PM

SDK Development for Bridging Heterogeneous Data Sources Through Connect Bridge Platform

MASTER PROJECT

Filipa José Faria Nóbrega
MASTER IN COMPUTER ENGINEERING

UNIVERSIDADE da MADEIRA
A Nossa Universidade
www.uma.pt

September | 2018

SDK Development for Bridging Heterogenous Data Sources Through Connect Bridge Platform

MASTER PROJECT

Filipa José Faria Nóbrega
MASTER IN COMPUTER ENGINEERING

SUPERVISOR
Leonel Domingos Telo Nóbrega

CO-SUPERVISOR
Stanislav Pernecký

This page intentionally left blank

Dedication and Acknowledgements

First of all, my deep sense of gratitude goes to all the extraordinary people who have been generous enough to dedicate some of their time and knowledge during the course of my dissertation work. The attitude of the people made this dissertation possible.

I am extremely thankful to my supervisor Prof. Dr. Leonel Nóbrega for his guidance, constant encouragement, valuable suggestions and critical discussions throughout my dissertation work. During my dissertation-writing period, he provided encouragement, good teaching, sound advice, and many good ideas. I also thank him for his patience and kindness. His dedication to work and perfectionism will have a far-reaching impact on my life, academic and beyond.

I would also like to thank my supervisor Eng. Stanislav Pernecký for his guidance, inspiring suggestions and critical discussions throughout my dissertation work.

I am especially grateful to Connecting Software for creating excellent working conditions and providing financial support.

My heartfelt thanks to my friends for providing me with unfailing support and continuous encouragement.

Special thanks to all participants who spend their time to contribute to the outcome of this dissertation.

I would also like to thank Max for his love, support, humour, and energy that has given me strength in life.

Finally, thanks to my family for their unconditional love and support.

Thank you all!

RESUMO

Nesta dissertação apresentou-se um SDK para a criação de conectores a integrar com o *CB Server*, que pretende: acelerar o desenvolvimento, garantir melhores práticas e simplificar as diversas atividades e tarefas no processo de desenvolvimento. O SDK fornece uma API pública e simples, suportada por um conjunto de ferramentas, que facilitam o processo de desenvolvimento, explorando as facilidades disponibilizadas através da API. Para analisar a exatidão, viabilidade, integridade e acessibilidade da solução apresentam-se dois exemplos e casos de estudo. Através dos casos de estudo foi possível identificar uma lista de problemas, de pontos sensíveis e melhorias na solução proposta. Para avaliar a usabilidade da API, uma metodologia baseada em vários métodos de avaliação de usabilidade foi estabelecida. O múltiplo caso de estudo funciona como o principal método de avaliação, combinando vários métodos de pesquisa. O caso de estudo consiste em três fases de avaliação: um workshop, uma avaliação heurística e uma análise subjetiva. O caso de estudo envolveu três engenheiros de software (incluindo programadores e avaliadores). A metodologia aplicada gerou resultados com base num método de inspeção, testes de utilizador e entrevistas. Identificou-se não só pontos sensíveis e falhas no código-fonte, mas também problemas estruturais, de documentação e em tempo de execução, bem como problemas relacionados com a experiência do utilizador. O contexto do estudo é apresentado de modo a tirar conclusões acerca dos resultados obtidos. O trabalho futuro incluirá o desenvolvimento de novas funcionalidades. Adicionalmente, pretende-se resolver problemas encontrados na metodologia aplicada para avaliar a usabilidade da API, nomeadamente problemas e falhas no código fonte (por exemplo, validações) e problemas estruturais.

Palavras-chave: SDK, API, Metamodelo, Modelo de dados relacional, SQL, Usabilidade

Abstract

In this dissertation, we present an SDK for the creation of connectors to integrate with CB Server which accelerates deployment, ensures best practices and simplifies the various activities and tasks in the development process. The SDK provides a public and simple API leveraged by a set of tools around the API developed which facilitate the development process by exploiting the API facilities. To analyse the correctness, feasibility, completeness, and accessibility of our solution, we presented two examples and case studies. From the case studies, we derived a list of issues found in our solution and a set of proposals for improvement. To evaluate the usability of the API, a methodology based on several usability evaluation methods has been established. Multiple case study works as the main evaluation method, combining several research methods. The case study consists of three evaluation phases – a hands-on workshop, a heuristic evaluation and subjective analysis. The case study involved three computer science engineers (including novice and expert developers and evaluators). The applied methodology generated insights based on an inspection method, a user test, and interviews. We identify not only problems and flaws in the source code, but also runtime, structural and documentation problems, as well as problems related to user experience. To help us draw conclusion from the results, we point out the context of the study. Future work will include the development of new functionalities. Additionally, we aim to solve problems found in the applied methodology to evaluate the usability of the API, namely problems and flaws in the source code (e.g. validations) and structural problems.

Keywords: SDK, API, Metamodel, Relational data model, SQL, Usability

CONTENTS

LIST OF TABLES

LIST OF FIGURES

List of Abbreviations and Acronyms

API	Application Programming Interface
C#	C Sharp
CB	Connect Bridge
CBDK	Connect Bridge Development Kit
CLR	Common Language Runtime
CM	Connector Metamodel
CRC	Class-Responsibility-Collaborator
CRUD	Create, read, update and delete
CWM	Common Warehouse Metamodel
DBMS	Database Management System
EA	Enterprise Architect
EDM	Entity Data Model
FIFO	First-In-First-Out
FR	Functional Requirement
HCI	Human-Computer Interaction
IT	Information Technology
JDBC	Java Database Connectivity
JSON	JavaScript Object Notation
MDA	Model-driven Architecture
MDD	Model-driven Development
MOF	Meta Object Facility
MS	Microsoft
MSSQL	Microsoft SQL Server
NFR	Non-functional Requirement
O/R	Object/Relational
OData	Open Data Protocol
ODBC	Open Database Connectivity
OMG	Object Management Group
OO	Object-Oriented
RDBMS	Relational Database Management System
RE	Requirement Engineering

RSA	Rivest–Shamir–Adleman
SDK	Software Development Kit
SQL	Structured Query Language
UML	Unified Modelling Language
UX	User Experience
VS	Visual Studio

List of Appendices

Chapter 1

Introduction

Data integration is one of the oldest research fields in the database area and has emerged proximately after database systems were first made known to the business world [1]. Nowadays, numerous organizations are increasingly recognizing the need to utilize data integration tools to meet business and customer needs. In this setting, data integration is becoming an important, yet very difficult challenge once data is stored and represented in different data models [1], [2].

The integration of multiple information systems purposes to combine a set of independent and heterogeneous information systems into a homogeneous logical view of data [1]. To accomplish this, all data has to be represented using the same abstraction principles (a unified global data model and unified semantics).

This chapter briefly introduces the background of the project, explicitly, the software development company for which this project was developed and the platform for software integration challenges. Moreover, the motivation, problem, objectives, methodology, and contributions of the dissertation are outlined. Finally, the structure of the dissertation is presented.

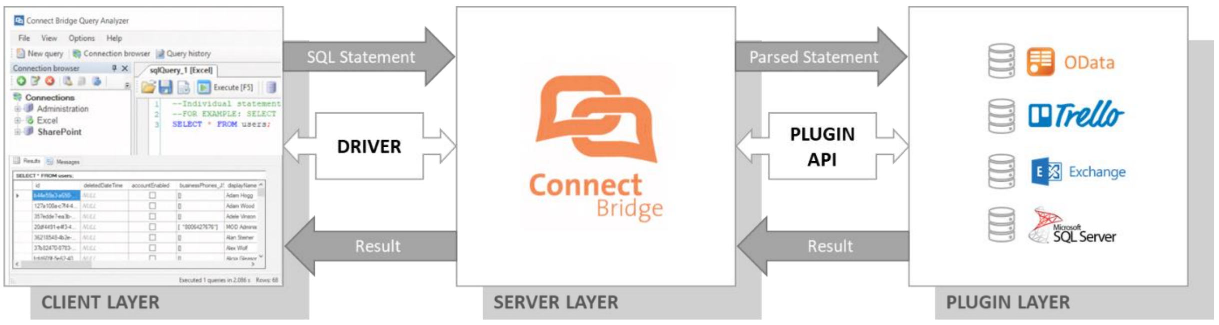
1.1 Background

For years, the key to success for any business solution has been data. Selecting the right solution to incorporate all types of organizational data from multiple autonomous and heterogeneous sources into a unified and consolidated source is a pervasive challenge. This has led us to the emergence of data integration, which should be a strategic topic in all organizations because it affects everything the organization does. Thereby, many organizations are increasingly recognizing the need to use complete and comprehensive data integration tools that are capable of integrating datasets, warehouses, enterprise applications, and systems.

