Universitat Politècnica de Catalunya

Facultat d'Informàtica de Barcelona

Master Thesis

Implementation of the multidimensional schemas integration method ORE

Director:
Petar Jovanovic
Advisor:
Alberto Abelló

Author:
Daria Mayorova

A thesis submitted in fulfilment of the requirements for the degree of Master in Information Technology

December 2013

Acknowledgements

I would like to thank all the people helped me to make this thesis possible.

First of all my thanks go to Petar Jovanovic for all the support he gave me throughout the work, and for great collaboration. Also, I am grateful to my advisor, Alberto Abelló, for believing in me, for his guidance and patience.

I would also like to thank Oscar Romero for his useful advice, and all the professors of the UPC who taught me during this master program for giving me necessary knowledge, which not only helped me greatly in the elaboration of this thesis, but also is very important for my professional life.

I wish to express my sincere gratitude to my family, especially to my mother who has always supported me and has been there for me. And, finally, thanks to all my friends for encouraging and believing in me.

Contents

Acknowledgements i

List of Figures v

List of Tables vii

1 Overview 1
- 1.1 Introduction . . . 1
- 1.2 Motivation . . . 2
- 1.3 Project goals . . . 2
- 1.4 Document structure . . . 3

2 State of the Art 5
- 2.1 Data warehousing . . . 5
- 2.2 Multidimensional model . . . 6
  - 2.2.1 Facts and dimensions . . . 6
  - 2.2.2 Kinds of multidimensional schemas . . . 8
  - 2.2.3 ETL processes . . . 8
- 2.3 Problem definition . . . 10
- 2.4 Related works . . . 10

3 ORE approach 13
- 3.1 Overview . . . 13
- 3.2 Inputs . . . 13
  - 3.2.1 Data sources . . . 13
  - 3.2.2 MDI . . . 14
- 3.3 ORE process . . . 18
- 3.4 Integration operations . . . 21
- 3.5 Cost model . . . 23
- 3.6 Stage 1: Matching Facts . . . 23
- 3.7 Stage 2: Matching Dimensions . . . 26
- 3.8 Stage 3: Complementing the MD design . . . 28
- 3.9 Stage 4: Integration . . . 29

4 Development 32
- 4.1 Development process . . . 32
  - 4.1.1 Overview of the existing process models . . . 32

4.1.2 Agile software development . . . 33
4.1.3 Process model chosen for the project . . . 33
4.2 Iterations . . . 36
4.2.1 Iteration 1. Initial prototype . . . 36
4.2.2 Iteration 2. Redesign of the initial prototype . . . 38
4.2.3 Iteration 3. Allowing several integration options . . . 38
4.2.4 Iteration 4. Basic GUI . . . 38
4.2.5 Iteration 5. Integration stage . . . 39
4.2.6 Iteration 6. CLI for batch input . . . 39
4.2.7 Iteration 7. Integration with the DB . . . 40
4.2.8 Iteration 8. Bugfixing and improvements . . . 40
4.3 Design and implementation . . . 41
4.3.1 Used technologies . . . 41
4.3.1.1 Programming language . . . 41
4.3.1.2 Development tools . . . 42
4.3.1.3 Version control system . . . 42
4.3.1.4 Database . . . 43
4.3.2 Reuse of existing code . . . 45
4.3.2.1 MGraph . . . 45
4.3.2.2 Classes for working with data sources . . . 46
4.3.3 Architecture and description of the packages . . . 47
4.3.3.1 General concepts . . . 47
4.3.3.2 Three-layer architecture . . . 48
4.3.3.3 Main class and controllers . . . 50
4.3.3.4 ORE process implementation . . . 51
4.3.4 Traceability Metadata . . . 52
4.3.4.1 Integration operations . . . 55
4.3.5 Transitive Closures Cache . . . 57
4.3.6 Database design . . . 60
4.4 Testing . . . 64
4.4.1 General overview . . . 64
4.4.2 Test cases for ORE . . . 65
4.4.3 ORE experiments . . . 68

5 Project planning and cost **70**
5.1 Initial planning . . . 70
5.2 Final planning . . . 70
5.3 Project cost estimation . . . 73

6 Conclusions **75**
6.1 My contribution . . . 75
6.2 Future work . . . 76

Bibliography **78**

Glossary **81**

Acronyms **83**

List of Figures

2.1 Examples of dimensions . . . 7
2.2 Star schema . . . 9
2.3 Snowflake schema . . . 9
2.4 Constellation schema . . . 9

3.1 Architecture of GEM and ORE . . . 14
3.2 Diagrammatic representation of the TPC-H ontology . . . 15
3.3 Single MD interpretations for IR1-IR5 . . . 16
3.4 Diagrammatic representation of the LearnSQL ontology . . . 17
3.5 Multidimensional interpretations for the requirement . . . 18
3.6 ORE stages . . . 19
3.7 ORE algorithm . . . 20
3.8 Examples of fact matching . . . 26
3.9 Matching the MD Interpretation for IR2 . . . 28
3.10 Matching facts with $rollupFacts$ . . . 28
3.11 INT algorithm . . . 30

4.1 Agile software development . . . 34
4.2 Extreme programming project . . . 35
4.3 Extreme programming iteration . . . 35
4.4 MGraph class diagram . . . 45
4.5 Ontology Reader class diagram . . . 47
4.6 Source Mappings Reader class diagram . . . 48
4.7 Three layer architecture . . . 49
4.8 Main class and controllers class diagram . . . 51
4.9 Traceability Metadata structure . . . 52
4.10 Merged facts in MDSchema . . . 54
4.11 MDI . . . 55
4.12 MGraph representation . . . 55
4.13 OREGraph representation . . . 55
4.14 Integration operations . . . 57
4.15 Transitive Closures Cache . . . 58
4.16 Transitive Closures class diagram . . . 59
4.17 Test data using TPC-H ontology . . . 66
4.18 Intermediate results for TPC-H ontology test . . . 67
4.19 Final results for TPC-H ontology test . . . 67
4.20 ORE execution time . . . 68

5.1 Initial planning. Page 1 71
5.2 Initial planning. Page 2 72

List of Tables

3.1 Integration operations 23

5.1 Project costs 74

Chapter 1

Overview

1.1 Introduction

Any kind of business generates big amounts of data. Since the society stepped into the digital era, data storage technologies have been evolving constantly. Database management systems (DBMSs), which are commonly used to store, maintain and retrieve data nowadays, have become an indispensable part of information technology infrastructure of any company.

The amounts of data stored in organizations have grown tremendously and continue growing at incredible rates. But it is very clear that raw data, understood as a set of records in databases, while being a crucial part of the information system supporting companys operation, doesnt bring much value for the company in long term. However, a proper analysis of this data can reveal useful information, transforming bytes of data into valuable knowledge about how the organization is functioning, what are the strong and the weak points, and helping to make decisions, which would improve the current situation.

Modern business environment is characterized by high competitiveness, and being able to analyze the existing data efficiently and make decisions using the extracted knowledge is essential for companies, which want to gain competitive advantage. However, this task is not easy, especially taking into account the amounts of data, which need to be processed. Business Intelligence (BI) is a concept, which relates to a wide range of techniques, processes and tools, which facilitate the discovery and analysis of information and support the decision-making.

One of the key technological components of business intelligence solutions is data warehouse a special data store, which integrates relevant data collected from heterogeneous sources, and provides tools to access and query the information. Due to the character of the queries performed over the data, which consists in analyzing pieces of information from different perspectives, most data warehouses are based on multidimensional design.

The design of the multidimensional schema of the data warehouse is a complex task. On the one hand, the multidimensional schema should satisfy the information requirements posed by the business users, and conform to the business glossary used by them. On the other hand, the design of the data warehouse is determined by the data sources, which tend to be heterogeneous, and the relations among the data must be considered.

This document presents the results of the elaboration of the masters final project, devoted to the problem of the multidimensional design of data warehouses. More specifically, the project addresses the integration of the multidimensional schemas in an iterative way following a novel approach called ORE which will be explained in detail in the following chapters of the document.

1.2 Motivation

The initial idea of the project came from the research performed in the Department of the Services and Systems Engineering of the Polytechnic University of Catalonia in collaboration with HP labs. In the paper [1] the co-authors Petar Jovanovic, Alberto Abelló, Oscar Romero and Alkis Simitsis presented a new approach for design and evolution of the multidimensional schemas. The main goal of this master thesis is to analyze this innovative theoretical approach from a practical point of view, and provide the implementation of the method in order to be able to evaluate the approach by performing experiments.

1.3 Project goals

The main goal of this project is the implementation of the ORE method, which will allow carrying out experiments of the method and evaluating its applicability for performing the design and evolution of the data warehouses multidimensional schemas in a semi-automatic way. The objectives of the project are the following:

1. Perform the implementation of the ORE method. The development of the software implementing the ORE method includes planning, analysis and definition of the requirements, design, implementation and testing. It includes the design of the necessary data structures, choosing of the technologies to be used, the existing libraries for reuse etc., implementation of the algorithms and providing the user interface.

2. Perform the experiments of the implemented ORE system. A suite of test cases should be elaborated and the testing performed in order to evaluate the ORE approach both in terms of the results quality and time spent for getting the solutions from the input data.

The requirements which need to be satisfied during the elaboration of the project are the following:

- Familiarize with the ORE method and other research in the field. Before starting with the implementation it is necessary to examine carefully and understand the ORE method. Also it is useful to familiarize with other works on the similar topics to get a broader view of the previous attempts of solving the raised issues.

- Document the project. The work on the project should be documented, and in the end the project report should be produced, containing theoretical part as well as the description of the development process and the experiments results.

1.4 Document structure

This document is a final report of the work performed during the elaboration of the master project. It is organized as follows:

Chapter 2 presents the State of the Art in the field of multidimensional design. First it introduces the main concepts of data warehousing and multidimensionality, later on the problem of the data warehouse design is defined more clearly and several approaches to solve it are discussed.

Chapter 3 provides the detailed description of the novel approach to the iterative data warehouse design based on requirements ORE.

Chapter 4 covers the development part of the project, including the description of the development process, the architecture of the application, the design of the data structures and the algorithms. The chapter also contains the overview of the technologies used for

the implementation and the libraries and previously implemented packages, which will be reused.

Chapter 5 discusses the aspects of project planning and the costs related to the elaboration of the project.

Chapter 6 presents the results of the projects and the conclusions extracted from this work. The ideas for future work are also proposed.

Chapter 2

State of the Art

This section first introduces the concepts of data warehousing and multidimensional models, later on the problem of multidimensional design based on requirements is defined more clearly, and some previous related works are discussed.

2.1 Data warehousing

A data warehouse, according to the definition given by Bill Inmon in [2], is "a subject-oriented, integrated, time-variant and non-volatile collection of data in support of management's decision making process".

Data warehouses being subject oriented means that they provide information about a particular subject, which is of interest to the users, involved in decision-making. The data which is irrelevant for the analysis can be disregarded.

When referring to data warehouse as integrated, it means that the data is gathered from multiple sources, which tend to be heterogeneous, and merged together to form a coherent whole, where the data is consistent and follows the predefined structure, naming conventions, types etc.

The time variant characteristics of data warehouses has several aspects. Firstly, it means that each piece of data stored in the data warehouse has some kind of time stamp, which allows to identify it with a particular time period. Thus, all the data can be viewed as a series of snapshots, ordered in time. Another aspect is the importance of historical data. Unlike in operational systems, which mostly deal with current, up-to-date data values, data warehouses provide information from historical perspective (several years). This time variance makes it possible to see how the data changes in large periods of time and analyze trends and relationships between data.

Non-volatility means that the data in the warehouse stays stable. Once the data is loaded into the data warehouse, it is not changed anymore. New data can be added, but by appending, not by modifying the existing data. Outside of the data loading procedures, which usually occur at a predefined schedule, the data in the data warehouse is accessible only for reading.

2.2 Multidimensional model

It can be clearly seen that data warehouses are very different from databases used in operational systems. Besides the already mentioned characteristics (time coverage, volatility, data integration, design orientation), another important difference is how the data is accessed. Operational systems are used to process the day-to-day transactions of an organization. Usually they perform On-Line Transactional Processing (OLTP), and so are often referred to as OLTP-systems. Such systems are optimized for fast execution of small transactions (which include inserting, updating and deleting data) and simple data queries; therefore operational databases are usually normalized. On the other hand, in data warehouses, as it has been already mentioned, the data is accessed in a read-only mode, and the queries tend to be very large and complex, performed over a very big amount of data, and in some cases ad hoc queries are allowed. Relational data model, which suits for OLTP systems, doesnt perform that well in case of data warehouses. For that reason, data warehouses require a special data structure which would optimize such queries.

2.2.1 Facts and dimensions

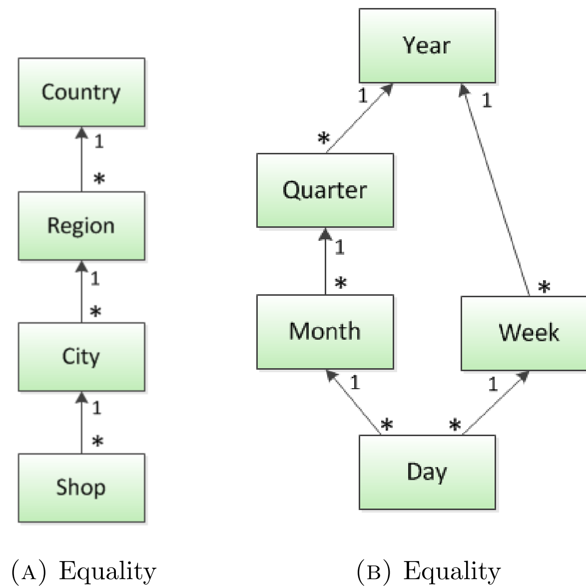
Multidimensional model is widely accepted as a good solution for data warehouse design. Multidimensionality is based on the fact/dimension dichotomy, where the **fact** is understood as the subject of analysis, and **dimensions** represent different perspectives, from which the subject can be analyzed, as if placed in an n-dimensional space.

Facts usually represent events important for decision-making; they contain quantitative attributes, which are relevant to analysis, and which are called measures. Some examples of facts and their **measures** (inside the parenthesis) could be: sale (price, discount), webpage visit (duration, number of clicks), university enrolment (number of credits, enrolment fee). Fact measures are usually numeric values and they can be aggregated.

Dimensions consist of attributes, which describe the factual data, and each dimension can be represented as a hierarchy of **levels**, where each level represent different granularity (level of detail) of the data.

The root of the hierarchy is represented by a single level, which is called "atomic level", and it is related to the fact by means of "1 – *" relationship. The atomic level corresponds to the finest granularity of the dimension. The relationships between the levels, from the atomic level, deeper to the hierarchy, have to be "to-one" relationships, i.e. there must be functional dependency between the finer level and a coarser level.

Each level contains attributes which will be called **descriptors**. A typical example of a dimension is Time dimension, which can contain the levels Year, Month, and Day. Other examples of dimensional concepts for a fact Sale and their descriptors (inside the parenthesis) are: Product dimension, which can be composed of the levels Product (id, color, description, size), Category (name, description); Customer dimension, which can hold only one level with the descriptors (id, name, address, phone number); Place where the sale was realized, with the levels Shop (id, name, address, phone), City (name of the city), Region (name of the region), Country (name of the country).



(A) Equality (B) Equality

FIGURE 2.1: Examples of dimensions

Figure 2.1 shows the examples of the hierarchies of dimensions. Figure 2.1a presents a simple hierarchy, where Shop is atomic level. The *Time* dimension shown in Figure 2.1b has multiple hierarchies defined on it. There is still a single atomic level (*Day*), but then the levels follow different paths: Week and Month-Quarter-Year. It provides more flexibility to the user, as in this way he can query the same data using different aggregation levels.

Looking at the Figure 2.1 we can notice that a dimension resembles as a directed graph, where the vertices are individual levels and the directed edges between them are "to-one" relationships. It is important to note that the graph has to be acyclic (DAG – Directed Acyclic Graph) in order to not violate the MD integrity constraints.

Multidimensional space is commonly referred to as data cube, where each cell holds a fact, and each dimension represents a perspective of analysis. Strictly speaking, a cube is only formed in case when three dimensions are considered, however the term data cube is used to define any n-dimensional space, or sometimes also the term hypercube is used.

2.2.2 Kinds of multidimensional schemas

There are several multidimensional schemas that are used in data warehouse design:

- Star schema Star schema consists of one fact and a number of dimensions, there is a 1:N relationship between each dimension and the fact. Each dimension is represented only by one dimensional concept, and holds all the attributes of this dimension. (See example in Figure 2.2)
- Snowflake schema As star schema, snowflake schema consists of a single fact, but each dimension is represented as a hierarchy of levels. Each level with its attributes is stored in a separate table. Thus, snowflake schema is a star schema, where the dimensions are normalized, in this way reducing data redundancy. (See example in Figure 2.3)
- Constellation schema Constellation schema is produced by several star or snowflake schemas, in which the facts share the same dimensions. In such schema there are several facts, stored in different tables, and there exists a possibility to perform complex queries involving more than one fact. (See example in Figure 2.4)

2.2.3 ETL processes

In order to feed the data warehouse with the data coming from multiple heterogeneous sources, the data should be transformed in a certain way, in order to conform to the data warehouse design. The processes which are responsible for this are called **ETL processes** (Extract Transformation Load). ETL processes include the following stages:

1. Extraction. This stage is used to select the raw data from the data sources, and physically extract them in order to process them in the next stages.

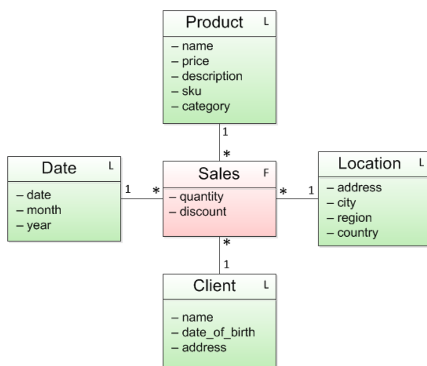


FIGURE 2.2: Star schema

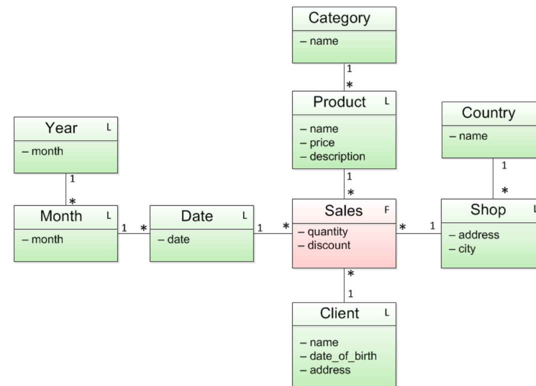


FIGURE 2.3: Snowflake schema

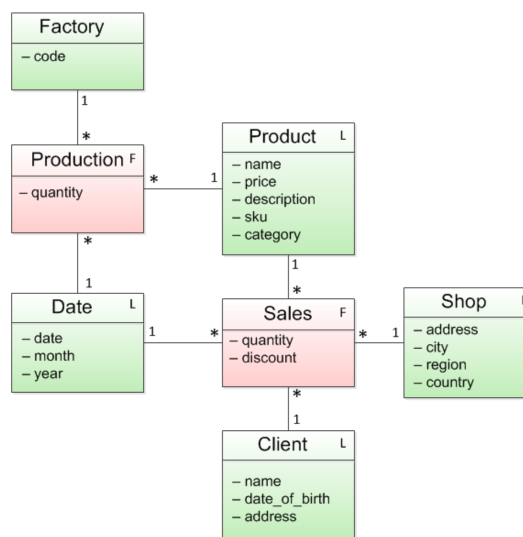


FIGURE 2.4: Constellation schema

2. Cleansing. Cleansing is a process which operates over raw data, and improves the quality of the data by removing duplicates, fixing errors in data (like missing, wrong or inconsistent values). As a result the high quality data are obtained, free from inconsistencies.

3. Transformation is one of the most complex stages, as its purpose is the reconciliation of the data obtained from heterogeneous sources and preparing them for loading into the data warehouse. At this stage, the data which represents equivalent concepts in different sources should be identified, matched and converted to the uniform format. Also, some data aggregation can be computed at this stage.

4. Load is the final stage of the ETL process and it consists in loading the already cleansed and transformed data into the data warehouse.

2.3 Problem definition

Designing a data warehouse is a difficult task.

One of the reasons for that is the heterogeneity of the data sources. Data warehouse is fed with the data coming from different operational databases; each database has its own structure, data formats, table-naming conventions etc. However, different tables and their fields can represent concepts, equivalent from the business point of view, this should be considered in the design of the data warehouse schema and in ETL processes.

Another reason is that the data warehouse design must comply with the information requirements posed by the business users, involved in the decision-making. The initial phase of the data warehouse project tends to be the most laborious for the data warehouse designer, as it requires tight communication with business users in order to elicit the information requirements and include the required factual and dimensional concepts into the multidimensional schema. Different users can require different information, at distinct aggregation levels, and the task of the designer is to reconcile all the requirements in order to build a data warehouse design which would satisfy all the users needs.

Another issue is that it is very improbable that once all the requirements have been defined, they will not be subject to change. In highly dynamical business environments, in order to stay competitive, enterprises need to be able to react to the changes in the market, search for new perspectives for data analysis, which can give valuable knowledge for decision-making. Thereby, the data warehouse designer should be able to deal with the new or changing information requirements, which can be quite a burdensome task.

Performing the design of data warehouse manually is very time-consuming and error-prone. The designer should have deep knowledge of the business terminology and users need on the one hand, and the structure of the data stored in the underlying sources, on the other hand. Automating the process of DW design can reduce significantly the time and effort required, and also keep controlling the compliance of the resulting schema with the multidimensional constraints, and the quality of the solution.

2.4 Related works

The problem of automating the data warehouse design and building MD schemas has been addressed by a number of research works. In [3] the authors presented a comprehensive overview of the approaches for data warehouse design.

