsection 4.3.1; as this realization does not differ from the initial design, it is not covered again in this section. Besides that, the controller is responsible for creating a response for the editor application, which can be transmitted over a network connection. That is, the success and error messages, as well as the result lists of search operations are converted into a JSON object. Using the *Jackson Core* library one therefore is provided with the *JsonObjectBuilder* class. An instance of this class allows to add properties to it, by stating the property name and and its value; finally one can call *build* to receive a JSON object (see Listing 5.2). For a resulting JSON object of a catalogue, refer to Appendix C.

```
1 JsonObjectBuilder job = Json.createObjectBuilder();
2 job.add("result", str);
3 JsonObject j = job.build();
```

Listing 5.2: JsonObjectBuilder usage

The structure of the response objects created by the controller is always the same and includes only one property, namely the *result*. In Listing 5.2 the string (*str*) value added to the result property therefore may be an error or success message or a previously generated JSON string representing a search result, i.e. organization or content elements.

```
1 response.resume(Response.ok().entity(result).
2                 header("Access-Control-Allow-Origin", "*")
3                 .header("Access-Control-Allow-Credentials", "true")
4                 .header("Access-Control-Allow-Headers",
5                         "origin, content-type, accept")
6                 .header("Access-Control-Allow-Methods","POST").build());
```

Listing 5.3: AsyncResponse for responding to the editor

To be able to respond to the editor application, the controller needs information about the sender. This information is provided by the *InterfaceProvider*, which generates an *AsyncResponse* object (see Listing 5.3). An *AsyncResponse* provides a resume function, which takes a *Response* object as a result and transmits it to the sender; the *Response* corresponds to a HTTP response and includes a status, headers and in this case the created JSON object. The header settings in Listing 5.3 correspond to the openness requirement defined by Shahzad et al. [16] and allow access for all applications.

5.1.5 Interface Provider

The implementation of the *interface provider* provides a REST interface for the repository application, that allows for the editor to send OLAP pattern language commands using HTTP post messages. The interface provider, just like all other components of the repository application, corresponds to a java class hosted on the application server. It allows to send JSON objects to the

server using the following resource: *url/repository*; the *url* depends on the address of the server it is hosted on. The command must therefore be packed into a JSON object with one property, namely *command*, which is associated with the OLAP pattern statement.

For implementing such an REST interface, the Java API for RESTful Web Services (JAX-RS) is version 2.1 is used. A JAX-RS Application, must include one class overriding the provided *Application* super class, which is the *RepositoryApplication* in Listing 5.4, that can be hosted on a server – the glassfish server in this case. This class does not contain any logic but states which resources the user can access, i.e. the */repository* resource hidden behind the interface provider.

```
1  @ApplicationPath("/")
2  //The java class declares root resource and provider classes
3  public class RepositoryApplication extends Application{
4      @Override
5      public Set<Class<?>> getClasses() {
6          HashSet h = new HashSet<Class<?>>();
7          h.add( IntefaceProvider.class );
8          return h;
9      }
10 }
```

Listing 5.4: RepositoryApplication class

The *InterfaceProvider* class in contrast contains the *executeCommand* method (Listing 5.5) for processing statements; in terms of JAX-RS it must be annotated with *@Path("/repository")* to declare the path. As determined by the design, the *executeCommand* method extracts the OLAP pattern statement from the HTTP request. Subsequently, it provides the controller with an *AsyncResponse* object enabling it to respond to the editor, and passes the statement to the language processor.

```
1  @POST
2  @Produces(MediaType.APPLICATION_JSON)
3  @Consumes(MediaType.APPLICATION_JSON)
4  public void executeCommand(final JsonObject operation, @Suspended final
       AsyncResponse response) {
5    String command = operation.getString("command");
6
7    controller.setResponse(response);
8    try{
9      Initializer.languageProcessingUnit.process(command);
10   }
11   catch (Exception e){
12     Initializer.controller.respond(e.getMessage());
13   }
14 }
```

Listing 5.5: execute command

To signalise JAX-RS that *executeCommand* is responsible for handling post request for */repository*, one needs to annotate it with *@POST*. Additionally, by adding *@Consumes* and *@Produces*, it can

be signalized to the clients, which formats the information in the request and the response must have.

5.2 Editor Application

The editor application corresponds to the client of the OLAP pattern repository and is implemented as an *Angular* web application. Consequently, it is executed in the client device's browser, whereby almost every device can be considered a possible client. The Figure 5.7 gives an impression on the editors graphical user interface (GUI).

Figure 5.7: GUI of the editor application

Accordingly, the editor GUI consists of three elements, a text field to enter commands, a send button and a response field. The send button generates a JSON object in the form *"command" : "SEARCH ..."*, i.e it includes a command property, which is associated with the statement in the text field, and sends it to the repository application using a HTTP post message. Additionally, it registers a listener method which visualizes the response, as generated by the repository application's controller, in the response field. Therefore it is distinguished between message- (success and error messages) and object responses; whereas messages are shown using the HTML *label* element, objects are visualized using the *ngx-json-viewer*, the latter allows to view JSON objects in form of drop-down list as in Figure 5.7 That is, for the SERACH statement in the command field, one repository object was returned, with the id 1 and the name **"Austrian Milk Models"**. By clicking on the catalogues, glossaries or mdm arrays, one would further be able to see more details and even the comprised content elements.

Chapter **6**

Evaluation

In this chapter the implementation (chapter 5) as well as the design (chapter 4) is evaluated by checking whether general repository requirements (Table 2.4) and system- (subsection 3.2.2) and user-specific (subsection 3.2.1) requirements are met. Therefore, in section 6.1 it is detailed how the requirements following Shahzad et al. [16] are fulfilled. Subsequently, in section 6.2 it is shown how the system requirements are met by the application design. Finally, in section 6.3 it is demonstrated how the user requirements are met, that is, how users are supported by the OLAP pattern repository in their practical work.

6.1 Repository Requirements

This section evaluates whether the requirements as introduced by Shahzad et al. [16] are met by the OLAP pattern repository. Table 6.1 provides an overview of the general repository requirements with the degree of realization, i.e., whether the requirement is implemented exhaustively, partially, or not at all.

Table 6.1: Evaluation of general repository requirements following Shahzad et al. [16]

#	Requirement	Realization
1	Extensible	exhaustive
2	Flexible	partly
3	Openness	exhaustive
4	Accepting	partly
5	Usable	partly
6	Navigable	exhaustive

First, the OLAP pattern repository is *extensible* considering organisation as well as content elements. By supporting `CREATE` statements it is possible to add new organization structures and to insert content into existing ones. Second, *flexibility* is partly provided as OLAP patterns can be instantiated yielding different versions (see also the organization of patterns in subsection 2.2.3). Even though considered by design, the implementation does not allow to instantiate business terms in the current version, consequently, the degree of realization is classified only as partially. Third, there is no differentiation between users or groups of users, that is, the implementation provides sufficient *openness*. In addition, openness is also fostered by providing a REST interface, allowing any application, other than the editor, to access the repository application's functionality as well.

Fourth, the requirement concerning *accepting* can be met only partially. While it is possible to structure the repository's content (using organization elements), the users may only chose from a set of predefined organization elements (repositories, MDMs, catalogues or glossaries). That is, users cannot define custom structures even though it might be more beneficial to them. Fifth, this work focuses on the provision of all necessary functionality to realize the OLAP pattern language statements and not on providing a sophisticated user interface, consequently, the degree of *usability* realization can only be claimed partially. Although the OLAP pattern repository does provide a GUI as suggested by Shahzad et al. [16] a future work must determine how the usability is considered by practitioners. Finally, the OLAP pattern repository is *navigable*, as the implementation of `SEARCH` and `SHOW` statements allow for users to find specific elements as well as get an impression of which content is comprised by organisation elements.

6.2 System Requirements

In this section the system requirements are evaluated by taking the design and the implementation into account. It is discussed how certain requirements are covered by the design respectively the implementation of the repository application.

The design is based on a component-based architecture to meet the interchangeability requirement (ICR). Interfaces are defined grouping a set of functionality that one component has to implement in order to provide it to other components. In turn, components using an interface only need to know the comprised methods, their parameters and results, but not how it is implemented nor which component implements it. This enables the replacement of components by other components that implement the same interface.

In addition, the design defines classes organized in hierarchies that represent the data models (section 4.2) of organization and content elements. The object representation requirement (OOR) is met as these classes follow an object-orientated notion.