Connecting Software is a software development company started in 2004 that provides an easily extensible and versatile integration platform for software integration challenges termed Connect Bridge and, pre-assembled products using its capabilities. By enabling developers to integrate data virtually from any source (e.g. Microsoft OneDrive, Microsoft SharePoint, Microsoft Exchange, Google Drive, etc.), this company dismantles the boundaries of inter-software communication which allows systems and applications to communicate in one universal language for managing data, so to speak: Structured Query Language (SQL).

Connect Bridge (CB) is an independent solution based on a plugin architecture whose main goal is to provide a uniform query interface to a multitude of data sources, thereby freeing the casual user from having to locate data sources, interact with each one in isolation and manually combine results.

Connect Bridge is divided into **three major layers** (see Figure 1.1):



```
--Individual statement
--FOR EXAMPLE: SELECT
SELECT * FROM users;
```

Figure 1.1 Basic Workflow in Connect Bridge Platform

1. **Client Layer.** This layer provides connectivity between client applications and the Server Layer. A client application is an application that utilizes Open Database Connectivity (ODBC), Java Database Connectivity (JDBC) drivers or Web Services to connect with the Server Layer, which can be written in any programming language.

2. **Server Layer.** This layer provides connectivity between Client and Plugin layers. When a connection is requested by the client application, Server Layer performs authentication and authorization based on the configuration of the system enabling the client application to manipulate data through SQL statements. As soon as the client application is authenticated and authorized, the SQL statement can be sent to the Server Layer to be processed. Server Layer receives the SQL

statement for processing in a parse tree once the SQL statement has been evaluated according to the rules of a formal grammar – lexical, syntactic and semantic analysis. A parse tree – a data structure that represents a parsed statement – is formerly available to the plugins through a set of common interfaces. Server Layer receives all data from Plugin Layer to be sent back to one of the drivers and, subsequently, to the client application.

3. **Plugin Layer.** This layer provides connectivity between the data provider and the Server Layer. A plugin (local or remote) is an external software component that adds features to a host application (i.e. CB Server). A plugin has to implement a set of common interfaces exposed by the Connector Application Programming Interface (API)[1] to intermediate the communication between the data provider and CB Server.

1.2 MOTIVATION

This section discusses the three major reasons that motivate the work undertaken at Connecting Software. First, there is a need to develop a Software Development Kit (SDK), which has a set of software development tools, libraries, documentation, code samples, and guides for the creation of connectors in order to help developers. Second, there is the confidence that the applicability of CB may well increase, and the number of connectors may well growth. Third, the taste for innovation and technological development coupled with a personal wish to create new and practical solutions beyond knowledge and experience.

1.3 PROBLEM

At the moment, Connecting Software provides an SDK for the creation of connectors which is complex, intricate and hides functionality. Furthermore, the existing SDK does not have documentation. For these reasons, connectors are developed internally by the Connecting Software developers which set a limitation on the number of available connectors, since the resources for their development are limited.

[1] MG Framework

1.4 OBJECTIVES

The main objective of this dissertation is to develop an SDK containing a set of subroutine definitions, interfaces, common structures, utilities, algorithms and tools that describes and prescribes the expected behaviour for the creation of connectors to integrate with Connect Bridge Server, which can be used with minimal technical support by the developers.

The specific objectives are:

1. To make available a complete and clear SDK documentation.
2. To make available a Step-by-Step guide for creating a connector from scratch.
3. To create sample connector projects to illustrate the usage of the SDK.
4. To create a Visual Studio project template holding a connector skeleton.
5. To make available useful Visual Studio code snippets for typical code blocks.

1.5 METHODOLOGY

The research methodology that was followed for the purposes of this dissertation involves two different types of research: primary and secondary research. For the primary research, various methodologies namely case study, heuristic evaluation, and interview were chosen to assess the feasibility and the usability of the SDK. For the secondary research, a literature review was held on previous academic research regarding database systems, data models, SQL, and so forth. Furthermore, Object Management Group (OMG) specifications were gathered to serve as a formal reference for modelling the SDK.

The software development methodology that was chosen for the purposes of this dissertation was agile software development under which a system is developed through repeated cycles, termed iteration. Each *three-week* iteration includes planning (man-day estimation), requirements gathering and analysis, design, implementation, testing (unit testing), and documentation.

1.6 CONTRIBUTIONS

One of the main contributions of this dissertation is to provide an SDK which accelerates deployment, ensures best practices and simplifies the creation of connectors. As well, this dissertation contributes to the application area of computer science, namely in the fields of relational databases, data integration, and data modelling. Finally, another contribution is to provide an open-source product which has a financial value.

1.7 DISSERTATION STRUCTURE

The dissertation is structured as follows:

- **Chapter 1 Introduction** contains the context of this work, the description of the problems to tackle and the process followed to solve them. Additionally, the motivating reasons and objectives of this dissertation are presented.
- **Chapter 2 Literature Review** introduces database and database systems. It also contains the definition of data model, a brief history of data model development and an overview of the relational data model. As well, the ANSI/ISO SQL standard is briefly described. Furthermore, models, metamodels and Model-Driven Development (MDD) concepts are presented along with the OMG's metamodeling framework.
- **Chapter 3 Development** contains the system requirements, architectural design and implementation details.
- **Chapter 4 Case Studies** addresses two examples and case studies conducted to assess the feasibility of the SDK.
- **Chapter 4 Evaluation** addresses a multiple case study conducted to evaluate the usability of the SDK and subsequent analysis.
- **Chapter 5 Conclusions and Future Work** contains the closure of this work by presenting a reflection about the development and evaluation chapters. Also, the future work is provided.

Chapter 2

Literature Review

Over the past few years, database research has contributed a great deal to the database system becoming the most significant development in the field of software engineering. The database is now the basic framework of the information system, changing the way many companies and individuals work [3]. The importance of the database system has increased in the last few years with significant developments in this technology resulting in increased availability of database systems that are more powerful, intuitive to use. Nowadays, by far the most popular database system available is the relational database management system (RDBMS). As a result of the development of the relational databases, SQL has emerged and become the standard relational database language most widely used [4], [5].

This chapter was structured using a *constructivist* approach – a recognised educational approach by which "new" knowledge is built on prior knowledge and reflective thinking. In the *constructivist* approach, the big concepts are first emphasized, beginning with the whole (more concrete) and expanding to include the parts (more abstract). As a result, we start to define the database and database management system (DBMS). We then introduce the definition of data model – a type of abstraction that serves as an architectural focus for the design of databases and DBMS. Additionally, we provide a brief history of data model development along with an overview of the relational data model. We then introduce briefly the ANSI/ISO SQL standard – a query language for defining and manipulating data stored in a database. In addition, we review some of the commands and data types supported by the standard. We then define even more abstract concepts such as models, metamodels and Model-Driven Development (MDD) – note that a data model is an abstract language which can be described by creating a metamodel of a language [6]. Finally, we introduce the OMG's metamodeling framework for specifying, managing, interchanging, constructing, and integrating models in software systems.

2.1 DATABASE SYSTEMS

A **database** is defined as an organized "*collection of self-describing data*" that stores a collection of both data and metadata [3]. Data is raw facts of interest to the user [4]. Metadata describes the data's structure within a database (i.e. data characteristics and the set of relationships that links the data), through which the data is integrated and managed [4], [5].

A **database management system** (DBMS) is defined as a software that manages the database structure and controls access to the data stored in the database [4]. The *major components* of DBMS include a data dictionary containing descriptive information about the data within a database, a data definition language for further describing the conceptual organization of the entire database (*schema*), a query language for simple access to the database, and features to support data security, integrity, backup, and recovery [4], [7], [8]. Each of these components is explained below.

- **Data dictionary**. The data dictionary (also named *catalog* or *system catalog*) is used to maintain definitions of the data elements and their relationships (*metadata*) within the scope of the database system.
- **Data Definition Language (DLL)**. The DLL is used to define the database schema components.
- **Query Language**. The query language is a specialized language that enables data access through *queries*.
- **Security, Integrity, Backup and Recovery Functions**. DBMS performs several functions to guarantee the integrity and security of the data in the database. To guarantee integrity, DBMS enforces integrity rules to minimize data redundancy and maximize data consistency. To guarantee security, DBMS enforces security rules that control user's access to the database. To ensure backup and recovery, DBMS provides special backup and recovery of the database after a failure.

2.2 DATA MODELS

A **data model** is a type of data abstraction that serves as an architectural focus for the design of databases and DBMS [7], [9], [10]. A data model is defined as "*an integrated* *collection of concepts for describing and manipulating data, relationships between data, and constraints on the data*" [3], [9]. The most important components of a data model include [9]:

(1) A collection of data types and data structure types (the database building blocks);

(2) A collection of operators or rules of inference;

(3) A collection of general integrity rules.