The methodologies can be divided into two groups: requirement-driven and data-driven. Requirement-driven (also known as demand-driven) methods start from the definition of the business requirements; the multidimensional schema is derived from the requirements, and at some point the schema is reconciled with the data sources. Data-driven (supply-driven) approaches are based on the analysis of the data sources; the relational schema of the operational database is examined in order to identify structures and patterns which could form part of the multidimensional schema, and later the correspondence between the schema and the users requirements should be found. There exist also so-called hybrid methodologies, which combine the two approaches, and consider both data sources and users information requirements at the same time.

The professors and research assistants of the Department of the Services and Systems Engineering (ESSI) of the UPC in collaboration with other researchers have been working on finding the solution to the problems of the design and evolution of the multidimensional schemas for decision-making systems.

In the [4] the UPC professors Alberto Abelló and Oscar Romero, and Alkis Simitsis from the HP labs, presented the framework called GEM (Generating Etl and Multidimensional designs), which provided an innovative semi-automatic approach for generating multidimensional designs based on the formalized user requirements and the information about the data sources represented in the form of ontology. The business requirements should be formalized and represented in XML format, describing the measures (including aggregated values), dimensions at different granularity levels, and descriptors, used to slice the data. The data sources are represented in form of an OWL ontology, which captures the information about the structure of underlying sources in form of ontology concepts and relations among them, and stores the mappings of these concepts to the tables of the operational databases.

The method consists of five stages:

- **Requirement validation.** At this stage it is checked whether all the concepts in the business requirement have corresponding concepts in the data sources (ontology). For the concepts, for which the mapping has been found, are tagged with their multidimensional role (Measure, Level, Descriptor).
- **Requirement completion stage.** At this stage the system determines intermediate concepts in the ontology, which are needed to answer the users information queries, however, have not been requested by the user explicitly.
- **Multidimensional validation.** Here the previously defined concepts are tagged as either factual or dimensional, and the schema is validated according to multidimensional design principles.

- **Operation identification** is the stage where the ETL operation needed to transform the data from the data sources and load them to the data warehouse.

The four stages described above are performed for each information requirement separately. The result of the execution of these stages consists of the multidimensional schema of the data warehouse which satisfies the user requirement, and the corresponding ETL operations.

- **Conciliation** is the stage where MD schemas and ETL processes, corresponding to individual requirements, are merged together in order to obtain a single data warehouse design (including MD schema and ETL processes) which would satisfy the complete set of the business requirements.

The approach is semi-automatic, at some stages the system may interact with the data warehouse designer for receiving feedback in cases where there are several alternatives and the decision between them should be made. The work on the GEM framework was continued by Petar Jovanovic in his Master thesis. The work included the implementation of the parts of the GEM system and integrating it with the modules previously developed in the ESSI department, and the result of it was the complete software solution for multidimensional designs (MDIs) based on user requirements and source data information, and providing the ETL processes required for the transformation.

The next step was providing a way to integrate the multidimensional designs for each of the requirements in order to get a single schema for the data warehouse, which would satisfy the entire set of the information requirements. The paper [1] by Petar Jovanovic, Oscar Romero, Alkis Simitsis and Alberto Abelló presents the results of the research in this field and describes a semi-automatic approach which allows to build the unified multidimensional schema. The approach is described in detail in Chapter 3.

Chapter 3

ORE approach

ORE (Ontology-based data warehouse REquirement evolution and integration) is an iterative approach to the design and evolution of multidimensional schemas.

3.1 Overview

The goal of the ORE method is to produce a single multidimensional schema satisfying a set of business requirements by integrating the multidimensional interpretation of individual requirements, which can be produced by different means. In this thesis it will be assumed that the inputs for ORE will be produced by the GEM framework. As a final result ORE and GEM systems must be integrated. The general overview of the architecture is shown on Figure 3.1.

3.2 Inputs

ORE uses the following input data:

3.2.1 Data sources

The knowledge about the data sources is captured in two input assets: **ontology** and **source mappings**. The applicability of ontologies for representing data sources is discussed in [5]. Unlike conceptual formalizations based on entity-relationships (ER) or UML, which are commonly used for this purpose, ontologies provide reasoning mechanisms, that allow to discover relationships between the concepts and facilitate automation. For the GEM framework Web Ontology Language (OWL) was chosen for

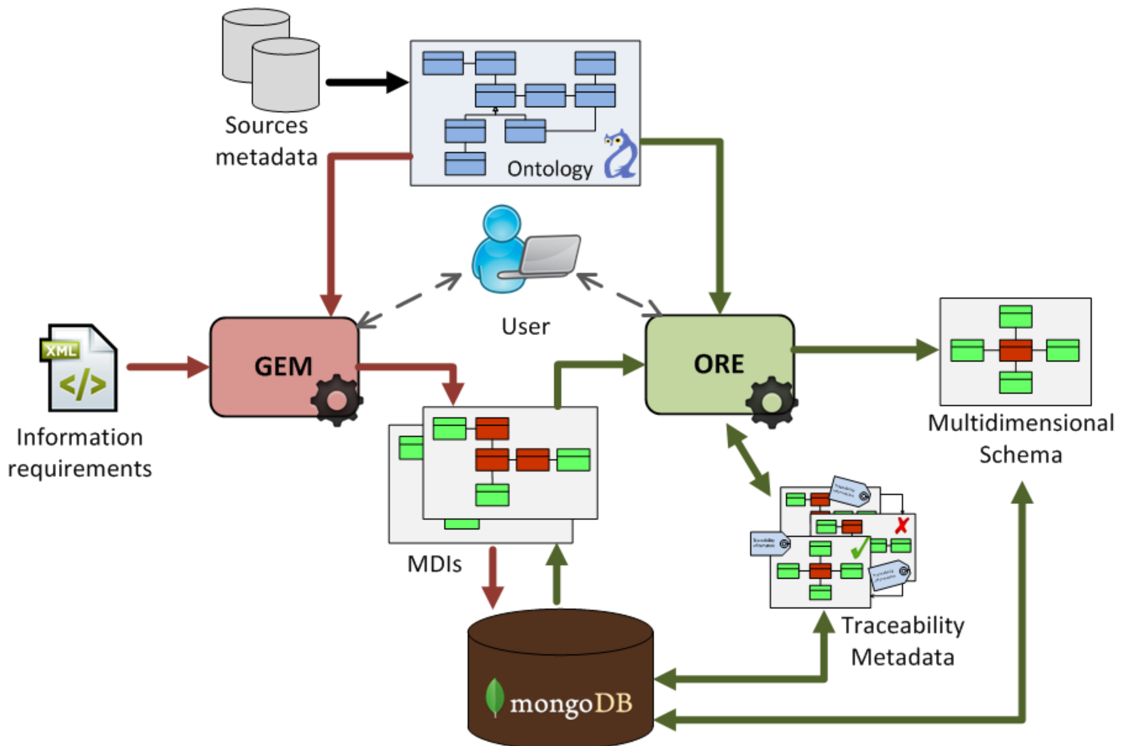


Figure 3.1: Architecture of GEM and ORE

the reasons described in [6]. Briefly, OWL is one of the most commonly used ontology languages and is supported by a number of reasoners (inference engines), also it is recommended by the World Wide Web Consortium (W3C). The structure of the source data stores is represented by ontology classes, their attributes and associations are represented by datatype and object properties respectively. The source mappings are represented in form of XML structure, which stores the information about how the concepts in the ontology relate to the tables of the actual data stores. This data is crucial for performing automatic design of the ETL processes, and both ontology and source mappings are used as inputs in GEM framework to produce MDIs and ETL operations. ORE method doesnt require the information about the physical data stores, it operates over the conceptual model of the data sources, but it is important to keep track of this information, because it will be useful for creating ETL designs of the integrated data warehouse schema.

3.2.2 MDI

Multidimensional interpretation (MDI) is a subset of the ontology, representing the data sources, which captures all the concepts needed to answer a certain information requirement, and is compatible with the multidimensional paradigm. That is, each

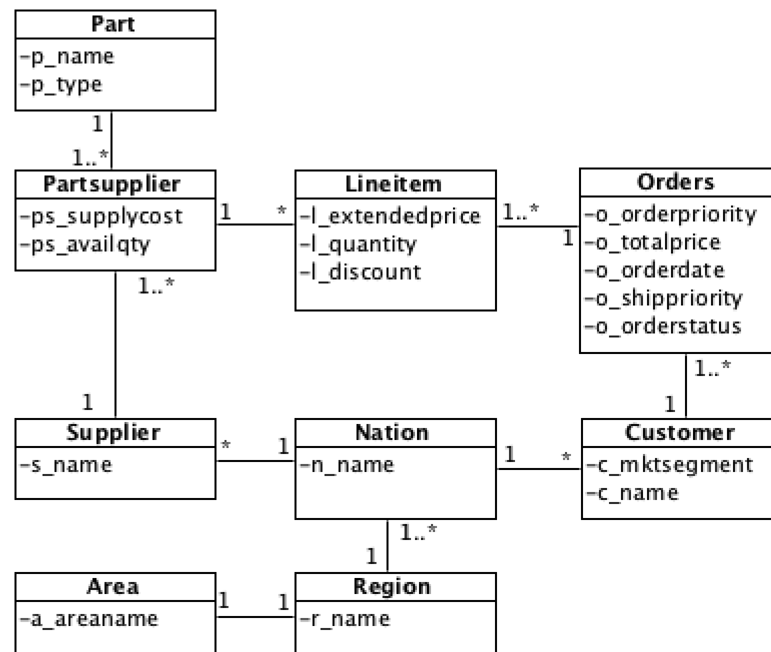


Figure 3.2: Diagrammatic representation of the TPC-H ontology

concept forming part of the MDI is tagged according to the multidimensional role that it plays in the information query: factual or dimensional.

This will be illustrated with an example, based on the TPC-H benchmark. TPC-H [7] is a decision support benchmark, which provides a data set for populating the database and a set of business oriented ad-hoc queries and concurrent data modifications. The ontology, describing the data schema of the TPC-H benchmark is presented on Figure 3.2.

Figure 3.3 shows the multidimentional interpretations (MDIs), corresponding to the following 5 information requirements (IR):

- IR1: The total *quantity* of the *parts* shipped from Spanish *suppliers* to French *customers*.
- IR2: For each *nation*, the *profit* for all supplied parts, *shipped* after 01/01/2011.
- IR3: The total *revenue* of the *parts* supplied from East Europe.
- IR4: For German *suppliers*, the total *available stock value* of supplied *parts*.
- IR5: *Shipping priority* and total potential *revenue* of the parts *ordered* before certain *date* and *shipped after certain date* to a *customer* of a given *segment*.

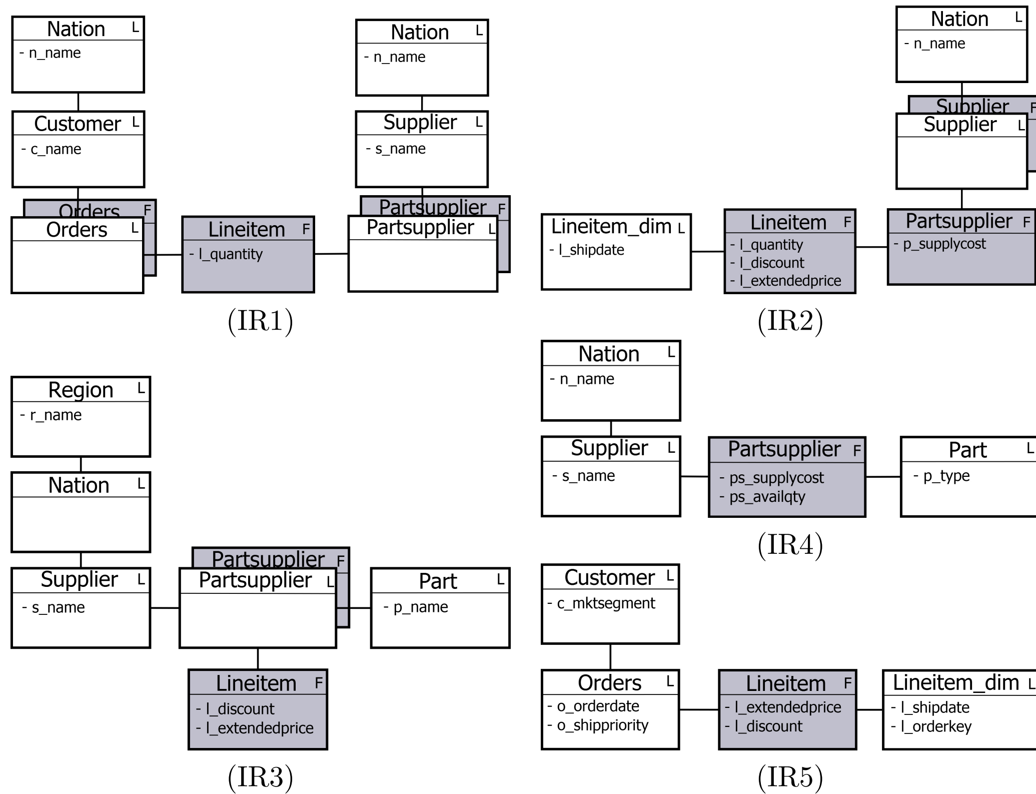


FIGURE 3.3: Single MD interpretations for IR1-IR5

The concepts on the MDI schemas on Figure 3.3 are labeled with their multidimensional role, factual (F) or dimensional (L).

MDIs are produced by the GEM framework, which in its turn receives the following inputs:

- Information requirements previously formalized and represented in XML format;
- Information about the data sources, which is represented in two assets:
  - OWL ontology captures all the concepts and relationships among them.
  - Source mappings store the information about how the data sources, represented as ontology elements, are mapped to real data stores.

GEM process is performed for each requirement separately the result of the process is a set of MDIs and ETL operations.

Apart from the concepts, required explicitly by the users, MDIs can also include additional concepts, which do not contain relevant information, but are needed to answer the queries. These intermediate concepts, as well as the explicitly required ones, are tagged with the multidimensional role that they play in the schema. It is important to remark that such concepts can be either factual or dimensional, therefore one requirement can

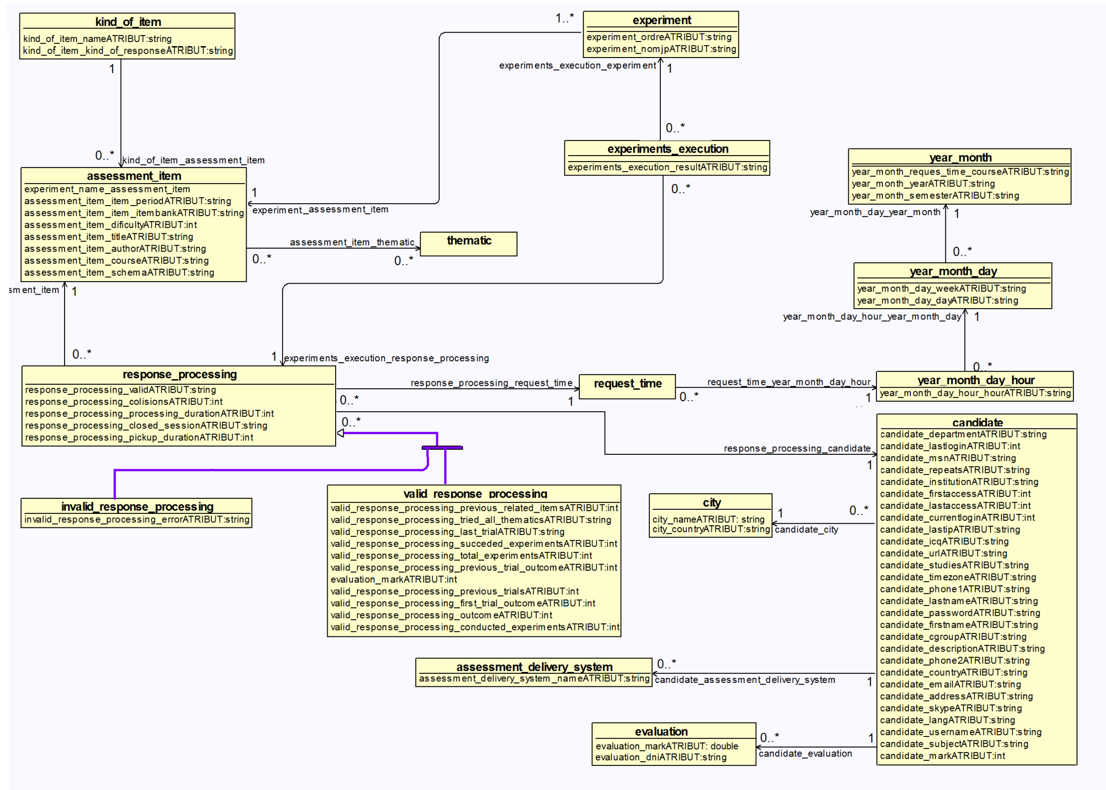


FIGURE 3.4: Diagrammatic representation of the LearnSQL ontology

produce several MDIs, which will differ by the multidimensional role of the intermediate concepts. All MDIs should be considered when building the integrated multidimensional schema, as they can lead to different solutions for the final data warehouse design. The following example of the ontology and MDIs illustrates this.

The ontology shown on Figure 3.4 represents the concepts of the system called LearnSQL an e-learning platform used at UPC for some subjects related to databases and SQL. The main purpose of the system is to provide the means of automatic assessment of exercises for students self-evaluation and also for exams. The system captures the data about exercises, solution submissions made by students, the results of each execution, and the evaluation.

The data stored in the LearnSQL system can be analyzed to extract the information about the students performance in general and regarding certain exercises, which can be very useful for understanding how the students master the learned topics, reveal the difficulties that they encounter and thus improve the contents of the courses. An example of the information requirement for the LearnSQL system can be the following:

*Analyze the average **valid response processing** outcome for each **semester** where the **subject** is DABD.*

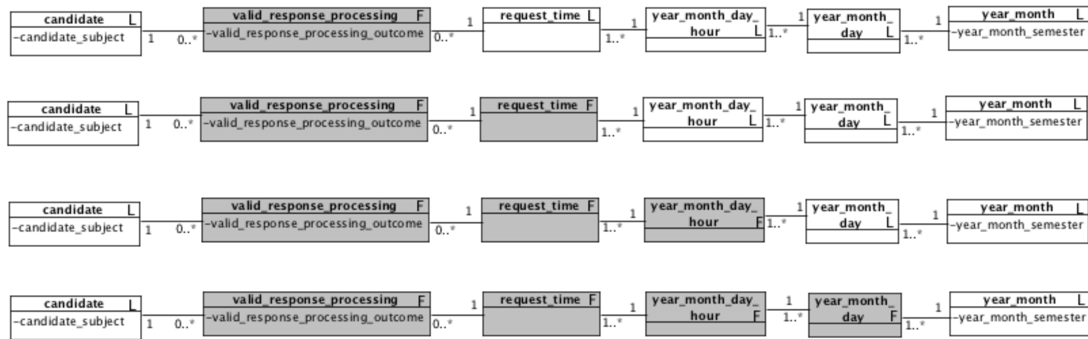


Figure 3.5: Multidimensional interpretations for the requirement

Given this requirement and the ontology GEM produces several multidimensional interpretations, each represent a subset of the concepts of the ontology needed to answer the information query.

Figure 3.5 shows the produced graphs. It can be seen that the concept *Valid response processing* has been identified as a Fact, as it hold the measure *Outcome*, which is the subject of interest. The concept *Year-month* has dimensional role, the attribute *Semester* is a dimension descriptor, over which the factual data will be aggregated. *Candidate* serves as another dimension for the fact *Valid response processing*. It can be seen that the concepts *Request time, Year Month Day Hour and Year Month Day* do not hold any attributes. These concepts have been identified by GEM as intermediate concepts, which are needed to relate the *Valid response processing* fact to its dimension *Year Month*. Thus, these concepts can play both roles in the schema dimensional or factual. By combining these roles in different ways (always controlling the correctness of each schema from the point of view of multidimensional modelling), several MDIs are produced. In this example one requirement results in 4 MDIs (factual concepts are marked with grey color, dimensional concepts are white).

The same can be observed in the examples given for the TPC-H schema (See Figure 3.3). The concepts, which do not hold any attributes (*Orders and Partsupplier* in IR1, *Supplier* in IR2 and *Partsupplier* in IR3) can play either factual or dimensional role in the MDI.

3.3 ORE process

ORE method consists of four stages: Matching facts, Matching dimensions, Complementing the MD design and Integration. A schematic representation of the ORE stages is shown on Figure 3.6.

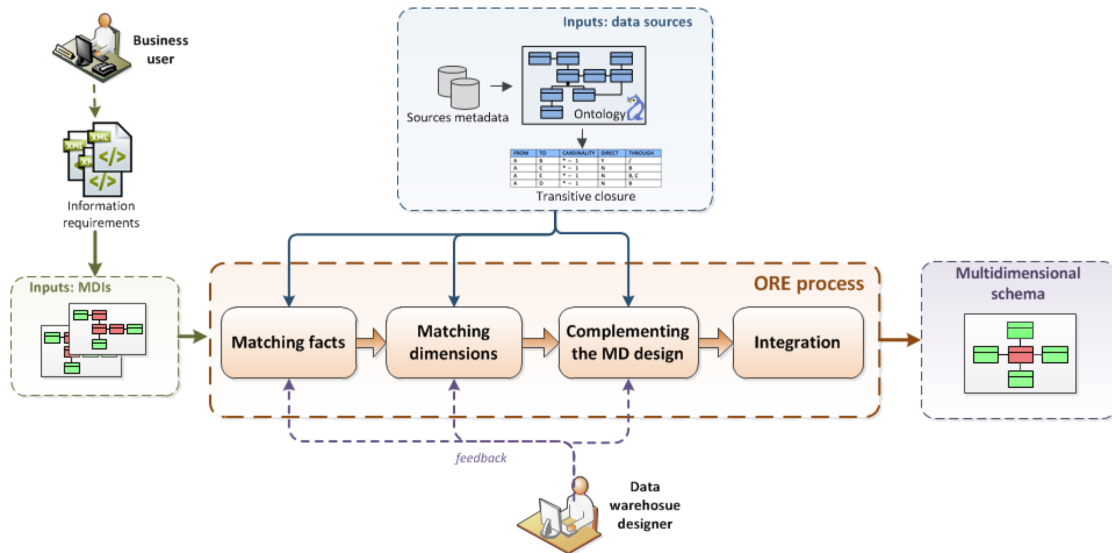


FIGURE 3.6: ORE stages

Given the MDIs and using the data sources ontology, ORE first matches factual and dimensional concepts appearing in different requirements, interacting with the data warehouse designer in cases where several matching options are available. At the third stage ORE explores different design alternatives, in attempt to enrich the schema; again, the designer feedback is considered. The last stage performs the integration of the MDIs, considering the designers feedback, and produces the final multidimensional schema which satisfies all the requirements received at the input. The output of the ORE method is a multidimensional schema (MD schema), which consists of one or more star schemas, which answer one or more information requirements. Such star schema represents a superset of nodes and edges of the MDIs that form part of it. The algorithm for the ORE process was presented formally in [1] and is shown on Figure 3.7.