The data models already take into account complex and collection types, even though, the prototypical implementation does not rely on them. This, however, allows to easily extend the current implementation by collection variables as described by Kovacic et al. [1], which meets the extensibility requirement (EXTR). It should be noted that these extensions are considered in the logical schema of the database as well. The OLAP pattern language (Appendix A) also already contains all necessary rules concerning these extension elements. Only the *SemanticAnalyzer* must be adapted to fully integrate the extensions into the repository application. By using the ANTLR4 parser generator, the remaining task would be to simply override the corresponding listener methods.

Finally, language statement processing requirement (LSPR) is fulfilled by the language processor and the template processor as both components perform all three steps necessary to process language statements. This means, they perform a lexical and syntactic analysis to generate and AST from an input language statement; based on the AST both components perform a semantic analysis. According to their tasks the results of the language processing differs, i.e. the language processor generates an object representation of the statement, whereas the template processor substitutes macro calls.

6.3 User Requirements

In this section it is demonstrated how the user requirements are fulfilled by the OLAP pattern repository application. Hence, the focus is especially set on how data warehouse (DWH) and domain experts are supported in the definition and usage of patterns. To this end the steps of a DWH expert to create a catalogue, to add a pattern, and to search for a specific pattern and for a domain expert the steps of instantiating and executing a specific pattern are demonstrated in the following section. For each step the input statement, e.g., the OLAP pattern language statement defined in the Listings of section 3.1, and the output messages, e.g., success messages, executable queries, object representations, are depicted. Each statement is entered in the "Command" text field and is sent to the repository application using the below "Send" button (see Figure 5.7). The output is depicted in the text area below the "Send" button, which visualizes the responses received from the repository application. The running example introduced in section 1.3 is used to demonstrate the implementation.

Firstly, a new repository structure must be created by the DWH expert using the command in Figure 6.1. Thereby, a new organization structure is initialized, allowing to depict the eMDM of the *Austrian Milk Company* in the subsequent steps. The **"Repository created!"** message confirms the success of the operation; it therefore covers the requirement CRR.

Figure 6.1: Create "Austrian Milk Models" repository

Within the repository structure, the *Austrian Milk Company's* MDM must be depicted. Hence, the statement in Figure 6.2 is executed by the DWH expert, creating a new, empty MDM following Listing 3.1 line 4. The repository application returns **"MDM created!"** which assures that the new MDM is persisted in the repository structure. That is, the CMR is fulfilled.

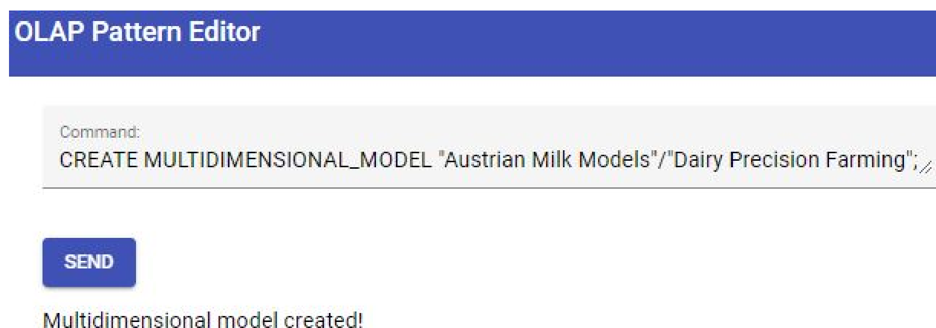


Figure 6.2: Create "Dairy Precision Farming" MDM

Inside the MDM, in a first step the DWH expert needs to create the dimensions, which can further be used in the cube definitions. That is to create the **"Time"** dimension as described in Listing 3.5 the DWH expert must execute the corresponding statement (see Figure 6.3). **"Dimension created!"** is visualized in the response field once the dimension is created fulfilling the CDR.

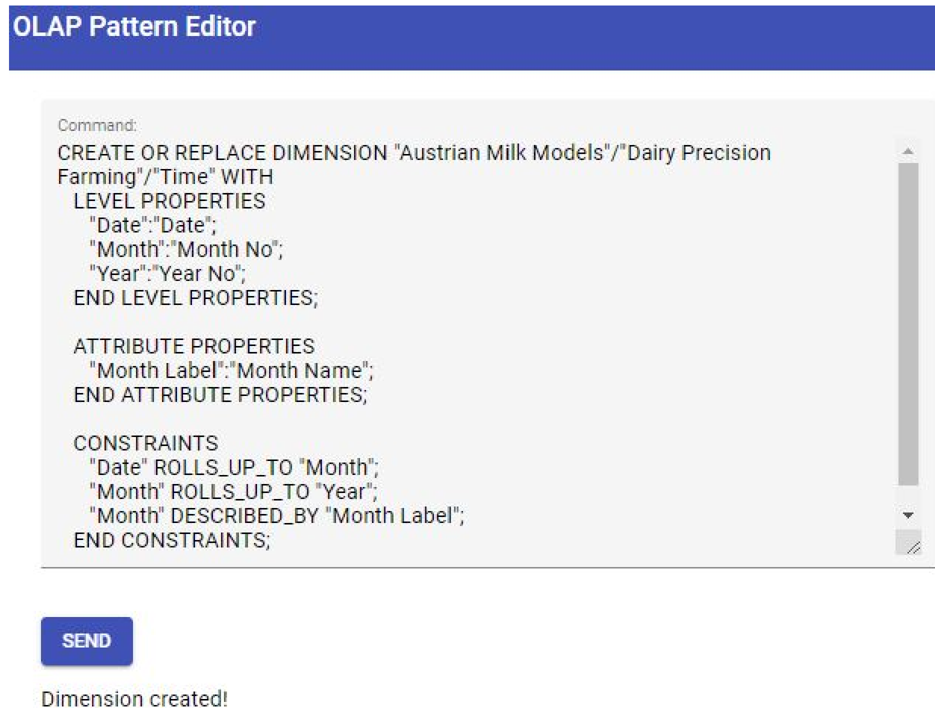


Figure 6.3: Create "Time" dimension

Based on the **"Time"**- and other dimensions, the DWH expert can define the **"Feeding"** cube (see Listing 3.6) as depicted in Figure 6.4. Therefore, the requirement regarding the creation of cubes, namely the CCR is fulfilled.

Figure 6.4: Create "Feeding" cube

Once the cubes and dimensions are comprised in the MDM, the domain's vocabulary can be depicted within a glossary. Accordingly, the statement to create a new glossary (see Listing 3.1 line 3) must be executed as in Figure 6.5. The OLAP pattern repository therefore meets the CGR.

Figure 6.5: Create "Dairy Business Terms" glossary

Subsequently, the DWH expert is able to add business terms to the glossary. Thus, in a first step a context must be added; for the *Austrian Milk Company*, the Jersey business term is deemed important (see Listing 3.7) and used as an example here. The DWH expert therefore executes the command as depicted in Figure 6.6. **"Term created!"** states, that the term's context was successfully created; in consequence, the CBCR is fulfilled.

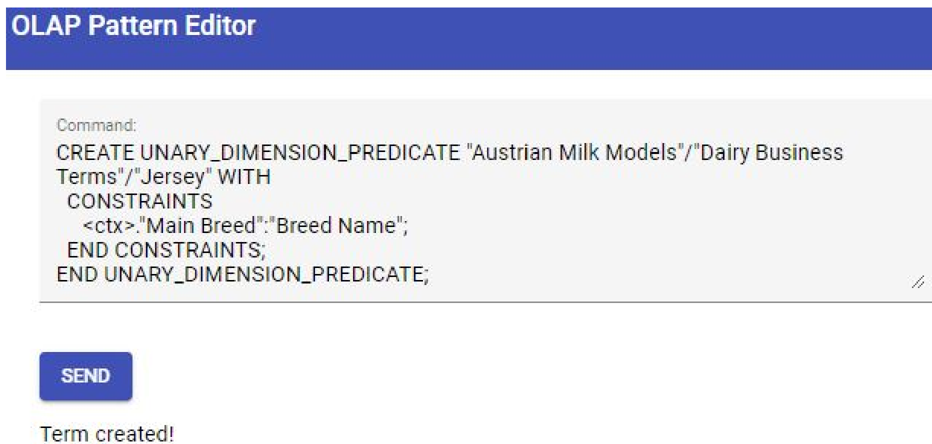


Figure 6.6: Create "Jersey" business term context

Thereafter a template for the business term must be added; for the Jersey breed one can see a SQL template in Listing 3.9. Accordingly, the DWH expert uses the corresponding statement (see Figure 6.7). This means, that the OLAP pattern repository also meets the CBTR.

