Universitat d'Alacant
Universidad de Alicante

Data Warehouse Design with UML

PhD Thesis

Sergio Luján-Mora

Advisor: Juan Trujillo

Department of Software and Computing Systems
University of Alicante

June 2005

Preface

Everything started in Argentina...

In the summer of 1998, I finished my Degree in Computer Science. I did not want to take the obvious next step: find a job and start to work for the rest of my life. So, I applied for a grant of the Spanish Agency of International Cooperation (*Agencia Española de Cooperación Internacional*[1]) and I chose three destinations: two in Argentina and one in Chile. As was to be expected, my mother told me I was mad and she asked me what I had lost in South America.

Fortunately, I got the grant and I spent two months in General Pico (La Pampa, Argentina). During those two months, I made some research on distributed databases with Dr. Hugo Alfonso at the *Universidad Nacional de la Pampa*[2]. I had such a good time, I ate many *churrascos* (barbecued meat) and *dulce de leche* (caramelized condensed milk), and I met very nice people that, obviously, I wanted to repeat next year. But I had to be related to the University in some way.

Therefore, I enroled the PhD courses at the Department of Software and Computing Systems (*Departamento de Lenguajes y Sistemas Informaticos*[3]) at the University of Alicante. And what you have in your hands is the result of that simple and fast decision. By the way, if you want to know it, the following year I did not go to Argentina and little by little, I lost the contact with the friends I left there. As John Lennon said, "*Life is what happens while you are busy making other plans*".

Seven years later, the end of the PhD is a good opportunity to take a look backwards and meditate about those seven years. During this journey, I have lost and won a lot of things. When I started to work at the Department of Software and Computing Systems (my second home due to the time I spend there), I asked Juan, who would be my future PhD advisor, "*Do we travel here?*". And, of course, we travel a lot here. Thanks to the papers I have got published, I have visited

[1] http://www.aeci.es/
[2] http://www.unlpam.edu.ar/
[3] http://www.dlsi.ua.es/

Brazil, England, France, China, Czech Republic, Finland, Germany, Austria, Greece, Portugal, Turkey, and USA.

Briefly, my last seven years can be divided in three parts. From 1998 to 2000, I attended the PhD program Linguistic Engineering, Automatic Learning, and Pattern Recognition at the Department of Software and Computing Systems, and I had to take eight different courses: Computational Learning, Introduction to Natural Language Processing, How to Write and Publish a Scientific Paper, Neural Networks for Sequence Processing, Automatic Translation: Foundation and Applications, Logic Programming, Information Extraction for Database Insertion, and Laboratory of Language Processing and Information Systems. From 2000 to 2001, I worked in my project for the PhD program and published my first papers in international conferences. Then, in the middle of 2001, I started to work in data warehouses until nowadays.

I owe my appreciation to many great people for contributing to this work. First of all, I would like to thank my family for their incredible support, understanding, and patience. There are no words for their help.

I would like to express words of gratitude for my advisor, Dr. Juan Trujillo, not only for his guidance in developing the present work, but also for being so supportive, encouraging, and understanding in good and bad times. The last years have been very difficult.

I would also like to show appreciation to Panos Vassiliadis, my supervisor during the four months I stayed in Greece for my PhD. During those months, we started a collaboration that continues nowadays.

Last, but certainly not least, I would like to thank my department colleagues. The list is too long, so I would like to highlight only two people. I give my thanks to Armando Suárez, because he wrote a recommendation letter to my grant for Argentina; it was a little effort, but a great contribution. And I also express my gratitude for Manolo Palomar, because he offered me the opportunity to be part of the Department of Software and Computing Systems.

Of course, I also give my thanks to Mario Piattini (University of Castilla-La Mancha, Spain), Il-Yeol Song (Drexel University, USA), Panos Vassiliadis (University of Ioannia, Greece), Jens Lechtenbörger (University of Münster, Germany), and Jaime Gómez (University of Alicante, Spain) for revising this PhD thesis and accepting being part of the jury. I am also grateful to the external reviewers Jorge Bernardino (Polytechnic of Coimbra, Portugal) and Timos Sellis (National Technical University of Athens, Greece). All of them sent useful, constructive, and instructive comments.

Finally, as we say in Spain, "*a door closes, and another opens*"...

Summary Table of Contents

Table of Contents

List of Figures

List of Tables

List of Acronyms

API *Application Program Interface*
A set of routines, protocols, and tools for building software applications. A good **API** makes it easier to develop a program by providing all the building blocks and a programmer "only" has to put the blocks together.

CASE *Computer Aided Software Engineering*
A category of software that provides tools to automate, manage and simplify the development process of software.

CRM *Customer Relationship Management*
CRM entails all aspects of interaction a company has with its customer, whether it be sales or service related. New technologies help to manage electronically the relationships with customers.

CWM *Common Warehouse Metamodel*
CWM is an **OMG** specification whose purpose is to enable easy interchange of metadata between data warehousing tools and metadata repositories in distributed heterogeneous environments. CWM is based on three key industry standards: **UML**, **MOF**, and **XMI**.

DBMS *Database Management System*
Software that enables you to store, modify, and extract information from a database. There are many different types of **DBMS**, ranging from small systems that run on personal computers to huge systems that run on mainframes. There exist different **DBMS**, mainly relational, network, flat, and hierarchical.

See also **RDBMS**.

DM *Data Mart*
Whereas a **DW** combines databases across an entire enterprise, **DM** are usually smaller and focus on a particular subject or

department. Some **DM**, called *dependent data marts*, are subsets of larger **DW**. **DM** are also called *high performance query structures* by some authors.

See also **DW**.

DSS *Decision Support System*
An interactive computerized system that gathers and presents data from a wide range of sources, typically for business purposes. **DSS** applications help people make decisions based on data that is gathered from a wide range of sources.

DTD *Document Type Definition*
A **DTD** defines the tags and attributes that can be used in an **SGML**, **XML** or **HTML** document. Moreover, a **DTD** also indicates which tags can appear within other tags. Due to the limitations of the simple structuring and typing mechanisms in **DTD**, XML Schema has been defined as a substitute.

See also **XML**.

DW *Data Warehouse*
A collection of data designed to support management decision making about their business. **DW** contain a wide variety of data that present a coherent picture of business conditions at a single point in time. Bill Inmon defines a **DW** as "*a subject-oriented, integrated, time-variant, nonvolatile collection of data in support of management's decisions*".

See also **DM**.

EER *Extended Entity-Relationship*
The **EER** includes all **ER** semantics but it uses additional semantic modeling concepts. For example, **EER** models can show subclasses, superclasses, specializations, generalizations, and categorization.

See also **ER**.

ER *Entity-Relationship*
The **ER** model was originally proposed by Peter Chen in 1976 as a way to unify the network and relational database views. The **ER** model is a conceptual data model that views the real world as entities (with attributes) and relationships.

See also **EER**.

ETL *Extraction, Transformation, Loading*
Process that extracts data out of one data source (most of the times a database) and load it into a data target. These activities can be defined as follows:

- *Extract*: the process of reading data from a data source.
- *Transform*: the process of converting the extracted data from its previous form into the form it needs to be in so that it can be placed into the data target. Transformation occurs by using rules or lookup tables or by combining the data with other data.
- *Load*: the process of writing the data into the data target.

HOLAP *Hybrid OLAP*
A kind of **OLAP**. **HOLAP** products combine **MOLAP** and **ROLAP**. With **HOLAP** products, a relational database stores most of the data. A separate **MD** database stores the most dense data, which is typically a small proportion of the data.

See also **MOLAP**, **OLAP**, **ROLAP**.

HTML *HyperText Markup Language*
The language used to create documents on the **WWW**. Basically, **HTML** is similar to **SGML**, although it is not a strict subset. **HTML** defines the structure and layout of a web document by using a variety of tags and attributes. **W3C** standardizes **HTML**.

See also **SGML**.

HTTP *HyperText Transfer Protocol*
The underlying protocol used by the **WWW**. **HTTP** defines how messages are formatted and transmitted, and what actions web servers and browsers should take in response to various commands. Currently, **HTTP** 1.1 (**RFC** 2616, June 1999) is the last version of this protocol.

ISO *International Organization for Standards*
An international organization composed of national standards agencies from over 75 countries. For example, some of its members are ANSI (United States of America), BSI (Great Britain), AFNOR (France), DIN (Germany), and UNE (Spain).

MD *Multidimensional*
Applies to any system that is designed for analyzing large groups of records organized in a dimensional space. For example, **MD** databases are organized around groups of records that share a common field value. **OLAP** is a related term.

See also **MOLAP**.

MDA *Model Driven Architecture*
An approach to application design and implementation. As

defined by the **OMG**, "*MDA is a way to organize and manage enterprise architectures supported by automated tools and services for both defining the models and facilitating transformations between different model types*".

Software development in the **MDA** starts with a **PIM** of an application. This model remains stable as technology evolves. Then, **MDA** development tools convert the **PIM** first to a **PSM** and finally to a working implementation (code).

See also **PIM**, **PSM**.

MOF *Meta Object Facility*
A standard from **OMG** that can be used to define and manipulate a set of interoperable meta-models and their corresponding models.

MOLAP *Multidimensional OLAP*
A kind of **OLAP**. **MOLAP** provides **MD** analysis of data by putting data in a cube structure. Most successful **MOLAP** products use a multicube approach in which a series of small, dense, precalculated cubes make up a hypercube.

See also **HOLAP**, **OLAP**, **ROLAP**.

OCI *Oracle Call Interface*
A low-level **API** used to interact with Oracle databases. **OCI** provides access to all of the Oracle Database functionality.

OCL *Object Constraint Language*
An expression language that can be used to define expressions in **OO** models, in particular **UML** models. These expressions enrich the respective model with precise and unambiguous annotations, thus preserving precious information about the underlying business domain.

ODBC *Open Data Base Connectivity*
A connection method to data sources (databases, plain text files, spreadsheet files, etc). It requires that you set up a data source using a specific driver. Most database systems support **ODBC**.

OLAP *OnLine Analytical Processing*
A category of software tools that provides analysis of data stored in a database. **OLAP** tools enable users to analyze data thanks to special functions that manipulate data. Vendors offer a variety of **OLAP** products that are grouped into three categories: **ROLAP**, **MOLAP**, and **HOLAP**.

See also **HOLAP**, **MOLAP**, **ROLAP**.

Contrast with **OLTP**.

OLE *Object Linking and Embedding*
A technology by Microsoft Corporation. **OLE** enables you to create objects with one application and then link or embed them in a second application. Embedded objects retain their original format and links to the application that created them.

OLEDB *Object Linking and Embedding DataBase*
The successor to **ODBC**, a set of software components that allow an application to connect with a data source, such as flat files, **DBMS**, etc.

OLTP *OnLine Transaction Processing*
The set of activities and systems associated with entering data reliably into a database. According to Ralph Kimball, "*The point of transaction processing is to process a very large number of tiny, atomic transactions without losing any of them*". Most frequently used with reference to relational databases, although **OLTP** can be used generically to describe any transaction processing environment.

Contrast with **OLAP**.

OMG *Object Management Group*
An international consortium that aims to provide a common framework for developing applications using **OO** programming techniques. **OMG** is responsible for different specifications, such as CORBA, **MOF**, **UML**, etc.

OO *Object Oriented*
A term that is generally used to describe a system that deals primarily with different types of objects, and where the actions you can take depend on what type of object you are manipulating. **OO** can mean different things depending on how it is being used:

- Object Oriented Language.
- Object Oriented Programming.
- Object Oriented Graphics.
- Object Oriented Database.

PIM *Platform Independent Model*
One of the models of the **MDA**. A highly abstracted model that is independent of any implementation technology. It describes a software system that supports a part, or the whole, of

business. The software system is modeled from the perspective of how it best supports the business. The type of technology on which the **PIM** is to be implemented does not form a part of the software system at all.

PSM *Platform Specific Model*
One of the models of the **MDA**. The **PIM** is transformed into one or more **PSM**, which describes in detail how the **PIM** is implemented on a specific platform, or technology.

QVT *Query View Transformation*
An **OMG** initiative that aims to provide a standard for expressing model transformations. The transformations are defined precisely in terms of the relationships between a source **MOF** metamodel and a target **MOF** metamodel.

RAID *Redundant Array of Inexpensive Disk*
A category of disk drives that employ two or more drives in combination for fault tolerance and performance. There are different **RAID** levels, such as level 0 (Striped Disk Array without Fault Tolerance), level 1 (Mirroring and Duplexing), level 2 (Error-Correcting Coding), level 3 (Bit-Interleaved Parity), etc.

RDBMS *Relational Database Management System*
A **DBMS** that supports the relational model. In a **RDBMS** data is stored in the form of related tables.

See also **DBMS**.

REI *Rose Extensibility Interface*
The mechanism that Rational Rose provides to extend and customize its capabilities. **REI** allows the user to customize menus, define new stereotypes and properties, execute Rose Scripts, etc.

RFC *Request For Comments*
A series of documents about the Internet, started in 1969. Each **RFC** is designated by a number. If a **RFC** gains enough interest, it may evolve into an Internet standard.

ROLAP *Relational OLAP*
A kind of **OLAP**. **ROLAP** products adapt traditional relational databases to support **OLAP**. **ROLAP** is based on Ralph Kimball's star schema.

See also **HOLAP**, **MOLAP**, **OLAP**.

SEP *Software Engineering Process*
SEP, also known as *Software Development Process*, defines a process in which user requirements are transformed into software.

SGML *Standard Generalized Markup Language*
A system for organizing and tagging elements of a document. **SGML** was developed and standardized by the **ISO** in 1986. **SGML** is used widely to manage large documents that are subject to frequent revisions and need to be printed in different formats.

See also **HTML**.

SQL *Structured Query Language*
A standardized query language for requesting information from a database. There are different dialects of **SQL** because every **DBMS** vendor defines extensions.

UML *Unified Modeling Language*
According to the UML Specification, "*The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. The UML offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components*". **UML** models can be exchanged between software tools as streams or files with **XMI**.

UP *Unified Process*
The *Unified Software Development Process*, also known as **UP**, is an industry standard **SEP** from the authors of the **UML**. Whereas the **UML** defines a visual modeling language, the **UP** specifies how to develop software using the **UML**.

URL *Universal Resource Locator*
URL, also known as *Uniform Resource Locator*, is the global address of documents and other resources on the **WWW**.

W3C *World Wide Web Consortium*
An international consortium of companies involved with the Internet and the **WWW**. It was founded in 1994 by Tim Berners-Lee, the "father" of the **WWW**. The purpose of this consortium is to develop open standards, such as **HTML** or **XML**.

WWW *World Wide Web*
WWW, also known as the Web, is a worldwide computer

system connected through the Internet. The communication protocol is called **HTTP**, the **URL** is used to locate a resource, and the documents are formatted in a markup language called **HTML** that supports links to other documents, as well as graphics, audio, and video files.

XHTML *eXtensible HyperText Markup Language*
HTML written following the rules and formats that **XML** marks. **XHTML** is a markup language written in **XML**; therefore, it is an **XML** application.

XMI *XML Metadata Interchange*
An **XML** application that facilitates the standardized interchange of object models and metadata among tools and applications from multiple vendors. **XMI** is based on three industry standards: **XML**, **UML**, and **MOF**.

XML *Extensible Markup Language*
Metalanguage based on the **SGML** and developed by the **W3C**. It was initially designed especially for the **WWW**, but nowadays **XML** is used in other scenarios. **XML** allows designers to create their own customized tags.

See also **DTD**.

XSL *Extensible Stylesheet Language*
A language for expressing stylesheets. **XSL** defines how an **XML** document should be styled, laid out, and paginated onto some presentation medium, such as a window in a web browser or a page in a book.

XSLT *Extensible Stylesheet Language Transformations*
A language for transforming **XML** documents into other **XML** documents. **XSLT** is designed for use as part of **XSL**, which is a stylesheet language for **XML**. Whereas **XSL** specifies the styling of an **XML** document, **XSLT** describes how the document is transformed into another **XML** document that uses the formatting vocabulary. **XSLT** is also designed to be used independently of **XSL**. However, **XSLT** is not intended as a completely general-purpose **XML** transformation language.

Chapter 1

What Do We Have Here?

In this chapter, the content and rationale of this thesis is introduced with a brief description of every chapter and appendix. Moreover, some basic questions that we think the reader of this thesis can consider are answered. Finally, some typographic conventions that are used throughout this thesis are explained.

Contents

1.1 What is The Problem?

During the last ten years, the interest to analyze data has increased significantly, because the competitive advantages that information can provide for the decision-making process. Nowadays, a key to survival in the business world is being able to analyze, plan and react to changing business conditions as fast as possible.

Many organizations own billions of bytes of data, but they suffer different problems that make difficult to take advantage of data: data are spread through different computer systems, data from different sources are incompatible, data are available too late, etc. In order to solve these problems, new concepts and tools have evolved into a new information technology called *data warehousing*.

Data Warehouse **(DW)** projects are expensive: they often need years to implement correctly and require millions of dollars in hardware, software, and consulting services.

The sales of **DW** and related products keep growing year after year. The **DW** tool market reached $7.9 billion in 2003 and experienced 11 percent growth in that year, more than three times the growth rate of the previous year [137]. Meanwhile, and according to [89], the ***OnLine Analytical Processing*** **(OLAP)** market grew from $1 billion in 1996 to $4.3bn in 2004 and showed an estimated growth at 15.7 percent in 2004.

Although a lot of advances have been achieved in the field of **DW**, nowadays there is not standard method or data model for the design of **DW**. On the other hand, various reports suggest that about 40-50% of **DW** projects fail [140, 32]. Therefore, a new **DW** method based on standards may help to develop **DW**.

1.2 Why Not Use the Entity Relationship Model?

The traditional data models and techniques, such as the well-known ***Entity-Relationship*** **(ER)** [25] and the different extensions of **ER** [126], are not appropriate for **DW** design, due to the complexity of the corresponding models. Different authors have highlighted this problem. For example, Ralph Kimball states in [63]:

> Entity relation data models are a disaster for querying because they cannot be understood by users and they cannot be navigated usefully by DBMS software. Entity relation models cannot be used as the basis for enterprise data warehouses.

However, later data models adapted for **DW**, such as the famous

For more information about the *data models for data warehouses*, consult chapter 3, pp. 17.

star schema of Ralph Kimball [63], neither they are able to consider the main peculiarities of **DW**. Moreover, every approach has its own set of symbols and terminology, resulting in a lot of confusion and frustration.

1.3 What is the Goal of this Thesis?

The aim of this thesis is to define a method that allows the designer to tackle the different phases and steps in the design of a **DW**. Our approach is based on three complementary parts:

- The visual modeling language we use is an extension of the ***Unified Modeling Language* (UML)** [97], an ***Object Oriented* (OO)** modeling language that has been widely accepted.
- The method we propose is based on the ***Unified Process* (UP)** [59], that guides us how we perform **OO** analysis and design.
- The use of standards in the development of a **DW**, such as **UML**, ***Extensible Markup Language* (XML)** [143], **OO** databases [22] and object-relational databases [11].

Moreover, one important requirement of our work is to define a method with a set of models that can be used by the **DW** designer to communicate the design to the end user.

1.4 Why UML?

Instead of defining a new modeling language, we propose the use of **UML**, a widely accepted **OO** modeling that unifies the methods most used around the world. **UML** combines elements from the three major **OO** design methods: Rumbaugh's OMT modeling [110], Booch's OO Analysis and Design [17], and Jacobson's Objectory [60].

UML definition: see UML (Foreword, XXV).

The **UML** Specification [97] defines:

> The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. The UML offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components.

The UML defines a rich set of graphical diagrams:

- Use case diagram.
- Class diagram.
- Behavior diagrams: statechart diagram, activity diagram, interaction diagrams (sequence diagram, collaboration diagram).
- Implementation diagrams: component diagram, and deployment diagram.

We consider that the use of **UML** as the modeling language of our approach is the best option *nowadays*[1]. This choice can be justified along at least five considerations:

- **UML** follows the **OO** paradigm, which has been proved to be semantically richer than other paradigms because **OO** models are closer to the user conception [1].
- Nowadays **UML** is a well-known language for software engineers. Therefore, any approach based on the **UML** will minimize the effort of developers in learning new notations or methodologies for every subsystem to be modelled.
- The use of a **UML** profile makes the design easier, because **DW** designers can use the concepts (fact, dimension, etc.) they are used to apply. In this manner, designers do not need to understand the entire **UML**. Therefore, **DW** designers can take advantage of the **UML** without having to be experts in **UML**.
- **UML** is an standard of the ***Object Management Group*** **(OMG)** and unifies many years of effort in **OO** analysis and design.
- **UML** has been widely accepted by the scientific and industrial communities and "*has emerged as the software industry's dominant modeling language*" [66]. Nowadays, there are many ***Computer Aided Software Engineering*** **(CASE)** tools that support **UML**.

1.5 Why Do We Extend UML?

In this research work, we propose the use of **UML** to design **DW**. Although **UML** is a general modeling language, there are some situations in which it needs to be customized to specific problem domains.

[1] **UML** also has many detractors. For example, in [114] the author states "*we dare to classify the UML as a modern dinosaur: It is a semantically retarded, mighty ruler oppressing the development of sophisticated methods for conceptual modelling and information system design*".

For more information about *UML extension mechanisms*, consult appendix C, pp. 217.

Nevertheless, an outstanding feature of the **UML** is that it is an extensible language in the sense that it provides mechanisms (stereotypes, tagged values, and constraints) to introduce new elements for specific domains if necessary, such as web applications, database applications, business modeling, software development processes, etc. In the **UML** jargon, a collection of enhancements that extend an existing diagram type to support a new purpose is called a *profile*.

Therefore, in this thesis we define four **UML** profiles for accurately modeling different aspects of **DW**: the *UML Profile for Multidimensional Modeling*, the *Data Mapping Profile*, the *ETL Profile*, and the *Database Deployment Profile*.

1.6 What is a Method?

There is some confusion around the terms "methodology" and "method". Firstly, both terms are poorly defined and are used very loosely and yet are used very extensively [13]. Secondly, methodology is often used when what is actually referred to is method [29]. Therefore, in this work we use the word method, as it refers to a specific way of approaching and solving problems, whereas methodology is the study of the methods. Normally, **UML** community uses the word "process" as synonym of method.

Basically, "*A method is an explicit way of structuring one's thinking and actions*" [39]. Specifically, a method (process) "*defines* who *is doing* what when *and* how *to reach a certain goal*" and is "*the total set of activities needed to transform a customer's requirement into a consistent set of artifacts representing a software product and –at a later point in time– to transform changes in those requirements into new versions of the software product*" [59].

On the other hand, the right characteristics of a method have not formally defined. In [122], what a good "method(ology)" should be is stated:

- *Modular*: divided into components, or steps, which can be included or excluded, depending on the requirements.
- *Scalable*: equally applicable to any size project, from the smallest customers to the largest.
- *Sequential*: start at the beginning and go to the end. The steps of the methodology should provide for a structured approach to implementation project planning and should be clearly delineated.
- *Comprehensive*: cover everything. A good methodology will not require the client to think of "the other things". It will

include all possible items that could be required for a successful implementation.

- *Flexible*: the methodology must be capable of being rearranged and adjusted to meet the client-specific requirements.

1.7 Structure of the Thesis

This thesis is divided into 12 chapters and 5 appendixes. The content of this thesis is organized so the readers do not have to read all the chapters to get the information they need. Therefore, some concepts are repeated across several chapters. Moreover, extensive cross-references point to other related sections in order to allow the reader to elaborate on the content.

The following list briefly describes each chapter and appendix:

- Chapter 2 (**Introduction to Data Warehouses**) provides a brief introduction to data warehousing and related technologies.
- Chapter 3 (**Related Work**) discusses the related work about ***Multidimensional* (MD)** modeling, **DW** design, ***Extraction, Transformation, Loading* (ETL)** process, and other related issues.
- Chapter 4 (**A Data Warehouse Engineering Process**) introduces our *Data Warehouse Engineering Process*, an engineering process based on **UP**.
- Chapter 5 (**Conceptual Modeling of Data Sources**) is dedicated to conceptual modeling of data sources.
- Chapter 6 (**Multidimensional Modeling in Data Warehouses**) covers our **OO MD** modeling approach, based on the **UML**, for the conceptual modeling of **DW**; our approach considers major relevant **MD** properties at the conceptual level in an elegant and easy way.
- Chapter 7 (**Data Mapping Diagrams for Data Warehouses**) presents a framework for the design of data mapping diagrams at the conceptual level.
- Chapter 8 (**Logical Modeling of Data Sources and Data Warehouses**) addresses the design at the logical level.
- Chapter 9 (**Modeling ETL Processes in Data Warehouses**) describes how to model **ETL** processes with **UML** thanks to a set of mechanisms that represent the common operations in **ETL** processes, such as the integration of different

data sources, the transformation between source and target attributes or the generation of surrogate keys.

- Chapter 10 (**Physical Modeling of Data Warehouses**) introduces a proposal for the modeling of the physical design of **DW**.
- Chapter 11 (**Contributions**) summarizes our main contributions.
- Chapter 12 (**Conclusions and Future Work**) presents the main conclusions and the future work we plan to carry out in a short, medium, and long term.
- Appendix A (**Advantages of the UML Profile for Multidimensional Modeling**) discusses some of the main advantages of our **MD** modeling approach presented in chapter 6.
- Appendix B (**UML Particularities**) clarifies some **UML** mechanisms not normally used but that we use in our proposal.
- Appendix C (**Extension Mechanisms in UML**) presents the *UML Extensibility Mechanism* and explains how to define a **UML** profile.
- Appendix D (**Multidimensional Model Representation with XML**) explains how to use **XML** and the different technologies it comprises to represent **MD** models.
- Appendix E (**Definition of an Add-in for Rational Rose**) explains how to define an add-in for the **CASE** tool Rational Rose.

This thesis ends with a list of the references used during the research, an index of important terms, and an index of the authors cited in the text.

In Figure 1.1, we graphically highlight the main chapters of this thesis and relate them to the three traditional levels of data modeling.

To help to read this work, the first paragraph of each chapter provides a synopsis of that chapter's content. Moreover, beginning in Chapter 5 and until Chapter 10, the chapters include a figure that is used as a road map for presenting our approach.

1.8 Typographic Conventions

In order to improve the legibility of the text, different typographic conventions have been applied across this thesis.

The typefaces used within the text are:

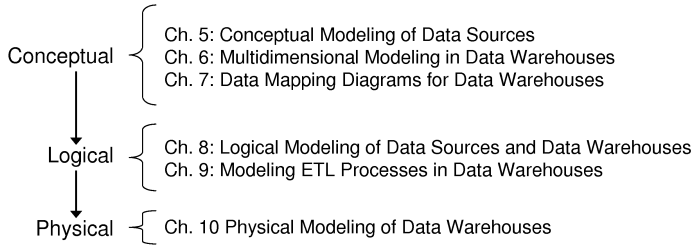


Figure 1.1: Main chapters of the thesis

- Some care has been taken to provide a complete list of the acronyms that appear throughout this work. If an acronym is included in the list of acronyms, then it is emphasized in **bold type**. For example: **DW**, **MD**, **UML**, etc.

- When an acronyms appears the first time, the full name is expanded in *italic type* and **bold type** and the acronym is shown between brackets and in **bold type**. For example: ***Extraction, Transformation, Loading*** (**ETL**), ***OnLine Analytical Processing*** (**OLAP**), etc.

- The names of the stereotypes are highlighted in sans serif and delimited between guillemets[2], and begin with uppercase[3]. For example: «StarPackage», «Fact», etc.

- The names of the tagged values are emphasized in sans serif and enclosed by a pair of braces ({ }). For example: {isTime}, {derivationRule}, {OS}, etc.

- The names of the diagrams we propose are highlighted in SMALL CAPS. For example: SOURCE CONCEPTUAL SCHEMA, DATA WAREHOUSE LOGICAL SCHEMA, SPS, DWCS, etc.

- The text that appears in a figure is shown in sans serif. For example: Level 1, Package, (from Core), etc.

[2]Guillemets are the quotation mark symbols used in French and certain other languages. A guillemet looks like a double angle-bracket (« »), but it is a single character in most extended fonts (≪ ≫). **UML** [97] states that "*Double angle-brackets may be used as a substitute by the typographically challenged*".

[3]To differentiate between the existing **UML** stereotypes and the ones that we propose, the names of the stereotypes we propose begin with an uppercase letter, whereas the existing **UML** stereotypes begin with a lowercase letter (as *Naming Conventions and Typography* section defines in [97]).

- Particular entities of the diagrams (classes, attributes, etc.) are referred to using sans serif. For example: Customer, FullName, Invoices, etc.

- The citations are shown in *italic type* and between the quotation marks (“ ”). For example: “*A data warehouse is a subject-oriented, integrated, time-variant, nonvolatile collection of data in support of management's decisions*”.

1.9 Cross-References

We have had some care to write each chapter as a whole, so the chapters can be read independently. Nevertheless, in some situations we have to refer some content that has previously appeared or that will appear in a following chapter. In those cases, we have included notes in the margin to provide extensive cross-references point to other related sections.

Analysis: see UP chapter 8, pp. 173.

There are two types of cross-references:

OCL: see UML chapter 6, pp. 6-1.

- The notes without a bounding box reference to a related section from **UML** [97] or **UP** [59]. For example, beside these lines there are two notes, one references **UP** and another references **UML**.

For more information about the *logical modeling* of data sources, consult chapter 8, pp. 121.

- The notes with a bounding box reference to a related section from the thesis.

1.10 Diagrams

Most of the diagrams in this thesis are **UML** diagrams. All the **UML** diagrams have been designed using Rational Rose 2003 with an add-in we have implemented for the **DW** design.

Chapter 2

Introduction to Data Warehouses

In this chapter, we make a brief introduction to **DW** and data modeling.

Contents

2.1 What is a Data Warehouse?

In the early nineties, Bill Inmon [57] coined the term **DW**: "*A data warehouse is a subject-oriented, integrated, time-variant, nonvolatile collection of data in support of management's decisions*". This definition contains four key elements that deserve a detailed explanation:

- Subject orientation means that the development of the **DW** will be done in order to satisfy the analytical requirements of managers that will query the **DW**. The topics of analysis differ and depend on the kind of business activities; for example, it can be product sales focusing on client interests in some sales company, the client behavior in utilization of different banking services, the insurance history of the clients, the railroad system utilization or changes in structure, etc.

- Integration relates to the problem that data from different operational and external systems have to be joined. In this process, some problems have to be resolved: differences in data format, data codification, synonyms (fields with different names but the same data), homonyms (fields with the same name but different meaning), multiplicity of data occurrences, nulls presence, default values selection, etc.

- Non-volatility implies data durability: data can neither be modified nor removed.

- Time-variation indicates the possibility to count on different values of the same object according to its changes in time. For example, in a banking **DW**, the average balances of client's account during different months for the period of several years.

On the other hand, Ralph Kimball [63] concisely defines a **DW** as "*a copy of transaction data specifically structured for query and analysis*". He provides a more precise definition by means of requirements:

1. The data warehouse provides *access* to corporate or organizational data.

2. The data in a data warehouse is *consistent.*

3. The data in a data warehouse can be separated and combined by means of every possible measure in a business (the classic **slice and dice** requirement).

4. The data warehouse is not just data, but also a set of tools to *query, analyze, and present* information.

5. The data warehouse is the place where we *publish used data*.

6. The quality of the data in the data warehouse is a *driver of business reengineering*.

Alternatively, the ***Decision Support System*** **(DSS)** scientists express that a **DW** is a database that is optimized for decision support. For example, in [100] the authors state that "*The concept of the data warehouse is often misunderstood. To minimize confusion, we have chosen to define a data warehouse as a read-only analytical database* that is used as the foundation of a decision support system".

Finally, other authors focus their interest on the final users of the **DW**. For example, in [62], a **DW** is defined as a "*collection of technologies aimed at enabling the knowledge worker (executive, manager, and analyst) to make better and faster decisions*".

2.2 Levels of Data Modeling

"*Data modeling is a technique that records the inventory, shape, size, contents, and rules of data elements used in the scope of a business process*" [7]. The result of data modeling is a kind of map (the model) that describes the data used in a process.

Traditionally, there are three levels of data modeling in databases and **DW**: conceptual, logical, and physical. These three levels provide a framework for developing a database structure or schema from the top down. This section will briefly explain the difference among the three and the order with which each one is created.

In Figure 2.1, we graphically represent the relationship among the three levels of data modeling. Whereas the conceptual level is closer to the user domain and the physical level is closer to the computer, the logical level serves as a bridge between the conceptual and the logical levels.

2.2.1 Conceptual Data Model

In the conceptual data model, we normally represent the important entities and the relationships among them. The goal of conceptual data modeling is to describe data in a way which is not governed by implementation-level issues and details.

The conceptual data model is closer to the problem space (the real world) than to the solution space (the implementation).

2.2.2 Logical Data Model

The logical data model usually includes:

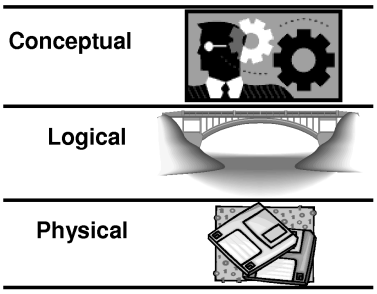


Figure 2.1: Conceptual, logical, and physical levels

- All entities and relationships among them.
- All attributes and the corresponding datatypes for each entity.
- The primary key for each entity specified.
- Foreign keys.

The goal of the logical data model is to describe the data in as much detail as possible, without regard to how they will be physically implemented in the database.

2.2.3 Physical Data Model

In the physical data model, we normally include the whole specification of all tables and columns following the rules of the implementation platform.

The physical data model determines the actual design of a database. This model is the basis of the code written to create tables, views, and integrity constraints.

2.2.4 Data Modeling and UML

Nowadays, the dominant trend in data modeling is the **OO** paradigm, because **OO** modeling supports complex and advanced data structures. The **OO** paradigm is semantically richer than others and it offers numerous advantages, but "*the most important advantage of conceptualizing by means of an OO model is that the result is closer to the user conception*" [1].

In Figure 2.2, we show a flow diagram adapted from [90]. In this diagram, each level of modeling (conceptual, logical, and physical) is shown within the major activities performed and the key **UML** elements that support that activity.

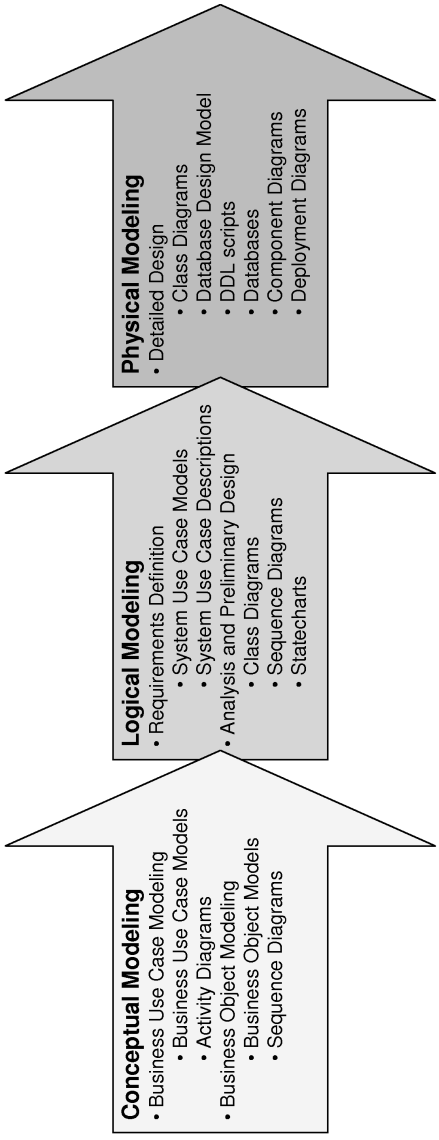


Figure 2.2: Stages of modeling and related UML constructs

Chapter 3

Related Work

In this section, we introduce the most important related work about **DW**, **MD** modeling, and **ETL** processes published during the last few years.

Contents

3.1 Introduction

Although the complete definition of a **DW** engineering process is not the main goal of this work, we have outlined a proposal based on the **UP**. Therefore, in Section 3.2, we briefly present some of the most important **DW** design methods proposed until now and point out the main shortcomings.

The main contribution of our work is the proposal of a **MD** modeling approach based on the **UML**. Hence, in Section 3.3, we summarize the most relevant conceptual **MD** modeling approaches proposed so far by the research community and we provide a comparison framework between them.

In **DW** environments, **ETL** processes are responsible for the extraction of data from heterogeneous operational data sources, their transformation (conversion, cleaning, normalization, etc.) and their loading into **DW**. We have proposed the modeling of **ETL** processes as part of our integrated and global approach for **DW** design. In Section 3.4, we review some works concerning the conceptual and logical modeling of **ETL** processes.

From our **ETL** modeling approach, we have addressed the problem of modeling data mappings between source and target data sources at the attribute level. In Section 3.5, we present some works related to data mappings.

Although several approaches have been proposed to model different aspects of a **DW**, few efforts have been dedicated to the modeling of the physical design (i.e. the physical structures that will host data together with their corresponding implementations). We have proposed an adaptation of the component and deployment diagrams of **UML** for the modeling of the physical design of a **DW**. In Section 3.6, we briefly comment some other works that have dealt with the physical design and deployment of a **DW**.

In our work, we have needed to extend **UML** in different ways in order to accomplish our research goals. In Section 3.7, we present some related work with regard to extending **UML**, applying **UML** packages, and using attributes as first-class modeling elements.

3.2 Data Warehouse Engineering Process

During the last few years, different data models [49, 20, 133, 55, 132] (see Section 3.3 for a review of the most important proposals), both conceptual and logical, have been proposed for the **DW** design. These approaches are based on their own visual modeling languages or make use of a well-known graphical notation, such as the **ER** model or the **UML**. However, none of these approaches has been

widely accepted as a standard **DW** model, because they present some important lacks.

On the other hand, different **DW** methods [63, 50, 20, 48, 23, 87] have also been proposed. However, all of them present some of these problems: they do not address the whole **DW** process, they do not include a visual modeling language, they do not propose a clear set of steps or phases, or they are based on a specific implementation (e.g., the star schema in relational databases).

In [63], different case studies of ***Data Mart* (DM)** are presented. The **MD** modeling is based in the use of the *star schema* and its different variations (snowflake and fact constellation). Moreover, the BUS matrix architecture, which integrates the design of several **DM**, is proposed. Although we consider this work as a fundamental reference in the **MD** field (R. Kimball provides a very sound discussion of star schema design), we miss a formal method for the design of **DW**. Furthermore, the conceptual and logical models coincide in this proposal, and the concepts about the BUS matrix architecture are a compilation of the personal experiences of the authors and the problems they have faced during the built of enterprise **DW** from **DM**. In [65], the **DW** lifecycle with the most relevant phases is presented: different tools and techniques are suggested, but a method (and a model) for all the process is not proposed.

In [50], the authors propose the *Dimensional-Fact Model* (DFM), a particular notation for the **DW** conceptual design. Moreover, they also propose how to derive a **DW** schema from the data sources described by **ER** schemas. From our point of view, this proposal is only oriented to the conceptual and logical design of **DW**, because it does not consider important aspects such as the design of **ETL** processes. Furthermore, the authors assume a relational implementation of the **DW** and the existence of all the **ER** schemas of the data sources, what is impossible many times. Finally, we think that the use of a particular notation makes difficult the application of this proposal.

In [20], the authors present the *Multidimensional Model*, a logical model for **OLAP** systems, and show how it can be used in the design of **MD** databases. The authors also propose a general design method, aimed at building an **MD** schema starting from an operational database described by an **ER** schema. Although the design steps are described in a logic and coherent way, the **DW** design is only based on the operational data sources, what is insufficient from our point of view, because the final users' requirements are very important in the **DW** design.

In [47], a framework to build a **DW** in three basic steps (planning, design and implementation, and support and enhancement) is presented. The author highlights the importance of using a method: "*Successfully implementing a DW requires a proven framework, or*

blueprint". Nonetheless, a model for the analysis and design of a **DW** is not provided: only the activities to be carried out and the decisions to be taken are shown.

In [34], a method based on the **UML** for the **DW** design is presented. From our point of view, the most significant aspect of this proposal is the incorporation of the **UML** use cases in order to specify the roles of each one of the members of the **DW** development team. Apart from that, this method does not study in depth some relevant aspects such as the conceptual or logical design of the **DW**, or the **ETL** processes; because of this, this approach cannot be considered a detailed method.

In [28], different **DW** architectures and the activities needed for the construction of a **DW** are discussed. Although the book has a chapter dedicated to "DW design methodology", only the steps needed for the construction of the "*preferred architecture of a DW*" are presented, and the modeling is based on the star schema.

In [87], the building of the star schema (and its different variations) from the conceptual schemas of the operational data sources is proposed another time. And again, it is supposed that the data sources are defined by means of **ER** schemas. This approach differs in that it does not propose a particular graphical notation for the conceptual design of the **DW**, but it uses the **ER** graphical notation.

In [6], the authors mainly focus on the definition of MD hierarchies, but they also sketch a **DW** design method based on the three usual modeling levels (conceptual, logical, and physical). The conceptual design is based on the **UML**, but the authors propose their *Unified Multidimensional Model* for the logical design.

Most recently, in [23] another method for the **DW** design is proposed. This method is based on a **MD** model called IDEA and it proposes a set of steps that address the conceptual, logical, and physical design of a **DW**. One of the most important advantages, with respect to the previous proposals, is that the operational data sources together with the final users' requirements are considered in the design. Nevertheless, this method only considers the data modeling and does not address other relevant aspects, such as the **ETL** processes.

In [21], different **DW** development methods are analyzed and a new method is proposed. This method stands out because it integrates the management of metadata. However, it lacks a model that can be used to reflect and document the **DW** design.

In [118], a comparison of **DW** methodologies is presented. The comparison is based on using a common set of attributes to determine which methodology to use in a particular data warehousing project. Nevertheless, the authors only focus on commercial methodologies. Moreover, the authors state that "*...the field of data warehousing*

is not very mature" and "*None of the methodologies reviewed in this article has achieved the status of a widely recognized standard as yet*".

Finally, the only work we know that uses the **UML** for the design of **DW** is [51], which explains the modeling of the star and snowflake schemas using the **UML**. However, this work only addresses a single step of the **DW** design process and does not propose a **UML** extension for **DW** design: it only shows how to achieve the star schema using the **UML**.

Therefore, and based on the previous considerations, we believe that currently there is not a general and standard formal method that comprises the main steps of the **DW** design.

3.3 Multidimensional Modeling

Lately, several **MD** data models have been proposed. Some of them fall into the logical level (such as the well-known star-schema by Ralph Kimball [63]). Others may be considered as formal models as they provide a formalism to consider main **MD** properties. A review of the most relevant logical and formal models can be found in [16] and [2].

In this section, we will only make brief reference to the most relevant models that we consider "pure" conceptual **MD** models. These models provide a high level of abstraction for the main **MD** modeling properties presented in Chapter 6 and are totally independent from implementation issues. One outstanding feature provided by these models is that they provide a set of graphical notations (such as the classical and well-known ***Extended Entity-Relationship*** **(EER)** model) that facilitates their use and reading. These are as follows: *The Dimensional-Fact (DF) Model* by Golfarelli *et al.* [49], *The Multidimensional/ER (M/ER) Model* by Sapia *et al.* [113, 112], *The starER Model* by Tryfona *et al.* [133], the Model proposed by Hüsemann *et al.* [55], and *Yet Another Multidimensional Model (YAM^2)* by Abelló *et al.* [3].

In [64], Kimball presents the dimensional modeling, a logical design technique used for **DW**. Kimball compares this new technique and **ER** and points out the many differences between the two techniques. Kimball considers that "*Dimensional modeling is the only viable technique for databases that are designed to support end-user queries in a data warehouse*".

In Table 3.1, we provide the coverage degree of each above mentioned conceptual model regarding the main **MD** properties described in the previous section. To start with, to the best of our knowledge, only YAM^2 provides a grouping mechanism to avoid flat diagrams and simplify the conceptual design when a system becomes complex

due to a high number of dimensions and facts sharing dimensions and their corresponding hierarchies. In particular, this model structures the **MD** modeling into different levels of complexity considering facts and dimensions at the first level, then classification hierarchies, and finally, the whole model. However, from our point of view, even though these different levels try to make the **MD** modeling easier, YAM^2 is a complex model not only for final users, but also for **DW** designers; mainly due to the high number of relations and classes that are needed in the design.

Regarding facts, only YAM^2 explicitly manages the term of multistar, which means that we are able to represent more than one fact in the same **MD** model (i.e. a star schema with more than one fact). Only the starER model and YAM^2 consider *many-to-many* relationships between facts and particular dimensions by indicating the exact cardinality (multiplicity) between them. However, none of these models explicitly represents the term *degenerate facts*. We understand by degenerate facts the measures recorded in a "intersection table" of *many-to-many* relationships [48]. Only YAM^2 considers derived measures together with their derivation rules as part of the conceptual schema. The DF and the M/ER models represent derived measures with the provided query patterns, but not as part of the conceptual schema itself. The DF, the starER and YAM^2 models consider the additivity of measures by explicitly representing the set of aggregation operators that can be applied on non-additive measures.

With reference to dimensions, only YAM^2 is able to have only one definition of a dimension and share it by different facts in *multistar* schemas, thereby avoiding defining the same dimension more than once and allowing the use of *conformed dimensions*. Moreover, only YAM^2 is able to define more than one role for a dimension regarding the same fact by connecting them through different associations. None of them allows us to share only few classification hierarchy levels from dimensions, instead they force us to share the whole classification hierarchy path including all levels. All of the models consider multiple and alternative path classification hierarchies by means of Directed Acyclic Graphs (DAG) defined on certain dimension attributes. However, only the starER and YAM ^{2} models consider non-strict classification hierarchies by specifying the exact cardinality between classification hierarchy levels, instead only the starER model considers adequate to represent complete classification hierarchies. As both the M/ER and the starER models are extensions of the **ER** model, they easily consider the categorization of dimensions by means of *Is-a* relationships. The YAM^2 model represents the categorization of dimensions by means of generalization relationships of the **OO** paradigm.

With reference to the dynamic level of **MD** modeling, the starER

Table 3.1: Comparison of conceptual multidimensional models

Multidimensional modeling properties	**Model**				
	DF	**M/ER**	**starER**	**Hüsemann**	**YAM^2**
Structural level					
Grouping mechanisms to avoid flat diagrams	No	No	No	No	Yes
Facts					
Multi-stars	No	No	No	No	Yes
many-to-many relations with particular dimensions	No	No	Yes	No	Yes
Degenerate facts	No	No	No	No	No
Atomic measures	Yes	Yes	Yes	Yes	Yes
Derived measures	No	No	No	No	Yes
Additivity	Yes	No	Yes	Yes	Yes
Dimensions					
Sharing dimensions (Conformed dimensions)	No	No	No	No	Yes
Different roles of a dimension with the same fact	No	No	No	No	Yes
Sharing few hierarchy levels	No	No	No	No	No
Multiple and alternative path classification hierarchies	Yes	Yes	Yes	Yes	Yes
Non-strict classification hierarchies	No	No	Yes	No	Yes
Complete classification hierarchies	No	No	Yes	No	No
Categorization of dimensions	No	Yes	Yes	Yes	Yes
Dynamic level					
Specifying initial user requirements	Yes	Yes	No	No	Yes
OLAP operations	No	Yes	No	No	Yes
Modeling the behavior of the system	No	Yes	No	No	No
Graphical notation	Yes	Yes	Yes	Yes	Yes
Automatic generation into a target commercial OLAP tool	No	Yes	No	No	No

model is the only one that does not provide an explicit mechanism to represent initial user requirements. On the other hand, only the M/ER model and YAM^2 provide a set of basic **OLAP** operations to be applied from these initial user requirements. Instead, only the M/ER model considers the behavior of the system by modeling the evolution of initial user requirements with state diagrams.

Finally, we note that all the models provide a graphical notation that facilitates the conceptual modeling task to the designer. On the other hand, only the M/ER model provides a framework for an automatic generation of the database schema into a target commercial **OLAP** tool (particularly into Informix Metacube and Cognos Powerplay).

From Table 3.1, one may conclude that none of the current conceptual modeling approaches considers all **MD** properties at both the structural and dynamic levels. From our point of view, the YAM^2 model is the richest one as it considers most of the major **MD** properties, mainly because it is based on the **OO** paradigm; although as previously-stated, we consider this model too complex to use and understand. Therefore, we claim that a standard conceptual model is needed to consider all **MD** modeling properties at both the structural and dynamic levels. We argue that an **OO** approach with the **UML** is the right way of linking structural and dynamic level properties in an elegant way at the conceptual level.

In [132], the authors propose an approach that provides a theoretical foundation for the use of **OO** databases and object-relational databases in **DW**. This approach introduces a set of minimal constraints and extensions to the **UML** for representing **MD** modeling properties for **DW**. However, the **UML** extension is not formally defined as a **UML** profile.

3.4 ETL

Little effort has been dedicated to propose a conceptual model that allows the **DW** designer to formally define **ETL** processes.

To the best of our knowledge, the best advance in this research line has been accomplished by the Knowledge and Database Systems Laboratory from the National Technical University of Athens (NTUA) [92]. In particular, they have proposed a conceptual model that provides its own graphical notation that allows the designer to formally define most of the usual technical problems regarding **ETL** processes [134]. In [135], the conceptual modeling of **ETL** processes is complemented with the logical design of **ETL** processes as graphs. Furthermore, this approach is accompanied by an **ETL** tool called ARKTOS as an easy framework for the design and maintenance of

these **ETL** processes [136].

Finally, Vassiliadis *et al.* do not employ standard **UML** notation because they need to treat attributes as "first class citizens" of their model, what we believe complicates the resulting **ETL** models: a **DW** usually contains hundreds of attributes, and therefore, an **ETL** model can become exceedingly complex if every attribute is individually represented as a model element.

3.5 Data Mapping

There is a relatively small body of research efforts around the issue of conceptual modeling of the **DW** back-stage.

In [14, 15], the *model management*, a framework for supporting meta-data related applications where models and mappings are manipulated is proposed. In [15], two scenarios related to loading **DW** are presented as case studies: on the one hand, the mapping between the data sources and the **DW**, on the other hand, the mapping between the **DW** and a data mart. In this approach, a mapping is a model that relates the objects (attributes) of two other models; each object in a mapping is called a *mapping object* and has three properties: *domain* and *range*, which point to objects in the source and the target respectively, and *expr*, which is an expression that defines the semantics of that mapping object. This is an isolated approach in which the authors propose their own graphical notation for representing data mappings. Therefore, from our point of view, there is a lack of integration with the design of other parts of a **DW**.

In [134] the authors attempt to provide a first model towards the conceptual modeling of the **DW** back-stage. The notion of provider mapping among attributes is introduced. In order to avoid the problems caused by the specific nature of **ER** and **UML**, the authors adopt a generic approach. The static conceptual model of [134] is complemented in [135] with the logical design of **ETL** processes as data-centric workflows. **ETL** processes are modeled as graphs composed of activities that include attributes as first-class citizens. Moreover, different kinds of relationships capture the data flow between the sources and the targets.

Regarding data mapping, in [35], the authors discuss issues related to the data mapping in the integration of data. A set of mapping operators is introduced and a classification of possible mapping cases is presented. However, no graphical representation of data mapping scenarios is provided, thereby making difficult using it in real world projects.

In terms of industrial approaches, the model that stems from [65] would be an informal documentation of the overall data mapping and

ETL process design. On the other hand, the ***Common Warehouse Metamodel* (CWM)** [94] is an open industry standard of the **OMG** for integrating data warehousing and business analysis tools, based on the use of shared metadata. This standard is based on three key industry standards: ***Meta Object Facility* (MOF)**, **UML** and ***XML Metadata Interchange* (XMI)**. These three standards provide the **CWM** with the foundation technology to perfectly represent the semantic of data warehousing by means of metadata, thereby allowing us the intereoperability of **DW** applications by sharing a common metadata specification. However, from our point of view, the **CWM** is too general to represent all main **MD** properties at the conceptual level.

3.6 Data Warehouse Deployment

So far, both the research community and companies have devoted few effort to the physical design of **DW** from the early stages of a **DW** project, and incorporate it within a global method that allows us to design all main aspects of **DW**.

In [65], the authors deal with the lifecycle of a **DW** and propose a method for the design, development and deployment of a **DW**. In that book, we can find a chapter devoted to the planning of the deployment of a **DW** and the authors recommend us documenting all different deployment strategies. However, the authors do not provide a standard technique for the formal modeling of the deployment of a **DW**.

In [100], the authors deal with the design of a **DW** from the conceptual modeling up to its implementation. They propose the use of non-standard diagrams to represent the physical architecture of a **DW**: on one hand, to represent data integration processes and, on the other hand, to represent the relationship between the *enterprise data warehouse* and the different *data marts* that are populated from it. Nevertheless, these diagrams represent the architecture of the **DW** from a high level, without providing different levels of detail of the ulterior implementation of the **DW**.

In [48], several aspects of a **DW** implementation are discussed. Also in that book, other aspects of a **DW** implementation such as the parallelism, the partitioning of data in a ***Redundant Array of Inexpensive Disk* (RAID)** system or the use of a distributed database are addressed, but the authors do not provide a formal or standard technique to model all these aspects.

Finally, in [108], we find that one of the current open problems regarding **DW** is the lack of a formal documentation that covers all design phases and provide multiple levels of abstraction (low level for

designers and people devoted to the corresponding implementation, and high level for final users). The author argues that this documentation is absolutely basic for the maintenance and the ulterior extension of the **DW**. In this work, three different detail levels for **DW** are proposed: *data warehouse level*, *data mart level* and *fact level*. At the first level, the use of the deployment diagrams of **UML** are proposed to document a **DW** architecture from a high level o detail. However, these diagrams are not integrated at all with the rest of techniques, models and/or methods used in the design of other aspects of the **DW**.

Therefore, we argue that there is a still a need for providing a standard technique that allows us to model the physical design of a **DW** from early stages of a **DW** project. Another important issue for us is that this proposal is totally integrated in an overall approach that allows us to cover other aspects of the **DW** design such the conceptual or logical design of the **DW** or the modeling of **ETL** processes.

3.7 Extending UML

In this section, we present some related work connected with the way we use **UML** in our approach: in Section 3.7.1, we present some works that propose some extensions of **UML** by defining **UML** profiles; in Section 3.7.2, we briefly comment some works that have dealt with the use of layering modeling diagrams; finally, in Section 3.7.3, we provide an overview of modeling approaches that treat attributes as first-class modeling elements.

3.7.1 Defining Profiles

With relation to the subject of this work (**DW**, **MD** modeling, data modeling), some proposals to extend the **UML** for database design have been presented during the last few years, due to the fact that the **UML** does not explicitly include a data model. In [8], "*...a profile that extends the existing class diagram definition to support persistence modeling*" is presented. This profile is intended to make objects persistent in different storages: files, relational databases, object-relational databases, etc. In [106], the *Data Modeling Profile* for the **UML** is described, "*...including descriptions and examples for each concept including database, schema, table, key, index, relationship, column, constraint and trigger*". In [90], the process of UML-based database modeling and design is explained: it presents the *UML Profile for Database Design* created by Rational Software Corporation. Finally, in [83] an *Object-Relational Database Design Methodology*

is presented. The methodology defines new **UML** stereotypes for Object-Relational Database Design and proposes some guidelines to translate a **UML** schema into an object-relational schema. However, these proposals do not reflect the peculiarities of **MD** modeling.

In [9], the author complains that data modeling is not yet covered by the **UML** and argues that the **UML** needs a data model. Therefore, he proposes his own data modeling profile. Finally, the author pleads for turning his proposal into an official **UML** profile.

Regarding other domains, some remarkable profiles have been proposed. In [27], a profile for designing web applications with **UML** is presented. In [96], a *UML Profile for CORBA* is presented; this profile is designed to provide a standard means for expressing the semantics of CORBA IDL using **UML** notation and thus to support expressing these semantics with **UML** tools.

Finally, some authors have criticized the **UML** extension mechanisms. For example, in [115], the attention is focused on the **UML** Meta Model and its shortcomings. The authors highlight that many adaptations often exceed the **UML** extension mechanisms and result in yet another **UML** variant. Therefore, the authors propose a robust meta model extension capability to support domain specific extensions in a standard way.

3.7.2 Using Packages

The benefits of layering modeling diagrams is widely recognized. Different modeling techniques, such as Data Flow Diagrams, Functional Modeling - IDEF0, and **ER** make use of some kind of layering mechanism. The benefits of layering modeling diagrams are twofold: to improve user understanding and to simplify documentation and maintenance.

If we focus on **ER**, different approaches can be commented. For example, in [41], the *Clustered Entity Model* is presented, one of the early attempts at layering **ER** diagrams. In this approach, an **ER** diagram at a lower level appears as an entity on the next level. In [125], a model and a technique for clustering entities in an **ER** diagram is described. This modeling technique refines **ER** diagrams into higher-level objects that leads to a description of the conceptual database on a single page. All the previous approaches are based on an already existing detailed **ER** diagram. Based on this, the diagrams are built bottom-up. In [61], the previous approaches to entity model clustering are extended to allow top-down design (in this proposal, bottom-up design can be used too). Then, in [46], the *Leveled Entity Relationship Model*, another layering formalism for **ER** diagrams, is introduced. According to the authors, this new model resolves some of the problems that the previously commented approaches present.