For this and the following algorithms used in this document, the following notation will be used:

- IR – information requirement
- MDI – multidimensional interpretation
- $\mathbb{MDI}$ – set of MDIs
- SS – star schema
- $\mathbb{S}$ – space of solutions consisting of alternative MD schemas
- $\mathbb{SS}$ – MD schema (set of star schemas)
- F – fact
- D – dimension

For referring to the steps of the algorithms the notation (Step algorithm:step) will be used.

Algorithm: ORE

inputs: IR_{new}, $\mathbb{S}$, **output:** $\mathbb{S}_{new}$

1. *options* $:= \emptyset$;
2. **For each** $\mathbb{SS}_{cur} \in \mathbb{S}$ **do**
   (a) **For each** $[MDI_i \in IR_{new}, SS_j \in \mathbb{SS}_{cur}]$ **do**
      i. *matchedFactsOpers* := FM(getFact(MDI_i),getFact(SS_j));
      ii. **If** *matchedFactsOpers* $\neq \{insertFact(F_{MDI_i})\}$ **do**
         A. $\mathbb{D}_{MDI_i}$:= searchDimsOverFact(F_{MDI_i});
         B. $\mathbb{D}_{SS_j}$:= searchDimsOverFact(F_{SS_j});
         C. *matchedDimsOpers* $\cup =$ $DM(\bigcup_{D_{MDI_i} \in \mathbb{D}_{MDI_i}} bottom(D_{MDI_i}), \bigcup_{D_{SS_j} \in \mathbb{D}_{SS_j}} bottom(D_{SS_j}))$;
         D. *options*$\cup =$ [$\mathbb{SS}_{cur} \setminus SS_j, SS_j$,matchedFactsOpers,matchedDimsOpers];
3. $\mathbb{S}_{new} := INT(complementingMDSchema(applyOperations(findTop(options))))$;
4. **For each** $o \in findBestN(options)$ **do**
   (a) $\mathbb{S}_{new} \cup = applyOperations(o)$;
5. **return** $\mathbb{S}_{new}$;

Figure 3.7: ORE algorithm

The algorithm receives a new information requirement which needs to be satisfied (IR_{new}) and the current set of alternative solutions for the data warehouse design ($\mathbb{S}$). The algorithm explores all the solutions, at the same time considering all the MDIs of the requirement, searching for valid correspondences among the multidimensional concepts of MDIs and the stars (SS), composing the MD schemas. In this way, ORE considers all possible combinations MDI-SS and evaluates the produced solutions based on their overall cost (the evaluation method is described further). For each MDI (MDI_i) and star schema (SS_j) of the solution, ORE first tries to find matchings between the factual concepts (Step ORE:2(a)i), and if the matching is found, ORE proceeds to conform the dimensions (Step ORE:2(a)iiC). While performing the matching, ORE identifies and collects the operations which need to be executed over the MDI concepts in order to integrate them into the existing MD design (*matchedFactsOpers* and *matchedDimsOpers*). In some cases there are more than one alternative ways to match the concepts and integrate MDI, so several sets of integration operations are considered by ORE as integration options. After that at the Complementing design stage ORE further explores the domain ontology in order to find new analytical perspectives and integrate them into the MD design. By applying the operations collected at previous stages over the existing schemas, ORE produces the new space of alternative solutions (Steps ORE:3 and ORE:4), and for each solution its cost is calculated using the selected cost model. Then the solution space can be pruned, keeping the best N solutions based on their overall costs. The user can provide his feedback by specifying the value of N or also by discarding some solutions manually. The best solution among these N is further used

by ORE to produce the final MD schema of the data warehouse.

Pruning the solution space is an important part of the process, as the number of valid solutions is growing fast, due to the fact that each requirement can have several multidimensional interpretations, and all of them should be considered. ORE should process all the combinations of MDIs of the input requirements, thus obtaining the number of solutions equal to $\prod_{k=1}^{n} \#MDI_k$, where n is the number of requirements, and $\#MDI_k$ is the number of multidimensional interpretation for the k-th requirement. For example, in case we have M requirements, and each of them has N MDIs, with each integrated requirement the solution space would grow exponentially, and in the end different solutions would be produced. For requirements having 4 MDIs, the number of solutions would reach 1024 after the 5th integrated requirement. Also, at the matching stages several alternatives could be identified, and thus produce even more alternative solutions. In order to reduce the solution space, the user can manually select a threshold N, which will instruct the system to keep only the N best solutions, and discarding the rest. The ORE algorithm repeats for each new requirement appearing at the input of the system. The following sections describe each stage of the ORE method in detail.

3.4 Integration operations

Integration operations allow modifying the MD schema by adding new concepts or attributes, making it able to satisfy new requirements. Operations can be designed in different ways. One of the examples of operations for MD schema evolution is presented in [8]. The authors propose the following operations for MD schema evolution: insert level, delete level, insert attribute, delete attribute, connect attribute to dimension level, disconnect attribute from dimension level, connect attribute to fact, disconnect attribute from fact, insert classification relationship, delete classification relationship, insert fact, delete fact, insert dimension into fact, delete dimension. These atomic operations can be grouped into more complex operations. ORE approach defines a different set of integration operations, although some similarities can be found. The main difference is that ORE introduces operations for merging the concepts from the MDIs coming from new information requirements and the existing MD design. All the integration operations proposed by ORE method can be divided into three groups:

1. operations for inserting new concepts into the existing design:
   - **insertFact** operation adds a new fact to the MD schema. Practically it means that a new star is introduced into the design.

- **insertLevel** introduces a new dimension level to the existing fact. It can be either a concept representing a new dimension, or a new level in a hierarchy of an already existing dimension. Thus, if compared with [8] approach, this operation can act similar to insert level or insert dimension to fact.
- **insertRollUp** operation complements the insertLevel in case when the inserted concept adds a new level in the existing dimensional hierarchy. The roll-up relation between the two levels is introduced, which allows the user to query the data at different levels of granularity (similar to insert classification relationship operation from [8]).

2. operations for enriching the existing conceptsk with new attributes:

   - **insertFactMeasure** operation adds a new attribute to the existing factual concept.
   - **insertDimDescriptor** opearation adds a new attribute to the existing dimensional concept. Unlike the insert attribute operation in [8], insertFactMeasure and insertDimDescriptor operations are self-sufficient, and include both adding the attribute and relating it to the appropriate dimensional or factual concept.

3. operations for combining new MD concepts with the existing ones:

   - **MergeFacts** operation performs the merging of the facts from the existing design and the new MDIs when the matching is found between them, thus obtaining a single fact which holds all the attributes of both merged facts.
   - **MergeLevels** operation merges two matching dimensional concepts.
   - **rollupFacts** operation is applied when there is a functional dependency between two facts, and the MD space of one fact can be rolled-up to the MD space of the other.
   - **renameConcept** operation can be identified when the matching is found between the concepts, but they are not equal.

In order to merge an MDI of a new requirement into the existing MD design a set of operations to be executed is required. These operations are first identified by Fact Matching and Dimension Matching algorithms of ORE, and later applied to the MD schema, which makes it satisfy the new requirement, as well as all the previously integrated ones.

3.5 Cost model

When new requirements are introduced, several alternative ways to integrate them into the existing design may appear. This is mainly caused by the fact that requirements may have several MD interpretations, and all of them are considered. In order to select one MD design from the set of available options, some metrics should be used to measure data warehouse quality factors, such as understandability, analizability, maintainability etc., which are influenced by the structural characteristics of the schema, such as number of factual and dimensional concepts, the number of attributes etc. In order to evaluate the alternative solutions and assist user in selecting the best option, ORE method proposes a cost-based approach: certain weight is assigned to each integration operation (discussed in Section 3.4) and accumulated cost of incorporating new requirements into the existing schema is calculated. The solutions with lower cost are prioritized, Table 3.1 lists the weights which are used by the ORE approach. These weights represent, how changes in the MD schema affect the quality factors. The values are based on the results of studies on DW quality factors, in which the quality metrics are empirically validated. The results of one of such studies are presented in [9].

TABLE 3.1: Integration operations

DW concept	Operation name	Weight
Dimensional	insertLevel	0,21
	insertRollUpRelation	0,27
	insertDimDescriptor	0,04
	MergeLevels	0,04 × #insertDimDescriptor
Factual	insertFact	0,31
	insertFactMeasure	0,36
	MergeFacts	0,36 × #insertFactMeasure
Factual/Dimensional	renameConcept	0

3.6 Stage 1: Matching Facts

The first stage of the ORE process searches for matching between the MDI of the current requirement and the existing MD schema. The matching is found when the two facts produce a compatible set of points in the MD space. The condition for that can be expressed formally as follows: The fact $F_{MDI_i} \in MDI_i$ matches the fact $F_{SS_j} \in SS_j$ if there is a bijective function f such that for each point x_{MDI_i} in the MD space arranged by the dimensions $\{D_1, D_2, .., D_n\}$ implied by F_{MDI_i}, there is one and only one point y_{SS_j} in the MD space arranged by the dimensions $\{D'_1, D'_2, .., D'_m\}$ implied by F_{SS_j}, such that $f(x_{MDI_i}) = y_{SS_j}$. In other words, the matching is found when F_{MDI_i} (F_{SS_j}) functionally determines all the dimensions of F_{SS_j} (F_{MDI_i}), i.e., when F_{MDI_i} (F_{SS_j}) is

Algorithm: FM

inputs: F_{MDI_i}, F_{SS_j}, **output:** *intOps*

1. **If** $F_{MDI_i} == F_{SS_j}$ **then** *intOps* := $\{mergeFacts(F_{MDI_i}, F_{SS_j})\}$;
2. **ElseIf** $F_{MDI_i} \rightarrow F_{SS_j} \vee F_{SS_j} \rightarrow F_{MDI_i}$ **then**
   (a) **If** $F_{MDI_i} \rightarrow F_{SS_j} \wedge F_{SS_j} \rightarrow F_{MDI_i}$ **then**
   intOps := $\{mergeFacts(F_{MDI_i}, F_{SS_j}), renameConcept(F_{MDI_i}, F_{SS_j})\}$;
   (b) **ElseIf** $F_{MDI_i} \rightarrow F_{SS_j}$ **then**
      i. **If** *acceptableGranularity*(MDI_i) **then** *intOps* := $\{rollupFacts(F_{MDI_i}, F_{SS_j})\}$;
   (c) **Else** // $F_{SS_j} \rightarrow F_{MDI_i}$
      i. **If** *acceptableGranularity*(SS_j) **then** *intOps* := $\{rollupFacts(F_{SS_j}, F_{MDI_i})\}$;
3. **Else**
   (a) $\mathbb{D}_{MDI_i}$:= searchDimsOverFact(F_{MDI_i});
   (b) $\mathbb{D}_{SS_j}$:= searchDimsOverFact(F_{SS_j});
   (c) **If** $F_{MDI_i} \rightarrow \mathbb{D}_{SS_j} \vee F_{SS_j} \rightarrow \mathbb{D}_{MDI_i}$ **then**
      i. **If** $F_{MDI_i} \rightarrow \mathbb{D}_{SS_j} \wedge F_{SS_j} \rightarrow \mathbb{D}_{MDI_i}$ **then**
      intOps := $\{mergeFacts(F_{MDI_i}, F_{SS_j}), renameConcept(F_{MDI_i}, F_{SS_j})\}$;
      ii. **ElseIf** $F_{MDI_i} \rightarrow \mathbb{D}_{SS_j}$ **then**
         A. **If** *acceptableGranularity*(MDI_i) **then** *intOps* := $\{rollupFacts(F_{SS_j}, F_{MDI_i})\}$;
      iii. **Else** // $F_{SS_j} \rightarrow \mathbb{D}_{MDI_i}$
         A. **If** *acceptableGranularity*(SS_j) **then** *intOps* := $\{rollupFacts(F_{MDI_i}, F_{SS_j})\}$;
   (d) **Else** *intOps* := $\{insertFact(F_{MDI_i})\}$;
4. **return** *intOps*;

related by means of "1 – 1" or "* – 1" relationship to each dimension of F_{SS_j} (F_{MDI_i}). The Fact Matching (FM) algorithm checks in several steps if this condition is satisfied, and collects the required integration operations.

First step (Step FM:1) of the algorithm checks whether the facts are equal (the full match is found between them). If the equality has been found, the *mergeFacts* operation is added, otherwise the algorithm proceeds to search for functional dependencies between the facts (Step FM:2). There can be a situation when F_{MDI_i} functionally determines F_{SS_j}, and at the same time F_{SS_j} functionally determines F_{MDI_i}, i.e. there is a "1 – 1" relationship between the facts (Step FM:2a), or the facts are *synonyms*. In this case *mergeFacts* operation is added, and it is complemented with a *renameConcept* operation. Alternatively, only one fact functionally determines the other (Step FM:2(b)i, Step FM:2(b)i), which means that there is a "1 – *" relationship between the facts. In this case the facts can be merged with *rollupFacts* operation, which rolls-up from the MD space of one fact to the MD space of the other; however it is only possible in case the coarser granularity is acceptable for all the requirements associated with the current MD schema. If no matching has been found between the facts, the FM algorithm follows to check the matching condition by searching for the functional dependencies between the fact in the new requirement (F_{MDI_i}) and the dimensions of the fact of the

current schema (F_{SS_j}), and vice versa (Step FM:3). The situations which can appear at Step FM:3c are identical to those dealt with at Step FM:2, but in this case they refer to the dimensions. Thus, when both facts functionally determine all the dimensions of the other fact (Step FM:3(c)i), the *mergeFacts* and *renameConcept* operations are added. If only one of the facts functionally determines the dimensions of the other (Step FM:3(c)ii and Step FM:3(c)iii), the *rollupFacts* operation is used to merge them. Again, in this case it should first be checked that the new granularity is acceptable for all the requirements.

The FM algorithm can be applied directly when the facts F_{SS_j} and F_{MDI_i} are represented by just one factual concept. However, as in case of multidimensional interpretation a fact can be represented as several related factual concept, a clarification on how such facts must be matched is required.

Figure 3.8 presents several examples of possible relationships between the concepts composing the facts being compared. In the example shown on Figure 3.8a equality is found between the concepts C1 in F_{MDI} and F_{SS}, the same applies to the concept C2; and the relationship C1 – C2 is identical in both schemas. This allows us to conclude that the facts F_{MDI} and F_{SS} are equal, in the FM algorithm this condition corresponds to the Step FM:1. It should be noted that the type of the relationship C1 – C2 doesn't make difference, the important is that the multiplicities of the relationships C1 – C2 coincide on both ends.

Figure 3.8b shows a slightly modified case, where both facts have additional concepts (C3 and C4), which do not have any matching concept in the other fact. C3 and C4 are related by means of "1 – 1" relationships to the concept C1 in F_{MDI} and C2 in F_{SS} respectively. These additional concepts, being synonyms, do not change the multidimensional space, so the facts can still be matched. However, as the facts don't coincide completely, they can't be considered equal, instead we consider that the two facts are synonyms. Besides, when creating the merging operation, these additional concepts must be taken into account, and the attributes of the concept C3 in F_{MDI} must be transferred to the integrated schema.

Figure 3.8c shows a more general case. When matching is found between the concepts C1 in F_{MDI} and F_{SS}, and C1 in F_{MDI} has other related concepts, the relationship C1 – C2 must be analysed. The case when C1 and C2 are related by means of "1 – 1" relationship, corresponds to the Step FM:2a – the facts are merged like synonyms. The same happens in the example shown on Figure 3.8d: if C1 and C3 are synonyms, and C1 is related to C2 by means of "1 – 1" relationship, the facts are considered synonyms. When C1 – C2 is a "1 – *" or "* – 1" relationship, the algorithm follows the steps Step FM:3(c)iii or Step FM:3(c)ii correspondingly in both examples — 3.8c and 3.8d.

The case shown on Figure 3.8e is one of the most complex. Here the relationship C1 – C2 is important. In case it is "1 – 1" the algorithm would produce a roll-up operation at Step FM:3(c)ii. However, if C1 – C2 is "1 – *" or "* – 1" a special attention is required to check the granularities of both facts and whether the integration is possible.

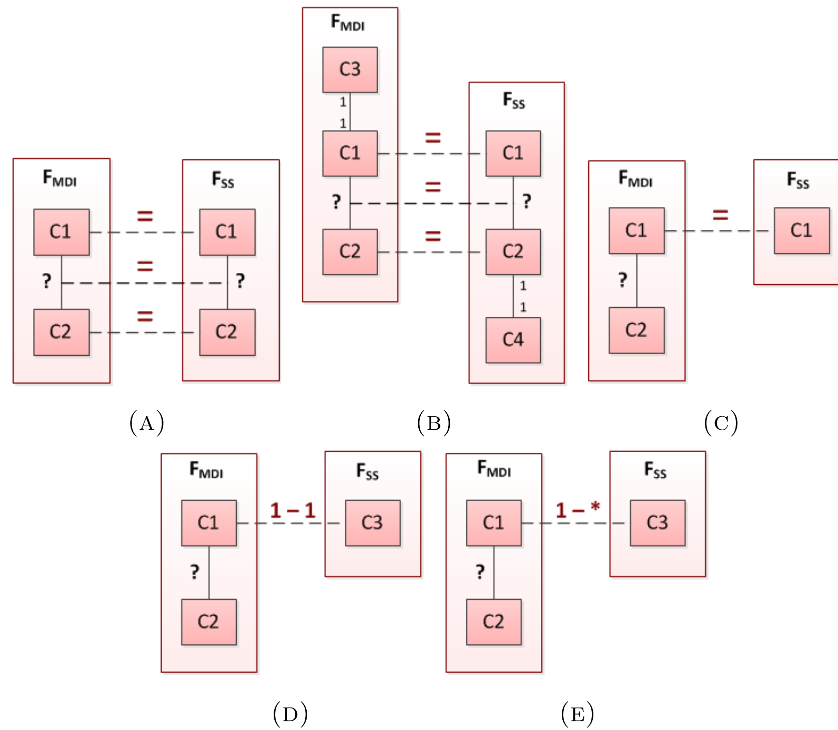


FIGURE 3.8: Examples of fact matching

3.7 Stage 2: Matching Dimensions

In the Matching Dimensions stage ORE conforms the dimensions of the facts F_{MDI_i} and F_{S_j}, for which matchings were found in the previous, fact matching stage. As it was previously discussed in 2.2.1, a dimension can be represented as directed acyclic graph. Therefore, the problem of finding the matching dimensions can be seen as a graph matching problem. This is a combinatorial problem that can be computationally expensive, but having in mind the special character of relationships among the dimensional concepts, a more specific method can be applied for sovling the problem. ORE approach proposes the DM algorithm which solves this problem in the context of dimensions.

DM algorithm is launched from the ORE algorithm (Step ORE:2(a)iiC) for the dimensions of the facts coming from MDI_i (F_{MDI_i}) and from SS_j (F_{SS_j}), which were previously matched by the FM algorithm.

Algorithm: DM

inputs: $Candidates_{MDI_i}$, $Candidates_{SS_j}$, **output:** *intOps*

1. **For each** $[L_{MDI_i} \in Candidates_{MDI_i}, L_{SS_j} \in Candidates_{SS_j}]$ **do**
   (a) **If** $L_{MDI_i} == L_{SS_j}$ **then**
   $intOps \cup = \{mergeLevels(L_{MDI_i}, L_{SS_j})\} \uplus DM(getNext(L_{MDI_i}), getNext(L_{SS_j}))$;
   (b) **ElseIf** $L_{MDI_i} \rightarrow L_{SS_j} \vee L_{SS_j} \rightarrow L_{MDI_i}$ **then**
      i. **If** $L_{MDI_i} \rightarrow L_{SS_j} \wedge L_{SS_j} \rightarrow L_{MDI_i}$ **then**
      $intOps \cup = \{mergeLevels(L_{MDI_i}, L_{SS_j}), renameConcept(L_{MDI_i}, L_{SS_j})\} \uplus DM(getNext(L_{MDI_i}), getNext(L_{SS_j}))$;
      ii. **ElseIf** $L_{SS_j} \rightarrow L_{MDI_i}$ **then**
      $intOps \cup = \{insertRollUp(L_{SS_j}, L_{MDI_i})\} \uplus DM(\{L_{MDI_i}\}, getNext(L_{SS_j}))$;
      iii. **Else** // $L_{MDI_i} \rightarrow L_{SS_j}$
      $intOps \cup = \{insertRollUp(L_{MDI_i}, L_{SS_j})\} \uplus DM(getNext(L_{MDI_i}), \{L_{SS_j}\})$;
   (c) **Else** // No matching for current levels
      i. $intOps \cup = DM(\{L_{MDI_i}\}, getNext(L_{SS_j}))$;
      ii. $intOps \cup = \{insertLevel(L_{MDI_i})\} \uplus DM(getNext(L_{MDI_i}), \{L_{SS_j}\})$;
2. **return** *intOps*;

DM tries to match the sets of all the dimensions of MDI_i and SS_j ($\mathbb{D}_{MDI_i}$ and $\mathbb{D}_{SS_j}$ correspondingly), starting from the atomic levels of these dimensions, and recursively moving forward through the corresponding hierarchy, searching for integration options and storing the information about the required integration operations (according to the Table 3.1). The levels which are currently in scope of the algorithm are called *candidate levels*. In each recursive call DM searches for matchings between all pairs of the candidate levels of MDI_i ($candidates_{MDI_i}$) and each candidate from the dimensions of SS_j ($candidates_{SS_j}$). For each pair DM first checks whether they exactly coincide (Step DM: 1a. If the matching is not found, DM searches for the functional dependencies between the levels (Step DM: 1b). There can be the case when both levels functionally determine each other, i.e. there is a "1 – 1" relationship between them (Step DM: 1(b)i), or alternatively, only one level functionally determines the other one, i.e. between L_{MDI_i} and L_{SS_j} there is either "1 – *" relationship (Step DM:1(b)ii) or "* – 1" relationship (Step DM:1(b)iii).