Figure 6.7: Create "Jersey" business term template

Lastly, the Jersey business term must be completed by adding a description, as depicted in Listing 3.8. That is, the DWH expert formulates a description adding information to the business term context and template (see Figure 6.7). Supporting this functionality, the CBDR is fulfilled, furhter the combination of the CBCR, CBTR and CBDR provides the functionality as described in CBR.

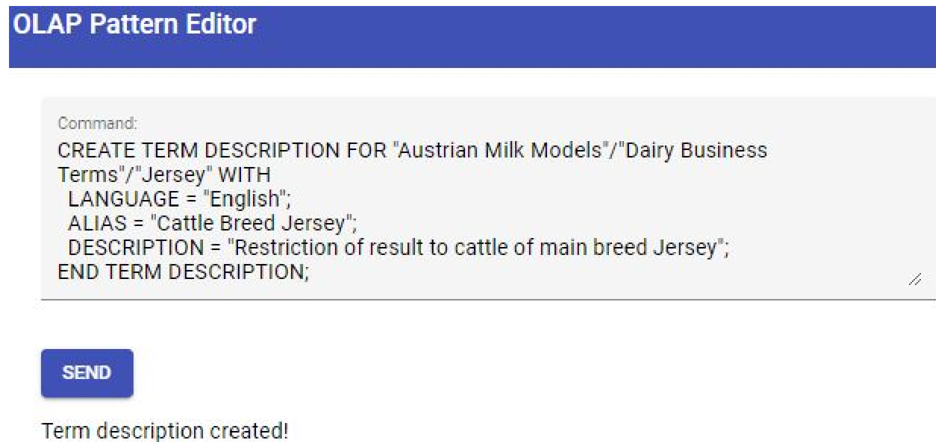


Figure 6.8: Create "Jersey" business term description

From this point on, it is assumed that the entire eMDM (see section 2.1) of the *Austrian Milk Company* is depicted within the repository application. Therefore, the DWH expert can start to define reasonable OLAP patterns. The first step of the DWH expert to be considered is the creation of a new catalogue named **"Diary OLAP Patterns"** in the **"Austrian Milk Models"** repository. Before creating a catalogue, the DWH expert, however checks whether the catalogue to be created already exists. The search statement in Listing 3.3 is therefore executed (satisfying the search organization element requirement (SOER)). As shown in Figure 6.9 no catalogue exists containing this search term as its name, which is indicated by the returned message **"No entries found for the given parameters"**.

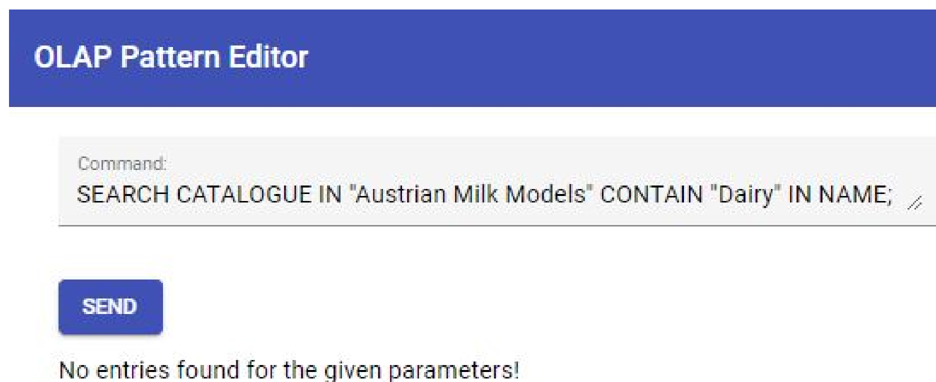


Figure 6.9: Search for catalogues

Once it is assured, that the catalogue does not already exist, the DWH expert creates the catalogue by executing the statement in Listing 3.1 line 2 (Figure 6.10). A **"Catalogue created!"** message is returned signalizing that the statement is syntactically correct and the operation is successfully executed (satisfying the create catalogue requirement (CCR)).

After the catalogue is successfully created, patterns can be added. The DWH expert has identified a new recurring type of information demand and an OLAP query composition strategy to satisfy

OLAP Pattern Editor

Command:

```
CREATE CATALOGUE "Austrian Milk Models"/"Dairy OLAP Patterns";
```

SEND

Catalogue created!

Figure 6.10: Create new catalogue "Dairy OLAP Patterns"

it, leading to the **"Breed-Specific Subset-Subset Comparison"**. The statement to create the **"Breed-Specific Subset-Subset Comparison"** and add it to the previously defined catalogue is depicted in Listing 3.10. The returned message **"Pattern created!"** in Figure 6.11 indicates that the pattern was successfully added to the newly created catalogue (satisfying the create pattern context requirement (CPCR)). It should be noted that this functionality also allows to fulfill the update content element requirement (UCER), as an update corresponds to a re-creation of a content element.

OLAP Pattern Editor

Command:

```
CREATE PATTERN "Austrian Milk Models"/"Dairy OLAP Patterns"/"Breed-
Specific Subset-Subset Comparison" WITH

    PARAMETERS
      <sourceCube>:CUBE;
      <baseCubeSlice>:UNARY_CUBE_PREDICATE;
      <animalBreedSlice>:UNARY_DIMENSION_PREDICATE;
      <compDimRole>:DIMENSION_ROLE;
      <iDimSlice>:UNARY_DIMENSION_PREDICATE;
      <cDimSlice>:UNARY_DIMENSION_PREDICATE;
```

SEND

Pattern created!

Figure 6.11: Create new pattern "Breed-Specific Subset-Subset Comparison"

To complete the **"Breed-Specific Subset-Subset Comparison"** pattern definition, the DWH experts subsequently adds a template, which is depicted in Figure 6.12. For the sake of brevity, only the first part of the template's expression in Figure 6.11 is shown, for the full expression refer to Appendix D. The returned message **"Pattern template created!"** in Figure 6.11 informs that the pattern is successfully added (satisfying the create pattern template requirement (CPTR)).

Subsequently, the DWH expert formulates an English problem description and a solution allowing for domain experts to judge the pattern's applicability to a certain problem. Therefore, the description statement (Listing 3.11) is entered in the command field (see Figure 6.13), sent to the repository application which acknowledges with the **"Pattern description created!"** message (satisfying the create pattern description requirement (CPDR)). This completes the definition of

OLAP Pattern Editor

Command:

```
CREATE PATTERN TEMPLATE FOR "Austrian Milk Models"/"Dairy OLAP
Patterns"/"Breed-Specific Subset-Subset Comparison" WITH
 LANGUAGE = "SQL" ;
 DIALECT = "ORACLEv11" ;
 EXPRESSION = "
*{ WITH baseCube AS (
  SELECT *
  FROM   <sourceCube> sc JOIN
       ""Animal"" a ON sc.""Cattle"" = a.}*$dimKey(""Animal"")*{
 WHERE  }* $expr(<baseCubeSlice>, sc) *{ AND }*
```

SEND

Pattern template created!

Figure 6.12: Add template to a pattern

the "Breed-Specific Subset-Subset Comparison" pattern (satisfying the create pattern repository requirement (CPR)).

OLAP Pattern Editor

Command:

```
CREATE PATTERN DESCRIPTION FOR "Austrian Milk Models"/"Dairy OLAP
Patterns"/"Breed-Specific Subset-Subset Comparison" WITH
 LANGUAGE = "English";
 ALIAS = "Breed-Specific Comparison";
 PROBLEM = "Aggregated measure values for two specified groups of
facts relating to a specific breed from a single source cube should be
compared in a meaningful way.";
 SOLUTION = "<table style=""width: 615px;"">
<tbody>
<tr>
```

SEND

Pattern description created!

Figure 6.13: Add description to a pattern

After the "Breed-Specific Subset-Subset Comparison" pattern has been defined, it can be used. A domain expert uses a pattern need to be adapted to the his/her current analysis situation. Therefore, names of eMDM elements bound to all pattern parameters, i.e., parameters are bound to the names of the cubes, dimensions, and business terms contained in the **"Dairy Precision Farming"** MDM and the **"Dairy Business Terms"** glossary. The domain expert's analysis situation is depicted by the instantiation statement in Listing 3.17. The pattern thus created should be stored in the **"Dairy OLAP Patterns"** catalogue of the **"Austrian Milk Models"** repository, under the name **"Austrian Milk Custom Breed-Specific Subset-Subset Comparison"**. The success of the instantiation is confirmed by the **"Pattern instantiated!"** message (Figure 6.14), which

also confirms that the instantiate pattern requirement (IPR) is successfully implemented by the OLAP pattern repository.