2.2.1 History of Data Models Development

As of 1960, several data models have been proposed for the management of structured data [3], [4], [9]. The first such data model to be proposed was the **file system**. As a result of the file system's critical shortcomings, the **hierarchical model** arose to handle and manage vast amounts of information for complex projects such as the Apollo rocket [3], [4]. The hierarchical model is based on a hierarchical structure, which conforms to an upside-down tree, composed of levels, or segments – a *segment* is the equivalent of a file system's record type. This model was defined by the process of abstraction from IBM's IMS [3], [9].

In the 1970s, another significant data model has emerged: **network model**. The network model was created to represent complex relationships, to improve database performance, and to impose a database standard. This model was defined by the process of abstraction from CODASYL's DBTG [3], [9]. The network model describes a set of 1:M relationships between a parent and its segments, in which a segment can have more than one parent, unlike the hierarchical model. The network model proposal defined some important concepts:

- The **schema**, which is the logical organization of the entire database.
- The **subschema**, which defines the part of the database seen by application programs.
- A **data management language**, which defines the language to manage and manipulate the data in the database.

– A schema **DDL**, which enables the database system to define the database schema components.

In the mid–1970s, Edgar F. Codd (IBM Researcher) published a theoretical paper "*A relational model of data for large shared data banks*" on **relational model** as a mathematical basis for the analysis and modelling of data, providing data independence and addressing several issues related to database [11]; these features made it the preferred data model for business applications. The relational model also has a downside such as limited modelling capabilities. The relational model is further discussed in Section 2.2.2.

In response to the increasing complexity of database applications, two "new" data models have emerged: the **object-oriented (OO)** and the **object/relational (O/R)** models [4]. The OO data model was proposed as an alternative to the relational data model and is based on a single structure known as *object* composed of data and their relationships [4], [7]. The O/R data model can be defined as an attempt to extend relational data model with the functionality necessary to support many of OO model's features, providing a bridge between relational and object-oriented paradigms [7].

Table 2.1 provides a summary of the historical development of major data models [4].

Table 2.1 Historical Development of Major Data Models

TIMEFRAME	DATA MODEL	EXAMPLES	OBSERVATIONS

1960s–1970s	File System	VMS/VSAM	The database system's pioneer;
			IBM mainframe systems;
			Manage records, not relationships;
			Each department stored and controlled its own data.

1970s	Hierarchical and Network	IMS IDS	Difficult to represent relationships; Lack of data independence (structural level dependence); No ad-hoc queries; Navigational access.
Mid–1970s to present	Relational	Oracle	Conceptual simplicity (structural independence);

		MS SQL-Server MySQL	Ad hoc queries; Set-oriented access.

Mid–1970s to present	Object-oriented
	Object/Relational
		PostgreSQL	More semantics;
			Support for complex objects;
			Inheritance;
			Behaviour;
			Unstructured data (XML);
			XML data exchanges.

2.2.2 Relational Model

In 1969, Edgar F. Codd (IBM Researcher) proposed the **relational model** based on mathematical set theory [12]. Its *structural part* consists of domains, relations (with *tables* as their principal theoretical representation), attributes, tuples, primary keys, and candidate keys. The principal theoretical representation consists of a set of tuples (rows), each tuple having the same set of attributes (columns) with the following properties [13] (see Figure 2.1):

1. There is no duplication of rows (i.e. all rows are distinct from one another in content);
2. Row and column order are immaterial;
3. All its entries are atomic values.

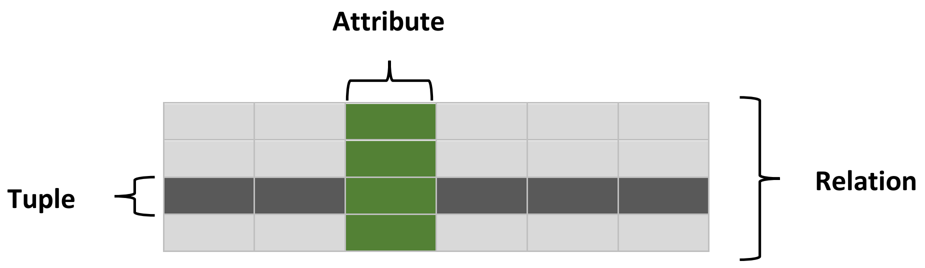


Figure 2.1 Structural part of the Relational Model

The *manipulative part* consists of the algebraic operators (select, join, project, etc.) which transform relations into relations. The *integrity part* consists of two integrity rules: *entity integrity* – the primary key of a table must contain a unique, non-null value for each row

– and *referential integrity* – if the foreign key contains a value, that value must make reference to an existing row in the parent table. In any application of a data model, it may be necessary to impose further integrity constraints, and thereby define a smaller set of consistent database states or changes of state.

The relational model is implemented through a very sophisticated **relational database management system** (RDBMS) – the predominant database system for business application at present [3], [4]. An RDBMS supports structural aspects of the relational model, the insert-update-delete rules, and a clearly defined language as powerful as the relational algebra [13].

2.3 Structured Query Language

The Structured Query Language (SQL) was created by IBM researchers as an industry–standardized language for defining and manipulating data stored in a database [5], [14]. An official standard for SQL was originally published by the American National Standards Institute (ANSI) and the International Standards Organization (ISO) in 1986 and was expanded in the following years [14]. SQL was originally developed to work on data in databases that follow the *relational model* [5]. In fact, SQL is the *de facto* query language and data access standard supported by most DBMSs [4].

2.3.1 Commands

The ANSI/ISO SQL standard is a combination of at least two major components [3]–[5]:

1. The **Data Definition Language** (DDL): SQL includes commands for defining the database structure and access rights to the data. Table 2.2 provides a summary of the data definition commands.

Table 2.2 SQL Data Definition Commands

Command or Option	Description
CREATE SCHEMA	Creates a database schema in the database
CREATE TABLE	Creates a new table in the database
NOT NULL	Ensures that a column will not have *null* values
UNIQUE	Ensures that a column will not have *duplicate* values
PRIMARY KEY	Defines a primary key for a table

FOREIGN KEY	Defines a foreign key for a table
DEFAULT	Defines a default value for a column
CHECK	Validates data in a column
CREATE INDEX	Creates an index
CREATE VIEW	Creates a virtual table from one or more tables in the database
ALTER TABLE	Modifies the structure of an existing table
CREATE TABLE AS	Creates a new table based on a query
DROP TABLE	Removes an unneeded table (and its data) from the database
DROP INDEX	Removes an unneeded index from the database
DROP VIEW	Removes a view from the database

2. The **Data Manipulation Language** (DML): SQL includes commands to insert, update, delete, and retrieve data within the database. Table 2.3 provides a summary of the data manipulation commands [4], [5], [14].

Table 2.3 SQL Data Manipulation Commands

Command or Option	Description
INSERT	Adds one or more rows of data to the database
SELECT	Retrieves data from the database
FROM	Specifies from which tables to retrieve data
WHERE	Restricts the selection of rows based on a search criterion
GROUP BY	Groups the selected rows based on one or more attributes
HAVING	Restricts the selection of grouped rows based on a search criterion
ORDER BY	Orders the selected rows based on one or more attributes
UPDATE	Modifies existing database data
DELETE	Removes one or more rows of data from the database
Operators	**Description**
COMPARISON	Used in search criteria
=	Equal to
<	Less than
>	Greater than
<=	Less than or equal to

>=	Greater than or equal to
<> or !=	Not equal to
LOGICAL	Used in search criteria
AND	Combines multiple conditions
OR	Combines multiple conditions
NOT	Negates a condition
SPECIAL	Used in search criteria
BETWEEN	Checks whether an attribute value is within a range
IS NULL	Checks whether an attribute value is null
LIKE	Checks whether an attribute value matches a given string pattern
IN	Checks whether an attribute value matches any value within a list
EXISTS	Checks whether a subquery returns any rows
DISTINCT	Confines values to unique values
ARITHMETIC	Used in search criteria
+	Add
-	Subtract
*****	Multiply
/	Divide
^	Raise to the power of
Aggregate Functions	**Description**
COUNT	The number of rows with non-null values for a given attribute
MIN	The minimum attribute value found in an attribute
MAX	The maximum attribute value found in an attribute
SUM	The sum of all values for a given attribute
AVG	The average of all values for a given attribute

2.3.2 Data Types

The ANSI/ISO SQL standard specifies several data types, which describes the type of data that can be stored in a SQL-based database system and manipulated by the SQL language.

Table 2.4 provides a summary of the SQL data types specified in the ANSI/ISO SQL standard [3], [5], [14].