If the levels are equal or the "1 – 1" relationship is found, DM creates a *mergeLevels* operation (along with a *renameConcept* operation in case of "1 – 1" relationship) and makes a recursive call, moving forward in both hierarchies. But in the cases when a "1 – *" or a "* – 1" relationship is identified, there can still be a possibility that the level on the to-one side of the relationship (L_1) matches the next level in the hierarchy of the level on the to-many side (L_*), which means a roll-up relationship can be created between L_* and L_1. In this case DM creates a *insertRollUp* operation and moves forward only in the hierarchy of the level L_*.

Finally, if no relationship is found between the levels (Step DM:1c), DM explores

both hierarchies by moving forward in first in the hierarchy of L_{SS_j} level, and then in the hierarchy of L_{MDI-is}, searching for matchings and building possible integration options.

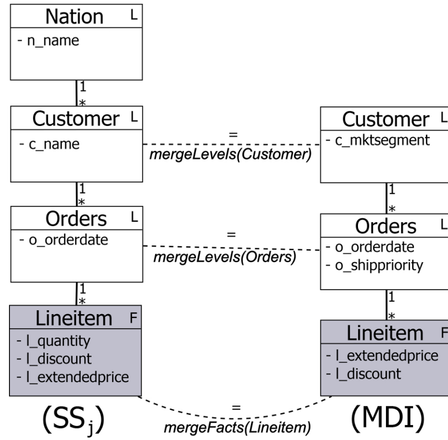


FIGURE 3.9: Matching the MD Interpretation for IR2

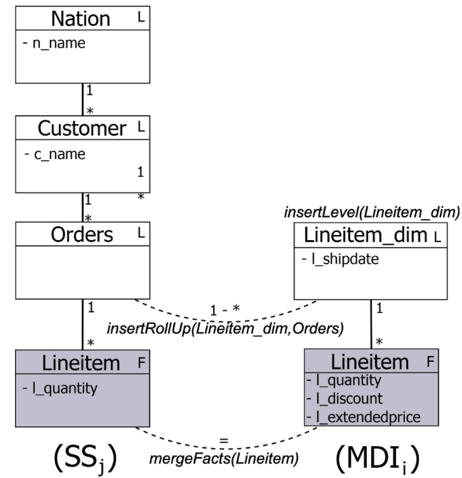


FIGURE 3.10: Matching facts with *rollupFacts*

Example 1. Figure 3.9 shows the case of integrating IR5 into the TM satisfying IR1-IR4. Here equality is found for both `Orders` and `Customer` levels of the corresponding MDI_i. Thus, the *mergeLevels* operations are proposed for both `Orders` and `Customer` levels and they respectively involve *insertDimDescriptor* operations for transfering `o_shippriority` and `c_mktsegment` descriptors.

Example 2. In Figure 3.10 there is a "* – 1" relationship between the levels `Lineitem_dim` and `Orders`, therefore a `insertRollUp(Lineitem_dim, Orders)` operation is proposed by DM, along with the insertion of the level `Lineitem_dim`.

3.8 Stage 3: Complementing the MD design

The result of the execution of the previous ORE stages is a space of alternative solutions for incorporating the coming information requirements (represented as MDI) into the existing multidimensional schema. The solutions include the set of operations to be performed over the existing schema so that it satisfies the new requirement. Based on the cost of these operations, and considering the user preferences, this solutions can be ordered. Thus, the solution on the top of such ordered space would be the most preferable.

The purpose of this stage, complementing the MD design, is to enrich the resulting schema with new analytically interesting concepts. This process requires the exploration

of the ontology and is quite expensive, therefore it makes sense to perform it only for the top solution in the solution space. Besides, this stage is optional and it may be disregarded by the user.

In this stage, for the top schema ORE explores the ontology in search of the concepts which might complement the current solution and provide the user with more analytical perspectives. The approach used for complementing includes the following steps:

- Analyze the ontology and search for properties which may serve as additional attributes for the existing facts and levels in the schema, taking into consideration the following:
  - Properties that have a numerical data type (e.g., *integer*, *double*), are considered as measures of the corresponding conceptual concepts.
  - If the domain concept of a property is identified as a level, the property is used as a new dimension descriptor. In these cases ORE identifies the operations *insertFactMeasure insertDimDescriptor*.
- Explore the functional dependencies ("to-one" relationships) in the ontology in order to identify new levels for the previously conformed dimensions. The integration operation *insertLevel* and *insertRollupRelation* may be proposed by ORE in this case.

Complementing the MD design is a delicate task, as it has an opposite effect on the resulting MD schema compared to the other stages. While the first two ORE stages try to minimize the schema, and include only those concepts and properties which are necessary to answer the integrated requirements, the complementing stage increases the complexity of the schema. However, such enhanced schema might open new analytical perspectives to the user, and in the future when new information requirements appear, it may happen that they will be already covered by the current schema. Anyway, it is important to mention that ORE can only suggest the ways to enrich the schema, the decision on whether to do it or not lies on the designer of the data warehouse.

3.9 Stage 4: Integration

The solution, selected as the best and optionally complemented in the previous stage, produces the schema, which is able to solve all the information requirements integrated until now. However, as we have seen in 3.2.2, the MDI, and thus the resulting schema may contain concepts, which do not hold any attributes. The ORE integration stage

Algorithm: INT

input: $\mathbb{SS}_{top}$, **output:** $\mathbb{SS}_{new}$

1. $SS_{new} := \emptyset$;
2. **For each** $SS_j \in SS_{top}$ **do**
   (a) $SS_{new} := [\emptyset, \emptyset, \emptyset]$;
   (b) $seedF :=$ findFactualConcept(SS_j);
   (c) $\mathbb{F} :=$ group($seedF$);
   (d) setFact(SS_{new},collapse($\mathbb{F}$));
   (e) **For each** $(f, L_0) \in F_{SS_j} \wedge f \in \mathbb{F} \wedge L_0 \notin \mathbb{F}$ **do**
      i. $\mathbb{D}$ = group(L_0);
      ii. setDimension(SS_{new},collapse($\mathbb{D}$));
   (f) $\mathbb{SS}_{new} \cup = \{SS_{new}\}$;
3. **return** $\mathbb{SS}_{new}$;

FIGURE 3.11: INT algorithm

is intended to increase the quality of the final schema according to such quality factors as structural complexity, understandability, analyzability, maintainability and others. For example, the structural complexity of the schema can be decreased by collapsing the adjacent levels, which simplifies the dimensions and lowers the number of roll-up relationships.

The Integration algorithm proposed in ORE method is presented on Figure 3.11. The process includes two phases:

1. **Grouping.** As the ontological concepts can be represented by a directed acyclic graph (DAG), these are combined together to produce different groups (subgraphs), so that all those in one group:
   (a) produce a connected subgraph and
   (b) have the same MD interpretation (i.e., all concepts are either factual or dimensional).
2. **Collapsing.** Starting from these groups of concepts we obtain the final star schema. Inside each subgraph captured by a single group, only the concepts currently required by the user are considered, either provided with the requirement at hand or discovered when complementing the MD design in the ontology. The concepts considered inside each group are then collapsed to produce one element (i.e., fact or dimension) of the final MD schema.

Integration stage relaxes the final schema from the irrelevant knowledge and hides the unnecessary concepts, but the TM structure should still keep all the knowledge about

the complete schema, containing all the concepts coming from the integrated MDIs and the mappings of these concepts to the original data sources. This is important for the future integration, and also for generating the ETL processes for the new DW design.

Chapter 4

Development

4.1 Development process

4.1.1 Overview of the existing process models

The implementation of the ORE method is a software development project. According to the best practices in software engineering, it is important for the project to follow some development process model, or, more generally, software development methodology – a framework used to structure, plan, and control the process of the development. Following a known methodology facilitates the planning, makes the process more transparent, which is very important when more than one person is involved in the project, makes the project results more predictable, and assists in ensuring the quality of the resulting product. There is a variety of software development methodologies, each having its strong and weak points. When choosing a methodology, the characteristics of the concrete project must be considered in order to find the one which would suit best in the given case.

Roughly, the approaches can be divided into two groups by the process model: "waterfall" and iterative model. *"Waterfall"* methodology was one of the first methodologies of software development, and it was popular during several decades. The main idea is that the development consists of several sequential phases, usually the following: requirements definition, design, implementation, testing, and maintenance. It assumes that all the requirements can be known and understood at the beginning of the development, and that they do not change later on. While this might work for some kinds of projects, the experience of the last decade shows that it doesn't give very good results [10]. Another group of methodologies, opposing the "waterfall" model, use so called *iterative approach*. Its main principle is that the development is elaborated in cycles

(iterations), each iteration includes the phases, similar to the ones of the “waterfall” model – requirements definition or refinement, design, testing, implementation, testing, but the difference is that these phases repeat several times, in each iteration. One of the examples of well-known frameworks based on iterative approach is the Rational Unified Proccess (RUP). It defines not only the process model, but also artifacts and roles of the software development project. RUP is a complex and elaborated methodology, and suits well for big organizations, however for small projects it is usually too heavyweight. An alternative to these approaches is Agile software development, which strictly speaking is not a methodology, but rather a philosophy, as it doesn’t define exact methods and practices, but describes the general principles.

4.1.2 Agile software development

The main principles of the Agile software development were published in 2001 as the Manifesto for Agile Software development [11]. According to the Manifesto the value is given to:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

This means that while the things stated on the right are important, the ones from the left side of each assertion have the preference.

Figure 4.1 shows the illustration of the agile process method. It can be seen that while there are several levels of the process: strategy level, release level, iteration, daily work – all of them are cyclic, and independently of the level, the result is always working software.

There is a number of methodologies which follow these principles, thus they can be called “agile methodologies”. The most well-known and widely used methodologies are Scrum and eXtreme programming (XP).

4.1.3 Process model chosen for the project

For the development of the current project it was decided to follow the agile principles. It suits well for the purpose of this project, as, being a research project, it is characterized

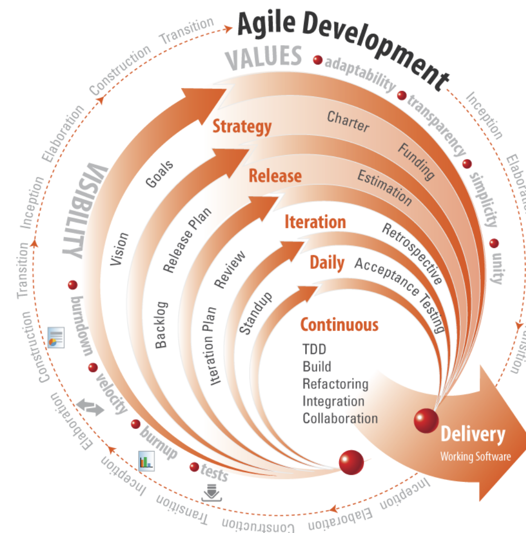


FIGURE 4.1: Agile software development

by uncertainty and high probability of changes in the requirements, or design, or the technologies used. Agile provides necessary flexibility to face these challenges, and being a "lightweight" approach, it allows to reduce the overhead of unnecessary documentation and complex processes.

The main principles of Agile development fit perfectly for the project of such kind. Individuals and interactions in this case have indeed more value than processes and tools, as, having a small number of people involved in the project, complex processes can just complicate the communication. The preference for working software over comprehensive documentation is also suitable for this master project. The main objective of the project is the implementation of the designed method, and demonstration that the method works as expected. The documentation is very important too in this case, firstly for the project report (this document), and secondly, for the people who will use the developed software and also those who will continue working on it by adding new features. However, this documentation is not as important as the software itself. Contract negotiation is not applied in this case, as there is no real customer, which we are related to with contractual commitment. The tutors act as customers, defining the requirements and accepting the work done. And indeed the collaboration with them is a very important part of the project, to ensure that the software will be useful. Finally, the more value given to responding to change in comparison to following a plan also applies for the current project, because, as it was mentioned before, due to the research character of

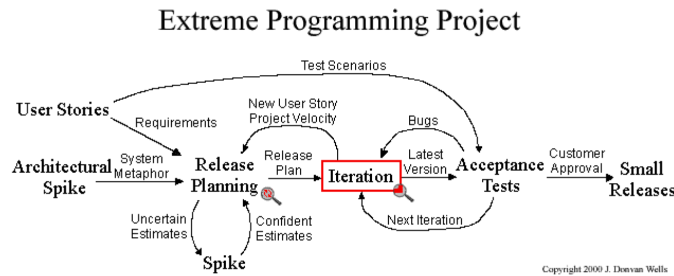


FIGURE 4.2: Extreme programming project

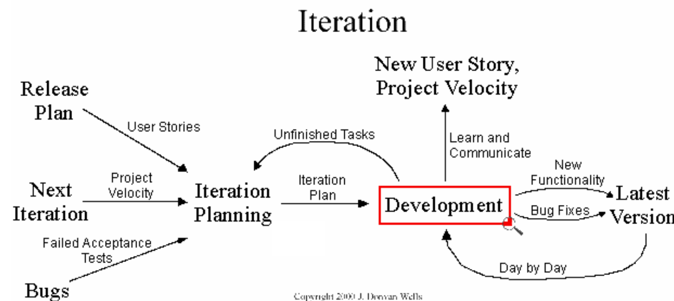


FIGURE 4.3: Extreme programming iteration

the project, it should be open for changes at any stage and allow to step back and try a different way in design, implementation or the used technologies.

After analysing Scrum [12] and XP [13] agile methodologies, I came to a conclusion that while both approaches can be applied to the current project, neither of them can be used in their entirety, and some adaptation is required. I found extreme programming methodology to be more easily adaptable.

Scrum uses quite specific roles for the team members, such as Scrum master (a role responsible for ensuring that the project is carried through according to the Scrum practices) and Product owner (responsible for maximizing the value of the product and the work of the development team), for which it was difficult to select one person. And also Scrum method has specific events, such as Daily Scrum – a short meeting of the development team the purpose of which is to synchronize activities and plan the work for the next 24 hours. This practice isn't well adapted to the project, as while there is just one main developer, daily meeting are not applied. In cases when collaboration with the advisers or other developers is needed, the meeting is agreed beforehand.

Extreme programming is similar in many aspects to Scrum. It also values the customer satisfaction, small but frequent releases, iterative development and readiness to changes. Figures 4.2 and 4.3 show the flow charts of the XP project in general and of an individual iteration, correspondingly.

As can be observed on the figures, the project is elaborated in the following way. The schedule of the upcoming released is defined in the release planning. Spikes are the solutions to exact architectural, design or technical problems, which help to estimate the difficulty of the features implementation and plan the work accordingly. After the release plan has been defined, the project enters in the phase of elaboration, where all the work is done in multiple iterations. Each iteration should also be planned. If some tasks can't be finished in one iteration, they can be moved to another one. Also, when the need for some new feature (User story) appears, this is also taken into account in the release plan.

Among the rules of Extreme programming [13] there are some that are not applied to the current project. For example, daily stand up meetings (analog of Daily Scrums), for the reason described above, paired programming and some others. But these practices can be omitted, while conserving the other rules, and such tailored methodology covers all the planning needs of the development part of this master thesis.

4.2 Iterations

According to the process model, described in 4.1.3, the development is divided in several iterations, which form the release plan. Some of the requirements were identified prior to the start of the development, others have been appearing in the course of the project. The iterations are enumerated, and the name of the iteration reflects the main feature or the objective of the iteration.

4.2.1 Iteration 1. Initial prototype

The goal of the first iteration of the product is the development of the prototype with reduced functionality which would serve as a proof of concept for the ORE method. The following requirements were defined for the first iteration:

- The system should read the required input data: MDIs (in .MGraph format), ontology and source mappings (in xml format)
- Given the input data, the system should produce integrated multidimensional schemas
- Command-line interface should be provided to receive the output the results of the execution
- ORE Fact matching stage should be implemented

- ORE Dimension matching stage should be implemented
- If several integration options exist, let the user choose one
- After each iteration the system should maintain N best solutions based on their cost.

At this first iteration the main data structures needed for the ORE process were designed and implemented. The application was a simple console application, which, given the input data, produced the result a set of multidimensional schemas obtained by integrating all possible combinations of the MDIs of different requirements. In order to simplify the task, it was decided to simplify the FM algorithm by disregarding its steps Step FM:2b and Step FM:2c.

In the end of the iteration, the experiments on the prototype were performed by Petar Jovanovic and myself in order to collect different indicators showing the functioning of the implemented ORE prototype. Some examples of the indicators are: time of each iteration of the process, time spent on facts matching stage, time spent on dimensions matching stage, overall time, size of the solution space. In order to collect these indicators, some additional requirements were defined and implemented:

- The system should measure the time spent at different stages
- The system should measure the overall time of the execution
- The system should count the number of factual concepts and dimensional concepts in the solutions
- The system should export the indicators to CSV format

As input data for the experiment the MDIs representing the information requirements for LearnSQL ontology were used. 13 requirements were defined, and each one of them had 2 to 7 different MDI representations, 4 in average. The results of the experiments confirmed that the number of the resulting solutions grows exponentially when adding new requirements, as every possible combination of the MDIs is maintained by ORE. The time of integrating each coming requirement into the current solution space grows drastically depending on the number of requirements already integrated in the solutions, and also on the number of MDIs. This proved the necessity of specifying a threshold which would prune the worst optimal solutions, based on their cost.

4.2.2 Iteration 2. Redesign of the initial prototype

One of the important results of the first iteration was the detection of a misconception about the *facts* in the MDI schemas. In the first prototype in the fact matching phase the matching was performed over individual factual concepts, while in reality the whole group of factual concepts should be considered as a single fact. Thus, the second iteration was devoted to the redesign of the original solution. The functional requirements didn't change, as the system had to maintain its functionality, while the internal structures and the implementations of the algorithms had to be modified. The design of the Traceability Metadata and Integration Operations which were defined in this iteration, are described in 4.3.4.

4.2.3 Iteration 3. Allowing several integration options

The requirement, which forms the plan for iteration 3 is the following:

- The system should provide the possibility to keep several alternatives for matching between an MDI and an MD schema from the solution space

This allows to keep different alternative results of the integration for each MDI, which gives the user a wider variety of options for choosing the final solution. In order to implement this feature, an additional structure was added (`IntegrationOption`), and some processing logic needed to be changed.

4.2.4 Iteration 4. Basic GUI

The objective of this iteration was to provide the user with a graphical interface in order to be able to interact with the system in a more comfortable way. The following requirements were defined:

- The system should provide a graphical interface which allows to:
  - enter the input data
  - view the results of the integration
  - provide feedback on the alternative schemas

As a result of the iteration, the graphical interface was added, which can show intermediate results of the integration, after each new integrated requirement, allow the user to view each schema individually and see its cost, to discard the schemas which he doesn't find appropriate, from the solution space, and finally to see the final result.

4.2.5 Iteration 5. Integration stage

Until iteration 5 the final result of the ORE process was the schema, integrating all the requirements added at the input. However, this resulting schema contains a number of factual concepts. The objective of this iteration is to implement the Integration stage of the ORE process. Requirement:

- ORE Integration stage should be implemented

In the Integration stage the factual concepts of the final solution are collapsed into one fact, and the concepts of each dimension are also collapsed, so in the end multidimensional schema of the Star type is obtained as final solution.

4.2.6 Iteration 6. CLI for batch input

While graphical interface is very useful for interactive mode, for testing purposes or for experiments requiring batch load, like the one performed in the end of the first iteration, and also for performance testing.

The requirements for this iteration were formulated as:

- The system should provide the command line interface to view the results of the ORE process
- The system should provide the command line interface to input the data
- The system should provide the command line interface to set the parameters of the execution

Implementation of this new feature required changes in the design – as a result an additional abstraction was defined – ORE Controller, which defines a common interface between the main ORE process and input/output operations. When the user starts the application by specifying a parameter he can choose the mode – command-line or graphical. From the listed requirements the current version of ORE implements only the first one. The second and the third were put off temporarily as they were not considered critical.

4.2.7 Iteration 7. Integration with the DB

Iteration 7 adds a new big feature, which is integrating the application with the database. Héctor Candón participated in the project at this stage. He implemented the interface to the chosen database (MongoDB – more information on this in 4.3.1.4) and provided the serialization – deserialization mechanisms, which allowed to transform the serialized MGraph objects used in ORE and GEM to XML format, and also JSON format, suitable storing in the database.

The following requirements regarding the new feature were defined for the ORE application:

- The system should provide the interface for connection to the database
- The system should take the input data from the database
- The system should store the data in the database

As a result of this stage the first two requirements were implemented in ORE. However, the possibility of loading the input data from the serialized MGraph objects was still conserved in the graphical interface, and the user can choose either way of entering the data.