Finally, in [85, 86], the *Levelled Data Model*, a method for decomposing a large data model into a hierarchy of models, is defined. In these works, a set of principles which prescribe the characteristics of a good decomposition are proposed and a genetic algorithm is described which automatically finds an optimal decomposition.

The **UML** mechanism for managing complex diagram is the package: it breaks a model into more manageable pieces. **UML** packages are used to group elements (classes, components, interfaces, etc.) that show strong cohesion with each other and loose coupling with elements in other packages. Every element in the model is owned by exactly one package. In addition, a package provides a namespace such that two different elements in two different packages can have the same name.

UML [97] does not formally define how to apply packages. Therefore, without some heuristics to group classes together, the use of packages become arbitrary. Many text books and authors have provided general guidelines to develop them. In [116, 117], the authors discuss the main drawbacks that the **UML** package presents, criticize its current form, and present a compact and precise definition of its visibility rules.

In [43], the author highlights the lack of semantics in the use of **UML** packages and declares: "*I use the term package diagram for a diagram that shows packages of classes and the dependencies among them*".

Regarding the use of **UML** for **MD** modeling, in [3], a grouping mechanism to avoid flat diagrams and simplify the conceptual design is provided. Thanks to the use of **UML** packages, the design is simplified when a system becomes complex due to a high number of dimensions and facts sharing dimensions and their corresponding hierarchies. In particular, this approach structures the **MD** modeling into different levels of complexity considering facts and dimensions at the first level, then classification hierarchies, and finally, the whole model.

3.7.3 Attributes as First-Class Modeling Elements

The issue of treating attributes as first-class modeling elements has generated several debates from the beginning of the conceptual modeling field [40]. More recently, some object-oriented modeling approaches such as OSM (Object Oriented System Model) [38] or ORM (Object Role Modeling) [54] reject the use of attributes (*attribute-free models*) mainly because of their inherent instability. In these approaches, attributes are represented with entities (objects) and relationships. However, an ORM diagram can be transformed into a **UML** diagram and vice versa [53].

Chapter 4

A Data Warehouse Engineering Process

Developing a **DW** is a complex, time consuming and prone to fail task. Different **DW** models and methods have been presented during the last few years. However, none of them addresses the whole development process in an integrated manner. In this chapter, we outline a **DW** development method, based on the **UML** and the **UP**, which addresses the design and development of both the **DW** back-stage and front-end. We extend the **UML** in order to accurately represent the different parts of a **DW**. Our proposal provides a seamless method for developing **DW**.

Contents

4.1 Introduction

Building a **DW** is a challenging and complex task because a **DW** concerns many organizational units and can often involve many people. Although various methods and approaches have been presented for designing different parts of **DW**, no general and standard method exists to date for dealing with the whole design of a **DW**.

In the light of this situation, the goal of our work is to develop a **DW** engineering process[1] to make the developing process of **DW** more efficient. Our proposal is an **OO** method, based on the **UML** [18, 111, 97] and the **UP** [59], which allows the user to tackle all **DW** design stages, from the operational data sources to the final implementation and including the definition of the **ETL** processes and the final users' requirements.

The rest of the chapter is structured as follows. In Section 4.2, we summarize our **DW** engineering process. Then, in Section 4.3, we present the diagrams we propose to model a **DW**. In Section 4.4, we describe the different workflows that make up our process, and we include an example in order to make easier the understanding of our proposal. Finally, we present the main conclusions in Section 4.5 and we introduce the following chapters in Section 4.6.

4.2 Data Warehouse Development

The goal of our work is to develop a **DW** engineering process to make the developing process of **DW** more efficient. In order to achieve this goal, we consider the following premises:

- Our method should be based on a standard visual modeling language.
- Our method should provide a clear and seamless method for developing a **DW**.
- Our method should tackle all **DW** design stages in an integrated manner, from the operational data sources to the final implementation and including the definition of the **ETL** processes and the final users' requirements.
- Our method should provide different levels of detail.

Therefore, we have selected the **UML** as the visual modeling language, our method is based on the well-accepted **UP**, we have extended the **UML** in order to accurately represent the different parts

[1] We use the terms "method" and "process" as synonyms: "*A software development process is the set of activities needed to transform a user's requirements into a software system*" [59].

of a **DW**, and we extensively use the **UML** packages with the aim of providing different levels of detail.

4.3 Data Warehouse Diagrams

The architecture of a **DW** is usually depicted as various layers of data in which data from one layer is derived from data of the previous layer [62]. Following this consideration, we consider that the development of a **DW** can be structured into an integrated framework with five stages and three levels that define different diagrams for the **DW** model, as shown in Figure 4.1:

- **Stages**: we distinguish five stages in the definition of a **DW**:
  - Source, which defines the structure of the operational data sources of the **DW**, such as ***OnLine Transaction Processing* (OLTP)** systems, external data sources (syndicated data, census data), etc.
  - Integration, which defines the mapping between the data sources and the **DW**.
  - Data Warehouse, which defines the structure of the **DW**.
  - Customization, which defines the mapping between the **DW** and the clients' structures.
  - Client, which defines special structures that are used by the clients to access the **DW**, such as **DM** or **OLAP** applications.

For more information about *levels of data modeling*, consult section 2.2, pp. 14.

- **Levels**: each stage can be analyzed at three levels or perspectives:
  - Conceptual: it defines the **DW** from a conceptual point of view.
  - Logical: it addresses logical aspects of the **DW** design, such as the definition of the **ETL** processes.
  - Physical: it defines physical aspects of the **DW**, such as the storage of the logical structures in different disks, or the configuration of the database servers that support the **DW**.
- **Diagrams**: each stage or level require different modeling formalisms. Therefore, our approach is composed of 15 diagrams, but the **DW** designer does not need to define all the diagrams in each **DW** project: for example, if there is a straightforward mapping between the Source Conceptual Schema (SCS)

	Source (S)	Integration	Data Warehouse (DW)	Customization	Client (C)
Conceptual	**SCS** Class diagram *Standard UML*	**DM** Class diagram *Data Mapping Profile*	**DWCS** Class diagram *Standard UML* *Multidimensional Profile*	**DM** Class diagram *Data Mapping Profile*	**CCS** Class diagram *Standard UML* *Multidimensional Profile*
Logical	**SLS** Class diagram *Different data modeling profiles*	**ETL Process** Class diagram *ETL Profile*	**DWLS** Class diagram *Different data modeling profiles*	**Exporting Process** Class diagram *ETL Profile*	**CLS** Class diagram *Different data modeling profiles*
Physical	**SPS** Comp. & deploy. diagrams *Database Deployment Profile*	**Transportation Diagram** Deployment diagram *Database Deployment Profile*	**DWPS** Comp. & deploy. diagrams *Database Deployment Profile*	**Transportation Diagram** Deployment diagram *Database Deployment Profile*	**CPS** Comp. & deploy. diagrams *Database Deployment Profile*

LEGEND: CS: Conceptual Schema, LS: Logical Schema, PS: Physical Schema, Comp. & deploy: Component and deployment

Figure 4.1: Data warehouse design framework

and the Data Warehouse Conceptual Schema (DWCS), the designer may not need to define the corresponding Data Mapping (DM). In our approach, we use the **UML** [97] as the modeling language, because it provides enough expressiveness power to address all the diagrams. As the **UML** is a general modeling language, we can use the **UML** extension mechanisms (stereotypes, tag definitions, and constraints) to adapt the **UML** to specific domains. In Figure 4.1, we provide the following information for each diagram:

For more information about *UML extension mechanisms*, consult appendix C, pp. 217.

- Name (**in bold face**): the name we have coined for this diagram.
- **UML** diagram: the **UML** diagram we use to model this **DW** diagram. Currently, we use class, deployment, and component diagrams.
- Profile (*in italic face*): we show the diagrams where we propose a new profile; in the other cases, we use a standard **UML** diagram or a profile from other authors.

The best advantage of our global approach is that we always use the same notation (based on **UML**) for designing the different **DW** schemas and the corresponding transformations in an integrated manner. Moreover, the different diagrams of the same **DW** are not independent but overlapping: they depend on each other in many ways. For example, changes in one diagram may imply changes in another, and a large portion of one diagram may be created on the basis of another diagram. For example, the DM is created by importing elements from the SCS and the DWCS.

We have presented in international conferences the different diagrams and the corresponding profiles we propose as follows:

- *Multidimensional Profile*, for the Data Warehouse Conceptual Schema (DWCS) and the Client Conceptual Schema (CCS), in [78, 79].

- *Data Mapping Profile*, for the Data Mapping (DM) between the Source Conceptual Schema (SCS) and the DWCS, and between the DWCS and the CCS, in [81].
- *ETL Profile*, for the ETL Process between the Source Logical Schema (SLS) and the Data Warehouse Logical Schema (DWLS), and the Exporting Process between the DWLS and the Client Logical Schema (CLS), in [128].
- *Database Deployment Profile*, for the Source Physical Schema (SPS), the Transportation Diagram, the Data Warehouse Physical Schema (DWPS), and the Client Physical Schema (CPS), in [75, 76].

4.4 Data Warehouse Engineering Process

Our method, called *Data Warehouse Engineering Process* (DWEP), is based on the Unified Software Development Process, also known as **UP** [59]. The **UP** is an industry standard ***Software Engineering Process* (SEP)** from the authors of the **UML**. Whereas the **UML** defines a visual modeling language, the **UP** specifies how to develop software using the **UML**.

Use-case driven, architecture centric, iterative and incremental: see UP chapter 1, pp. 4.

The **UP** is a generic **SEP** that has to be instantiated[2] for an organization, project or domain[3]. DWEP is our instantiation of the **UP** for the development of **DW**. Some characteristics of our DWEP inherited from **UP** are: use case (requirement) driven, architecture centric, iterative and incremental. This three key words make the **UP** unique, therefore it is important to provide a brief overview of them:

- Use case (requirement) driven: means that use cases are used for specifying the requirements of a system, but they also drive its design, implementation, and test.
- Architecture centric: the software architecture embodies the most significant static and dynamic aspects of the system and it is described as different views of the system being built.
- Iterative and incremental: the develop of the software products is divided into smaller slices called iterations that results in an increment that refers to growth in the product. Moreover, the diagrams will not stay intact but should be expected to evolve

[2]Instantiate means add in-house standards, define a lifecycle strategy, select what diagrams to use, define activities and workers, etc.

[3]Some popular instantiations of **UP** are Rational Unified Process [56] and Enterprise Unified Process [109].

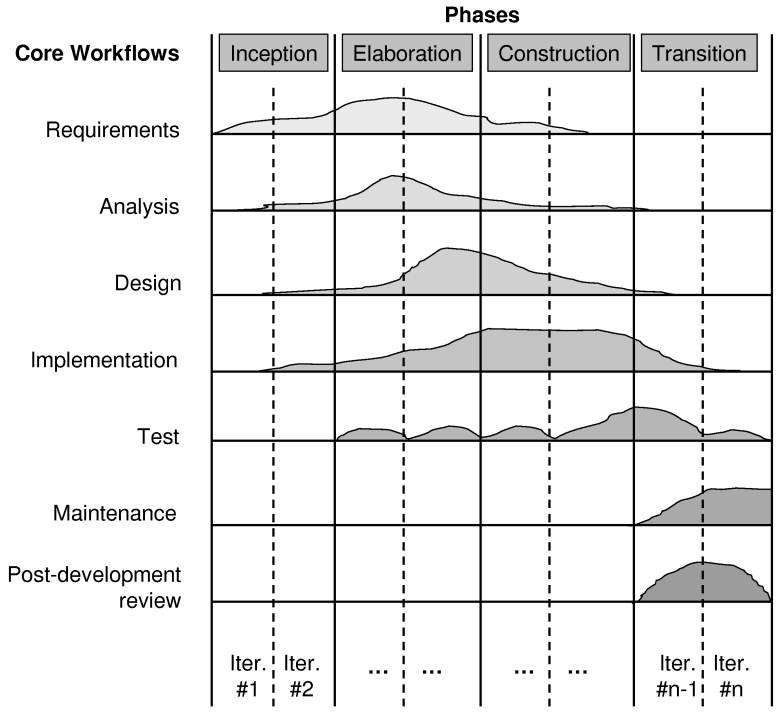


Figure 4.2: DWEP workflows

as time passes, as new requirements are uncovered, and as the schedules changes, causing feature changes.

Phases: see UP chapter 1, pp. 11.

According to the **UP**, the project lifecycle is divided into four phases (Inception, Elaboration, Construction, and Transition) and five core workflows (Requirements, Analysis, Design, Implementation, and Test). We have added two more workflows to the **UP** workflows: *Maintenance* and *Post-development review*. During the developing of a project, the emphasis shifts over the iterations, from requirements and analysis towards design, implementation, testing, and finally, maintenance and post-development review, but different workflows can coexist in the same iteration.

In Figure 4.2 (adapted from [59]), we show that the seven workflows (listed in the left-hand column) take place over the four phases. For each workflow, the curve represents approximately the extent to which the workflow is carried out in each phase. Moreover, each phase is usually subdivided into iterations, and an iteration goes through all the seven workflows.

For each one of the workflows, we use different **UML** diagrams (techniques) to model and document the development process, but a model can be modified in different phases because models evolve over time. In the following sections, we comment the main details of the workflows and highlight the diagrams we use in each workflow.

4.4.1 Requirements

Requirements: see UP chapter 6, pp. 111.

During this workflow, what the final users expect to do with the **DW** is captured: the final users should specify the most interesting measures and aggregations, the analysis dimensions, the queries used to generate periodical reports, the update frequency of the data, etc. As proposed in [19], we model the requirements with use cases. The rationale of use cases is that focusing "*on what the users need to do with the system is much more powerful that other traditional elicitation approaches of asking users what they want the system to do*" [19]. Once the requirements have been defined, the **DW** project is established and the different roles are designated.

Use case modeling is a simple way to [82]:

- Understand the system's existing functions.
- Elicit the desired requirements and functions for the new system is being created.
- Establish who will be interacting with the system and how.

The **UML** provides the use case diagram for visual modeling of uses cases. Nevertheless, there is no **UML** standard for a use case specification. However, we follow the common template defined in [12], which specifies for every use case a name, a unique identifier, the actor involved in the use case, the system state before the use can begin (*preconditions*), the actual steps of the use case (*flow of events*), and the system state when the use case is over (*postconditions*). In [90], a more complex use case description template can be found.

For example, the use case in Figure 4.3 is about sales managers making a query about the quarterly sales of the products in the computer category.

4.4.2 Analysis

Analysis: see UP chapter 8, pp. 173.

The goal of this workflow is to refine and structure the requirements output in the previous workflow. Moreover, the pre-existing operational systems that will feed the **DW** are also documented: the different candidate data sources are identified, the data content is revised, etc.

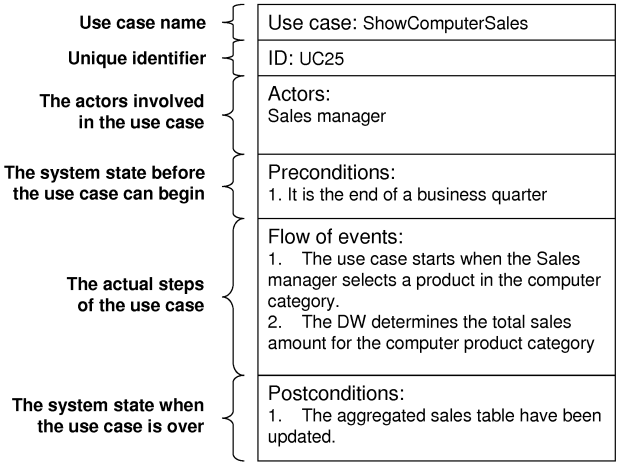


Figure 4.3: UML use case template

We use the Source Conceptual Schema, Source Logical Schema, and the Source Physical Schema (SCS, SLC, and SPS) (Figure 4.1) to model the data sources at different levels of detail. To get quality data in the **DW**, the different data sources must be well identified.

For example, in Figure 4.4 we show the Source Logical Schema (SLS) of a transactional system that manages the sales of a company. This system will feed with data the **DW** that will be defined in the following workflow. For the SLS we make use of the *UML for Profile Database* [90] that defines a series of stereotypes like «Database», «Schema», «Tablespace», or «Table». In the diagram shown in Figure 4.4, each element represents a class with the stereotype «Table» that represents a table in a relational database[4]; we have hidden the attributes (fields or columns) for the sake of simplicity.

For more information about the *logical modeling* of data sources, consult chapter 8, pp. 121.

4.4.3 Design

At the end of this workflow, the structure of the **DW** is defined. The main output of this workflow is the conceptual model of the **DW**. Moreover, the source to target data map is also developed at a conceptual level.

Desgin: see UP chapter 9, pp. 215.

In this workflow, the main diagrams are the Data Warehouse Conceptual Schema (DWCS), the Client Conceptual Schema

[4] A class symbol with a stereotype icon may be "collapsed" to show just the stereotype icon, with the name of the class either inside the class or below the icon. Other contents of the class may be suppressed.

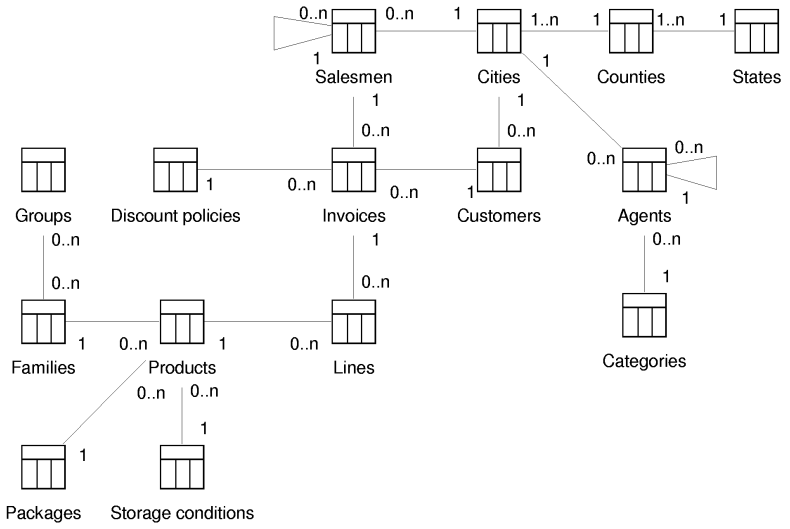


Figure 4.4: Source Logical Schema

(CCS), and the Data Mapping (DM). The DM shows the relationships between the SCS and the DWCS and between the DWCS and the CCS.

For more information about the *multidimensional profile*, consult chapter 6, pp. 53.

For the DWCS and the CCS, we have presented [78, 79] an extension of the **UML** by means of a **UML** profile. This profile is defined by a set of stereotypes and tagged values to elegantly represent main **MD** properties at the conceptual level. We make use of the ***Object Constraint Language* (OCL)** to specify the constraints attached to the defined stereotypes, thereby avoiding an arbitrary use of these stereotypes. The main advantage of our proposal is that it is based on a well-known standard modeling language, thereby designers can avoid learning a new specific notation or language for **MD** systems.

For example, in Figure 4.5 we show level 1 of a Data Warehouse Conceptual Schema, composed of three schemas (Production schema, Sales schema, and Salesmen schema). The dashed arrows that connect the different schemas indicate that the schemas share some dimensions that have been firstly defined in the Sales schema. In Figure 4.6 we show level 2 of the Sales schema from level 1, composed of one fact (Sales fact) and four dimensions (Stores dimension, Times dimension, Products dimension, and Customers dimension). Finally, in Figure 4.7, the definition of the Customers dimension with the different hierarchy levels is showed.

For the DM, we have presented [81] the *Data Mapping Profile*

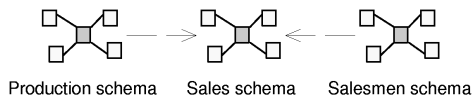


Figure 4.5: Data Warehouse Conceptual Schema (level 1)

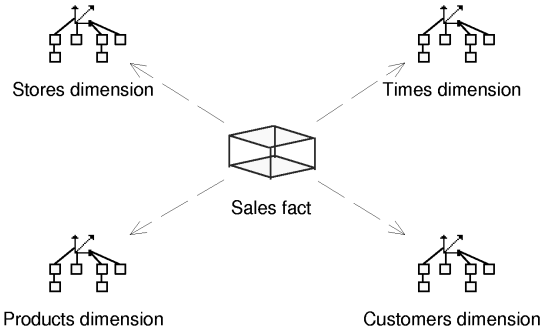


Figure 4.6: Data Warehouse Conceptual Schema (level 2)

that introduces the data mapping diagram. In this new diagram, we treat attributes as first-class modeling elements of the model. In this way, attributes can participate in associations that determine the inter-attribute mapping, along with any necessary transformation and constraints.

For more information about the *data mapping*, consult chapter 7, pp. 97.

4.4.4 Implementation

During this workflow, the **DW** is built: the physical **DW** structures are built, the **DW** is populated with data, the **DW** is tuned for an optimized running, etc. Different implementation diagrams can be created to help this workflow.

Implementation: see UP chapter 10, pp. 267.

The main diagrams in this workflow are the Data Warehouse Logical Schema, the Data Warehouse Physical Schema, the Client Logical Schema, the Client Physical Schema, the ETL Process, the Exportation Process, and the Transportation Diagram. In the ETL Process, the cleansing and quality control activities are modelled.

For more information about *ETL process modeling*, consult chapter 9, pp. 133.

For example, in Figure 4.8 we show part of a Data Warehouse Physical Schema. For this diagram, we have presented the *Database Deployment Profile* [76, 77]. In this example, both the components and the nodes are stereotyped: the components are adorned with the «Database» and «Tablespace» stereotypes, and the nodes with the «Server» and «Disk» stereotypes.

For more information about *physical modeling* of data warehouses, consult chapter 10, pp. 155.

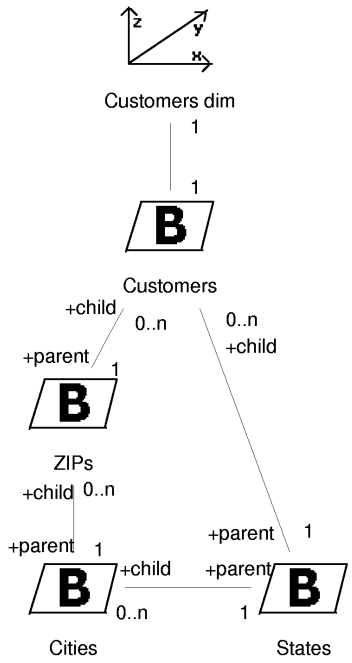


Figure 4.7: Data Warehouse Conceptual Schema (level 3)

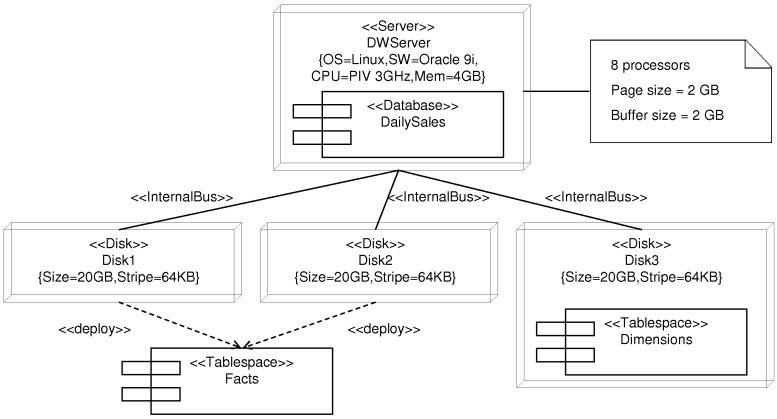


Figure 4.8: Data Warehouse Physical Schema

4.4.5 Test

The goal of this workflow is to verify that the implementation works as desired. More specifically, the purposes of testing are to:

Test: see UP chapter 11, pp. 295.

- Plan the tests required.
- Design and implement the tests by creating test cases.
- Perform the tests and analyze the results of each test.

No new diagrams are created, but previous diagrams (mainly design and implementation diagrams) may be modified according the corrective actions that are taken.

4.4.6 Maintenance

Unlike most systems, the **DW** is never done. The goal of this workflow is to define the refresh and loading processes needed for keeping the **DW** up to date. This workflow starts when the **DW** is built and delivered to the final users, but it does not have an end date (it lasts during the life of the **DW**).

During this workflow, the final users can state new requirements, such as new queries, which triggers the beginning of a new iteration (**UP** is an iterative process) with the Requirements workflow.

4.4.7 Post-development Review

This is not a workflow of the development effort, but a review process for improving future projects. We look back at the development of the **DW**, revise the documentation created, and try to identify both opportunities for improvement and major successes that should be taken into account. If we keep track of the time and effort spent on each phase, this information can be useful in estimating time and staff requirements for future projects.

4.4.8 Top-down or Bottom-up?

Nowadays, there are two basic strategies in the building of a **DW**: the top-down and bottom-up approaches [52, 37, 138]. The top-down approach recommends the construction of a **DW** first and then the construction of **DM** from the parent **DW**. The bottom-up approach uses a series of incremental **DM** that are finally integrated to build the goal of the **DW**. Each approach has its own set of strengths and weaknesses. However, in almost all projects, the **DM** are built rather independently without the construction of an integrated **DW**, which

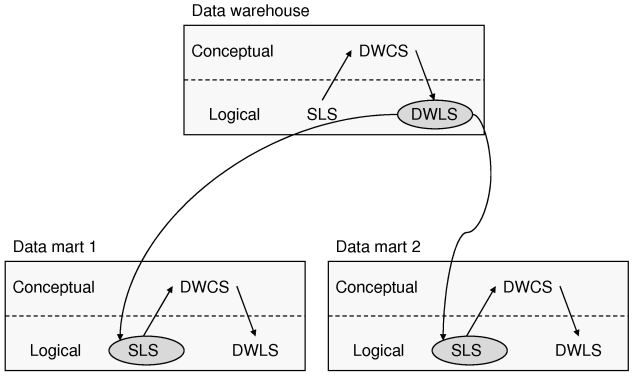


Figure 4.9: Top-down approach

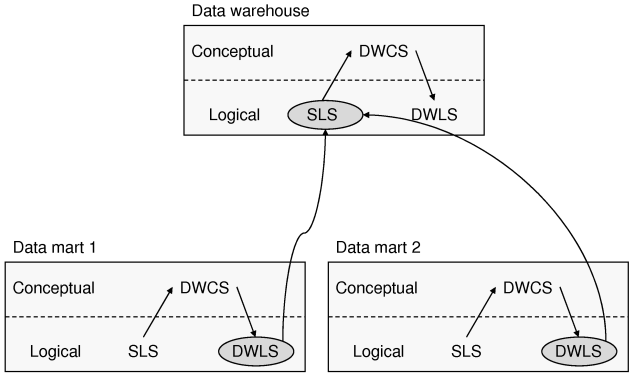


Figure 4.10: Bottom-up approach

is indeed viewed no more as a monolithic repository but rather as a collection of **DM**.

In the early years, the top-down approach was favored and it is considered the most elegant design approach [52]. However, high rates of failure for initial **DW** projects have led the majority of current projects to the bottom-up approach. With the bottom-up approach, the results are seen a lot sooner than implementing the **DW** using a top-down approach. Moreover, risk is minimized and it is more feasible to successfully deploy the **DW** in time.

Our method also allows both approaches. In the top-down approach (see Figure 4.9), the **DW** is built first and the data sources are the transactional systems; then, each **DM** is built independently by using our method, and the **DW** becomes the only data source

for all of them. Nevertheless, in the bottom-up approach (see Figure 4.10), the **DM** are built first from the transactional systems; then, the **DW** is built and the data sources are the **DM**.

4.5 Conclusions

In this chapter, we have presented our *Data Warehouse Engineering Process* (DWEP), a **DW** development process based on the **UML** and the **UP**. **UP** is a generic and stable process that we have instantiated to cover the development of **DW**. Our main contribution is the definition of several diagrams (techniques) and **UML** profiles [78, 79, 128, 81, 75] in order to model **DW** more properly. Whereas the different diagrams provide different views or perspectives of a **DW**, the engineering process specifies how to develop a **DW** and ties up all the diagrams together. The main advantages of our approach are:

- The use of a development process, the **UP**, which is the outcome of more than 20 years of experience.
- The use of the **UML**, a widely accepted visual modeling language, for designing the different **DW** diagrams and the corresponding transformations.
- The use of the **UML** as the modeling language provides much better tool support than using an own modeling language.
- The proposal of a **DW** development process that addresses both the back-end and the front-end of **DW** in an integrated manner.

4.6 Next Chapters

In the following chapters, the different diagrams that have been introduced in this chapter will be presented. At the beginning of every chapter, the schema shown in Figure 4.11 will be used as a road map and will highlight the diagrams from our **DW** design framework that are presented. The diagrams of our approach are presented as follows:

- Chapter 5 (**Conceptual Modeling of Data Sources**): Source Conceptual Schema (SCS).
- Chapter 6 (**Multidimensional Modeling in Data Warehouses**): Data Warehouse Conceptual Schema (DWCS) and Client Conceptual Schema (CCS).
- Chapter 7 (**Data Mapping Diagrams for Data Warehouses**): Data Mapping (DM).

	Source	Integration	Data Warehouse	Customization	Client
Conceptual	SCS	DM	DWCS	DM	CCS
Logical	SLS	ETL Process	DWLS	Exporting Process	CLS
Physical	SPS	Transportation Diagram	DWPS	Transportation Diagram	CPS

Figure 4.11: Schema shown at the beginning of every chapter

- Chapter 8 (**Logical Modeling of Data Sources and Data Warehouses**): Source Logical Schema (SLS), Data Warehouse Logical Schema (DWLS), Client Logical Schema (CLS).
- Chapter 9 (**Modeling ETL Processes in Data Warehouses**): ETL Process and Exporting Process.
- Chapter 10 (**Physical Modeling of Data Warehouses**): Source Physical Schema (SPS), Transportation Diagram, Data Warehouse Physical Schema (DWPS), Transportation Diagram, Client Physical Schema (CPS).

Part I

Conceptual Level

Chapter 5

Conceptual Modeling of Data Sources

	Source	Integration	Data Warehouse	Customization	Client
Conceptual	SCS	DM	DWCS	DM	CCS
Logical	SLS	ETL Process	DWLS	Exporting Process	CLS
Physical	SPS	Transportation Diagram	DWPS	Transportation Diagram	CPS

In this chapter, we address the design of the Source Conceptual Schema. The goal of this diagram is to have a good understanding of the data sources that will feed the **DW**.

Contents

5.1 Introduction

In 1974, the ANSI/X3/SPARC Study Group on ***Database Management System*** **(DBMS)** presented a status report where the term "conceptual schema" was introduced for the first time [99]. In a conceptual schema or conceptual data model, we normally represent the important entities and the relationships among them from the application world in terms independent of any particular data model. A conceptual model permits the designer to focus on what is essential in a problem, independently of the solution. One of the most popular conceptual modeling approaches is the **ER** model.

Basically, the goals of the conceptual modeling are:

- To understand the real-world domain of a problem.
- To reason about the real-world domain.
- To achieve a consensus about the real-world domain.

In Section 5.2, we discuss the similarities and differences between **ER** and **UML** regarding the conceptual data modeling. Then, in Section 5.3 we explain how to achieve the conceptual modeling of the data sources of a **DW**.

5.2 Entity-Relationship and UML

The **ER** model was originally proposed by Peter Chen in 1976 [25]. **ER** has been the conceptual data model par excellence during the last twenty-five years. The **ER** model views the real world as entities and relationships. Since Peter Chen presented the **ER**, different extensions have been introduced (e.g., **EER**) [126]. Nowadays, **ER** is commonly used for database design because it maps well to the relational model and it is simple and easy to understand with a minimum of training. However, **ER** suffers from three major problems [101]:

1. There is no standard and several variations exist.
2. **ER** diagrams tend to be messy and difficult to read.
3. The **ER** approach is weak at handling object-oriented design issues, such as inheritance (subtypes) and composition.

On the other hand, **UML** supports what **ER** notations support and beyond. **UML** was built with **ER** in mind: **UML** is a superset of **ER** notations. Therefore, **UML** allows the designer more flexibility

to achieve "*a valid representation of the entities, their attributes and relationships that will fulfill the needs of the business*" [82].

For data modelling purpose, **UML** uses the class diagram, which resolves the problems that **ER** suffers by defining a standard diagram that handles all of the situations needed by database designers. Although class diagrams may include implementation details, it is possible to use them for analysis by omitting such as details. When used in this way, class diagrams provide an extended **ER** notation [121]. For example, in [88], Robert Muller explains how to use the **UML** to develop and implement databases and he states that "*Object modeling with the UML takes the place of ER modeling in modern database design*".

According to [101], the basic similarities between **ER** and **UML** class diagrams are:

1. Entities (classes) are drawn as boxes.
2. Binary relationships (associations) are drawn as connecting lines.
3. N-ary associations (relationships) are drawn as diamonds.

But, the primary differences lie in the details:

1. Attributes are written in the class box.
2. Multiplicity of an association is shown as simple numerical notation instead of a cryptic icon.
3. Several association (relationship) types have predefined drawing methods.
4. Associations can be directionally named.
5. Comments and labels are explicitly supported.
6. Complex diagrams can be split into packages.

5.3 Source Conceptual Schema

The goal of the Source Conceptual Schema (SCS) is to know what data is available for the **DW**. For the SCS, we apply **UML** in a plain style by simply using classes, attributes and their associations to other entities. As we have expound in the previous section, **UML** is more powerful for conceptual modeling than **ER**.

Chapter 6

Multidimensional Modeling in Data Warehouses

	Source	Integration	Data Warehouse	Customization	Client
Conceptual	SCS	DM	DWCS	DM	CCS
Logical	SLS	ETL Process	DWLS	Exporting Process	CLS
Physical	SPS	Transportation Diagram	DWPS	Transportation Diagram	CPS

MD modeling is the foundation of **DW**, **MD** databases, and **OLAP** applications. In the past few years, there have been some proposals, providing their own formal and graphical notations, for representing the main **MD** properties at the conceptual level. However, unfortunately none of them has been accepted as a standard for conceptual **MD** modeling. In this chapter, we present an extension of the **UML** by means of a **UML** profile. This profile is defined by a set of stereotypes and tagged values to elegantly represent main **MD** properties at the conceptual level. We make use of the **OCL** to specify the constraints attached to the defined stereotypes, thereby avoiding an arbitrary use of these stereotypes. The main advantage of our proposal is that it is based on a well-known standard modeling language, thereby designers can avoid learning a new specific notation or language for **MD** systems. Moreover, our proposal is ***Model Driven Architecture*** **(MDA)** compliant and we use the ***Query View Transformation*** **(QVT)** approach for an automatic generation of the implementation in a target platform.

Contents

6.1 Introduction

DW, **MD** databases, and **OLAP** applications provide companies with many years of historical information for decision making processes. It is widely accepted that these systems are based on **MD** modeling. **MD** modeling structures information into facts and dimensions. A fact contains interesting measures of a business process (sales, deliveries, etc.), whereas a dimension represents the context for analyzing a fact (product, customer, time, etc.). The benefit of using this **MD** modeling is two-fold. On the one hand, the **MD** model is close to the way of thinking of data analyzers and, therefore, helps users understand data. On the other hand, the **MD** model supports performance improvement as its simple structure allows us to predict final users' intentions.

Some approaches have been proposed lately (see related work in Section 3.3) to accomplish the conceptual design of these systems. Unfortunately, none of them has been accepted as a standard for **DW** conceptual modeling. These proposals try to represent main **MD** properties at the conceptual level with special emphasis on **MD** data structures (i.e. facts and dimensions). However, from our point of view, none of them considers all the main properties of **MD** systems at the conceptual level. Furthermore, these approaches provide their own graphical notations, which forces designers to learn a new specific model together with its corresponding **MD** modeling notation.

On the other hand, the **UML** [18, 97] has been widely accepted as the standard object-oriented (OO) modeling language for modeling various aspects of software systems. Therefore, any approach using the **UML** will minimize the effort of developers in learning new notations or methodologies for every subsystem to be modeled. Another outstanding feature of the **UML** is that it is an extensible language in the sense that it provides mechanisms (stereotypes, tagged values, and constraints) to introduce new elements for specific domains if necessary, such as web applications, database applications, business modeling, software development processes, etc. [27, 90]. A collection of enhancements that extend an existing diagram type to support a new purpose is called a *profile*. Furthermore, the **UML** follows the **OO** paradigm, which has been proved to be semantically richer than other paradigms for **MD** modeling [1].

For more information about *UML extension mechanisms*, consult appendix C, pp. 217.

In this chapter, we present a **UML** profile for a coherent and unified conceptual **MD** modeling. This profile expresses for each measure its **MD** context in terms of relevant dimensions and their hierarchies and allows us to easily and elegantly consider main **MD** properties at the conceptual level, such as the many-to-many relationships between facts and dimensions, degenerate dimensions and facts, multiple and alternative path classification hierarchies, and non-strict

OCL: see UML chapter 6, pp. 6-1.

For more information about our *Rational Rose add-in*, consult appendix E, pp. 253.

and complete hierarchies. Our extension uses the **OCL** [97, 139] for expressing well-formedness rules of the new defined elements, thereby avoiding an arbitrary use of this extension. Moreover, we program this extension in a well-known model-driven development tool such as Rational Rose [107] to show its applicability.

In summary, we intend to achieve a proposal with the following properties:

- Accurate: a profile that allows us to represent all major important features of **MD** modeling at the conceptual level.
- Consistent: we allow to import a previously-defined element in our model whenever is possible so we avoid having different definitions and properties for the same concept throughout a model.
- Simple: as simple as possible, but not too simple. We limit our graphical notation to a subset of the **UML** notation that allows us to correctly describe main **MD** properties at the conceptual level.
- Understandable: we attempt to make a proposal understandable for the intended audience (both **DW** designers and final users). When complex and huge **DW** systems are built, it is highly important to have a modeling approach to successfully communicate the different actors that take part in the **DW** design.

The remainder of this chapter is structured as follows: to avoid misunderstanding resulting from the great amount of terminology in **MD** modeling, Section 6.2 introduces the main properties and aspects that a conceptual approach for **MD** modeling should take into consideration. Section 6.3 describes how we make use of the **UML** to consider all major properties of **MD** modeling at the conceptual level. Section 6.4 formally defines the new **UML** extension (*profile*) we propose for **MD** modeling. Section 6.5 describes the transformation of **MD** models based on the **QVT** approach. Finally, Section 6.6 presents the main conclusions.

6.2 Multidimensional Modeling

In **MD** modeling, information is structured into **facts** and **dimensions**[1]. A fact is an item of interest for an enterprise, and is described

[1] We avoid the terms *fact table* or *dimension table* during conceptual modeling, as table suggests logical storage in a ***Relational Database Management System*** **(RDBMS)**.

through a set of attributes called **measures** or **fact attributes** (atomic or **derived**), which are contained in cells or points in the data cube. A data cube is a **MD** representation of data that can be viewed from different perspectives. Therefore, a data cube is based on a set of dimensions that determine the granularity adopted for representing facts. On the other hand, dimensions provide the context in which facts are to be analyzed. Moreover, dimensions are also characterized by attributes, which are usually called **dimension attributes**.

Let us introduce a **DW** modeling example inspired by a case study presented by Giovinazzo in [48], which will be used throughout the rest of this chapter. This example relates to a company that comprises different dealerships that sell automobiles (cars and vans) across several states. The **DW** contains three **DM**[2], such as *automobile sales*, *part sales* and *service works* (they are separated because they are going to be used by different final users). However, these data marts share some common dimensions[3] such as *dealership* or *time*, although they also have their own particular dimensions, such as *salesperson* or *service*:

- Automobile sales (AS): considers the sales of automobiles.
- Part sales (PS): represents the sales of parts of automobiles such as spare wheels or light bulbs.
- Service works (SW): considers the services realized by dealerships such as the change of lubricating oil or brake oil.

Every one of these models has the corresponding fact which contains the specific measures to be analyzed. Furthermore, they consider the following dimensions to analyze measures: *dealership*, *time*, *customer*, *salesperson* and *auto* for the AS; *dealership*, *time*, *service*, *mechanic* and *parts* for the SW; and *dealership*, *time*, *service*, *mechanic* and *parts* for the SW. On the left hand side of Figure 6.1, we can observe a data cube typically used for representing a **MD** model. In this particular case, we have defined a cube for the AS for analyzing measures along the *auto*, *customer* and *time* dimensions.

We note that *many-to-one* relationships exist between the fact and every particular dimension, and thus facts are usually considered to

[2]A **DM** is a type of **DW** primarily designed for addressing a specific function or department's needs: whereas a **DW** combines databases across an entire enterprise, a **DM** is usually smaller and focus on a particular subject or department. According to [48], there are two kinds of **DM**: "*dependent data marts receive their data from the enterprise data warehouse; independent data marts receive data directly from the operational environment*".

[3]Common dimensions used in different data marts are usually called *conformed dimensions*[63].

have *many-to-many* relationships between any of two dimensions. In the previous AS, an *autosales* fact is related to only one *auto* that is sold by one *dealership* and purchased by just one *customer* at one *time*.

Nevertheless, there are some cases in which ***many-to-many*** relationships may exist between the fact and some particular dimensions. For example, the *autosales* fact of AS is considered to have a particular *many-to-many* relationship to the *salesperson* dimension, as more than one *salesperson* may have participated in selling one *auto* (although every auto is still purchased by only one customer in just one *dealership* store and at one *time*).

When having a *many-to-many* relationship with a particular dimension as previously-described, we usually need to consider specific attributes to provide further features for every instance combination in this particular relationship. In doing so, the measures provided are usually called ***degenerated facts*** [63, 48]. In the previous example, we may be interested in recording the specific *commission* that a *salesperson* obtains for every particular auto sales he/she participates.

There are some cases in which we do not consider a dimension explicitly because we believe that most of its properties is already represented throughout other elements (facts and dimensions) in our **MD** model. However, we still believe that we need some attribute or property in the fact to uniquely identify fact instances. When this occurs, we usually call these dimensions as ***degenerated dimensions*** [63, 48]. Therefore, a degenerate dimension is one whose identifier exists only in a fact, but which is not materialized as an actual dimension. This provides other fact features in addition to the measures for analysis. In our example, instead of considering the *autosales*, we could had represented the bill of an *autosales* and consider the *bill* and *bill line numbers* as other bill features (while not having a *bill* dimension materialized).

With reference to measures, the concept of **additivity** or summaribility [16, 49, 63, 132, 133] on measures along dimensions is crucial for **MD** data modeling. A measure is additive along a dimension if the SUM operator can be used to aggregate attribute values along all hierarchies defined on that dimension. The aggregation of some fact attributes (roll-up[4] in **OLAP** terminology), however, might not be semantically meaningful along all dimensions. For example, all measures that record a static level, such as inventory levels, financial account balances or temperatures, are not inherently additive along the *time* dimension. In our particular warehouse example, the measure *quantity* from that records the quantity of a specific *auto* in a

[4]Roll-up is the presentation of data at a higher level of detail, whereas drill-down is the presentation of data at a lower level of detail.

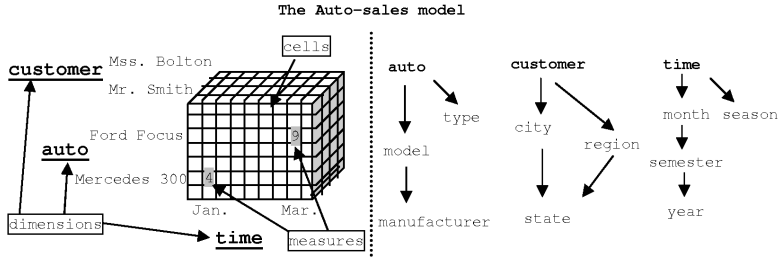


Figure 6.1: A data cube and classification hierarchies defined on dimensions

sale at a given *time* is not additive along the *salesperson* dimension. However, other aggregation operators (e.g. MAX, MIN and AVG) could still be used along the same *salesperson* dimension. Moreover, *quantity* can be additive along the *auto* dimension. Thus, a measure such as *quantity* is called semiadditive since it is additive along one dimension, but non-additive along another dimension.

Regarding dimensions, the **classification hierarchies** defined on certain dimension attributes are crucial because the subsequent data analysis will be addressed by these classification hierarchies. A dimension attribute may also be aggregated (related) to more than one hierarchy, and therefore, **multiple classification hierarchies** and **alternative path hierarchies** are also relevant. For this reason, a common way of representing and considering dimensions with their classification hierarchies is by means of Directed Acyclic Graphs (DAG).

On the right hand side of Figure 6.1, we can observe different classification hierarchies defined on the *auto*, *customer* and *time* dimensions from the AS[5]. On the *auto* dimension, we have considered a multiple classification hierarchy to be able to aggregate data values along two different hierarchy paths: (i) *auto*, *model*, *manufacturer* and (ii) *auto*, *type*. There may exist other attributes that are not used for aggregating purposes and provide features for other dimension attributes (e.g. *auto description*). On the *customer* dimension, we have defined an alternative path classification hierarchy with two different paths that converge into the same hierarchy level: (i) *customer*, *city*, *state* and (ii) *customer*, *region* and *state*. Finally, we have also defined another multiple classification hierarchy with the following paths on the *time* dimension: (i) *time*, *month*, *semester*, *year* and (ii) *time* and *season*.

[5]These classification hierarchies are different from those specifically presented by Giovinnazo in [48] as ours will allow us to consider more peculiarities.

Nevertheless, classification hierarchies are not so simple in most cases. The concepts of **strictness** and **completeness** are important, not only for conceptual purposes, but also for further design steps of **MD** modeling [133]. "Strictness" means that an object of a lower level of a hierarchy belongs to *only* one of a higher level, e.g. a *city* is related to only one *state*. "Completeness" means that all members belong to one higher-class object and that object consists of those members only. For example, suppose we say that the classification hierarchy between the *state* and *city* levels is "complete". In this case, a *state* is formed by *all* the *cities* recorded and all the *cities* that form the *state* are recorded.

OLAP scenarios sometimes become very large as the number of dimensions increases significantly, and therefore, this fact may lead to extremely sparse dimensions and data cubes. In this way, there are attributes that are normally valid for all elements within a dimension while others are only valid for a subset of elements (also known as the **categorization of dimensions** [68, 133]). For example, attributes *number of passengers* and *number of airbags* would only be valid for *cars* and will be "null" for *vans*. Thus, a proper **MD** data model should be able to consider attributes only when necessary, depending on the categorization of dimensions.

Furthermore, let us suppose that apart from a high number of dimensions (e.g. 20) with their corresponding hierarchies, we have a considerable number of facts (e.g. 8) sharing dimensions and classification hierarchies. This would lead us to a very complex design, thereby increasing the difficulty in reading the modeled system. Therefore, a **MD** conceptual model should also provide techniques to **avoid flat diagrams**, allowing us to group dimensions and facts under some criteria to simplify the final model.

Once the structure of the **MD** model has been defined, final users usually identify a set of initial queries as a starting point for the subsequent data analysis phase. From these initial queries, users can apply a set of operations (usually called **OLAP** operations [24, 63]) to the **MD** view of data for further data analysis. These **OLAP** operations are usually as follows: roll-up (increasing the level of aggregation) and drill-down (decreasing the level of aggregation) along one or more classification hierarchies, slice-dice (selection and projection) and pivoting (re-orienting the **MD** view of data which also allows us to exchange dimensions for facts; i.e., symmetric treatment of facts and dimensions).

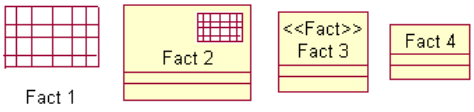


Figure 6.2: Different representations for a stereotyped class

6.3 Object-Oriented Multidimensional Modeling

Throughout this section, we will use a running example to illustrate the basics and the applicability of our **OO MD** approach. We use the same example presented in Section 6.2 and inspired by a case study from [48].

As our proposal addresses the **DW** design at a conceptual level, some implementation issues such as primary and foreign keys or data types are not our first priority. Therefore, the goal of our proposal is the representation of the main structural aspects of **MD** modeling at the conceptual level.

Class diagram: see UML (3.19, 3-34).

In our approach, the main structural properties of **MD** models are specified by means of a **UML** class diagram in which the information is clearly separated into facts and dimensions. The main features considered are the many-to-many relationships between facts and dimensions, degenerate facts and dimensions, multiple and alternative path classification hierarchies, and non-strict and complete hierarchies. Our approach proposes the use of **UML** packages in order to group classes together into higher level units creating different levels of abstraction, and therefore, simplifying the final model. In this way, when modeling complex and large **DW** systems, the designer is not restricted to use flat **UML** class diagrams.

Packages: see UML (2.15.2.4, 2-184), (3.13, 3-16).

Profile: see UML (2.6, 2-73), (2.15.4.2, 2-193).

Our proposal is formally defined as a **UML** extension by means of a **UML** profile. Although we provide the complete formal definition of our extension in the next section, we introduce the main stereotypes and some tagged values in this section. In a diagram, **UML** allows us to represent a stereotype in four different ways. In Figure 6.2, we show four possible representations of a class with the «Fact» stereotype (one of the stereotypes we propose): icon (the stereotype icon is displayed), decoration (the stereotype decoration is displayed inside the element), label (the stereotype name is displayed and appears inside guillemets), and none (the stereotype is not indicated).

6.3.1 Different Levels of Detail

Style issues, such as avoiding crossing lines, affect understandability of diagrams: messy diagrams are harder to read than clean ones. Moreover, the level of detail in models, also affect understandability because a highly detailed model is harder to comprehend than a less detailed one. In our **MD** modeling approach, thanks to the use of **UML** packages, we can elegantly represent huge and complex models at different levels of complexity.

Based on our experience in real-world cases, we have developed a set of design guidelines for using **UML** packages[6] in **MD** modeling. In **UML**, a package defines a *namespace*, so that two distinct elements contained in two distinct packages may have the same name. We summarize all the design guidelines in Table 6.1.

`Guideline 0a` is the foundation of the rest of the guidelines and summarizes our overall approach. This guideline closely resembles how data analyzers understand **MD** modeling. We have divided the design process into three levels (Figure 6.3 shows a summary of our proposal and in Table 6.1 we indicate in which level each guideline is applied):

Dependency: see UML (2.5.2.15, 2-33), (3.51, 3-90).

Level 1 : Model definition. A package represents a star schema[7] of a conceptual **MD** model. A dependency between two packages at this level indicates that the star schemas share at least one dimension, allowing us to consider *conformed dimensions*.

Level 2 : Star schema definition. A package represents a fact or a dimension of a star schema. A dependency between two dimension packages at this level indicates that the packages share at least one level of a dimension hierarchy.

Level 3 : Dimension/fact definition. A package is exploded into a set of classes that represent the hierarchy levels defined in a dimension package, or the whole star schema in the case of the fact package.

The **MD** model is designed in a top-down fashion by further decomposing a package. We have limited our proposal to three levels because "*deep hierarchies tend to be difficult to understand, since each level carries its own meanings*" [27].

[6] Package diagrams are a subset of class diagrams, but developers sometimes treat them as a separate technique.

[7] Although we use the concept star schema, it does not imply any relational implementation of the **DW**. We prefer to use a well-known concept instead of inventing a new term.

N°	Level	Guideline
0a		At the end of the design process, the **MD** model will be divided into three levels: model definition, star schema definition, and dimension/fact definition
0b		Before starting the modeling, define facts and dimensions and remark the shared dimensions and dimensions that share some hierarchy levels
1	1	Draw a package for each star schema, i.e., for every fact considered
2a	1	Decide which star schemas will host the definition of the shared dimensions; according to this decision, draw the corresponding dependencies
2b	1	Group together the definition of the shared dimensions in order to minimize the number of dependencies
2c	1	Avoid cycles in the dependency structure
3	2	Draw a package for the fact (only one in a star package) and a package for each dimension of the star schema
4a	2	Draw a dependency from the fact package to each one of the dimension packages
4b	2	Never draw a dependency from a dimension package to a fact package
5	2	Do not define a dimension twice; if a dimension has been previously defined, import it
6	2	Draw a dependency between dimension packages in order to indicate that the dimensions share hierarchy levels
7	3	In a dimension package, draw a class for the dimension class (only one in a dimension package) and a class for every classification hierarchy level (the base classes)
8	3	In a fact package, draw a class for the fact class (only one in a fact package) and import the dimension classes with their corresponding hierarchy levels
9	3	In a dimension package, if a dependency from the current package has been defined at level 2, import the corresponding shared hierarchy levels
10	3	In a dimension package, when importing hierarchy levels from another package, it is not necessary to import all the levels

Table 6.1: Multidimensional modeling guidelines for using packages

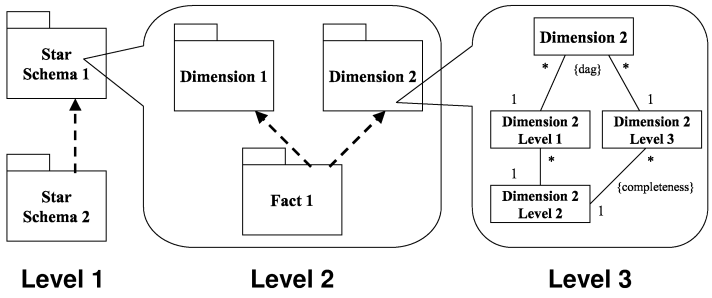


Figure 6.3: The three levels of a MD model explosion using packages

`Guidelines 2b` and `2c` make sure that cross-package dependencies result only in acyclic graphs[8] in order to keep things simple. Circular dependencies may be reduced by:

- Splitting one of the questionable packages into two smaller packages.
- Introducing a third intermediate package (try to factor the common elements out into a third package).
- Merging the questionable packages.

For example, in Figure 6.4 (a) the two «StarPackage» (stereotyped packages represented by means of icons) form a cycle that has been broken in Figure 6.4 (b) thanks to the introduction of a third «StarPackage» that contains the shared dimensions; this new package, that we call *utility package*, does not contain a «FactPackage», just the definition of the common elements to both packages. In Figure 6.4 (c) we show an alternative solution: the two «StarPackage» have been merged into a single one called StarPackage1-2, eliminating the shared elements, and therefore, avoiding repeating already-defined elements.

Applying Package Design Guidelines

The **DW** of our running example consists of three data marts: automobile sales, part sales, and service works. Figure 6.5 shows the first level of the model: on the left hand side, the packages are displayed with the normal **UML** presentation and the corresponding stereotype icon is placed in the upper right corner of the package

[8] Fowler states: "*As a rule of thumb, it is a good idea to remove cycles in the dependency structure*" [43].

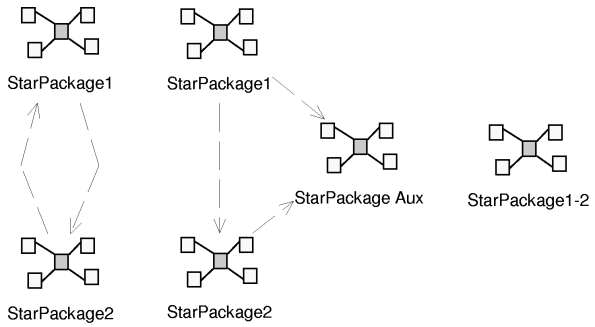


Figure 6.4: Model definition with and without cycles

symbol; on the right hand side, the entire package symbol has been "collapsed" into the corresponding stereotype icon. Through the rest of this chapter, we have adopted the second form of representing the stereotypes, because we consider it more expressive and symbolic, as well as it is also more understandable for the final users.

In the example shown in Figure 6.5, the first level is formed by three «StarPackage» that represent the different data marts that form the **DW** (G.1). A dashed arrow from one package to another one denotes a dependency between packages, i.e., the packages have some dimensions in common (G.2a). The direction of the dependency indicates that the common dimensions shared by the two packages were first defined in the package pointed to by the arrow (to start with, we have to choose a «StarPackage» to define the dimensions, and then, the other «StarPackage» can use them with no need to define them again). If the common dimensions had been first defined in another package, the direction of the arrow would have been different. In any case, it is highly recommended to group together the definition of the common dimensions in order to reduce the number of dependencies (G.2b) and also to avoid circular dependencies (G.2c).

Note: see UML (3.11, 3-13), (3.16, 3-26).

At any level of our proposal, the **DW** designer can use **UML** notes to add more information, remark some characteristic, clarify some ambiguous situation, or describe some concept in final users' terms. For example, in Figure 6.5, we have used three **UML** notes to remark the content of each package.