OLAP Pattern Editor

Command:

```
INSTANTIATE PATTERN "Austrian Milk Models"/"Dairy OLAP
Patterns"/"Breed-Specific Subset-Subset Comparison" AS
    "Austrian Milk Models"/"Dairy OLAP Patterns"/"Austrian Milk Custom
Breed-Specific Subset-Subset Comparison" WITH
 BINDINGS
   <sourceCube> = "Feeding",
     <animalBreedSlice> = "Jersey",
   <baseCubeSlice> = "Low Daily Feed Consumption",
   <compDimRole> = "Cattle",
   <iDimSlice> = "Young Cattle",
```

SEND

Pattern instantiated!

Figure 6.14: Instantiate "Breed-Specific Subset-Subset Comparison" pattern

The **"Austrian Milk Custom Breed-Specific Subset-Subset Comparison"** pattern is parameter-free as all parameters were bound in the course of the instantiation, that is, it can be executed. The domain expert formulates the statement in Listing 3.18 to execute the pattern and thus receive all queries representing the analysis problem. The resulting queries are visualized in the response area (Figure 6.15). As only one template, representing the patterns structure in SQL, was defined the result contains a single OLAP query in SQL respectively. It should be noted that Figure 6.15 only depicts a snippet of the query generated by the repository application, nevertheless it can be seen that, for example, **"Feeding"** was inserted as <sourceCube> in the expression (see Appendix E for the whole generated query). Thus, the execute pattern requirement (EPR) is satisfied as far as intended for this work, i.e., the behavior for the grounding is not implemented.

OLAP Pattern Editor

Command:

```
EXECUTE PATTERN "Austrian Milk Models"/"Dairy OLAP
Patterns"/"Austrian Milk Custom Breed-Specific Subset-Subset
Comparison"
FOR "Austrian Milk Models"/"Dairy Precision Farming"
```

SEND

```
WITH baseCube AS (
  SELECT *
  FROM   "Feeding" sc JOIN
       "Animal" a ON sc."Cattle" = a."Animal"
 WHERE   sc."Feed Consumption" < 80  AND  a."Main Breed" = "Jersey" ),
```

Figure 6.15: Execute "Austrian Milk Custom Breed-Specific Subset-Subset Comparison" pattern

The repository application also allows to gain an overview of patterns in the **"Dairy OLAP Patterns"** catalogue. To this end a domain or DWH expert can formulate a SHOW statement to retrieve the names of all patterns included. The result is a representation showing the catalogue name, followed by the list of comprised patterns, which is in this case the **"Breed-Specific Subset-Subset Comparison"** and **"Austrian Milk Custom Breed-Specific Subset-Subset Comparison"** pattern (see Figure 6.16). Accordingly, the implementation also fulfills the show organization element requirement (SOLR).

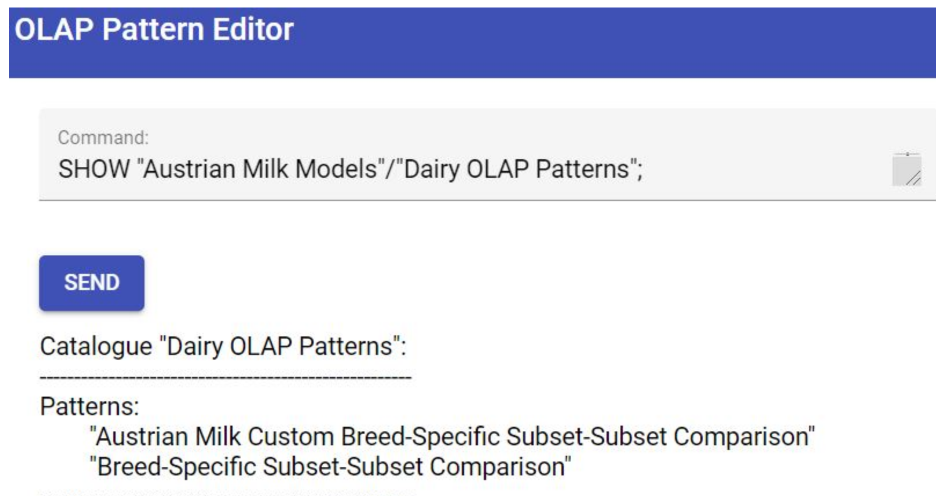


Figure 6.16: Show "Dairy OLAP Patterns" catalogue

In addition to get an overview of patterns in a catalogue, the repository application also support the search for specfic patterns. To find a pattern that allows to handle subsets, i.e., the name attribute contains the search term **"Subset"**, a domain or DWH expert can formulate the statement in Figure 6.17. The repository application provides the object representation of the patterns matching the search criteria. For the current search, the previously defined **"Breed-Specific Subset-Subset Comparison"** and its instantiation **"'Austrian Milk Custom Breed-Specific Subset-Subset Comparison"** are returned as JSON objects (Satisfying the search content element requirement (SCER)).

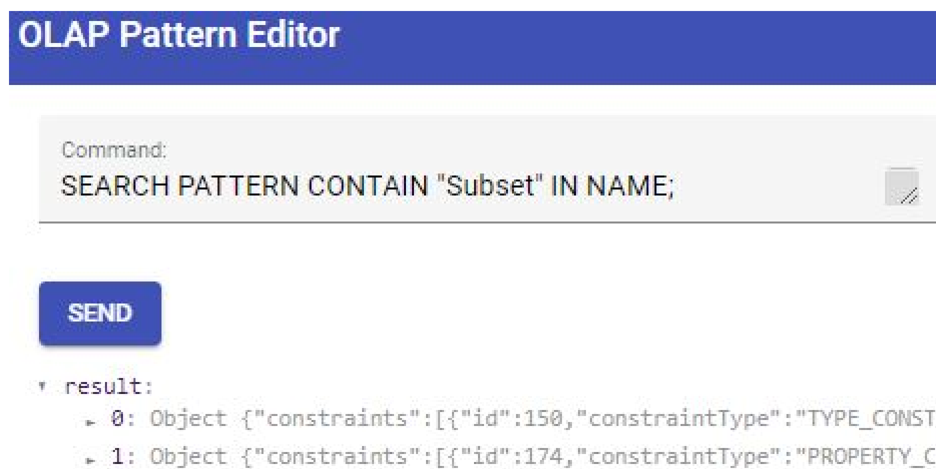


Figure 6.17: Search for patterns containing "Subset" in their names

The repository application also allows to get the detailed definitions of specific patterns by formulating corresponding SHOW statements. The definition of the **"Breed-Specific Subset-Subset**

Comparison" can be obtained by a domain or DWH expert by formulating the corresponding SHOW statement in Figure 6.18. The returned definition in in Figure 6.18 shows that the show content element requirement (SCDR) is satisfied by the repository application.