4.2.8 Iteration 8. Bugfixing and improvements

Iteration 8 initially was planned for the implementation of the Complementing design stage of the ORE process. However, after the testing performed in the previous stage, it was decided to devote this iteration to fixing of the issues found in testing. For example, an issue was found in the fact matching process, when the source data included synonyms. Previously the testing was performed using the ontology, which didn't contain "1 – 1" relationships between concepts which can potentially become factual concepts, for this reason this issue hadn't been detected earlier. Another issue found was related to the incorrect use of the transitive closure cache, which resulted in very elevated latency for the operations using the ontology. This was localized and fixed. Apart from that, some improvements were made to the system. Some metrics gathering was added to the system, in order to support the performing to the experiments (see 4.4.3), graphical interface was enriched to allow the user to change the threshold of the top solutions and costs of the operations easily, and some other improvements.

4.3 Design and implementation

4.3.1 Used technologies

This section describes which technologies were used for the implementation of the project, and the reasons why they were preferred to alternatives.

4.3.1.1 Programming language

Java has been chosen as the main technology for for implementing ORE. Java technology is a platform and a programming language.

- Java platform

  The Java platform is a software-only platform, which can run on top of other platforms, including different hardware platforms and operating systems. The main components of the Java platform are the Java Virtual Machine (JVM) and the Java Application Programming Interface (API). JVM provides a runtime environment, which executes Java bytecode – the code expressed with a special instruction set, which is generated by the Java compiler from the source code written in Java or some other programming languages. JVMs are available for different hardware and software platforms, and they all can run the same bytecode. This makes Java cross-platform – the code written and compiled to the bytecode on one platform, can be run in the JVM on another platform. JVM also provides exception handling and memory management system, which is called Garbage Collection (GC). GC keeps track of the objects, which don't have any references to them and thus are unreachable, and frees the occupied memory, reducing the memory leaks.

  Java API provides the core functionality of the Java programming language, it is implemented in Java Class Libraries (JCL), which are contained in the distributions of the platform. JCLs is a Java language standard libraries and they include such facilities as data types and data structures (lists, trees, hash tables etc.), algorithms (sorting, hash functions etc.) and means for interacting with the host platform (input/output, operating system calls etc.).

- Java programming language

  Java is a high-level general purpose object-oriented language with strong type system. The object orientation paradigm is strict in Java – all the code should be organised in classes, and there should always exist the Main class, which serves as an entry point to the application. The code should be organized in a way that

one source code file (with .java extention) hold only one public class (that can be used directly from outside), and the names of the file and the class must coincide. However, the language allows defining the so called “nested classes”. The Java classes are organized in packages. A package groups related classes and provides a common namespace. Java gained a lot of popularity, mostly due to its portability and platform independence, and it is currently one of the most popular programming languages, and is widely used for building all kinds of applications: desktop, web, mobile. Such success of the Java language in the developer’s community led to the appearance of a large number of libraries, which can be reused.

The main reasons for choosing Java as the base technology, apart from its platform independence, is that it was used from implementing the GEM framework. Using the same technology will allow to reuse the existing code, and to integrate ORE into the GEM framework seamlessly.

4.3.1.2 Development tools

The project was developed using NetBeans IDE.

NetBeans is an integrated development environment (IDE) for developing primarily in Java language. It has a wide range of features which help to build Java applications of any type, and as most IDEs it includes a source editor, tools for compiling, building and debugging, version control tools etc. It is free and open source, and its features can be extended by means of plugins. NetBeans IDE is written in Java itself, so it can run on any platform which supports a compatible JVM.

When choosing an IDE for developing the project, Eclipse IDE was also considered, as both tools are free, robust and extensible, and have a similar feature set. The main reason why the decision was taken in favour of NetBeans, is that it provides a solid GUI builder “out-of-box”, which allows to design user interfaces in a comfortable way, by dragging and positioning GUI components in the visual editor. NetBeans automatically generates the code for the application GUI and updates it each time a change was made in the visual editor.

4.3.1.3 Version control system

Version control (also known as source control or revision control) is an very important practice in the software development process. It allows to keep track of all the changes introduced into the source code and other software project assets, and significantly simplifies collaboration, in case several developers work on the same project, by providing

the tools to synchronize the versions of the files and merge the changes made by different collaborators. There are many version control systems (VCS), the most well-known and used are Apache Subversion (SVN), Git and Mercurial. All the three systems provide similar basic functionality. The main difference is that Subversion uses centralized approach, while Git and Mercurial are distributed VCSs, for this reason Git and Mercurial are a bit more complex, have a wider set of commands and more flexible in terms of the variety of version control workflows they support. However, for the development of this project Subversion was chosen. The following reasons for that can be named:

- There was already an existing Subversion server used by the department.
- All the collaborators are familiar with Subversion.
- The features of Subversion are enough for the purpose of this project, distributed systems do not bring significant benefit.

4.3.1.4 Database

It was decided to introduce a database into the ORE project. The main purpose for using a database is that it would facilitate the integration between the ORE application and the previously developed GEM framework. The architecture of the resulting integrated system is shown in Figure 3.1. NoSQL technology was chosen for storing the data of ORE and GEM.

NoSQL is a relatively young database paradigm which was developed as an alternative to the traditional relational database model. The term *NoSQL* originally meant “not SQL”, emphasizing the contrast with the RDBMSs which commonly use SQL as their query language, however nowadays the term is usually interpreted as “Not only SQL”, which is a less strict definition, and is more generic, as it admits that NoSQL databases may actually use SQL-like languages to query the data.

The advantages of the NoSQL technology over the relational database technology are usually associated with scalability and performance, which are highly demanded by information systems nowadays, due to the constant growth of the volumes of data stored in organizations, and the need to access this data in an efficient way. NoSQL databases achieve this by supporting horizontal scalability and auto-sharding. But one of the main differences, which can also be a huge benefit, is a different approach to data modelling. While RDBMSs are based on relational model, with fixed structure and data types, NoSQL databases provide alternative kinds of data schemas, which are usually dynamic, making this type of databases very flexible. The modern NoSQL databases can

be classified in the following groups by the data model they use: key-value, document, column-oriented and graph databases.

For the purposes of ORE and GEM the most suitable kind is a document store. These databases are composed of collections of semistructured documents equipped by indexes. A document typically has a specified format, like JSON or XML. Among the document database available currently **MongoDB** was chosen.

MondoDB is one of the leading NoSQL databases, it is free and open sources, and has a big community of developers and users. The documents are stored in *collections*, and typically, all documents in a collection have a similar or related purpose, in this was a collection in MongoDB is the equivalent of an RDBMS table. The closest equivalent to a table row in MongoDB would be a document. MongoDB uses its own document format called BSON, which is binary-enconded serialization of JSON-like documents. BSON is designed to be lightweight, traversable, and efficient. Like JSON, it supports the embedding of object and arrays within other objects and arrays. Each document has a unique id within a collection, which is stored in the `_id` field of the document. By default, for the `_id` field MongoDB uses values of a special 12-byte BSON type called ObjectId, generated based on timestamp, machine ID, process ID, and a process-local incremental counter, which guarantees the uniqueness of the id within a collection.

BSON also contains other extensions that allow representation of data types that are not part of the JSON specification, for example Date type and a BinData type. The latter is a very interesting feature, it allows to store any data as a binary byte array inside a MongoDB document. There is a limitation, however, on the size of a MongoDB document, which is 16MB. For the documents exceeding this limit, there is another specification, called GridFS. Instead of storing a file in a single document, GridFS divides a file into parts, or chunks (limited to 256k by default), and stores each of those chunks as a separate document. GridFS uses two collections to store files: one for storing the file chunks, and the other one for the file metadata.

MongoDB supports indexing at the collection level, including compound and multikey indexes. It also provides a very rich query language, and drivers and client libraries for most of the commonly used programming languages, including Java, which will be used for the development of the ORE project.

So, the reasons for choosing MongoDB can be summarized in the following points:

- free product, well-documented and supported by a big community;
- flexible document format based on JSON, which doesn't require a strictly defined schema;

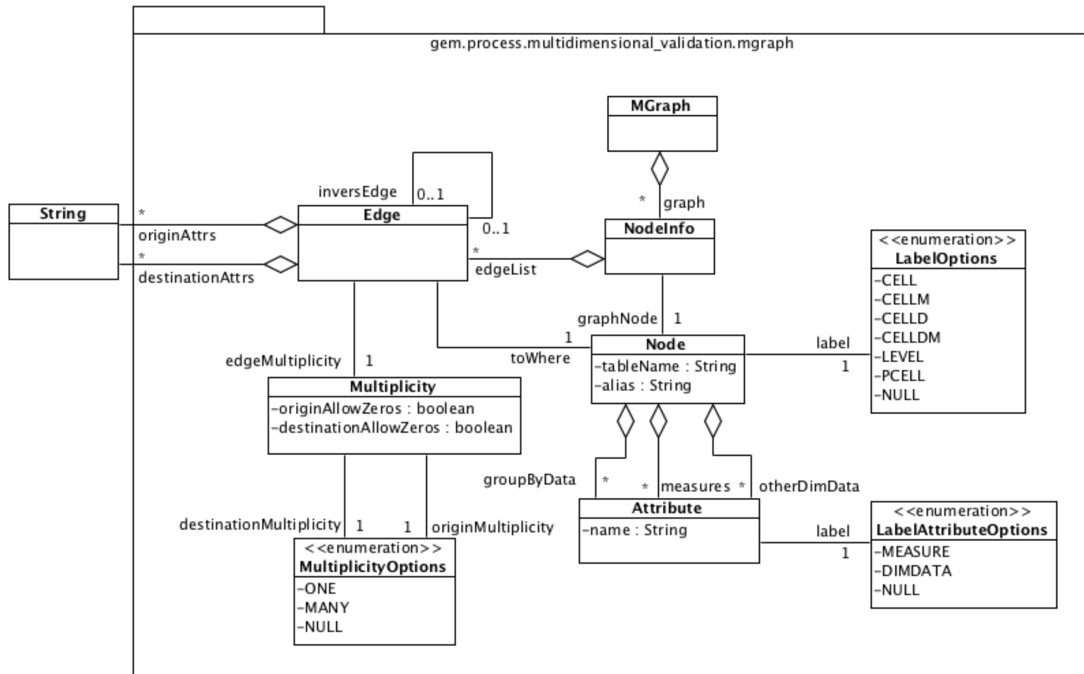


FIGURE 4.4: MGraph class diagram

- possibility to store binary data;
- reach query language, which allows making complex queries;
- easy to use and well-documented Java driver.

4.3.2 Reuse of existing code

This subsection describes the already existing code developed for the GEM framework or previous related works, which was reused in for the elaboration of the ORE project.

4.3.2.1 MGraph

MGraph is one of the principle concepts used both in GEM framework and in ORE. This class was originally implemented by Oscar Romero as part of the MDBE (Multidimensional Design By Examples) tool ([14]). MGraph structure is a graph consisting of the ontology concepts (tagged with their multidimensional role) and relations between these concepts. Figure 4.4 shows the class diagram of the `MGraph` package, which contains the `MGraph` class itself and related classes.

The `MGraph` consists of the set of `NodeInfo` object. Each `NodeInfo` contains a `Node`, which actually stores the information about the nodes of the graph, that is the name of

the concept and the alias. The `name` attribute corresponds to the name of the concept in the ontology, and the `alias` defines its alias in the information requirement, which can appear if there are more than one nodes corresponding to the same concept, so they need to be distinguished. Thus, in the `MGraph` structure the combination `name` + `alias` should be unique for the Nodes. The `Node` also contains attributes, which are represented by the instances of `Attribute` class, and in a `Node` structure they are grouped into several sets:

- measures – for the ontology concepts tagged as measures;
- groupByData – for the ontology concepts that are tagged as dimensions;
- otherDimData – for the ontology concepts that are tagged as descriptors.

The role of the concept in the multidimensional model is represented by the `label` attribute of the `Node`, which can take values listed in the `LabelOption` enumeration.

The relationships between the concepts are represented by the class `Edge`. Each `NodeInfo` object contains a set of edges (`edgeList` attribute), and each `Edge` in its turn has a reference to another `NodeInfo` object (`toWhere` attribute). The `Edge` also stores the multiplicities of the relation.

4.3.2.2 Classes for working with data sources

In GEM Framework a lot of classes for working with the information requirement, ontologies and source mappings were implemented. Among other features, they provide the methods to read the XML-files (describing the ontology, source mappings, and user requirements) and transform them into the internal structures, find relationships between the concepts in the ontology, including the cardinalities of the relationships. These features can be reused for the purposes of the ORE system.

Below is an overview of the related packages.

GEM uses Jena framework as a base for working with OWL ontologies. Apache Jena is a free and open source Java framework for building semantic web and Linked Data applications. The framework is composed of different APIs for various technologies and standards based on the Resource Description Framework (RDF). The Jena Ontology API is the one that is used for the GEM, as it supports OWL ontologies. Jena API provides object classes for representing ontologies and methods for manipulating them, and also methods to retrieve and parse the ontologies from files and saving them. Besides, Jena also provides an inference engine (or reasoner), which allows to derive new data

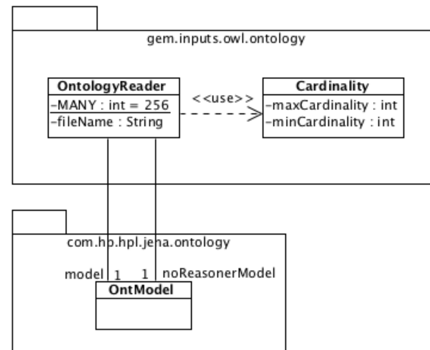


FIGURE 4.5: Ontology Reader class diagram

from data that is already known by examining the knowledge base and drawing logical conclusions, based on predefined rules.

GEM uses the classes provided by Jena and builds more functionality on top of them. Figure 4.5 shows the diagram of the package with classes for reading the ontology files. The main class is called `OntologyReader`, it uses Jena `OntModel` interface to represent the ontology model. `OntologyReader` also provides methods to get the cardinality of the relationships between the concepts. For this purpose the class `Cardinality` is used, it keep two integer attributes: `maxCardinality` and `minCardinality` to represent maximum and minimum cardinality of the relationship respectively. In OWL ontology the cardinalities are represented as property restrictions of the source class, and its value should be an integer. In order to represent the cardinality $MANY$, the value "-1" is used, and `OntologyReader` considers that when translating the cardinality.

The parsing and manipulation over the source mappings is performed by the class `SourceMappingXMLReader`. This class and its dependent classes are shown on the class diagram in Figure 4.6

4.3.3 Architecture and description of the packages

4.3.3.1 General concepts

As the ORE application is developed in Java programming language, it is organized in packages. The classes, which perform together a specific function or which are closely related are grouped in packages.

The application uses external libraries, distributed as .jar-files. The management of the dependencies is performed with help of NetBeans environment.

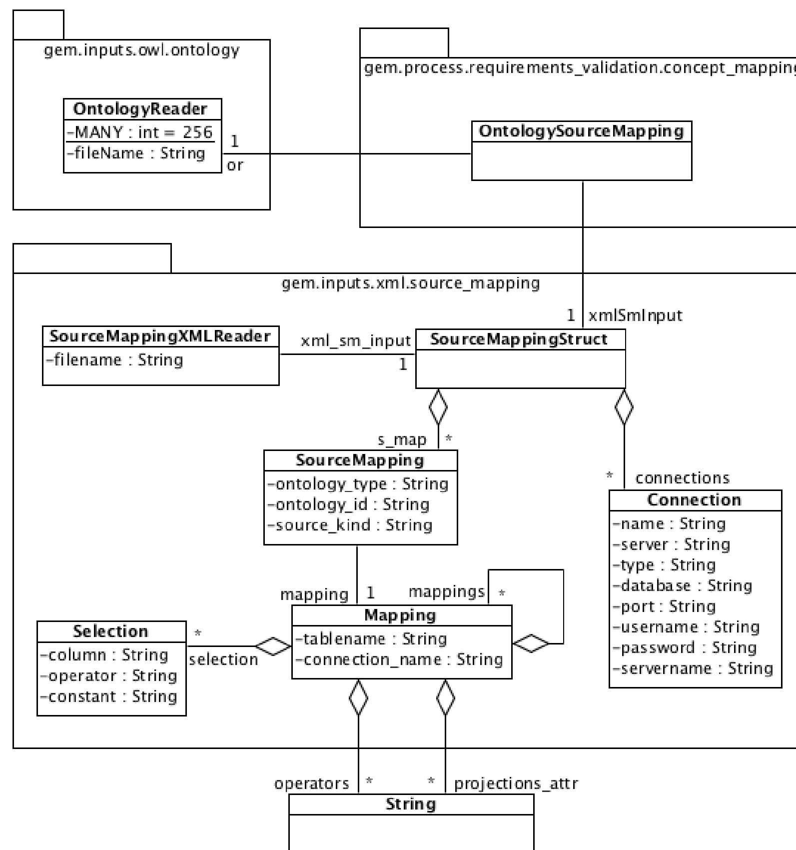


FIGURE 4.6: Source Mappings Reader class diagram

4.3.3.2 Three-layer architecture

The main architectural pattern used in the ORE application is Three Layer (also called Three Tier) Architecture. According to this pattern the components of the application are divided into three groups (layers): presentation, domain, and data management. Figure 4.7 illustrates this separation.

The responsibilities are divided among the layers as follows:

- **Presentation Layer** is responsible for interaction with the user and provides the interface for data input and showing the results. It receives the data, introduced by the user, passes it to the lower levels for processing, and after the result is obtained, the presentation layer shows it to the user in an understandable way. Alternatively, the presentation layer can be driven by the events happening in the lower levels, and it is in charge of showing the processes happening in the system to the user. This is a common way of interaction, for example, for monitoring software. The application presentation layer is typically implemented as a graphical user interface (GUI), or command-line interface (CLI). ORE was designed to

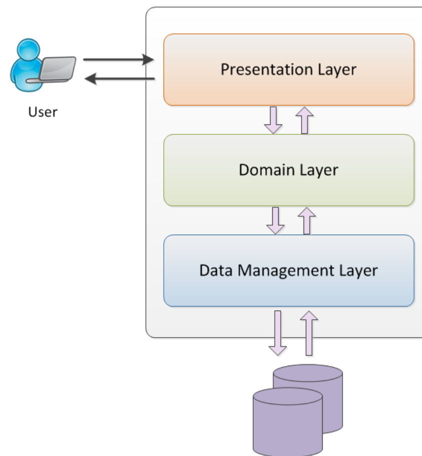


FIGURE 4.7: Three layer architecture

support both interfaces, when starting the application the user can choose which mode he wants to work in by using parameters. GUI is a full-featured interface, which provides a way to input the data, review the preliminary results and receive the final results. CLI mode is used mainly for testing purposes, when performing a batch processing of a big amount of data. Currently in the CLI mode the user only gets the log of the operations which are being executed in the system, but cannot interact with the system during the execution. The CLI can be enhanced if needed.

- **Domain Layer**, also known as Business Logic Layer, is responsible for most of the functionality of the application. It processes the data, received from the data management layer, or from the user via the presentation layer (makes modification to the data, performs calculations, etc.), and gives back the result. The domain is the part of the application where all the domain concepts and relationships between them are modelled, and all the actual logic is implemented.

- **Data Management Layer** is responsible for the persistence of the information in the system. Data layer doesn't have the knowledge how to operate over the data, but it provides the means to store and retrieve data from the file system or database management systems.

The components in each layer can only interact with the components of the same or adjacent layers. Thus, for example, the components of the presentation layer can't interact directly with the data management layer; and the user can't access directly the components of the domain layer.

The main benefits of the three layer pattern is the increase in changeability, portability and reusability of the system. The modular structure based on the separation of the

presentation of the data, its processing and persistence, allows to change one module for another, without affecting the rest of the system. For example, if in the system, where file system is used for persistence, a need to introduce a database occurs, the changes should be made only in the data management layer, and the rest of the system remains unchanged. In a similar way, the GUI of the application can be changed for another one, and if it uses the same interface with the domain layer, there is no need to modify the components of the domain and data managements layers.

In the ORE application the separation between the layers is based on packages. There are two packages at the presentation level: `ore.cli` and `ore.gui`, two packages at the database level: `ore.DB` and `gem.mongo`; the rest of the packages belong to the domain layer.

4.3.3.3 Main class and controllers

The main class of the application which serves as an entry point is `ORE`. When running the application a parameter defining the mode can be specified: either `--gui` (shorthand `-g`) for GUI mode, or `--cli` (shorthand `-c`) for the command line mode (GUI is the default). For parsing the parameters of the application the Apache Commons CLI library was used. It helps to define the allowed parameters and takes care of parsing and validating command line options passed to the programs. In the current implementation of ORE it doesn't bring much benefit, however, it will help significantly for the command-line interface of the application, for defining the parameters of the application (such as the threshold for number of schemas or weights of the operations), and the database connection settings.

The responsibility of the `ORE` class is to start the application using the kind of interface selected by the user.

The interaction between the domain layer of the ORE application and the presentation layer is performed by the controllers. They coordinate the communication between the layers, and allow having two different presentation modes, using the same interface with the domain level. The controller also manages access to the database.

`ORECtrl` is an abstract class which defines the necessary operations and implements some of them. It is specialized in two subclasses, `OREGUICtrl` and `ORECLICtrl`, which serve for GUI and CLI execution modes correspondingly. Figure 4.8 illustrates the hierarchy. The `OREGUICtrl` has an association to the `OREGUI`, which is the main class for the GUI of the application. This association is navigable in two ways, i.e. the GUI can also make requests to the domain layer through the `ctrl` attribute.