Table 2.4 SQL Data Types

Data Type	Description	Declarations
Boolean	Consists of the distinct truth values TRUE and FALSE, as well as UNKNOWN.	BOOLEAN
Character	Consists of a sequence of characters. It may be defined as having a fixed or varying length.	CHAR [length] VARCHAR [length]
Bit	Consists of a sequence of binary digits.	BIT [length]
Exact numeric	Express the value of a number exactly. The number consists of digits, an optional decimal point, and an optional sign.	NUMERIC [precision [, scale]] DECIMAL [precision [, scale]] INTEGER SMALLINT BIGINT
Approximate numeric	Express the value of a number that does not have an exact representation.	FLOAT [precision] REAL DOUBLE PRECISION
Datetime	Defines points in time to a certain degree of accuracy.	DATE TIME [precision] [TIME ZONE] TIMESTAMP [precision] [TIME ZONE]
Interval	Represents periods of time.	INTERVAL
Large objects (LOBs)	Holds a large amount of data, such as a long text file or a graphics file.	CHARACTER LARGE OBJECT (CLOB) BINARY LARGE OBJECT (BLOB)

2.4 Models, Metamodels and Model-Driven Development

Modelling and models are the key tools in software engineering [15], [16]. Software designers and developers often create models when designing and analysing large or complex systems. A model is an abstraction of a system and its environment that has proven to be highly successful throughout the software development process. Still, there is a great deal of disbelief regarding the practical value of modelling software [15], [17]. There is a reason for this. Because models are translated into code manually, they are not kept up-to-date as implementation progresses, and hence they become inaccurate, out-dated and useless with time. Additionally, many modelling languages had very weak semantic specifications. Consequently, the software engineering community's view is that models have only value in the early phases of development, playing a secondary, mostly specification and documentation, role [17]–[19].

Model-driven development (MDD) is an approach to software development in which models do not constitute documentation, but rather enjoy the same status as code, as their implementation is automated – that is, models play a central and active role [15]. Models, modelling, and Model-Driven Architecture (MDA) are the foundation of MDD [20].

The MDA initiative by the Object Management Group (OMG) has contributed significantly to the interest in software modelling and model-driven techniques [15], [21]. As a result, models received anew attention from the research community and IT industry [22]. This led to the today's central view wherein models become essential artefacts of the development process, rather than simply serving a dispensable supporting purpose.

2.4.1 Model

The ability to find and use models is the basis of our abstract reasoning [23]. Thus, models are very important. Yet, the term "model" is hard to define like other central notions.

In software engineering, models are the first artefacts of the development process to systematically describe a software architecture. Their application can be found in various areas and applications of software engineering and has become more popular with the advent of the Unified Modelling Language (UML) in 1996. Yet, their application in software engineering is at an early stage.

In this dissertation, a model is defined as an abstraction or a coherent set of formal elements describing a system for some purpose of analysis or synthesis. The ability to use models in replacement of systems under study is the broad sense we adopt for this notion.

2.4.1.1 What is a model?

The term “model” is derived from the Latin *modulus*, which means pattern, measure, example or rule to be followed [23]. As stated by Jean-Marie Favre in [24], the several meanings for this term can be grouped into four groups:

1. **Model as Representation**: the notion of model is used as a simplified description of a complex entity or process. The typical example of such model is a map.
2. **Model as Example**: the notion of model is used to describe a representative of a category, which is a synonym of exemplar and instance. The typical examples of such model are visual arts.
3. **Model as Type**: the notion of model is used to designate a set of elements distinguished by some common characteristic or quality. A typical example of such model is automobile industry where each new type of car is designated by a model.
4. **Model as Mould**: the notion of model is used as a representative form or pattern worthy of imitation.

In software engineering, even though the term “model” is frequently recognized **as representation** of a new or existing system, it can also be recognized **as mould**, as is the case with design patterns.

2.4.1.2 The Model Criteria

To distinguish a model from other artefacts, we need to define *criteria*. As pointed out by Ludewig in [23], in 1973 Stachowiak proposed three criteria that any model candidate must satisfy:

1. **Mapping criterion**: there is an original object or phenomenon that is mapped to the model.

2. **Reduction criterion**: not all the properties of "the original" are mapped on the model, nonetheless, the model is somehow a simplification. Yet, the model must reflect at least some properties of the original.
3. **Pragmatic criterion**: the model can substitute the original for some purpose – that is, the model is useful.

The **mapping criterion** does not infer the real existence of the original; note that the model may be planned, suspected, or even fictitious. Frequently, in software engineering, the models act as the original of another model – the aim is to use the models in the production process leading to the original.

At first sight, since not all the properties of the original are in the model, the **reduction criterion** may imply a weakness of the model. However, that loss is the real strong point of models; frequently, while the original cannot be handled the model can. Figure 2.2 illustrates the mapping between original and model according to [23]. Note that the model is not necessarily like the original and the attributes may be mapped in various ways.

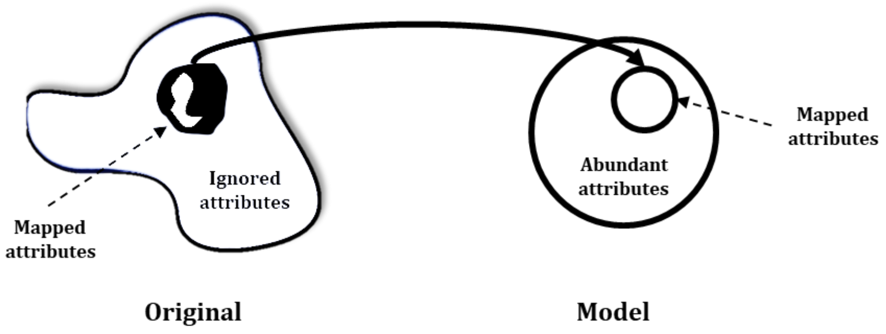


Figure 2.2 Original and model

The **pragmatic criterion** is the explanation why we use models. The truth is that often we are not able or not willing to use the original, so we may use the model as an alternative.

So, the question that arises is, "what models can do for software development?" A model of some system is a simplification of that system, where the properties of interest are highlighted for a given viewpoint [17]. According to Bran Selic [17], a valuable model should provide at least the following:

1. **Abstraction**: the model should remove or hide all irrelevant detail.
2. **Understandable**: the model should be expressed in a way that takes only the essential information directly and precisely.
3. **Accurate**: the model must correctly mirror the properties of interest of the system being modelled to within the acceptable tolerances required.
4. **Predictive**: the model must be able to precisely predict the behaviour and other properties of the modelled system (which depends on the model's accuracy).

Abstraction is deeply related to the reduction criterion introduced by Stachowiak – the model is an abstraction of "the original" – but exposes the process by which the models are obtained. The use of abstraction is commonly observed in the design of computer software, in which a model/abstraction describes only the essential details of the system to be modelled, ignoring the irrelevant details. Thus, there is always a simplification of the modelled system which satisfies the reduction criterion.

For Bran Selic, a valuable model in software development (more precisely, in the context of the MDA initiative) must be "sufficiently" understandable, accurate and predictable; which is related to the mapping criterion where there is always an original system. Thus, models are able to be developed, constructed, manipulated, and viewed as established in the OMG's proposal.

2.4.1.3 Definitions

While significant efforts have been made to define the term model, there is no universally accepted definition. Yet, the definitions attempt to include several concepts previously mentioned. Let's cite some existing definitions. For Bran Selic, the term model is defined as follows [17]:

> *"… model of some **system** is a reduced representation of that **system** that highlights the properties of interest for a given viewpoint."*

Bézivin and Gerbé provide another definition [25]:

> *"A model is a simplification of a **system** built with an intended goal in mind. The model should be able to answer questions in place of the actual system."*

Seidewitz defines the term model as [26]:

> *"... a set of statements about some* ***system*** *under study (SUS)."*

In the context of OMG's MDA Guide [27], the term model is defined as follows:

> *"A model of a* ***system*** *is a description or specification of that system and its environment for some certain purpose"*

From these definitions, we are able to identify three central notions: the notion of model, the notion of system and the relationship between these notions. The notion of system is the key element of discourse when talking about software modelling and model-driven techniques in software engineering [28]. The relationship between model and system is called *RepresentationOf*, which is a synonym of *InstanceOf*. Figure 2.3 illustrates the three notions before mentioned.

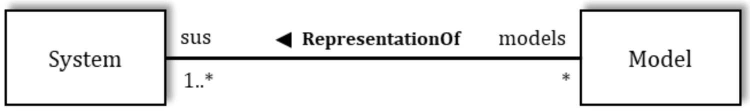


Figure 2.3 Relationship between model and system

It is important to understand that this relation is well-defined between systems: being a model or system under study (SUS) is a relative notion, not an inherent property of an artefact. In fact, a model of a model implies that a model is a system under study and, consequently, this model is a system, emphasizing the idea that a model is also a system as shown in Figure 2.4.