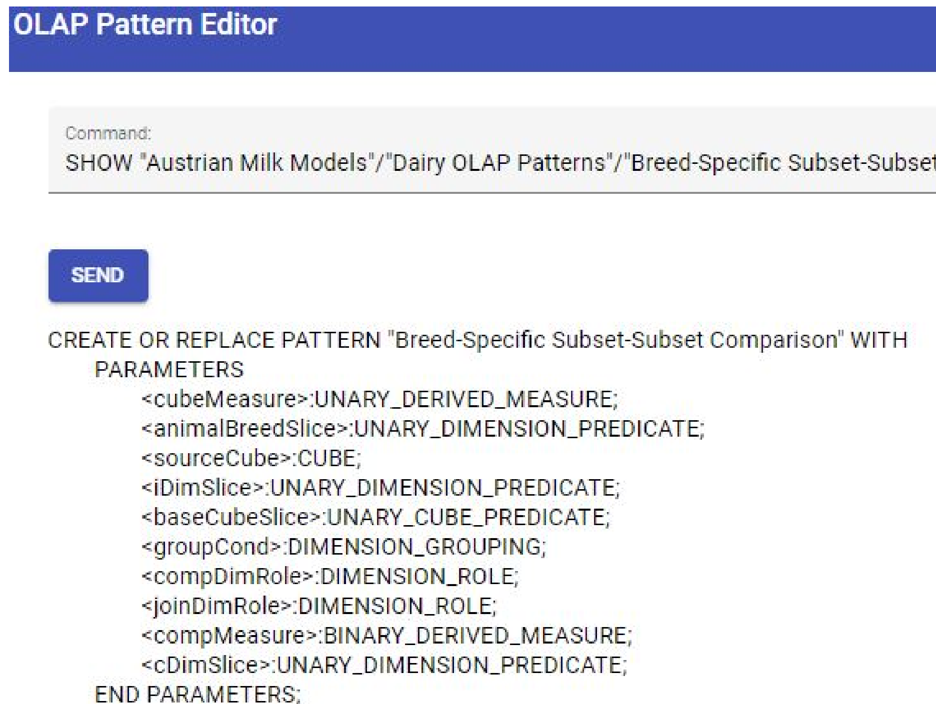


Figure 6.18: Show "Breed-Specific Subset-Subset Comparison" patterns

Finally, the repository application also support the removal of catalogues. To delete the created catalogue "Dairy OLAP Patterns" a DWH expert can formulate the statement in Listing 3.2. The repository application responds with **"Catalogue deleted!"** informing that the operation is successfully executed (see Figure 6.19). It is worth noting that any pattern, pattern template or pattern description contained has been deleted as well. Consequently, the delete organization element requirement (DOER) is fulfilled as delete statements for other organisation elements follow the same process.

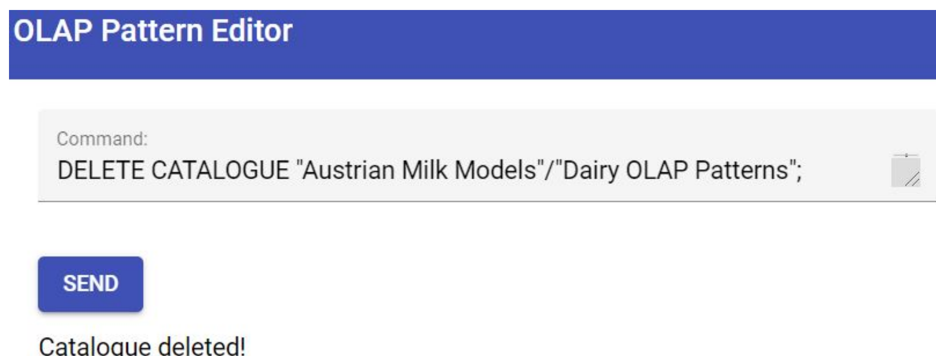


Figure 6.19: Delete catalogue

The DHW expert can check whether the catalogue is delete by formulating a subsequent show (and equivalently a search) statement regarding one of the comprised patterns. As depicted in Figure 6.20 the formulated show statement results in error; the repository application responds with **"Path not found"**. This shows that the patterns were deleted as well (cascading delete),

which demonstrates that the delete content element requirement (DCER) is also fulfilled.

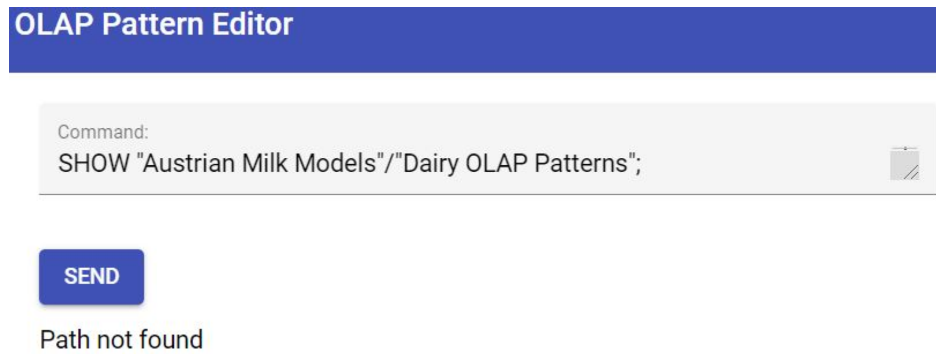


Figure 6.20: Show deleted pattern

Accordingly, all user requirements as defined in subsection 3.2.1 to support both domain and DWH experts can be fulfilled by the design and implementation as proposed in this work. A demonstration with potential users, was not the aim of this demonstration. Nevertheless, a user study is crucial for future development of the OLAP pattern repository, to obtain a detailed assessment of potential users regarding the benefits of the application.

Chapter 7

Conclusion

This thesis proposes a repository application for OLAP patterns supporting both their definition and usage within organizations. Accordingly, the OLAP pattern repository application provides all the functionality necessary to define generic, best practice solutions for OLAP queries, adapt them to certain analysis situations, and automatically generate a specific, executable OLAP query satisfying the user's needs. To this end, the definition language is analyzed to define requirements that must be fulfilled in order to support users in their work with patterns. From these requirement, a general, component-based repository design is derived. In addition a prototypical implementation of the OLAP pattern repository – following the design – allows for users to depict the multidimensional data models used by their organizations consisting of cubes and dimensions and the vocabulary from the organization's domain in form of business terms. Based on these enriched multidimensional model elements, users can define, instantiate, and execute OLAP patterns. Besides the definition of patterns, both design and implementation also consider OLAP pattern extensions to be added in future evolution steps. Furthermore, the repository application allows for DWH experts to maintain the content by providing means of structuring as well as update and delete functionalities.

Considering the novelty of the approach and the fact that practical experience regarding the use of OLAP patterns is rare, numerous future tasks can be identified. The editor application currently provides a simple (command line) interface to send commands (text) to the repository application and to output the result. This is a sufficient way to support users in executing any operation in context of the OLAP pattern approach, however, leaves room for further improvements. One possible extension could be to orientate towards existing, similar tools available and take over prominent functions from these applications. For example, similar to tools for writing database queries, the editor application introduced in this work can be extended by features such as syntax highlighting and auto-completion which could make writing statements more comfortable and less error prone. Thus, these features may be worth adding in a future expansion stage. Another possible way of extending the editor application might be to offer a grapfical user interface, enabling the user to depict MDMs and create patterns and terms using the graphical notation introduced with the OLAP pattern approach. This means that users could define cubes, dimensions, business terms, and patterns by adding graphical items instead of formulating textual commands. Since several extensions are conceivable, it would nevertheless be reasonable to continue research by conducting a study on which functionality is expected by practitioners. Regarding the repository application, the most obvious task is the integration of the KBS component to be developed. For the comprehensiveness, the extensions to the basic OLAP pattern approach – introduced

by Kovacic et al.[1] – might be worth adding to the implementation such as complex types and collections and generic business terms. Finally, to support data warehouse implementations that do not follow the naming of the conceptual representation, mapping tables could be considered allowing to match logical and conceptual elements.

Bibliography

[1] I. Kovacic, C. G. Schuetz, B. Neumayr and M. Schrefl, 'OLAP Patterns: A Pattern-Based Approach to Multidimensional Data Analysis', *Arbeitspapier am Institut für Wirtschaftsinformatik - Data & Knowledge*, 2020.

[2] T. W. Guenther, 'Conceptualisations of 'controlling'in German-speaking countries: analysis and comparison with Anglo-American management control frameworks', *Journal of Management Control*, vol. 23, no. 4, pp. 269–290, 2013.

[3] T. Reichmann, 'Controlling mit Kennzahlen und Management-Tools', *Die systemgestützte Controlling-Konzeption*, vol. 7, 2006.

[4] W. H. Inmon, *Building the data warehouse*, 3. ed., ser. Wiley computer publishing Timely, practical, reliable. New York, N.Y.: Wiley, 2002, ISBN: 0-471-08130-2.

[5] A. A. Vaisman and E. Zimányi, 'Data Warehouse Systems - Design and Implementation', Data-Centric Systems and Applications, 2014. DOI: 10.1007/978-3-642-54655-6. [Online]. Available: https://doi.org/10.1007/978-3-642-54655-6.

[6] M. Golfarelli, D. Maio and S. Rizzi, 'The dimensional fact model: A conceptual model for data warehouses', *International Journal of Cooperative Information Systems*, vol. 7, no. 02n03, pp. 215–247, 1998.