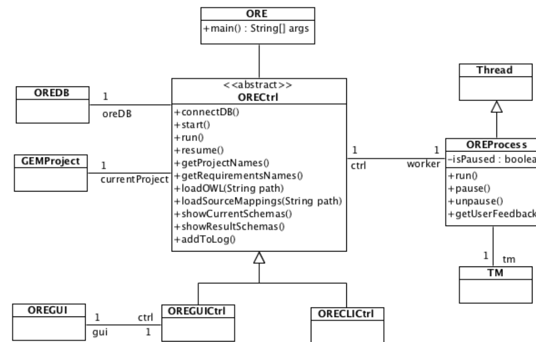


FIGURE 4.8: Main class and controllers class diagram

The appearance of the concept of class `GEMProject` in ORE is partly related with the introduction of database and the attempt to integrate the ORE and GEM projects. A project in ORE and GEM is an entity which has a name, has a single ontology and a source mapping structure related to it, and a set of requirements. Basically a `GEMProject` is a representation of a real data warehouse project, the purpose of which is to perform the design of the data warehouse and ETL processes given the conceptual model of the data sources (Ontology) and the physical design of the operational databases (source mappings).

`OREProcess` is the class which holds all the main logic of ORE and implements the *ORE* algorithm. It is responsible for creating and manipulating the TM structure. `OREPrcocess` extends Java `Thread` class. This allows to create a new thread for the ORE process, which in particular is necessary for the GUI mode of the application, as GUI should be executed in a separate thread.

4.3.3.4 ORE process implementation

As discussed in the previous section, the main ORE process is implemented in the class `OREProcess`, contained in the `ore.process` package. Apart from it, the package contains a class for each implemented ORE stage: `FactMatchingStage`, `DimensionMatchingStage` and `IntegrationStage`. The main methods of these classes implement the FM, DM and INT algorithms correspondingly.

Another important asset in the `ore.process` package is the class `OREConfig`. It stores the configuration of the ORE application instance, including such parameters as the threshold of the solution space size (maximum number of schemas which is kept in the TM structure), and the weights of the integration operation. In fact, the weights of the operations are stored in a separate class – `CostModel` which keeps the weights for each operation in a `EnumMap` – a map, which has the type of the operation (a value of the `OperationType` enumeration) as a key, and a number of type `double` as a value.

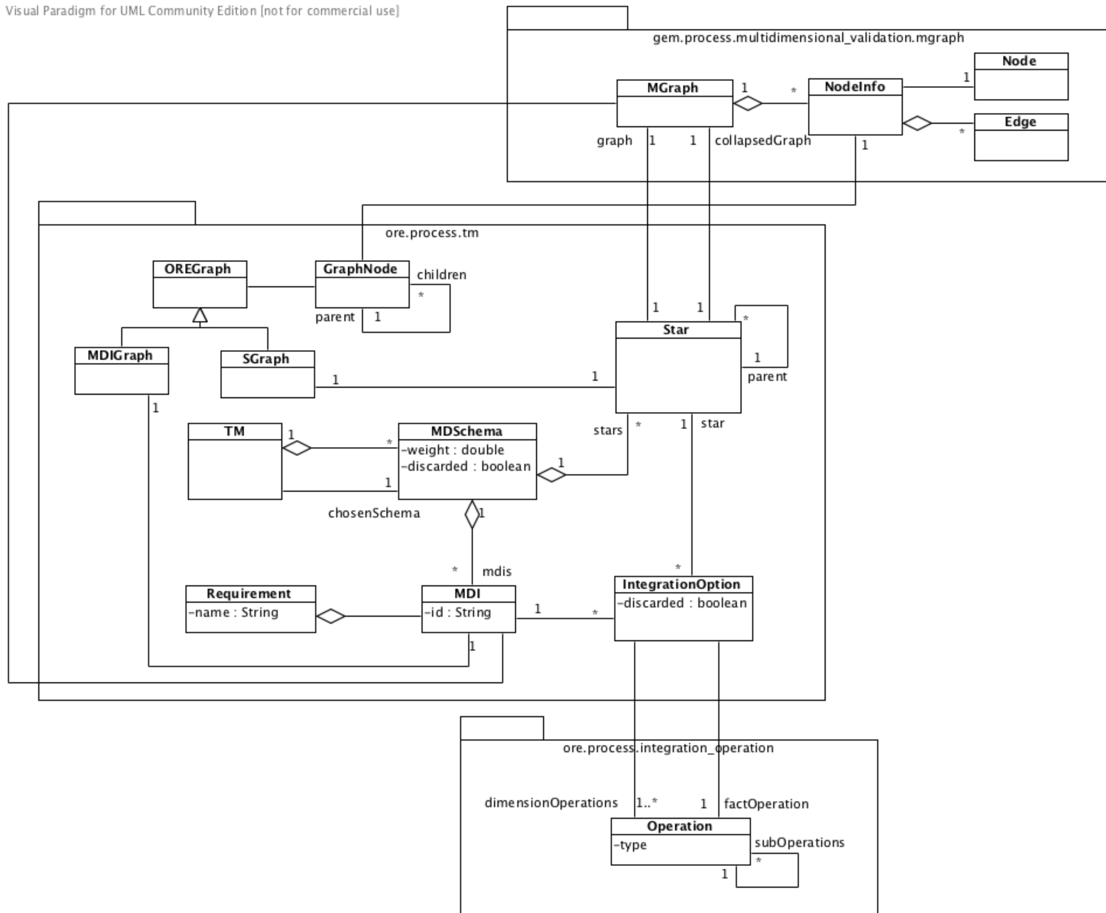


FIGURE 4.9: Traceability Metadata structure

`OREConfig` uses the Singleton pattern, i.e. there can be only one instance of `OREConfig` in the application, and it can be accessed by any component.

4.3.4 Traceability Metadata

Traceability Metadata (TM) is a structure that stores all the information about the inputs and the results of the ORE process along with all the intermediate information which is needed during the execution of the process. Figure 4.9 presents the class diagram of the TM structure.

The class called `TM` is the main class which serves as a container for all the traceability information. It stores the complete set of alternative solutions, which satisfy the information requirements, and keeps the information about how the integration was performed.

The information kept in `TM` can be classified into the following types:

- Requirement-related metadata

The information requirements (IRs) are represented by the `Requirement` class. Each `Requirement` has a name and stores a set of multidimensional interpretations, produced in the GEM process and represented by the `MDI` class. The main part of the `MDI` is a graph (`MGraph`, see 4.3.2.1) which captures the concepts of the MDI as `Nodes`, and the relationships between them as `Edges`.

- Resulting MD schema

The multidimensional schemas which are generated as a result of the integration process are stored in the `MDSchema` structures. Essentially `MDSchema` consists of one or several star schemas. One schema can be obtained in case when the facts of the MDIs are matched and merged together, otherwise the fact insertion takes place, and thereby the final solution consists of several star schemas. One star schema is represented by the class `Star`. The `Star` uses the same graph structure as `MDI` — `MGraph`, which provides homogeneity and makes it possible to use the ORE approach for integrating two multidimensional designs with minor changes in the algorithms implementation.

For facilitating the execution of the fact matching and dimensions matching algorithms, additional data structures have been introduced, namely, `OREGraph`, `MDIGraph` and `SGraph`. As it has been discussed above, the facts and the dimensions in MDI can be seen as directed acyclic graphs, and the ORE algorithms take advantage of it. `OREGraph` uses a simple Directed Acyclic Graph structure, implemented in the class `GraphNode`. `GraphNode` is a generic class which accepts any type of objects to be stored as the node, and each `GraphNode` object also holds a list of the references to the adjacent nodes of the graph, the parent → child relationship defines the direction of the edge. `OREGraph` contains such graph structures (using `NodeInfo` as a type) separately for the fact and for the dimensions, and essentially serves as a wrapper for the MGraph structure.

The figures below illustrate how the `OREGraph` works. Figure 4.11 shows an example of an MDI. In `MGraph` it would be represented as a plain list of nodes (see Figure 4.12). The nodes organized in sub-graphs using the `OREGraph` structure are shown on the Figure 4.13.

An important feature of the `OREGraph` structure is that it can keep several subgraphs of the factual nodes in the resulting MDSchema. This allows keeping all the nodes of the integrated MDIs in cases when the facts are matched as having the same multidimensional space. An example can be observed in Figure 4.10. The facts in MDI 1 and MDI 2 do not match, however they share the same multidimensional space, and they can be merged according to Step FM:3c. If they merged into one concept right away, the original concepts and relationships would

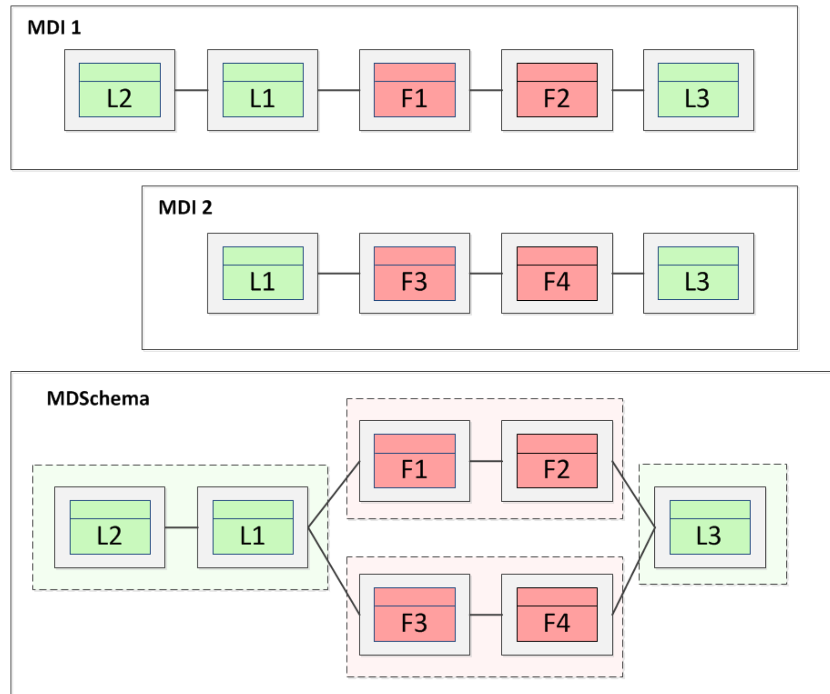


FIGURE 4.10: Merged facts in MDSchema

be lost, which would make it difficult to use the ontology for the further integrations. This is overcome as shown on the Figure 4.10: in the resulting *MDSchema* both graphs of factual concepts are preserved for future reference, which is reflected in the `OREGraph` structure.

This only applies to the MDSchemas, as in MDIs there can be only one subgraph of related factual concepts.

`MDIGraph` and `SGraph` are the two subclasses of the `OREGraph` class, they are associated with and keep a reference to `MDI` and `Star` instances respectively.

- MD integration metadata

  The information about the integration options between the schema and the new MDIs, coming with each iteration, is stored stored in the `MDSchema` for each integrated MDI as a list of `IntegrationOption` objects. The use of the integration operations in the ORE approach has been explained in 3.4. In this section the implementation aspects will be discussed.

  During the execution of the process, integration options can be marked as discarded by the user (a boolean attribute `discarded`), which allows to prune the solution space, disregarding the alternative schemas which the designer considers inappropriate.

  `IntegrationOption` consists of the list of integration operations. The description of the operations is provided in 4.3.4.1.

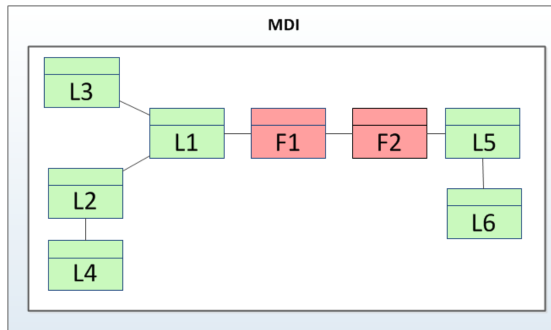


FIGURE 4.11: MDI

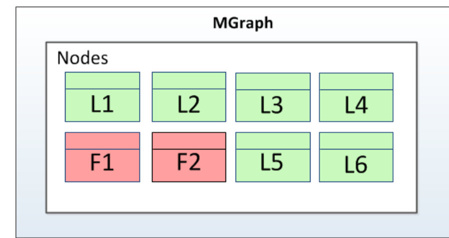


FIGURE 4.12: MGraph representation

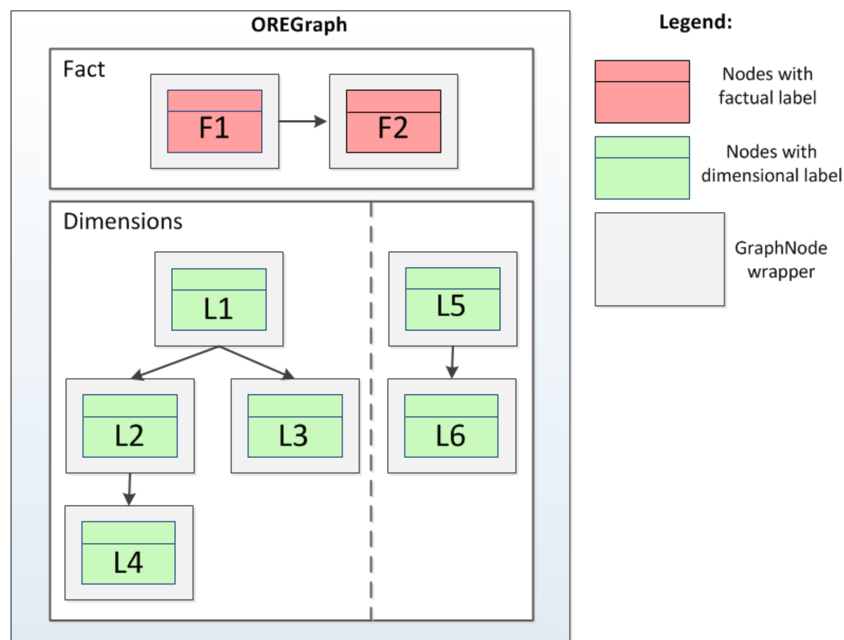


FIGURE 4.13: OREGraph representation

4.3.4.1 Integration operations

An `IntegrationOption` instance contains one operation for matching the facts (either `MergeFacts` or `InsertFact`), and a list of operations related to dimensions matching (`MergeLevels`, `InsertLevel`, `InsertRollUp`). Some of the operations in their turn can contain sub-operations (e.g. `InsertFactMeasure` and `InsertDimDescriptor` for fact and level merging operations respectively).

Figure 4.14 shows the classes representing the integration operations. These classes do not correspond exactly to the operations as described in 3.4. In the implementation, some additional levels of abstractions have been added.

The base class, `Operation`, is abstract, and it only holds the `type` attribute (a value of the `OperationType` enumeration), and the declarations of the methods `getWeight()`,

which returns the weight of the operations, according to the cost model, and `apply()`, the method which performs actual modifications over the MD schema depending on the type of operation. These methods are overridden in subclasses.

`MergeConcepts` is the operation responsible for merging individual concepts of the MDI and the current schema, where the concepts are represented by `NodeInfo` objects inside the corresponding `MGraph` structures. Depending on the multidimensional role that the concepts have, the class can be specialized either as `MergeFactConcepts` or as `MergeLevels`, where the latter coincides with the *MergeLevels* operation as it was defined in 3.4.

`AttributeOperation` is an abstract class which is used for inserting individual attributes of the concepts; it has two subclasses: `InsertDimDescriptor` and `InsertFactMeasure`, the type should correspond to the role of the concept enclosing the attribute.

It is worth mentioning that the attributes of the dimensional concepts (labelled as *levels*) can also have different roles: they can either be used for grouping the data, or for slicing; it is reflected by the attribute set they belong to in the `Node` structure `groupByData` or `otherDimData` correspondingly (see 4.3.2.1 for more information). This is taken into account in the `InsertDimDescriptor` operation in the attribute `descriptorType`, which can take the values GROUP_BY_DATA or OTHER_DIM_DATA (value of `DimDescriptorType` enumeration).

`MergeFactConcepts` and `MergeLevels` operations may contain sub-operations of `InsertFactMeasure` type or `InsertDimDescriptor` type correspondingly, and thus their weight is calculated as the sum of the weights of the sub-operations.

`InsertConcept` is an operation which is used to insert a new concept from the MDI into the current schema. The data about the concept to be inserted is stored in the `mdiNode` (`NodeInfo`) attribute. Apart from that, when the concept is inserted, it is needed to know, how it should be introduced into the schema with respect to other concepts. In order to keep track of the relationships that will need to be recreated, the `InsertConcept` operation also keeps the list of graph edges (`Edge` type), which are used to recreate the relationships between the concepts when the operation is applied.

For the factual concepts the `InsertConcept` operation can be applied as it is, but when dealing with dimensions, some additional information is needed to perform the operation. The class `InsertLevel`, extending the `InsertConcept`, is used for the dimensional concepts. In case when the inserted concept is a level in an already existing dimension hierarcy, a roll-up relation needs to be inserted. The operation *insertRollUp* (see 3.4) is represented by the class `InsertRollUp`, which holds a node (`rollUpTo`),

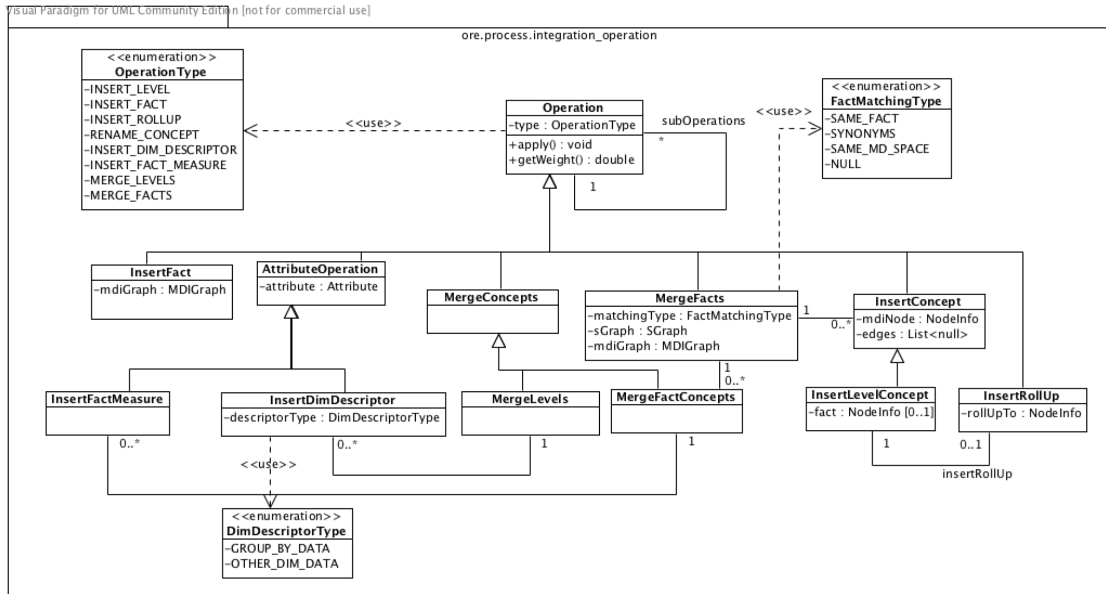


FIGURE 4.14: Integration operations

which serves as the "to-one" end of the relationship. Alternatively, when the inserted level serves as the atomic level of a new dimension, the information about the corresponding factual node is needed. This information is kept in the `fact` attribute of the `InsertLevel` operation.

MergeFacts (see 3.4) is the most complex operation, as one fact can contain multiple concepts, which are related among them. As type of matching between the facts is defined in the `matchingType` attribute and can take the values SAME_FACT, SYNONYMS, SAME_MD_SPACE (values of the enumeration `FactMatchingType`). The `subOperations` attribute contains the set of operations over the individual factual concepts used to integrate the two facts. The sub-operations can be either `InsertConcept` or `MergeFactConcepts`.

4.3.5 Transitive Closures Cache

When matching different concepts or looking for new analytical perspectives the ontology is explored for searching the possible relationships between the concepts. Depending on the complexity of the ontology this can be a costly operation, and as the ontology is queried often in the ORE method, it could become a bottleneck in the system. For improving the performance a special component, called *Transitive Closure Cache* (*TCC*) is proposed. *TCC* works like a cache, maintaining the relationships between the concepts, which have already been searched for. It receives the request for searching the relationships, e.g., (A,B). If no entry is found, meaning that the relationships starting from the concept (e.g., *A*) are not yet loaded to the *TCC*'s structures, *TCC* accesses

From	To	Cardinality	Direct	Through
A	B	$* - 1$	Y	/
A	C	$* - 1$	N	B
A	C	$* - 1$	N	B, C
A	D	$* - 1$	N	B

FIGURE 4.15: Transitive Closures Cache

the ontology and looks for the requested relationship. At the same time it explores the ontology to find the transitive closure of the relationships ("1 - 1", "1 - *" or "* - 1") starting from the same concept (e.g., A) and loads the corresponding entries to its structures. Figure 4.15 depicts schematically the *Transitive Closures* structure.

Figure 4.16 shows the class diagram of the package implementing the transitive closure structure. The package was implemented by Petar Jovanovic. The TCC uses intensely the ontology and the GEM classes for accessing it (see 4.3.2.2), and also, some code developed by Petar for the GEM requirements completion stage could be reused (with some modifications) for exploring the ontology and finding the relationships between the concepts.

The class which is accessed directly from ORE is `TransitiveClosures`. The structure used to save the relationships between the concepts is implemented as a hash map (`closures`), where the key is the URI of the start node of the relationship, and the value is a set of the paths starting from the specified node to other nodes, each path is represented by the class `OntologyRelationPath`.

`OntologyRelationPath` keeps the nodes which appear on the path from the start concept to the end concept as a set of objects of the class `OntologyNode`, and the edges between the nodes as a set of `OntProperty` objects (attribute `pathEdges`). Each `OntologyNode` has a reference to the corresponding node from the ontology – `OntClass`, and an attribute which specifies the alias of the node in the schema (`alias` of type `String`). `OntProperty` and `OntClass` interfaces are provided by Jena.