A package that represents a star schema is shown as a simple icon with names. The content of a package can be dynamically accessed by "zooming-in" to a detailed view. For example, Figure 6.6 shows the content of the package Auto-sales schema (level 2). The «Fact-Package» Auto-sales fact is represented in the middle of Figure 6.6,

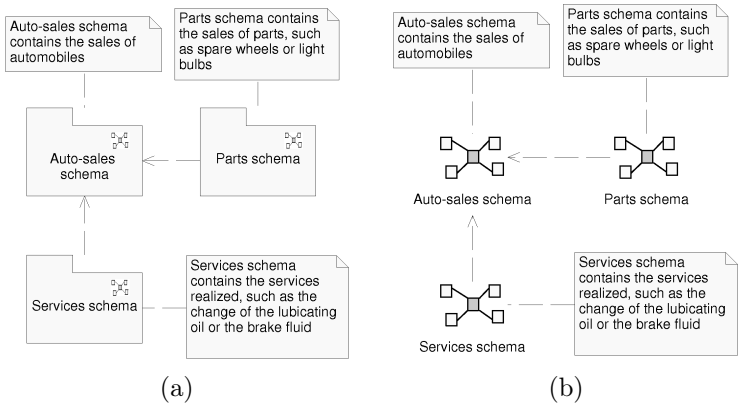


Figure 6.5: Level 1: different star schemas of the running example

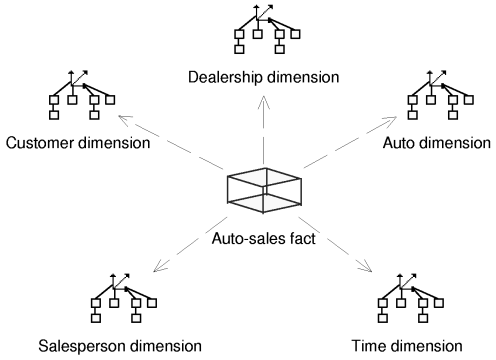


Figure 6.6: Level 2: Auto-sales schema

while the different «DimensionPackage» are placed around the «FactPackage» (G.3). As seen in Figure 6.6, a dependency is drawn from the «FactPackage» to each one of the «DimensionPackage», because the «FactPackage» comprises the whole definition of the star schema, and therefore, uses the definitions of dimensions related to the fact (G.4a). At level 2, it is possible to create a dependency from a «FactPackage» to a «DimensionPackage» or between «DimensionPackage», but we do not allow a dependency from a «DimensionPackage» to a «FactPackage», since it is not semantically correct in our proposal (G.4b).

Figure 6.7 shows the content of the package Services schema (level 2). As in the Auto-sales schema, the «FactPackage» is placed in the

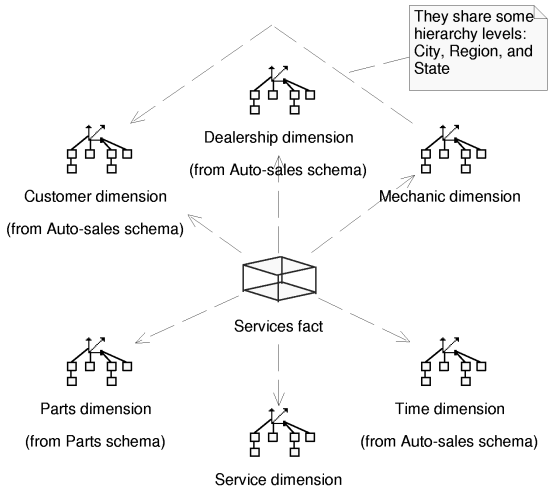


Figure 6.7: Level 2: Services schema

middle of Figure 6.7 and the «DimensionPackage» are placed around the «FactPackage» in a star fashion. The three «DimensionPackage» (Customer dimension, Dealership dimension, and Time dimension) have been previously defined in the Auto-sales schema (Figure 6.6), and Parts dimension has been previously defined in the Parts schema (not shown in this chapter), so all of them are imported in this package (G.5). Our approach does not forbid to define another dimension with or without the same name or properties in different «StarPackage». However, we highly recommend not to do it as we believe that this situation can lead us to a confusing or misleading diagram. Therefore, the name of the «StarPackage» where they have been previously defined appears below the package name (from Auto-sales schema and from Parts schema respectively). In this example, we can notice that it is possible to import packages defined in different «StarPackage». On the other hand, since Mechanic dimension and Service dimension have been defined in the current package, they do not show a package name. At this level, a dependency between «DimensionPackage» indicates that they share some hierarchy levels (G.6). For example, a dependency between Mechanic dimension and Customer dimension is represented because there is a shared hierarchy[9] (City, Region, and State), as we will see next.

Importing: see UML (2.5.2.32, 2-47), (3.38, 3-62).

The benefit of the **UML** importing mechanism is twofold. On

[9]We have decided to share a hierarchy for both dimensions to obtain a clearer design, although the designer may have decided not to do it if such sharing is not totally feasible.

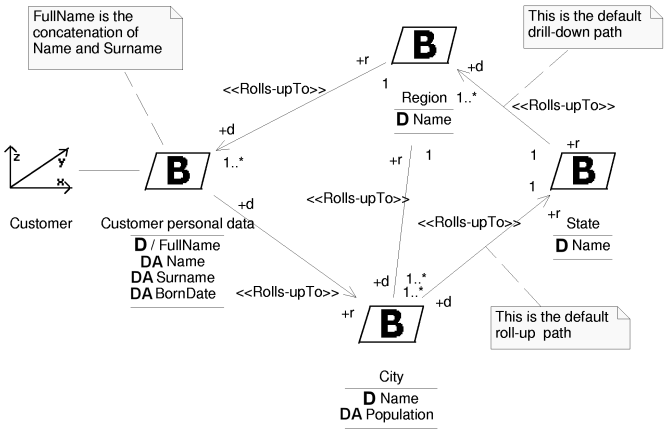


Figure 6.8: Level 3: Customer dimension

one hand, the **DW** designer only needs to define the different **MD** elements once, and therefore, they can be used anywhere in the model. On the other hand, as the **MD** elements are defined only once, any possibility of duplication and ambiguity is removed.

The content of the «DimensionPackage» and «FactPackage» is represented at level 3. The diagrams at this level are only comprised of classes and associations among them. For example, Figure 6.8 shows the content of the package Customer dimension (level 3), that contains the definition of the «Dimension» class (Customer) and the different hierarchy levels (Customer personal data, City, Region, and State) that are represented by «Base» classes (`G.7`). The hierarchy of a dimension defines how the different **OLAP** operations (roll-up, drill-down, etc.) can be applied [63].

As previously commented, Mechanic dimension and Customer dimension share some hierarchy levels, and therefore, there is a dependency between them (see Figure 6.7). Figure 6.9 shows the content of Mechanic dimension: this dimension contains six hierarchy levels, but three of them (City, Region and State) have been imported from another dimension.

Regarding «FactPackage», Figure 6.10 shows the content of the package Auto-sales fact (level 3). In this package, the whole star schema is displayed: the «Fact» class is defined with the corresponding measures (Commission and Price), and the «Dimension» classes with their corresponding hierarchy levels are imported (`G.8`). This level may become very complex because the dimensions may be very complex and of a considerable size due to a high number of dimension

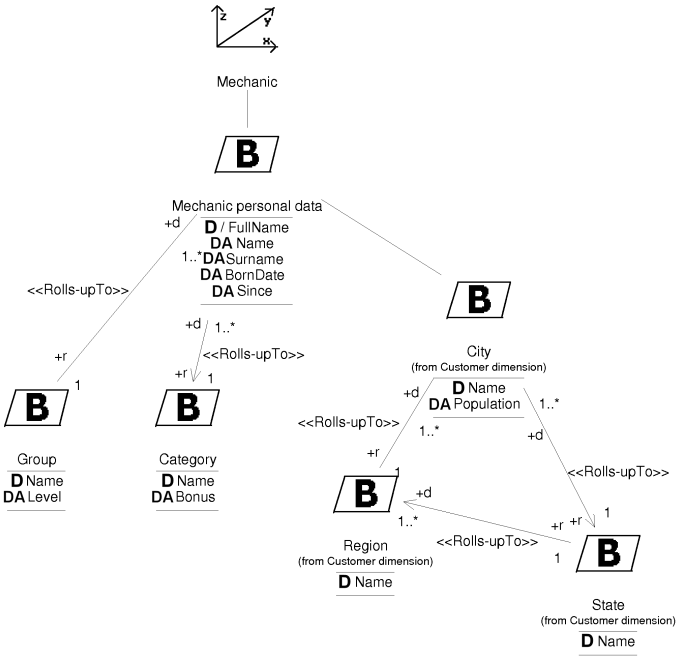


Figure 6.9: Level 3: Mechanic dimension

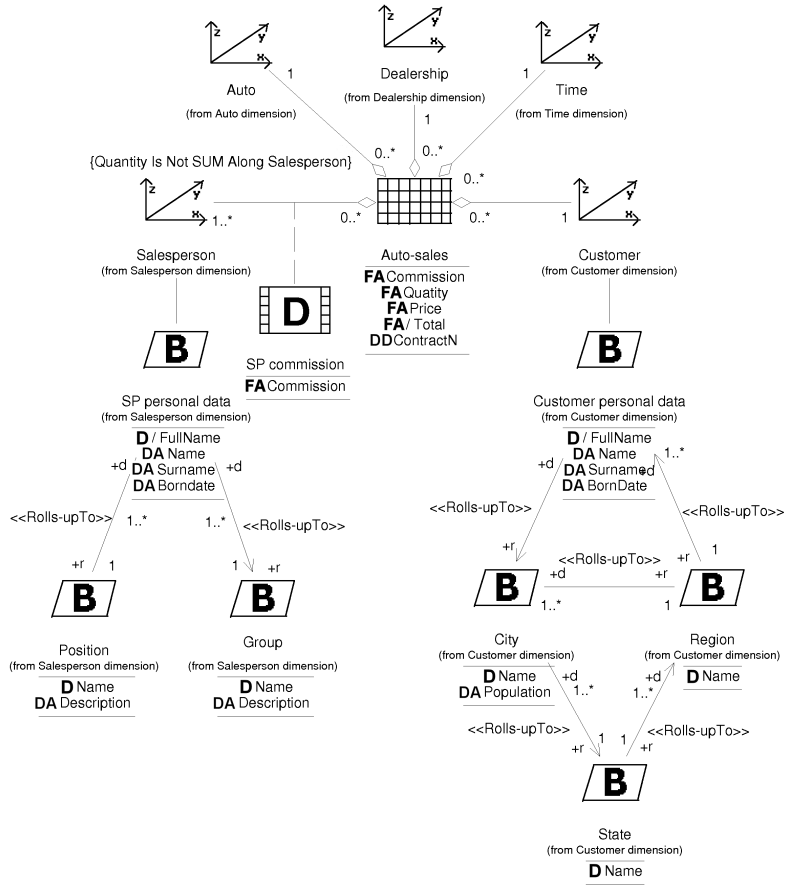


Figure 6.10: Level 3: Auto-sales fact

levels. However, the **DW** designer only has to import them from the corresponding «DimensionPackage». In Figure 6.10, we have hidden a part of the attributes and methods of some «Base» classes for the sake of simplicity.

Advantages of multi-fact schemas

Our approach allows **DW** designers to define a **MD** model that comprises multiple facts (multiple «StarPackage») linked between them by shared dimensions. This feature, commonly known as *fact constellation*, provides the structure to allow the final user to traverse the schema to perform an analysis known as *drill-across*. Moreover, keeping common dimensions will facilitate the future implementa-

tion of the **MD** model, e.g., in the case of a **DW**, the loading and refreshment processes will be simpler.

6.3.2 Facts and Dimensions

Facts and dimensions are represented by «Fact» and «Dimension» classes, respectively. Then, «Fact» classes are specified as composed classes by means of aggregation relationships of n «Dimension» classes, represented by a hollow diamond attached to the end of the relationship next to the «Fact» class. The flexibility of the aggregation in the **UML** allows us to represent *many-to-many* relationships between «Fact» and particular «Dimension» by indicating the 1..* cardinality at the end of the aggregation near the «Dimension». In our example shown in Figure 6.10, we can see how the «Fact» class Auto-sales has a many-to-one relationship with Auto, Dealership, Time, and Customer dimensions, but a many-to-many relationship with the Salesperson dimension.

6.3.3 Dimensions and Classification Hierarchy Levels

«Dimension» classes are composed of *classification hierarchy levels*; every classification hierarchy level is specified by a class called «Base» class. An association (represented by a stereotype called «Rolls-upTo») between «Base» classes specifies the relationship between two levels of a classification hierarchy. The only prerequisite is that these classes must define a Directed Acyclic Graph (DAG) rooted in the «Dimension» class. The DAG structure can represent both alternative path and multiple classification hierarchies.

Following Hüsemann's definitions [55], a «Dimension» contains a unique first hierarchy (or dimension) level called *terminal dimension level*. A roll-up path (or aggregation path in [55]) is a subsequence of dimension levels, which starts in a terminal dimension level (lower detail level) and ends in an implicit dimension level (not graphically represented) that represents all the dimension levels.

Rolename: see UML (3.43.2.6, 3-72).

We use roles to represent the way the two classes see each other in a «Rolls-upTo» association: role R represents the direction in which the hierarchy rolls-up, whereas role D represents the direction in which the hierarchy drills-down. Moreover, we use roles to detect and avoid cycles in a classification hierarchy, and therefore, help us to achieve the DAG condition. For example, on the left hand side of Figure 6.11, a classification hierarchy composed of three «Base» classes is represented. On the right hand side of Figure 6.11, a graph that symbolizes the classification hierarchy is shown and the direction of the arrows is based on the roles of the «Rolls-upTo» associations:

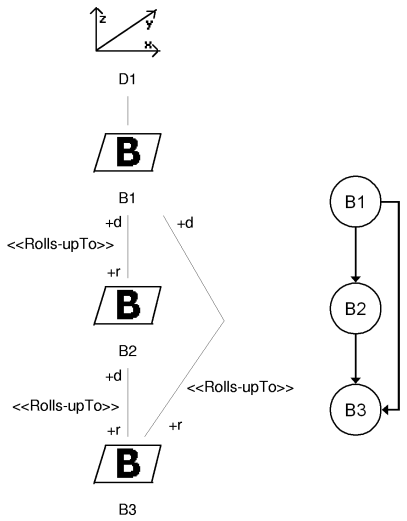


Figure 6.11: Classification hierarchy without cycles

from role D to role R (in the direction of rolling-up). As we can see in this figure, this classification hierarchy does not contain any cycle. However, the classification hierarchy shown in Figure 6.12 presents a cycle («Base» classes B2, B3, and B4), and therefore, this classification hierarchy is absolutely incorrect in our model.

Navigability: see UML (3.43.2.4, 3-72).

In **UML**, an arrow may be attached to the end of an association to indicate that navigation is supported toward the class attached to the arrow. In our proposal, the navigation is always supported toward both ends of an association (it is always possible to roll-up or drill-down on both directions), but the **DW** designer can use the **UML** navigability to deliberately represent a default roll-up or drill-down path when a «Base» class participates in multiple classification hierarchies[10]. However, only one default roll-up and one default drill-down path can start from a «Base» class. For example, in Figure 6.13 (a) we have represented a classification hierarchy that is incorrect because two default roll-up paths start from B1 (B1 rolls-up to B2 and B1 rolls-up to B3), and two default drill-down paths start from B4 (B4 drills-down to B2 and B4 drills-down to B3). The default roll-up and drill-down paths that are in conflict have been remarked with a dashed circle. This same classification hierarchy is

[10]Please, note that the navigability is an optional feature of our approach and it is not mandatory to always specify a roll-up or drill-down default path. Moreover, it is not necessary to draw the navigability when there is only one roll-up or drill-down path.

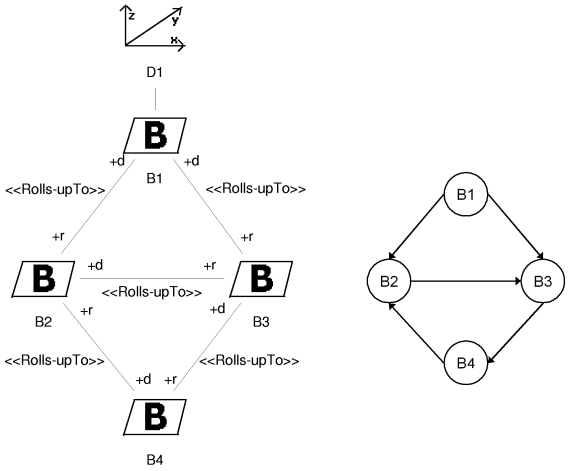


Figure 6.12: Classification hierarchy with one cycle

correctly represented in Figure 6.13 (b); note that it would have also been possible to define other default paths, such as B1 rolls-up to B3 and B4 drills-down to B3. Regarding our running example, we can see the use of the navigability (default path) concept in Figure 6.8, 6.9, and 6.10.

The multiplicity *1* and *1..** defined in the role R of a classification hierarchy level addresses the concepts of *strictness* and *non-strictness*, respectively. Strictness means that an object at a hierarchy's lower level belongs to only one higher-level object (e.g., as one month can be related to more than one season, the relationship between them is non-strict). In a **DW**, it is very important to identify and define non-strict hierarchies, because if they are not correctly treated, some problems such as double-counting can appear when aggregations are calculated in further design steps.

Moreover, defining an association as «Completeness» addresses the completeness of a classification hierarchy. By completeness we mean that all members belong to one higher-class object and that object consists of those members only; for example, all the recorded seasons form a year, and all the seasons that form the year have been recorded. Our approach assumes all classification hierarchies are non-complete by default.

In a **DW**, time is the dominant dimension. Many forms of analysis involve either trends or inter-period comparisons. Inmon [57] defines "*A data warehouse is a subject-oriented, integrated,* ***time-variant****, nonvolatile collection of data in support of management's*

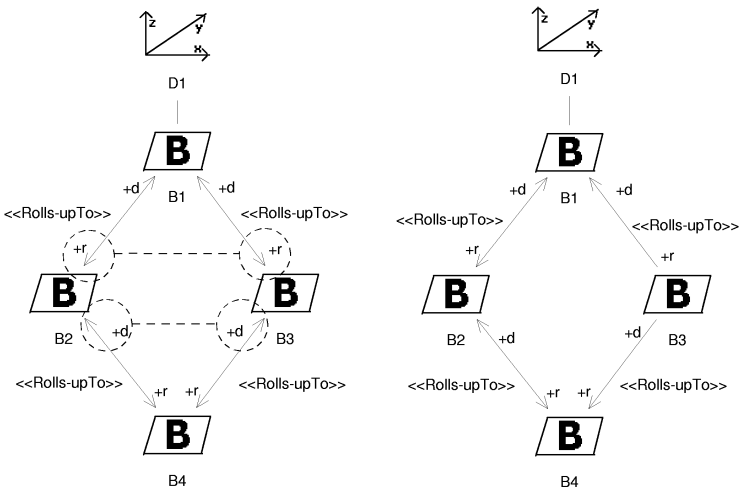


Figure 6.13: Classification hierarchy with wrong and right navigability

decisions", and Kimball [63] says that "*The **time dimension** is the one dimension virtually guaranteed to be present in every data warehouse, because virtually every data warehouse is a time series*". Due to this important fact, in our proposal a «Dimension» class includes a boolean tagged value called {isTime} that indicates whether it is a time dimension or not[11].

6.3.4 Categorization of Dimensions

Generalization: see UML (2.5.2.24, 2-38), (3.50, 3-86).

The *categorization of dimensions*, used to model additional features for a class's subtypes, is represented by means of UML generalization-specialization[12] relationships in our approach. However, only the parent of a categorization can belong to both a classification and generalization-specialization hierarchy at the same time. Moreover, multiple inheritance is not allowed in our approach.

An example of categorization for the Auto dimension is shown in Figure 6.14: Car and Van belong to a generalization-specialization relationship rooted in Auto general information; we have created this

[11] This will allow us an automatically generation of particular time structures in a target commercial **OLAP** tool.

[12] Generalization is a relationship between model elements indicating that one element (child or subclass) is a "type of" another element (parent or superclass). Objects of the child are substitutable for objects of the parent.

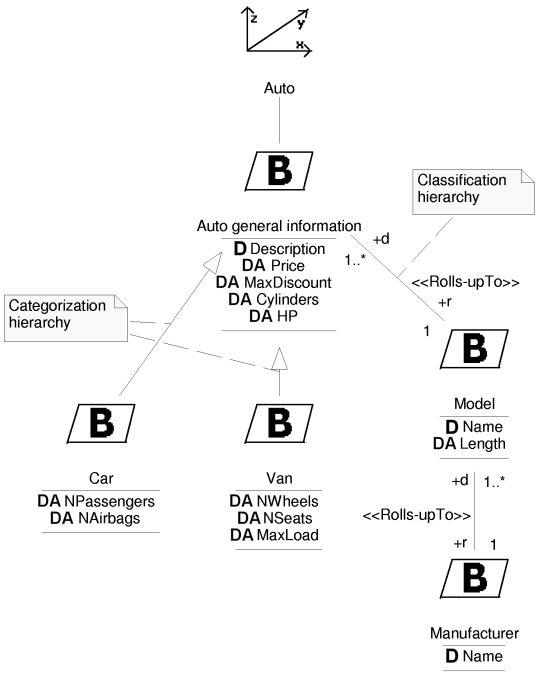


Figure 6.14: Level 3: Auto dimension

categorization because Car and Van contain different attributes.

6.3.5 Attributes

Only «Fact», «Base», and «DegenerateFact» (see Section 6.3.7) classes can have attributes. «Dimension» classes do not contain attributes, because they represent the concept of dimension and they are used as "anchorage points": the information about a dimension is represented in the corresponding hierarchy levels («Base» classes).

«Fact» classes consist of two kinds of attributes: «FactAttribute», which represent measures (the transactions or values being analyzed), and «DegenerateDimension» (see Section 6.3.6).

On the other hand, «Base» classes consist of three kinds of attributes: «OID», «Descriptor», and/or «DimensionAttribute». Every «Base» class can have one «OID» attribute (an identifying attribute) and must have one «Descriptor» attribute[13]. These attributes are necessary for an automatic exportation process into com-

[13]A descriptor attribute will be used as the default label in the data analysis in **OLAP** tools.

mercial **OLAP** tools, as these tools store this information in their metadata (if the «OID» attribute is not provided, then it will be automatically created in the exportation process). A «DimensionAttribute» provides descriptive information about dimension instances. A «DimensionAttribute» can be optional: it needs not be specified for each element of the corresponding level and therefore may contain *null* values. As a «DimensionAttribute» can be used to delimit the resulting set of a query, it is important to know if an attribute is optional (considered in our approach as *tagged values*, see Section 6.4.3), because then the results may be incomplete [55].

Derived element: see UML (2.5.2.27, 2-42), (3.52, 3-93).

«FactAttribute», «Descriptor», and «DimensionAttribute» can also be derived attributes. This situation is indicated by placing / before the corresponding name, and the derivation rule is defined as a tagged value called {derivationRule} of the corresponding stereotype. For example, in Figure 6.10, FullName attributes of SP personal data and Customer personal data are derived, because they are obtained by joining Name and Surname attributes (the derivation rules are not shown in order to avoid a cluttered diagram).

6.3.6 Degenerate Dimensions

Our approach also allows the **DW** designer to define degenerate dimensions in the «Fact» class, by using the stereotype «DegenerateDimension» for an attribute. A *degenerate dimension* is a «Dimension» that is stored as an attribute of the «Fact» class, but we do not explicitly represent it as a dimension in our diagram. *Degenerated dimensions* are useful when attempting to associate the facts in the **DW** with the original data sources [48, 63]. For example, in Figure 6.10, ContractN is a «DegenerateDimension» of Auto-sales that represents the identification number of the sale contract.

6.3.7 Degenerate Facts

In [48], the *degenerate fact* concept is defined as a measure recorded in the intersection table of a *many-to-many* relationship between the fact table and a dimension table. In our approach, we represent a «DegenerateFact» as a **UML** association class attached to a many-to-many aggregation relationship between a «Fact» class and a «Dimension» class[14]. This «DegenerateFact» class can contain «FactAttribute» and «DegenerateDimension».

Association class: see UML (2.5.2.4, 2-21), (3.46, 3-77).

For example, in Figure 6.10, SP commision is a «DegenerateFact» attached to the aggregation relationship between Auto-sales fact and

[14] Actually, an association class is an association that also has class properties (or a class that has association properties). Even though it is drawn as an association or a class, it is really just a single model element containing attributes.

Salesperson dimension. This «DegenerateFact» is the commission percentage that a salesperson received for a particular sale. The relationship between Auto-sales and Salesperson is many-to-many because different salespersons can take part in the same sale (they share the total commission), and a salesperson can also take part in different sales.

6.3.8 Additivity

We consider all measures as additive by default, i.e. the SUM operator can be applied to aggregate their measure values along all dimensions. Non-additivity and semi-additivity are considered by defining constraints on measures between brackets and placing them somewhere around the fact class. These constraints are represented in a property tag of the **UML** notation for clarity reasons, although they have formal underlying formulae and contain the allowed operators, if any, along the dimension that the measure is not additive. However, in large **MD** models, the readability can be reduced due to a great amount of additivity rules shown in a diagram. In these cases, we use *summarizability appendices*, as described in [55].

For example, in Figure 6.10, we can see that the attribute Quantity in the Auto-sales class cannot be aggregated along the Salesperson dimension by using the SUM operator. However, the AVG, MIN and MAX aggregation operators can still be applied to aggregate this attribute along the Salesperson dimension.

6.3.9 Merged Level 2 and 3

In some occasions, the **DW** designer needs to have a general overview of all facts, dimensions, and dependencies that a **DW** comprises. In our approach, this can be achieved if all the star schema definitions (level 2) are merged into one diagram. This diagram is automatically built and the **DW** designer cannot make changes in it because it is read-only. The same can be done at level 3, but the resulting diagram can be extraordinary complex in a big real **DW** with tens of dimensions and hundreds of hierarchy levels.

For example, in Figure 6.15 we show the merged level 2 of our running example with the three «FactPackage» and the different «DimensionPackage»; for each one of them, the legend (from ...) indicates in which «StarPackage» it has been defined. Moreover, the dependencies show where each «DimensionPackage» is used.

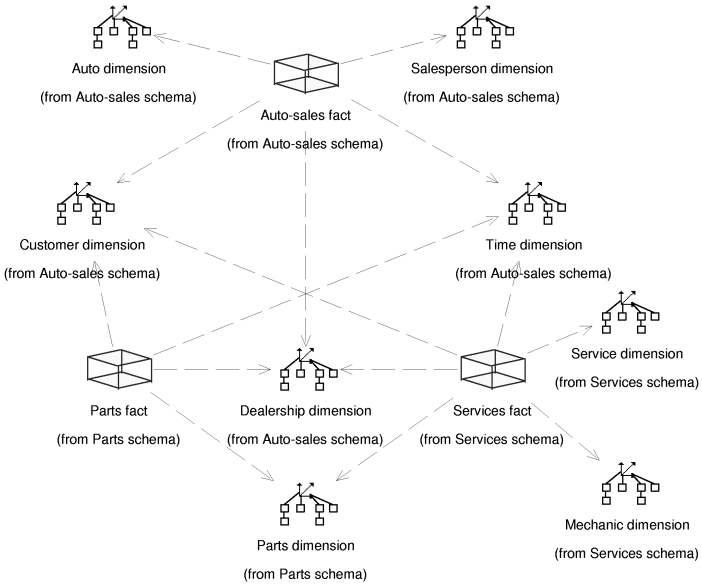


Figure 6.15: Merged level 2: representation of all the fact and dimension packages together

Figure 6.16: Metamodel divided into three packages

6.3.10 Metamodel

In this section, we present the metamodel of our **OO** conceptual **MD** approach using a **UML** class diagram. For the sake of simplicity, we have divided this diagram into three packages, as it is shown in Figure 6.16.

In Figure 6.17 we show the content of the package Level1. This package specifies the modeling elements that can be applied in the level 1 of our approach. In this level, only the «StarPackage» model element is allowed. We use the navigability of an association to denote the direction of a dependency or an importation. For example, a «StarPackage» may import «DimensionPackage»s from another «StarPackage». We also show the modeling elements that a

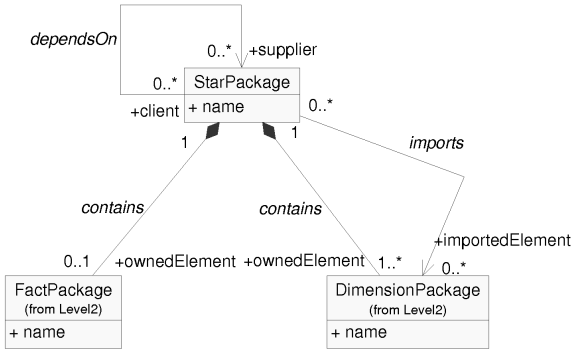


Figure 6.17: Metamodel: level 1

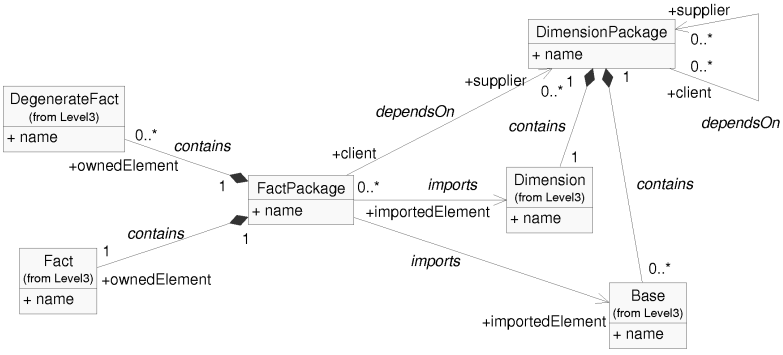


Figure 6.18: Metamodel: level 2

«StarPackage» can contain: «FactPackage» and «DimensionPackage».

In Figure 6.18 we show the content of the package Level2. In this level, the modeling elements that can be used are «FactPackage» and «DimensionPackage». A «FactPackage» may contain only one «Fact» and various «DegenerateFact», whereas a «DimensionPackage» may contain only one «Dimension» and various «Base».

Finally, in Figure 6.19 we show the content of the package Level3. This diagram represents the main **MD** properties of our modeling approach. In this way, dimensions and facts are represented using the classes «Dimension» and «Fact», respectively.

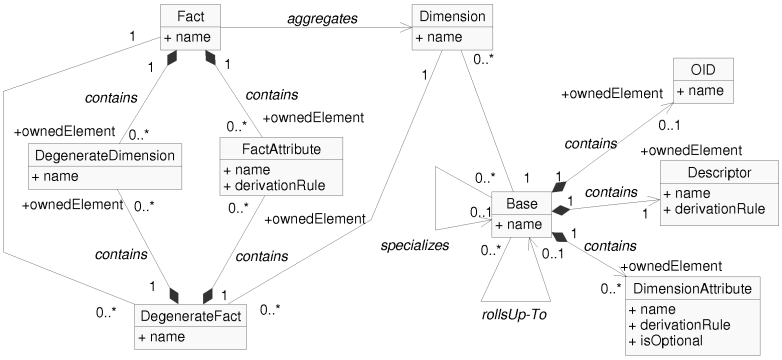


Figure 6.19: Metamodel: level 3

6.4 A UML Profile for Multidimensional Modeling

In this section, we present an extension to the **UML** in the form of a profile. Unfortunately, there does not exist a standard way for defining a **UML** profile.

According to [27], "*An extension to the UML begins with a brief description and then lists and describes all of the stereotypes, tagged values, and constraints of the extension. In addition to these elements, an extension contains a set of well-formedness rules. These rules are used to determine whether a model is semantically consistent with itself*". Therefore, based on this quote and our personal experience, we define our **UML** extension for **MD** modeling following the schema shown in Table 6.2.

For the definition of the stereotypes and tagged values, we follow the structure of the examples included in the **UML** [97]. In Table 6.3 and Table 6.4 we show the schemas followed in our definition of the stereotypes and the tagged values, respectively.

For the definition of well-formedness rules and constraints we use the **OCL** [97]. In this way, we avoid an arbitrary use of the profile. Moreover, using **OCL** has several benefits: it is a well-known constraint language, we do not need invest effort on defining a new language, and there is tool support for **OCL**. For the sake of readability, in the constraint definitions we use the conventions stated in the **UML**:

Conventions: see UML (2.3.3, 2-10).

- `self`, which can be omitted as a reference to the element defining the context of the invariant, has been kept for clarity. For example, we write self.feature instead of only feature.

- **Description**: A little description of the extension in natural language.
- **Prerequisite Extensions**: It indicates whether the current extension needs the existence of previous extensions.
- **Stereotypes**: The definition of the stereotypes.
- **Well-Formedness Rules**: The static semantics of the metaclasses are defined as a set of invariants defined by means of **OCL** expressions.
- **Comments**: Any additional comment, decision or example, usually written in natural language.

Table 6.2: Extension definition schema

- **Name**: The name of the stereotype.
- **Base class** (also called Model class): The UML metamodel element that serves as the base for the stereotype.
- **Description**: An informal description with possible explanatory comments.
- **Icon**: It is possible to define a distinctive visual cue (an icon).
- **Constraints**: A list of constraints defined by means of **OCL** expressions associated with the stereotype, with an informal explanation of the expressions.
- **Tagged values**: A list of all tagged values that are associated with the stereotype.

Table 6.3: Stereotype definition schema

- **Name**: The name of the tagged value.
- **Type**: The name of the type of the values that can be associated with the tagged value.
- **Multiplicity**: The maximum number of values that may be associated with the tagged value.
- **Description**: An informal description with possible explanatory comments.

Table 6.4: Tagged value definition schema

- In expressions where a collection is iterated, an iterator is used for clarity, even when formally unnecessary. However, the type of the iterator is usually omitted. For example, we write:
  self.contents->forAll(me | not me.oclIsKindOf(Package))
  instead of
  self.contents->forAll(not me.oclIsKindOf(Package)).

- The `collect` operation is left implicit where possible.For example, we write:
  self.connection.participant
  instead of
  self.connection->collect(participant).

We have defined fourteen stereotypes: three specialize in the Package model element[15], three specialize in the Class model element, one specializes in the AssociationClass model element, five specialize in the Attribute model element, and two specialize in the Association model element. In Figure 6.20, we have represented a portion of the UML metamodel[16] to show where our stereotypes fit. We have only represented the specialization hierarchies, as the most important fact about a stereotype is the base class that the stereotype specializes. In this figure, new stereotypes are colored in grey, whereas classes from the **UML** metamodel remain white.

Metamodel: see UML (2.2.1, 2-4), (2.4, 2-11).

Some issues of our **MD** approach, such as the derivation rule or the initial value of an attribute, are not defined in our stereotypes because these concepts have already been defined in the **UML** metamodel. We provide a list of these concepts in Table 6.5.

In the following, we present our extension following the extension definition schema shown in Table 6.2.

6.4.1 Description

This **UML** extension defines a set of stereotypes, tagged values, and constraints that enable us to design **MD** models. The stereotypes are applied to certain components that are particular to **MD** modeling, allowing us to represent them in the same model and on the same diagrams that describe the rest of the system. The **MD** models are divided into three levels: model definition (level 1), star schema definition (level 2), and dimension/fact definition (level 3).

The major elements to **MD** modeling are the Fact class and the Dimension class. A Fact class consists of FactAttributes and DegenerateDimensions. The hierarchy levels of a Dimension are represented

[15] We have based our **MD** extension on the most semantically similar constructs in the **UML** metamodel.

[16] All the metaclasses come from the Core Package, a subpackage of the Foundation Package.

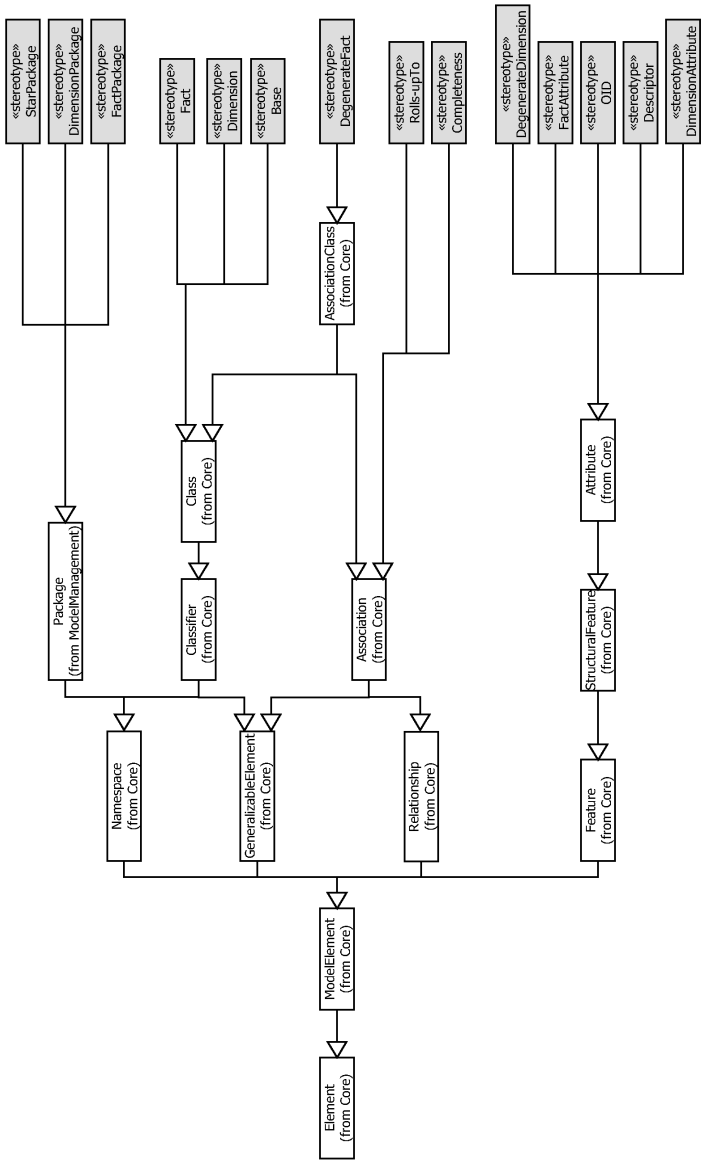


Figure 6.20: Extension of the UML with stereotypes

Table 6.5: Concepts inherited from the UML metamodel

Concept	**Comes from**	**Description**	**Used by**
name	`ModelElement`	It is an identifier for the `ModelElement`	Base, Completeness, Descriptor, Dimension, DimensionAttribute, Fact, FactAttribute, OID
documentation	`Element`	It is a comment, description or explanation of the `Element` to which it is attached	Base, Completeness, Descriptor, Dimension, DimensionAttribute, Fact, FactAttribute, OID
type	`StructuralFeature`	Designates the classifier whose instances are values of the feature	Descriptor, DimensionAttribute, FactAttribute, OID
initialValue	`Attribute`	An expression specifying the value of the `Attribute` upon initialization	Descriptor, DimensionAttribute, FactAttribute, OID
derived	`ModelElement`	A true value indicates that the `ModelElement` can be completely derived from other model elements and is therefore logically redundant	Descriptor, DimensionAttribute, FactAttribute

by means of Base classes. A Base class consists of OIDs, Descriptors, and DimensionAttributes. Finally, Rolls-upTo and Completeness association are also defined.

The correct use of this extension is assured thanks to the definition of 51 constraints specified both in natural language and in **OCL** expressions (to avoid redundancy).

6.4.2 Prerequisite Extensions

No other extension to the language is required for the definition of this extension.

6.4.3 Stereotypes

The stereotypes are presented depending on the base class that specializes: Package, Class, AssociationClass, Attribute, and Association.

Stereotypes of Package

Three stereotypes have been defined from the Package model element: StarPackage, DimensionPackage, and FactPackage.

- Name: **StarPackage**
- Base class: Package
- Description: Packages of this stereotype represent **MD** star schemas
- Icon: Figure 6.21 (a)
- Constraints:
  - A StarPackage can only contain FactPackages or DimensionPackages:[17]
    self.contents->forAll(me | me.oclIsTypeOf(FactPackage) or me.oclIsTypeOf(DimensionPackage))
  - A StarPackage can only contain one FactPackage:[18]
    self.ownedElement->select(me | me.oclIsTypeOf(FactPackage))->size <= 1
  - A StarPackage cannot import a FactPackage from another StarPackage (only DimensionPackage):
    self.importedElement->forAll(me | me.oclIsTypeOf(DimensionPackage))

[17] Some operations used in our extension are not from the **OCL** standard. For example, contents is an additional operation defined in the UML Specification [97]: "*The operation contents results in a Set containing the ModelElements owned by or imported by the Package*".

[18] It is not mandatory that every StarPackage has a FactPackage, because it is possible to have utility packages with only DimensionPackages for defining conformed dimensions to be imported by by other packages.

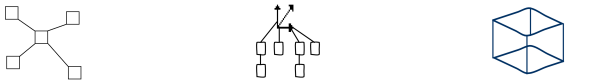
  - There are no cycles in the dependency structure:[19]
    not self.allSuppliers->includes(self)

- Tagged values: None

- Name: **DimensionPackage**
- Base class: Package
- Description: Packages of this stereotype represent **MD** dimensions
- Icon: Figure 6.21 (b)
- Constraints:
  - It is not possible to create a dependency from a DimensionPackage to a FactPackage (only to another DimensionPackage):
    self.clientDependency->forAll(d | d.supplier->forAll(me | me.oclIsTypeOf(DimensionPackage)))
  - There are no cycles in the dependency structure:
    not self.allSuppliers->includes(self)
  - A DimensionPackage cannot contain Packages:
    self.contents->forAll(me | not me.oclIsKindOf(Package))
  - A DimensionPackage can only contain Dimension or Base classes:
    self.contents->select(co | co.oclIsKindOf(Class))->forAll(f | f.oclIsTypeOf(Dimension) or f.oclIsTypeOf(Base))
  - A DimensionPackage must have a Dimension class (and only one):
    self.contents->select(me | me.oclIsTypeOf(Dimension))->size = 1
- Tagged values: None

- Name: **FactPackage**
- Base class: Package
- Description: Packages of this stereotype represent **MD** facts
- Icon: Figure 6.21 (c)
- Constraints:
  - There are no cycles in the dependency structure:
    not self.allSuppliers->includes(self)
  - A FactPackage cannot contain Packages:
    self.contents->forAll(me | not me.oclIsKindOf(Package))
  - A FactPackage can only contain Fact, DegenerateFact, Dimension or Base classes:
    self.contents->select(co | co.oclIsKindOf(Class))->forAll(f | f.oclIsTypeOf(Fact) or f.oclIsTypeOf(DegenerateFact) or f.oclIsTypeOf(Dimension) or f.oclIsTypeOf(Base))
  - A FactPackage must have a Fact class (and only one):
    self.contents->select(me | me.oclIsTypeOf(Fact))->size = 1
- Tagged values: None

[19] allSuppliers is an additional operation defined in the UML Specification [97]: "*The operation allSuppliers results in a Set containing all the ModelElements that are suppliers of this ModelElement, including the suppliers of these ModelElements. This is the transitive closure*".



StarPackage (a) DimensionPackage (b) FactPackage (c)

Figure 6.21: Stereotype icons of Package

Stereotypes of Class

Three stereotypes have been defined from the Class model element: Fact, Dimension, and Base.

- Name: **Fact**
- Base class: Class
- Description: Classes of this stereotype represent facts in a **MD** model
- Icon: Figure 6.22 (a)
- Constraints:
  - All attributes of a Fact must be DegenerateDimension or FactAttribute:
    self.feature->select(fe | fe.oclIsKindOf(Attribute))->forAll(f | f.oclIsTypeOf(DegenerateDimension) or f.oclIsTypeOf(FactAttribute))
  - All associations of a Fact must be aggregations[20] (neither *none* nor *composite*):
    self.association->forAll(as | as.aggregation = #aggregate)
  - A Fact can only be associated with Dimension classes:[21]
    self.allOppositeAssociationEnds->forAll(ae | ae.participant.oclIsTypeOf(Dimension))
- Tagged values: None

- Name: **Dimension**
- Base class: Class
- Description: Classes of this stereotype represent dimensions in a **MD** model
- Icon: Figure 6.22 (b)
- Constraints:
  - A Dimension cannot have neither attributes nor methods:
    self.feature->size = 0
  - All associations of a Dimension with a Fact must be aggregations at the end of the Fact (the opposite end):
    self.association.association->forAll(as | as.associationEnd.participant.oclIsTypeOf(Fact) implies as.associationEnd.aggregation = #aggregate)

[20] The part may be contained in other aggregates.

[21] allOppositeAssociationEnds is an additional operation defined in the UML specification [97]: "*The operation allOppositeAssociationEnds results in a set of all AssociationEnds, including the inherited ones, that are opposite to the Classifier*".

  - All associations of a Dimension with a Fact must not be aggregations at the end of the Dimension (the current end):
    self.association.association->forAll(as |
    as.associationEnd.participant.oclIsTypeOf(Fact) implies
    as.aggregation <> #aggregate)
  - A Dimension can only be associated with Fact or Base classes:
    self.allOppositeAssociationEnds->forAll(ae |
    ae.participant.oclIsTypeOf(Fact) or ae.participant.oclIsTypeOf(Base))
  - A Dimension can only be associated with one Base class:
    self.allOppositeAssociationEnds->select(ae |
    ae.participant.oclIsTypeOf(Base))->size <= 1

- Tagged values:
  - isTime:
    * Type: UML::Datatypes::Boolean
    * Multiplicity: 1
    * Description: Indicates whether the dimension represents a time dimension or not[22]

- Name: **Base**
- Base class: Class
- Description: Classes of this stereotype represent dimension hierarchy levels in a **MD** model
- Icon: Figure 6.22 (c)
- Constraints:
  - All attributes of a Base must be OID, Descriptor, or DimensionAttribute:
    self.feature->select(fe | fe.oclIsKindOf(Attribute))->forAll(f |
    f.oclIsTypeOf(OID) or f.oclIsTypeOf(Descriptor) or
    f.oclIsTypeOf(DimensionAttribute))
  - A Base must have a Descriptor attribute (and only one):
    self.feature->select(fe | fe.oclIsKindOf(Attribute))->select(f |
    f.oclIsTypeOf(Descriptor))->size = 1
  - A Base may have an OID attribute:
    self.feature->select(fe | oclIsKindOf(Attribute))->select(f |
    f.oclIsTypeOf(OID))->size <= 1
  - A Base can only be associated with Dimension or Base classes:
    self.allOppositeAssociationEnds->forAll(ae |
    ae.participant.oclIsTypeOf(Dimension) or
    ae.participant.oclIsTypeOf(Base))
  - A Base cannot be associated with itself (in order to avoid cycles):
    self.allOppositeAssociationEnds->forAll(ae | ae.participant <> self)
  - A Base class may only inherit from another Base class:
    self.generalization->size > 0 implies self.generalization.parent->
    forAll(me | me.oclIsTypeOf(Base))
  - A Base class may only be parent of another Base class:
    self.specialization->size > 0 implies self.specialization.child->forAll(me
    | me.oclIsTypeOf(Base))

[22]The "Time dimension" is treated differently from the others in **OLAP** tools.

Figure 6.22: Stereotype icons of Class and AssociationClass

  - A Base can only be child in one generalization (no multiple inheritance):
    self.generalization->size <= 1
  - A Base cannot simultaneously be a child in a generalization or specialization hierarchy and belong to an association hierarchy:
    (self.generalization->size = 1) implies (self.association->size = 0)

- Tagged values: None

Stereotypes of AssociationClass

One stereotype has been defined from the AssociationClass model element: DegenerateFact.

- Name: **DegenerateFact**
- Base class: AssociationClass
- Description: Classes of this stereotype represent degenerate facts in a **MD** model
- Icon: Figure 6.22 (d)
- Constraints:
  - All attributes of a DegenerateFact class must be DegenerateDimension or FactAttribute:
    self.feature->select(fe | fe.oclIsKindOf(Attribute))->forAll(f | f.oclIsTypeOf(DegenerateDimension) or f.oclIsTypeOf(FactAttribute))
  - A DegenerateFact association can only be connected to two elements[23]:
    self.connection->size = 2
  - One of the ends of a DegenerateFact has to be a Fact and the other end has to be a Dimension:
    self.connection.participant->exist(me | me.oclIsTypeOf(Fact)) and
    self.connection.participant->exist(me | me.oclIsTypeOf(Dimension))
- Tagged values: None

[23]In UML, an association can be connected to two or more elements.

Stereotypes of Attribute

Five stereotypes have been defined from the Attribute model element: DegenerateDimension, FactAttribute, OID, Descriptor, and DimensionAttribute.

- Name: **DegenerateDimension**
- Base class: Attribute
- Description: Attributes of this stereotype represent degenerate dimensions in a **MD** model
- Icon: Figure 6.23 (a)
- Constraints:
  - A DegenerateDimension cannot be derived:
    not self.derived
  - A DegenerateDimension can only belong to a Fact or a DegenerateFact:
    self.owner.oclIsTypeOf(Fact) or self.owner.oclIsTypeOf(DegenerateFact)
- Tagged values: None

- Name: **FactAttribute**
- Base class: Attribute
- Description: Attibutes of this stereotype represent attributes of Fact or DegenerateFact classes in a **MD** model
- Icon: Figure 6.23 (b)
- Constraints:
  - A FactAttribute can only belong to a Fact or a DegenerateFact:
    self.owner.oclIsTypeOf(Fact) or self.owner.oclIsTypeOf(DegenerateFact)
  - If a FactAttribute is derived, then it needs a derivation rule (an **OCL** expression):
    self.derived implies self.derivationRule.oclIsTypeOf(OclExpression)
- Tagged values:
  - derivationRule:
    - Type: UML::Datatypes::String
    - Multiplicity: 1
    - Description: If the attribute is derived, this tagged value represents the derivation rule

---

- Name: **OID**
- Base class: Attribute
- Description: Attributes of this stereotype represent OID attributes of Base classes in a **MD** model[24]
- Icon: Figure 6.23 (c)
- Constraints:

[24]See Section 3 or [132] for further information on OID and Descriptor attributes.

  - An OID can only belong to a Base:
    self.owner.oclIsTypeOf(Base)
  - An OID cannot be derived:
    not self.derived
- Tagged values: None

- Name: **Descriptor**
- Base class: Attribute
- Description: Attributes of this stereotype represent descriptor attributes of Base classes in a **MD** model
- Icon: Figure 6.23 (d)
- Constraints:
  - A Descriptor can only belong to a Base:
    self.owner.oclIsTypeOf(Base)
  - If a Descriptor is derived, then it needs a derivation rule (an **OCL** expression):
    self.derived implies self.derivationRule.oclIsTypeOf(OclExpression)
- Tagged values:
  - derivationRule:
    * Type: UML::Datatypes::String
    * Multiplicity: 1
    * Description: If the attribute is derived, this value represents the derivation rule

---

- Name: **DimensionAttribute**
- Base class: Attribute
- Description: Attributes of this stereotype represent attributes of Base classes in a **MD** model
- Icon: Figure 6.23 (e)
- Constraints:
  - A DimensionAttribute can only belong to a Base:
    self.owner.oclIsTypeOf(Base)
  - If a DimensionAttribute is derived, then it needs a derivation rule (an **OCL** expression):
    self.derived implies self.derivationRule.oclIsTypeOf(OclExpression)
- Tagged values:
  - derivationRule:
    * Type: UML::Datatypes::String
    * Multiplicity: 1
    * Description: If the attribute is derived, this value represents the derivation rule
  - isOptional:
    * Type: UML::Datatypes::Boolean
    * Multiplicity: 1
    * Description: An optional attribute needs not be specified for each element of the corresponding Base class and therefore may contain “null” values

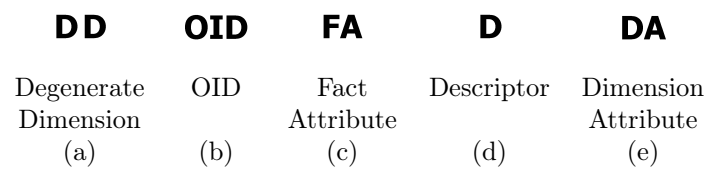


Figure 6.23: Stereotype icons of Attribute

Stereotype of Association

Two stereotypes have been defined from the Association model element: Rolls-upTo and Completeness.

- Name: **Rolls-upTo**
- Base class: Association
- Description: Associations of this stereotype represent associations between Base classes
- Icon: None
- Constraints:
  - The ends of a Rolls-upTo association can only be Base classes:
    self.connection.participant->forAll(pa | pa.oclIsTypeOf(Base))
  - A Rolls-upTo association can only be connected to two elements:[25]
    self.connection->size = 2
  - In a Rolls-upTo association, one of the ends contains the role r and the other end contains the role d:[26]
    self.associationEnd->exists(ae | ae.name = 'r') and self.associationEnd->exists(ae | ae.name = 'd')
- Tagged values: None

- Name: **Completeness**
- Base class: Association
- Description: Associations of this stereotype represent complete associations[27] between Base classes
- Icon: None
- Constraints:
  - The ends of a Completeness association can only be Base classes:
    self.connection.participant->forAll(pa | pa.oclIsTypeOf(Base))
  - A Completeness association can only be connected to two elements:
    self.connection->size = 2
  - In a Completeness association, one of the ends contains the role r and the other end contains the role d:
    self.associationEnd->exists(ae | ae.name = 'r') and self.associationEnd->exists(ae | ae.name = 'd')
- Tagged values: None

[25] In the UML, an association can have more that two association ends.
[26] The role is the name of the AssociationEnd.
[27] A complete association means that all members belong to one higher-class object and that object consists of those members only.

6.4.4 Well-Formedness Rules

Namespace

- All the classes in a **MD** model must be Fact, Dimension, or Base:[28]
  self.allContents->forAll(oclIsKindOf(Class) implies (oclIsTypeOf(Fact) or oclIsTypeOf(Dimension) or oclIsTypeOf(Base)))
- All the packages in a **MD** model must be StarPackage, FactPackage, or DimensionPackage:
  self.allContents->forAll(oclIsKindOf(Package) implies (oclIsTypeOf(StarPackage) or oclIsTypeOf(FactPackage) or oclIsTypeOf(DimensionPackage)))

6.4.5 Comments

Next, we summarize the UML elements we have just used or defined to consider the main relevant **MD** properties:

- Facts and dimensions: they are represented by means of Fact and Dimension stereotypes.
- Many-to-many relationships: thanks to the flexibility of the shared-aggregation relationships, we can consider many-to-many relationships between facts and particular dimensions by means of the 1..* cardinality on the dimension class role. In these cases, a DegenerateFact can be defined to add more information.
- Derived measures: they are represented by means of derived attributes from the UML metamodel and the tagged value derivationRule we have defined in the Descriptor, DimensionAttribute, and FactAttribute stereotypes.
- Classification hierarchies: they are considered by means of the association between Dimension and Base stereotypes.
- Strictness: the multiplicity 1 and 1..* defined in the target associated class role of a classificaton hierarchy address the concepts of strictness and nonstrictness.
- Completeness: the stereotype Completeness addresses the completeness of a classification hierarchy.
- Categorizing dimensions: we use generalization-specialization relationships to categorize a Dimension.

[28] allContents is an additional operation defined in the UML specification [97]: "*The operation allContents results in a Set containing all ModelElements contained by the Namespace*".

6.5 Implementation of Multidimensional Models

MDA [98] is an **OMG** standard that addresses the complete life cycle of designing, deploying, integrating, and managing applications. **MDA** separates the specification of system functionality from the specification of the implementation of that functionality on a specific technology platform, i.e. a ***Platform Independent Model*** **(PIM)** can be transformed into multiple ***Platform Specific Model*** **(PSM)** in order to execute on a concrete platform (see left hand side of Figure 6.24).

MOF 2.0 **QVT** is an under-developing standard for expressing model transformations, which can define transformation rules between two **MOF** compliant models. In response to the Request for Proposal (RFP) of **QVT**, different transformation approaches have been proposed over the last two years [33]. One of the most remarkable approaches is QVT-Partners [103].

QVT-Partners proposes a possibly extended version of **OCL** 2.0 as the query language and provides a standard language called MTL[29] (Model Transformation Language) for relations and mappings. In QVT-Partners, complex transformations can be built by composing simpler transformations using composition functions. Moreover, QVT-Partners suggests a sequence of steps that lead to an executable transformation that can be executed thanks to a model transformation engine (e.g. Inria MTL Engine [58]).

Model transformation is the process of converting one model to another model. In [44], model transformations are categorized along vertical (a source model is transformed into a target model at a different level of abstraction) and horizontal (a source model is transformed into a target model that is at the same level of abstraction) dimensions.

We have aligned our **MD** proposal with the **MDA** approach; thus, as presented through this chapter, we accomplish the conceptual modeling of a **DW** without considering any aspect of the implementation in a concrete target platform, thereby providing a **PIM**. We have developed an algorithm that, from the **MD** models accomplished by using our **UML** profile, generates the corresponding implementation in different platforms (relational, object-relational, etc.) through a vertical transformation, thereby allowing different **PSM**. In this section, we present the transformation process from an **MD** model to a relational one; on the right hand side of Figure 6.24, we show a high-level view of a transformation process from an **MD** model to the relational model, in which we generate the specific platform struc-

[29] The syntax of this language resembles C, C++ and Java language family.

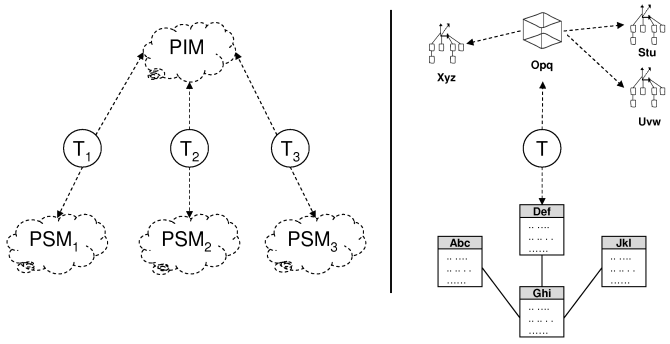


Figure 6.24: Transformation of a Multidimensional Model

tures according to the modeling elements.