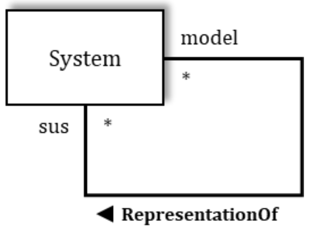


Figure 2.4 Relationship between systems

There is clearly a compromise between simplicity and expressiveness – Figure 2.4 illustrates a more reduced but less explicit diagram when compared to Figure 2.3. Yet, Figure 2.4 fails to present the notion of model as a central notion and specifies that any system can be a model, by simply identifying which system is being modelled. Therefore, Figure 2.5 illustrates the model as a specialization of the system, in which the notion of model appears as a central notion, preserving the relationships and specifying the particularity that a model is also a system.

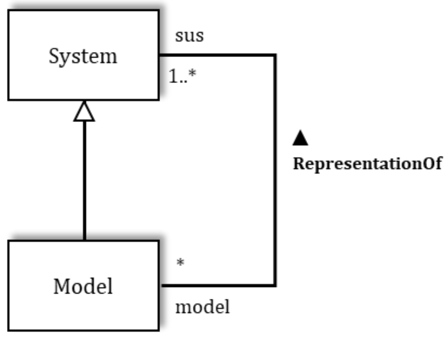


Figure 2.5 Model as a specialization of the system

In summary, a model is a system that should be able to answer questions in place of the system under study.

2.4.2 Metamodels

In the previous section, various definitions of the term model were cited. None of these definitions refers to the notion of **language** or of **metamodel**. Kleppe et al. give in [29] a more restrictive definition of what is a model: "*A model is a description of (part of) a system written in a* ***well-defined language***". In this definition, the concept of model is strongly connected to the notion of language and therefore to the notion of metamodel. Since these two concepts are often confused, this section clarifies the relationships between these concepts.

2.4.2.1 What is a metamodel?

For Jean-Marie Favre in [30], "*a language is a set of systems*", more precisely, "*a language is formalized as a set of sentences*". In fact, the notion of language is an abstract notion itself since it does not have any materialization. For instance, we know that the Portuguese language exists since, given a sentence, we can say if it is Portuguese or not; because we use some *models of the language* – a grammar or a dictionary –, which provide practical means to get answers.

So, the question that arises is, "what a model of a language is?". In accordance with the definition of model given previously, a *model of a language* is a system that could provide answers about the language under study. For natural languages, grammars are models of a language (note that a grammar is typically described using the language being modelled). For programming languages, Backus-Naur Form (BNF) language is typically used to define the grammar – a programming language is a set of words over an alphabet which is defined by the conforming grammar. For the UML modelling language, the modelling language's definition is accomplished by building a model of the modelling language – termed metamodel (The prefix meta is derived from the Greek *metá*, which means "after" or "beyond"). In other words, the definition of UML modelling language is defined in terms of itself: UML modelling language is written in UML.

But what about the other way around – that is languages of models? For Jean-Marie Favre in [30], a modelling language is a set whose elements are models which can be expressed through the language itself. In short,

(A) *"A **modelling language** is a set of models."*

By introducing the concept of modelling language, it is possible to give an accurate definition of what is a metamodel. For Jean-Marie Favre in [30], metamodels make modelling languages explicit.

(B) *"A **metamodel** is a model of a modelling language."*

In fact, replacing **(A)** into **(B)** leads logically to:

(C) *"A **metamodel** is a model of **a set of models**."*

By reason of uniformity, the relation between the language and model will be called *ElementOf* according to [30]. Note that nothing prevents a model of a set being itself an element of that set. In fact, it is necessary to know the Portuguese language to understand what is written in the Portuguese grammar. Yet, not all metamodels are part of the language they model. Figure 2.6 illustrates the relationship between the concepts of metamodel, language and model.

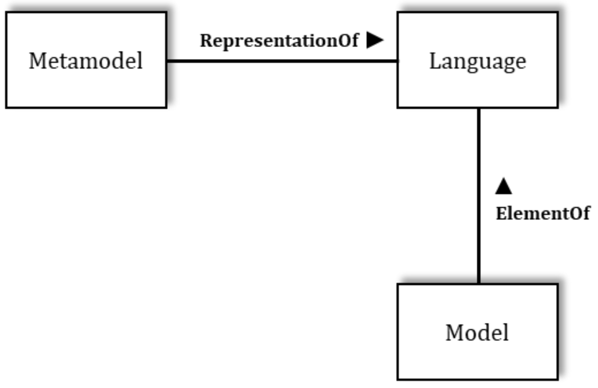


Figure 2.6 Relationships between the concepts of metamodel, language and model

In Figure 2.6 there is no direct relation between the concepts of metamodel and model, but rather between the language and model. However, from the practical point of view, the *ElementOf* is not operational since languages are usually infinite and abstract systems. To decide whether a model is an element of the language or not, it is necessary to confront the model with the metamodel chosen for the language. This use of metamodel requires a derived relation from *RepresentationOf* which is called (*Linguistic*) *InstanceOf* in Atkinson and Kühne [31] and *ConformantTo* in Bézivin [32]. In fact, checking the conformance between a model and a metamodel makes it possible to decide whether a model is an element of the language or not. For instance, the spelling checker is a metamodel that checks if a given text is conformant or not – note that if the spelling checker admits as valid a given Portuguese text, this does not mean that the given text is an element of the Portuguese language. Similarly, a Java parser is a metamodel that checks whether their programs are conformant or not – note that if the Java parser admits as valid a program, this does not mean that the program is an element of the Java language; by the contrary, a program is valid according to a grammar. The same problem occurs with the modelling language and its models. In fact, one can argue that a model conforms to a chosen metamodel to represent a modelling language. Similarly, one can think in a language induced from metamodel, in which the conformance with the metamodel ensures that a model is an element of the language induced from metamodel.

Figure 2.7 illustrates the concepts of metamodel, language and model, highlighting the relationships between these concepts. In [30], Jean-Marie Favre introduces the *Megamodel* to model the notion of metamodel, language and model, which contributed a great deal to avoid the “meta-muddle”. In [33], Bézivin defines a *Megamodel* as "*a model which elements represent models, metamodels and other global entities*".

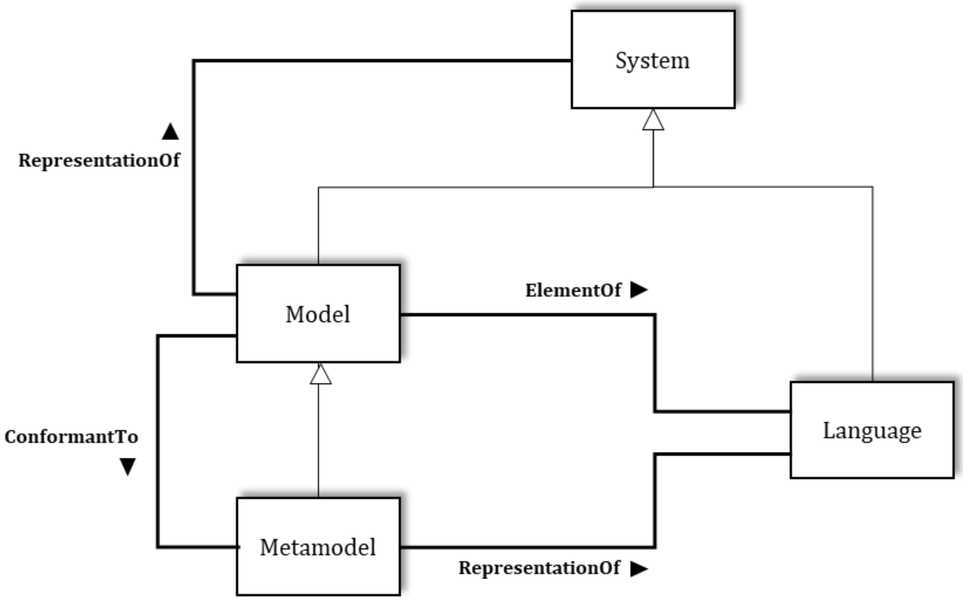


Figure 2.7 The Mega-model

The definitions of the concepts of language and metamodel are definitively very weak, however, the understanding regarding these concepts is "good enough" for the purposes of this dissertation.

2.4.2.2 Definitions

The notion of metamodel is poorly defined in software engineering. As pointed out by Jean-Marie Favre in [28], there is some confusion around the notion of metamodel; an example can be found on the help documentation on Eclipse Modelling Framework: "*A metamodel is simply the model of a model*" [24]. With the emergence of metamodeling as a crucial area of software engineering, it is expected to progressively overcome the misunderstanding and confusion.

OMG stated in [34] that "*a metamodel is a model used to model modeling itself*", which has contributed to the "meta-muddle". This definition, unfortunately, came to replace the previous one presented in [35]: "*A meta-model is a model that defines the language for expressing a model*", which is more coherent with the definitions previously mentioned.