The cardinality of the path is stored in the attribute `pathCardinality`, which takes values defined in the `CardinalityOptions` enumeration: `ManyToOne`, `OneToMany`, `OneToOne`, `Taxonomy`, `NULL`. When the specific path is accessed for the first time, the cardinality of the relationship between the start and the end concept is calculated by passing through the whole chain of concepts, and stored in this attribute. In case of a recurring access, the precalculated value is used, thus the costly process of finding the cardinality of the relationships is performed only once.

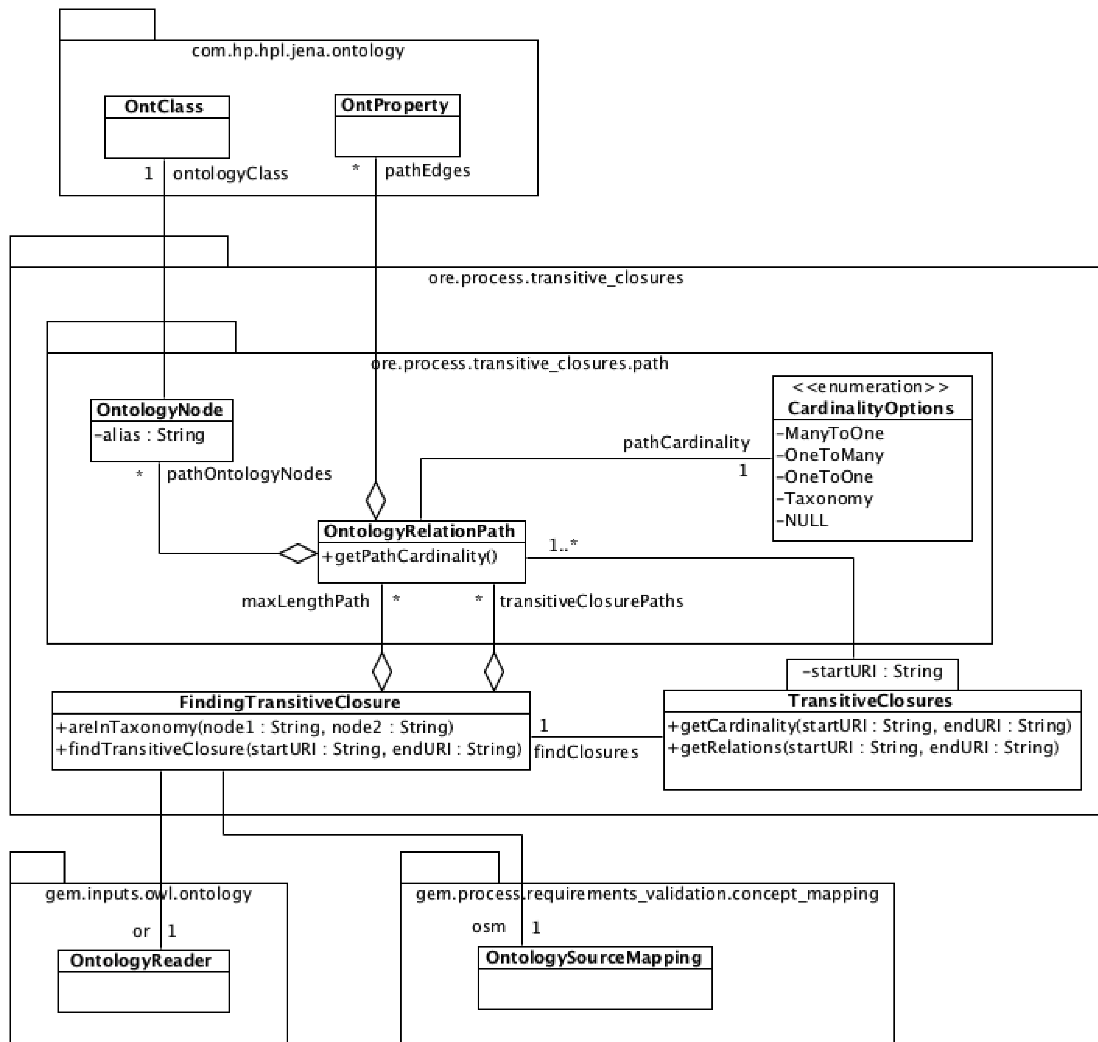


Figure 4.16: Transitive Closures class diagram

When the request for cardinality between two concepts is received by the `TransitiveClosure`, first the hashmap is accessed using `startURI` as the key. If the entry is found, the paths starting from the specified concept are examined, searching for the paths ending with the *endURI*. The cardinalities of the corresponding relationships are checked, and the result is returned to the requester. Otherwise, if the entry is not found, which means that it is the first time the concepts are requested, the `TransitiveClosure` has to check the ontology.

The class which has the responsibility for accessing the ontology is `FindingTransitiveClosure`. It has references to the ontology and source mapping readers (`OntologyReader` and `OntologySourceMapping`), and it provides methods that explore the ontology and collect all the possible paths of concepts in the ontology that start with the one passed in the `startURI` parameter, which are subsequently saved in the hashmap for a further fast access.

4.3.6 Database design

As explained in 4.3.1.4, MongoDB was chosen as a database for ORE-GEM integration. In this section the organization of the data in the database will be discussed. The database design was performed by Héctor Candón under the guidance of Oscar Romero. A several meetings were held to discuss the design, where Alberto Abelló, Petar Jovanovic and myself also participated in the discussions.

There are several categories of data which is stored in the database:

- OWL ontology
- source mappings
- MDIs produced by GEM
- the integrated MD schemas produced as a result of the ORE process

Apart from this, another important concept is *project*. *GEM project* is a way of organizing the data related to a data warehouse project, from its early stages though the rest of its lifecycle, including maintenance. *GEM project* also facilitates the integration among GEM framework and ORE, and presumable other components of the system that would be integrated in the future. A *GEM project* is characterized by the data sources, and the set of the information requirements, defined for the data warehouse, all this information needs to be stored in the database.

As explained in 4.3.1.4, data in MongoDB is organized in collections, which are similar to tables in relational databases. For the GEM-ORE project the collections are organized in the following way:

1. Metadata
2. information requirements collections
3. MDIs produced by GEM
4. the integrated MD schemas produced as a result of the ORE process

The `Metadata` collection stores the data related to the projects, such as:

- the name
- the ontology

- the source mapping

The name of the project should be unique inside the `Metadata` collection. As the ontology and the source mapping have XML format, there are two ways to store them in MongoDB: translating them from XML to a JSON-like format, used but MongoDB (BSON), or storing them in a serialized form as an attribute of type BinData (binary data). Translating XML into JSON is not as straightforward as it might seem. The reason is that while in XML there are two ways of representing data: elements and attributes, in JSON only attributes are used. In addition, there is no default support of namespaces in JSON, and thus no clear way how to translate XML elements using namespace prefixes, which is heavily used in OWL, for example. For these reasons, it was decided, at least for the time being, that the ontology and the source mapping will be stored in binary format. The format of the document in the `Metadata` collection is represented by an example shown in Listing 4.1. Here, _id is a document id of BSON type ObjectId, created by MongoDB by default.

```
{
        "_id" : ObjectId("5215f16b1a88452a73bbe5b2"),
        "Name" : "Project name",
        "Ontology" : BinData(0,"PD94b \ldots ncz4=")
        "SMappings" : BinData(0,"PD94b \ldots tPg0K")
}
```

Listing 4.1: Document structure in Metadata collection

Thus, all projects are described in the `Metadata` collection, and each project has one document corresponding to it. For the rest of the artifacts, the following collection schema was chosen. Each project has three collections related to it: Requirements, MDIs and MDSchema. In order to distinguish the collections belonging to different projects a prefix equal to the name of the project is added to the collection name, for example, for the project with the name *Project1* the corresponding collections would be: *Project1_Requirements*, *Project1_MDI* and *Project1_MDSchema*. As the project name is unique in the database, there will be no collapse of the collection names.

In the *Requirements* collection each document corresponds to one information requirement. As originally the requirements were represented as XML documents, the translation is to be made in order to store the document in the database and be able to modify it later. The structure of the document in the *Requirements* collection is shown on Listing 4.2.

```
{
        "_id" : ObjectId("521e3bce1a889a9783154c80"),
        "Name" : "Requirement-1",
        "Retrieved" : false,
```

```
        "Requirement" : <json-like representation of the XML document>
}
```

Listing 4.2: Document structure in Requirements collection

Here it can be seen that the requirement again has a unique ID, a name, given by the user, a field for storing the requirement translated from XML to BSON-compatible format (is not shown in the example, as the translation isn't yet implemented). The attribute `Retrieved` is a flag, which is to be used in order to synchronize the GEM and ORE processes, and it indicates, whether this requirement has already been processed or not by GEM, i.e. whether the MDI representations has already been generated. The flag set to `true` indicates to ORE that this requirement should be taken to the processing.

The MDI collection keeps the MDIs of the project. Listing 4.3 shows the collection document format.

```
{
  "_id" : ObjectId("52a8b6de4728b9fdda5cbfcb"),
  "Requirement_id" : ObjectId("52a8b6de4728b9fdda5cbfca"),
  "MGraph": {
    "MGraph": {
      "graphNodes": {
        "nodeInfo": [
          {
            "graphNode": {
              "attr_id": "Lineitem.Lineitem",
              "extraInfo": "",
              "attr_labelOption": "CELLM",
              "attributes": {
                "attribute": [
                  {
                    "name": "Lineitem_l_extendedpriceATRIBUT",
                    "attr_type": "MEASURE"
                  }
                ]
              }
            },
            "edges": {
              "edge": {
                "attr_id": "http://www.owl-ontologies.com/unnamed.owl#Contained",
                "inverseEdge": {
                  "attr_id": "http://www.owl-ontologies.com/unnamed.owl#Contained",
                  "attr_contextId": 0,
                  "iMultiplicity": {
                    "iDestiny": {
                      "attr_type": "ONE",
                      "attr_zeroAllowed": false
                    },
                    "iOrigin": {
                      "attr_type": "MANY",
                      "attr_zeroAllowed": false
                    }
```

```
                  }
                },
                "multiplicity": {
                  "origin": {
                    "attr_type": "MANY",
                    "attr_zeroAllowed": false
                  },
                  "destiny": {
                    "attr_type": "ONE",
                    "attr_zeroAllowed": false
                  }
                },
                "pointingNode": {
                  "attr_refID": "Orders.Orders"
                },
                "attr_contextId": 0
              }
            }
          }
        ]
      },
      "attr_HasProblems": false,
      "attr_Valid": true
    }
  }
}
```


LISTING 4.3: Document structure in MDI collection

As in other collections, the document in this collection has a unique ID, the JSON representation of MGraph, and also a `Requirement_id` attribute, which is a reference to the corresponding Requirements document (analog of foreign key).

For the ORE input data, represented as MDIs, the problem of storing in the data and retrieving it later base consists of several steps:

1. translate MDI object to XML
2. translate XML object to JSON for storing in MongoDB
3. translate JSON back to XML
4. deserialize JSON representation of MDI back to MDI object

While there was a possibility of skipping the step of XML translation, and transform the object directly to JSON, it was decided to keep the XML format too, as it is still considered a standard format of data interchange. The classes for transforming the main attribute of MDI – MGraph, were implemented by Héctor Candón.

The last collection – MDSchemas, which has to store the results of the ORE process, hasn't yet been implemented.

4.4 Testing

4.4.1 General overview

Testing is an important stage of software development. Software testing is performed to verify that the implemented software behaves as expected, i.e. that satisfies the requirements which were defined previously. Testing contributes to the ensuring the quality of the software. The output of the testing process usually consists of errors and other defects, found in the software, that are consequently fixed.

Depending on the development methodology used for the project, testing can be performed at different stages of the development process. For example, in the "waterfall" approach testing is performed after all the implementation has been finished. According to the agile software development approach, testing is not considered a separate phase, but is an integral part of the development process, which is done along with coding.

There are several levels of testing, which are distinguished by the test target:

- *Unit testing* is used for testing the behaviour of small components of the application (units), such as classes, methods or modules. This allows localizing more precisely the source of the defect, and reducing the time for its fixing. Unit tests are developed and executed along with coding.
- *Integration testing* is applied to verify the interfaces between the components, after their integration.
- *System testing* is applied to test a completely integrated system, containing all the modules. The purpose of this testing is to verify that the system meets its requirements.
- *Acceptance testing* is a test performed by the customer in order to verify that the software meets the customer's expectations.

Different types of testing can be performed against the integrated systems. The two most important types are functional testing and non-functional testing. *Functional testing*, as the name suggests, is used to test the functionality of the application, and verify that for given inputs the output of the application is correct. *Non-functional testing*

term defines a whole group of types of tests, which are not related to the functionality of the system. These could be *performance testing* (testing responsiveness and stability of the system under a particular workload), *load testing* (testing system's behaviour under both normal and anticipated peak load conditions), *usability testing* and others.

Another important type of testing is *regression testing*. The idea behind this kind of testing is to ensure that changes such as enhancements, patches, or configuration changes has not introduced new bugs in existing functional and non-functional areas of a system.

There are different methods that can be used for software testing: black box, white box, and grey box. *White box* method deals with internal structures of an application, thus knowledge about the internal perspective of the system in order to design test cases, which cover all the paths through the software. *Black box* method uses an external perspective, and requires a software specification in order to define the test cases. Black box testing deals with inputs and outputs, without knowing about the internals of the execution, the objective is to check that given the input, the system produces the expected output. The third type, *gray box* testing, is the combination of the previous two methods. The tester applying grey box method partially knows the internal structure of the system and takes it into account, but, as in black box method, the testing is based on inputs and outputs. All these methods can be applied practically at each level, though it usually makes more sense to apply white box or grey box testing at unit level, and black box or grey box at system and integration level. Acceptance testing is always performed using black box approach.

4.4.2 Test cases for ORE

As the current project follows agile principles, the testing was performed along with the development, and the integration and system testing were performed at the end of each iteration.

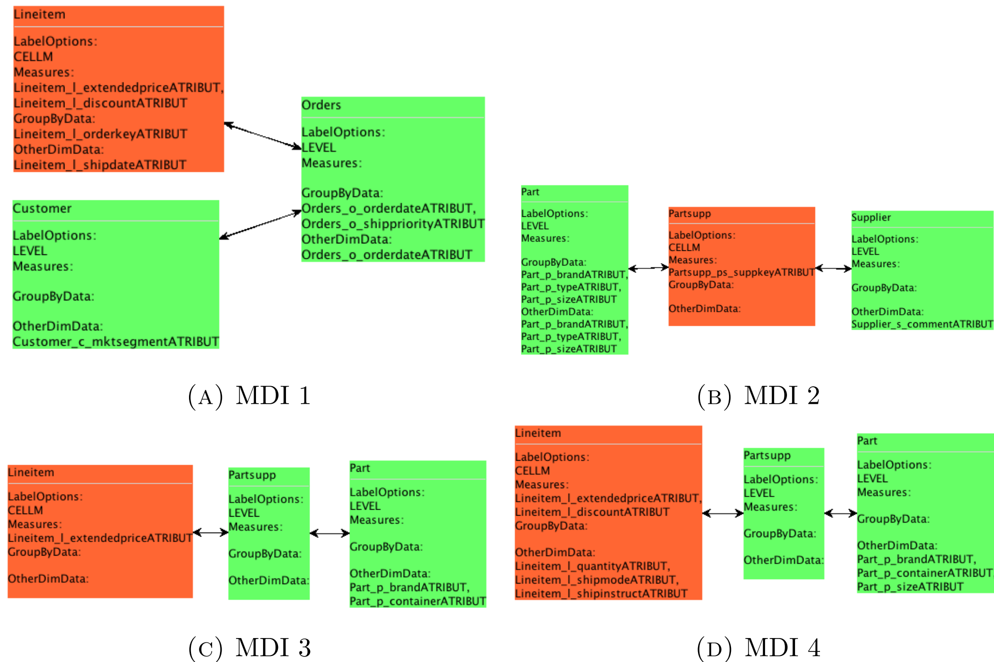
Although it is a good practice to distinguish the roles of tester and developer, but in this case it was not possible, so the two roles were performed simultaneously.

Unit testing was performed using both white box and black box approach. The code of the implemented algorithms was analyzed in order to determine all possible execution flows. At the integration and system level, on the other hand, the black box technique was preferred. The system was provided test input data (MDIs), and the output was checked for correctness.

Two different data sets were used for performing the tests.

One data set is based on **TPC-H benchmark**. TPC-H provides a database schema, which defines the source tables, relationships between them and the attributes. This schema was previously transformed into the corresponding OWL ontology. This transformation was made by Petar Jovanovic when developing the previous parts of the GEM framework.

An example using TPC-H ontology is presented below. In this example ORE is provided with four requirements, the corresponding MDIs of which are shown in Figure 4.17



(A) MDI 1 (B) MDI 2

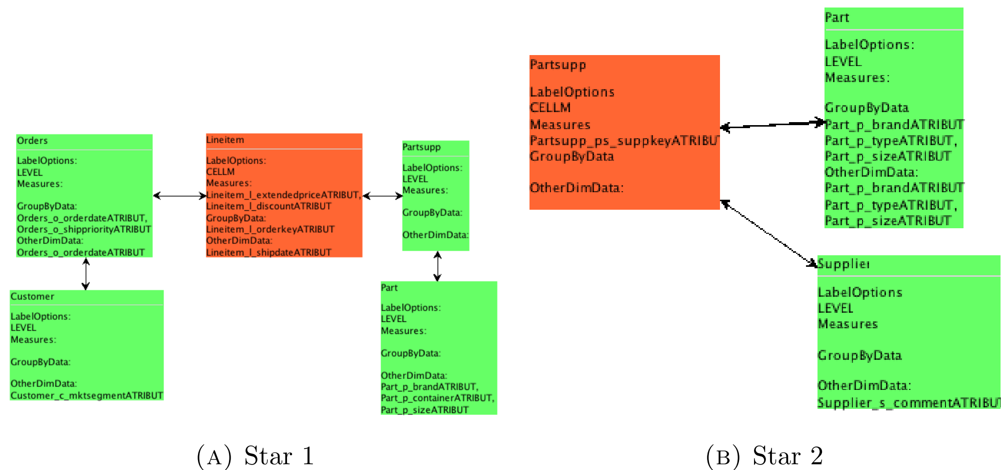
(C) MDI 3 (D) MDI 4

FIGURE 4.17: Test data using TPC-H ontology

The result of ORE execution is shown in Figure 4.18 (the result after the execution of Fact Matching and Dimension Matching stages). As it can be seen, an MD schema which consists of two stars is produced, one star having LineItem concept as fact, and the other – Partsup. Three MDIs which have LineItem as factual concept (4.17a, 4.17c and 4.17d) have been integrated in one star, and all the missing levels of the corresponding dimensions were inserted incrementally.

Figure Figure 4.19 shows the same MD schema, but after performing the integration stage of ORE. There, as can be observed, all levels of each dimension have been merged to form one denormalized dimension level, having all the attributes of all the levels of the dimension levels.

Another data set used for testing is based on the **Learn-SQL ontology**. The data set consisted of 13 requirements, the same that were used in the experiments performed on the initial prototype (see 4.2.1). This ontology provides a wider variety of relationships



(A) Star 1 (B) Star 2

Figure 4.18: Intermediate results for TPC-H ontology test

(A) Star 1

(B) Star 2

Figure 4.19: Final results for TPC-H ontology test

between the concepts (for example, taxonomy), thus giving possibility to test more complex cases. However, it was still not enough for covering all possible cases which may appear in the input and which ORE has to be able to process. For this reason, additional test cases were generated, in which the ontology was slightly modified in order to add concepts with "1 – 1" relationships (synonyms).

In fact, testing and debugging took a big part of the time spent on the project. The reason for that is that the data structures used in ORE are quite complex, and sometimes it was difficult to find the source of the incorrect behaviour. For the same reason it was not that easy to use automatic unit testing tools, like JUnit [15]. However, it is considered to develop automatic unit tests for the future development of the ORE component of the GEM framework. The debugging tool of NetBeans IDE, the integrated

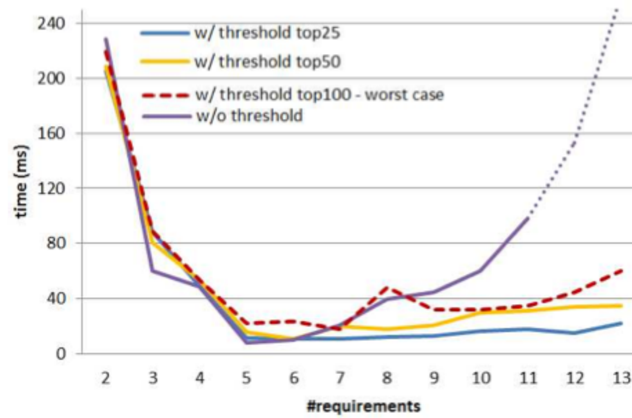


FIGURE 4.20: ORE execution time

development environment chosen for the project, has proven to be a very useful tool, which helped significantly in testing.

4.4.3 ORE experiments

After the development of the ORE system was finished, a set of experiments was conducted again, in the same way as it was made in the end of the first iteration (see 4.2.1). The experiments were performed by Petar Jovanovic and myself. The aim of the experiments was to scrutinize ORE for measuring its performance characteristics (time, computational complexity etc.), scalability characteristics (in terms of the number of requirements and produced results), results quality and other characteristics of the algorithm.

The same input data as before – 13 information requirements based on the Learn-SQL ontology – was chosen for these experiments, each requirement had from 1 to 8 MDIs. If all the combinations of MDIs were considered, the number of the solutions would reach 590000, or it could be even higher, if more than one solutions was considered for a pair of an MDI and a current solution. Dealing with such a big set of solutions is very expensive. In order to limit the solutions space, a cost-based approach based on the cost model is used in ORE, according to which only the top N solutions are selected for the further processing at each iteration. By adjusting the value of N, it is possible to analyze how ORE behaves in each case and whether it meets the quality objectives, i.e. the resulting MD schema has the minimal structural complexity.