[7] S. Chaudhuri and U. Dayal, 'An Overview of Data Warehousing and OLAP Technology', *SIGMOD Rec.*, vol. 26, no. 1, pp. 65–74, 1997. DOI: 10.1145/248603.248616. [Online]. Available: https://doi.org/10.1145/248603.248616.

[8] I. Kovacic, C. G. Schuetz, S. Schausberger, R. Sumereder and M. Schrefl, 'Guided Query Composition with Semantic OLAP Patterns', in *Proceedings of the Workshops of the EDBT/ICDT 2018 Joint Conference (EDBT/ICDT 2018), Vienna, Austria, March 26, 2018*, N. Augsten, Ed., ser. CEUR Workshop Proceedings, vol. 2083, CEUR-WS.org, 2018, pp. 67–74. [Online]. Available: http://ceur-ws.org/Vol-2083/paper-11.pdf.

[9] W. Eckerson, 'Pervasive business intelligence Techniques and Technologies to Deploy BI on an Enterprise Scale', *TDWI Best Practices Report*, 2008.

[10] G. Allen and J. Parsons, 'Is Query Reuse Potentially Harmful? Anchoring and Adjustment in Adapting Existing Database Queries', *Inf. Syst. Res.*, vol. 21, no. 1, pp. 56–77, 2010. DOI: 10.1287/isre.1080.0189. [Online]. Available: https://doi.org/10.1287/isre.1080.0189.

[11] C. G. Schuetz, S. Schausberger, I. Kovacic and M. Schrefl, 'Semantic OLAP Patterns: Elements of Reusable Business Analytics', in *On the Move to Meaningful Internet Systems. OTM 2017 Conferences - Confederated International Conferences: CoopIS, C&TC, and ODBASE 2017, Rhodes, Greece, October 23-27, 2017, Proceedings, Part II*, H. Panetto, C. Debruyne, W. Gaaloul, M. P. Papazoglou, A. Paschke, C. A. Ardagna and R. Meersman, Eds., ser. Lecture Notes in Computer Science, vol. 10574, Springer, 2017, pp. 318–336. DOI: 10.1007/978-3-319-69459-7_22. [Online]. Available: https://doi.org/10.1007/978-3-319-69459-7%5C_22.

[12] C. Alexander, *A pattern language: towns, buildings, construction*. Oxford university press, 1977.

[13] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*, 1st ed. Amsterdam: Addison-Wesley Longman, 1995, 37. Reprint (2009), ISBN: 0201633612.

[14] C. G. Schuetz, S. Schausberger and M. Schrefl, 'Building an active semantic data warehouse for precision dairy farming', *J. Organ. Comput. Electron. Commer.*, vol. 28, no. 2, pp. 122–141, 2018. DOI: 10.1080/10919392.2018.1444344. [Online]. Available: https://doi.org/10.1080/10919392.2018.1444344.

[15] M. Elias, K. Shahzad and P. Johannesson, 'A business process metadata model for a process model repository', in *Enterprise, Business-Process and Information Systems Modeling*, Springer, 2010, pp. 287–300.

[16] K. Shahzad, B. Andersson, M. Bergholtz, A. Edirisuriya, T. Ilayperuma, P. Jayaweera and P. Johannesson, 'Elicitation of Requirements for a Business Process Model Repository', in *Business Process Management Workshops, BPM 2008 International Workshops, Milano, Italy, September 1-4, 2008. Revised Papers*, D. Ardagna, M. Mecella and J. Yang, Eds., ser. Lecture Notes in Business Information Processing, vol. 17, Springer, 2008, pp. 44–55. DOI: 10.1007/978-3-642-00328-8_5. [Online]. Available: https://doi.org/10.1007/978-3-642-00328-8%5C_5.

[17] D. Spinellis, 'Notable design patterns for domain-specific languages', *J. Syst. Softw.*, vol. 56, no. 1, pp. 91–99, 2001. DOI: 10.1016/S0164-1212(00)00089-3. [Online]. Available: https://doi.org/10.1016/S0164-1212(00)00089-3.

[18] A. W. Appel, *Modern compiler implementation in C*. Cambridge university press, 2004.

[19] W. M. Waite and G. Goos, *Compiler construction*. Springer Science & Business Media, 2012.

[20] I. Sommerville, *Software engineering*, 9th ed. Boston: Pearson, 2011, ISBN: 9780137053469.

[21] T. J. Parr and R. W. Quong, 'ANTLR: A Predicated- *LL(k)* Parser Generator', *Softw. Pract. Exp.*, vol. 25, no. 7, pp. 789–810, 1995. DOI: 10.1002/spe.4380250705. [Online]. Available: https://doi.org/10.1002/spe.4380250705.

[22] E. J. O'Neil, 'Object/Relational Mapping 2008: Hibernate and the Entity Data Model (Edm)', ser. SIGMOD '08, Vancouver, Canada: Association for Computing Machinery, 2008, pp. 1351–1356, ISBN: 9781605581026. DOI: 10.1145/1376616.1376773. [Online]. Available: https://doi.org/10.1145/1376616.1376773.