Seidewitz provides another definition [26]:

> *"A metamodel is a **specification model** for a class of SUS where each SUS in the **class** is itself a valid model expressed in a certain **modelling language**."*

This last definition, though valuable, has also flaws. Yet, it is reasonable to accept that a meta-model can be "descriptive" and not only "prescriptive", just like other models. Note that from this last definition we can identify the notion of class to refer to the set of models that constitute the modelling language.

The most conciseness definition is proposed in OMG's MDA Guide [27]:

> *"**metamodel**: a model of models"*

Although this last definition is correct, those who missed the last "s" may misinterpret that "*a metamodel is a model of a model*".

2.4.3 Model-Driven Development

Model-Driven Development (MDD) is one of the most prominent paradigms of contemporary software development in which models and model technologies are primary artefacts to raise the level of abstraction at which developers create and evolve software, with the goal of both simplifying and formalizing the various activities and tasks in the software development process [16], [17], [36]. MDD paradigm addresses a core set of problems which are present in software development – complexity and ability to change [16]. According to Selic in [17], the essence of MDD consists of two key resolutions:

– **Raising the level of Abstraction** of specifications to reason about a problem domain by using modelling languages with higher-level and better-behaved constructs.

– **Raising the level of Automation** by using computer-based tools and integrated environments.

Models, modelling, and model transformation are the heart of MDD [20]. Modelling and models are the key tools in software engineering. However, there is a big difference between "*what models represent*" and "*how there are used*". Martin Fowler in [37] states that "*most new ideas in software developments are really new variations on old ideas*". In fact, the idea of MDD arose from early experiments with automatic code generation from computer-based software models [17]. The idea of MDD was later expanded with the advent of Computer Aided Software Engineering (CASE) at the beginning of the nineties

[15]. However, CASE tools have proven itself remarkable ineffective. As a result, not only the tools but also the model-based software development approach were abandoned. Eventually, the weaknesses of CASE tools led to the formation of the OMG's MDA initiative. The OMG's MDA initiative has contributed a great deal to the interest in software modelling and model-driven techniques [15]. MDA comprehends a set of OMG standards that enables the specification of models and their transformation into other models and complete systems while separating the specification of the operation of a system from the implementation [18], [27]. The result is an architecture that is not tied to any language, vendor, or platform [16].

The core of OMG's MDA, at the middle of Figure 2.8, is based on OMG's well-known modelling standards: UML, Meta Object Facility (MOF) and Common Warehouse Metamodel (CWM) [21].

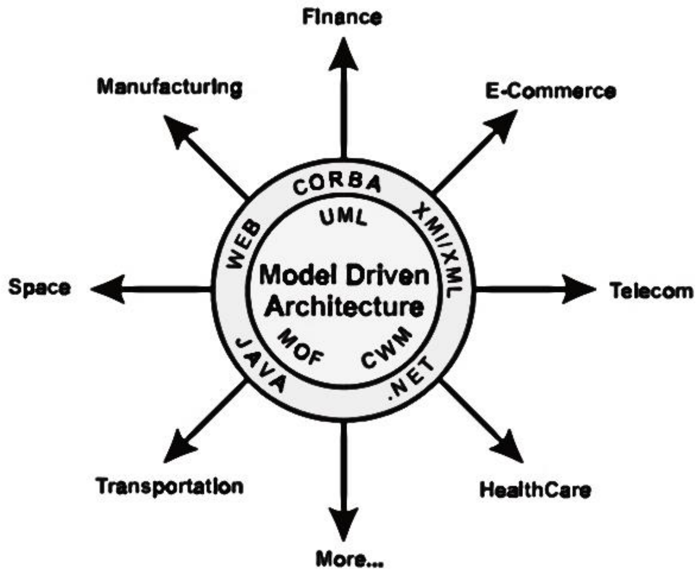


Figure 2.8 OMG's MDA

According to Brown et al. in [20], the four principles underlying the OMG's view of MDA are as follows:

1. Models expressed in a well-defined notation are fundamental to understand systems for enterprise-scale solutions.
2. Building systems can be prearranged around a set of models by imposing a series of transformations between models organized into an architectural framework of layers and transformations.

3. A formal underpinning for describing models in a set of metamodels facilitates meaningful integration and transformation among models so that automation through tools is possible.
4. Industry standards provide openness to consumers, and foster competition among vendors.

The primary benefits of MDA are portability, cross-platform interoperability, reusability and productivity through architectural separation of concerns [16]. Other goals are improved application quality, the rapid inclusion of new technology, reduced cost and development time.

2.5 OMG'S METAMODELING FRAMEWORK

The Object Management Group (OMG) is the world's major international, open membership, non-profit technology standards consortium that creates and maintains software interoperability specifications to help computer users solve integration problems, founded in 1986 [18], [38]. As of 1986, OMG has proven to be able to stay abreast of evolving business and IT needs and remain a foremost force in the industry [39]. By encouraging industrywide adoption of guidelines and object management specifications, OMG fosters the software development that supports open architecture, enabling multiple systems to work together [40].

Although OMG was originally established for the standardization and promotion of distributed OO systems, it is now focused on modelling and model-based standards, providing the IT industry with a proven process for establishing and promoting model-based standards. OMG's model-based standards extend beyond programming, enabling powerful visual design, execution, and maintenance of software [39].

OMG's model-based standards for analysis and design include Unified Modelling Language (UML) and the repository standard Meta-Object Facility (MOF) [41]. The UML is a result of fusing the best features of the various OO approaches into one modelling language and notation [42]. Adopted as an OMG specification in 1996, it represents a collection of best engineering practices that have proven best and most successful exemplar in the modelling of large and complex systems and is a well-defined, widely accepted response to business and IT needs. As a result, UML became the first *de facto* standard for the IT industry [41], [42]. The MOF is OMG's standard for metamodeling and metadata repositories and lies at the core of MDA. Fully integrated with UML, it uses the UML notation for describing repository metamodels. MOF is basically a meta-metamodel, or model of the metamodel (named an *ontology*[2]) capable of defining a common, abstract language for specifying metamodels. Although UML existed before the MOF, UML is an instance of the MOF [41]. This is because UML was originally not formally defined. The MOF was defined later to specify UML formally based on the MOF.

The OMG's metamodeling framework is a model-driven, OO framework for specifying, managing, interchanging, constructing, and integrating models in software systems. The aim of the framework is to support any type of model and to allow new types of models to be added as required. To achieve this, the framework uses a four-layer metamodeling architecture proposed by OMG.

2.5.1 Design Principles

The *framework* has been architected according to the following design principles [44]:

4. **Modularity**. The principle of strong cohesion and loose coupling, introduced by Larry Constantine, is applied to group constructs into packages and organize features into meta-classes.
5. **Layering**. Layering is applied in two different ways: 1) to separate the core constructs from the higher-level constructs that use them; 2) to consistently apply the 4-layer metamodel architectural pattern to separate concerns across layers of abstraction.
6. **Partitioning**. Partitioning is used to organize conceptual areas within the same layer.
7. **Extensibility**. The UML can be extended in two ways: 1) to define a new dialect by using *profiles*; 2) A new language related to UML can be specified by reusing part of UML's metamodel library and augmenting with appropriate meta-classes and meta-relationships.

[2] *Ontologies* are "*formal explicit specifications of a shared conceptualization*" [43].

8. **Reuse**. A metamodel library is provided that is reused to define the UML metamodel, as well as other architecturally related metamodels, such as the Meta Object Facility (MOF) and the Common Warehouse Metamodel (CWM).

2.5.2 The Four-Layer Metamodeling Architecture

The four-layer metamodeling architecture proposed by OMG consists of a hierarchy of model layers where each (except the top) is characterized as an *instance of* the layer above [31], [45]:

- **M3 Layer**: The foundation of the four-layer metamodeling architecture. The MOF, a self-describing meta-metamodel capable of defining the language for specifying metamodels.
- **M2 Layer**: The metamodel capable of defining a language for specifying models, that is, the structure and semantics. Examples of metamodels are UML and CWM.
- **M1 Layer**: The model capable of defining languages that describe semantic domains, that is, the organization and behaviour of a system.
- **M0 Layer**: The concrete data objects the system is designed to manipulate at some point in time. The run-time instances of model elements defined in a model.

Figure 2.9 illustrates an example of the four-layer metamodeling hierarchy and how these layers relate to each other [44], [46]. Note that the information on each layer is a more abstract representation of the information at the layer below it.