For example, the next code represents the **QVT** implementation of the mapping for the «Fact» class into a table. The body of a mapping contains an object expression that creates an object (method `new`) and produces the output. In the body of a mapping, **OCL** is used to select and filter the model elements.

```
mapping FactToTable {
  domain {
    (Fact)[name = fn, attibutes = atts, associations = ass]
  }
  body {
    ta = new Table()
    ta.name = fn
    ta.columns = atts->iterate(a cols = {} | cols +
                 FactAttributeToColumn(a))
    ta.keys = atts->forAll(a keys = {} | keys + DDToKey(a))
    ta.foreignKeys = ass->forAll(a fkeys = {} | fkeys +
                     AggregationToForeignKey(a))
  }
}

mapping FactAttributeToColumn {
  domain {
    (FactAttribute)[name = fn, type = ty]
  }
  body {
    co = new Column()
    co.name = fn
    co.type = ty
  }
}

mapping DDToKey {
  domain {
```

```
    (DegenerateDimension)[name = fn, type = ty]
  }
  body {
    ke = new Key()
    ke.name = fn
    ke.type = ty
  }
}

mapping AggregationToForeignKey {
  domain {
    (Association)[name = fn, source = aggS , destination = aggD ]
  }
  body {
    fk = new ForeignKey()
    fk.name = fn
    // ForeignKey is autoincrement
    fk.type = 'auto'
  }
}
```

6.6 Conclusions

In this chapter, we have presented an extension of the **UML** as a *profile* that allows us to accomplish the conceptual modeling of **DW** by representing the major relevant **MD** properties at the conceptual level. We have invested some effort on ensuring that all the concepts have a well-defined semantic basis. Therefore, this extension contains the needed stereotypes, tagged values and constraints for a complete and powerful **MD** modeling. We have used the **OCL** to specify the constraints attached to these new defined elements, thereby avoiding an arbitrary use of them. We have also programmed this extension in a well-known visual modeling tool, Rational Rose. The main relevant advantage of our approach is that it uses the **UML**, a widely-accepted object-oriented modeling language, which saves developers from learning a new model and its corresponding notations for specific **MD** modeling. Furthermore, the **UML** allows us to represent some **MD** properties that are hardly considered by other conceptual **MD** proposals. On the other hand, a frequent criticism highlighted at diagrammatic notations is their scalability; in our approach, thanks to the use of packages, we can elegantly represent huge and complex models at different levels of complexity without suffering the scalability problem.

For more information about our *add-in for Rational Rose*, consult appendix E, pp. 253.

Chapter 7

Data Mapping Diagrams for Data Warehouses

	Source	Integration	Data Warehouse	Customization	Client
Conceptual	SCS	DM	DWCS	DM	CCS
Logical	SLS	ETL Process	DWLS	Exporting Process	CLS
Physical	SPS	Transportation Diagram	DWPS	Transportation Diagram	CPS

In **DW** scenarios, **ETL** processes are responsible for the extraction of data from heterogeneous operational data sources, their transformation (conversion, cleaning, normalization, etc.) and their loading into the **DW**. In this chapter, we present a framework for the design of the **DW** back-stage (and the respective **ETL** processes) based on the key observation that this task fundamentally involves dealing with the specificities of information at very low levels of granularity including transformation rules at the attribute level. Specifically, we present a disciplined framework for the modeling of the relationships between sources and targets in different levels of granularity (including coarse mappings at the database and table levels to detailed inter-attribute mappings at the attribute level). In order to accomplish this goal, we extend **UML** (Unified Modeling Language) to model attributes as first-class citizens. In our attempt to provide views of the design artifacts in different levels of detail, our framework is based on a principled approach in the usage of **UML** packages, to allow zooming in and out the design of a scenario.

Contents

7.1 Introduction

In **DW** scenarios, **ETL** processes are responsible for the extraction of data from heterogeneous operational data sources, their transformation (conversion, cleaning, normalization, etc.) and their loading into the **DW**. **DW** are usually populated with data from different and heterogeneous operational data sources such as legacy systems, relational databases, COBOL files, Internet (**XML**, web logs) and so on. It is well recognized that the design and maintenance of these **ETL** processes (also called **DW** back stage) is a key factor of success in data warehousing projects for several reasons, the most prominent of which is their critical mass; in fact, **ETL** development can take up as much as 80% of the development time in a data warehousing project [123, 124].

For more information about *ETL*, consult chapter 9, pp. 133.

As in most Information System projects, the early stages of a **DW** project are crucial for the success of the overall project. First of all, a **DW** conceptual model should be used to obtain a **DW** conceptual schema based on main user requirements. Although some authors do not pay too much attention to this phase ([65]), we argue that a **DW** conceptual model is a valuable tool in the design process, for various reasons to be explained in the next paragraphs. Still, a key observation should be made at this point: *the conceptual modeling that concerns the integration of computerized, database-centric systems is fundamentally different from the one involving the capturing of user requirements from the part of humans.* In particular, whereas the latter can afford to be imprecise, in order to capture the time-varying, possibly fuzzy, user requirements, or even to direct the end-user towards a reasonable goal, the former simply cannot afford this luxury. In fact, if any integration is ever to take place, the analysis of the involved systems must eventually deal with the lowest granule of information (typically attributes) in a *precise* manner. At the same time, interaction is absolutely difficult: it is practically impossible to alter the data sources in order to fit the needs of the integration effort. To make the problem harder, flexibility is another desideratum: the overall design process requires several levels of abstraction as it evolves over time. *Therefore, the conceptual modeling for the integration of data-centric systems has to retain flexibility without paying the price of imprecision.* As we will briefly show in this chapter, the specification of these transformations in real world projects lead us to a very complex documentation and is difficult to read and understand.

Despite the importance of designing the mapping of the data sources to the **DW** structures along with any necessary constraints and transformations, unfortunately, there are few models that can be used by the designers to this end. So far, the front end of the **DW** has monopolized the research on the conceptual part of **DW** modeling,

while few attempts have been made towards the conceptual modeling of the back stage [134, 128]. Still, to this day, there is no model that can combine (a) the desired detail of modeling data integration at the attribute level and (b) a widely accepted modeling formalism such as the **ER** model or **UML**. One particular reason for this, is that both these formalisms are simply not designed for this task; on the contrary, they treat attributes as second-class, weak entities, with a descriptive role. Of particular importance is the problem that in both models attributes cannot serve as an end in an association or any other relationship.

One might argue that the current way of modeling is sufficient and there is no real need to extend it in order to capture mappings and transformations at the attribute level. There are certain reasons that we can list against this argument:

- The design artifacts are acting as blueprints for the subsequent stages of the **DW** project. If the important details of this design (e.g., attribute interrelationships) are not documented, the blueprint is problematic. Actually, one of the current issues in **DW** research involves the efficient documentation of the overall process [108]. Since design artifacts are means of communicating ideas, it is best if the formalism adopted is a widely used one (e.g., **UML** or **ER**).

- The design should reflect the architecture of the system in a way that is formal, consistent and allows the what-if analysis of subsequent changes. Capturing attributes and their interrelations as first-class citizens improves the design significantly with respect to all these goals. At the same time, the way this issue is handled now would involve a naive, informal documentation through **UML** notes.

- Thanks to modeling attribute interrelationships, we can treat the design artifact as a graph and actually measure the aforementioned design goals. Again, this would be impossible with the current modeling formalisms.

To address all the aforementioned issues, in this chapter, we present an approach that enables the tracing of the **DW** back-stage (**ETL** processes) particularities at various levels of detail. This is enabled by an additional view of a **DW**, called the DATA MAPPING. In this new diagram, we treat attributes as first-class modeling elements of the model. This gives us the flexibility of defining models at various levels of detail. Naturally, since **UML** is not initially prepared to support this behavior, we solve this problem thanks to the extension mechanisms that it provides. Specifically, we employ a formal,

strict mechanism that maps attributes to proxy classes that represent them. Once mapped to classes, attributes can participate in associations that determine the inter-attribute mappings, along with any necessary transformations and constraints.

In summary, the main contributions of this chapter are:

- The presentation of a design diagram, called the data mapping diagram, defined at various levels of granularity, that implements the flexibility desideratum: while concepts are still fuzzy, the designers can choose high level representations of the **DW** back-stage scenario, whereas in later stages they can specialize the design all the way to the attribute level.
- The representation of attributes as first-class modeling elements in **UML** class diagrams for the purpose of tracing the particularities of the integration, filtering, and transformation of data at the attribute level (as required in designing **ETL** processes). This involves the extension of **UML** and the definition of the intra-attribute mappings.
- The definition of new stereotypes to accommodate the extension of **UML** and the specificities of the data mapping diagram: «Attribute» and «Contain» for representing attributes as first-class modeling elements, and «Mapping», «Map», «MapObj», «Domain», «Range», «Input», «Output», and «Intermediate» for the definition of data mappings.

The rest of this chapter is structured as follows. In Section 7.2, we introduce a motivating example that will be followed throughout the chapter. In Section 7.3, we show how attributes can be represented as first-class modeling elements in **UML**. In Section 7.4, we present our approach to model data mappings in **ETL** processes at the attribute level. Finally, in Section 7.5 we present the main conclusions.

7.2 Motivating Example

To motivate our discussion we will introduce a running example inspired by the example presented in [30]. In our setting, we consider that the designer wants to build a **DW** from the retail system of a company. Naturally, we consider only a small part of the **DW**, where the target fact table has to contain only the quarterly sales of the products belonging to the computer category, whereas the rest of the products are discarded.

In Figure 7.1, we show a bird's eye view of the **DW**, composed of three stereotyped packages that represent the SCS (Source Conceptual Schema), the DWCS (Data Warehouse Conceptual

Figure 7.1: Bird's eye view of the data warehouse

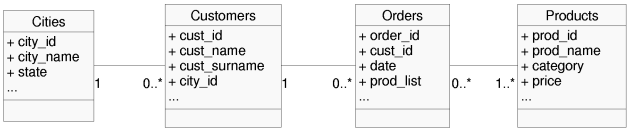


Figure 7.2: Source Conceptual Schema (SCS)

Schema) and the DM (Data Mapping) that define the mapping between the SCS and the DWCS.

Naturally, this high-level view can be further explored and detailed. In Figure 7.2, we zoom-in the definition of the SCS, which represents the sources that feed the **DW** with data. In this example, the data source is composed of four entities represented as **UML** classes: Cities, Customers, Orders, and Products. The meaning of the classes and their attributes, as depicted in Figure 7.2 is straightforward. The "..." shown in this figure simply indicates that other attributes of these classes exist, but they are not displayed for the sake of simplicity (this use of "..." is not a **UML** notation).

Finally, the DWCS of our motivating example is shown in Figure 7.3. The **DW** is composed of one fact (ComputerSales) and two dimensions (Products and Time).

In Table 7.1, we show a partial example of a basic form [48] for documenting an **ETL** transformation process between data sources and a **DW**. The form is composed of three main columns: source (it identifies the system and the field from which the data originates), transformation (it describes the transformation that must be performed), and destination (it describes the target in the **DW**). Nev-

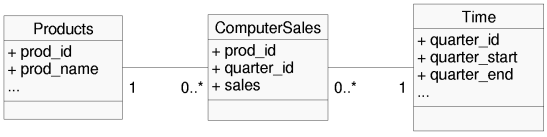


Figure 7.3: Data Warehouse Conceptual Schema (DWCS)

ertheless, documenting a transformation process in this way presents some important problems, such as the lack of coherence between the model diagrams and the transformation form, or the difficulty in understanding and managing a transformation form in the case of huge **DW** in real world projects. Therefore, and based on our experience in designing real world **DW**, we argue that these transformation processes should be defined in the model diagrams as other parts of the **DW**.

In this chapter, we present an additional view of a **DW**, called the DATA MAPPING that shows the relationships between the data sources (SCS) and the **DW** (DWCS) and between the **DW** (DWCS) and the clients' structures (CCS) at the conceptual level. In this new diagram, we need to treat attributes as first-class modeling elements of the models, since we need to depict their relationships at attribute level. Therefore, we also propose a **UML** extension to accomplish this goal in this chapter. To the best of our knowledge, this is the first proposal of representing attributes as first-class modeling elements in **UML** diagrams.

7.3 Attributes as First-Class Modeling Elements in UML

Both in the **ER** model and in **UML**, attributes are embedded in the definition of their comprising "element" (an entity in the **ER** or a class in **UML**), and it is not possible to create a relationship between two attributes. As we have already explained in the introduction of this chapter, in some situations (e.g., data integration, constraints over attributes, etc.) it is desirable to represent attributes as first-class modeling elements. Therefore, in this section we will present an extension of **UML** to accommodate attributes as first-class citizens. We have chosen **UML** instead of **ER** on the grounds of its higher flexibility in terms of employing complementary diagrams for the design of a certain system. We anticipate that a similar extension is also feasible for the **ER** model, too.

Association: see UML (2.5.2.3, 2-19).

Throughout this chapter, we frequently use the term *first-class citizens* for elements of our modeling languages. Conceptually, first-class citizens refer to fundamental modeling concepts, on the basis of which our models are built. Technically, first-class citizens involve an identity of their own, and they are possibly governed by integrity constraints (e.g., relationships must have at least two ends refering to classes.). In a **UML** class diagram, two kinds of modeling elements are treated as first-class citizens. Classes, as abstract representations of real-world entities are naturally found in the center of the modeling effort. Being first-class citizens, classes stand-alone entities also

Table 7.1: Example of transformation form

Source			**Transformation**	**Destination**	
System	**Table or file**	**Field or column**		**Table**	**Field or column**
...	...	...	...	...	...
...	...	...	...	...	...
OLTP	Orders	date	Check not null Transform into quarter	ComputerSales	quarter_id
...	...	...	...	...	...
...	...	...	...	...	...
OLTP OLTP	Orders Products	prod_list price	Split prod_list to obtain quantity SUM(quantity * price)	ComputerSales	sales
...	...	...	...	...	...
...	...	...	...	...	...
OLTP	Products	prod_id	Generate surrogate key by lookup	Products	prod_id
OLTP	Products	prod_name	Check not null	Products	prod_name
...	...	...	...	...	...
...	...	...	...	...	...

Figure 7.4: Dual view: class diagram and attribute/class diagram

acting as attribute containers. The relationships between classes are captured by associations. Associations can be also first-class modeling elements, called association classes. Even though an association class is drawn as an association and a class, it is really just a single model element [97]. An association class can contain attributes or can be connected to other classes. However, the same is not possible with attributes.

Naturally, in order to allow attributes to play the same role in certain cases, we propose the representation of attributes as first-class modeling elements in **UML**. In our approach, classes and attributes are defined as normally in **UML**. However, in those cases where it is necessary to treat attributes as first-class modeling elements, classes are imported to the *attribute/class diagram*, where attributes are automatically represented as classes; in this way, the user only has to define the classes and the attributes once[1]. In Figure 7.4 we schematically represent this dual "view" we propose in terms of packages: our proposed attribute/class diagram imports the elements from the class diagram; we use a **UML** standard «import» dependency because "*the public contents of the target package are added to the namespace of the source package*" [97], and therefore it is not necessary to qualify the element names with the package name.

In the importing process from the class diagram to the attribute/-class diagram, we refer to the class that contains the attributes as the *container class* and to the class that represents an attribute as the *attribute class*. In the sequel, we formally define attribute/class diagrams, along with the new stereotypes, «Attribute» and «Contain».

Definition. *Attribute classes are materializations of the «Attribute» stereotype, introduced specifically for representing the attributes of a class.*

The following constraints apply for the correct definition of an attribute class as a materialization of an «Attribute» stereotype:

- *Naming convention*: the name of the attribute class is the name

[1]Obviously, this is a CASE tool functionality: the corresponding CASE tool that supports this kind of diagram should dynamically generate the corresponding attribute/class diagram.

of the corresponding container class, followed by a dot and the name of the attribute.

- *No features*: an attribute class can contain neither attributes nor methods.

Attribute: see UML (2.5.2.6, 2-24), (3.25, 3-41).

- *Tag definitions*: an attribute class contains the following tag definitions that represent the properties of an attribute model element:
  - changeability: a characterization that determines whether the value of the attribute may be modified after the object is created. Valid values for this property are *changeable*, *frozen*, and *addOnly*.
  - initialValue: an expression specifying the value of the attribute upon initialization.
  - multiplicty: the possible number of data values for the attribute that may be held by an instance.
  - ordering: specifies whether the set of values is ordered. This property is only relevant if the multiplicty is greater than one. Possibilities are *unordered*, *ordered*, and user-defined values (such as *sorted*).
  - ownerScope: specifies whether the attribute appears in each instance of the classifier[2] or whether there is just a single instance of the attribute for the entire classifier. Valid values for this property are *instance* and *classifier*.
  - property-string: indicates property values that apply to the attribute.
  - stereotype: in case the attribute is stereotyped, it indicates the name of the stereotype.
  - type: designates the classifier whose instances are values of the attribute. Possible values are a *Class*, an *Interface*, or a *DataType* (*Date*, *Long*, *String*, etc.).
  - visibility: specifies whether the attribute can be used by other Classifiers. Possible values are *public*, *protected*, *private*, and *package*.

In an attribute/class diagram, the involved container class is imported from its respective package and linked to its attribute classes

[2]The abstract notion of a type of an object is a classifier (Class, Interface, etc.), and the specific, concrete objects themselves are instances of these types.

through instances of the «Contain» stereotype. A «Contain» relationship is formally defined as follows:

Definition. *A contain relationship is a composite aggregation between a container class and its corresponding attribute classes, originated at the end near the container class and highlighted with the «Contain» stereotype*[3].

Once having defined «Attribute» classes and «Contain» relationships we are ready to define attribute/class diagrams.

Definition. *An attribute/class diagram is a regular **UML** class diagram extended with «Attribute» classes and «Contain» relationships.*

Coming back to our motivating example (Figure 7.2), on the left hand side of Figure 7.5, the traditional **UML** representation of a class called Orders that contains four attributes (order_id, cust_id, date, and prod_list) is shown. On the right hand side of the same figure, the same class is represented by means of our approach: the container class has been imported from another package (the legend (from OLTP system) indicates the package where it has been defined firstly, see Figure 7.1); with respect to the naming convention, the names of the four attribute classes follow our naming convention (container class name + "." + attribute name); the attribute classes are labeled with the «Attribute» stereotype; finally, the container class and the attribute classes are related by a «Contain» composite aggregation. In order to avoid a complex diagram, we assume some default values for the tag definitions of the attribute classes (1 for multiplicity, public for visibility, etc.); therefore, the type is the only tag definition shown in this diagram.

7.4 The Data Mapping Diagram

Once we have introduced the extension mechanism that enables **UML** to treat attributes as first-class citizens, we can proceed in defining a framework on its usage. In this section, we will introduce the *data mapping diagram*, which is a new kind of diagram, particularly customized for the tracing of the data flow, at various degrees of detail, in a **DW** environment. Data mapping diagrams are complementary to the typical class and interaction diagrams of **UML** and focus on

[3]In the composite aggregation, the part is strongly owned by the composite and may not be part of any other composite. This means that the composite object is responsible for the creation and destruction of the parts.

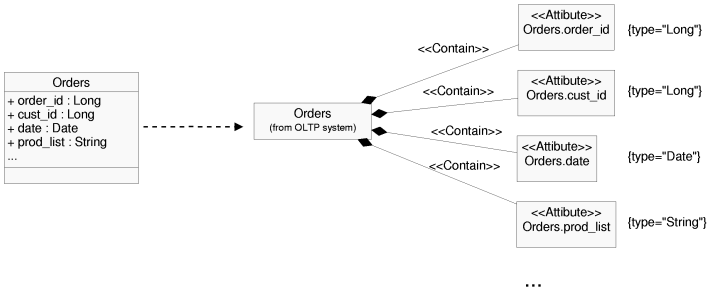


Figure 7.5: Attributes represented as first-class modeling elements

the particularities of the data flow and the interconnections of the involved data stores. A special characteristic of data mapping diagrams is that a certain **DW** scenario is practically described by a set of complementary data mapping diagrams, each defined at a different level of detail. In this section, we will introduce a principled approach to deal with such complementary data mapping diagrams.

To capture the interconnections between design elements, in terms of data, we employ the notion of *mapping*. Broadly speaking, when two design elements (e.g., two tables, or two attributes) share the same piece of information, possibly through some kind of filtering or transformation, this constitutes a semantic relationship between them. In the data warehousing context, this relationship, involves three logical parties: (a) the *provider* entity (schema, table, or attribute), responsible for generating the data to be further propagated, (b) the *consumer*, that receives the data from the provider and (c) their intermediate *matching* that involves the way the mapping is done, along with any transformation and filtering.

Mappings can be defined in all levels of granularity, i.e., at the schema, table or attribute level. A mapping consists of connections between instances of two models that belong to the same or different model schemas. Most of the time, mappings are defined among entities of the same granularity (i.e., attributes are mapped to attributes, tables to tables, etc). In a **DW** setting, a data mapping diagram typically relates tables/attributes at the data sources (the SCS) and the **DW** (the DWCS), and the **DW** (the DWCS) and the clients' structures (CCS).

As explained in [14], there are different levels of detail at which to specify mappings: at one extreme, a mapping could specify the full semantic relationships between the two models; at the other extreme, a mapping could be structural, specifying only the objects in the two models that are related to each other, without any mapping

semantics. Our approach allows the **DW** designer to address different possibilities between both extremes.

Since a data mapping diagram can be very complex, our approach offers the possibility to organize it in different levels thanks to the use of **UML** packages. Our layered proposal consists of four levels (see Figure 7.6), with one data mapping diagram per level:

Database Level (or **Level 0**). In this level, each schema of the **DW** environment (e.g., data sources at the conceptual level in the SCS, conceptual schema of the **DW** in the DWCS, etc.) is represented as a package [73]. The mappings among the different schemata are modeled in a single mapping package, encapsulating all the lower-level mappings among different schemata.

Dataflow Level (or **Level 1**). This level describes the data relationship among the individual source tables of the involved schemata towards the respective targets in the **DW**. Practically, a data mapping diagram at the database level is zoomed-in to a set of more detailed data mapping diagrams, each capturing how a target table is related to source tables in terms of data.

Table Level (or **Level 2**).Whereas the mapping diagram of the dataflow level describes the data relationships among sources and targets using a single package, the data mapping diagram at the table level, details all the intermediate transformations and checks that take place during this flow. Practically, if a data mapping is simple, a single package that represents the data mapping can be used at this level; otherwise, a set of packages is used to segment complex data mappings in sequential steps.

Attribute Level (or **Level 3**). In this level, the data mapping diagram involves the capturing of inter-attribute mappings. Practically, this means that the diagram of the table is zoomed-in and the mapping of provider to consumer attributes is traced, along with any intermediate transformation and cleaning. As we shall describe later, we provide two variants for this level.

At the leftmost part of Figure 7.6, a simple relationship among the DWCS and the SCS exists: this is captured by a single Data Mapping package and these three design elements constitute the data mapping diagram of the database level (or Level 0). Assuming that there are three particular tables in the **DW** that we would like to populate, this particular Data Mapping package abstracts the fact that there are three main scenarios for the population of the **DW**, one for each of this tables. In the dataflow level (or Level 1) of our

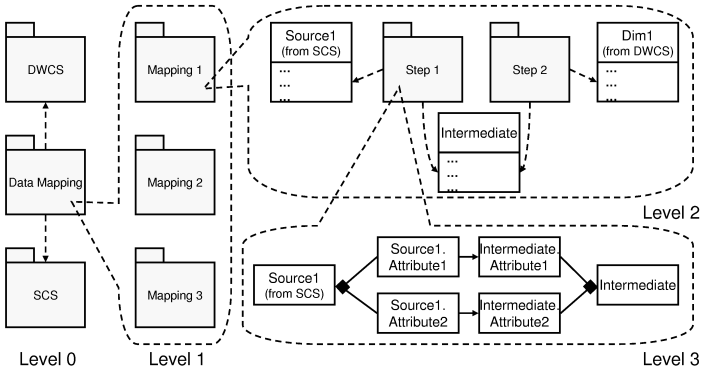


Figure 7.6: Data mapping levels

framework, the data relationships among the sources and the targets in the context of each of the scenarios, is practically modeled by the respective package. If we zoom in one of these scenarios, e.g., Mapping 1, we can observe its particularities in terms of data transformation and cleaning: the data of Source 1 are transformed in two steps (i.e., they have undergone two different transformations), as shown in Figure 7.6. Observe also that there is an Intermediate data store employed, to hold the output of the first transformation (Step 1), before passed on to the second one (Step 2). Finally, at the right lower part of Figure 7.6, the way the attributes are mapped to each other for the data stores Source 1 and Intermediate is depicted. Let us point out that in case we are modeling a complex and huge **DW**, the attribute transformation modelled at level 3 is hidden within a package definition, thereby avoiding the use of cluttered diagrams.

The constructs that we employ for the data mapping diagrams at different levels are as follows:

- The database and dataflow diagrams (Levels 0 and 1) use traditional **UML** structures for their purpose. Specifically, in these diagrams we employ (a) packages for the modeling of data relationships and (b) simple dependencies among the involved entities. The dependencies state that the mapping packages are dependent upon the changes of the employed data stores.

- The table level (Level 2) diagram extends **UML** with three stereotypes: (a) «Mapping», used as a package that encapsulates the data interrelationships among data stores, (b) «Input» and «Output» which explain the roles of providers and consumers for the «Mapping».

- The diagram at the attribute level (Level 3) is also using several newly introduced stereotypes, namely «Map», «MapObj», «Domain», «Range», «Input», «Output», and «Intermediate» for the definition of data mappings.

We will detail the stereotypes of the table level in the next section and defer the discussion for the stereotypes of the attribute level to Section 7.4.2.

7.4.1 The Data Mapping Diagram at the Table Level: Segmenting Data Diagrams

During the integration process from data sources into the **DW**, source data may undergo a series of transformations, which may vary from simple algebraic operations or aggregations to complex procedures. In our approach, the designer can segment a long and complex transformation process into simple and small parts represented by means of **UML** packages that are materialization of a «Mapping» stereotype and contain an attribute/class diagram. Moreover, «Mapping» packages are linked by «Input» and «Output» dependencies that represent the flow of data. During this process, the designer can create *intermediate classes*, represented by the «Intermediate» stereotype, in order to simplify or clarify the models. These classes represent intermediate storage that may or may not exist actually, but they help to understand the mappings.

In Figure 7.7, a schematic representation of a data mapping diagram at the table level is shown. This level specifies data sources and target sources, to which these data are directed. At this level, the classes are represented as usually in **UML** with the attributes depicted inside the container class. Since all the classes are imported from other packages, the legend (from ...) appears below the name of each class. The mapping diagram is shown as a package decorated with the «Mapping» stereotype and hides the complexity of the mapping, because a vast number of attributes can be involved in a data mapping. This package presents two kinds of stereotyped dependencies: «Input» to the data providers (i.e., the data sources) and «Output» to the data consumers (i.e., the tables of the **DW**).

7.4.2 The Data Mapping Diagram at the Attribute Level: Integration in Detail

As already mentioned, in the attribute level, the diagram includes the relationships between the attributes of the classes involved in a data mapping. At this level, we offer two design variants:

- Compact variant: the relationship between the attributes is

For more information about *UML notes*, consult appendix B, pp. 211.

represented as an *association*, and the semantic of the mapping is described in a **UML** *note* attached to the target attribute of the mapping.

- Formal variant: the relationship between the attributes is represented by means of a *mapping object*, and the semantic of the mapping is described in a *tag definition* of the mapping object.

For more information about *UML tag definitions*, consult appendix C, pp. 217.

With the first variant, the data mapping diagrams are less cluttered, with less modeling elements, but the data mapping semantics are expressed as **UML** notes that are simple comments that have no semantic impact. On the other hand, the size of the data mapping diagrams obtained with the second variant is larger, with more modeling elements and relationships, but the semantics are better defined as tag definitions.

Compact Variant

In this variant, the relationship between the attributes is represented as an association decorated with the stereotype «Map», and the semantic of the mapping is described in a **UML** note attached to the target attribute of the mapping.

The content of the package Mapping diagram from Figure 7.7 is defined in the following way (recall that Mapping diagram is a «Mapping» package that contains an attribute/class diagram):

- The classes DS1, DS2, ..., and Dim1 are imported in Mapping diagram.
- The attributes of these classes are suppressed because they are shown as «Attribute» classes in this package.
- The «Attribute» classes are connected by means of association relationships and we use the navigability property to specify the flow of data from the data sources to the **DW**.
- The association relationships are adorned with the stereotype «Map» to highlight the meaning of this relationship.
- A **UML** note can be attached to each one of the target attributes to specify how the target attribute is obtained from the source attributes. The language for the expression is a choice of the designer (e.g., a LAV vs. a GAV approach [69] can be equally followed).

For more information about *UML navigability*, consult appendix B, pp. 211.

In Figure 7.8, we show a data mapping diagram according to this variant. The classes shown in Figure 7.7 have been imported and their attributes are shown as «Attribute» classes. Different «Map»

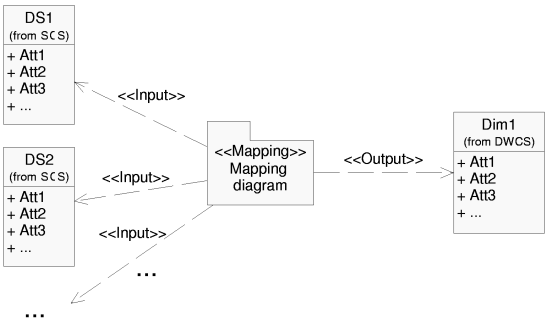


Figure 7.7: Level 2 of a data mapping diagram

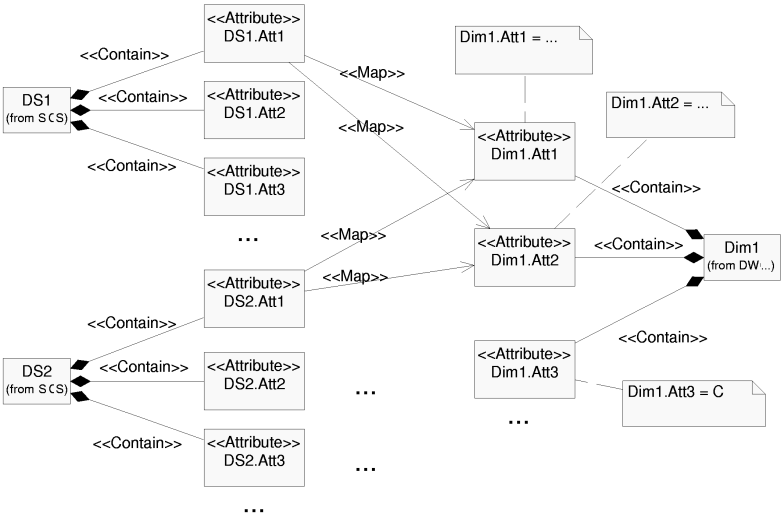


Figure 7.8: Level 3 of a data mapping diagram (compact variant)

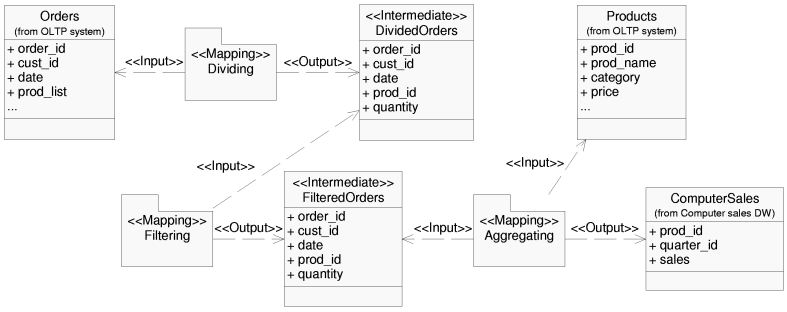


Figure 7.9: Level 2 of a data mapping diagram

associations have been defined and the semantic of the mappings is described in **UML** notes. In this example, Dim1.Att1 and Dim1.Att2 depend on different source attributes and the notes attached to them define the semantic of the mapping. However, Dim1.Att3 does not depend on the data sources and it always takes the same value (a constant named "C"), but a function that generates a timestamp, a random number or other kinds of values can also be specified instead of a constant.

7.4.3 Motivating Example Revisited

From the **DW** example shown in Figures 7.1, 7.2, and 7.3, we define the corresponding data mapping diagram shown in Figure 7.9. The goal of this data mapping is to calculate the quarterly sales of the products belonging to the computer category. The result of this transformation is stored in ComputerSales from the DWCS. The transformation process has been segmented in three parts: Dividing, Filtering, and Aggregating; moreover, DividedOrders and FilteredOrders, two «Intermediate» classes, have been defined.

Following with the data mapping example shown in Figure 7.9, attribute prod_list from Orders table contains the list of ordered products with product ID and (parenthesized) quantity for each. Therefore, Dividing splits each input order according to its prod_list into multiple orders, each with a single ordered product (prod_id) and quantity (quantity), as shown in Figure 7.10. Note that in a data mapping diagram the designer does not specify the processes, but only the data relationships. We use the one-to-many cardinality in the association relationships between Orders.prod_list and DividedOrders.prod_id and DividedOrders.quantity to indicate that one input order produces multiple output orders. We do not attach any note in this diagram because the data are not transformed, so the mapping

is direct.

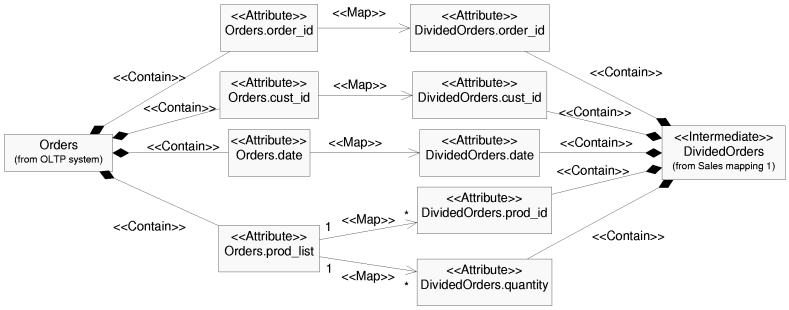


Figure 7.10: Dividing Mapping

Filtering (Figure 7.11) filters out products not belonging to the computer category. We indicate this action with a **UML** note attached to the prod_id mapping, because it is supposed that this attribute is going to be used in the filtering process.

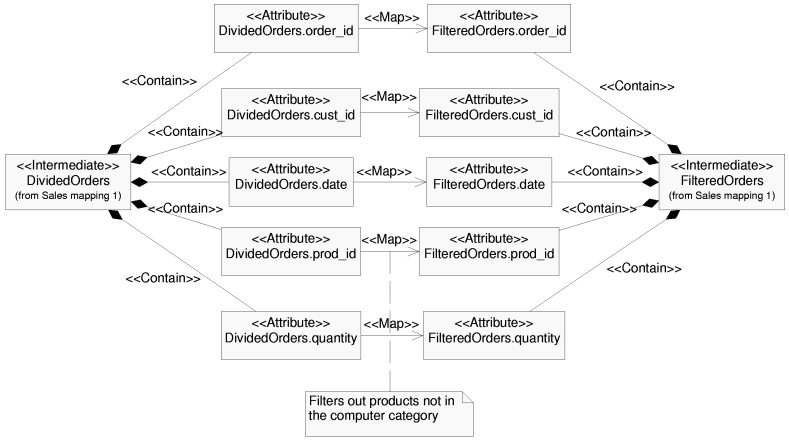


Figure 7.11: Filtering Mapping

Finally, Aggregating (Figure 7.12) computes the quarterly sales for each product. We use the many-to-one cardinality to indicate that many input items are needed to calculate a single output item. Moreover, a **UML** note indicates how the ComputerSales.sales attribute is calculated from FilteredOrders.quantity and Products.price. The cardinality of the association relationship between Products.price and ComputerSales.sales is one-to-many because the same price is used

in different quarters, but to calculate the total sales of a particular product in a quarter we only need one price (we consider that the price of a product never changes along time).

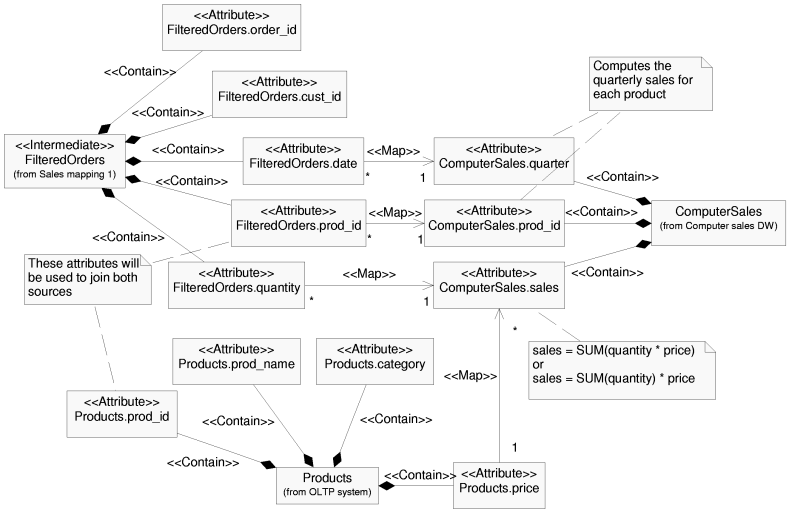


Figure 7.12: Aggregating Mapping

At this point we would like to come back to our original statements at the introductory section and discuss briefly our gains by adopting attribute-level modeling.

- We can easily detect inconsistencies, either though some computational engine or even by simple observation of the diagram. For example, if the attributes of a **DW** table are not populated, then, an inconsistency occurs.

- We can treat the design artifact as a graph [135], where provider and consumer relationships are treated as incoming and outgoing edges. In this sense, we can even *measure* the properties of our modeling in a straightforward fashion. For example, we can measure the aforementioned inconsistencies or we can even highlight hot-spots in our design: for example, in Figure 7.10 we can observe that the attribute Orders.prod_list is responsible for populating more than one target attribute in the **DW**.

- Both the visualization and the measurement of the design properties can significantly aid the impact analysis that needs to be performed in the presence of changes in the design. As an example, assume the case where a source attribute is to be deleted, or

the definition of a primary key to be altered. Our data mapping diagrams can easily depict and measure the affected attributes. Hot-spots are really important, in that sense.

7.5 Conclusions

In this chapter, we have presented a framework for the design of the **DW** back-stage (and the respective **ETL** processes) based on the key observation that this task fundamentally involves dealing with the specificities of information at very low levels of granularity. Specifically, we have presented a disciplined framework for the modeling of the relationships between sources and targets in different levels of granularity (i.e., from coarse mappings at the database level to detailed inter-attribute mappings at the attribute level). Unfortunately, standard modeling languages like the **ER** model or **UML** are fundamentally handicapped in treating low granule entities (i.e., attributes) as first class modeling elements. Therefore, in order to formally accomplish the aforementioned goal, we have extended **UML** to model attributes as first-class citizens. In our attempt to provide complementary views of the design artifacts in different levels of detail, we have based our framework on a principled approach in the usage of **UML** packages, to allow zooming in and out the design of a scenario.

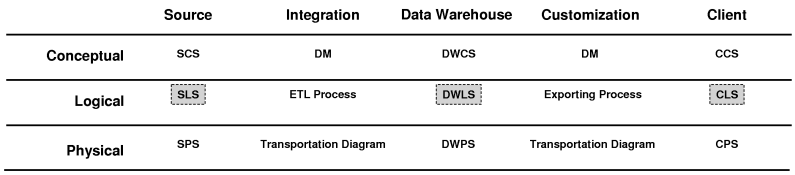
Although we have developed the representation of attributes as first-class modeling elements in **UML** in the context of data warehousing, we believe that our solution can be applied in other application domains as well, e.g., definition of indexes and materialized views in databases, modeling of **XML** documents, specification of web services, etc.

Part II

Logical Level

Chapter 8

Logical Modeling of Data Sources and Data Warehouses



In this chapter, we address the design of the Source Logical Schema, the Data Warehouse Logical Schema, and the Client Logical Schema. These diagrams can be defined independently, or they can be derived from the corresponding conceptual models (Source Conceptual Schema, Data Warehouse Conceptual Schema, and Client Conceptual Schema). We use the *UML Profile for Database Design* to model these diagrams that define the database structures.

Contents

8.1 Introduction

In the previous chapters, we have tackled the conceptual modeling of the data sources and the **DW** itself. In this chapter, the modeling effort transitions from the conceptual analysis to the logical design of the database. In the following, we will focus on relational databases [26], the most popular **DBMS** nowadays, and we will leave the study of other **DBMS** for the future.

The **UML** offers some advantages for the logical database design that are not generally considered in traditional notations. For example, the **UML** provides full support for modeling generalization and specialization relationships or stored procedures. Moreover, the **UML** provides the concept of packages, which logically group the elements of a model in different units.

To achieve the logical modeling of the data sources (Source Logical Schema), the **DW** (Data Warehouse Logical Schema), and the structures used by the final users (Client Logical Schema) we apply the *UML Profile for Database Design* [90].

There are two basic directions on where to go next. One direction is to build the logical models from the conceptual models by means of a mapping between the different diagrams. The other direction is to build the logical models independently from the conceptual models; however, we advise against this last direction because the advantages of starting from the conceptual level and maintaining a coherent mapping between the two levels are lost. Therefore, we recommend to build the logical models based on the conceptual models.

There are multiple ways to map models. In our approach, the classes are mapped to tables, attributes to columns, types to datatypes, and associations to relationships. In this mapping process, some situations have to be considered: not all elements in each model will be mapped, e.g., some attributes from the conceptual model may not be represented in the logical model because they are not stored in the database. For example, an attribute called Total_Sales, which represents the sum of multiple columns in the database is not stored because it is just a calculation in the application (it is a derived attribute).

The remainder of this chapter is structured as follows: Section 8.2 introduces the *UML Profile for Database Design*; Section 8.3 focuses on mapping classes to tables; then, Section 8.4 moves into mapping attributes of a class to columns of a table; and Section 8.5 discusses mapping types to datatypes. Finally, Section 8.6 points out some conclusions.

8.2 The UML Profile for Database Design

The *UML Profile for Database Design* [90] developed by Rational Software Corporation provides stereotypes and tagged values that represent tables, views, columns, relationships, triggers, etc. This profile also includes some icons to more easily visualize the database elements and rules to enforce about the creation of a relational database design.

The main diagram elements that this profile defines are described below and their associated graphical representations are shown in Figure 8.1:

- **Table**: a grouping of information in a database about the same subject, made up to columns.
- **View**: a virtual table that, from the user's perspective, behaves exactly like a typical table but has no independent existence of its own.
- **Domain**: the valid set of values for an attribute or column.
- **Column**: a component of a table that holds a single attribute of the table.
- **Primary key**: the candidate key that is chosen to identify rows in a table.
- **Foreign key**: a column or set of columns within a table that map to the primary key of another table.
- **Identifying relationship**: a relationship between two tables in which the child table must coexist with the parent table.
- **Non-identifying relationship**: a relationship between two tables in which each table can exist independently of the other.

In addition to the elements shown in Figure 8.1, the *UML Profile for Database Design* defines more stereotypes, such as «Database», «Schema», «Tablespace», «Index», «Check», etc. The common presentation of a stereotype is to use the standard symbol for the base element but to place the name of the stereotype above the name of the element.

Stereotype notation: see UML (3.18.2, 3-31).

However, in order to permit limited graphical extension of the **UML** notation as well, a graphic icon can be associated with a stereotype . The icon can be used in one of two ways:

- It may be used instead of, or in addition to, the stereotype keyword string as part of the symbol for the base model element that the stereotype is based on.

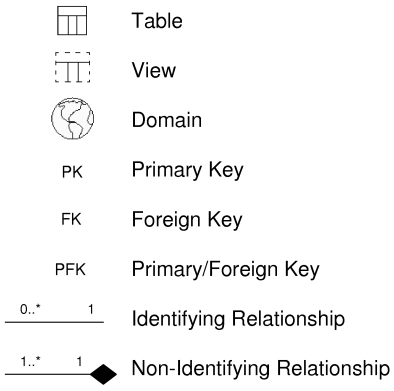


Figure 8.1: Diagram elements and their appropiate icons

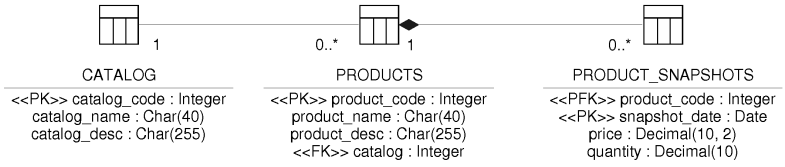


Figure 8.2: Stereotype display: Icon

- The entire base model element symbol may be "collapsed" into an icon containing the element name or with the name above or below the icon.

For example, in Figure 8.2, Figure 8.3, and Figure 8.4 we illustrate various notational forms of the stereotype notation. The three figures are alternatives of each other. In Figure 8.2, the icon of the stereotype is used instead of the symbol of the base element. However, in Figure 8.3, the icon is placed in the upper right corner of the class compartment. Finally, in Figure 8.4, the stereotype is placed above the name of the class being described.

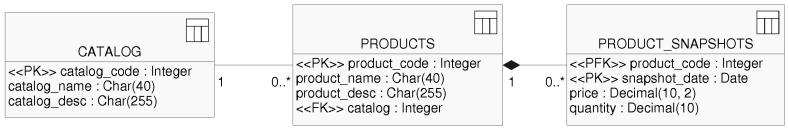


Figure 8.3: Stereotype display: Decoration

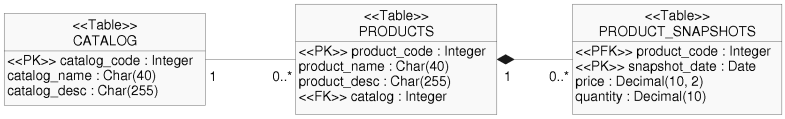


Figure 8.4: Stereotype display: Label

8.3 Mapping Classes to Tables

There are four basic ways to map classes to tables: one-to-one, one-to-many, many-to-one, and many-to-many. There are different reasons for selecting the right mapping, but there exist some mappings that occur based on general relational database approaches. In the following, we are going to study two of the most common mappings: many-to-many associations and inheritance hierarchies (supertype/subtype relationships).

8.3.1 Many-to-many Associations

Many-to-many associations must be broken into two one-to-many relationships by creating an auxiliary table. This table contains foreign keys to the two tables that the classes of the many-to-many association map to. Moreover, the auxiliary table may have additional columns.

For example, on the left hand side of Figure 8.5 we represent the conceptual model of a data source. This data source stores the sales of a company and a sale comprises some products and a product can appear in different sales. Therefore, there is a many-to-many association between Sales and Products. On the right hand side, we show the mapping of the conceptual model to the logical model. In this mapping, one table per class has been created and a new table called SalesLines corresponds to the many-to-many association. This table contains two primary keys that are also foreign keys: idsale to Sales table and codeproduct to Products.

8.3.2 Inheritance Hierarchy

When mapping inheritance hierarchies to tables, there are three basic choices: one table per class, one table per concrete class, and one table per hierarchy.

In *one table per class*, each class is mapped directly to a corresponding table. *One table per concrete class* is also known as "rolling down" the supertype table into its subtypes: the attributes from the superclass are placed as columns in tables that map to the subtype

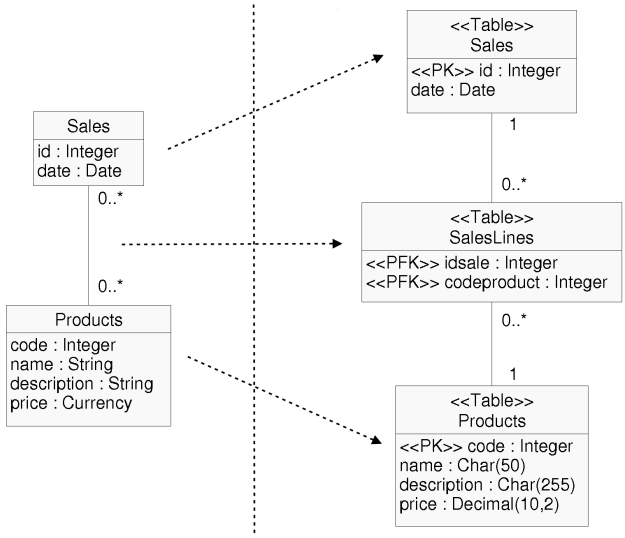


Figure 8.5: Transforming a many-to-many association

classes. Finally, *one table per hierarchy* is also known as "rolling up" the subtypes to the supertype: the attributes in the subtype classes are placed as column in a single table that maps both the supertype and subtypes.

For example, in Figure 8.6 we show a conceptual data model that represents the products that a company sales. There three types of products: TV, Radio, and Video_player, and each type has its own attributes. Therefore, an inheritance hierarchy has been modelled, with a supertype class and three subtype classes.

In Figure 8.7, we show a one table per class mapping. The transformation is direct, and the only outstanding situation is the transformation of the generalization/specialization relationships into common associations. These associations represent one-to-one relationships between the Products table and the subtype tables.

In Figure 8.8, we show a one table per concrete class mapping. Three tables that map to the subtype classes have been defined. The attributes from the supertype class have been replicated in every table.

Finally, in Figure 8.9, we show a one table per hierarchy mapping. The whole inheritance hierarchy is represented as a single table. This table maps to both the supertype and subtypes and it contains the attributes from all the classes of the hierarchy. Moreover, there is a new column created (product_type) that does not map to any attribute in the classes of the hierarchy. This column defines the type

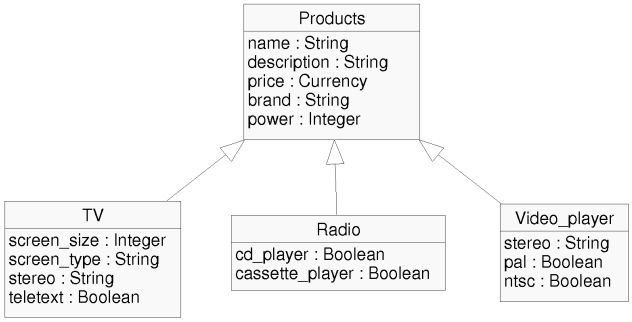


Figure 8.6: A conceptual data model with a inheritance hierarchy

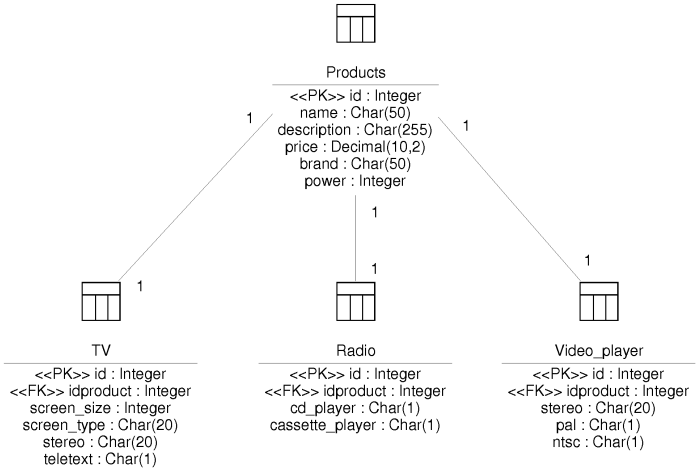


Figure 8.7: Transforming a inheritance hierarchy: one table per class

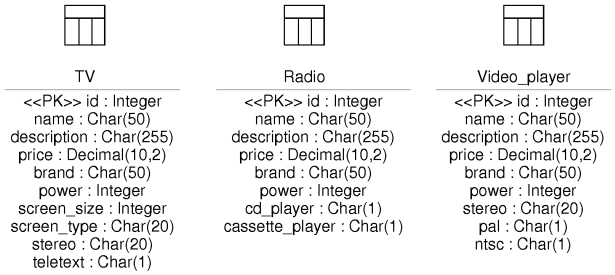


Figure 8.8: Transforming a inheritance hierarchy: one table per concrete class

of Product. In a **UML** note we point out the three valid values of this column.

8.4 Mapping Attributes to Columns

There are many ways to map attributes to columns. But they are affected by the class-to-table mapping. Therefore, both must be considered together. Moreover, some attributes will not become columns and some columns will be defined for the correct operation of the database.

When mapping the attributes to columns, some considerations must be taken into account:

- Database performance.
- Secure access: in most databases, it is not possible to assign security access to an individual columns but only to the entire table. Therefore, in order to secure some columns, a new table can be created only to store the secure columns.
- Derived attributes: derived attributes usually do not exist in the database, but performance requirements may force to store derived attributes.

8.5 Mapping Types to Datatypes

Conceptual models generally contain attributes that have generic types. These types are well enough descriptive but not specific to any implementation.

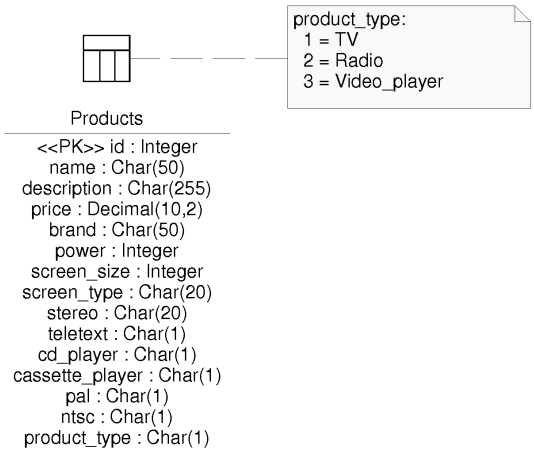


Figure 8.9: Transforming a inheritance hierarchy: one table per hierarchy

Generic type	**Description**
Boolean	Used to represent the logical values of True or False
Currency	Used to declare variables capable of holding fixed-point numbers with 15 digits to the left of the decimal point and 4 digits to the right
Date	Used to hold date and time values
Double	Used to declare variables capable of holding real numbers with 15-16 digits of precision
Integer	Used to declare whole numbers with up to 4 digits of precision
Long	Used to hold numbers with up to 10 digits of precision
Single	Used to declare variables capable of holding real numbers with up to 7 digits of precision
String	Used to hold an unlimited number of characters

Table 8.1: Generic types and their description

Generic type	Datatype
Boolean	Bit
Currency	Decimal
Date	Date
Double	Double Precision
Integer	Integer
Long	Decimal
Single	Decimal
String	Char

Table 8.2: Generic types mapped to ANSI SQL datatypes

In Table 8.1, we show some of the generic types that are commonly used. These generic types are available in most **CASE** tools.

In Table 8.2, we show a mapping of the generic types shown in Table 8.1 to specific types defined in ***Structured Query Language* (SQL)** [11]. These table is only an example, because designers should define their own mappings for every project. For example, Integer may be also mapped to Numeric, Decimal, or Smallint.

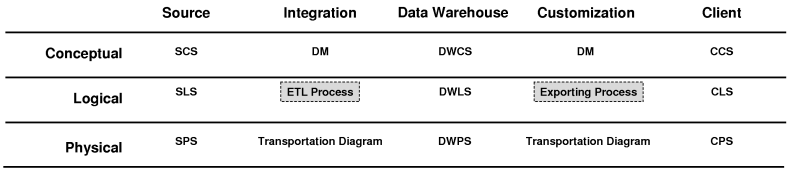
The previous figures, we have included some mapping from attribute types to column datatypes. For example, in Figure 8.5, the String type is transformed into Char(50) and Char(255), whereas the Currency type is turned into Decimal(10,2). In Figure 8.6 and Figure 8.7, the Boolean type is turned into Char(1).

8.6 Conclusions

In this chapter, we have explained how we tackle the logical modeling of data sources and **DW**. Our approach in based on the *UML Profile for Database Design* [90], created by Rational Software Corporation for use when designing a database. From the conceptual model of the data sources and the **DW**, we have shown how to proceed to the logical model applying some mappings.

Chapter 9

Modeling ETL Processes in Data Warehouses



ETL processes are responsible for the *extraction* of data from heterogeneous operational data sources, their *transformation* (conversion, cleaning, normalization, etc.) and their *loading* into **DW**. **ETL** processes are a key component of **DW** because incorrect or misleading data will produce wrong business decisions, and therefore, a correct design of these processes at early stages of a **DW** project is absolutely necessary to improve data quality. However, not much research has dealt with the modeling of **ETL** processes. In this chapter, we present our approach that allows us to accomplish the modeling of these **ETL** processes together with the logical schema of the target **DW** in an integrated manner. We provide the necessary mechanisms for an easy and quick specification of the common operations defined in these **ETL** processes such as, the integration of different data sources, the transformation between source and target attributes, the generation of surrogate keys and so on. Moreover, our approach allows the designer a comprehensive tracking and documentation of entire **ETL** processes, which enormously facilitates the maintenance of these processes. Another advantage of our proposal is the use of the **UML** (standardization, ease-of-use and functionality) and the seamless integration of the design of the **ETL** processes with the **DW** logical schema. Finally, we show how to use our integrated

approach by using a well-known modeling tool such as Rational Rose.

Contents

9.1 Introduction

Recalling Bill Inmon's **DW** definition [57], "*A data warehouse is a subject-oriented, integrated, time-variant, nonvolatile collection of data in support of management's decisions*". A **DW** is "integrated" because data are gathered into the **DW** from a variety of sources and merged into a coherent whole. **ETL** processes are responsible for the extraction of data from heterogeneous operational data sources, their transformation (conversion, cleaning, normalization, etc.) and their loading into **DW**.