APPENDIX A

OLAP pattern language

In Listing A.1 the grammar definition of the OLAP pattern language is depicted. For a detailed description of the grammar please refer to the publication Kovacic et al. [1].

```
1 grammar emdml;
2
3 emdm_stmt: ( (c_stmt | d_stmt | g_stmt | i_stmt | x_stmt | s_stmt | f_stmt )
   ↪ ';' )* ;
4
5 /****** CREATE STATEMENTS *******/
6 c_stmt: CREATE ( OR REPLACE )? ( cp_stmt | ct_stmt | cm_stmt | cr_stmt ) ;
7 cp_stmt: PATTERN ( p_def | p_descr | p_temp ) ;
8 ct_stmt: t_def | ( TERM ( t_descr | t_temp  ) );
9 cm_stmt : cube_def | dim_def;
10 cr_stmt : r_exp;
11
12 /****** DELETE STATEMENTS *******/
13 d_stmt: DELETE ( p_del | t_del | m_del | r_exp ) ;
14
15 /****** EXECUTE STATEMENTS *******/
16 x_stmt  : EXECUTE ( p_exec );
17
18 /****** INSTANTIATE STATEMENTS *******/
19 i_stmt  : INSTANTIATE ( p_inst );
20
21 /****** EVALUATE STATEMENTS  *******/
22 g_stmt  : GROUND ( p_grnd );
23
24 /********** SHOW STATEMENTS *********/
25 s_stmt  : SHOW path_exp;
26
27 /********** SEARCH STATEMENTS *********/
28 f_stmt: SEARCH s_trgt ( IN path_exp )? s_exp+ ;
29 s_trgt: REPOSITORY  | CATALOGUE | GLOSSARY  | MULTIDIMENSIONAL_MODEL | PATTERN
   ↪ | TERM | CUBE | DIMENSION;
30 s_exp: CONTAIN s_str IN ( s_sct (',' s_sct )* )+;
31 s_sct:  NAME | LANGUAGE | ALIAS | PROBLEM | SOLUTION | EXAMPLE | RELATED ;
32
33 /********** REPOSITORY RELATED STATEMENTS *********/
34 r_exp: ( REPOSITORY | CATALOGUE | GLOSSARY | MULTIDIMENSIONAL_MODEL ) s_name ;
```

```
35
36 /********** PATTERN RELATED STATEMENTS **********/
37 p_def:  p_name WITH
38   ( PARAMETERS param_decl+ END PARAMETERS COL )?
39   ( DERIVED ELEMENTS derv_decl+ END DERIVED ELEMENTS COL )?
40   ( LOCAL CUBES lc_decl+ END LOCAL CUBES COL )?
41   ( CONSTRAINTS cstr_decl+ END CONSTRAINTS COL)? END PATTERN;
42 p_descr: DESCRIPTION FOR p_name WITH
43   ( ( LANGUAGE '=' lang_name
44       | ALIAS '=' alias_name ( ',' alias_name )*
45       | PROBLEM '=' prob_txt
46       | SOLUTION '=' sol_txt
47       | EXAMPLE '=' ex_txt
48       | RELATED '=' p_loc_name ( ',' p_loc_name)* ) COL
49   )+ END PATTERN DESCRIPTION;
50 p_temp: TEMPLATE FOR p_name WITH temp_elem+ END PATTERN TEMPLATE;
51 p_inst: PATTERN p_name AS p_name WITH BINDINGS?
52     binding_exp ( ',' binding_exp )*
53   (END BINDINGS)?
54   ( FOR path_exp  )?;
55 p_del:  PATTERN ( p_del_descr | p_del_temp | p_name ) ;
56 p_del_descr: DESCRIPTION FOR p_name ( WITH LANGUAGE '=' lang_name)? ;
57 p_del_temp: TEMPLATE FOR p_name ( WITH
58     ( temp_elem_meta )+
59     ( ',' temp_elem_meta )* )?;
60 p_grnd: PATTERN p_name AS p_name FOR mdm_name;
61 p_exec: PATTERN p_name FOR mdm_name USING voc_name
62   ( WITH TEMPLATE
63     ( temp_elem_meta )+ ( ',' temp_elem_meta )*
64   )?;
65
66 /********** TERM RELATED STATEMENTS **********/
67 t_def:  t_type t_name WITH
68   ( PARAMETERS param_decl+ END PARAMETERS COL )?
69   ( CONSTRAINTS cstr_decl+ END CONSTRAINTS COL )?
70   ( RETURNS value_set_name COL )? END t_type ;
71 t_descr: DESCRIPTION FOR t_name WITH (
72     ( LANGUAGE '=' lang_name
73       | ALIAS '=' t_name ( ',' t_name )*
74       | DESCRIPTION '=' descr_txt ) COL )+
75     END TERM DESCRIPTION ;
76 t_temp: TEMPLATE FOR t_name WITH temp_elem+ END TERM TEMPLATE;
77 t_del:  TERM ( t_del_descr  | t_del_temp | t_name ) ;
78 t_del_descr: DESCRIPTION FOR t_name (
79     WITH LANGUAGE '=' lang_name )? ;
80 t_del_temp: TEMPLATE FOR t_name ( WITH
81     ( temp_elem_meta)+ (',' temp_elem_meta)* )?;
82
83 /********** PATTERN AND TERM RELATED STATEMENTS **********/
84 temp_elem: ( temp_elem_meta | EXPRESSION '=' temp_txt) COL ;
85 temp_elem_meta:
86   DATA_MODEL '=' model_name
87   | VARIANT '=' variant_name
88   | LANGUAGE '=' lang_name
```

```
89    | DIALECT '=' dialect_name;
90  param_decl: var_decl COL;
91  derv_decl:  var_decl
92    ( for_exp var_exp )? '<=' elem_exp '.' ( elem_exp | RETURNS ) COL;
93  for_exp: FOR (var_exp
94      | '(' var_exp ( ',' var_exp)+ ')' ) IN var_exp WITH;
95  var_decl  : var_exp ':' type op_exp?;
96
97  /* CONSTRAINT DECLARATIONS */
98  cstr_decl: ( sv_cstr_decl | mv_cstr_decl) COL ;
99  mv_cstr_decl: for_exp sv_cstr_decl(',' sv_cstr_decl )*;
100 sv_cstr_decl: type_cstr_decl | dom_cstr_decl | prop_cstr_decl |
      ↪ return_cstr_decl | app_cstr_decl | scope_cstr_decl | rollup_cstr_decl |
      ↪ descr_cstr_decl | term_cstr_decl ;
101 type_cstr_decl: elem_exp ':' type;
102 dom_cstr_decl:  elem_exp '.' elem_exp ':' elem_exp;
103 prop_cstr_decl: elem_exp HAS m_prop_type elem_exp;
104 return_cstr_decl: elem_exp RETURNS elem_exp;
105 app_cstr_decl: elem_exp IS_APPLICABLE_TO
106   ( elem_exp | '(' elem_exp ',' elem_exp ')' );
107 scope_cstr_decl: ( var_exp WITH )? scope_exp  (',' scope_exp )*;
108 scope_exp:  var_exp IN var_exp;
109 rollup_cstr_decl: elem_exp '.' elem_exp ROLLS_UP_TO elem_exp '.' elem_exp;
110 descr_cstr_decl:  elem_exp '.' elem_exp DESCRIBED_BY elem_exp '.' elem_exp;
111 term_cstr_decl: TERM elem_exp WITH
112   ( EXPECTED PARAMETERS param_decl+ END EXPECTED PARAMETERS COL)?
113   ( EXPECTED CONSTRAINTS ( sv_cstr_decl COL )+ END EXPECTED CONSTRAINTS COL
114   )? END TERM ;
115
116 /* FRAGMENT DECLARATIONS */
117 lc_decl: ( sv_lc_decl | mv_lc_decl ) COL;
118 mv_lc_decl: for_exp sv_lc_decl ( ',' sv_lc_decl )* ;
119 sv_lc_decl: type_lc_decl | dom_lc_decl | prop_lc_decl ;
120 type_lc_decl: const_exp ':' type;
121 dom_lc_decl:  elem_exp '.' elem_exp ':' ( elem_exp );
122 prop_lc_decl: elem_exp HAS m_prop_type elem_exp;
123 rollup_frag_decl: elem_exp '.' elem_exp ROLLS_UP_TO elem_exp '.' elem_exp;
124 descr_frag_decl:  elem_exp '.' elem_exp DESCRIBED_BY elem_exp '.' elem_exp;
125 binding_exp:  var_exp '+'? '=' val_exp;
126 val_exp:  sv_exp | mv_exp;
127 sv_exp: const_exp
128   | ( '(' elem_acc_exp (',' elem_acc_exp )+ ')' );
129 mv_exp: '{' sv_exp ( ',' sv_exp )* '}';
130 elem_acc_exp: const_exp | var_acc_exp;
131
132 /* MULTI-VALUED EXPRESSIONS */
133 var_acc_exp:
134   '<' var_label '>'
135   ( '()' tuple_acc_exp? )?
136   ( array_acc_exp | map_simple_acc_exp )? ;
137 array_acc_exp:  '[' idx_no ']';
138 map_simple_acc_exp: '*' '['
139   ( idx_name | ( '(' idx_name ( ',' idx_name )+ ')' ) ) ']' ;
140 tuple_acc_exp:  '.[' idx_no ']';
```

```
141 elem_exp: const_exp | var_exp;
142 var_exp: '<' var_label '>' tuple_exp? ( map_exp | array_exp )?;
143 tuple_exp: ( '()' tuple_acc_exp?
144   | '(' ( idx_exp ( ',' idx_exp )+ )? ')' );
145 array_exp:  ( '[]' | array_acc_exp );
146 map_exp:  '*' map_acc_exp?;
147 map_acc_exp:  '[' idx_exp ']';
148 idx_exp: idx_name | idx_no | sv_exp | var_exp ;
149 const_exp:  const_name;
150 op_exp: IS_OPTIONAL;
151 path_exp: elem_name ( '/' elem_name )*;
152
153 /**********  MULTIDIMENSIONAL MODEL RELATED STATEMENTS **********/
154 cube_def:
155   CUBE cube_name WITH MEASURE PROPERTIES meas_decl+ END MEASURE PROPERTIES COL
156   DIMENSION_ROLE PROPERTIES dim_role_decl+ END DIMENSION_ROLE PROPERTIES COL
157   END CUBE;
158 dim_def: DIMENSION dim_name WITH (
159     ( LEVEL PROPERTIES lvl_decl+ END LEVEL PROPERTIES COL ) |
160     ( ATTRIBUTE PROPERTIES attr_decl+ END ATTRIBUTE PROPERTIES COL ) |
161     ( CONSTRAINTS ( roll_up_rel_decl | descr_by_rel_decl )+ END CONSTRAINTS
        ↪ COL)
162   )+ END DIMENSION;
163 m_del: ( CUBE cube_name | DIMENSION dim_name );
164 meas_decl:  meas_name ':' val_set_name COL;
165 dim_role_decl:  dim_role_name ':' dim_loc_name COL;
166 lvl_decl: lvl_name ':' val_set_name COL;
167 attr_decl:  attr_name ':' val_set_name COL;
168 roll_up_rel_decl: lvl_name ROLLS_UP_TO lvl_name COL;
169 descr_by_rel_decl:  lvl_name DESCRIBED_BY attr_name COL;
170
171 /**********  AVAILABLE TYPES **********/
172 type: m_entity_type | m_prop_type | t_type | v_type | a_type;
173 t_type: UNARY_CUBE_PREDICATE | BINARY_CUBE_PREDICATE |
      ↪ UNARY_CALCULATED_MEASURE | BINARY_CALCULATED_MEASURE | CUBE_ORDERING |
      ↪ UNARY_DIMENSION_PREDICATE | BINARY_DIMENSION_PREDICATE |
      ↪ DIMENSION_GROUPING | DIMENSION_ORDERING;
174 m_entity_type:  CUBE | DIMENSION;
175 m_prop_type: MEASURE | DIMENSION_ROLE | CUBE_PROPERTY | LEVEL | ATTRIBUTE |
      ↪ DIMENSION_PROPERTY;
176 v_type: NUMBER_VALUE_SET | STRING_VALUE_SET | val_set_name;
177 a_type: BINARY_TUPLE | TERNARY_TUPLE | QUARTERNARY_TUPLE;
```