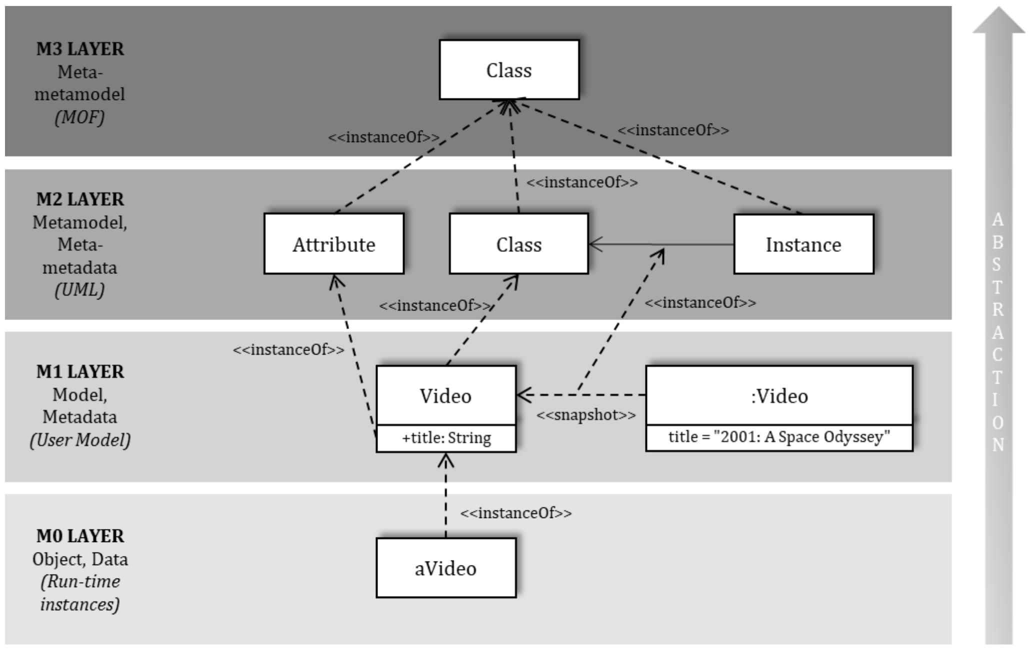


Figure 2.9 An example of the four-layer metamodeling hierarchy

The four-layer metamodeling architecture provides the capability of representing information at multiple layers of abstraction [46]. By describing the information on each layer according to the model on the layer above, it increases the precision and correctness of the specification [44]. Moreover, it has the advantage of easily accommodating new modelling standards as MOF instances at the M2 layer [31].

2.5.3 Meta Object Facility

The Meta Object Facility (MOF) is an OMG standard for metamodeling and metadata repositories and constitutes the core of the MDA [27], [46]. MOF is basically a meta-metamodel, or model of the metamodel (named an *ontology*[2]) capable of defining a common, abstract language for specifying metamodels [41]. Fully integrated with UML, it uses the UML notation for describing repository metamodels, which can cause confusion since UML is an instance of the MOF. Even though UML existed before the MOF, UML was originally not formally defined – that is, it was only defined in "*words*" [15]. The MOF was defined later to specify UML formally based on the MOF (UML 2.0). Thus, MOF is a self-describing meta-metamodel. Formally, this is possible through the specification of packages/namespaces for model elements and the MOF layered architecture where model elements can be in different meta-layers. The MOF defines the model elements, syntax, and structure of metamodels that are used to construct them [41]. The model elements are defined in terms of abstract syntax, well-formedness rules, and semantics.

The MOF is not only important as a formal, solid foundation of metamodels, it is also essential for the OMG's standardization efforts and the tool to define the meta-metamodel completely, formally, and correctly and to guarantee software interoperability and portability [15], [41].

2.5.4 Common Warehouse Metamodel

The Common Warehouse Metamodel (CWM) is an OMG standard for metadata interchange in the data warehousing and business analysis environments, proposed by IBM, Oracle, and Unisys in 1998 [41]. CWM extends the OMG's well-known metamodeling architecture defining a common metamodel and interchange mechanism for interoperable databases, tools, and applications. The CWM specification is built upon existing OMG technology standards for metadata and repository architectures – that is, UML, MOF, and XML-based Metadata Interchange (XMI). The CWM is organized in small, understandable packages prearranged into five functional layers, where each package addresses the modelling requirements of some subdomain of data warehousing and business analysis as shown in Figure 2.10.

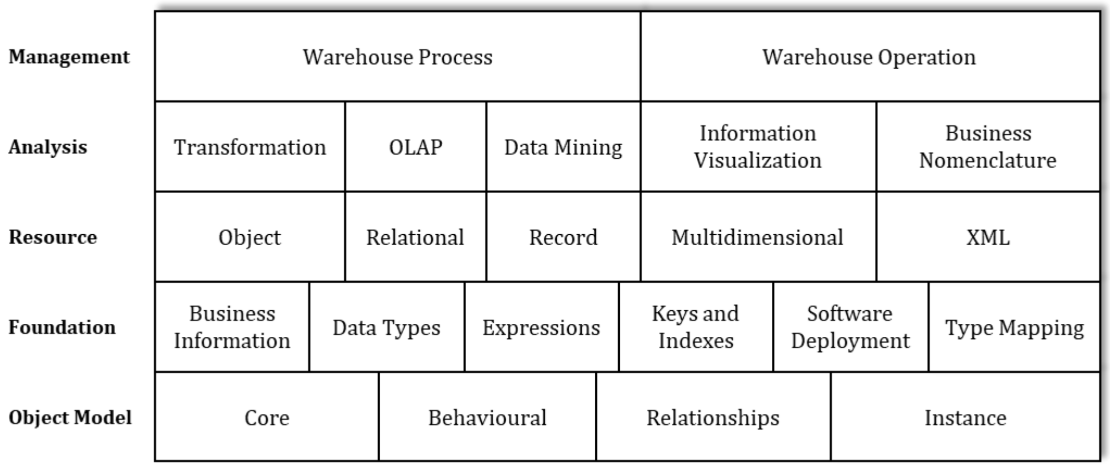


Figure 2.10 CWM packages prearranged into five functional layers

- ***Object Model Layer***, which represents the lowest layer in the CWM, a subset of the UML used as the base metamodel for all other CWM packages.

- ***Foundation Layer***, which consists of metamodels that extends *Object Layer* to facilitate the modelling of common services.

- ***Resource Layer***, which consists of metamodels used to construct metadata defining relational, record-oriented and multidimensional databases. The layer contains metamodel packages such as *Relational*, *Record*, *Multidimensional* and *XML* that extend both the *Object Model* and *Foundation* layers. The *Relational package* represents the largest single package in CWM and is SQL-compliant, wherein the SQL logical aspects are described – that is, tables, columns, views, trigger, procedures, and so forth.
- ***Analysis Layer***, which consists of metamodels for defining business analysis metadata on top of the various resource models.
- ***Management Layer***, which consists of metamodels for the modelling of common data warehouse management.

CHAPTER 3

DEVELOPMENT

This chapter presents the software requirements gathering and analysis as the first step in developing this dissertation work. The outcome consists of a comprehensive set of functional requirements (FRs) and non-functional requirements (NFRs). The system design is then established from the software requirements. The software to satisfy the system design has to be produced. As a result, we provide a brief overview of the system implementation.

3.1 FUNCTIONAL AND NON-FUNCTIONAL REQUIREMENTS

The first step in developing a successful software project is to properly capture the software requirements in the process of *requirement engineering* (RE). The output of this phase is a comprehensive set of functional requirements (FRs) and non-functional requirements (NFRs). Functional requirements define what the system should do, whereas non-functional requirements provide constraints and guides to the system architecture [47].

The following challenges and issues faced by the IT developers were gathered:

1. Because the data transfer is ineffective, the server receives all data from the connector and then saves it in memory until all data is sent to the client.
2. The system needs to support more than one *Result Set*.
3. The connector does not have a means of sharing important information with the client.
4. The system has limited possibilities regarding how to test a connection with the data provider according to connector properties.
5. The system does not support different types of parameter's direction, such as OUT, INOUT and RETURN.

6. The Connector API[3] is difficult of learning, lacks understandability – developers do not understand how it works –, and hides functionality. Furthermore, the database schema was designed for static schemas making difficult to implement dynamic schemas, and the conditional expression is not well-made containing several exceptions and missing important information.
7. The system needs to provide means of throwing meaningful exceptions. A set of exceptions with a specified error code to get additional information is needed.
8. Multiline INSERT is not supported.
9. The connectors cannot set a maximum length for a string attribute; therefore, it is treated as an "unlimited" string. As a result, the server cannot predict the size and allocate memory.
10. The system needs to improve compatibility with T-SQL.
11. The system needs to have a possibility to create custom stored procedures and views.
12. The system needs to have detailed statistics of executed queries.

After understanding the challenges and issues, we discussed with the IT developers the functional requirements, explicitly, the features and functions of the Connect Bridge Development Kit (CBDK). In other words, what does the system do? In the end, they agreed that CBDK has the functional requirements shown in Table 3.1.