For more information about Bill Inmon's *data warehouse definition*, consult Section 2.1, pp. 13.

It is highly recognized that the design and maintenance of these **ETL** processes is a key factor of success in **DW** projects for several reasons [123, 124]. **DW** are usually populated by data from different and heterogeneous operational data sources such as legacy systems, relational databases, COBOL files, Internet (**XML**, web logs) and so on. Therefore, different routines have to be defined and configured for accessing these heterogeneous systems and loading the correct data into the common **DW** repository.

Moreover, data from the operational systems are usually specified in different schemas and have to be extracted and transformed to collect them into a common **DW** repository [104]. Some of the more common technical tasks that have to be accomplished with these data are as follows. Data coming from different sources have to be joined into a unique target in the **DW**. Data usually have to be aggregated in order to facilitate the definition of the queries and improve the performance of the **DW**. Data are usually in different types and formats and they need to be converted into a common format. Data in the operational systems are usually managed by different primary keys representing for example, product or store codes and so on. In **DW** we usually use surrogate keys, and therefore, we need an efficient mechanism to assign surrogate keys to the operational data in the **DW** repository. Furthermore, as data are coming from different sources, we usually need to check the different primary and foreign keys to assure a high quality of data. Moreover, we also need a high number of filters to verify the right data to be uploaded in the **DW** and many more problems.

Due to the high difficulty in designing and managing these **ETL** processes, there has lately been a proliferation in the number of available **ETL** tools that try to simplify this task [45, 67]. During 2001, the **ETL** market grew to about $667 million [5]. Currently, companies expend more than thirty percent out of the total budget for **DW** projects in expensive **ETL** tools, but "*It's not unusual for the ETL effort to occupy 60 percent to 80 percent of a data warehouse's implementation effort*" [124]. Nevertheless, it is widely recognized that the design and maintenance of these **ETL** processes has not yet been

solved [5].

Therefore, we argue that a model and method is needed to help the design and maintenance of these **ETL** processes from the early stages of a **DW** project; as Kimball states, "*Our job as data warehouse designers is to star with existing sources of used data*" [63]. Unfortunately, little effort has been dedicated to propose a logical model that allows us to formally define these **ETL** processes.

In this chapter, we present a logical model based on the **UML** [97] for the design of **ETL** processes which deals with the more common technical problems above-presented. As the **UML** has been widely accepted as the standard for **OO** analysis and design, we believe that our approach will minimize the efforts of developers in learning new diagrams or methods for modeling **ETL** processes. Furthermore, the logical modeling of these **ETL** processes is totally integrated in our global approach. Therefore, our approach reduces the development time of a **DW**, facilitates managing data repositories, **DW** administration, and allows the designer to perform dependency analysis (i.e. to estimate the impact of a change in the data sources in the global **DW** schema).

The rest of this chapter is organized as follows. Section 9.2 provides an overview of **ETL** processes and their surrounding data quality problems. Section 9.3 describes in detail how to accomplish the logical modeling of **ETL** processes using our proposal. Then, Section 9.4 shows how some interesting **ETL** problems are solved applying our approach. Finally, Section 9.5 presents the main conclusions.

9.2 ETL

In an **ETL** process, the data extracted form a source system pass through a sequence of transformations before they are loaded into a **DW**. The repertoire of source systems that contribute data to a **DW** is likely to vary from standalone spreadsheets to mainframe-based systems many decades old. Complex transformations are usually implemented in procedural programs, either outside the database (in C, Java, Pascal, etc.) or inside the database (by using any 4GL). The design of an **ETL** process is usually composed of six tasks:

1. Select the sources for extraction: the data sources to be used in the **ETL** process are defined. It is very common in an **ETL** process to access different heterogeneous data sources.

2. Transform the sources: once the data have been extracted from the data sources, they can be transformed or new data can be derived. Some of the common tasks of this step are: filtering data, converting codes, performing table lookups, calculating

derived values, transforming between different data formats, automatic generation of sequence numbers (surrogate keys), etc.

3. Join the sources: different sources can be joined in order to load together the data in a unique target.

4. Select the target to load: the target (or targets) to be loaded is selected.

5. Map source attributes to target attributes: the attributes (fields) to be extracted from the data sources are mapped to the corresponding target attributes.

6. Load the data: the target is populated with the transformed data from the sources.

The transformation step of the **ETL** processes can also perform data cleaning tasks, although **ETL** tools typically have little built-in data cleaning capabilities. Data cleaning deals with detecting and removing errors and inconsistencies from data in order to improve the data quality [104]. Data quality problems are very significant: it has been estimated that poor quality customer data cost U.S. businesses $611 billion a year in postage, printing, and staff overhead [36].

The manual creation and maintenance of **ETL** processes increases the cost of development, deployment, running, and maintenance of a **DW**. That is why the logical modeling of **ETL** processes can be of a crucial help.

9.3 Modeling ETL processes

In this section we present our **ETL** modeling proposal that is integrated in our global **DW** modeling approach presented in Chapter 4. Our approach allows the designer to decompose a complex **ETL** process into a set of simple processes. This approach helps the designer to easily design and maintain **ETL** processes. Moreover, our approach allows the **DW** designer to tackle the design of **ETL** processes from different detail levels: (i) the designer can define a general overview of the process and let the database programmer to specify them or, (ii) the designer can provide a detailed description of each one of the attribute transformations.

Based on our personal experience, we have defined a reduced and yet highly powerful set of **ETL** mechanisms. We have decided to reduce the number of mechanisms in order to reduce the complexity of our proposal. We have summarized these mechanisms in Table 9.1. We consider that these mechanisms process data in the form

of records composed of attributes[1]. Therefore, we provide the «Wrapper» mechanism to transform any source into a record based source.

In our approach, an **ETL** process is composed of **UML** packages, which allow the user to decompose the design of an **ETL** process into different logical units. Every particular **ETL** mechanism is represented by means of a stereotyped class. Moreover, we have defined a different icon for each **ETL** mechanism (see Table 9.1). This icon can be used in a **UML** model instead of the standard representation of a class.

Dependency: see UML (2.5.2.15, 2-33), (3.51, 3-90).

The **ETL** mechanisms are related to each other by means of **UML** dependencies. A dependency in the **UML** is represented as a dashed line with an arrowhead. The model element at the tail of the arrow (the client) depends on the model element at the arrowhead (the supplier). A dependency states that the implementation or functioning of one or more elements requires the presence of one or more other elements. This implies that if the source is somehow modified, the dependents must be probably modified.

A **UML** note can be attached to every **ETL** mechanism to (i) explain the functioning of the mechanism and, (ii) define the mappings between source and target attributes of the **ETL** mechanisms. These mappings conform to the following syntax: target_attribute = source_attribute. To avoid overloading the diagram with long notes, when source and target attributes' names match, the corresponding mappings can be omitted. Furthermore, when some kind of ambiguity may exist (e.g., two attributes with the same name in different sources), the name of the source can be indicated together with name of the attribute (e.g., Customers.Name and Suppliers.Name).

We do not impose any restriction on the content of these notes, in order to allow the designer the greatest flexibility, but we highly recommend a particular content for each mechanism. The designer can use the notes to define **ETL** processes at the desired detail level. For example, the description can be general, specified by means of a natural language, or very detailed, specified by means of a programming language.

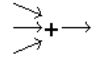
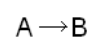
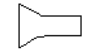
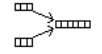
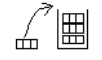
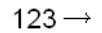
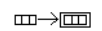
In the following, we provide a deeper description of each one of the **ETL** mechanisms presented in Table 9.1 together with the more appropriated contents for the corresponding attached notes.

9.3.1 Aggregation

The «Aggregation» mechanism aggregates data based on some criteria. This mechanism is useful to increase the aggregation level of a

[1] In our approach, the concept of attribute is similar to the concepts of column, property or field.

Table 9.1: ETL mechanisms and icons

ETL Mechanism (Stereotype)	**Description**	**Icon**
«Aggregation»	Aggregates data based on some criteria	
«Conversion»	Changes data type and format or derives new data from existing data	
«Filter»	Filters and verifies data	
«Incorrect»	Reroutes incorrect data	
«Join»	Joins two data sources related to each other with some attributes	
«Loader»	Loads data into the target of an ETL process	
«Log»	Logs activity of an ETL mechanism	
«Merge»	Integrates two or more data sources with compatible attributes	
«Surrogate»	Generates unique surrogate keys	
«Wrapper»	Transforms a native data source into a record based data source	

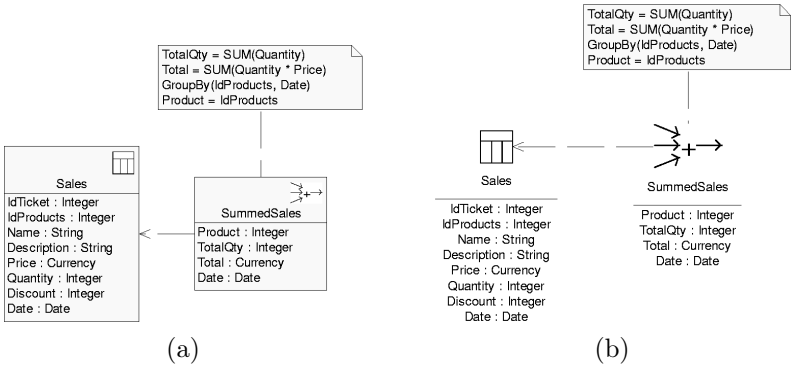


Figure 9.1: Aggregation example by using standard UML class notation and the defined stereotype icons

data source[2]. The designer can define the grouping criteria and the aggregation functions employed (SUM, AVG, MAX/MIN, COUNT, STDDEV, VARIANCE and so on) in the attached note to this mechanism.

For example, in Figure 9.1 (a), we have represented a portion of a loading process in a **DW**[3] by using standard **UML** notation in which the stereotype icons are placed in the upper right-hand corner of the corresponding class. It may also be observed that the icon used in the Sales class corresponds to the «Table» stereotype icon defined in the *UML Profile for Database Design* [90], which is used for the logical modeling of databases. As the grain of Sales is ticket line, we need the daily total sales in the **DW**. Therefore, Sales are grouped and summed up by product and date in SummedSales. We have decided to specify these aggregation tasks in the corresponding attached note. Figure 9.1 (b) represents the same **ETL** process using our **ETL** icons. From now on, we will use this representation throughout the rest of the chapter.

For more information about *logical modeling*, consult chapter 8, pp. 121.

9.3.2 Conversion

The «Conversion» mechanism is used to change data types and formats or to calculate and derive new data from existing data. The conversions are defined in the attached note by means of conversion functions applied to source attributes. The syntax of these conver-

[2]Partial summarization of data under different criteria is a very common technique used in **DW** to facilitate complex analysis. Summarization helps to reduce the size of the resulting **DW** and increase the query performance [63].

[3]From now on, partial examples are used to describe each **ETL** mechanism.

sions is target_attribute = Function(source_attributes), where Function is any kind of function that can be applied to source attributes.

Based on our experience, we have defined conversion functions for the most common situations. However, this is not a closed set as the designer can define their own user-defined functions for more particular and complex situations:

- Data type conversions: convert data from a data type into another data type. For example: Price = StringToCurrency(Price).
- Arithmetic conversions: perform arithmetic operations (add, multiply, etc.) with data. For example: Total = Quantity * Price.
- Format conversions: convert a value (currency, date, length, etc.) from one format into another one. For example: Price = DollarToEuro(Price).
- String conversions: transform strings (upper and lower-case, concatenate, replace, substring, etc.). For example: Name = Concatenate(FirstName, " ", Surname).
- Split conversions: break a value into different elements. For example, the following expression breaks a name ("John Doe") into first name ("John") and surname ("Doe"): FName = FirstName(Name); SName = Surname(Name).
- Standardization conversions: standardize attributes to contain identical values for equivalent data elements. We can use a set of rules or look-up tables to search for valid values. For example, the following expression substitutes "Jan." or "1" with "January": Month = StdMonth(Month).
- Value generator: generates a constant value or a variable value from a function. The new value does not depend on any source attribute. For example: Date = Timestamp().
- Default value: when a value is missing (null, empty string, etc.), it is possible to define a default value. The syntax of this option is target_attribute ?= value. For example: Type ?= "unknown".

Figure 9.2 presents an example in which different conversions are applied through the ConvertedCustomers stereotype. As it can be easily seen from the attached note, Name and Surname are concatenated; Address is split into street type, name and number; and Born is converted using a date format. Furthermore, all the activity is audited by CustomerLog (see next section).

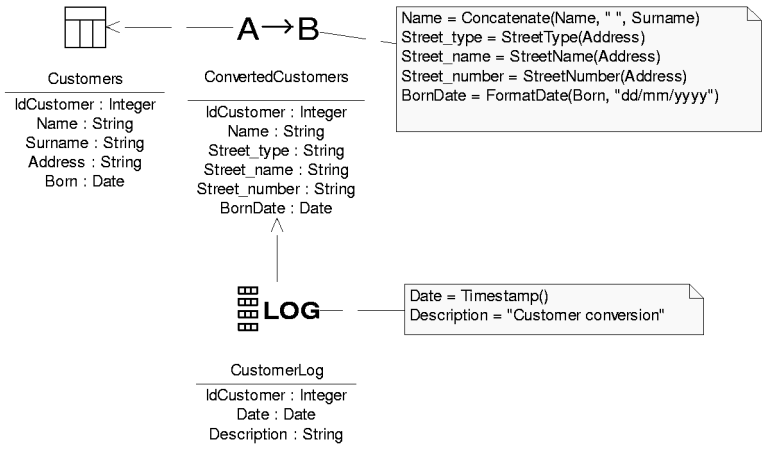


Figure 9.2: An example of Conversion and Log processes

9.3.3 Log

The «Log» mechanism can be connected to any **ETL** mechanism as it controls the activity of another **ETL** mechanism. This mechanism is useful to audit and produce operational reports for each transformation. The designer can add any kind of additional information in the note attached to this mechanism.

For example, in Figure 9.2, the activity of a «Conversion» mechanism is controlled by the «Log» mechanism called CustomerLog.

9.3.4 Filter

The «Filter» mechanism filters unwanted data and verifies the correctness of data based on constraints. In this way, this mechanism allows the designer to load only the required data or the data that meet an established quality level in the **DW**. The verification process is defined in the attached note by means of a set of Boolean expressions that must be satisfied. The Boolean expressions can be expressed by means of a set of rules or by means of look-up tables that contain the correct data. Some common tasks for this mechanism are checks for null values, missing values, values out of range, and so on. The data that do not satisfy the verification can be rerouted to an «Incorrect» mechanism (see Section 9.3.7).

For example, in Figure 9.3, Customers are filtered and only those that were born before 1950 are accepted for a subsequent processing.

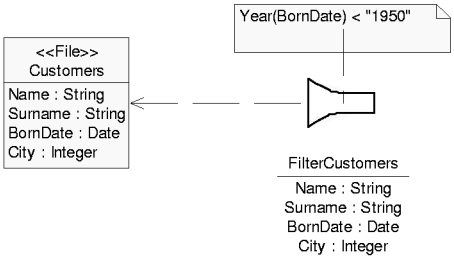


Figure 9.3: An example of Filter process

9.3.5 Join

The «Join» mechanism is used to join two data sources related to each other with some attributes (defined by means of a restrictive condition). The designer can define the following information in the attached note:

- The type of join: Join(conditional_expression), where Join can be InnerJoin (includes only the records that match the conditional expression), LeftJoin (includes all of the records from the first (left) of the two data sources, even if there are no matching values for records in the second (right) data source), RightJoin (includes all of the records from the second (right) of the two data sources, even if there are no matching values for records in the first (left) data source), and FullJoin (includes all of the records from both data sources, even if there are no matching values for records in the other data source).

- If LeftJoin, RightJoin or FullJoin are used, then the designer can define the values that substitute the non-existing values. The syntax of this option is target_attribute ?= value.

In Figure 9.4, we have represented an **ETL** process that joins three data sources. Due to the fact that «Join» can only be applied to two sources, and in this particular case we are dealing with three sources, two «Join» mechanisms are needed. In the CitiesStates join, a LeftJoin is performed to join cities' names and states' names. When it is not possible to join a city with a state (because Cities.State is missing or is incorrect), the corresponding state name is replaced by "unknown". Finally, in the CompleteCustomers join, the result of the previous join is joined with Customers.

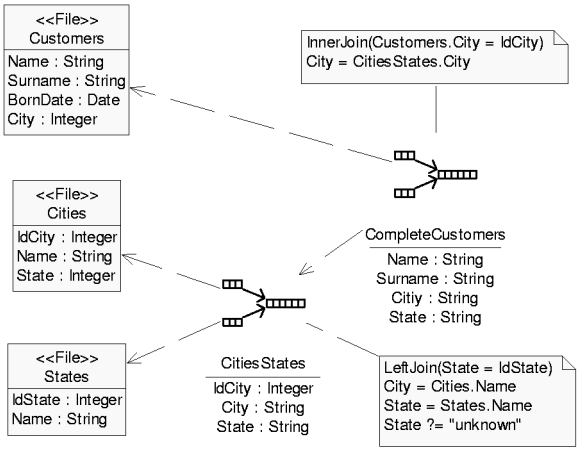


Figure 9.4: An example of Join process

9.3.6 Loader

The «Loader» mechanism loads data into the target of an **ETL** process such as a dimension or a fact in a **DW**. Every **ETL** process should have at least one «Loader» mechanism. Two operation modes are supported in the «Loader»:

- Free loading: the «Loader» mechanism does not verify any constraint as the target applies its own constraints to the new data.
- Constrained loading: the designer can apply primary and foreign key constraints during the loading process. Moreover, the designer can also define how to manage existing data in the target. The following information can be attached to the «Loader» mechanism:
  - PK(source_attributes): defines the attributes of the source data that define a unique record in the target. This information is used for constraining the loading process and it is also used for detecting the old data that should be updated.
  - FK(target_attributes; source_attributes): defines the attributes of the source data that should previously exist in a target.
  - Append: the target need not be empty before loading the new data; new data are loaded and old data are updated.
  - Delete: the target need be empty before loading the data.

- Insert: only new data are loaded in the target; old data are not loaded again or updated, although they have changed.
- Update: only existing data in the target are updated with the corresponding data, but new data are not loaded.

Append, Delete, Insert, and Update are mutually exclusive: only one of them can be used in a «Loader» mechanism.

For example, in Figure 9.5, CustomerLoader updates existing data in CustomersDim with data coming from Customers. Furthermore, due to the high probability of errors when making the loading process, those records that do not satisfy the constraints are rerouted to DiscardedCustomers. Customers and CustomersDim represent tables in a relational database; the icon corresponds to the «Table» stereotype defined in [90].

9.3.7 Incorrect

The «Incorrect» mechanism is used to reroute bad or discarded records and exceptions to a separate target. In this way, the **DW** designer can track different errors. This mechanism can only be used with the «Filter», «Loader», and «Wrapper», because these mechanisms constrain the data they process. The designer can add additional information in the note attached to this mechanism, such as a description of the error or a timestamp of the event.

For example, in Figure 9.5, the records that do not satisfy the constraints of CustomerLoader (primary key constraint on IdCustomer and only update existing data in the target) are rerouted to DiscardedCustomers, which collects the erroneous data and adds the Date attribute that is a timestamp of the event.

9.3.8 Merge

The «Merge» mechanism integrates two or more data sources with compatible attributes. Two data sources are compatible as long as both of them contain a subset of the attributes defined in the target: the attributes used in the integration must have the same names in all the data sources. If the attributes do not have the same names, the «Conversion» mechanism can be previously applied in order to standardize them. The attached note to this mechanism is used to define the mapping between the data sources and the target.

For example, in Figure 9.6, MergedCustomers is used to integrate data coming from a file and from a database table. Firstly, WrappedCustomers is used to transform a file into a record based source (see next section). Then, ConvertedCustomers changes the names of the attributes (CName and CSurname) and adds a new attribute (BornDate) with a default value.

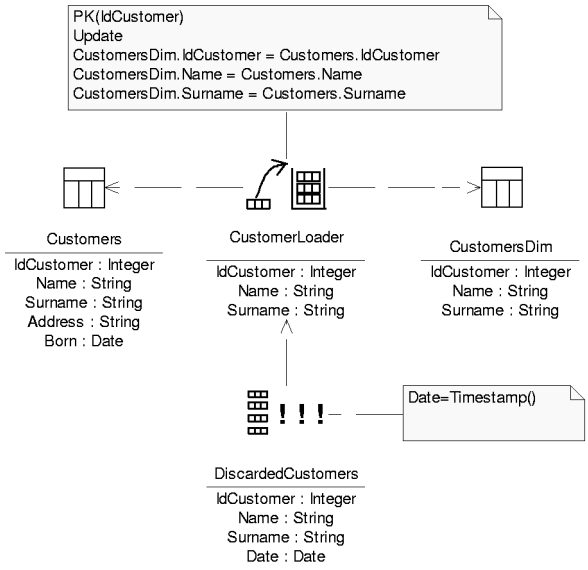


Figure 9.5: An example of Loader and Incorrect processes

9.3.9 Wrapper

The «Wrapper» mechanism allows us to define the required transformation from a native data source into a record based data source. Different native sources are possible in an **ETL** process: fixed and variable format sources, COBOL files (line sequential, record sequential, relative files, and indexed files), multiline sources, **XML** documents, and so on. The needed code to implement the «Wrapper» is not relevant as we are at the logical level, although the designer can define in the attached note all the information that considers relevant to help the programmer at the implementation phase.

In Figure 9.6, WrappedCustomers is used to transform data from a fixed format file. Some information about the format of the file is included in the attached note.

9.3.10 Surrogate

The «Surrogate» mechanism generates unique surrogate keys. In a **DW**, it is very important that primary keys of tables remain stable. Because of this, surrogate key assignment is a common process in **DW**, employed in order to replace the original keys of the data sources with a uniform key. The attached note to this mechanism is used to define the source attributes used to define the surrogate key.

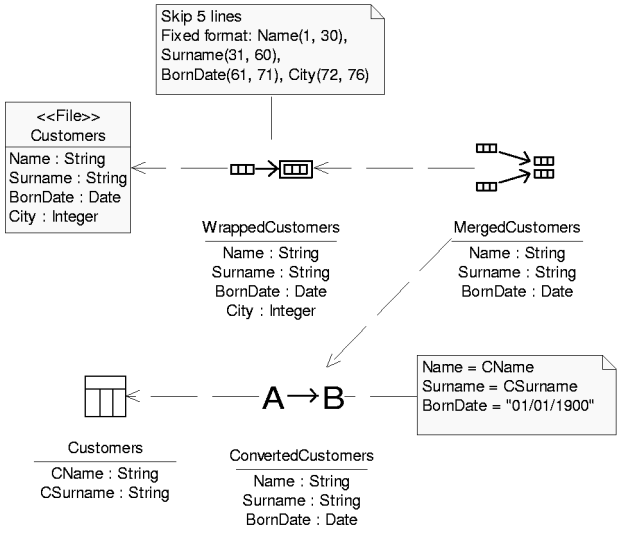


Figure 9.6: An example of Merge and Wrapper processes

Surrogate keys could have been defined in the «Conversion» mechanism, however, due to the importance that surrogate keys represent in **DW**, we have decided to define an own mechanism.

For example, in Figure 9.7, SurrogatedCities adds a surrogate key (IdCity) based on the attributes Name, State, and Country before loading the data into the **DW**.

9.4 ETL Examples

In previous sections, we have shown short and specific examples to explain how we apply the particular **ETL** mechanism aim of study. In this section, we present three interesting **ETL** examples that clarify our approach. In the first example, we show how to transform columns into rows. Then, in the second example, we show how to merge two different data sources and deal with a multi-target loading. Finally, the last example is an **ETL** scenario in which we show how to easily model different **ETL** mechanisms keeping a high grade of simplicity and yet very powerful **ETL** process model. These three examples show the expressiveness power of our **ETL** modeling approach.

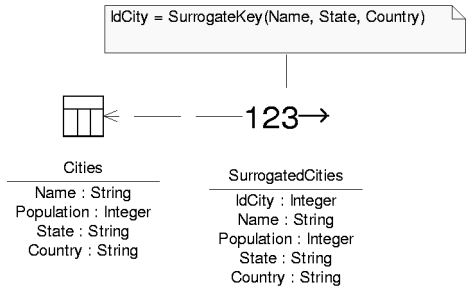


Figure 9.7: An example of Surrogate process

9.4.1 Transform Columns into Rows

In **RDBMS**, pivot (also called transpose) and unpivot are complementary data manipulation operators that modify the role of rows and columns in a relational table [31]:

- Pivot transforms a group of rows into a group of fewer rows with additional columns.
- Unpivot makes the inverse transformation, removing a number of columns and creating additional rows that contain the column names.

In this example, it is necessary to divide a data source that contains data about suppliers: name, surname, and different telephone numbers. In order to separate the telephone numbers, five «Conversion» mechanisms are defined. These mechanisms select the appropriate attributes in each case and add a new attribute (Type) that indicates the type of telephone number: 'Phone', 'Mobile' or 'Fax'. Then, SupplierPhones («Merge» mechanism) integrate all the telephone numbers of a supplier. Therefore, an unpivot operation is executed in this scenario.

9.4.2 Merging Two Different Data Sources and Multi-target Loading

The product list from two independent data source has to be loaded in two targets depending on the discount of each product.

Firstly, it is necessary to merge the two incompatible data sources: the sources have different attribute number and the common attributes have different names. For the first data source, ProductWrapper is used to transform a multiline file into a record based data source

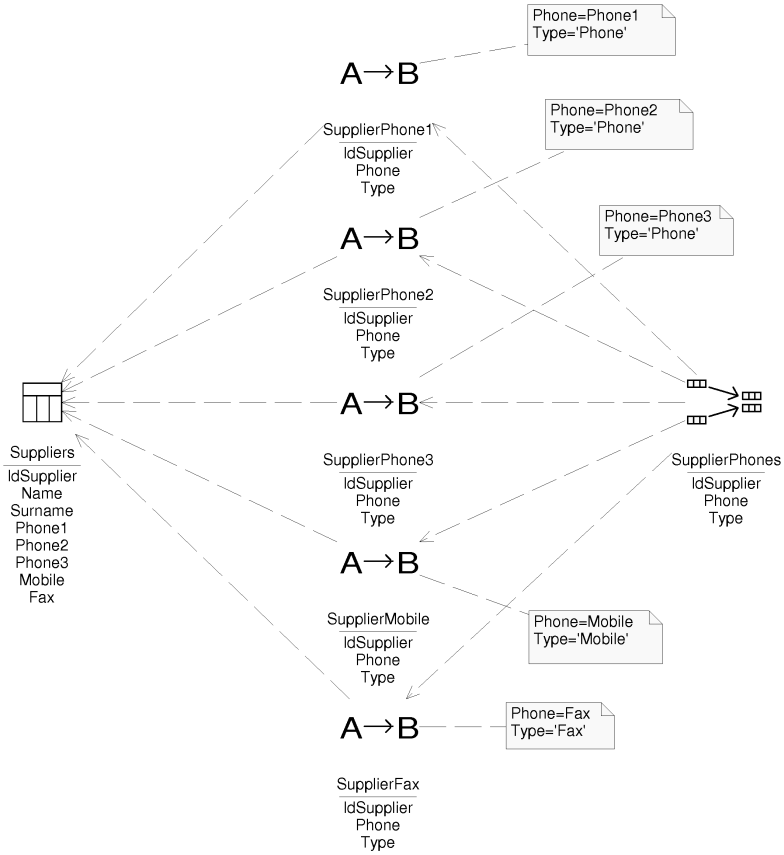


Figure 9.8: Transform columns into rows (unpivot)

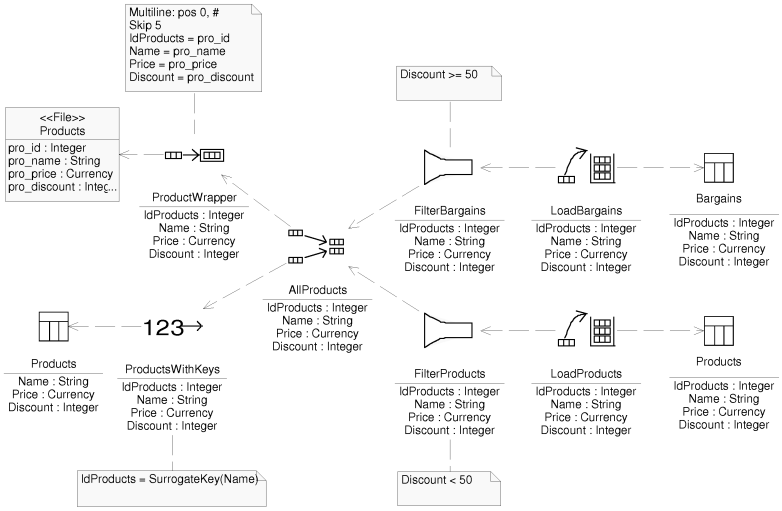


Figure 9.9: Merging two different data sources and multi-target loading

and rename the attributes' names. For the second data source, ProductsWithKeys («Surrogate» mechanism) generates a surrogate key from the Name attribute. Then, AllProducts («Merge» mechanism) is used to integrate the two sources. Finally, all the products have to be loaded into two targets depending on the discount of each product: the products with a discount lower than 50 percent have to be loaded into Products dimension, whereas the products with a discount equal or greater than 50 percent have to be loaded into Bargains dimension. FilterProducts and FilterBargains are used to filter the products and select the corresponding products.

9.4.3 Aggregate and Surrogate Key Process

In Figure 9.10, we have represented a loading process into a **DW**. The grain level of Sales (the data source) is ticket line, but we need the daily total sales in the **DW** (Sales fact table at the right hand side of Figure 9.10). Therefore, Sales are grouped and summed up by product and date in SummedSales. Then, SurrogatedSales adds a surrogate key (Time) based on the Date attribute before loading the data into the **DW**. This surrogate key is used in the **DW** to establish a relation between Sales fact and Time dimension (Sales contains a foreign key to Time). Finally, SalesLoader loads summarized sales into Sales fact table, ProductsLoader loads the product list into Products dimension, and TimeLoader loads time data into Time dimension.

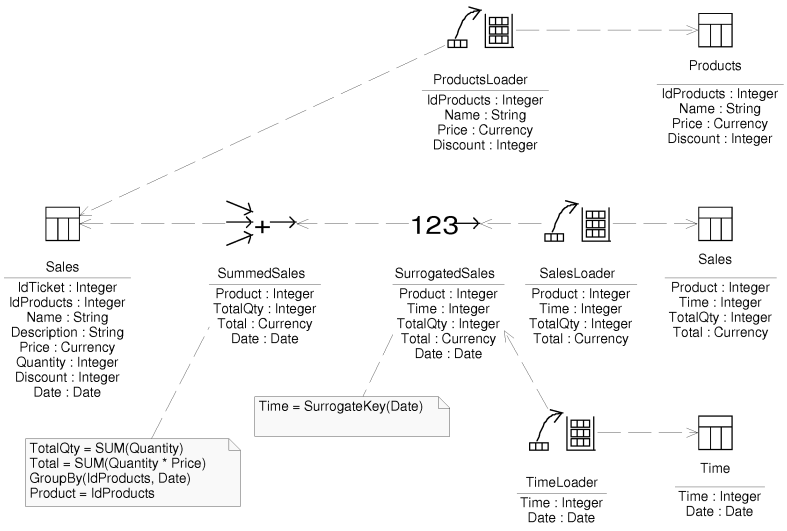


Figure 9.10: Aggregate and surrogate key process

9.5 Conclusions

In this chapter, we have presented the modeling of **ETL** processes as part of our integrated and global approach for **DW** design. Thanks to the use of the **UML**, we can seamlessly model different aspects of a **DW** architecture such as operational data sources, logical schema and **ETL** processes in an integrated manner. In this way, it is very easy to detect inconsistencies between the different schemas of a **DW** architecture and it helps the designer to estimate the feasibility of the development.

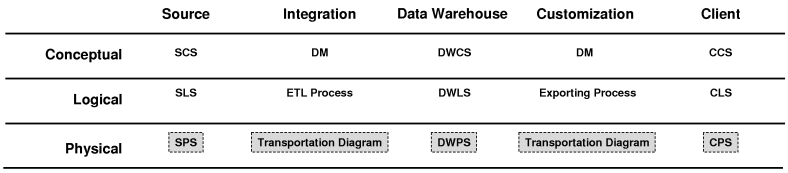
Our approach for modeling **ETL** processes defines a set of **UML** stereotypes that represent the most common **ETL** tasks such as the integration of different data sources, the transformation between source and target attributes, the generation of surrogate keys, and so on. Furthermore, thanks to the use of the **UML** package mechanism, large **ETL** processes can be easily modeled in different **ETL** packages obtaining a very simple but yet powerful approach. Thanks to its simplicity, our approach facilitates the design and subsequent maintenance of **ETL** processes at any modeling phase. Finally, we have implemented our approach in Rational Rose through the ***Rose Extensibility Interface*** **(REI)** [107].

Part III

Physical Level

Chapter 10

Physical Modeling of Data Warehouses



During the few last years, few efforts have been dedicated to the modeling of the physical design (i.e. the physical structures that will host data together with their corresponding implementations) of a **DW** from the early stages of a **DW** project. In this chapter, we present a proposal for the modeling of the physical design of **DW** by using the *component diagram* and *deployment diagram* of **UML**. With these diagrams, we can anticipate important physical design decisions that may reduce the overall development time of a **DW** such as replicating dimension tables, horizontal partitioning of a fact table, the use of particular servers for certain **ETL** processes and so on.

Contents

10.1 Introduction

For more information about the *multidimensional modeling*, consult section 6.2, pp. 56.

Unfortunately, most of the research efforts in designing and modeling **DW** has been focused on the development of **MD** data models [2], while the interest on the physical design of **DW** has been very poor (see related work in Chapter 3). Nevertheless, an outstanding physical design is of a vital importance and highly influences the overall performance of the **DW** [93] and the ulterior maintenance.

In **DW**, as in any other software project, once the conceptual and logical design have been accomplished, we have to deal with the physical design that implements the corresponding specification. Nevertheless, in **DW** and mainly due to the large volume of data that they manage, we normally face with a high number of implementation problems such as the storage of fact tables in different hard disks, copying the same table, vertical and horizontal partitioning and so on. Due to the idiosyncrasy of **DW**, we can adopt several decisions regarding the physical design from the early stages of a **DW** project (in which final users, designers and analysts, and administrators participate). We believe that these decisions will normally reduce the total development time of the **DW**. It should be taken into consideration that we are not saying to accomplish the conceptual modeling of a **DW** taking into account physical issues, instead we argue to model physical aspects and ulterior implementations together with the conceptual modeling of the **DW** from the early stages of a **DW** project.

In this chapter, we present a proposal to accomplish the physical design of **DW** from early stages of a **DW** project. To accomplish this, we propose the use of the *component diagram* and *deployment diagram* of **UML**. Both *component* and *deployment* diagrams must be defined at the same time by **DW** designers and people who will be in charge of the ulterior implementation and maintenance. This is mainly due to the fact that, while the former know how to design and build a **DW**, the latter have a better knowledge in the corresponding implementation and the real hardware and software needs for the correct functioning of the **DW**.

The modeling of the physical design of a **DW** from the early stages of a **DW** project with our proposal provides us many advantages:

- We deal with important aspects of the implementation before we start with the implementation process, and therefore, we can reduce the total development time of the **DW**. This is mainly due to the fact that, after the conceptual modeling has been accomplished, we can have enough information to take some decisions regarding the implementation of the **DW** structures such as replicating dimension tables or making the horizontal

partitioning of a fact table.

- We have a rapid feedback if we have a problem with the **DW** implementation as we can easily track a problem to find out its main reasons.
- It facilitates the communication between all people involved in the design of a **DW** since all of them use the same notation (based on the **UML**) for modeling different aspects of a **DW**.
- It helps us choose both hardware and software on which we intend to implement the **DW**. This also allows us to compare and evaluate different configurations based on user requirements.
- It allows us to verify that all different parts of the **DW** (fact and dimension tables, **ETL** processes, **OLAP** tools, etc.) perfectly fit together.

The rest of this chapter is organized as follows. In Section 10.2, we present main issues that can be specified by using both component and deployment diagrams of **UML**. In Section 10.3, we describe our proposal for using both component and deployment diagrams for the physical design of **DW**. Finally, in Section 10.4, we present our conclusions.

10.2 UML Component and Deployment Diagrams

Implementation diagrams: see UML (Part 11, 3-169).

According to **UML** [97], "*Implementation diagrams show aspects of physical implementation, including the structure of components and the run-time deployment system. They come in two forms: 1) component diagrams show the structure of components, including the classifiers that specify them and the artifacts that implement them; and 2) deployment diagrams show the structure of the nodes on which the components are deployed*".

10.2.1 Component Diagram

Component: see UML (2.5.2.12, 2-30), (3.98.1, 3-174).

The UML says that "*A component represents a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces*". Components represent physical issues such as Enterprise JavaBeans, ActiveX components or configuration files. A component is typically specified by one or more classifiers (classes, interfaces, etc.) that reside on the component. A subset of these classifiers explicitly define the component's external interfaces. Moreover, a component can also contain other components. However,

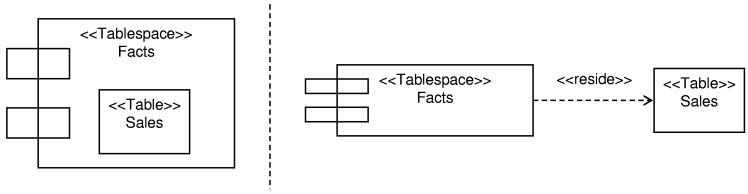


Figure 10.1: Different component representations in a component diagram

a component does not have its own features (attributes, operations, etc.).

Component diagram: see UML (3.95, 3-169).

On the other hand, a component diagram is a graph of components connected by dependency relationships, which shows how classifiers are assigned to components and how the components depend on each other. In a component diagram (see Figure 10.1), a component is represented using a rectangular box, with two rectangles protruding from the left side.

In Figure 10.1, we show the two different representations of a component and the classifiers it contains:

- On the left hand side of the figure, the class (Sales) that resides on the component (Facts) is shown as nested inside the component (this indicates residence and not ownership).
- On the right hand side of the figure, the class is connected to the component by a «reside» dependency.

In this example, both the component and the class are stereotyped: the component is adorned with the «Tablespace» stereotype and the class with the «Table» stereotype; these stereotypes are defined in [90]. This example represents the relationship between a tablespace and a table in a **RDBMS**.

For more information about the *logical modeling* of databases, consult chapter 8, pp. 121.

10.2.2 Deployment Diagram

Deployment diagram: see UML (3.96, 3-171).

According to the **UML**, "*Deployment diagrams show the configuration of run-time processing elements and the software components, processes, and objects that execute on them*". A deployment diagram is a graph of nodes connected by communication associations. A deployment model is a collection of one or more deployment diagrams with their associated documentation.

In a deployment diagram, a node represents a piece of hardware (a computer, a device, etc.) or a software artifact (web server, database,

etc.) in the system, and it is represented by a three-dimensional cube. A node may contain components, which indicates that the components run or execute on the node.

An association of nodes, which is drawn as a solid line between two nodes, indicates a line of communication between the nodes; the association may have a stereotype to indicate the nature of the communication path (e.g. the kind of channel, communication protocol or network).

There are two forms of deployment diagram:

- The descriptor form: it contains types of nodes and components. This form is used as a first-cut deployment diagram during the design of a system, when there is not a complete decision about the final hardware architecture.
- The instance form: it contains specific and identifiable nodes and components. This form is used to show the actual deployment of a system at a particular site, therefore it is normally used in the last steps of the implementation activity, when the details of the deployment site are known.

A deployment diagram is normally used to [10]:

- Explore the issues involved with installing your system into production.
- Explore the dependencies that your system has with other systems that are currently in, or planned for, your production environment.
- Depict a major deployment configuration of a business application.
- Design the hardware and software configuration of an embedded system.
- Depict the hardware/network infrastructure of an organization.

UML deployment diagrams normally make an extensive use of visual stereotypes, because it makes easy to read the diagrams at a glance. Unfortunately, there are no standard palettes of visual stereotypes for **UML** deployment diagrams.

As it is suggested in [10], each node in a deployment diagram may have tens if not hundreds of software components deployed to it: the goal is not to depict all of them, but it is merely to depict those components that are vital to the understanding of the system.

In Figure 10.2, we show the two different representations of a node and the components it contains:

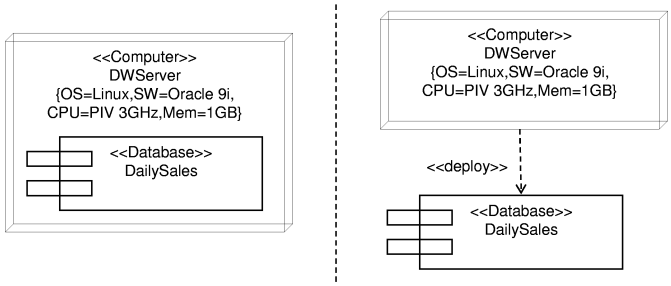


Figure 10.2: Different node representations in a deployment diagram

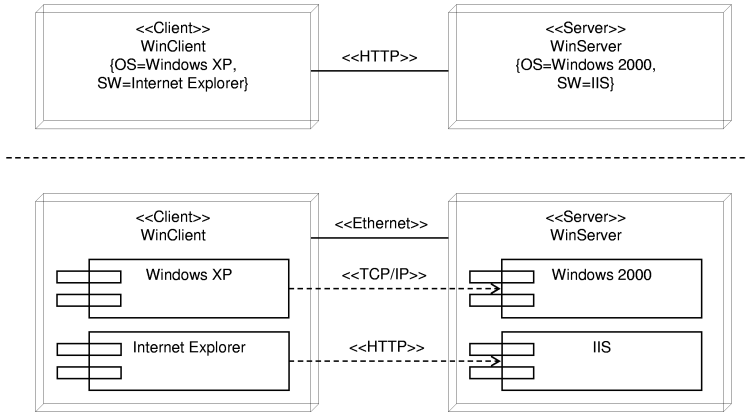


Figure 10.3: Different levels of detail in a deployment diagram

- On the left hand side of the figure, the component (DailySales) that is deployed on the node (DWServer) is shown as nested inside the node.
- On the right hand side of the figure, the component is connected to the node by a «deploy» dependency.

In this example, both the node and the component are stereotyped: the node with the «Computer» stereotype and the component with the «Database» stereotype. Moreover, the node DWServer contains a set of tagged values ({OS}, {SW}, {CPU}, and {Mem}) that allow the designer to describe the particular characteristics of the node.

A deployment diagram can be specified at different levels of detail. For example, in Figure 10.3, we show two versions of the same

deployment diagram. At the top of Figure 10.3, the software deployed in the nodes is specified by means of tagged values. Moreover, the association between the nodes is only adorned with the «HTTP» stereotype, although different protocols can be used in the communication. At the bottom of Figure 10.3, the software deployed in the nodes is depicted as components and different stereotyped dependencies («TCP/IP» and «HTTP») indicate how one component uses the services of another component. However, there are more display possibilities: for example, the designer can omit the tagged values in the diagram and capture them only in the supported documentation.

10.3 Data Warehouse Physical Design

In Chapter 4, we have described our design method for **DW**. Within this method, we use the component and deployment diagrams to model the physical level of **DW**. To achieve this goal, we propose the following five diagrams, which correspond with the five stages presented in Section 4.3:

- Source Physical Schema (SPS): it defines the physical configuration of the data sources that populate the **DW**.
- Integration Transportation Diagram (ITD): it defines the physical structure of the **ETL** processes that extract, transform and load data into the **DW**. This diagram relates the SPS and the next diagram.
- Data Warehouse Physical Schema (DWPS): it defines the physical structure of the **DW** itself.
- Customization Transportation Diagram (CTD): it defines the physical structure of the exportation processes from the **DW** to the specific structures employed by clients. This diagram relates the DWPS and the next diagram.
- Client Physical Schema (CPS): it defines the physical configuration of the structures employed by clients in accessing the **DW**.

The SPS, DWPS, and CPS are based on the **UML** component and deployment diagrams, whereas ITD and CTD are only based on the deployment diagrams.

The five proposed diagrams use an extension of **UML** that we have called *Database Deployment Profile*, which is formed by a series of stereotypes and tagged values.

Throughout the rest of this chapter, we are going to use an example to introduce the different diagrams we propose. In this example,

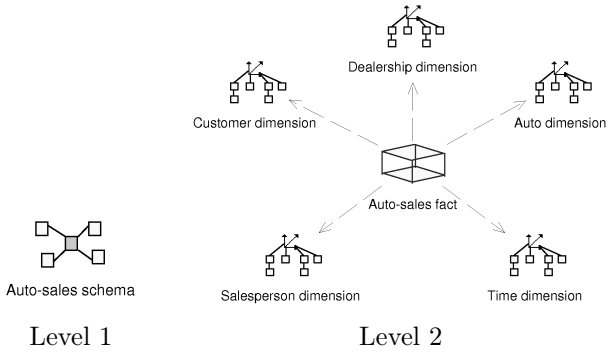


Figure 10.4: Data Warehouse Conceptual Schema

final users need a **DW** that contains the daily sales of a company that do business with automobiles (cars and trucks). There exist two data sources: the sales server, which contains the data about transactions and sales, and the ***Customer Relationship Management* (CRM)** server, which contains the data about the customers who buy products.

In Figure 10.4, we show the Data Warehouse Conceptual Schema (DWCS), which represents the conceptual model of the **DW**. Following our approach [79], we structure the conceptual model into three levels: model definition (level 1), star schema definition (level 2), and dimension/fact definition (level 3). In Figure 10.5, we show the level 3 of the DWCS. In order to avoid a cluttered diagram, we only show the attributes of the fact class (Auto-sales) and two dimension classes (Salesperson and Customer).

For more information about the *multidimensional profile*, consult chapter 6, pp. 53.

In Figure 10.6, we show the Data Warehouse Logical Schema (DWLS), which represents the logical model of the **DW**. In this example, a ***Relational OLAP* (ROLAP)** system has been selected for the implementation of the **DW**, which means the use of the relational model in the logical design of the **DW**. In Figure 10.6, six classes adorned with the stereotype «Table» are showed: Auto, Customer, Dealership, Salesperson, and Time are represented by means of the icon of the stereotype, whereas the table Auto-sales appears with the icon of the stereotype inside the typical representation of a class in **UML**.

For more information about the *logical modeling* of databases, consult chapter 8, pp. 121.

In order to avoid a cluttered diagram, we only display the attributes of Auto-sales and Salesperson. In the Auto-sales table, the attributes IdAuto, IdCustomer, IdDealership, IdSalesperson, and IdTime are the foreign keys that connect the fact table with the dimension tables, whereas the attributes Commission, SP_Commission, Quan-

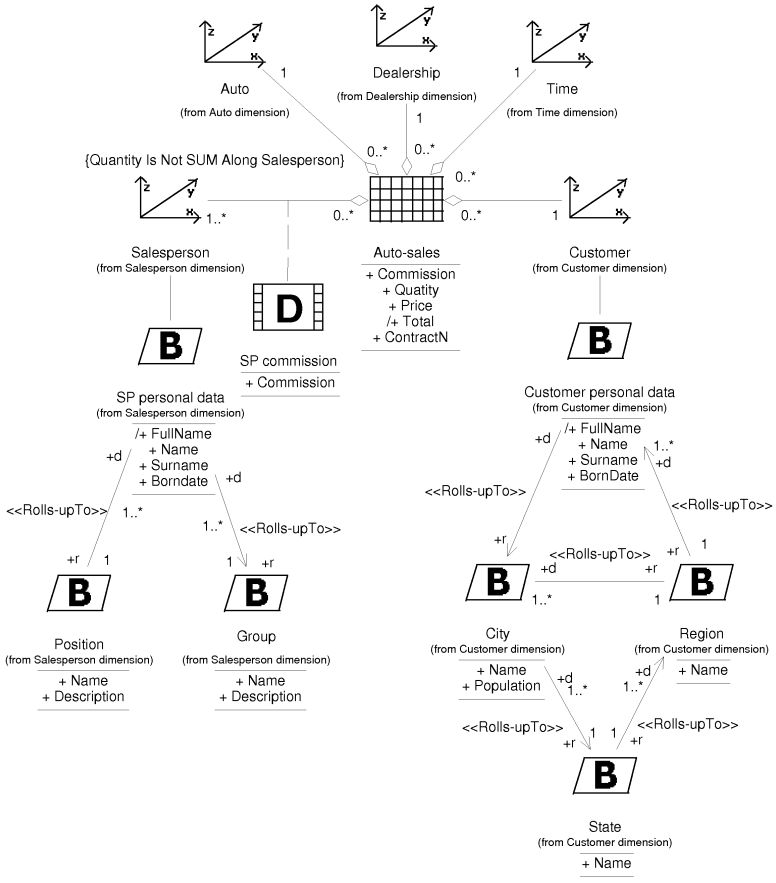


Figure 10.5: Data Warehouse Conceptual Schema (level 3)

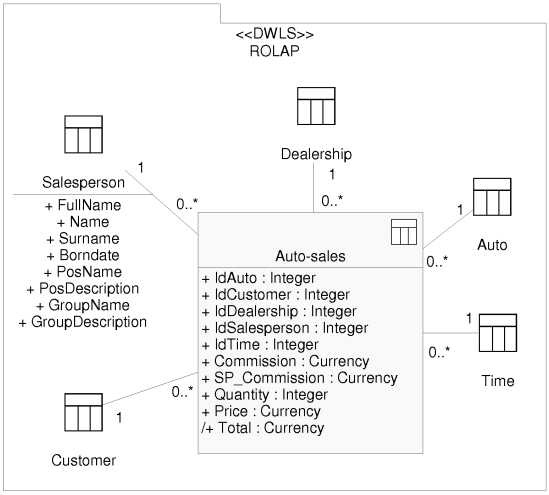


Figure 10.6: Logical model (ROLAP) of the data warehouse

tity, Price, and Total (derived attribute that is precalculated for performance reasons) represent the measures of the fact table. In the Salesperson table, we can notice that this table contains all the attributes of the different dimension levels (see Figure 10.5) following the star schema approach [63]; some attributes have changed their names in order to avoid repeated names. Moreover, some design decisions have been taken: the degenerate dimension represented by the ContractN attribute in Auto-sales fact class (see Figure 10.5) has been omitted, and the degenerate fact represented by the SP commission class is represented by the SP_Commission attribute in the Auto-sales table.

10.3.1 Source Physical Schema

The SPS describes the origins of data of the **DW** from a physical point of view. In Figure 10.7, we show the SPS of our example, which is formed by two servers called SalesServer and CRMServer; for each one of them, the hardware and software configuration is displayed. The first server hosts a database called Sales, whereas the second server hosts a database called Customers.

In our *Database Deployment Profile*, when the storage system is a **RDBMS**, we make use of the *UML Profile for Database Design* [90] that defines a series of stereotypes like «Database» or «Tablespace». Moreover, we have defined our own set of stereotypes: in Figure 10.7, we can see the stereotypes «Server» that defines a computer that

For more information about the *UML Profile for Database Design*, consult Section 8.2, pp. 124.

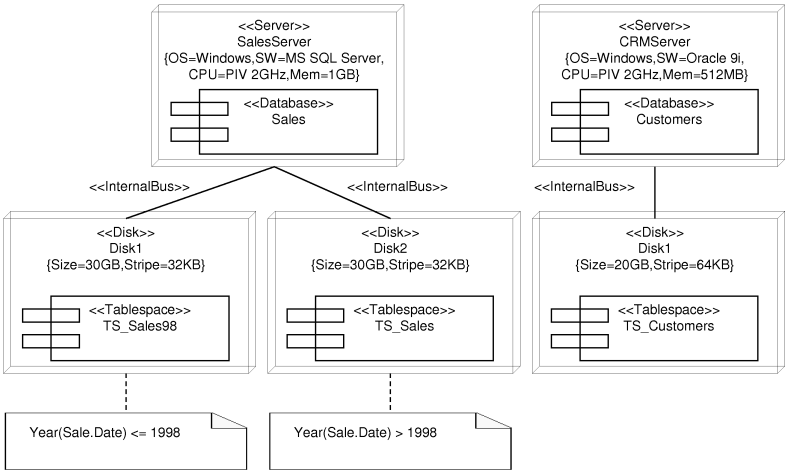


Figure 10.7: Source Physical Schema: deployment diagram

performs server functions, «Disk» to represent a physical disk drive and «InternalBus» to define the type of communication between two elements. Whenever we need to specify additional information in a diagram, we make use of the **UML** notes to incorporate it. For example, in Figure 10.7 we have used two notes to indicate how the data is distributed into the two existing tablespaces.

10.3.2 Data Warehouse Physical Schema

The DWPS shows the physical aspects of the implementation of the **DW**. This diagram is divided up into two parts: the component and deployment diagrams. In the first diagram, the configuration of the logical structures used to store the **DW** is shown. For example, in Figure 10.8, we can observe that the **DW** AutoSales is formed by two tablespaces called Facts and Dimensions: the first tablespace hosts the table Auto-sales and the second tablespace hosts the tables Auto, Customer, Dealership, Salesperson, and Time. Below the name of each table, the text (from ROLAP) is included, which indicates that the tables have been previously defined in a package called ROLAP (Figure 10.6).

In the second diagram, the deployment diagram, different aspects relative to the hardware and software configuration are specified. Moreover, the physical distribution of the logical structures previously defined in the component diagrams is also represented. For example, in Figure 10.9, we can observe the configuration of the server

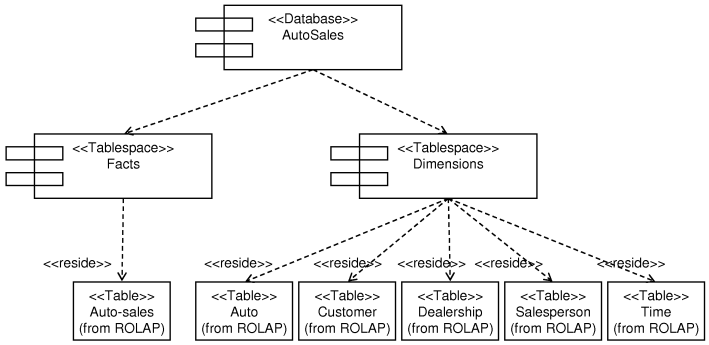


Figure 10.8: Data Warehouse Physical Schema: component diagram

that hosts the **DW**.

One of the advantages of our proposal is that it allows to evaluate and to discuss different implementations during the first stages in the design of a **DW**. In this way, the designer can anticipate some implementation or performance problems. For example, an alternative configuration of the physical structure of the **DW** can be established, as we show in Figure 10.10. In this second alternative, a **RAID** 0 systems has been chosen to host the tablespace Facts in order to improve the response time of the disk drive and the performance of the system in general.

10.3.3 Integration Transportation Diagram

The ITD defines the physical structure of the **ETL** processes used in the loading of data in the **DW** from the data sources. On the one hand, the data sources are represented by means of the SPS and, on the other hand, the **DW** is represented by means of the DWPS. Since the SPS and the DWPS have been defined previously, in this diagram we do not have to define them again, but they are imported.