Listing A.1: Syntax of the OLAP pattern language

Appendix B

Macro language

The Listing B.1 depicts the island grammar design to analyse the expression of a pattern template following [1].

```
1 grammar macrol3;
2
3 macrol: ( macro_call | txt )* ;
4 txt: QUERY_TEXT ;
5 macro_call: macro_name '(' macro_call_arguments ')'
6   ( '{' macro_body '}')?;
7 macro_name: DIMKEY | EXEC ;
8 macro_body: macrol;
9 macro_call_arguments: macro_call_argument
10   ( ',' macro_call_argument )* ;
11 macro_call_argument: QUOTED_NAME | UNQUOTED_NAME;
```

Listing B.1: Syntax of the macro language

Appendix C

JSON response

Listing C.1 depicts a JSON response sent to the controller if a search statement for the Jersey business term is performed.


```
{
    "result": [
        {
            "constraints": [
                {
                    "id": 50,
                    "constraintType": "DOMAIN_CONSTRAINT",
                    "entity": {
                        "id": 89,
                        "kind": "SINGLE_VARIABLE",
                        "name": "ctx",
                        "position": 1,
                        "variableRole": "CONTEXT_PARAMETER",
                        "valueKind": "SINGLE",
                        "optional": "N",
                        "value": null
                    },
                    "property": {
                        "id": 88,
                        "kind": "SINGLE_VALUE",
                        "value": "\"Main Breed\""
                    },
                    "domain": {
                        "id": 87,
                        "kind": "SINGLE_VALUE",
                        "value": "\"Breed Name\""
                    }
                },
                {
                    "id": 49,
                    "constraintType": "TYPE_CONSTRAINT",
                    "element": {
                        "id": 89,
                        "kind": "SINGLE_VARIABLE",
```

```
35                          "name": "ctx",
36                          "position": 1,
37                          "variableRole": "CONTEXT_PARAMETER",
38                          "valueKind": "SINGLE",
39                          "optional": "N",
40                          "value": null
41                      },
42                      "type": {
43                          "id": 90,
44                          "kind": "SINGLE_VALUE",
45                          "value": "DIMENSION"
46                      }
47                  }
48              ],
49              "derivationRules": [],
50              "targets": [
51                  {
52                      "id": 89,
53                      "kind": "SINGLE_VARIABLE",
54                      "name": "ctx",
55                      "position": 1,
56                      "variableRole": "CONTEXT_PARAMETER",
57                      "valueKind": "SINGLE",
58                      "optional": "N",
59                      "value": null
60                  },
61                  {
62                      "id": 88,
63                      "kind": "SINGLE_VALUE",
64                      "value": "\"Main Breed\""
65                  },
66                  {
67                      "id": 90,
68                      "kind": "SINGLE_VALUE",
69                      "value": "DIMENSION"
70                  },
71                  {
72                      "id": 87,
73                      "kind": "SINGLE_VALUE",
74                      "value": "\"Breed Name\""
75                  }
76              ],
77              "templates": [
78                  {
79                      "id": 19,
80                      "dataModel": null,
81                      "language": "\"SQL\"",
82                      "dialect": "\"ORACLEv11\"",
83                      "variant": null,
84                      "expression": "\"*{ <ctx>.\"\"Main Breed\"\" =
```

```
                        \"\"Jersey\"\" }*\""
85                  }
86              ],
87              "fragments": [],
88              "id": 19,
89              "name": "\"Jersey\"",
90              "type": {
91                  "id": 2,
92                  "name": "UNARY_DIMENSION_PREDICATE"
93              },
94              "returnType": null,
95              "descriptions": [
96                  {
97                      "id": 19,
98                      "language": "\"English\"",
99                      "aliases": [
100                         "\"Cattle Breed Jersey\""
101                     ],
102                     "description": "\"Restriction of result to cattle of
                            main breed Jersey\""
103                 }
104             ]
105         }
106     ]
107 }
```

Listing C.1: JSON representation of the Jersey business term

APPENDIX D

Template Expression

Listing D.1 depicts a the expression for the **"Breed-Specific Subset-Subset Comparison** pattern in the Oracle version 11 SQL dialect [1].

```
*{ WITH baseCube AS (
   SELECT *
   FROM   <sourceCube> sc JOIN
          ""Animal"" a ON sc.""Cattle"" = a.}*$dimKey(""Animal"")*{
  WHERE   }* $expr(<baseCubeSlice>, sc) *{ AND }*
          $expr(<animalBreedSlice>, a)*{),

interestCube AS (
  SELECT }* $expr(<groupCond>, jd)*{,
          }* $expr(<cubeMeasure>, bc)*{ AS <cubeMeasure>
  FROM    baseCube bc JOIN
          <joinDim> jd ON bc.<joinDimRole>=jd.}*$dimKey(<joinDim>)*{ JOIN
          <compDim> cd ON bc.<compDimRole>=cd.}*$dimKey(<compDim>)*{
  WHERE   }*$expr(<iDimSlice>, cd)*{
  GROUP BY }*$expr(<groupCond>, jd)*{),

comparisonCube AS (
  SELECT }*$expr(<groupCond>, jd)*{,
          }*$expr(<cubeMeasure>, bc)*{ AS <cubeMeasure>
  FROM    baseCube bc JOIN
          <joinDim> jd ON bc.<joinDimRole>=jd.}*$dimKey(<joinDim>)*{ JOIN
          <compDim> cd ON bc.<compDimRole>=cd.}*$dimKey(<compDim>)*{
  WHERE   }*$expr(<cDimSlice>, cd)*{
  GROUP BY }*$expr(<groupCond>, jd)*{)

SELECT }*$expr(<groupCond>, ic)*{,
       ic.<cubeMeasure> AS ""Group of Interest"",
       cc.<cubeMeasure> AS ""Group of Comparison"",
       }*$expr(<compMeasure>, ic, cc)*{ AS <compMeasure>
FROM   interestCube ic JOIN
       comparisonCube cc ON }*$expr(<groupCond>, ic)*{ = }*
           $expr(<groupCond>, cc)"
```

Listing D.1: Template Expression for Jersey business term

Appendix E

Generated OLAP Query

Listing E.1 depicts an OLAP query as generated for the **"Austrian Milk Custom Breed-Specific Subset-Subset Comparison"**.

```
 WITH baseCube AS (
   SELECT *
   FROM    "Feeding" sc JOIN
           "Animal" a ON sc."Cattle" = a."Animal"
  WHERE    sc."Feed Consumption" < 80  AND  a."Main Breed" = "Jersey" ),

interestCube AS (
  SELECT   jd."Farm Id" ,
           AVG(bc."Feed Consumption")  AS "Average Feed Consumption"
  FROM    baseCube bc JOIN
          "Farm" jd ON bc."Farm"=jd."Farm Id" JOIN
          "Animal" cd ON bc."Cattle"=cd."Animal"
  WHERE    trunc((SYSDATE - cd."Date Of Birth")/365.25) < 3
  GROUP BY  jd."Farm Id" ),

comparisonCube AS (
  SELECT   jd."Farm Id" ,
           AVG(bc."Feed Consumption")  AS "Average Feed Consumption"
  FROM    baseCube bc JOIN
          "Farm" jd ON bc."Farm"=jd."Farm Id" JOIN
          "Animal" cd ON bc."Cattle"=cd."Animal"
  WHERE    trunc((SYSDATE - cd."Date Of Birth")/365.25) >= 3
  GROUP BY  jd."Farm Id" )

SELECT  ic."Farm Id" ,
       ic."Average Feed Consumption" AS "Group of Interest",
       cc."Average Feed Consumption" AS "Group of Comparison",
        ic."Average Feed Consumption")/cc."Average Feed Consumption"  AS
           "Average Feed Consumption Ratio"
FROM   interestCube ic JOIN
       comparisonCube cc ON  ic."Farm Id"  =  cc."Farm Id"
```

Listing E.1: Generated OLAP query