In order to measure the performance of ORE in terms of time, a set of experiments with varying input loads and threshold (number of top solutions) were conducted. The results of these experiments are illustrated inFigure 4.20.

It can be observed that when the number of the solutions is not limited, after only 10 requirements (considering all the MDIs of those requirements) the time to iteratively integrate each requirement starts to grow rapidly and it later bursts for additional requirements. When setting a threshold (top N solutions based on structural complexity) according to the cost-based approach, it can be observed that the time tends to stay in a certain range depending on the value for N (the experiments performed for N = {25, 50, 100}). Figure 4.20 also shows that peaks may appear. This is due to the fact that the size of inputs (i.e. the number of MDIs per requirement) is not limited. However, the cost-based pruning keeps the problem manageable. It is worth mentioning that the analysis of the MD schemas produced as the result of the experiments showed that the optimal solution was always in top N, independent of the size of N.

Another interesting observation is the higher latency at the beginning of the process, when the integration of the first few requirements occurs. The explanation for this comes from the fact that in the beginning the ontology is accessed directly in order to explore the relationships between the concepts, and this is a costly operation. However, as the Transitive Closure Cache (TCC) is used in ORE (see 4.3.5), over time it is filled with more concepts, and in further iterations most of the concepts can be already found in the cache, thus avoiding the direct access.

The results of the experiments demonstrated the feasibility of the ORE approach and validated the cost-based approach, showing that setting a threshold for the top solutions maintains the linear complexity of the approach, regardless of the number of requirements in the input, while still keeping the optimal solution in the resulting solution space.

Chapter 5

Project planning and cost

5.1 Initial planning

As discussed in 4.1.3 agile approach was chosen for the elaboration of the project, which means that the development process is iterative and incremental. The development is split into several iterations, and new features are added to the software incrementally.

In order to make the planning of the project the key milestones were defined in such a way that each iteration of the software adds some important features. In the beginning of each iteration the analysis is performed and requirements are defined in more detail. The initial planning of the project is shown in Figures 5.1 and 5.2.

5.2 Final planning

During the process of the elaboration the ORE system, the requirements sometimes were changed or redefined, but the development mostly followed the original plan consisting of 8 iterations. A detailed description of what was developed at each iteration can be found in 4.2. However, the time length of the project significantly increased.

One of the reasons for that was a non-optimal planning of the elaboration of the project documentation. The tasks related to writing the document were overlapping with the development tasks. This approach is good in a sense that the documenting is performed along with the development, and it is not left for the end. However, I think that a more detailed planning with respect to the daily hours dedicated to development and documentation tasks could have worked better. As a result, the project was not finished at the estimated date (end of July).

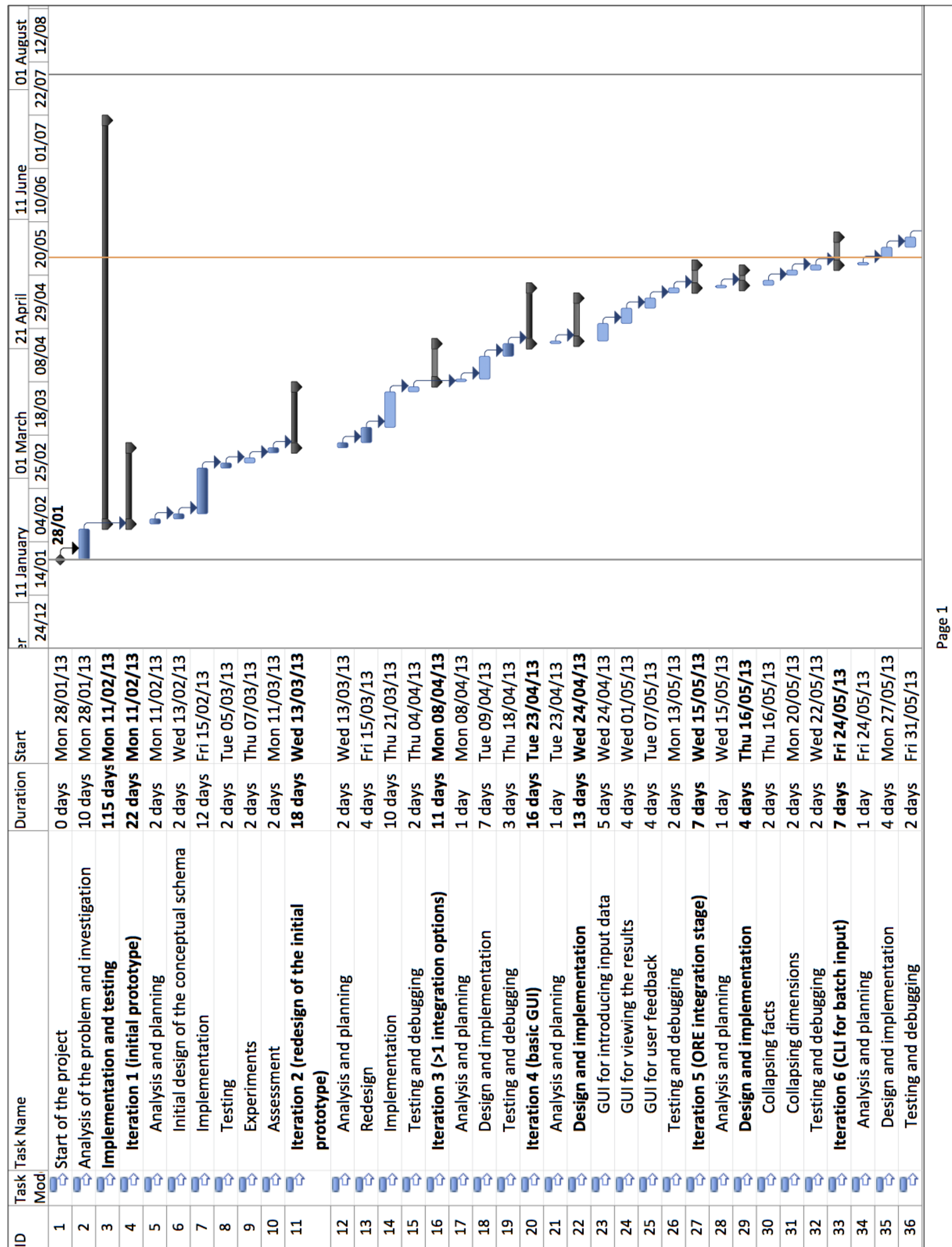


FIGURE 5.1: Initial planning. Page 1

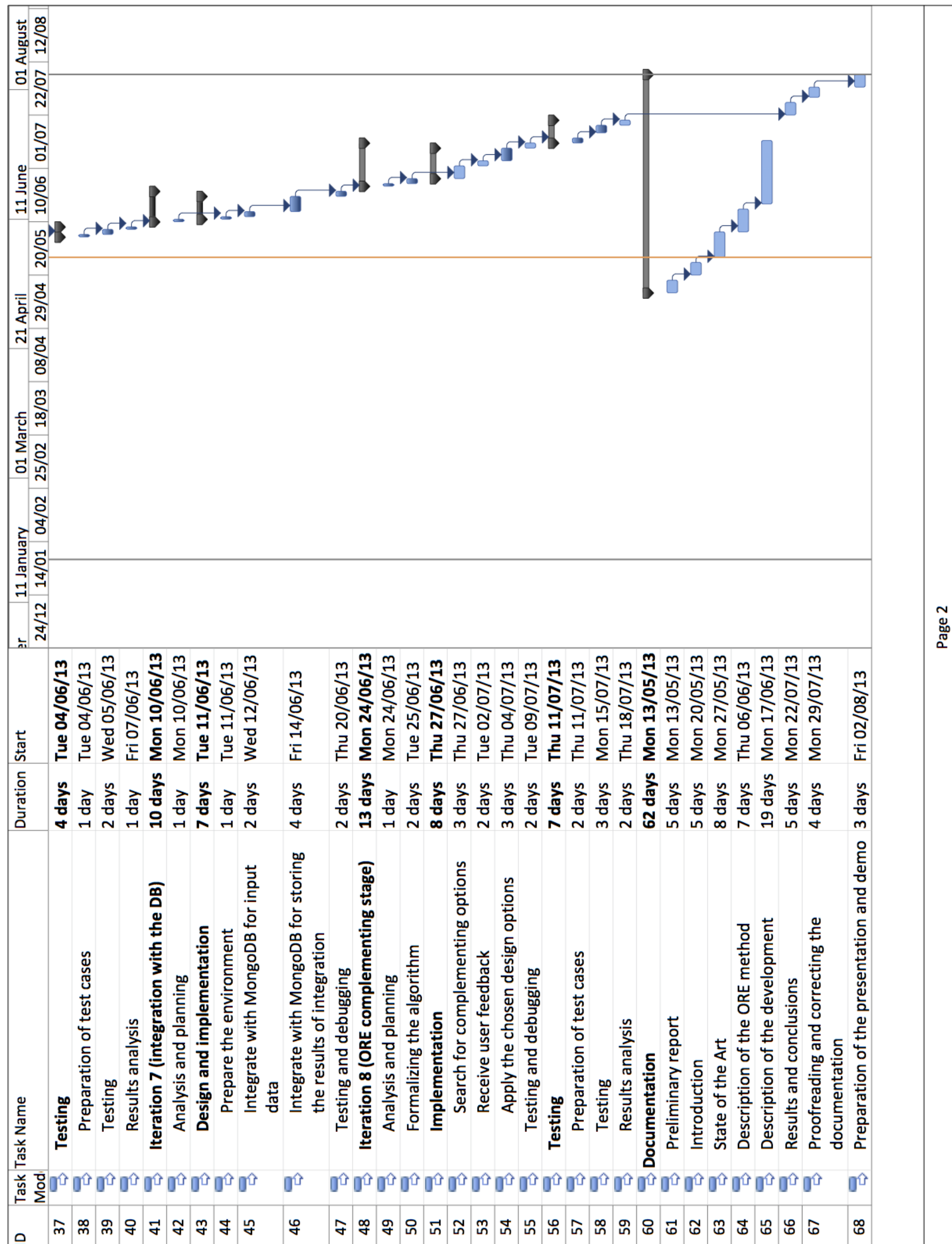


FIGURE 5.2: Initial planning. Page 2

For personal reasons I could dedicate only little time to working on the project in August. In September the software development was finished, but the documentation was still missing a significant part. Finishing of the documentation was complicated, as I started working full-time and couldn't devote much time to the thesis, which was the main reason of the delay in the finalizing the thesis.

5.3 Project cost estimation

In this section the cost of the project is estimated. Despite the length of the project turned out to be greater than in the initial planning, total dedication of hours was approximately the same. For that reason the estimation of cost will be performed using the initial planning, with the length of the project of 6 months.

The expenses which have to be taken into account include:

- **hardware resources**
  This includes the cost of the necessary equipment: personal computers, servers etc. As I as the developer used my personal laptop for the elaboration of the project, this resource didn't imply expenses.

- **software resources**
  The software used during the elaboration of the project was mostly free or open source: NetBeans IDE, LaTeX, svn, texmaker, Zotero.
  As a tool for designing UML diagrams I used Visual Paradigm for UML Community Edition, which is distributed for free. The Community Edition version has some limitations, for example, when more than one diagram is added to the project, when exporting the diagrams to image files the software adds a watermark, which makes the image not very well visible. However, there are two workarounds for this: either saving each diagram in a separate file, or using a screenshot tool to obtain the diagram in image format (i.e. .png, .jpg etc.), this way avoiding additional expenses.
  The non-free software which was used in the project included Microsoft Project and Microsoft Visio, which also required Microsoft Windows operating system installed. This could lead to significant expenses, but in my case being a student of UPC I could get this software free of charge for the period of the elaboration of the thesis.

- **office resources**
  This category covers the equipment of the working place, which includes a desk and

a chair, electricity and internet connection. I considered a monthly membership with one of the co-working spaces as the most suitable option for covering this need. For example, MOB Barcelona has a membership “Full Movil”, which includes an access to an individual working place from 9:30 to 19:30 from Monday to Friday, internet and printing service, and in addition 2 hours of meeting room a month. The monthly price of the service is 110 € + VAT (21 %), or 133.1 € total.

- **human resources**
  In order to estimate the costs of human resources the initial estimation of the length of the project is taken as a base.
  If we consider the project length of 137 days, and 8 hour daily dedication, it will count a total of **1096** hours. The wage rate of a developer of 9 €/hour seems reasonable in case when the worker contracted directly, and not through outsourcing, which would imply a substantially higher rate.

The total cost of the project is calculated in Table 5.1.

Table 5.1: Project costs

Resources	**Rate**	**Time/Period**	**Cost**
office resources	133.1 € /month	6 months	786.6 €
human resources	9 € /hour	1096 hours	9864 €
		Total	**10650.6 €**

Chapter 6

Conclusions

This chapter summarises the results of the elaboration of the master thesis. The master thesis was devoted to the implementation of the ORE method – an iterative approach to design the data warehouse multidimensional schema in a semi-automatic way, by integrating individual MD interpretations of the information requirements. The developed ORE application is integrated into the GEM framework, thus providing a complete solution for requirement-driven data warehouse design.

6.1 My contribution

My work on the thesis consisted of two parts: theoretical and practical. In theoretical part I familiarized myself better with the data warehousing and the problems, which exist in the field of the DW multidimensional design. I made a research of the existing methods which aim to solve these problems, and studied deeply the ORE approach, described in Chapter 3. In the practical part I performed the design and the implementation of the ORE method. The result of the implementation is a standalone application, which can be used in command-line mode, or in graphical mode, which provides intuitive and visual way of interaction for the user with help of GUI in the desktop environment.

Three of the four stages of the ORE method were implemented (excluding the optional complementing design stage). The developed ORE application was integrated with the GEM framework by means of a database. When designing the application, I followed low coupling principle, whenever possible, thus facilitating maintainability and changeability of the system in the future. Using the three-tier architecture also favours these software quality factors.

In the course of the implementation some requirements have been changed, or replaced with others, so the final feature set is a bit smaller than it was planned initially. But in general, the objectives defined for the project in the beginning, have been achieved.

6.2 Future work

While the GEM framework has made a significant step forward with the introduction of the ORE system, there are still a lot of things, which can be improved in the future. Here are some of them:

- ORE method implementation
  - In the Fact Matching algorithm the cases of the "0 – *" relationships between the facts have been omitted for this project. These cases should be implemented in the future iterations of the ORE application.
  - The Complementing design stage should be implemented.
  - This project has tackled the creation of the MD schema from the requirements. However, the problems of maintenance and evolution of the initial design haven't been covered completely. Currently, only adding new requirements is supported, however, if there is a need to remove a requirement, or change some parameters in the requirement, such changes can't be reflected automatically in the MD schema. The reconstruction of the final MD schema is needed in this case, which is performed by executing the ORE process for the new set of requirements. Implementing a more flexible and efficient way of schema evolution would be a great improvement.
- GEM framework
  - The improvement which can be made for the GEM framework is tighter integration between the system components, currently, GEM and ORE. This can be achieved by using a unified user interface, which would allow the user to execute all the steps of the process, from introducing the requirements to obtaining the final MD schema, in a single application. The work on the development of such unified interface was started, but not finished yet.
  - Another possible future direction consists in the implementation of the module, performing the integration of ETL processes for single information requirements, so that the resulting ETL match the MD design produced by

ORE. The approach for this, Coal, is introduced in [16]. The module implementing Coal method should be integrated with the GEM framework.

- Database improvements
  - Using a NoSQL database (MongoDB) provides flexibility in defining the structure of the stored data, and in the future it is possible to benefit from it, especially when dealing with evolution and changes of MD schemas. It might be useful to readdress the problem of translation from XML to JSON and the other way around. Avoiding storing the project artifacts as binary data, and treating them as objects instead could be very beneficial, as it would allow to change certain parts of the documents in an easy way, using the database query language, and also make it possible to use additional features such as searching by field values.

Bibliography

[1] Petar Jovanovic, Oscar Romero, Alkis Simitsis, and Alberto Abelló. ORE: an iterative approach to the design and evolution of multi-dimensional schemas. In *DOLAP*, pages 1–8, 2012.

[2] William H. Inmon. *Building the Data Warehouse*. John Wiley & Sons, Inc., New York, NY, USA, 1992. ISBN 0471569607.

[3] Oscar Romero and Alberto Abelló. A survey of multidimensional modeling methodologies. *IJDWM*, 5(2):1–23, 2009.

[4] Oscar Romero, Alkis Simitsis, and Alberto Abelló. GEM: requirement-driven generation of ETL and multidimensional conceptual designs. In *DaWaK*, pages 80–95, 2011.

[5] Oscar Romero and Alberto Abelló. A framework for multidimensional design of data warehouses from ontologies. *Data Knowl. Eng.*, 69(11):1138–1157, 2010.

[6] Petar Jovanovic. *Integration of Multidimensional and ETL design*. Master thesis, Universitat Politèctica de Catalunya, 2011.

[7] TPC-H, 2013. URL `http://www.tpc.org/tpch/`.

[8] Markus Blaschka, Carsten Sapia, and Gabriele Höfling. On schema evolution in multidimensional databases. In *Proceedings of the First International Conference on Data Warehousing and Knowledge Discovery*, DaWaK '99, page 153–164, London, UK, UK, 1999. Springer-Verlag. ISBN 3-540-66458-0.

[9] Manuel Angel Serrano, Coral Calero, Houari A. Sahraoui, and Mario Piattini. Empirical studies to assess the understandability of data warehouse schemas using structural metrics. *Software Quality Control*, 16(1):79–106, March 2008. ISSN 0963-9314. doi: 10.1007/s11219-007-9030-7. URL `http://dx.doi.org/10.1007/s11219-007-9030-7`.

[10] Dean Leffingwe. *Scaling Software Agility: Best Practices for Large Enterprises.*

[11] The agile manifesto. URL www.agilemanifesto.org.

[12] The scrum guide. the definitive guide to scrum: The rules of the game. URL https://www.scrum.org/Portals/0/Documents/Scrum%20Guides/2013/Scrum-Guide.pdf.

[13] The rules of extreme programming. URL http://www.extremeprogramming.org/rules.html.

[14] Oscar Romero and Alberto Abelló. MDBE: automatic multidimensional modeling. In *Proceedings of the 27th International Conference on Conceptual Modeling*, ER '08, page 534–535, 2008.

[15] JUnit: a programmer-oriented testing framework for java. URL www.junit.org.

[16] Petar Jovanovic, Oscar Romero, Alkis Simitsis, and Alberto Abelló. Integrating ETL processes from information requirements. In *DaWaK*, pages 65–80, 2012.

[17] Java (programming language). URL http://en.wikipedia.org/wiki/Java_(programming_language).

[18] About the java technology, 2013. URL http://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html.

[19] Gartner IT-glossary. URL http://www.gartner.com/it-glossary/.

[20] Oscar Romero and Alberto Abelló. Multidimensional design methods for data warehousing. In *Integrations of Data Warehousing, Data Mining and Database Technologies*, pages 78–105. 2011.

[21] Ines Gam, Camille Salinesi, et al. A requirement-driven approach for designing data warehouses. *Requirements Engineering: Foundation for Software Quality (REFSQ)*, 2006.

[22] Petar Jovanovic, Oscar Romero, Alkis Simitsis, and Alberto Abelló. Requirement-driven creation and deployment of multidimensional and ETL designs. In *ER Workshops*, pages 391–395, 2012.

[23] Oscar Romero and Alberto Abelló. Automatic validation of requirements to support multidimensional design. *Data Knowl. Eng.*, 69(9):917–942, 2010.

[24] NoSQL for scale, agility and ROI. URL http://www.mongodb.com/learn/nosql#mongodb.

[25] MDBE demonstration. URL http://www.essi.upc.edu/~oromero/MDBE.html.

[26] Apache jena – getting started. URL http://jena.apache.org/getting_started/index.html.

[27] Software development methodology – wikipedia, . URL http://en.wikipedia.org/wiki/Software_development_methodology.

[28] Software development process – wikipedia, . URL http://en.wikipedia.org/wiki/Software_development_process.

[29] Andrew Phillips. Software development methodologies. URL http://www.codeproject.com/Articles/124732/Software-Development-Methodologies.

[30] The home of scrum. URL www.scrum.org.

Glossary

Agile software development

a group of software development methods based on iterative and incremental development.

BSON

a binary-encoded serialization of JSON-like documents.

Business Intelligence

an umbrella term that includes the applications, infrastructure and tools, and best practices that enable access to and analysis of information to improve and optimize decisions and performance.

Data Warehouse

a subject-oriented, integrated, time-variant and non-volatile collection of data in support of managements decision making process.

IDE

Integrated development environment – a software application that provides comprehensive facilities to computer programmers for software development.

JSON

an open standard format that uses human-readable text to transmit data objects consisting of attributevalue pairs.

Ontology

a formal representation of knowledge, consisting of a set of concepts within a domain, using a shared vocabulary to denote the types, properties and interrelationships of those concepts.

OWL

Web Ontology Language – is a family of knowledge representation languages or ontology languages for authoring ontologies or knowledge bases, facilitating greater

machine interpretability of the content by providing additional vocabulary along with a formal semantics.

Snowflake schema

a variation of the Star schema in which the dimensional tables from a star schema are organized into a hierarchy by normalizing them.

Star schema

a multidimensional model which consists of a fact table with a single table for each dimension.

XML

a markup language that defines a set of rules for describing data using embedded tags, such that that is both human-readable and machine-readable.

Acronyms

BI
: Business Intelligence.

BSON
: Binary JSON.

CLI
: command-line interface.

DB
: database.

DBMS
: database management system.

GUI
: graphical user interface.

IDE
: Integrated development environment.

JSON
: JavaScript Object Notation.

MD
: multidimentional.

MDI
: multidimentional interpretation.

RDBMS
: relational database management system.

TCC

Transitive Closure Cache.

VAT

Value Added Tax.

XML

Extensible Markup Language.