For example, the ITD for our running example is shown in Figure 10.11. On the left hand side of this diagram, different data source servers are represented: SalesServer and CRMServer, which have been previously defined in Figure 10.7; on the right hand side, the DWServer, previously defined in Figure 10.9, is shown. In this figure, the ETLServer is introduced, an additional server that is used to execute the **ETL** processes. This server communicates with the rest of the servers by means of a series of specific protocols: ***Object Linking and Embedding DataBase*** **(OLEDB)** to communicate with

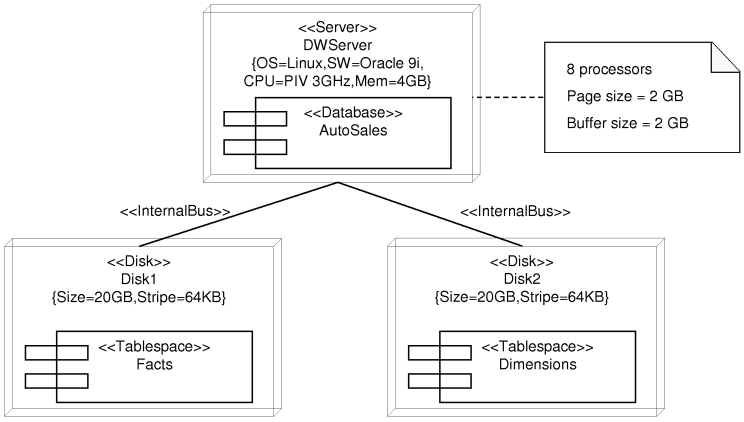


Figure 10.9: Data Warehouse Physical Schema: deployment diagram (version 1)

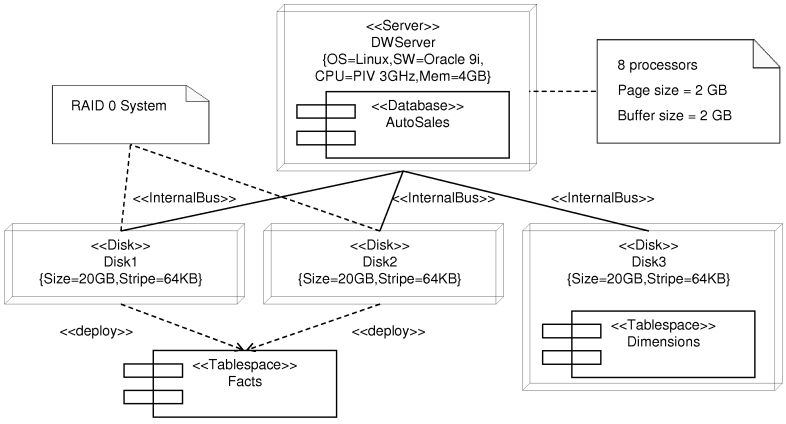


Figure 10.10: Data Warehouse Physical Schema: deployment diagram (version 2)

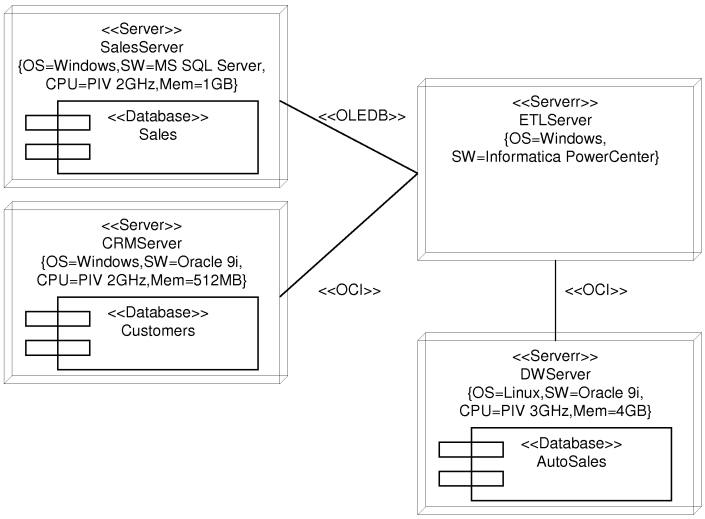


Figure 10.11: Integration Transportation Diagram: deployment diagram

SalesServer because it uses Microsoft SQLServer[1] and ***Oracle Call Interface* (OCI)** to communicate with CRMServer and DWServer because both of them use Oracle.

10.3.4 Client Physical Schema

The CPS defines the physical structure of the specific structures that are used by the clients to access the **DW**. Diverse configurations exist that can be used: exportation of data to **DM**, use of an **OLAP** server, etc. In our example, we have chosen a client/server architecture and the same **DW** server provides access to data for the clients. Therefore, we do not need to define a specific structure for the clients.

10.3.5 Customization Transportation Diagram

The CTD defines the exportation processes from the **DW** towards the specific structures used by the clients. In this diagram, the **DW** is represented by means of the DWPS and clients are represented by means of the CPS. Since the DWPS and the CPS have been previously defined, in this diagram we do not have to define them again, but they are directly imported.

[1]The configuration of a server is defined by means of tagged values: {OS}, {SW}, {CPU}, etc.

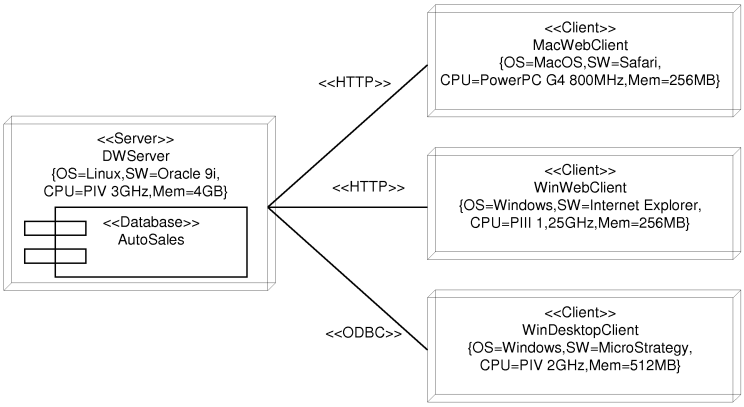


Figure 10.12: Customization Transportation Diagram: deployment diagram

For example, in Figure 10.12, the CTD of our running example is shown. On the left hand side of this diagram, part of the DWPS, which has been previously defined in Figure 10.9, is shown; on the right hand side, three types of clients who will use the **DW** are shown: a Web client with operating system Apple Macintosh, a Web client with operating system Microsoft Windows and, finally, a client with a specific desktop application (MicroStrategy) with operating system Microsoft Windows. Whereas both Web clients communicate with the server by means of ***HyperText Transfer Protocol*** **(HTTP)**, the desktop client uses ***Open Data Base Connectivity*** **(ODBC)**.

10.4 Conclusions

In this chapter, we have presented an adaptation of the component and deployment diagrams of **UML** for the modeling of the physical design of a **DW**. One of the advantages of this proposal is that these diagrams are not used in an isolated way, instead they are used together with other diagrams that we use for the modeling of other aspects of a **DW** (conceptual and logical design, modeling of **ETL** processes, etc.) in the context of our overall method for designing **DW**.

Thanks to the use of the component and deployment diagrams, a **DW** designer can specify both hardware, software, and middelware needs for a **DW** project. The main advantages provided by our approach are as follows:

- Traceability of the design of a **DW**, from the conceptual model up to the physical model.
- Reducing the overall development cost as we accomplish implementation issues from the early stages of a **DW** project. We should take into account that modifying these aspects in ulterior design phases may result in increasing the total cost of the project.
- Different levels of abstraction by providing different levels of details for the same diagram.

Part IV

Finale

Chapter 11

Contributions

In this section, we enumerate the main contributions obtained as a result of our research process. Moreover, we also present the research production that has been materialized as publications in conferences and journals.

Contents

11.1 Introduction

The aim of this chapter is to outline the main contributions of this thesis. Section 11.2 highlights the main elements we have presented in this work. Then, Section 11.3 contains the publications that generated this thesis work. We would like to remark that some publications written during the elaboration of this thesis, that cannot be regarded as proper thesis work, have been omitted.

11.2 Main Contributions

This thesis defines the following new elements:

- The *UML Profile for Multidimensional Modeling*, the definition of an extension of **UML** in the form of a profile to model main **MD** properties in the conceptual design of **DW**.
- The definition of an extension of **UML** to model attributes as first-class modeling elements.
- The *Data Mapping Diagram*, which is a new kind of diagram, particularly customized for the tracing of the data flow, at various degrees of detail, in a **DW** environment.
- The *ETL Profile*, an extension of **UML** for the design of **ETL** processes.
- The *Database Deployment Profile*, for modeling different aspects of the physical level of a **DW**.
- The development of an add-in for Rational Rose that allows using the *UML Profile for Multidimensional Modeling*.

11.3 Research Production

Different contributions of the research presented in this thesis have been presented in national and international scientific forums: conferences, journals, and book chapters. In this section, we present our main publications that are directly related to the research of this thesis; nevertheless, we have excluded some publications that are partly related to this thesis, but they are not proper thesis work.

All the publications we present underwent a standard peer review process by two or more qualified reviewers to evaluate the contribution and ensure technical veracity. The reviewers provided substantial criticism and feedback that allowed us to improve the quality of our proposals.

In the following sections, the different contributions are presented in chronological order, according to the moment they were presented in a conference or published in a journal. For each one of the contributions, we include the whole bibliography reference, the abstract, and the acceptance rate if it is known.

The summary of contributions is:

- National conference (1): ADTO'01.
- International conferences (12): ICEIS'01, XMLDM'02, PHD-OOS'02, BNCOD'02, UML'02, ER'02, DMDW'03, ER'03, ICEIS'04, ADVIS'04, ER'04, DOLAP'04.
- International journals (3): IJCIS'02, JDM'04, JDM'06.
- Book chapter (1): IDEA'03.

The most remarkable contributions are:

- UML'02, where a **UML** profile for the **MD** modeling is introduced.
- ER'02, where the use of **UML** packages for the **MD** modeling is presented.
- ER'03, where the conceptual modeling of **ETL** processes is presented.
- JDM'04, where the use of UML class, state, and interaction diagrams for **MD** modeling is presented.
- ER'04, where the data mapping diagram for **DW** and the use of attributes as first-class modeling elements in **UML** is introduced.
- JDM'06, where the use of **UML** component and deployment diagrams for physical modeling of **DW** is presented.

In Figure 11.1, we graphically show the distribution of the publications through the four years of thesis work. Beside the acronym of each publication, a little text describes the content or the achievement of the publication.

11.3.1 ICEIS'01

S. Luján-Mora and E. Medina. Reducing Inconsistency in Data Warehouses. In *Proceedings of the 3rd International Conference on Enterprise Information Systems (ICEIS'01)*, pages 199–206, Setúbal, Portugal, July 7 - 10 2001. ICEIS Press

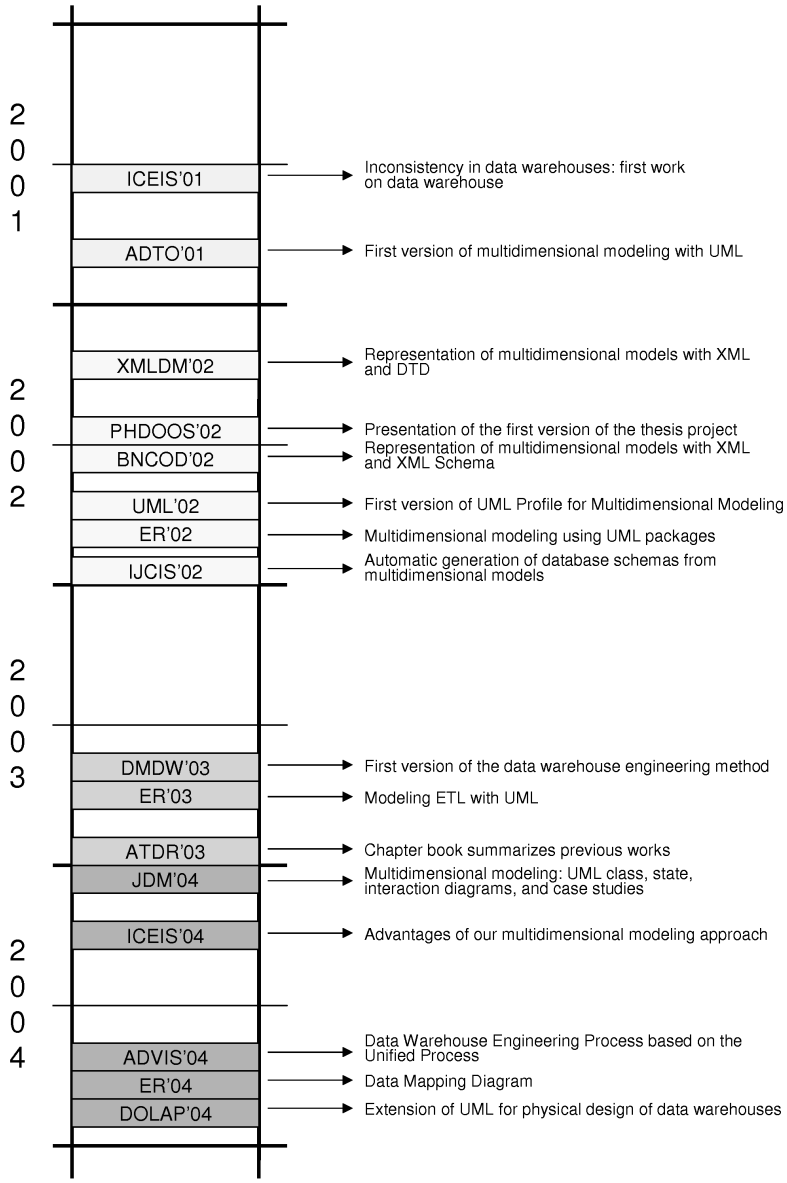


Figure 11.1: Chronology of the contributions

Abstract: A data warehouse is a repository of data formed of a collection of data extracted from different and possible heterogeneous sources (e.g., databases or files). One of the main problems in integrating databases into a common repository is the possible inconsistency of the values stored in them, i.e., the very same term may have different values, due to misspelling, a permuted word order, spelling variants and so on. In this paper, we present an automatic method for reducing inconsistency found in existing databases, and thus, improving data quality. All the values that refer to a same term are clustered by measuring their degree of similarity. The clustered values can be assigned to a common value that, in principle, could substitute the original values. Thus, the values are uniformed. The method we propose provides good results with a considerably low error rate.

Acceptance rate: More than 200 submissions, 68 papers accepted. $AR \simeq 0.34$

11.3.2 ADTO'01

J. Trujillo, S. Luján-Mora, and E. Medina. Utilización de UML para el modelado multidimensional. In *I Taller de Almacenes de Datos y Tecnología OLAP (ADTO 2001), VI Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2001)*, pages 12–17, Almagro, Spain, November 22 2001

Abstract: Los almacenes de datos (AD), las bases de datos multidimensionales (BDM) y las aplicaciones de Procesamiento Analítico en Línea (On-Line Analytical Processing, OLAP) están basadas en el modelado multidimensional (MD). En este artículo presentamos cómo se puede utilizar el Lenguaje de Modelado Unificado (Unified Modeling Language, UML) para llevar a cabo el diseño conceptual de estos sistemas. La estructura del modelo se especifica mediante un diagrama de clases UML que considera las principales propiedades del modelado MD por medio de un número reducido de restricciones sobre UML. Además, proponemos una notación de clase (clase cubo) compatible con UML para representar los requisitos OLAP iniciales de usuario. El comportamiento del sistema se modela a través de los diagramas de estados e interacciones. Nuestra propuesta está soportada por una herramienta CASE que facilita la tarea del modelado conceptual y genera de una forma semiautomática la implementación de un modelo conceptual en una herramienta OLAP específica.

Acceptance rate: Unknown.

11.3.3 XMLDM'02

S. Luján-Mora, E. Medina, and J. Trujillo. A Web-Oriented Ap-

proach to Manage Multidimensional Models through XML Schemas and XSLT. In *Proceedings of the XML-Based Data Management and Multimedia Engineering (EDBT 2002 Workshops)*, volume 2490 of *Lecture Notes in Computer Science*, pages 29–44, Prague, Czech Republic, March 24 2002. Springer-Verlag

Abstract: Multidimensional (MD) modeling is the foundation of data warehouses, MD databases, and OLAP applications. In the last years, there have been some proposals to represent MD properties at the conceptual level. In this paper, we present how to manage the representation, manipulation, and presentation of MD models on the web by means of eXtensible Stylesheet Language Transformations (XSLT). First, we use eXtensible Markup Language (XML) to consider main MD modeling properties at the conceptual level. Next, an XML Schema allows us to generate valid XML documents that represent MD models. Finally, we provide XSLT stylesheets that allow us to automatically generate HTML pages from XML documents, thereby supporting different presentations of the same MD model easily. A CASE tool that gives support to all theoretical issues presented in the paper has been developed.

Acceptance rate: Approximately 130 submissions, 48 papers accepted. $AR \simeq 0.37$

11.3.4 PHDOOS'02

S. Luján-Mora. Multidimensional Modeling using UML and XML. In *Proceedings of the 12th Workshop for PhD Students in Object-Oriented Systems (PhDOOS 2002)*, volume 2548 of *Lecture Notes in Computer Science*, pages 48–49, Málaga, Spain, June 10 - 14 2002. Springer-Verlag

Abstract: Multidimensional (MD) modeling is the foundation of data warehouses, MD databases, and On-Line Analytical Processing (OLAP) applications. In the past years, there have been some proposals for representing the main MD properties at the conceptual level providing their own notations. In this paper, we present an extension of the Unified Modeling Language (UML), by means of stereotypes, to elegantly represent main structural and dynamic MD properties at the conceptual level. Moreover, we use the eXtensible Markup Language (XML) to store MD models. Then, we apply the eXtensible Stylesheet Language Transformations (XSLT), that allow us to automatically generate HTML pages from XML documents, thereby supporting different presentations of the same MD model easily. Finally, we show how to accomplish all these tasks using Rational Rose 2000.

Acceptance rate: Unknown.

11.3.5 BNCOD'02

E. Medina, S. Luján-Mora, and J. Trujillo. Handling Conceptual Multidimensional Models using XML through DTDs. In *Proceedings of 19th British National Conference on Databases (BNCOD 2002)*, volume 2405 of *Lecture Notes in Computer Science*, pages 66–69, Sheffield, UK, July 17 - 19 2002. Springer-Verlag

Abstract: In the last years, several approaches have been proposed to easily capture main multidimensional (MD) properties at the conceptual level. In this paper, we present how to handle MD models by using the eXtensible Markup Language (XML) through a Document Type Definition (DTD) and the eXtensible Stylesheet Language Transformations (XSLT). To accomplish this objective, we start by providing a DTD which allows to directly generate valid XML documents that represents MD properties at the conceptual level. Afterwards, we provide XSLT stylesheets to automatically generate HTML pages that correspond to different presentations of the same MD model.

Acceptance rate: Unknown.

11.3.6 UML'02

S. Luján-Mora, J. Trujillo, and I. Song. Extending UML for Multidimensional Modeling. In *Proceedings of the 5th International Conference on the Unified Modeling Language (UML'02)*, volume 2460 of *Lecture Notes in Computer Science*, pages 290–304, Dresden, Germany, September 30 - October 4 2002. Springer-Verlag

Abstract: Multidimensional (MD) modeling is the foundation of data warehouses, MD databases, and On-Line Analytical Processing (OLAP) applications. In the past few years, there have been some proposals for representing the main MD properties at the conceptual level providing their own notations. In this paper, we present an extension of the Unified Modeling Language (UML), by means of stereotypes, to elegantly represent main structural and dynamic MD properties at the conceptual level. We make use of the Object Constraint Language (OCL) to specify the constraints attached to the defined stereotypes, thereby avoiding an arbitrary use of these stereotypes. The main advantage of our proposal is that it is based on a well-known standard modeling language, thereby designers can avoid learning a new specific notation or language for MD systems. Finally, we show how to use these stereotypes in Rational Rose 2000 for MD modeling.

Acceptance rate: 99 submissions, 30 papers accepted. $AR \simeq 0.30$

11.3.7 ER'02

S. Luján-Mora, J. Trujillo, and I. Song. Multidimensional Modeling with UML Package Diagrams. In *Proceedings of the 21st International Conference on Conceptual Modeling (ER'02)*, volume 2503 of *Lecture Notes in Computer Science*, pages 199–213, Tampere, Finland, October 7 - 11 2002. Springer-Verlag

Abstract: The Unified Modeling Language (UML) has become the *de facto* standard for object-oriented analysis and design, providing different diagrams for modeling different aspects of a system. In this paper, we present the development of multidimensional (MD) models for data warehouses (DW) using UML package diagrams. In this way, when modeling complex and large DW systems, we are not restricted to use flat UML class diagrams. We present design guidelines and illustrate them with various examples. We show that the correct use of the package diagrams using our design guidelines will produce a very simple yet powerful design of MD models. Furthermore, we provide a UML extension by means of stereotypes of the particular package items we use. Finally, we show how to use these stereotypes in Rational Rose 2000 for MD modeling.

Acceptance rate: Approximately 130 submissions, 30 papers accepted. $AR \simeq 0.23$

11.3.8 IJCIS'02

J. Trujillo and S. Luján-Mora. Automatically Generating Structural and Dynamic Information of OLAP Applications from Object-Oriented Conceptual Models. *International Journal of Computer & Information Science*, 3(4):227–236, December 2002

Abstract: Graphical conceptual models for On-Line Analytical Processing (OLAP) applications should semi-automatically generate the database schema and the corresponding multidimensional (MD) model for a specific target commercial OLAP tool. However, this generation process is not immediate as the semantics represented by these conceptual models are different from those considered by the underlying MD models of OLAP tools. Therefore, some transformations for these differences are needed in this process.

In the context of graphical conceptual models, we provide an object-oriented conceptual model that provides a Unified Modeling Language (UML) graphical notation to represent both structural and dynamics properties of MD models and initial user requirements at the conceptual level. In this paper, on one hand, we present how to semi-automatically generate the database schema and the underlying MD model for one of the most leading commercial OLAP tools from our model. In this process, some semantics represented in the

model are transformed into those considered by the underlying MD model of the target OLAP tool. On the other hand, initial user requirements are translated into their corresponding definitions in the target OLAP tool. In this way, the final user is able to start the analysis process from the initial requirements specified at the conceptual level. Finally, we present the prototype of the Computer Aided Software Engineering (CASE) tool that gives support to both the model definition and this generation process.

Acceptance rate: Unknown.

11.3.9 DMDW'03

S. Luján-Mora and J. Trujillo. A Comprehensive Method for Data Warehouse Design. In *Proceedings of the 5th International Workshop on Design and Management of Data Warehouses (DMDW'03)*, pages 1.1–1.14, Berlin, Germany, September 8 2003

Abstract: A data warehouse (DW) is a complex information system primarily used in the decision making process by means of On-Line Analytical Processing (OLAP) applications. Although various methods and approaches have been presented for designing different parts of DWs, such as the conceptual and logical schemas or the Extraction-Transformation-Loading (ETL) processes, no general and standard method exists to date for dealing with the whole design of a DW. In this paper, we fill this gap by presenting a method based on the Unified Modeling Language (UML) that allows the user to tackle all DW design phases and steps, from the operational data sources to the final implementation and including the definition of the ETL processes. The main advantages of our proposal are: the use of a standard modeling notation (UML) in the models accomplished in the different design phases, the integration of different design phases in a single and coherent framework and the use of a grouping mechanism (UML packages) that allows the designer to layer the models according to different levels of detail. Finally, we also provide a set of steps that guide the DW design.

Acceptance rate: 21 submissions, 10 papers accepted. $AR \simeq 0.47$

11.3.10 ER'03

J. Trujillo and S. Luján-Mora. A UML Based Approach for Modeling ETL Processes in Data Warehouses. In *Proceedings of the 22nd International Conference on Conceptual Modeling (ER'03)*, volume 2813 of *Lecture Notes in Computer Science*, pages 307–320, Chicago, USA, October 13 - 16 2003. Springer-Verlag

Abstract: Data warehouses (DWs) are complex computer systems whose main goal is to facilitate the decision making process of knowledge workers. ETL (Extraction-Transformation-Loading) processes are responsible for the extraction of data from heterogeneous operational data sources, their transformation (conversion, cleaning, normalization, etc.) and their loading into DWs. ETL processes are a key component of DWs because incorrect or misleading data will produce wrong business decisions, and therefore, a correct design of these processes at early stages of a DW project is absolutely necessary to improve data quality. However, not much research has dealt with the modeling of ETL processes. In this paper, we present our approach, based on the Unified Modeling Language (UML), which allows us to accomplish the conceptual modeling of these ETL processes together with the conceptual schema of the target DW in an integrated manner. We provide the necessary mechanisms for an easy and quick specification of the common operations defined in these ETL processes such as, the integration of different data sources, the transformation between source and target attributes, the generation of surrogate keys and so on. Moreover, our approach allows the designer a comprehensive tracking and documentation of entire ETL processes, which enormously facilitates the maintenance of these processes. Another advantage of our proposal is the use of the UML (standardization, ease-of-use and functionality) and the seamless integration of the design of the ETL processes with the DW conceptual schema. Finally, we show how to use our integrated approach by using a well-known modeling tool such as Rational Rose.

Acceptance rate: Unknown.

11.3.11 ATDR'03

J. Trujillo, S. Luján-Mora, and I. Song. *Advanced Topics in Database Research*, volume 2, chapter Applying UML for designing multidimensional databases and OLAP applications, pages 13–36. Idea Group Publishing, 2003

Abstract: Multidimensional (MD) modeling is the basis for Data warehouses (DW), multidimensional databases (MDB) and On-Line Analytical Processing (OLAP) applications. In this chapter, we present how the Unified Modeling Language (UML) can be successfully used to represent both structural and dynamic properties of these systems at the conceptual level. The structure of the system is specified by means of a UML class diagram that considers the main properties of MD modeling with minimal use of constraints and extensions of the UML. If the system to be modeled is too complex, thereby leading us to a considerable number of classes and relationships, we sketch out how to use the package grouping mechanism provided by the UML to

simplify the final model. Furthermore, we provide a UML-compliant class notation (called cube class) to represent OLAP initial user requirements. We also describe how we can use the UML state and interaction diagrams to model the behavior of a data warehouse system. We believe that our innovative approach provides a theoretical foundation for simplifying the conceptual design of multidimensional systems and our examples illustrate the use of our approach.

Acceptance rate: Unknown.

11.3.12 JDM'04

J. Trujillo, S. Luján-Mora, and I. Song. Applying UML and XML for designing and interchanging information for data warehouses and OLAP applications. *Journal of Database Management*, 15(1):41–72, January-March 2004

Abstract: Multidimensional (MD) modeling is the basis for Data warehouses (DW), multidimensional databases (MDB) and On-Line Analytical Processing (OLAP) applications. In this paper, we present how the Unified Modeling Language (UML) can be successfully used to represent both structural and dynamic properties of these systems at the conceptual level. The structure of the system is specified by means of a UML class diagram that considers the main properties of MD modeling with minimal use of constraints and extensions of the UML. If the system to be modeled is too complex, thereby leading us to a considerable number of classes and relationships, we describe how to use the package grouping mechanism provided by the UML to simplify the final model. Furthermore, we provide a UML-compliant class notation (called cube class) to represent OLAP users' initial requirements. We also describe how we can use the UML state and interaction diagrams to model the behavior of a data warehouse system. To facilitate the interchange of conceptual MD models, we provide a Document Type Definition (DTD) which allows us to represent the same MD modeling properties that can be considered by using our approach. From this DTD, we can directly generate valid eXtensible Markup Language (XML) documents that represent MD models at the conceptual level. We believe that our innovative approach provides a theoretical foundation for simplifying the conceptual design of MD systems and the examples included in this paper clearly illustrate the use of our approach.

Acceptance rate: Indexed in the Journal Citation Report.

11.3.13 ICEIS'04

S. Luján-Mora, J. Trujillo, and P. Vassiliadis. Advantages of UML for Multidimensional Modeling. In *Proceedings of the 6th International*

Conference on Enterprise Information Systems (ICEIS 2004), pages 298–305, Porto, Portugal, April 14 - 17 2004. ICEIS Press

Abstract: In the last few years, various approaches for the multidimensional (MD) modeling have been presented. However, none of them has been widely accepted as a standard. In this paper, we summarize the advantages of using object orientation for MD modeling. Furthermore, we use the UML, a standard visual modeling language, for modeling every aspect of MD systems. We show how our approach resolves elegantly some important problems of the MD modeling, such as multistar models, shared hierarchy levels, and heterogeneous dimensions. We believe that our approach, based on the popular UML, can be successfully used for MD modeling and can represent most of frequent MD modeling problems at the conceptual level.

Acceptance rate: 605 submissions, 277 papers accepted. $AR \simeq 0.45$

11.3.14 ADVIS'04

S. Luján-Mora and J. Trujillo. A Data Warehouse Engineering Process. In *Proceedings of the 3rd Biennial International Conference on Advances in Information Systems (ADVIS'04)*, volume 3261 of *Lecture Notes in Computer Science*, pages 14–23, Izmir, Turkey, October 20 - 22 2004. Springer-Verlag

Abstract: Developing a data warehouse (DW) is a complex, time consuming and prone to fail task. Different DW models and methods have been presented during the last few years. However, none of them addresses the whole development process in an integrated manner. In this paper, we present a DW development method, based on the Unified Modeling Language (UML) and the Unified Process (UP), which addresses the design and development of both the DW back-stage and front-end. We extend the UML in order to accurately represent the different parts of a DW. Our proposal provides a seamless method for developing DWs.

Acceptance rate: 203 submissions, 61 papers accepted. $AR \simeq 0.30$

11.3.15 ER'04

S. Luján-Mora, P. Vassiliadis, and J. Trujillo. Data Mapping Diagrams for Data Warehouse Design with UML. In *Proceedings of the 23rd International Conference on Conceptual Modeling (ER'04)*, volume 3288 of *Lecture Notes in Computer Science*, pages 191–204, Shanghai, China, November 8 - 12 2004. Springer-Verlag

Abstract: In Data Warehouse (DW) scenarios, ETL (Extraction, Transformation, Loading) processes are responsible for the extraction of data from heterogeneous operational data sources, their transformation (conversion, cleaning, normalization, etc.) and their loading into the DW. In this paper, we present a framework for the design of the DW back-stage (and the respective ETL processes) based on the key observation that this task fundamentally involves dealing with the specificities of information at very low levels of granularity including transformation rules at the attribute level. Specifically, we present a disciplined framework for the modeling of the relationships between sources and targets in different levels of granularity (including coarse mappings at the database and table levels to detailed inter-attribute mappings at the attribute level). In order to accomplish this goal, we extend UML (Unified Modeling Language) to model attributes as first-class citizens. In our attempt to provide complementary views of the design artifacts in different levels of detail, our framework is based on a principled approach in the usage of UML packages, to allow zooming in and out the design of a scenario.

Acceptance rate: 295 submissions, 57 papers accepted. $AR \simeq 0.19$

11.3.16 DOLAP'04

S. Luján-Mora and J. Trujillo. Modeling the Physical Design of Data Warehouses from a UML Specification. In *Proceedings of the ACM Seventh International Workshop on Data Warehousing and OLAP (DOLAP 2004)*, pages 48–57, Washington D.C., USA, November 12 - 13 2004. ACM

Abstract: During the few last years, several approaches have been proposed to model different aspects of a Data Warehouse (DW), such as the conceptual model of the DW, the design of the ETL (Extraction, Transformation, Loading) processes, the derivation of the DW models from the enterprise data models, etc. At the end, a DW has to be deployed to a database environment and that takes many decisions of a physical nature. However, few efforts have been dedicated to the modeling of the physical design (i.e. the physical structures that will host data together with their corresponding implementations) of a DW from the early stages of a DW project. From our previously presented DW engineering process, in this paper we present our proposal for the modeling of the physical design of DWs by using the *component diagrams* and *deployment diagrams* of the Unified Modeling Language (UML). Our approach allows the designer to anticipate important physical design decisions that may reduce the overall development time of a DW such as replicating dimension tables, vertical and horizontal partitioning of a fact table,

the use of particular servers for certain ETL processes and so on. Moreover, our approach allows the designer to cover all main design phases of DWs, from the conceptual modeling phase until the final implementation, as we show with an example in this paper.

Acceptance rate: 29 submissions, 14 papers accepted. $AR \simeq 0.48$

11.3.17 JDM'06

S. Luján-Mora and J. Trujillo. Physical Modeling of Data Warehouses by using UML Component and Deployment Diagrams: design and implementation issues. *Journal of Database Management*, 17(1), January-March 2006. Accepted to be published

Abstract: Several approaches have been proposed to model different aspects of a Data Warehouse (DW) during the last years, such as the modeling of DW at the conceptual and logical level, the design of the ETL (Extraction, Transformation, Loading) processes, the derivation of the DW models from the enterprise data models, the customization of a DW schema etc. At the end, a DW has to be deployed to a database environment and that takes many decisions of a physical nature. However, few efforts have been dedicated to the modeling of the physical design (i.e. the physical structures that will host data together with their corresponding implementations) of a DW from the early stages of a DW project. However, we argue that some physical decision can be taken from gathering main user requirements.

From our previously presented DW engineering process, in this paper we present our proposal for the modeling of the physical design of DWs by using the *component diagrams* and *deployment diagrams* of the Unified Modeling Language (UML). Our approach allows the designer to anticipate important physical design decisions that may reduce the overall development time of a DW such as replicating dimension tables, vertical and horizontal partitioning of a fact table, the use of particular servers for certain ETL processes and so on. Moreover, our approach allows the designer to cover all main design phases of DWs, from the conceptual modeling phase until the final implementation, as we show with a case study that is implemented on top of a commercial DW management server.

Acceptance rate: Indexed in the Journal Citation Report.

Chapter 12

Conclusions and Future Work

In this last chapter, we present the conclusions of our thesis work. Then, we introduce our immediate future work and we outline some research lines continuing this work.

Contents

12.1 Conclusions

Developing a **DW** is a complex, expensive, time consuming, and prone to fail task. Different **DW** models and methods have been presented during the last few years. However, none of them addresses the whole development process in an integrated manner. In this thesis, we have presented our **DW** development method, based on the **UML** and the **UP**, which addresses the analysis and design of both the **DW** back-stage and front-end. For this task, we have extended the **UML** in order to accurately represent the different parts and properties of a **DW**. Our proposal provides a seamless method for developing **DW** and it is a great help when designing, implementing and deploying a **DW**.

For more information about the *contributions* of this thesis, consult chapter 11, pp. 175.

Following our approach, we design a **DW** as follows:

1. We use **UML** to model the data sources of the **DW** at the conceptual level.
2. Then, we use our *Multidimensional Profile* for the design of the **DW** at the conceptual level.
3. We apply our *Data Mapping Profile* for creating the mapping between the data sources and the **DW** at the conceptual level.
4. We use different **UML** extensions to model the data sources and the **DW** at the logical level: *UML Profile for Database Design* [90], a **UML** profile for data modeling [10], a **UML** extension for the modeling of **XML** documents by means of ***Document Type Definition*** **(DTD)** and XML Schemas [105], etc.
5. Then, we design the **ETL** processes that will be responsible for the gathering, transforming and uploading data from the data sources into the **DW**. We use our *ETL Profile* for this task.
6. Finally, we use our *Database Deployment Profile* for taking physical design decisions.

The main advantages of our proposal are:

- The definition of a set of **UML** profiles, which define an extension to the **UML** but keep the **UML** metamodel intact.
- The use of the same notation (**UML**) for designing the different **DW** models and the corresponding transformations in an integrated manner.
- The use of the **UML** importing mechanism, which guarantees the designer that each element is defined once, because the same element can be used in different models.

12.2 Future Work

Obviously, this thesis work can be continued following several different research lines, because one has to draw the line somewhere and decide when to stop. The **DW** design can be related to other areas; actually, some research has been done from this thesis about quality metrics for **DW** conceptual models [119, 120] and **DW** security [42].

In the following, we present the future work divided into short term, medium term, and long term.

12.2.1 Short Term

- We would like to publish this thesis as a book: the content and structure will be adapted and extended.
- We are working on the definition of a set of complexity metrics for the data mapping diagrams. These metrics will allow the designer to compare different design choices and select the best based on objective measures.
- We are also working on a template specification mechanism for frequently used transformations in data mapping diagrams.
- We are also studying different languages for the formal definition of the transformations in data mapping diagrams, specifically **OCL** [97], Datalog [91] and **QVT** [103].

12.2.2 Medium Term

- We are going to adapt our approach to **UML** 2.0[1] as soon as it is accepted as a standard.
- **UML** 2.0 will change the extension mechanism. Therefore, once **UML** 2.0 has been accepted, we will finish the formal definition of the different profiles we present in this work.
- Once the profiles have been defined, we are going to implement them as Rational Rose add-ins.
- We are currently working on some automatic transformations from the different models to the implementation. We are considering the implementation of the **MD** conceptual models on pure **MD** databases, object-relational databases, and **OO** databases. We are also thinking about the transformation of the **ETL** models into commercial target platforms.

[1] Current adopted version is 1.5 (March, 2003), version 2.0 is in finalization underway.

12.2.3 Long Term

- We would like to provide a detail description of our *Data Warehouse Engineering Process* and develop some case studies based on our approach.
- We plan to include new **UML** diagrams (sequence, collaboration, statechart, and activity diagrams) to model dynamic properties of **DW**.
- We would like to carry out an empirical evaluation of our proposal, in order to validate the correctness and usefulness of our approach.
- Finally, we would like to align our approach with the **MDA** initiative [98].

Appendix A

Advantages of the UML Profile for Multidimensional Modeling

In this appendix, we summarize the advantages of using object orientation for **MD** modeling. We show how our approach resolves elegantly some important problems of the **MD** modeling, such as multistar models, shared hierarchy levels, and heterogeneous dimensions.

Contents

A.1 Introduction

In Chapter 6, we have presented our **MD** modeling approach based on the **UML**. We take advantage of the flexibility of the **UML** to elegantly represent main **MD** properties at a conceptual level. Moreover, our approach imposes a three-layered schema that guides the designer in modeling the **MD** schema and the final user in navigating in the schema [79].

In this appendix, we show how our approach resolves the following important problems of the **MD** modeling:

- Multistar models.
- Shared dimensions.
- Shared hierarchy levels.
- Multiple and alternative classification hierarchy levels.
- Heterogeneous dimensions.
- Shared aggregation.
- Derivation rules.

The remainder of this appendix is organized as follows. Section A.2 highlights the main situations where the use of **UML** shows great advantages for **MD** modeling with respect to other approaches. Finally, Section A.3 presents the main conclusions.

A.2 Advantages for Multidimensional Modeling

In this section we highlight the main situations where the use of **UML** means a considerable advantage for **MD** modeling regarding other approaches. To exemplify our approach, we will use a simplified version of the **DW** example taken from [63]. In this example, there are three separate inventory definitions within the same model, representing different approaches of the inventory problem. The first approach is the *inventory snapshot*, which measures the inventory levels in a regular period of time (every day, every week, etc.). The second is the *delivery status inventory*, which tracks the disposition of all the items in a delivery until they leave the warehouse. Finally, the third is the *transaction inventory*; in this case every change of status of delivered products is recorded throughout the deliver flow of the product.

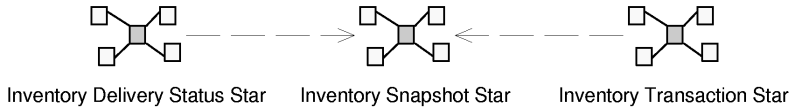


Figure A.1: Multistar multidimensional model

A.2.1 Multistar Models

The *multistar* concept, also called *fact constellation*, refers to the situation where a single **MD** model has multiple facts, and therefore, creating multiple star schemas. Basically, this structure is required when the facts do not share all the dimensions [63].

For more information about the *three-layered model*, consult section 6.3.1, pp. 62.

Our approach is based on a three-layered model defined by means of UML packages. At level 1, multiple packages that represent different star schemas can be specified. For example, in Figure A.1, the first level of the warehouse example is depicted. According to our **MD** approach, we have defined three packages that represent a star schema each one: Inventory Delivery Status Star, Inventory Snapshot Star, and Inventory Transaction Star. A UML dependency (represented as a dotted line with an arrow) connecting two packages indicates that one package uses elements (e.g. dimensions, hierarchy levels) defined in the other. The direction of the dependency indicates that the common elements shared by the two packages were first defined in the package pointed to by the arrow. To simplify the design, and therefore, reducing the number of dependencies, we highly recommend to choose a star schema to define the dimensions. Then, other schemas can use them with no need to define them again. If the common elements had been first defined in another package, the direction of the arrow would have been different.

A.2.2 Support for Different Building Perspectives

There exist two "extreme" perspectives of building a DW [65]:

- To build the whole DW all at once from a central, planned perspective (*the monolithic approach*).
- To build separate subject areas (*data marts*) whenever is needed (*the stovepipe approach*).

Our **MD** modeling approach allows the user to apply any of these perspectives or a mixing of them. Thanks to the use of the UML packages, the DW designer can define the DW gradually or all at

once. Moreover, thanks to the UML importing mechanism, the user can reutilize a concept defined in a package in other packages.

For example, regarding the **MD** model depicted in Figure A.1, on the one hand the DW designer could have modeled and implemented each one of the star schemas one by one or, on the other hand, the DW designer could have modeled the three star schemas firstly and then could have implemented all of them together.

A.2.3 Shared Dimensions

In multistar models, two or more star schemas can share some dimensions. The use of the same dimension in different «StarPackage»[1] provides several advantages:

- Creating a set of shared dimensions takes 80% of the up-front data architecture effort [65], because a single dimension can be used against multiple fact tables
- The final user is allowed to perform drill-across operations: requesting data from two or more facts in a single report.
- Sharing dimensions provides consistent definitions and data contents: it avoids the redefinition of the same concept twice and inconsistent user interfaces. Moreover, shared dimensions provide consistent information for similar queries. For example, requiring that all star schemas use the same shared time dimension enforces consistency of results summarized by time.

For example, following the example presented in Figure A.1, Inventory Snapshot Star shares some dimensions with Inventory Delivery Status Star and Inventory Transaction Star, but the last two ones do not share any dimension between them. Then, if we explore each package diagram at a second level, we can observe which dimensions are shared. For example, Figure A.2 shows the content of the «StarPackage» Inventory Snapshot Star. The «FactPackage» Inventory Snapshot Fact is represented in the middle of the figure, while the «DimensionPackage» (Product Dimension, Time Dimension, and Warehouse Dimension) are placed around the fact package.

On the other hand, Figure A.3 shows the content of the «StarPackage» Inventory Delivery Status Star; three of the dimension packages have been previously defined in the Inventory Snapshot Star, so they are imported in this package. Because of this importation, the name

[1] "*When multiple fact tables are tied to a dimension table, the fact tables should all link to that dimension table. When we use precisely the same dimension table with each of the fact tables, we say that the dimension is 'conformed' to each fact table*" [63].

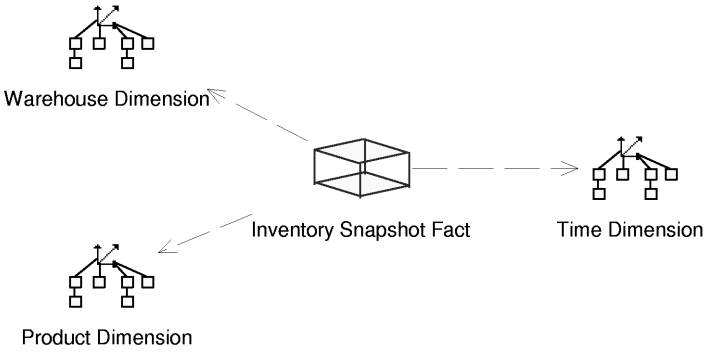


Figure A.2: Level 2 of Inventory Snapshot Star

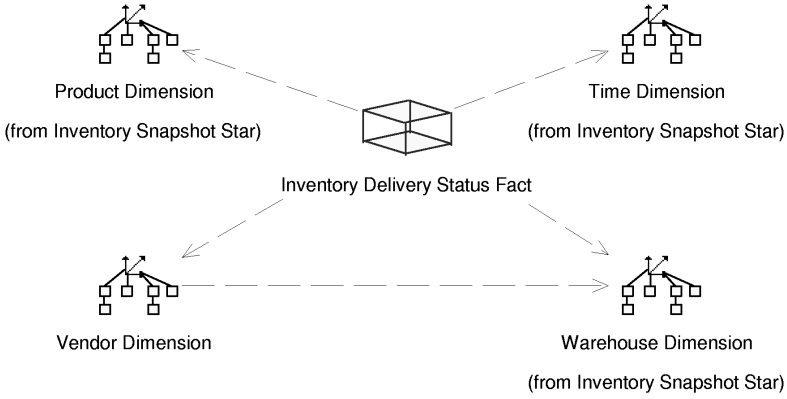


Figure A.3: Level 2 of Inventory Delivery Status Star

of the packages where they have been firstly defined appears below the package name; the name of the package also acts as a *name space*, therefore avoiding name conflicts when importing packages from different sources: it is possible to import «DimensionPackage» with the same name but defined in different «StarPackage». Moreover, a dependency has been drawn from Vendor Dimension to Warehouse Dimension because both dimensions share some hierarchy levels, as we will show in the next section.

A.2.4 Shared Hierarchy Levels

In some cases, two or more dimensions share some hierarchy levels. As in the case of shared dimensions, the use of the same levels in

different dimensions avoids redefinitions and inconsistencies in the data.

For example, Figure A.4 shows the content of the package Warehouse Dimension (from Figure A.2) and Figure A.5 shows the content of Vendor Dimension (from Figure A.3) at level 3. In a «DimensionPackage», a class is defined for the «Dimension» class (Warehouse and Vendor respectively) and one class for every classification hierarchy level (WarehouseFeatures, ZIP, City, SubRegion, SubZone, etc.). For the sake of simplicity, only the attributes of the first «Base» class have been depicted in both diagrams; we can distinguish two kinds of attributes: «Descriptor», represented by means of a D icon, and «DimensionAttribute», represented by means of a DA icon.

In this example, Warehouse and Vendor share some hierarchy levels: ZIP, City, County, and State. These levels have been firstly defined in the Warehouse Dimension; therefore, the name of the package where they have been previously defined appears below the class name (from Warehouse Dimension) in the Vendor Dimension (see Figure A.5). Moreover, both dimensions contain some hierarchy levels that do not contain the other: SubRegion and Region in the Warehouse Dimension, and SubZone and Zone in the Vendor Dimension.

In this example we also notice a salient feature of our approach: two dimensions, that share hierarchy levels, do not need to share the whole hierarchy. The package mechanism allows us to import only the required levels, thereby providing a higher level of flexibility. Moreover, we have decided to share a hierarchy for both dimensions to obtain a clearer design, although the designer may have decided not to do it if such sharing is not totally feasible.

A.2.5 Multiple and Alternative Classification Hierarchies

Defining dimension classification hierarchies is highly crucial because these classification hierarchies provide the basis for the subsequent data analysis. Thanks to the flexibility of UML association relationships, we can represent *multiple* and *alternative classification hierarchies*. On the one hand, a classification hierarchy is multiple when a dimension has two or more classification hierarchies, therefore data can be rolled-up or drilled-down along two different hierarchies at least; on the other hand, two or more classification hierarchies are alternative when they converge into the same hierarchy level.

In Figure A.5, Vendor Dimension presents a multiple classification hierarchy: (i) PersonalData, ZIP, City, County, and State, and (ii) PersonalData, SubZone, and Zone. On the other hand, Warehouse Dimension (see Figure A.4) presents an alternative classification hierarchy, because we have defined two classification hierarchies that

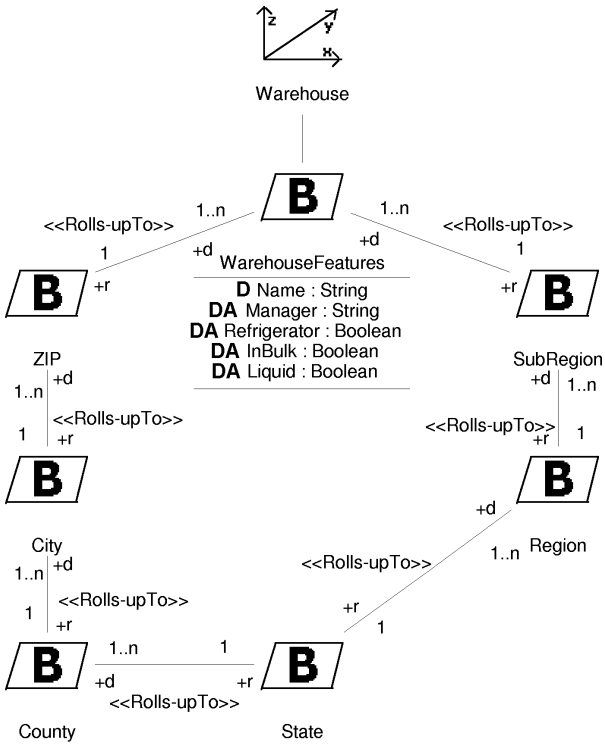


Figure A.4: Level 3 of Warehouse Dimension

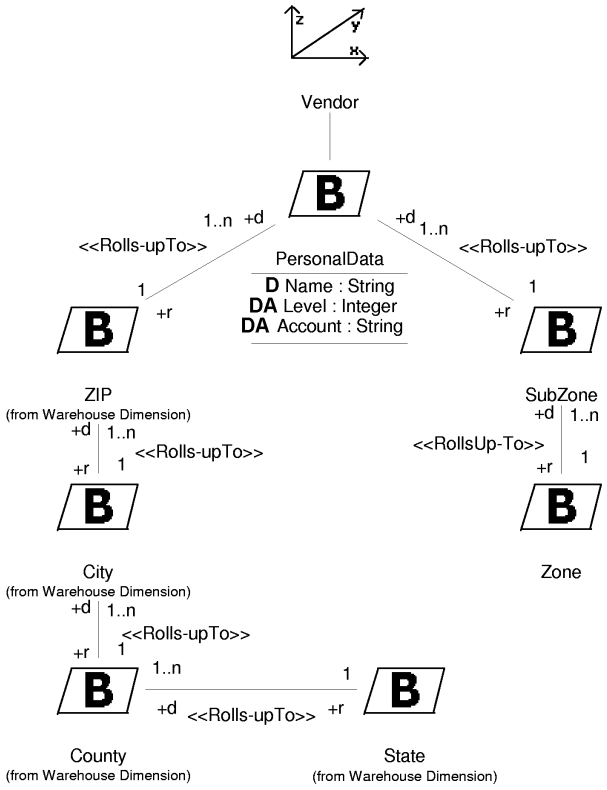


Figure A.5: Level 3 of Vendor Dimension

converge into State «Base» class.

A.2.6 Heterogeneous Dimensions

A heterogeneous dimension is a dimension that describes a large number of heterogeneous items with different attributes [63]. Our **MD** modeling approach allows the DW designer to elegantly represent heterogeneous dimensions by means of generalization-specialization hierarchies. In our approach, the different items can be grouped together in different categorization levels depending on their properties. In this way, our approach allows us to have elements at the same aggregation level that have different attributes.

For example, Figure A.6 shows the Product Dimension at level 3. The Product «Fact» has been modeled depending on the different subtypes –Liquid or Solid, Alcohol or Refreshment, etc.–, and each one of the subtypes contains particular properties –volume, weight, expiration, etc.–. For the sake of simplicity, we have omitted sone of the attributes of the «Base» classes.

A.2.7 Shared Aggregation

In our **MD** modeling approach, «Fact» classes are specified as composite classes in shared aggregation relationships of n «Dimension» classes. The flexibility of shared aggregation in the UML allows us to represent *many-to-many* relationships between «Fact» classes and particular «Dimension» classes by indicating the *1..n* cardinality on the «Dimension» class role.

For example, in Figure A.7 we can see how the «Fact» class Inventory Delivery Status has a many-to-one relationship with the «Dimension» classes Time, Vendor, Product, and Warehouse (not completely shown in the diagram). For the sake of simplicity, we have omitted all the attributes of the «Dimension» and «Base» classes. There are three shared aggregation relationships between Inventory Delivery Status and Time: Ordered, Received, and Inspected. Thanks to the use of UML named relationships, we can define more than one relationship between two classes. In this way, we can use the same «Dimension» and avoid redundancy and inconsistency problems. The «Fact» class Inventory Delivery Status contains six «FactAttribute» (represented by means of a FA icon) and two «DegenerateDimension» (represented by means of a DD icon). PO_number is the key to the purchase order header record and it is useful to the final user because it serves as the grouping key for pulling together all the products ordered on one purchase order [63].

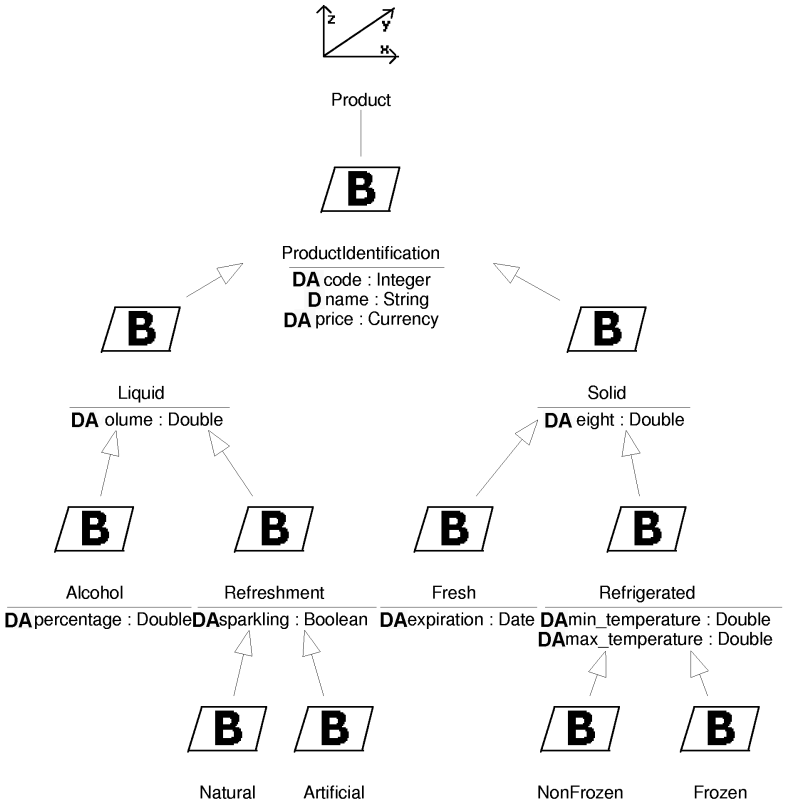


Figure A.6: Level 3 of Product Dimension

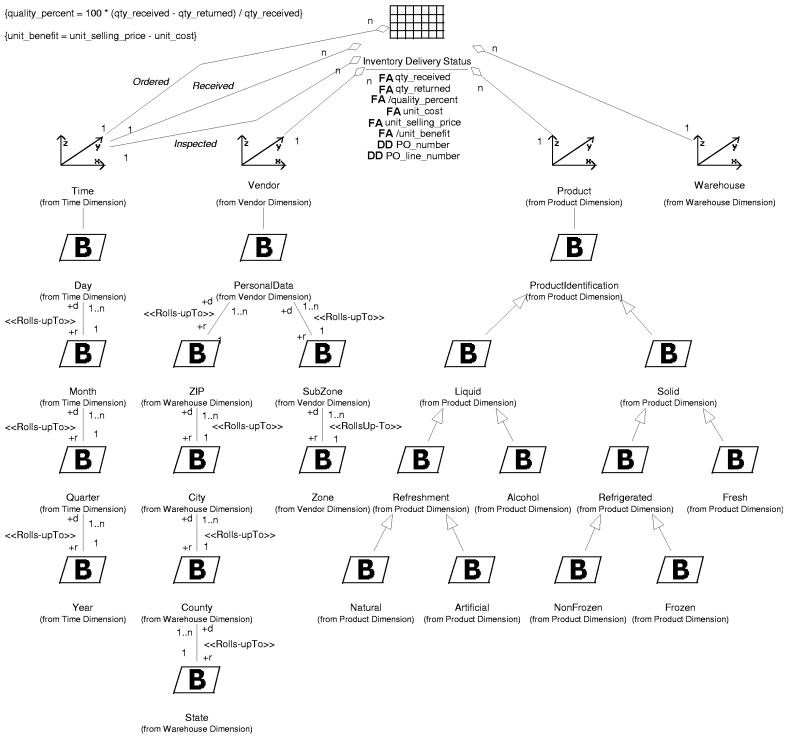


Figure A.7: Level 3 of Inventory Delivery Status Fact

A.2.8 Derivation Rules

In the **UML**, derived attributes are identified by placing / before the name of the attribute. In our **MD** modeling approach, the derivation rules are explicitly defined by means of **OCL** [97] expressions. In this way, we provide a precise and formal mechanism to define derivation rules.

For example, in Figure A.7 we can see two derivation rules for quality_percent and unit_benefit. The inclusion of the definition of the derived attributes at the conceptual design phase avoids the incorrect definition in the following phases. Moreover, the derivation rules can be used in a later implementation phase.

A.3 Conclusions

In this appendix, we have presented the main advantages of our **OO** conceptual **MD** modeling approach based on the **UML**. We have highlighted the main situations where the use of the **UML** means a considerable advantage. For example, we have exhibited how the usage of package diagrams leads to an exceptionally clean **MD** design of huge and complex systems, because package diagrams allow us to structure **MD** models at different levels of abstraction. Moreover, the importation mechanism in the **UML** simplifies the use of an element from one package in another package. In this way, we avoid the problems related to the redefinition of an element several times: redundancy, inconsistency, and ambiguity.

Appendix B

UML Particularities

In this appendix, we explain some features of **UML** that are not normally used but we widely use in this work.

Contents

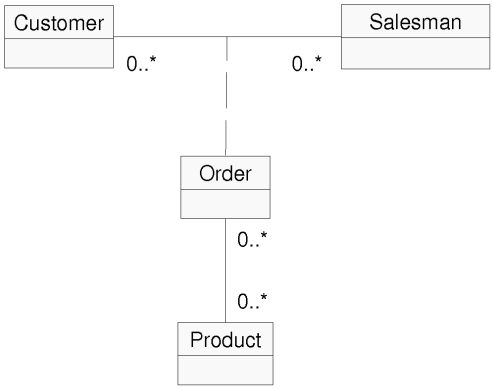


Figure B.1: Example of association class

B.1 Introduction

The *Pareto principle* (also known as the *80-20 rule*, the law of *the vital few and the trivial many*, or the *principle of factor sparsity*) states that for many phenomena 20% of something always are responsible for 80% of the results. Following this principle, designers normally use 20% of **UML** for modeling 80% of the elements in a diagram. Therefore, designers normally ignore many of the features of **UML**. Because of this, in this appendix we give some details about some **UML** modeling elements that we profusely use in our approach but that are not widely known and used.

The different concepts are presented in alphabetic order: Association classes in Section B.2, Navigability in Section B.3, Notes in Section B.4, Packages in Section B.5, and Roles in Section B.6.

B.2 Association Classes

AssociationClass: see UML (2.5.2.4, 2-21), (2.5.4.2, 2-67), (3.46, 3-77).

An association class (AssociationClass) is an association that is also a class. AssociationClass is a subclass of both Association (it connects a set of classifiers) and Class (it defines a set of features that belong to the relationship).

Even though it is drawn as an association and a class, it is really just a single model element. An association class is shown as a class symbol (rectangle) attached by a dashed line to an association path. For example, in Figure B.1 we show an association class class called Order that connects the classes Customer and Salesman; furthermore, there is also an association between the association class and Product.

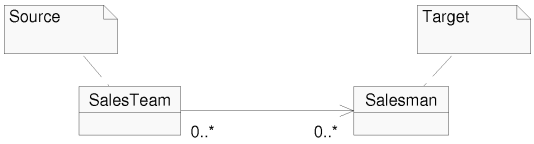


Figure B.2: Example of navigability in an association

Navigability: see UML (2.5.2.5, 2-22), (2.5.4.1, 2-65), (3.43.2.4, 3-72).

B.3 Navigability

Navigability shows that an association can be navigated by the source class and the target rolename can be used in navigation expressions. Specification of each direction across the association is independent.

Graphically, navigability is represented by means of an arrow attached to the end of an association to indicate that navigation is supported toward the class attached to the arrow.

For example, in Figure B.2, we show a **UML** class diagram where navigability is possible from SalesTeam to Salesman, but not in the opposite direction.

B.4 Notes

Note: see UML (3.11, 3-13), (3.16, 3-26).

A note is a graphical symbol containing textual information. It is a notation for rendering various kinds of textual information from the metamodel, such as constraints, comments, method bodies, and tagged values.

Graphically, a note is shown as a rectangle with a "bent corner" in the upper right corner. It contains arbitrary text. It appears on a particular diagram and may be attached to zero or more modeling elements by dashed lines. The connection between a note and the element it applies to is shown by a dashed line without an arrowhead as this is not a dependency.

A note may represent:

- A comment.
- A constraint.
- A tagged value.
- The body of a procedure of a method.
- Other string values within modeling elements.

In Figure B.3 we show a **UML** diagram that contains two notes, one of them attached to a class.

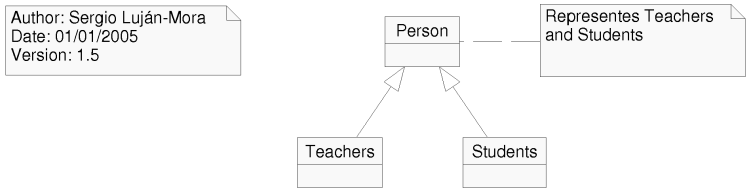


Figure B.3: Example of note

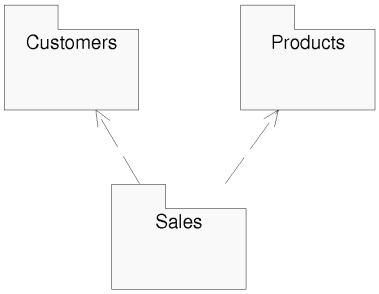


Figure B.4: Example of package

B.5 Packages

Packages: see UML (2.15.2.4, 2-184), (3.13, 3-16).

A package is a grouping of model elements. Packages themselves may be nested within other packages. A package may contain subordinate packages as well as other kinds of model elements. All kinds of UML model elements can be organized into packages.

Packages own model elements; therefore, each element can be directly owned by a single package, so the package hierarchy is a strict tree. Nevertheless, packages can reference other packages by using a dependency, so the usage network is a graph. Elements from a package can be used in another package thanks to the importing mechanism.

Importing: see UML (2.5.2.32, 2-47), (3.38, 3-62).

Graphically, a package is shown as a large rectangle with a small rectangle (a "tab") attached to the left side of the top of the large rectangle, i.e., it is the common folder icon used in different operating systems.

For example, in Figure B.4 we show a **UML** diagram with three packages and a dependency from Sales to Customers and from Sales to Products.

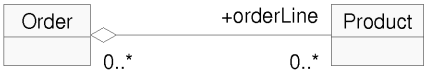


Figure B.5: Example of role

B.6 Roles

Rolename: see UML (3.43.2.6, 3-72).

A rolename is a name string near the end of an association. It indicates the role played by the class attached to the end of the association near the role.

Rolenames should be placed near the end of the association so that they are not confused with a different association. They may be placed on either side of the line (top or bottom, left or right).

In Figure B.4 we show an association between two classes with a rolename called orderLine attached to the end of the association near the Product class.

Appendix C

UML Extension Mechanisms

UML provides the Extensibility Mechanism package that allows the designer to adapt the **UML** to a particular domain, context or model. In this appendix, we summarize how to define a *UML profile* with the different elements (stereotypes, tagged values, and constraints) that are included.

Contents

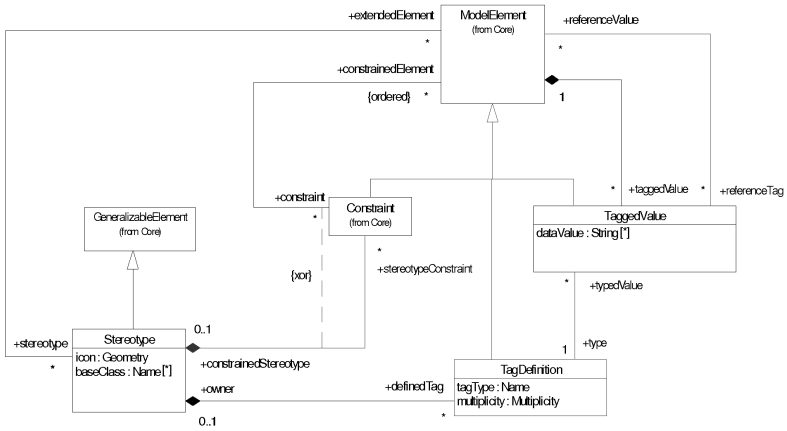


Figure C.1: Extension Mechanisms package

C.1 Introduction

UML [97] is a general modeling language (not oriented to any specific domain). **UML** provides a rich set of modeling concepts and notations that meet the needs of typical software modeling projects. However, it might happen that **UML** does not correctly adapt to some projects, and users may require additional features. In order to avoid users defining new modeling elements in an uncontrolled manner, **UML** incorporates its built-in extension mechanisms that enable new kinds of modeling elements to be added to the modeler's repertoire as well as to attach free-form information to modeling elements.

Extension Mechanisms: see UML (2.6, 2-73).

The *UML Extensibility Mechanism* package is the subpackage from the **UML** metamodel that specifies how particular **UML** model elements are customized and extended with new semantics by using *stereotypes*, *tagged values*, and *constraints*. Briefly, a stereotype defines a new building element, a tagged value specifies a new property of an existing or new element, and a constraint describes the semantics of the new elements. A coherent set of such extensions, defined for specific purposes, constitutes a *UML profile*. For example, **UML** includes a standard profile for modeling software development processes and another one for business modeling.

UML example profiles: see UML (4, 4-1).

In Figure C.1, we show the Extension Mechanisms package, which belongs to the Foundation packages of **UML**. The Extension Mechanisms package specifies how model elements are customized and extended with new semantics.

Standard Element Name	Applies to Base Element	Kind
«documentation»	Element	Tag
«metaclass»	Class	Stereotype
«metamodel»	Package	Stereotype
«profile»	Package	Stereotype
xor	Association	Constraint

Table C.1: UML Standard Elements

UML Standard Elements: see UML (Appendix A, page A-1.)

C.1.1 UML Standard Elements

UML [97] contains a list of predefined standard elements for **UML**. The standard elements are stereotypes, constraints and tagged values used in the standard but they are not considered to be part of the the specification. The names used for **UML** predefined standard elements are considered reserved words. In Table C.1, we show some of the standard elements defined in [97].

Stereotypes: see UML (2.6.2.3, 2-78), (3.18, 3-31).

C.1.2 Stereotypes

A *stereotype* is the principal extension mechanism. It is a model element that provides a way of defining virtual subclasses of **UML** metaclasses, with new metaattributes (defines additional values based on tagged values), and additional semantics (based on constraints), and optionally a new graphical representation (an icon): a stereotype allows us to attach a new semantic meaning to a model element. Different formats can be used to represent a stereotyped element (see some examples in Figure C.2):

- Icon: the base element symbol may be substituted by an icon containing the element name or with the name above or below the icon.
- Adorned element: the icon is used as part of the symbol for the base model element that the stereotype is based on. For example, in a class it is placed in the upper right corner of the rectangle that represents the class.
- Label: the stereotype name is generally placed above or in front of the name of the model element being described. The name of the stereotype is represented within matched guillemets[1].
- None.

[1] Guillemets are the quotation mark symbols used in French and certain other languages. For example, «metaclass» or «metamodel».

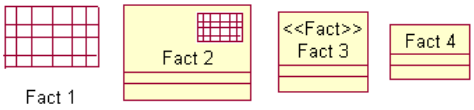


Figure C.2: Different representations for a stereotyped class

Because a stereotype is user-defined, a stereotype declaration is an element that appears at the "model" layer of the **UML** four-layer metamodeling hierarchy although it conceptually belongs in the layer above, the metamodel layer. Following the **MOF** paradigm [95], stereotypes are defined in the M2 level (metamodels), whereas the normal models designed by means of **UML** are situated in the M1 level (models), as shown in Figure C.3.

Metamodel: see UML (2.2.1, 2-4), (2.4, 2-11).

C.1.3 Tag Definitions

A *tag definition* specifies a new kind of property that may be attached to a model element. Among other things, tag definitions can be used to define the virtual metaattributes of the stereotype to which they are attached. The actual properties of individual model elements are specified using *tagged values*.

Tag definition: see UML (2.6.2.4, 2-78).

A *tagged value* allows information to be attached to any model element in conformance with its tag definition. A tagged value is rendered as a string enclosed by a pair of braces ({ }) and placed below the name of another element. A tagged value has the form name = value where name is the name of the tag definition and value is an arbitrary string that denotes its value (the real tagged value).

Tagged value: see UML (2.6.2.5, 2-79).

Element properties: see UML (3.17, 3-29).

There exists a confusion between tag definition and tagged value: a *tag definition* specifies the tagged values that can be attached to a kind of model element, whereas a *tagged value* is the actual value of a tag definition in a particular model.

In Figure C.4, we show the definition of a class with three tag definitions: author, deadline, and status.

C.1.4 Constraints

A *constraint* can be attached to any model element to refine its semantics. As it is stated in [139], "*A constraint is a restriction on one or more values of (part of) an object-oriented model or system*". In the **UML**, a constraint is rendered as a string between a pair of braces ({ }) and placed near the associated model element. A constraint on a stereotype is interpreted as a constraint on all types on which the stereotype is applied. A constraint can be defined by means of an

Constraint: see UML (2.6.2.1, 2-76), (3.16, 3-26).

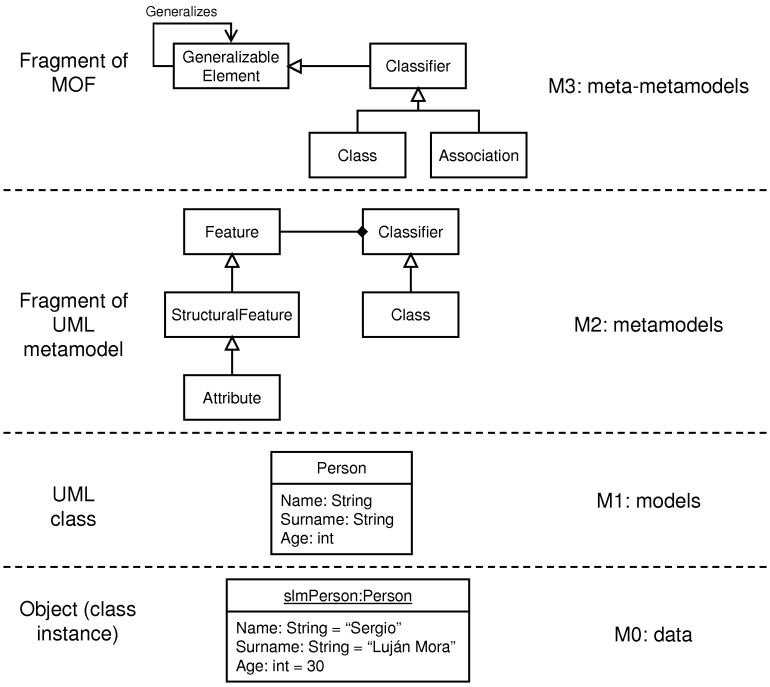


Figure C.3: MOF levels

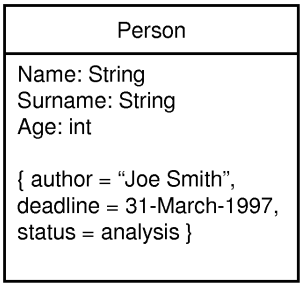


Figure C.4: A class with tagged values

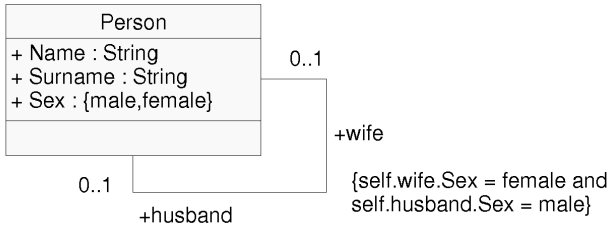


Figure C.5: UML class diagram with a constraint attached to an association

informal explanation in Natural Language and/or by means of **OCL** [97, 139] expressions. The **OCL** is a declarative language that allows software developers to write constraints over object models.

OCL: see UML chapter 6, pp. 6-1.

Depending on the number of elements affected by a constraint, the graphical representation differs:

- For a single element (a class, an association, a package,etc.), the constraint string may be placed near the element, preferably near the name of the element.
- For two elements, the constraint is shown as a dashed arrow from one element to the other element labeled by the constraint string (in braces). The direction of the arrow is relevant information within the constraint. The client (tail of the arrow) is mapped to the first position and the supplier (head of the arrow) is mapped to the second position in the constraint.
- For three or more elements, the constraint string is placed in a note symbol and attached to each of the elements by a dashed line. This notation may also be used for the other cases.

For example, in Figure C.5 we show a class with a constraint attached to an association to itself.

C.2 Profile

Profile: see UML (2.15.2.4, 2-184), (2.15.4.2, 2-193).

A profile is a stereotyped package that contains model elements (an abstraction drawn from the system being modeled) that have been customized for a specific domain or purpose using the extension mechanisms (stereotypes, tagged definitions, and constraints).

A **UML** extension defined using a profile must be strictly additive to the standard **UML** semantics. This means that such extension

must not conflict with or contradict the standard semantics. Therefore, there are restrictions on how a profile can extend the **UML** metamodel.

Appendix D

Representation of Multidimensional Models with XML

In this appendix, we present how to handle the representation, manipulation and presentation of **MD** models using **XML** and related technologies. First, we use **XML** to consider main **MD** modeling properties at the conceptual level. Then, we provide a **DTD** which allows us to directly generate valid **XML** documents that represents **MD** models. Next, an XML Schema allows us to to describe and constraint the structure of **XML** documents more accurately than **DTD**. Finally, we use ***Extensible Stylesheet Language Transformations*** **(XSLT)** to automatically generate ***HyperText Markup Language*** **(HTML)** pages from **XML** documents, thereby supporting different presentations of the same **MD** model easily.

Contents

D.1 Introduction

XML [143] is rapidly being adopted as a specific standard syntax for the exchange of semi-structured data [4]. Furthermore, **XML** is an open neutral platform and vendor independent meta-language standard, which allows to reduce the cost, complexity, and effort required in integrating data within enterprises and between enterprises. One common feature of this semi-structured data is the lack of schema, so the data is describing itself. Therefore, **XML** documents can have different structures and can represent heterogeneous kinds of data (biological, statistical, medical, etc.).

Nevertheless, **XML** documents can be associated to a **DTD** or an XML Schema [144], both of which allow the designer to describe and constraint the structure of **XML** documents. In this way, an **XML** document can be validated against these **DTD** or XML Schemas to check its correctness. Moreover, thanks to the use of XSL [142] and XSLT [141], **XML** documents can be automatically transformed into other formats, e.g. **HTML** documents. An immediate consequence is that we can define different **XSLT** stylesheets to provide different presentations of the same **XML** document.

In this appendix, to facilitate the interchange of conceptual **MD** models, we provide a **DTD** (Section D.2) and an XML Schema (Section D.3) which allow us to represent the **MD** models with **XML**. Then, we provide different presentations of the **MD** models by means of **XSLT** (Section D.4).

For more information about the *multidimensional modeling*, consult section 6.2, pp. 56.

D.2 DTD

A **DTD** provides a logic structure for **XML** documents, and restrictions that how elements can be related to each other. More specifically, a **DTD** is composed of:

- Element types and sub-element types.
- Attributes.
- Terminal strings such as `ENTITY`, `PCDATA`, and `CDATA`.
- Constraints on element and sub-element types, including "*" (set with zero or more elements), "+" (set with one or more elements), "?" (optional), and "|": (or).

Unfortunately, there is no concept of a root element of a **DTD**. Therefore, an **XML** document conforming to a **DTD** can be rooted at any element defined in the **DTD** if a root element in `DOCTYPE` is not specified.

We have defined a **DTD** that determines the correct structure and content of **XML** documents that represent **MD** models. Moreover, this **DTD** can be used to automatically validate the **XML** documents.

Our **DTD** contains 38 elements (tags). We have defined additional elements (in plural form) in order to group common elements together, so that they can be exploited to provide optimum and correct comprehension of the model, e.g. elements in plural like `PKSCHEMAS` or `DEPENDENCIES`.

The **DTD** follows the three-level structure of our **MD** approach:

- An `MDMODEL` (the root element) contains `PKSCHEMAS` (star schema packages) at level 1.
- A `PKSCHEMA` contains at most one `PKFACT` (fact package) and many `PKDIMS` (dimension packages) and `IMPPKDIMS` (imported dimensions) at level 2.
- A `PKFACT` contains at most one `FACTCLASS` and a `PKDIM` contains at most one `DIMCLASS` and many `BASECLASSES` at level 3.

```
<!ENTITY % Boolean '(true|false)'>
<!ELEMENT MDMODEL (PKSCHEMAS, DEPENDENCIES)>
<!ATTLIST MDMODEL
    id ID #REQUIRED
    name CDATA #REQUIRED>
<!ELEMENT PKSCHEMAS (PKSCHEMA*)>
<!ELEMENT PKSCHEMA (PKFACT?, PKDIMS, DEPENDENCIES)>
<!ATTLIST PKSCHEMA
    id ID #REQUIRED
    name CDATA #REQUIRED>
<!ELEMENT PKFACT (FACTCLASS)>
<!ATTLIST PKFACT
    id ID #REQUIRED
    name CDATA #REQUIRED>
<!ELEMENT PKDIMS (PKDIM*)>
<!ELEMENT PKDIM (DIMCLASS, BASECLASSES)>
<!ATTLIST PKDIM
    id ID #REQUIRED
    name CDATA #REQUIRED>
<!ELEMENT DEPENDENCIES (DEPENDENCY*)>
<!ELEMENT DEPENDENCY EMPTY>
<!ATTLIST DEPENDENCY
    id ID #REQUIRED
    start IDREF #REQUIRED
    end IDREF #REQUIRED>
<!ELEMENT FACTCLASS (FACTATTS, METHODS, SHAREDAGGS)>
<!ATTLIST FACTCLASS
    id ID #REQUIRED
    name CDATA #REQUIRED>
<!ELEMENT FACTATTS (FACTATT*)>
<!ELEMENT FACTATT EMPTY>
<!ATTLIST FACTATT
```

```
    id ID #REQUIRED
    name CDATA #REQUIRED
    derived %Boolean; "false"
    derivationRule CDATA #IMPLIED
    type CDATA #REQUIRED
    initial CDATA #IMPLIED
    DD %Boolean; "false">
<!ELEMENT METHODS (METHOD*)>
<!ELEMENT METHOD EMPTY>
<!ATTLIST METHOD
    id ID #REQUIRED
    name CDATA #REQUIRED>
<!ELEMENT DEGFACT (FACTATTS, METHODS)>
<!ATTLIST DEGFACT
    id ID #REQUIRED
    name CDATA #IMPLIED>
<!ELEMENT SHAREDAGGS (SHAREDAGG*)>
<!ELEMENT SHAREDAGG (DEGFACT?)>
<!ATTLIST SHAREDAGG
    id ID #REQUIRED
    dimclass IDREF #REQUIRED
    name CDATA #IMPLIED
    description CDATA #IMPLIED
    roleA CDATA #IMPLIED
    roleB CDATA #IMPLIED>
<!ELEMENT DIMCLASS EMPTY>
<!ELEMENT BASECLASSES (BASECLASS*)>
<!ELEMENT BASECLASS (DIMATTS, (RELATIONASOCS | RELATIONCATS)?,
METHODS)>
<!ELEMENT DIMATTS (DIMATT*)>
<!ELEMENT DIMATT EMPTY>
<!ATTLIST DIMATT
    id ID #REQUIRED
    name CDATA #REQUIRED
    derived %Boolean; "false"
    derivationRule CDATA #IMPLIED
    type CDATA #REQUIRED
    initial CDATA #IMPLIED
    OID %Boolean; "false"
    D %Boolean; "false">
<!ELEMENT RELATIONASOCS (RELATIONASOC*)>
<!ELEMENT RELATIONASOC EMPTY>
<!ATTLIST RELATIONASOC
    id ID #REQUIRED
    child IDREF #REQUIRED
    name CDATA #IMPLIED
    roleA CDATA #IMPLIED
    roleB CDATA #IMPLIED
    completeness %Boolean; "false">
<!ELEMENT RELATIONCATS (RELATIONCAT*)>
<!ELEMENT RELATIONCAT EMPTY>
<!ATTLIST RELATIONCAT
    id ID #REQUIRED
    child IDREF #REQUIRED
    name CDATA #IMPLIED>
<!ATTLIST BASECLASS
```

```
    id ID #REQUIRED
    name CDATA #REQUIRED>
<!ATTLIST DIMCLASS
    id ID #REQUIRED
    name CDATA #REQUIRED
    baseclass IDREF #IMPLIED
    isTime %Boolean; "false">
```

D.3 XML Schema

The purpose of XML Schemas is to specify the structure of instance elements together with the data type of each element/attribute. The motivation for XML Schemas is the dissatisfaction with **DTD** mainly due to their syntax and their limited data type capability, not allowing us to define new specific data types. Therefore, XML Schemas are a tremendous advancement over **DTD**: they include most basic programming types such as integer, byte, string and floating point numbers; they allow us to create enhanced data types and valid references; they are written in the same syntax as instance documents (an XML Schema is in form of a well-formed **XML** document); they can define multiple elements with the same name but different content (namespace); they can define substitutable elements, and many more features (sets, unique keys, nil content, etc.).

With respect to the structure of an XML Schema, there are two main possibilities: flat and "Russian doll" designs. The former is based on a flat catalog of all the elements available in the instance document and, for each of them, lists of child elements and attributes. We have used the later design as it allows us to define each element and attribute within its context in an embedded manner. In this sense, the representation of our XML Schema as a tree structure is illustrated in Figure D.1 and Figure D.2 for the sake of clearness and comprehension[1]. As it can be observed, we denote every node of the tree with a label. Then, every label has its correspondence with one element in the XML Schema. Following a left-right path in the tree, we clearly identify all the **MD** properties supported by our **MD** modeling approach.

Our XML Schema contains the definition of 25 elements (tags) and follows the three-level structure of our **MD** approach:

- An `MDMODEL` contains `PKSCHEMAS` (star schema packages) at level 1.

[1] In these figures we use the following notation: the box with three linked dots represents a sequence of elements, the range in which an element can occur is showed with numbers (the default minimum and maximum number of occurrences is 1) and graphically (a box with a dashed line indicates that the minimum number of occurrences is 0).

- A PKSCHEMA contains at most one PKFACT (fact package) and many PKDIM (dimension packages) grouped by a PKDIMS element at level 2.
- A PKFACT contains at most one FACTCLASS and a PKDIM contains at most one DIMCLASS and many BASECLASSES at level 3.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" attributeFormDefault="unqualified">

<xs:attributeGroup name="id_name">
<xs:annotation>
<xs:documentation>Common attributes to different elements (id y name)
</xs:documentation>
</xs:annotation>
<xs:attribute name="id" type="xs:ID" use="required"/>
<xs:attribute name="name" type="xs:string" use="required"/>
</xs:attributeGroup>

<xs:attributeGroup name="dim_fact_atts">
<xs:annotation>
<xs:documentation>Common attributes to dimension and fact classes
</xs:documentation>
</xs:annotation>
<xs:attribute name="id" type="xs:ID" use="required"/>
<xs:attribute name="name" type="xs:string" use="required"/>
<xs:attribute name="derived" type="xs:boolean" default="false"/>
<xs:attribute name="derivationRule" type="xs:string" use="optional"/>
<xs:attribute name="type" type="xs:string" use="required"/>
<xs:attribute name="initial" type="xs:string" use="optional"/>
</xs:attributeGroup>

<xs:element name="MDMODEL">
<xs:annotation>
<xs:documentation>Root element of the model</xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
<xs:element ref="PKSCHEMAS"/>
<xs:element name="DEPENDENCIES">
<xs:annotation>
<xs:documentation>Group of dependencies</xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
<xs:element name="DEPENDENCY">
<xs:annotation>
<xs:documentation>Dependency between two packages</xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:attribute name="id" type="xs:ID" use="required"/>
<xs:attribute name="start" type="xs:IDREF" use="required"/>
<xs:attribute name="end" type="xs:IDREF" use="required"/>
</xs:complexType>
```

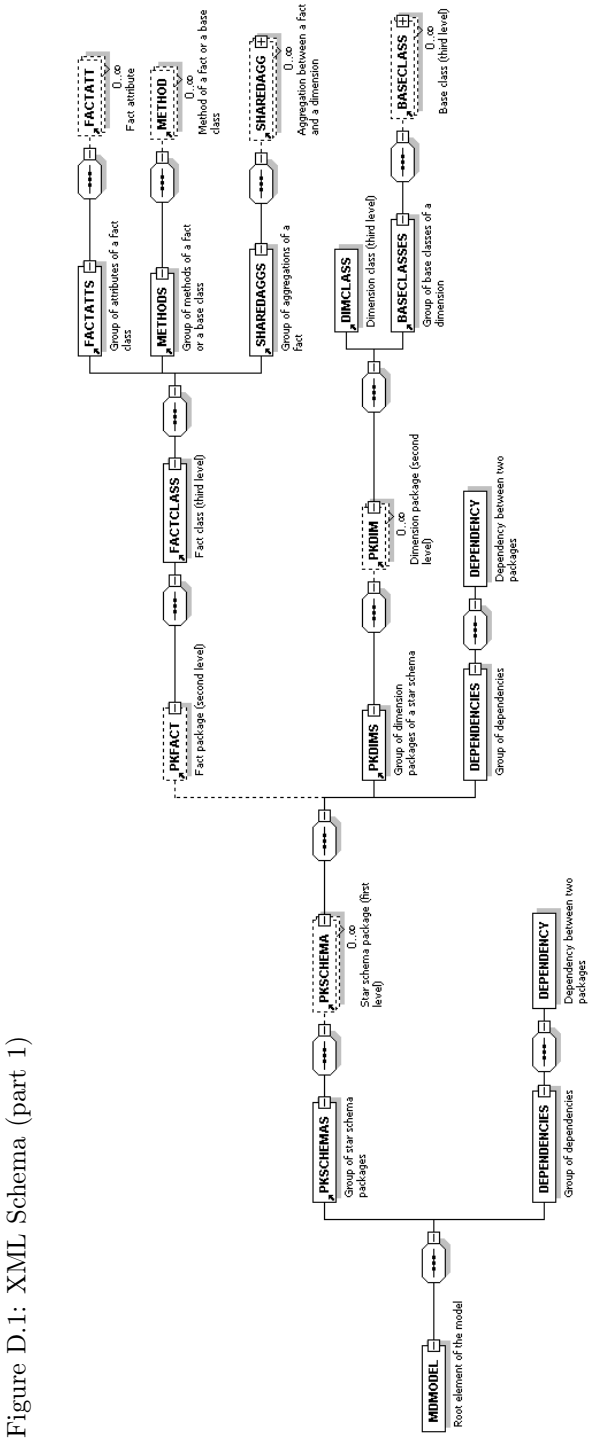


Figure D.1: XML Schema (part 1)

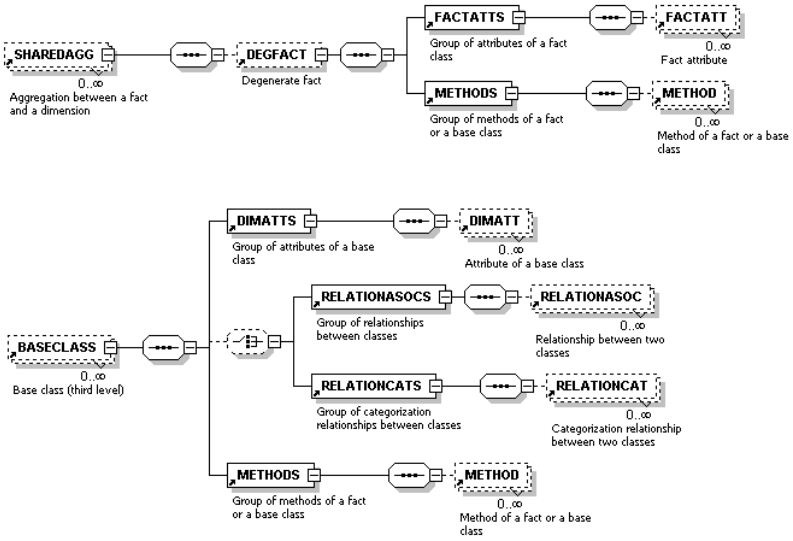


Figure D.2: XML Schema (part 2)

```
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
<xs:attributeGroup ref="id_name"/>
</xs:complexType>
<xs:key name="PKSCHEMAKey">
<xs:selector xpath="PKSCHEMAS/PKSCHEMA"/>
<xs:field xpath="@id"/>
</xs:key>
<xs:keyref name="startPKSCHEMAKey" refer="PKSCHEMAKey">
<xs:selector xpath="DEPENDENCIES/DEPENDENCY"/>
<xs:field xpath="@start"/>
</xs:keyref>
<xs:keyref name="endPKSCHEMAKey" refer="PKSCHEMAKey">
<xs:selector xpath="DEPENDENCIES/DEPENDENCY"/>
<xs:field xpath="@end"/>
</xs:keyref>
</xs:element>


<xs:element name="PKSCHEMAS">
<xs:annotation>
<xs:documentation>Group of star schema packages</xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
```

```
<xs:element ref="PKSCHEMA" minOccurs="0" maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>
</xs:element>


<xs:element name="PKSCHEMA">
<xs:annotation>
<xs:documentation>Star schema package (first level)</xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
<xs:element ref="PKFACT" minOccurs="0"/>
<xs:element ref="PKDIMS"/>
<xs:element name="DEPENDENCIES">
<xs:annotation>
<xs:documentation>Group of dependencies</xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
<xs:element name="DEPENDENCY">
<xs:annotation>
<xs:documentation>Dependency between two packages</xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:attribute name="id" type="xs:ID" use="required"/>
<xs:attribute name="start" type="xs:IDREF" use="required"/>
<xs:attribute name="end" type="xs:IDREF" use="required"/>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
<xs:attributeGroup ref="id_name"/>
</xs:complexType>
<xs:key name="DIMCLASSKey">
<xs:selector xpath="PKDIMS/PKDIM/DIMCLASS"/>
<xs:field xpath="@id"/>
</xs:key>
<xs:keyref name="sharedaggDIMCLASSKey" refer="DIMCLASSKey">
<xs:selector xpath="PKFACT/FACTCLASS/SHAREDAGGS/SHAREDAGG"/>
<xs:field xpath="@dimclass"/>
</xs:keyref>
<xs:key name="PKDIMKey">
<xs:selector xpath="PKDIMS/PKDIM"/>
<xs:field xpath="@id"/>
</xs:key>
<xs:key name="PKFACTKey">
<xs:selector xpath="PKFACT"/>
<xs:field xpath="@id"/>
</xs:key>
<xs:keyref name="startPKDIMKey" refer="PKDIMKey">
<xs:selector xpath="DEPENDENCIES/DEPENDENCY"/>
<xs:field xpath="@start"/>
</xs:keyref>
```

```
<xs:keyref name="startPKFACTKey" refer="PKFACTKey">
<xs:selector xpath="DEPENDENCIES/DEPENDENCY"/>
<xs:field xpath="@start"/>
</xs:keyref>
<xs:keyref name="endPKDIMKey" refer="PKDIMKey">
<xs:selector xpath="DEPENDENCIES/DEPENDENCY"/>
<xs:field xpath="@end"/>
</xs:keyref>
</xs:element>

<xs:element name="PKFACT">
<xs:annotation>
<xs:documentation>Fact package (second level)</xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
<xs:element ref="FACTCLASS"/>
</xs:sequence>
<xs:attributeGroup ref="id_name"/>
</xs:complexType>
</xs:element>

<xs:element name="FACTCLASS">
<xs:annotation>
<xs:documentation>Fact class (third level)</xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
<xs:element ref="FACTATTS"/>
<xs:element ref="METHODS"/>
<xs:element ref="SHAREDAGGS"/>
</xs:sequence>
<xs:attributeGroup ref="id_name"/>
</xs:complexType>
</xs:element>

<xs:element name="FACTATTS">
<xs:annotation>
<xs:documentation>Group of attributes of a fact class</xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
<xs:element ref="FACTATT" minOccurs="0" maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>
</xs:element>

<xs:element name="FACTATT">
<xs:annotation>
<xs:documentation>Fact attribute</xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:attributeGroup ref="dim_fact_atts"/>
<xs:attribute name="DD" type="xs:boolean" default="false"/>
</xs:complexType>
</xs:element>
```

```
<xs:element name="DEGFACT">
<xs:annotation>
<xs:documentation>Degenerate fact</xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
<xs:element ref="FACTATTS"/>
<xs:element ref="METHODS"/>
</xs:sequence>
<xs:attributeGroup ref="id_name"/>
</xs:complexType>
</xs:element>

<xs:element name="METHODS">
<xs:annotation>
<xs:documentation>Group of methods of a fact or a base class
</xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
<xs:element ref="METHOD" minOccurs="0" maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>
</xs:element>

<xs:element name="METHOD">
<xs:annotation>
<xs:documentation>Method of a fact or a base class</xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:attributeGroup ref="id_name"/>
</xs:complexType>
</xs:element>

<xs:element name="SHAREDAGGS">
<xs:annotation>
<xs:documentation>Group of aggregations of a fact</xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
<xs:element ref="SHAREDAGG" minOccurs="0" maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>
</xs:element>

<xs:element name="SHAREDAGG">
<xs:annotation>
<xs:documentation>Aggregation between a fact and a dimension
</xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
<xs:element ref="DEGFACT" minOccurs="0"/>
</xs:sequence>
<xs:attribute name="id" type="xs:ID" use="required"/>
```

```
<xs:attribute name="dimclass" type="xs:IDREF" use="required"/>
<xs:attribute name="name" type="xs:string" use="optional"/>
<xs:attribute name="description" type="xs:string" use="optional"/>
<xs:attribute name="roleA" type="xs:string" use="optional"/>
<xs:attribute name="roleB" type="xs:string" use="optional"/>
</xs:complexType>
</xs:element>


<xs:element name="PKDIMS">
<xs:annotation>
<xs:documentation>Group of dimension packages of a star schema
</xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
<xs:element ref="PKDIM" minOccurs="0" maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>
</xs:element>

<xs:element name="PKDIM">
<xs:annotation>
<xs:documentation>Dimension package (second level)</xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
<xs:element ref="DIMCLASS"/>
<xs:element ref="BASECLASSES"/>
</xs:sequence>
<xs:attributeGroup ref="id_name"/>
</xs:complexType>
<xs:key name="BASECLASSKey">
<xs:selector xpath="BASECLASSES/BASECLASS"/>
<xs:field xpath="@id"/>
</xs:key>
<xs:keyref name="dimclassBASECLASSKey" refer="BASECLASSKey">
<xs:selector xpath="DIMCLASS"/>
<xs:field xpath="@baseclass"/>
</xs:keyref>
<xs:keyref name="relationasocBASECLASSKey" refer="BASECLASSKey">
<xs:selector xpath="BASECLASSES/BASECLASS/RELATIONASOCS/RELATIONASOC"/>
<xs:field xpath="@child"/>
</xs:keyref>
<xs:keyref name="relationcatBASECLASSKey" refer="BASECLASSKey">
<xs:selector xpath="BASECLASSES/BASECLASS/RELATIONCATS/RELATIONCAT"/>
<xs:field xpath="@child"/>
</xs:keyref>
</xs:element>

<xs:element name="DIMCLASS">
<xs:annotation>
<xs:documentation>Dimension class (third level)</xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:attributeGroup ref="id_name"/>
```

```
<xs:attribute name="baseclass" type="xs:IDREF" use="optional"/>
<xs:attribute name="isTime" type="xs:boolean" default="false"/>
</xs:complexType>
</xs:element>

<xs:element name="BASECLASSES">
<xs:annotation>
<xs:documentation>Group of base classes of a dimension</xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
<xs:element ref="BASECLASS" minOccurs="0" maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>
</xs:element>

<xs:element name="BASECLASS">
<xs:annotation>
<xs:documentation>Base class (third level)</xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
<xs:element ref="DIMATTS"/>
<xs:choice minOccurs="0">
<xs:element ref="RELATIONASOCS"/>
<xs:element ref="RELATIONCATS"/>
</xs:choice>
<xs:element ref="METHODS"/>
</xs:sequence>
<xs:attributeGroup ref="id_name"/>
</xs:complexType>
</xs:element>

<xs:element name="DIMATTS">
<xs:annotation>
<xs:documentation>Group of attributes of a base class</xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
<xs:element ref="DIMATT" minOccurs="0" maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>
</xs:element>

<xs:element name="DIMATT">
<xs:annotation>
<xs:documentation>Attribute of a base class</xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:attributeGroup ref="id_name"/>
<xs:attribute name="derived" type="xs:boolean" default="false"/>
<xs:attribute name="derivationRule" type="xs:string" use="optional"/>
<xs:attribute name="type" type="xs:string" use="required"/>
<xs:attribute name="initial" type="xs:string" use="optional"/>
<xs:attribute name="OID" type="xs:boolean" default="false"/>
<xs:attribute name="D" type="xs:boolean" default="false"/>
```

```
</xs:complexType>
</xs:element>

<xs:element name="RELATIONASOCS">
<xs:annotation>
<xs:documentation>Group of relationships between classes
</xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
<xs:element ref="RELATIONASOC" minOccurs="0" maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>
</xs:element>

<xs:element name="RELATIONASOC">
<xs:annotation>
<xs:documentation>Relationship between two classes</xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:attribute name="ID" type="xs:ID" use="required"/>
<xs:attribute name="child" type="xs:IDREF" use="required"/>
<xs:attribute name="name" type="xs:string" use="optional"/>
<xs:attribute name="roleA" type="xs:string" use="optional"/>
<xs:attribute name="roleB" type="xs:string" use="optional"/>
<xs:attribute name="completeness" type="xs:boolean" default="false"/>
</xs:complexType>
</xs:element>

<xs:element name="RELATIONCATS">
<xs:annotation>
<xs:documentation>Group of categorization relationships between classes
</xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
<xs:element ref="RELATIONCAT" minOccurs="0" maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>
</xs:element>

<xs:element name="RELATIONCAT">
<xs:annotation>
<xs:documentation>Categorization relationship between two classes
</xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:attribute name="id" type="xs:ID" use="required"/>
<xs:attribute name="child" type="xs:IDREF" use="required"/>
<xs:attribute name="name" type="xs:string" use="optional"/>
</xs:complexType>
</xs:element>
</xs:schema>
```

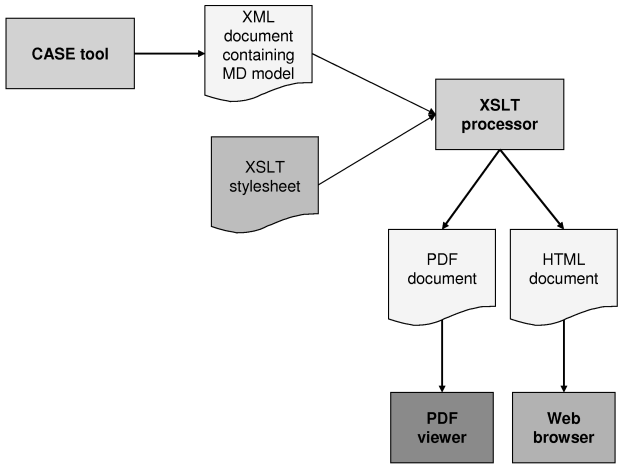


Figure D.3: Generating different presentations from the same multidimensional model

D.4 XSLT

Another relevant issue of our approach was to provide different presentations of the **MD** models in the Web. To solve this problem, **XSLT** [141] is a technology that allows us to define the presentation for **XML** documents. **XSLT** stylesheets describe a set of patterns (templates) to match both elements and attributes defined in an XML Schema, in order to apply specific transformations for each considered match. Thanks to **XSLT**, the source document can be filtered and reordered in constructing the resulting output

Figure D.3 illustrates the overall transformation process for a **MD** model. The **MD** model is stored in an **XML** document and an **XSLT** stylesheet is provided to generate different presentations of the **MD** model: e.g., **HTML**.

Below, we include the code of an **XSLT** stylesheet. We can notice that **XSLT** instructions and **HTML** tags are intermingled. The **XSLT** processor copies the **HTML** tags to the transformed document and interprets any **XSLT** instruction encountered

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="1.1"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:fo="http://www.w3.org/1999/XSL/Format">

<xsl:output method="html" encoding="iso-8859-1" indent="yes"/>

<xsl:template match="/">
```

```
<xsl:apply-templates select="//PKSCHEMA"/>
<xsl:apply-templates select="//FACTCLASS"/>
<xsl:apply-templates select="//DIMCLASS"/>
<xsl:document method="html" encoding="iso-8859-1" href="tree.html">
<xsl:apply-templates select="MDMODEL" mode="tree"/>
</xsl:document>
<xsl:document method="html" encoding="iso-8859-1" href="index.html">
<html>
<head>
<title>MDMODEL <xsl:value-of select="@name"/></title>
<meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1"/>
</head>
<frameset cols="30%,*">
<frame src="tree.html"/>
<frame src="" name="body"/>
</frameset>
</html>
</xsl:document>
</xsl:template>

<!-- ========================== Tree ========================== -->

<xsl:template match="MDMODEL" mode="tree">
<html>
<head>
<style type="text/css">
body { margin: 8; scrollbar-track-color: #FFFFFF;  scrollbar-face-color:
#006699; scrollbar-arrow-color: #FFFFFF; scrollbar-hightlight-color: ;
scrollbar-shadow-color: #FFFFFF; scrollbar-3dlight-color: #006699;
scrollbar-darkshadow-color: #006699 }
pre { font: 8pt Verdana; display: inline }
td.linea { background: url(images/elmLinea.gif) repeat-y left }
td.texto { padding-left: 2px }
a { color: #0000AA; text-decoration: none }
a:visited { color: #0000AA }
a:hover { color: #0099FF }
a.rama { cursor: default }
img { border: 0 }
</style>
</head>
<body>
<table border="0" cellspacing="0" cellpadding="0">
<tr>
<td><img src="images/package.gif" valign="middle"/></td>
<td class="texto"><pre>Schemas</pre></td>
</tr>
</table>
<table border="0" cellspacing="0" cellpadding="0">
<tr>
<td>
<img src="images/blanco.gif" width="3" height="16"/>
</td>
<td>
<xsl:apply-templates select="PKSCHEMAS/PKSCHEMA" mode="tree"/>
</td>
```

```
</tr>
</table>
</body>
</html>
</xsl:template>

<xsl:template match="PKSCHEMA" mode="tree">
<table border="0" cellspacing="0" cellpadding="0">
<tr>
<td>
<xsl:choose>
<xsl:when test="count(following-sibling::PKSCHEMA)>0">
<img src="images/elmInter.gif" valign="middle"/>
</xsl:when>
<xsl:otherwise>
<img src="images/elmFinal.gif" valign="middle"/>
</xsl:otherwise>
</xsl:choose>
<img src="images/package.gif" valign="middle"/>
</td>
<td class="texto"><a href="{generate-id()}.html" target="body"><pre>
<xsl:value-of select="@name"/></pre></a></td>
</tr>
</table>
<table border="0" cellspacing="0" cellpadding="0">
<tr>
<xsl:choose>
<xsl:when test="count(following-sibling::PKSCHEMA)>0">
<td class="linea">
<img src="images/blanco.gif" width="17" height="16"/>
</td>
</xsl:when>
<xsl:otherwise>
<td>
<img src="images/blanco.gif" width="17" height="16"/>
</td>
</xsl:otherwise>
</xsl:choose>
<td>
<xsl:apply-templates select="PKFACT/FACTCLASS" mode="tree"/>
<xsl:if test="PKDIMS/PKDIM">
<xsl:apply-templates select="PKDIMS" mode="tree"/>
</xsl:if>
</td>
</tr>
</table>
</xsl:template>

<xsl:template match="FACTCLASS" mode="tree">
<table border="0" cellspacing="0" cellpadding="0">
<tr>
<td>
<xsl:choose>
<xsl:when test="count(../../PKDIMS/PKDIM)>0">
<img src="images/elmInter.gif" valign="middle"/>
</xsl:when>
```

```
<xsl:otherwise>
<img src="images/elmFinal.gif" valign="middle"/>
</xsl:otherwise>
</xsl:choose>
<img src="images/class.gif" valign="middle"/>
</td>
<td class="texto"><a href="{generate-id()}.html" target="body"><pre>
<xsl:value-of select="@name"/></pre></a></td>
</tr>
</table>
</xsl:template>

<xsl:template match="PKDIMS" mode="tree">
<table border="0" cellspacing="0" cellpadding="0">
<tr>
<td>
<img src="images/elmFinal.gif" valign="middle"/>
<img src="images/package.gif" valign="middle"/>
</td>
<td class="texto"><pre>Dimensions</pre></td>
</tr>
</table>
<table border="0" cellspacing="0" cellpadding="0">
<tr>
<td><img src="images/blanco.gif" width="17" height="16"/></td>
<td>
<xsl:apply-templates select="PKDIM" mode="tree"/>
</td>
</tr>
</table>
</xsl:template>

<xsl:template match="PKDIM" mode="tree">
<table border="0" cellspacing="0" cellpadding="0">
<tr>
<td>
<xsl:choose>
<xsl:when test="count(following-sibling::PKDIM)>0">
<img src="images/elmInter.gif" valign="middle"/>
</xsl:when>
<xsl:otherwise>
<img src="images/elmFinal.gif" valign="middle"/>
</xsl:otherwise>
</xsl:choose>
<img src="images/class.gif" valign="middle"/>
</td>
<td class="texto"><a href="{generate-id(DIMCLASS)}.html" target="body">
<pre><xsl:value-of select="DIMCLASS/@name"/></pre></a></td>
</tr>
</table>
</xsl:template>

<!-- ========================== Schemas ========================== -->

<xsl:template match="PKSCHEMA">
<xsl:document method="html" encoding="iso-8859-1"
```

```
href="{generate-id()}.html">
<html>
<head>
<title>GOLD Model</title>
<style type="text/css">
body { font-family: Tahoma,Arial,Verdana,sans-serif; font-size: 14px;
color: #3D5066; }
table { font-size: 12px; margin-top: 2px; margin-bottom: 5px;
background: #DBE0E5; border: 1px solid #A1ACB9; }
table.data { font-size: 12px; border: none; }
th { font-size: 14px; font-weight: bold; text-align: left;
color: #3D5066; }
th.title { font-size: 20px; font-weight: bold; text-align: center;
color: #3D5066; background: #A1ACB9; }
th.name { font-size: 12px; font-weight: bold; text-align: left;
background: #F4F6F7; }
td.data { background: #F4F6F7; }
td.name { background: #F4F6F7; }
td.value { color: #005CB1; background: #F4F6F7; }
a { color: #005CB1; }
</style>
</head>
<body>
<div align="center"><a href="javascript:history.go(-1)">Back</a></div>
<br/>
<table cellpadding="4" cellspacing="0" border="0" align="center">
<tr>
<th class="title"><xsl:value-of select="@name"/></th>
</tr>
<tr>
<th>General information</th>
</tr>
<tr>
<td class="data">
<table class="data" cellpadding="2" cellspacing="0" border="0">
<tr>
<td class="name"><xsl:text disable-output-escaping="yes">
&amp;nbsp;&amp;nbsp;</xsl:text>Description:</td>
<td class="value"><xsl:value-of select="@name"/></td>
</tr>
</table>
</td>
</tr>
<tr>
<th>Fact classes</th>
</tr>
<xsl:if test="PKFACT">
<tr>
<td class="name"><xsl:text disable-output-escaping="yes">
&amp;nbsp;&amp;nbsp;</xsl:text>
<a href="{generate-id(PKFACT/FACTCLASS)}.html">
<xsl:value-of select="PKFACT/FACTCLASS/@name"/></a></td>
</tr>
</xsl:if>
<tr>
<th>Dimension classes</th>
```

```
</tr>
<xsl:for-each select="PKDIMS/PKDIM">
<tr>
<td class="name"><xsl:text disable-output-escaping="yes">
&amp;nbsp;&amp;nbsp;</xsl:text>
<a href="{generate-id(DIMCLASS)}.html">
<xsl:value-of select="DIMCLASS/@name"/></a></td>
</tr>
</xsl:for-each>
</table>
</body>
</html>
</xsl:document>
</xsl:template>

<!-- ========================== Facts ========================== -->

<xsl:template match="FACTCLASS">
<xsl:document method="html" encoding="iso-8859-1"
href="{generate-id()}.html">
<html>
<head>
<title>Fact class: <xsl:value-of select="@name"/></title>
<style type="text/css">
body { font-family: Tahoma,Arial,Verdana,sans-serif; font-size: 14px;
color: #3D5066; }
table { font-size: 12px; margin-top: 2px; margin-bottom: 5px;
background: #DBE0E5; border: 1px solid #A1ACB9; }
table.data { font-size: 12px; border: none; }
th { font-size: 14px; font-weight: bold; text-align: left;
color: #3D5066; }
th.title { font-size: 20px; font-weight: bold; text-align: center;
color: #3D5066; background: #A1ACB9; }
th.name { font-size: 12px; font-weight: bold; text-align: left;
background: #F4F6F7; }
td.data { background: #F4F6F7; }
td.name { background: #F4F6F7; }
td.value { color: #005CB1; background: #F4F6F7; }
a { color: #005CB1; }
</style>
</head>
<body>
<div align="center"><a href="javascript:history.go(-1)">Back</a></div>
<br/>
<table cellpadding="2" cellspacing="2" border="0" align="center">
<tr>
<th class="title"><xsl:value-of select="@name"/></th>
</tr>
<tr>
<th>General information</th>
</tr>
<tr>
<td class="data">
<table class="data" cellpadding="2" cellspacing="0" border="0">
<tr>
<td class="name"><xsl:text disable-output-escaping="yes">
```

```
&amp;nbsp;&amp;nbsp;</xsl:text>Caption:</td>
<td class="value"><xsl:value-of select="@name"/></td>
</tr>
</table>
</td>
</tr>
<tr>
<th>Measures</th>
</tr>
<xsl:if test="FACTATTS/FACTATT">
<tr>
<td class="data">
<table class="data" cellpadding="2" cellspacing="2"
border="0" width="100%">
<tr>
<th class="name">Name</th>
<th class="name">Type</th>
<th class="name">Initial</th>
<th class="name">Derivation Rule</th>
<th class="name">DD</th>
</tr>
<xsl:for-each select="FACTATTS/FACTATT">
<tr>
<td class="value"><xsl:value-of select="@name"/></td>
<td class="value"><xsl:value-of select="@type"/></td>
<td class="value"><xsl:value-of select="@initial"/></td>
<td class="value"><xsl:value-of select="@derivationRule"/></td>
<td class="value">
<xsl:choose>
<xsl:when test="@DD"><xsl:value-of select="@DD"/></xsl:when>
<xsl:otherwise>false</xsl:otherwise>
</xsl:choose>
</td>
</tr>
</xsl:for-each>
</table>
</td>
</tr>
</xsl:if>
<tr>
<th>Methods</th>
</tr>
<xsl:if test="METHODS/METHOD">
<tr>
<td class="data">
<table class="data" cellpadding="2" cellspacing="2" border="0"
width="100%">
<tr>
<th class="name">Name</th>
</tr>
<xsl:for-each select="METHODS/METHOD">
<tr>
<td class="value"><xsl:value-of select="@name"/></td>
</tr>
</xsl:for-each>
</table>
```

```
</td>
</tr>
</xsl:if>
<tr>
<th>Shared aggregations</th>
</tr>
<xsl:if test="SHAREDAGGS/SHAREDAGG">
<tr>
<td class="data">
<table class="data" cellpadding="2" cellspacing="2" border="0"
width="100%">
<tr>
<th class="name">Name</th>
<th class="name">Description</th>
<th class="name">Role A</th>
<th class="name">Role B</th>
</tr>
<xsl:for-each select="SHAREDAGGS/SHAREDAGG">
<tr>
<xsl:variable name="dimclass" select="@dimclass"/>
<td class="value"><a href="{generate-id(../../../../PKDIMS/PKDIM/
DIMCLASS[@id=$dimclass])}.html">
<xsl:value-of select="../../../../PKDIMS/PKDIM/
DIMCLASS[@id=$dimclass]/@name"/></a></td>
<td class="value"><xsl:value-of select="@description"/></td>
<td class="value"><xsl:value-of select="@roleA"/></td>
<td class="value"><xsl:value-of select="@roleB"/></td>
</tr>
</xsl:for-each>
</table>
</td>
</tr>
</xsl:if>
</table>
</body>
</html>
</xsl:document>
</xsl:template>

<!-- ======================== Dimensions ======================== -->

<xsl:template match="DIMCLASS">
<xsl:document method="html" encoding="iso-8859-1"
href="{generate-id()}.html">
<html>
<head>
<title>Fact class: <xsl:value-of select="@name"/></title>
<style type="text/css">
body { font-family: Tahoma,Arial,Verdana,sans-serif; font-size: 14px;
color: #3D5066; }
table { font-size: 12px; margin-top: 2px; margin-bottom: 5px;
background: #DBE0E5; border: 1px solid #A1ACB9; }
table.data { font-size: 12px; border: none; }
th { font-size: 14px; font-weight: bold; text-align: left;
color: #3D5066; }
th.title { font-size: 20px; font-weight: bold; text-align: center;
```

```
color: #3D5066; background: #A1ACB9; }
th.name { font-size: 12px; font-weight: bold; text-align: left;
background: #F4F6F7; }
td.data { background: #F4F6F7; }
td.name { background: #F4F6F7; }
td.value { color: #005CB1; background: #F4F6F7; }
a { color: #005CB1; }
</style>
</head>
<body>
<div align="center"><a href="javascript:history.go(-1)">Back</a></div>
<br/>
<table cellpadding="2" cellspacing="2" border="0" align="center">
<tr>
<th class="title"><xsl:value-of select="@name"/></th>
</tr>
<tr>
<th>General information</th>
</tr>
<tr>
<td class="data">
<table class="data" cellpadding="2" cellspacing="0" border="0">
<tr>
<td class="name"><xsl:text disable-output-escaping="yes">
&amp;amp;nbsp;&amp;amp;nbsp;</xsl:text>Caption:</td>
<td class="value"><xsl:value-of select="@name"/></td>
</tr>
<tr>
<td class="name"><xsl:text disable-output-escaping="yes">
&amp;amp;nbsp;&amp;amp;nbsp;</xsl:text>Is time?:</td>
<td class="value">
<xsl:choose>
<xsl:when test="@isTime"><xsl:value-of select="@isTime"/></xsl:when>
<xsl:otherwise>false</xsl:otherwise>
</xsl:choose>
</td>
</tr>
</table>
</td>
</tr>
<tr>
<th>Attributes</th>
</tr>
<xsl:variable name="baseclass" select="@baseclass"/>
<xsl:if test="../BASECLASSES/BASECLASS[@id=$baseclass]/DIMATTS/DIMATT">
<tr>
<td class="data">
<table class="data" cellpadding="2" cellspacing="2" border="0"
width="100%">
<tr>
<th class="name">Name</th>
<th class="name">Type</th>
<th class="name">Initial</th>
<th class="name">Derivation Rule</th>
<th class="name">OID</th>
<th class="name">Descriptor</th>
```

```
</tr>
<xsl:for-each select="../BASECLASSES/BASECLASS[@id=$baseclass]/
DIMATTS/DIMATT">
<tr>
<td class="value"><xsl:value-of select="@name"/></td>
<td class="value"><xsl:value-of select="@type"/></td>
<td class="value"><xsl:value-of select="@initial"/></td>
<td class="value"><xsl:value-of select="@derivationRule"/></td>
<td class="value">
<xsl:choose>
<xsl:when test="@OID"><xsl:value-of select="@OID"/></xsl:when>
<xsl:otherwise>false</xsl:otherwise>
</xsl:choose>
</td>
<td class="value">
<xsl:choose>
<xsl:when test="@D"><xsl:value-of select="@D"/></xsl:when>
<xsl:otherwise>false</xsl:otherwise>
</xsl:choose>
</td>
</tr>
</xsl:for-each>
</table>
</td>
</tr>
</xsl:if>
<tr>
<th>Methods</th>
</tr>
<xsl:if test="METHODS/METHOD">
<tr>
<td class="data">
<table class="data" cellpadding="2" cellspacing="2" border="0"
width="100%">
<tr>
<th class="name">Name</th>
</tr>
<xsl:for-each select="METHODS/METHOD">
<tr>
<td class="value"><xsl:value-of select="@name"/></td>
</tr>
</xsl:for-each>
</table>
</td>
</tr>
</xsl:if>
<tr>
<th>Association levels</th>
</tr>
<xsl:if test="../BASECLASSES/BASECLASS[@id=$baseclass]/RELATIONASOCS/
RELATIONASOC">
<tr>
<td class="data">
<table class="data" cellpadding="2" cellspacing="2" border="0"
width="100%">
<tr>
```

```
<th class="name">Name</th>
<th class="name">Completeness</th>
<th class="name">Role A</th>
<th class="name">Role B</th>
</tr>
<xsl:for-each select="../BASECLASSES/BASECLASS[@id=$baseclass]/
RELATIONASOCS/RELATIONASOC">
<tr>
<xsl:variable name="child" select="@child"/>
<!--<td class="value"><a href="{generate-id(../../../../PKDIMS/PKDIM/
DIMCLASS[@id=$dimclass])}.html"><xsl:value-of select="../../../../
PKDIMS/PKDIM/DIMCLASS[@id=$dimclass]/@name"/></a></td>-->
<td class="value"><xsl:value-of select="../../../
BASECLASS[@id=$child]/@name"/>
</td>
<td class="value">
<xsl:choose>
<xsl:when test="@completeness"><xsl:value-of select="@completeness"/>
</xsl:when>
<xsl:otherwise>false</xsl:otherwise>
</xsl:choose>
</td>
<td class="value"><xsl:value-of select="@roleA"/></td>
<td class="value"><xsl:value-of select="@roleB"/></td>
</tr>
</xsl:for-each>
</table>
</td>
</tr>
</xsl:if>
<tr>
<th>Categorization levels</th>
</tr>
<xsl:if test="../BASECLASSES/BASECLASS[@id=$baseclass]/RELATIONCATS/
RELATIONCAT">
<tr>
<td class="data">
<table class="data" cellpadding="2" cellspacing="2" border="0"
width="100%">
<tr>
<th class="name">Name</th>
</tr>
<xsl:for-each select="../BASECLASSES/BASECLASS[@id=$baseclass]/
RELATIONCATS/RELATIONCAT">
<tr>
<xsl:variable name="child" select="@child"/>
<!--<td class="value"><a href="{generate-id(../../../../PKDIMS/PKDIM/
DIMCLASS[@id=$dimclass])}.html"><xsl:value-of select="../../../../
PKDIMS/PKDIM/DIMCLASS[@id=$dimclass]/@name"/></a></td>-->
<td class="value"><xsl:value-of select="../../../
BASECLASS[@id=$child]/@name"/>
</td>
</tr>
</xsl:for-each>
</table>
</td>
```

```
</tr>
</xsl:if>
</table>
</body>
</html>
</xsl:document>
</xsl:template>

</xsl:stylesheet>
```

Appendix E

Definition of an Add-in for Rational Rose

In this appendix, we present the **REI** that allows the user to extend Rational Rose's capabilities. Moreover, we explain how to define an add-in for this **CASE** tool and we show our add-in for **MD** modeling.

Contents

E.1 Introduction

Instead of creating our own **CASE** tool to support our **MD** modeling approach, we have chosen to extend a well-known **CASE** tool available in the market, such as Rational Rose [102]. In this way, we believe that our contribution can reach a greater number of people.

Rational Rose is one of the most well-known visual modeling tools. Rational Rose is extensible by means of add-ins, which allows the user to package customizations and automation of several Rational Rose features through the **REI** [107] into one component.

Rational Rose provides several ways to extend and customize its capabilities to meet specific software development needs. In particular, Rational Rose allows the user to [107]:

- Customize Rational Rose menus.
- Automate manual Rational Rose functions with Rational Rose Scripts.
- Execute Rational Rose functions from within another application by using the Rational Rose Automation object.
- Access Rational Rose classes, properties and methods right within a software development environment by including the Rational Rose Extensibility Type Library in the project.
- Activate Rational Rose add-ins using the Add-In Manager.

An add-in is basically a collection of some combination of the following: main menu items, shortcut menu items, custom specifications, properties (**UML** tagged values), data types, **UML** stereotypes, online help, context-sensitive help, and event handling. In this appendix, we show how to define an add-in for Rational Rose[1] that allows the designer to achieve a **MD** model of a **DW** based on our **UML** profile.

For more information about the *multidimensional profile*, consult chapter 6, pp. 53.

The rest of this appendix is structured as follows. In Section E.2, we briefly describe the **REI**. Then, in Section E.3, we show how to apply our **MD** profile in Rational Rose. Finally, in Section E.4, we present our add-in and we include some parts of the implementation of the add-in.

E.2 Rational Rose Extensibility Interface

The **REI** Model is essentially a metamodel of a Rational Rose model, exposing the packages, classes, properties and methods that define

[1] We have tested our add-in with Rational Rose versions 2002 and 2003.

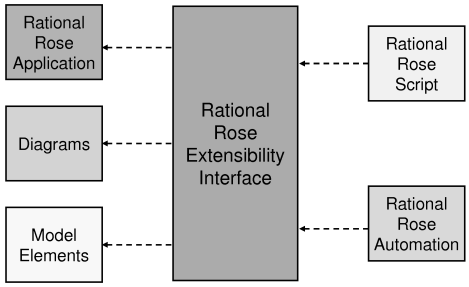


Figure E.1: Rose Application and Extensibility Components

and control the Rational Rose application and all of its functions. Figure E.1 shows the components of Rose and the **REI**, and illustrates the relationships between them. These components are:

- Rose Application: The Rose Extensibility objects that interface to Rose's application functionality.
- **REI**: This is the common set of interfaces used by Rose Script and Rose Automation to access Rose.
- Rose Script: The set of Rose Script objects that allow Rose Scripts to automate Rose functionality.
- Rose Automation: The set of Rose Automation objects that allow Rose to function as an ***Object Linking and Embedding*** **(OLE)** automation controller or server.
- Diagrams: The Rose Extensibility objects that interface to Rose's diagrams and views.
- Model Elements: The Rose Extensibility objects that interface to Rose's model elements.

Unfortunately, Rational Rose does not include and **OCL** editor/parser, but there are some products from third parties, such as *EmPowerTecs OCL-AddIn*[2] that offers support for **OCL**.

E.3 Using Multidimensional Modeling in Rational Rose

We have developed an add-in, which allows us to use our **MD** modeling approach in Rational Rose. Therefore, we can use this tool to

[2]Available in http://www.empowertec.de/products/rational-rose-ocl.htm.

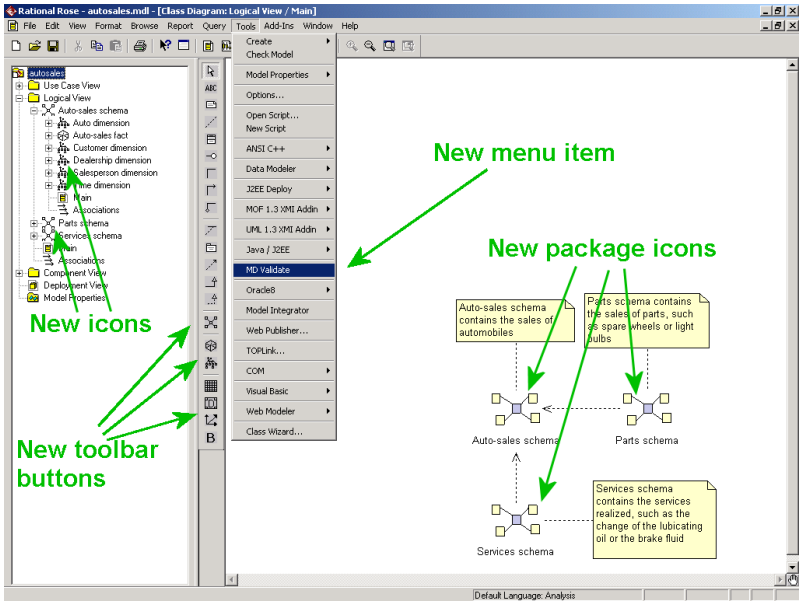


Figure E.2: A screenshot from Rational Rose: level 1 of a multidimensional model

easily accomplish **MD** conceptual models. Our add-in customizes the following elements:

- Stereotypes: We have defined the stereotypes by means of a stereotype configuration file.
- Properties: We have defined the tagged values by means of a property configuration file.
- Menu item: We have added the new menu item MD Validate in the menu Tools by means of a menu configuration file. This menu item runs a Rose script that validates a **MD** model: our script checks all the constraints that our **UML** profile defines.

In Figure E.2, we can see a screenshot from Rational Rose that shows the first level of the running example used in Section 6.3. Some comments have been added to the screenshot in order to remark some important elements: the new toolbar buttons, the new package icons, the new icons shown in the model browser, and the new menu item called MD Validate that checks the correctness of a **MD** model.

In Figure E.3, we can see a second screenshot from Rational Rose that shows the content of a star package (level 2). From the model

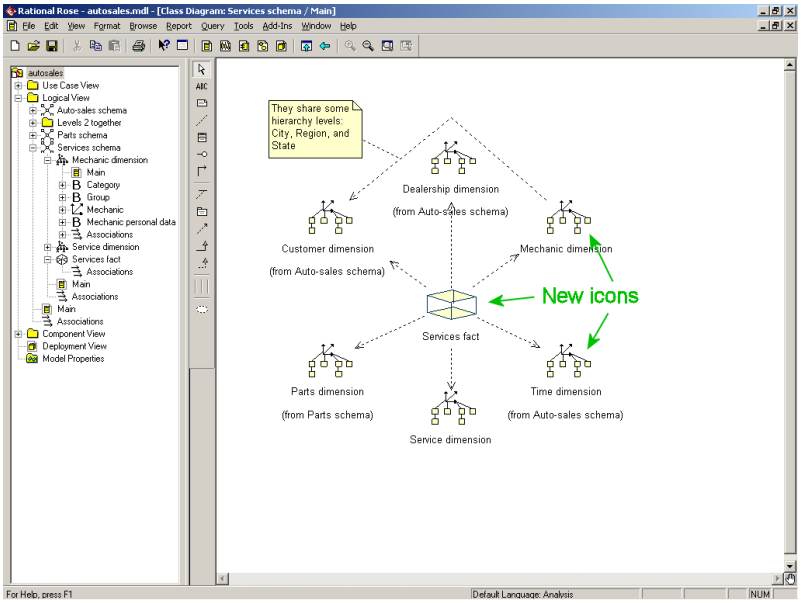


Figure E.3: A screenshot from Rational Rose: level 2 of a multidimensional model

shown in Figure E.2, Services schema has been exploded and the six dimension packages and the only fact package that compose the star package are shown. The legend (from ...) that appears below the name of some dimension packages indicates in which star package the dimension has been defined firstly. For example, Customer dimension has been defined in Auto-sales schema.

In Figure E.4, we can see a third screenshot from Rational Rose that shows the definition of a dimension (level 3). Some comments have been added to the screenshot in order to remark some important elements: the hierarchy structure of our proposal (Level 1, Level 2, and Level 3), the new buttons added to Rational Rose, the stereotyped attributes, and the different ways of displaying a stereotype (Icon, Decoration, and Label). In this screenshot, the content of Mechanic dimension from Services schema is shown (see Figure E.3); this dimension shares some hierarchy levels (City, Region, and State) with Customer dimension, the place where the shared hierarchy levels have been defined firstly (see Figure 6.8), and because of this, they are imported into this dimension.

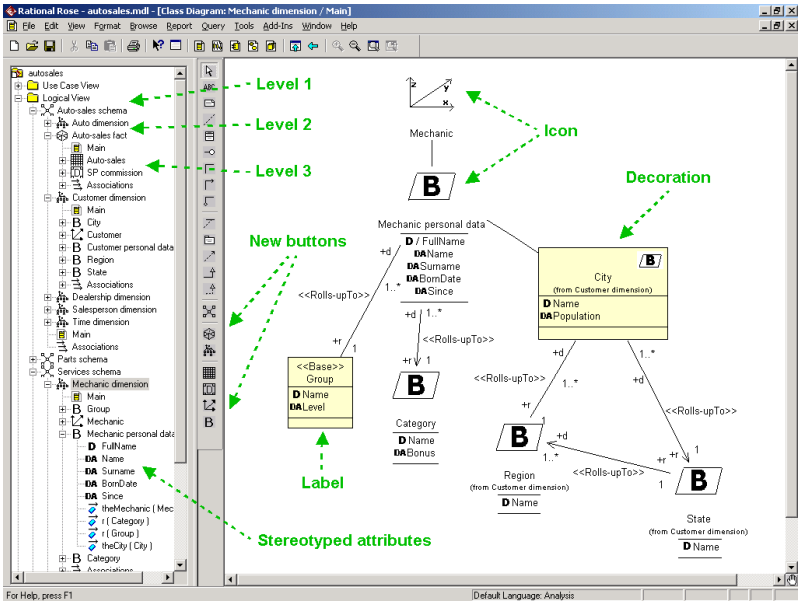


Figure E.4: A screenshot from Rational Rose: level 3 of a multidimensional model

E.4 Add-in Implementation

Our add-in is composed of five definition files and 33 graphic files for the different icons of the stereotypes. Briefly, the five definition files are:

- `mdm.reg`: installs the add-in in a system.
- `mdm.ini`: describes the new stereotypes.
- `mdm.pty`: defines new properties.
- `mdm.mnu`: configures the new menu items that are added to the Rational Rose menu.
- `mdvalidate.ebs`: contains the code that is executed.

E.4.1 Register

`mdm.reg` is the register file that install the add-in in a system. Once an add-in has been installed, the Add-In Manager allows the user to active or desactive it, as we can see in Figure E.5.

The register file updates the Microsoft Windows registry:

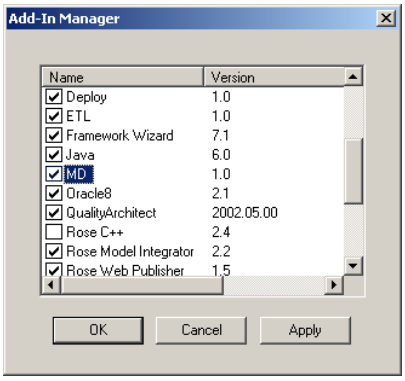


Figure E.5: Add-In Manager in Rational Rose

- Create a registry subkey for the add-in.
- Populate this subkey with the appropriate names and values (for example, `InstallDir`, `MenuFile`, etc.)
- Add the stereotype configuration file name (.ini) to the subkey called `StereotypeCfgFile`.

```
REGEDIT4

[HKEY_LOCAL_MACHINE\SOFTWARE\Rational Software\Rose\AddIns\MD]
"Active"="Yes"
"Company"="SLM"
"Copyright"="Copyright © 2002 Sergio Luján Mora"
"InstallDir"="C:\\Archivos de programa\\Rational\\Rose\\MD"
"LanguageAddIn"="No"
"MenuFile"="mdm.mnu"
"PropertyFile"="mdm.pty"
"StereotypeCfgFile"="mdm.ini"
"ToolDisplayName"="MD Modeling"
"ToolName"="MD Modeling"
"Version"="1.0"
```

In Figure E.6, we show part of the Microsoft Windows registry after our add-in has been registered.

E.4.2 Configuration File

`mdm.ini` is the configuration file that customizes the new stereotypes with the corresponding icons.

```
[General]
ConfigurationName=MDModeling
IsLanguageConfiguration=No
```

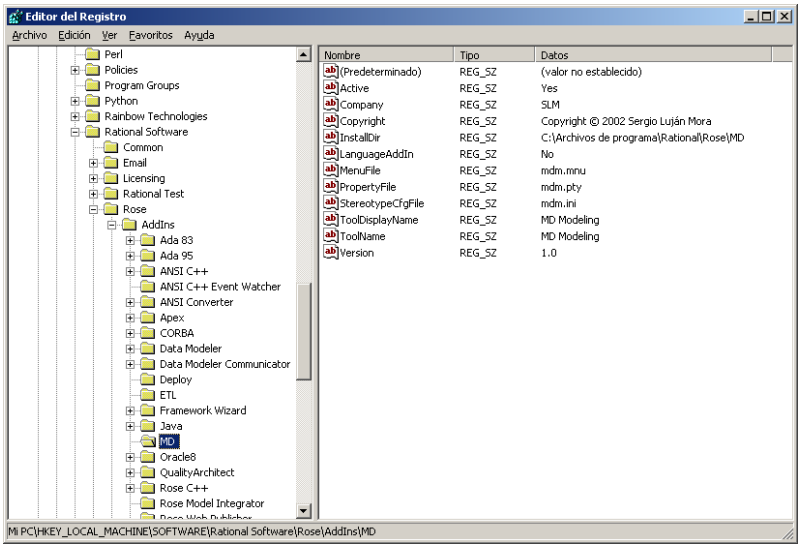


Figure E.6: Subkeys of the add-in created in Microsoft Windows registry

```
[Stereotyped Items]
Logical Package:Star
Logical Package:Fact
Logical Package:Dimension
Class:Fact
Class:DegenerateFact
Class:Dimension
Class:Base
Attribute:DegenerateDimension
Attribute:FactAttribute
Attribute:DimensionAttribute
Attribute:OID
Attribute:Descriptor
Association:Completeness
Association:Rolls-upTo

[Logical Package:Star]
Item=Logical Package
Stereotype=StarPackage
Metafile=&\stereotypes\slm\pk_star.wmf
SmallPaletteImages=&\stereotypes\slm\pk_star_s.bmp
SmallPaletteIndex=1
MediumPaletteImages=&\stereotypes\slm\pk_star_m.bmp
MediumPaletteIndex=1
ListImages=&\stereotypes\slm\pk_star_l.bmp
ListIndex=1
ToolTip=Creates a star package\nStar package
```

```
[Logical Package:Fact]
Item=Logical Package
Stereotype=FactPackage
Metafile=&\stereotypes\slm\pk_fact.wmf
SmallPaletteImages=&\stereotypes\slm\pk_fact_s.bmp
SmallPaletteIndex=1
MediumPaletteImages=&\stereotypes\slm\pk_fact_m.bmp
MediumPaletteIndex=1
ListImages=&\stereotypes\slm\pk_fact_l.bmp
ListIndex=1
ToolTip=Creates a fact package\nFact package

[Logical Package:Dimension]
Item=Logical Package
Stereotype=DimensionPackage
Metafile=&\stereotypes\slm\pk_dimension.wmf
SmallPaletteImages=&\stereotypes\slm\pk_dimension_s.bmp
SmallPaletteIndex=1
MediumPaletteImages=&\stereotypes\slm\pk_dimension_m.bmp
MediumPaletteIndex=1
ListImages=&\stereotypes\slm\pk_dimension_l.bmp
ListIndex=1
ToolTip=Creates a dimension package\nDimension package

[Class:Fact]
Item=Class
Stereotype=Fact
Metafile=&\stereotypes\slm\fact.wmf
SmallPaletteImages=&\stereotypes\slm\fact_s.bmp
SmallPaletteIndex=1
MediumPaletteImages=&\stereotypes\slm\fact_m.bmp
MediumPaletteIndex=1
ListImages=&\stereotypes\slm\fact_l.bmp
ListIndex=1
ToolTip=Creates a fact class\nFact class

[Class:DegenerateFact]
Item=Class
Stereotype=DegenerateFact
Metafile=&\stereotypes\slm\dfact.wmf
SmallPaletteImages=&\stereotypes\slm\dfact_s.bmp
SmallPaletteIndex=1
MediumPaletteImages=&\stereotypes\slm\dfact_m.bmp

MediumPaletteIndex=1
ListImages=&\stereotypes\slm\dfact_l.bmp
ListIndex=1
ToolTip=Creates a degenerate fact class\nDegenerate fact class

[Class:Dimension]
Item=Class
Stereotype=Dimension
Metafile=&\stereotypes\slm\dimension.wmf
SmallPaletteImages=&\stereotypes\slm\dimension_s.bmp
SmallPaletteIndex=1
```

```
MediumPaletteImages=&\stereotypes\slm\dimension_m.bmp
MediumPaletteIndex=1
ListImages=&\stereotypes\slm\dimension_l.bmp
ListIndex=1
ToolTip=Creates a dimension class\nDimension class

[Class:Base]
Item=Class
Stereotype=Base
Metafile=&\stereotypes\slm\base.wmf
SmallPaletteImages=&\stereotypes\slm\base_s.bmp
SmallPaletteIndex=1
MediumPaletteImages=&\stereotypes\slm\base_m.bmp
MediumPaletteIndex=1
ListImages=&\stereotypes\slm\base_l.bmp
ListIndex=1
ToolTip=Creates a base class\nBase class

[Attribute:DegenerateDimension]
Item=Attribute
Stereotype=DegenerateDimension
ListImages=&\stereotypes\slm\dd_l.bmp
ListIndex=1
ToolTip=Creates a degenerate dimension\nDegenerate dimension

[Attribute:FactAttribute]
Item=Attribute
Stereotype=FactAttribute
ListImages=&\stereotypes\slm\fa_l.bmp
ListIndex=1
ToolTip=Creates a fact attribute\nFact attribute

[Attribute:DimensionAttribute]
Item=Attribute
Stereotype=DimensionAttribute
ListImages=&\stereotypes\slm\da_l.bmp
ListIndex=1
ToolTip=Creates a dimension attribute\nDimension attribute

[Attribute:OID]
Item=Attribute
Stereotype=OID
ListImages=&\stereotypes\slm\oid_l.bmp
ListIndex=1
ToolTip=Creates an OID attribute\nOID attribute

[Attribute:Descriptor]
Item=Attribute
Stereotype=Descriptor
ListImages=&\stereotypes\slm\des_l.bmp
ListIndex=1
ToolTip=Creates a Descriptor attribute\nDescriptor attribute

[Association:Completeness]
Item=Association
Stereotype=Completeness
```

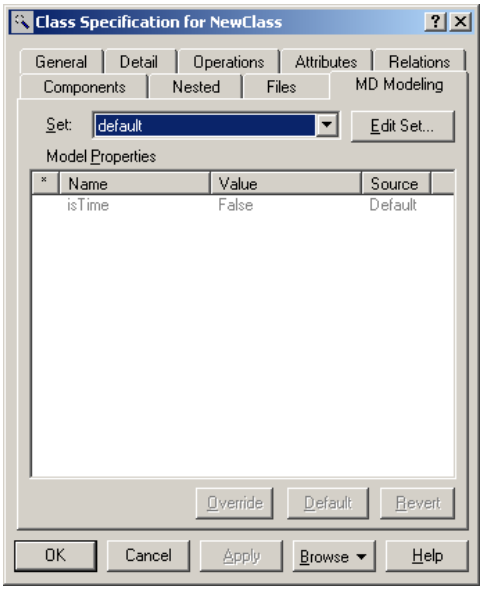


Figure E.7: New properties for a class element

```
[Association:Rolls-upTo]
Item=Association
Stereotype=Rolls-upTo
```

E.4.3 Tag Definitions

`mdm.pty` is the property file that defines a name space for its properties and a tab in the specification editor to hold the custom tool, sets, and properties.

Our add-in defines two sets of new properties: for the class element and for the attribute element. In Figure E.7, we show the new "tab" that is added to the specification windows of a class element, whereas in Figure E.8, we show the same "tab" for the attribute element. In both cases, we can notice the new properties.

```
(object Petal
    version     43
    _written    "SLM")

(list Attribute_Set
    (object Attribute
        tool    "MD Modeling"
        name    "default__Class"
        value   (list Attribute_Set
```

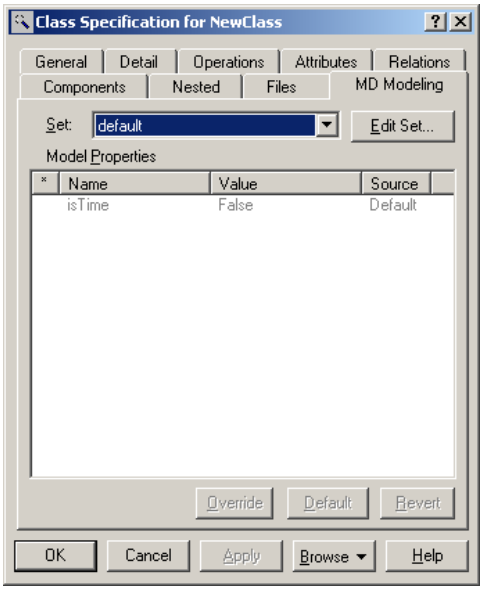


Figure E.8: New properties for an attribute element

```
                (object Attribute
                tool    "MD Modeling"
                name    "isTime"
                value   FALSE)))

(object Attribute
    tool    "MD Modeling"
    name    "default__Attribute"
    value   (list Attribute_Set
                (object Attribute
                tool    "MD Modeling"
                name    "isAtomic"
                value   TRUE)
                (object Attribute
                tool    "MD Modeling"
                name    "derivationRule"
                value   "")
                (object Attribute
                tool    "MD Modeling"
                name    "additivity"
                value   "")
                (object Attribute
                tool    "MD Modeling"
                name    "isOID"
                value   TRUE)
                (object Attribute
                tool    "MD Modeling"
                name    "isDescriptor"
```

```
                value   TRUE)
                (object Attribute
                tool    "MD Modeling"
                name    "description"
                value   ""))))
```

E.4.4 Menu Items

`mdm.mnu` defines the new menu items that are added to the Rational Rose's Tools menu (see Figure E.2).

```
Menu Tools {
  Separator
  option "MD Validate"
   {
    RoseScript $SCRIPT_PATH\MD\mdvalidate.ebs
   }


  option "MD to XML"
   {
    RoseScript $SCRIPT_PATH\MD\xmlgenerate.ebs
   }

  option "MD to DDL"
   {
    RoseScript $SCRIPT_PATH\MD\ddlgenerate.ebs
   }
 }
```

E.4.5 Rose Script

`mdvalidate.ebs` contains the Rose Script code of our add-in. This file contains more than 1100 lines of code.

Below, we only show the definition of the main procedure that is executed when MD Validate is selected in the Tools menu (see Figure E.2).

```
Sub Main
    Dim i As Integer
    Dim vc As Boolean
    Dim myModel As Model
    Dim theClasses As ClassCollection


    Set myModel = RoseApp.CurrentModel
    Set theClasses = myModel.GetAllClasses()

    numErrores=0

    FillingErrors 'Fill array with error messages (constraints)
```

```
    'Validation of Classes
    vc = True
    For i = 1 To theClasses.Count
        Select Case VClass(theClasses.GetAt(i))
            Case 1
                vc = False
            End Select
    Next i

    'Validation of packages and cicles
    If myModel.IsRootPackage() Then
        'Can't exist cicles in package collection
        'Apply vCicles to all categories in the model
        Select Case VCicles(myModel.GetAllCategories)
            Case 1
                vc = False
        End Select

        'This condition always is true
        Select Case VPackages(myModel.Categories)
            Case 1
                vc = False
        End Select
    End If

    RoseApp.WriteErrorLog numErrores & " errores encontrados"

    If vc Then
        MsgBox "The model has been succesfully validated",_
               ebInformation, "Validation Result"
    Else
        MsgBox "The model has not been succesfully validated." &_
               "See Log for details", ebCritical, "Validation Result"
    End If
End Sub
```

Bibliography

[1] A. Abelló, J. Samos, and F. Saltor. Benefits of an Object-Oriented Multidimensional Data Model. In *Proceedings of the Symposium on Objects and Databases in 14th European Conference on Object-Oriented Programming (ECOOP'00)*, volume 1944 of *Lecture Notes in Computer Science*, pages 141–152, Sophia Antipolis and Cannes, France, June 13 2000. Springer-Verlag.

[2] A. Abelló, J. Samos, and F. Saltor. A Framework for the Classification and Description of Multidimensional Data Models. In *Proceedings of the 12th Internatinal Conference on Database and Expert Systems Applications (DEXA'01)*, volume 2113 of *Lecture Notes in Computer Science*, pages 668–677, Munich, Germany, September 3 - 7 2001. Springer-Verlag.

[3] A. Abelló, J. Samos, and F. Saltor. YAM2 (Yet Another Multidimensional Model): An Extension of UML. In *International Database Engineering & Applications Symposium (IDEAS'02)*, pages 172–181, Edmonton, Canada, July 17 - 19 2002. IEEE Computer Society.

[4] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML.* Morgan Kaufmann, 1999.

[5] L. Agosta. Market Overview Update: ETL. Technical Report RPA-032002-00021, Giga Information Group, March 2002.

[6] J. Akoka, I. Comyn-Wattiau, and N. Prat. Dimension Hierarchies Design from UML Generalizations and Aggregations. In *Proceedings of the 20th Internatinal Conference on Conceptual Modeling (ER'01)*, volume 2224 of *Lecture Notes in Computer Science*, pages 442–455, Yokohama, Japan, November 27 - 30 2001. Springer-Verlag.

[7] S. Allen. *Data Modeling for Everyone.* Curlingstone Publishing, 2002.

[8] S.W. Ambler. Persistence Modeling in the UML. Software Development Online. Internet: http://www.sdmagazine.com/-documents/s=755/sdm9908q/, August 1999.

[9] S.W. Ambler. A UML Profile for Data Modeling. Internet: http://www.agiledata.org/essays/umlDataModelingProfile.html, 2002.

[10] S.W. Ambler. *The Elements of UML Style*. Cambridge University Press, 2002.

[11] ANSI/ISO/IEC. Database Language SQL. International Standard (IS) 9075:1999, ANSI/ISO/IEC, September 1999.

[12] J. Arlow and I. Neustadt. *UML and the Unified Process. Practical Object-Oriented Analysis & Design*. Object Technology Series. Addison-Wesley, 2002.

[13] D.E. Avison and G. Fitzgerald. *Information Systems Development: Methodologies, Techniques and Tools*. Blackwell Scientific Publications, 1988. (Last edition: 2nd edition, McGraw-Hill/Irwin, 1998).

[14] P.A. Bernstein, A.Y. Levy, and R.A. Pottinger. A Vision for Management of Complex Models. Technical Report MSR-TR-2000-53, Microsoft Research, June 2000.

[15] P.A. Bernstein and E. Rahm. Data Warehouse Scenarios for Model Management. In *Proceedings of the 19th International Conference on Conceptual Modeling (ER'00)*, volume 1920 of *Lecture Notes in Computer Science*, pages 1–15, Salt Lake City, USA, October 9 - 12 2000. Springer-Verlag.

[16] M. Blaschka, C. Sapia, G. Höfling, and B. Dinter. Finding your way through multidimensional data models. In *Proceedings of the 9th International Conference on Database and Expert Systems Applications (DEXA'98)*, volume 1460 of *Lecture Notes in Computer Science*, pages 198–203, Vienna, Austria, August 24 - 28 1998. Springer-Verlag.

[17] G. Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, 2 edition, 1994.

[18] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language: User Guide*. Object Technology Series. Addison-Wesley, 1999.

[19] R.M. Bruckner, B. List, and J. Schiefer. Developing Requirements for Data Warehouse Systems with Use Cases. In *Proceedings of the 7th Americas Conference on Information Systems (AMCIS'01)*, pages 329–335, Boston, USA, August 3 - 5 2001.

[20] L. Cabibbo and R. Torlone. A Logical Approach to Multidimensional Databases. In *Proceedings of the 6th International Conference on Extending Database Technology (EDBT'98)*, volume 1377 of *Lecture Notes in Computer Science*, pages 183–197, Valencia, Spain, March 23 - 27 1998. Springer-Verlag.

[21] L. Carneiro and A. Brayner. X-META: A Methodology for Data Warehouse Design with Metadata Management. In *Proceedings of 4th International Workshop on the Design and Management of Data Warehouses (DMDW'02)*, pages 13–22, Toronto, Canada, May 27 2002.

[22] R.G.G. Cattell. *The Object Database Standard: ODMG 3.0*. Morgan Kaufmann, 2000.

[23] J.M. Cavero, M. Piattini, and E. Marcos. MIDEA: A Multidimensional Data Warehouse Methodology. In *Proceedings of the 3rd International Conference on Enterprise Information Systems (ICEIS'01)*, pages 138–144, Setubal, Portugal, July 7 - 10 2001. ICEIS Press.

[24] S. Chaudhuri and U. Dayal. An Overview of Data Warehousing and OLAP Technology. *ACM Sigmod Record*, 26(1):65–74, March 1997.

[25] P. Chen. The Entity-Relationship Model – toward a Unified View of Data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36, March 1976.

[26] E.F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[27] J. Conallen. *Building Web Applications with UML*. Object Technology Series. Addison-Wesley, 2000. (Last edition: 2nd edition, Addison-Wesley, 2003).

[28] M. Corey, M. Abbey, I. Abramson, and B. Taub. *Oracle8i Data Warehousing*. Oracle Press. Osborne/McGraw-Hill, 2001.

[29] S. Cronholm and P. Agerfalk. On the Concept of Method in Information Systems Development. In *Proceedings of the 22nd Information Systems Research In Scandinavia (IRIS 22)*, Keuruu, Finland, August 7 - 10 1999.

[30] Y. Cui and J. Widom. Lineage Tracing for General Data Warehouse Transformations. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB'01)*, pages 471–480, Rome, Italy, September 11 - 14 2001.

[31] C. Cunningham, C.A. Galindo-Legaria, and G. Graefe. PIVOT and UNPIVOT: Optimization and Execution Strategies in an RDBMS. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB 2004)*, pages 998–1009, Toronto, Canada, August 31 - September 3 2004. Morgan Kaufmann.

[32] Cutter Consortium. 41% Have Experienced Data Warehouse Project Failures. The Cutter Edge. Internet: http://www.cutter.com/research/2003/edge030218.html, February 2003.

[33] K. Czarnecki and S. Helsen. Classification of Model Transformation Approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of Model-Driven Architecture*, Anaheim, USA, October 27 2003.

[34] N.T. Debevoise. *The Data Warehouse Method.* Prentice-Hall, New Jersey, USA, 1999.

[35] A. Dobre, F. Hakimpour, and K. R. Dittrich. Operators and Classification for Data Mapping in Semantic Integration. In *Proceedings of the 22nd International Conference on Conceptual Modeling (ER'03)*, volume 2813 of *Lecture Notes in Computer Science*, pages 534–547, Chicago, USA, October 13 - 16 2003. Springer-Verlag.

[36] W. Eckerson. Data Quality and the Bottom Line. *Application Development Trends*, May, 2002.

[37] W. Eckerson. Four ways to build a data warehouse. *Application Development Trends*, May, 2002.

[38] D.W. Embley, B.D. Kurtz, and S.N. Woodfield. *Object-oriented Systems Analysis: A Model-Driven Approach.* Prentice-Hall, 1992.

[39] H. Eriksson and M. Penker. *UML Toolkit.* John Wiley & Sons, 1998.

[40] E.D. Falkenberg. Concepts for modelling information. In *Proceedings of the IFIP Conference on Modelling in Data Base Management Systems*, pages 95–109, Amsterdam, Holland, September 1976.

[41] P. Feldman and D. Miller. Entity Model Clustering: Structuring a Data Model by Abstraction. *The Computer Journal*, 29(4):348–360, 1986.

[42] E. Fernández-Medina, J. Trujillo, R. Villarroel, and M. Piattini. Extending UML for Designing Secure Data Warehouses. In *Proceedings of the 23rd International Conference on Conceptual Modeling (ER'04)*, volume 3288 of *Lecture Notes in Computer Science*, pages 217–230, Shanghai, China, November 8 - 12 2004. Springer-Verlag.

[43] M. Fowler. *UML Distilled. Applying the Standard Object Modeling Language*. Object Technology Series. Addison-Wesley, 1998.

[44] R. France and J. Bieman. Multi-View Software Evolution: A UML-based Framework for Evolving Object-Oriented Software. In *Proceedings of the 17th International Conference on Software Maintenance (ICSM 2001)*, pages 386–397, Florence, Italy, November 6 - 10 2001. IEEE Computer Society.

[45] T. Friedman. ETL Magic Quadrant Update: Market Pressure Increases. Technical Report M-19-1108, Gartner, January 2003.

[46] M. Gandhi, E.L. Robertson, and D. Van Gucht. Leveled Entity Relationship Model. In *Proceedings of the 13th International Conference on Entity-Relationship Approach (ER'94)*, volume 881 of *Lecture Notes in Computer Science*, pages 420–436, Manchester, United Kingdom, December 13 - 16 1994. Springer-Verlag.

[47] S.R. Gardner. Building the Data Warehouse. *Communications of the ACM*, 41(9):52–60, September 1998.

[48] W. Giovinazzo. *Object-Oriented Data Warehouse Design. Building a star schema*. Prentice-Hall, New Jersey, USA, 2000.

[49] M. Golfarelli, D. Maio, and S. Rizzi. The Dimensional Fact Model: A Conceptual Model for Data Warehouses. *International Journal of Cooperative Information Systems (IJCIS)*, 7(2-3):215–247, June & September 1998.

[50] M. Golfarelli and S. Rizzi. A Methodological Framework for Data Warehouse Design. In *Proceedings of the ACM 1st International Workshop on Data Warehousing and OLAP (DOLAP'98)*, pages 3–9, Bethesda, USA, November 7 1998. ACM.

[51] D. Gornik. Data Modeling for Data Warehouses. Rational Software Corporation. Internet: http://www.rational.com/media/-whitepapers/tp161.pdf, 2002.

[52] D. Hackney. Data Warehouse Delivery: Who Are You? Part I. *DM Review Magazine*, February, 1998.

[53] T. Halpin. UML Data Models from an ORM Perspective: Part One. *Journal of Conceptual Modeling*, April(1), April 1998.

[54] T. Halpin and A. Bloesch. Data modeling in UML and ORM: a comparison. *Journal of Database Management*, 10(4):4–13, 1999.

[55] B. Hüsemann, J. Lechtenbörger, and G. Vossen. Conceptual Data Warehouse Modeling. In *Proceedings of the 2nd International Workshop on Design and Management of Data Warehouses (DMDW'00)*, pages 6.1–6.11, Stockholm, Sweden, June 5 - 6 2000.

[56] IBM. IBM Rational Unified Process (RUP). Internet: http://www.rational.com/products/rup/index.jsp, 2003.

[57] W.H. Inmon. *Building the Data Warehouse*. QED Press/John Wiley, 1992. (Last edition: 3rd edition, John Wiley & Sons, 2002).

[58] Institut National de Recherche en Informatique et en Automatique (INRIA). Model transformation at Inria. Internet: http://modelware.inria.fr/, 2004.

[59] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Object Technology Series. Addison-Wesley, 1999.

[60] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

[61] P. Jaeschke, A. Oberweis, and W. Stucky. Extending ER Model Clustering by Relationship Clustering. In *Proceedings of the 12th International Conference on Entity-Relationship Approach (ER'93)*, volume 823 of *Lecture Notes in Computer Science*, pages 451–462, Arlington, USA, December 15 - 17 1993. Springer-Verlag.

[62] M. Jarke, M. Lenzerini, Y. Vassiliou, and P. Vassiliadis. *Fundamentals of Data Warehouses*. Springer-Verlag, 2 edition, 2003.

[63] R. Kimball. *The Data Warehouse Toolkit*. John Wiley & Sons, 1996. (Last edition: 2nd edition, John Wiley & Sons, 2002).

[64] R. Kimball. A Dimensional Modeling Manifesto. *DBMS*, 10(9), August 1997.

[65] R. Kimball, L. Reeves, M. Ross, and W. Thornthwaite. *The Data Warehouse Lifecycle Toolkit*. John Wiley & Sons, 1998.

[66] C. Kobryn. UML 2001: A Standardization Odyssey. *Communications of the ACM*, 42(10):29–37, October 1999.

[67] L. Greenfield. Data Extraction, Transforming, Loading (ETL) Tools. The Data Warehousing Information Center. Internet: http://www.dwinfocenter.org/clean.html, 2003.

[68] W. Lehner. Modelling Large Scale OLAP Scenarios. In *Proceedings of the 6th International Conference on Extending Database Technology (EDBT'98)*, volume 1377 of *Lecture Notes in Computer Science*, pages 153–167, Valencia, Spain, March 23 - 27 1998. Springer-Verlag.

[69] M. Lenzerini. Data Integration: A Theoretical Perspective. In *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 233–246, Madison, USA, June 3 - 6 2002. ACM.

[70] S. Luján-Mora. Multidimensional Modeling using UML and XML. In *Proceedings of the 12th Workshop for PhD Students in Object-Oriented Systems (PhDOOS 2002)*, volume 2548 of *Lecture Notes in Computer Science*, pages 48–49, Málaga, Spain, June 10 - 14 2002. Springer-Verlag.

[71] S. Luján-Mora and E. Medina. Reducing Inconsistency in Data Warehouses. In *Proceedings of the 3rd International Conference on Enterprise Information Systems (ICEIS'01)*, pages 199–206, Setúbal, Portugal, July 7 - 10 2001. ICEIS Press.

[72] S. Luján-Mora, E. Medina, and J. Trujillo. A Web-Oriented Approach to Manage Multidimensional Models through XML Schemas and XSLT. In *Proceedings of the XML-Based Data Management and Multimedia Engineering (EDBT 2002 Workshops)*, volume 2490 of *Lecture Notes in Computer Science*, pages 29–44, Prague, Czech Republic, March 24 2002. Springer-Verlag.

[73] S. Luján-Mora and J. Trujillo. A Comprehensive Method for Data Warehouse Design. In *Proceedings of the 5th International Workshop on Design and Management of Data Warehouses (DMDW'03)*, pages 1.1–1.14, Berlin, Germany, September 8 2003.

[74] S. Luján-Mora and J. Trujillo. A Data Warehouse Engineering Process. In *Proceedings of the 3rd Biennial International Conference on Advances in Information Systems (ADVIS'04)*, volume 3261 of *Lecture Notes in Computer Science*, pages 14–23, Izmir, Turkey, October 20 - 22 2004. Springer-Verlag.

[75] S. Luján-Mora and J. Trujillo. Modeling the Physical Design of Data Warehouses from a UML Specification. In *Proceedings of the 8th IASTED International Conference on Software Engineering and Applications (SEA 2004)*, pages 7–12, Cambridge, USA, November 9 - 11 2004.

[76] S. Luján-Mora and J. Trujillo. Modeling the Physical Design of Data Warehouses from a UML Specification. In *Proceedings of the ACM Seventh International Workshop on Data Warehousing and OLAP (DOLAP 2004)*, pages 48–57, Washington D.C., USA, November 12 - 13 2004. ACM.

[77] S. Luján-Mora and J. Trujillo. Physical Modeling of Data Warehouses by using UML Component and Deployment Diagrams: design and implementation issues. *Journal of Database Management*, 17(1), January-March 2006. Accepted to be published.

[78] S. Luján-Mora, J. Trujillo, and I. Song. Extending UML for Multidimensional Modeling. In *Proceedings of the 5th International Conference on the Unified Modeling Language (UML'02)*, volume 2460 of *Lecture Notes in Computer Science*, pages 290–304, Dresden, Germany, September 30 - October 4 2002. Springer-Verlag.

[79] S. Luján-Mora, J. Trujillo, and I. Song. Multidimensional Modeling with UML Package Diagrams. In *Proceedings of the 21st International Conference on Conceptual Modeling (ER'02)*, volume 2503 of *Lecture Notes in Computer Science*, pages 199–213, Tampere, Finland, October 7 - 11 2002. Springer-Verlag.

[80] S. Luján-Mora, J. Trujillo, and P. Vassiliadis. Advantages of UML for Multidimensional Modeling. In *Proceedings of the 6th International Conference on Enterprise Information Systems (ICEIS 2004)*, pages 298–305, Porto, Portugal, April 14 - 17 2004. ICEIS Press.

[81] S. Luján-Mora, P. Vassiliadis, and J. Trujillo. Data Mapping Diagrams for Data Warehouse Design with UML. In *Proceedings of the 23rd International Conference on Conceptual Modeling (ER'04)*, volume 3288 of *Lecture Notes in Computer Science*, pages 191–204, Shanghai, China, November 8 - 12 2004. Springer-Verlag.

[82] R.A. Maksimchuk and E.J. Naiburg. Entity Relationship Modeling with UML. *DM Direct Newsletter*, January, 2003.

[83] E. Marcos, B. Vela, and J.M. Cavero. Extending UML for Object-Relational Database Design. In *Proceedings of the 4th International Conference on the Unified Modeling Language (UML'01)*, volume 2185 of *Lecture Notes in Computer Science*, pages 225–239, Toronto, Canada, October 1 - 5 2001. Springer-Verlag.

[84] E. Medina, S. Luján-Mora, and J. Trujillo. Handling Conceptual Multidimensional Models using XML through DTDs. In *Proceedings of 19th British National Conference on Databases (BNCOD 2002)*, volume 2405 of *Lecture Notes in Computer Science*, pages 66–69, Sheffield, UK, July 17 - 19 2002. Springer-Verlag.

[85] D.L. Moody. A Multi-Level Architecture for Representing Enterprise Data Models. In *Proceedings of the 16th International Conference on Conceptual Modeling (ER'97)*, volume 1331 of *Lecture Notes in Computer Science*, pages 184–197, Los Angeles, USA, November 3 - 5 1997. Springer-Verlag.

[86] D.L. Moody. A Methodology for Clustering Entity Relationship Models - A Human Information Processing Approach. In *Proceedings of the 18th International Conference on Conceptual Modeling (ER'98)*, volume 1728 of *Lecture Notes in Computer Science*, pages 114–130, Paris, France, November 15 - 18 1999. Springer-Verlag.

[87] D.L. Moody and M.A.R. Kortink. From Enterprise Models to Dimensional Models: A Methodology for Data Warehouse and Data Mart Design. In *Proceedings of the 2nd International Workshop on Design and Management of Data Warehouses (DMDW'01)*, pages 5.1–5.12, Stockholm, Sweden, June 5 - 6 2000.

[88] R.J. Muller. *Database Design for Smarties: Using UML for Data Modeling.* Morgan Kaufmann, 1999.

[89] N. Pendse. The 2004 OLAP market shares. The OLAP Report. Internet: http://www.olapreport.com/market.htm, 2005.

[90] E.J. Naiburg and R.A. Maksimchuk. *UML for Database Design*. Object Technology Series. Addison-Wesley, 2001.

[91] S. Naqvi and S. Tsur. *A Logical Language for Data and Knowledge Bases*. Computer Science Press, 1989.

[92] National Technical University of Athens (Greece). Knowledge and Database Systems Laboratory. Internet: http://www.dblab.ntua.gr/, 2003.

[93] M. Nicola and H. Rizvi. Storage Layout and I/O Performance in Data Warehouses. In *Proceedings of the 5th International Workshop on Design and Management of Data Warehouses (DMDW'03)*, pages 7.1–7.9, Berlin, Germany, September 8 2003.

[94] Object Management Group (OMG). Common Warehouse Metamodel (CWM) Specification 1.0. Internet: http://www.omg.org/cgi-bin/doc?ad/2001-02-01, February 2001.

[95] Object Management Group (OMG). Meta Object Facility (MOF) Specification 1.4. Internet: http://www.omg.org/cgi-bin/doc?formal/2002-04-03, April 2002.

[96] Object Management Group (OMG). UML Profile for CORBA 1.0. Internet: http://www.omg.org/cgi-bin/doc?formal/02-04-01, April 2002.

[97] Object Management Group (OMG). Unified Modeling Language (UML) Specification 1.5. Internet: http://www.omg.org/cgi-bin/doc?formal/03-03-01, March 2003.

[98] Object Management Group (OMG). Model Driven Architecture (MDA). Internet: http://www.omg.org/mda/, 2004.

[99] T.W. Olle, M. Daya, E.D. Falkenberg, B. Yormark, and R.W. Taylor. Panel: The conceptual schema controversy. In *Proceedings of the 1978 ACM SIGMOD International Conference on Management of Data*, pages 88–88, Austin, USA, May 31 - June 2 1978. ACM.

[100] V. Poe, P. Klauer, and S. Brobst. *Building a Data Warehouse for Decision Support*. Prentice-Hall, 2 edition, 1998.

[101] G.V. Post. *Database Management Systems*. Mcgraw-Hill, 2001.

[102] T. Quatrani. *Visual Modeling with Rational Rose and UML*. Object Technology Series. Addison-Wesley, 1998.

[103] QVT-Partners. Revised submission for MOF 2.0 Query / Views / Transformations RFP. Internet: http://qvtp.org/-downloads/1.1/qvtpartners1.1.pdf, 2003.

[104] E. Rahm and H. Do. Data Cleaning: Problems and Current Approaches. *IEEE Bulletin of the Technical Committee on Data Engineering*, 23(4):3–13, December 2000.

[105] Rational Software Corporation. Migrating from XML DTD to XML-Schema using UML. Internet: http://www.rational.com/media/whitepapers/-TP189draft.pdf, 2000.

[106] Rational Software Corporation. The UML and Data Modeling. Internet: http://www.rational.com/media/whitepapers/-Tp180.PDF, 2000.

[107] Rational Software Corporation. *Using the Rose Extensibility Interface*. Rational Software Corporation, 2001.

[108] S. Rizzi. Open problems in data warehousing: eight years later. In *Proceedings of the 5th International Workshop on Design and Management of Data Warehouses (DMDW'03)*, Berlin, Germany, September 8 2003.

[109] Ronin International. Enterprise Unified Process (EUP). Internet: http://www.enterpriseunifiedprocess.info/, 2003.

[110] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1992.

[111] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison-Wesley, 1999.

[112] C. Sapia. On Modeling and Predicting Query Behavior in OLAP Systems. In *Proceedings of the 1st International Workshop on Design and Management of Data Warehouses (DMDW'99)*, pages 1–10, Heidelberg, Germany, June 14 - 15 1999.

[113] C. Sapia, M. Blaschka, G. Höfling, and B. Dinter. Extending the E/R Model for the Multidimensional Paradigm. In *Proceedings of the 1st International Workshop on Data Warehouse and Data Mining (DWDM'98)*, volume 1552 of *Lecture Notes in Computer Science*, pages 105–116, Singapore, November 19 - 20 1998. Springer-Verlag.

[114] K.D. Schewe. *Information Modelling and Knowledge Bases XII*, volume 67 of *Frontiers in Artificial Intelligence and Applications*, chapter UML – A Modern Dinosaur?: A Critical Analysis of the Unified Modelling Language, pages 185–202. IOS Press, 2001.

[115] A. Schleicher and B. Westfechtel. Beyond Stereotyping: Metamodeling Approaches for the UML. In *Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)*, pages 1–10, Maui, USA, January 03 - 06 2001. IEEE Computer Society.

[116] A. Schürr and A.J. Winter. Formal Definition and Refinement of UML's Module/Package Concept. In *ECOOP'97 Workshop Reader*, volume 1357 of *Lecture Notes in Computer Science*, pages 211–215, Jyväskylä, Finland, June 9 - 13 1997. Springer-Verlag.

[117] A. Schürr and A.J. Winter. Formal Definition of UML's Package Concept. In *UML Workshop 1997*, pages 144–159, Mannheim, Germany, November 10 - 11 1997.

[118] A. Sen and A.P. Sinha. A Comparison of Data Warehousing Methodologies. *Communications of the ACM*, 48(3):79 – 84, March 2005.

[119] M. Serrano, C. Calero, J. Trujillo, S. Luján-Mora, and M. Piattini. Empirical Validation of Metrics for Conceptual Models of Data Warehouses. In *Proceedings of the 16th International Conference on Advanced Information Systems Engineering (CAiSE 2004)*, volume 3084 of *Lecture Notes in Computer Science*, pages 506–520, Riga, Latvia, June 7 - 11 2004. Springer-Verlag.

[120] M. Serrano, C. Calero, J. Trujillo, S. Luján-Mora, and M. Piattini. Towards a Metric Suite for Conceptual Models of Datawarehouse. In *Proceedings of the 1st International Workshop on Software Audit and Metrics (SAM'04)*, pages 105–117, Porto, Portugal, April 14 - 17 2004.

[121] D. Shah and S. Slaughter. *UML and the Unified Process*, chapter Transforming UML class diagrams into relational data models, pages 217–236. Idea Group Publishing, 2003.

[122] A. Snell. The Need for an Implementation Methodology. Internet: http://www.ilogos.com/en/expertviews/-articles/technology/20010703_AS.html, 2003.

[123] SQL Power Group. How do I ensure the success of my DW? Internet: http://www.sqlpower.ca/page/dw_best_practices, 2002.

[124] K. Strange. ETL Was the Key to this Data Warehouse's Success. Technical Report CS-15-3143, Gartner, March 2002.

[125] T.J. Teorey, G. Wei, D.L. Bolton, and J.A. Koenig. ER Model Clustering as an Aid for User Communication and Documentation in Database Design. *Communications of ACM*, 32(8):975–987, August 1989.

[126] B. Thalheim. *Entity-Relationship Modeling. Foundations of Database Technology.* Springer-Verlag, 2000.

[127] J. Trujillo and S. Luján-Mora. Automatically Generating Structural and Dynamic Information of OLAP Applications from Object-Oriented Conceptual Models. *International Journal of Computer & Information Science*, 3(4):227–236, December 2002.

[128] J. Trujillo and S. Luján-Mora. A UML Based Approach for Modeling ETL Processes in Data Warehouses. In *Proceedings of the 22nd International Conference on Conceptual Modeling (ER'03)*, volume 2813 of *Lecture Notes in Computer Science*, pages 307–320, Chicago, USA, October 13 - 16 2003. Springer-Verlag.

[129] J. Trujillo, S. Luján-Mora, and E. Medina. Utilización de UML para el modelado multidimensional. In *I Taller de Almacenes de Datos y Tecnología OLAP (ADTO 2001), VI Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2001)*, pages 12–17, Almagro, Spain, November 22 2001.

[130] J. Trujillo, S. Luján-Mora, and I. Song. *Advanced Topics in Database Research*, volume 2, chapter Applying UML for designing multidimensional databases and OLAP applications, pages 13–36. Idea Group Publishing, 2003.

[131] J. Trujillo, S. Luján-Mora, and I. Song. Applying UML and XML for designing and interchanging information for data warehouses and OLAP applications. *Journal of Database Management*, 15(1):41–72, January-March 2004.

[132] J. Trujillo, M. Palomar, J. Gómez, and I. Song. Designing Data Warehouses with OO Conceptual Models. *IEEE Computer, special issue on Data Warehouses*, 34(12):66–75, December 2001.

[133] N. Tryfona, F. Busborg, and J.G. Christiansen. starER: A Conceptual Model for Data Warehouse Design. In *Proceedings of the ACM 2nd International Workshop on Data Warehousing and OLAP (DOLAP'99)*, pages 3–8, Kansas City, USA, November 6 1999. ACM.

[134] P. Vassiliadis, A. Simitsis, and S. Skiadopoulos. Conceptual Modeling for ETL Processes. In *Proceedings of the ACM 5th International Workshop on Data Warehousing and OLAP (DOLAP 2002)*, pages 14–21, McLean, USA, November 8 2002. ACM.

[135] P. Vassiliadis, A. Simitsis, and S. Skiadopoulos. Modeling ETL Activities as Graphs. In *Proceedings of 4th International Workshop on the Design and Management of Data Warehouses (DMDW'02)*, pages 52–61, Toronto, Canada, May 27 2002.

[136] P. Vassiliadis, Z. Vagena, S. Skiadopoulos, N. Karayannidis, and T. Sellis. ARKTOS: towards the modeling, design, control and execution of ETL processes. *Information Systems*, 26(8):537–561, December 2001.

[137] D. Vesset. Worldwide Data Warehousing Tools 2004-2008 Forecast and 2003 Vendor Shares. Technical Report Doc #31616, IDC, July 2004.

[138] J. Vowler. Data warehouse design from top to bottom. ComputerWeekly.com. Internet: http://www.computerweekly.com/-Article110431.htm, March 2002.

[139] J. Warmer and A. Kleppe. *The Object Constraint Language. Precise Modeling with UML*. Object Technology Series. Addison-Wesley, 1998.

[140] B. Wixom and H.J. Watson. An Empirical Investigation of the Factors Affecting Data Warehousing Success. *MIS Quarterly*, 25(1):17–41, March 2001.

[141] World Wide Web Consortium (W3C). XSL Transformations (XSLT) Version 1.0. Internet: http://www.w3.org/TR/xslt, November 1999.

[142] World Wide Web Consortium (W3C). Extensible Stylesheet Language (XSL) 1.0. Internet: http://www.w3.org/TR/xsl/, October 2001.

[143] World Wide Web Consortium (W3C). Extensible Markup Language (XML) 1.0 (Third Edition). Internet: http://www.w3.org/TR/REC-xml/, February 2004.

[144] World Wide Web Consortium (W3C). XML Schema Part 0: Primer Second Edition. Internet: http://www.w3.org/TR/xmlschema-0/, October 2004.

Index

Authors Index