Table 3.1 Connect Bridge Development Kit FRs

No.	Requirement
FR-1	The software will support T–SQL data manipulation commands
FR-2	The software will support multiline INSERT
FR-3	The software will support all possible kinds of parameter's direction
FR-4	The software will provide error handling (e.g. custom exceptions)
FR-5	The software will support more than one *Result Set*
FR-6	The software shall allow defining most database objects

[3] MG Framework

FR-7	The software must provide detailed statistics of executed queries (e.g. number of affected rows)
FR-8	The software will support static and dynamic schemas
FR-9	The software will support large objects (BLOBs and CLOBs)
FR-10	The software shall provide means of returning the last inserted rows
FR-11	The software must maintain backwards compatibility with "old" connectors
FR-12	The software shall allow accessing database objects by unique name and ordinal position

We reviewed these functional requirements in terms of feasibility, completeness, and accessibility, and obtained the necessary sign-off.

Then, discussions with the IT developers revealed the non-functional requirements described in Table 3.2.

Table 3.2 Connect Bridge Development Kit NFRs

No.	Requirement
NFR-1	The software will include technical documentation
NFR-2	The software will include user documentation
NFR-3	The software will be easy to learn for both novices and seasoned developers
NFR-4	Developers shall be able to develop the connector's basic functionality in less than 10-man-day
NFR-5	The software shall provide developers with necessary functionalities and to permit developers with different needs to adapt and use the system
NFR-6	The software shall allow detecting and managing errors without leaving the error undetected
NFR-7	The software will work with the minimal use of resources (i.e. memory)

We reviewed these non-functional requirements in terms of feasibility, completeness, and impact on the system design, and obtained the necessary sign-off. These requirements were included in the following categories: usability, documentation, performance, and maintainability.

3.2 Architectural Design

The design goal of the CBDK is to provide a simple, public API for the creation of connectors to integrate with CB Server, which supports relational to object and relational to relational mapping. In this setting, the API exposes a set of common interfaces to migrate a data source (relational or non-relational) to SQL-based, relational database.

This section describes the architectural design of the API from the requirements summarized in Section 3.1. First, we design the metamodel that defines the abstract syntax of the metadata. An OMG specification is used as a formal reference to design the metamodel [34], [48]. Second, we design the command that defines the abstract syntax of the data manipulation language. The *Pipes and Filters* style is used to design the command. Finally, we design the connector to define the abstract syntax of a functional connector, as well as the exceptions that define a rich set of custom exceptions with additional information (e.g. error code, message, and so on), respectively. For modelling purposes, UML concepts and drawing conventions, which are detailed in Appendix A, are used.

3.2.1 Designing the Connector Metamodel

The main goal that is pursued with the Connector Metamodel (CM) is to provide a semi-formal model of a language for the definition of relational and non-relational data sources. To achieve this goal in an effective and consistent manner, we follow the MDD approach since it improves software development lifecycle by providing a higher level of abstraction for metamodeling [49]. In this setting, the MDA initiative by the OMG provides several important OMG's well-known modelling standards such as MOF, UML and CWM, which are designed according to the four-layer metamodeling architecture. MOF serves as the common model of CM.

At its core, the CM extends the OMG's established four-layer metamodeling architecture that customizes it for the needs and purposes of the relational or non-relational data sources. As such, it has a modular architecture built on the object-oriented basis that describes the parts from which it is constructed and how those parts are arranged. Figure 3.1 illustrates how the CM's architecture conforms to the classic four-layer metamodeling architecture. Additionally, CM supports a *model-driven approach*, in which formal models are constructed according to the specifications of the metamodel.

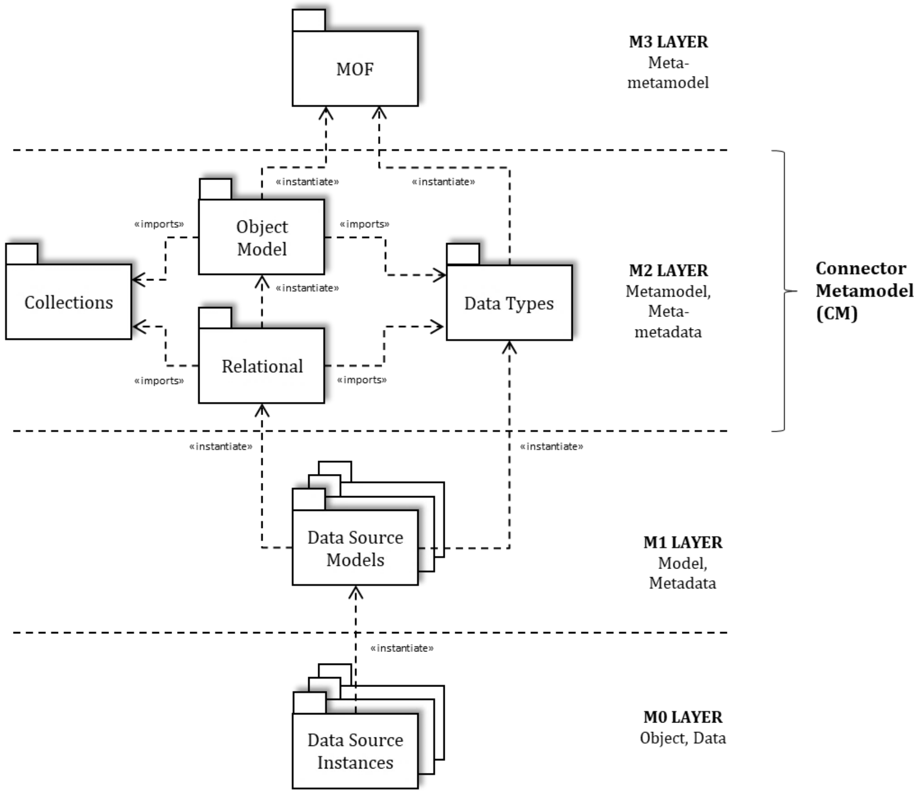


Figure 3.1 Connector Metamodel's Architecture

The CM results in class diagrams developed using MOF modelling constructs. We designed the metamodel in a modular, or *packages*, way. The metamodel is defined at OMG's M2 level and is organized in four small, logical *packages*: **Object Model**, **Data Types**, **Relational**, and **Collections** as shown in Figure 3.1. Each package covers dissimilar aspects of the metamodel.

3.2.1.1 Object Model Package

The Object Model package contains the basic concepts used to describe the structure of object-oriented data sources. We used the semantics defined in the M3 layer (MOF Model) for the Object Model. Then, a common vocabulary was studied for a better understanding of "new" basic concepts. Finally, several rules were applied which resulted in removals, additions, and replacements. The semantics in the Object Model package are as follows (see Figure 3.2):

Element (from MOF). The foundation for all other modelling concepts in the metamodel. An *Element* – a constituted of a model – has no attributes and provides a single point from which all objects can be found.

Descriptor. A textual annotation that can be attached to an element. Every kind of element may own a descriptor. The *descriptor* for an element adds no semantics but may represent information useful to the *reader* of the model.

Named Element (from MOF). An element in the model that has a name. The name is used for identification of the *Named Element* within the *Namespaces* wherein it is defined or accessible.

Namespace (from MOF). An element in the model that contains a set of *owned elements* that can be identified by name within the *Namespace*. Typically, nearly every element in the model will be owned by some *Namespace*, which is a *Named Element*. The composite association between *Namespace* and *Named Element* allows *Namespaces* to own *Named Elements*, and consequently, other *Namespaces*. This association allows *Named Elements* to be organized in hierarchical, or tree-like, arrangements wherein the parent *Namespace* is said to own its child *Named Elements*. Accordingly, if an element of a *Namespace* with the *name* "Production" is a *Named Element* with the *name* "Product", the fully *qualified name* of a *Named Element* is "*Production.Product*".

Constraint (from MOF). An assertion that indicates a restriction that must be satisfied by any valid realization of the model containing the constraint. A *constraint* is attached to a set of *constrained elements*, and it represents supplementary semantic information about those elements.

Data Type (from MOF). A named element that specifies a set of allowed data values. A *Data Type* constrains the values represented by a *Typed Element*.

Typed Element (from MOF). A named element that has a *type* specified for it.

Ordinal Element. A named element that has an *ordinal* position specified for it, which stipulates where an element is in an order.

Property (from MOF). A structural element that represents an attribute.

Queryable Element. A namespace that can be manipulated by the SQL language. As it is a *Namespace,* allows *Named Elements* to be collected in hierarchies. Typically, a *Queryable Element* owns a set of *Properties* and/or *Constraints*. However, this restriction is not implied in the model description. Considering this, we end up with an even more flexible model.

Model. A view of a physical system. In other words, an abstraction of the physical system.

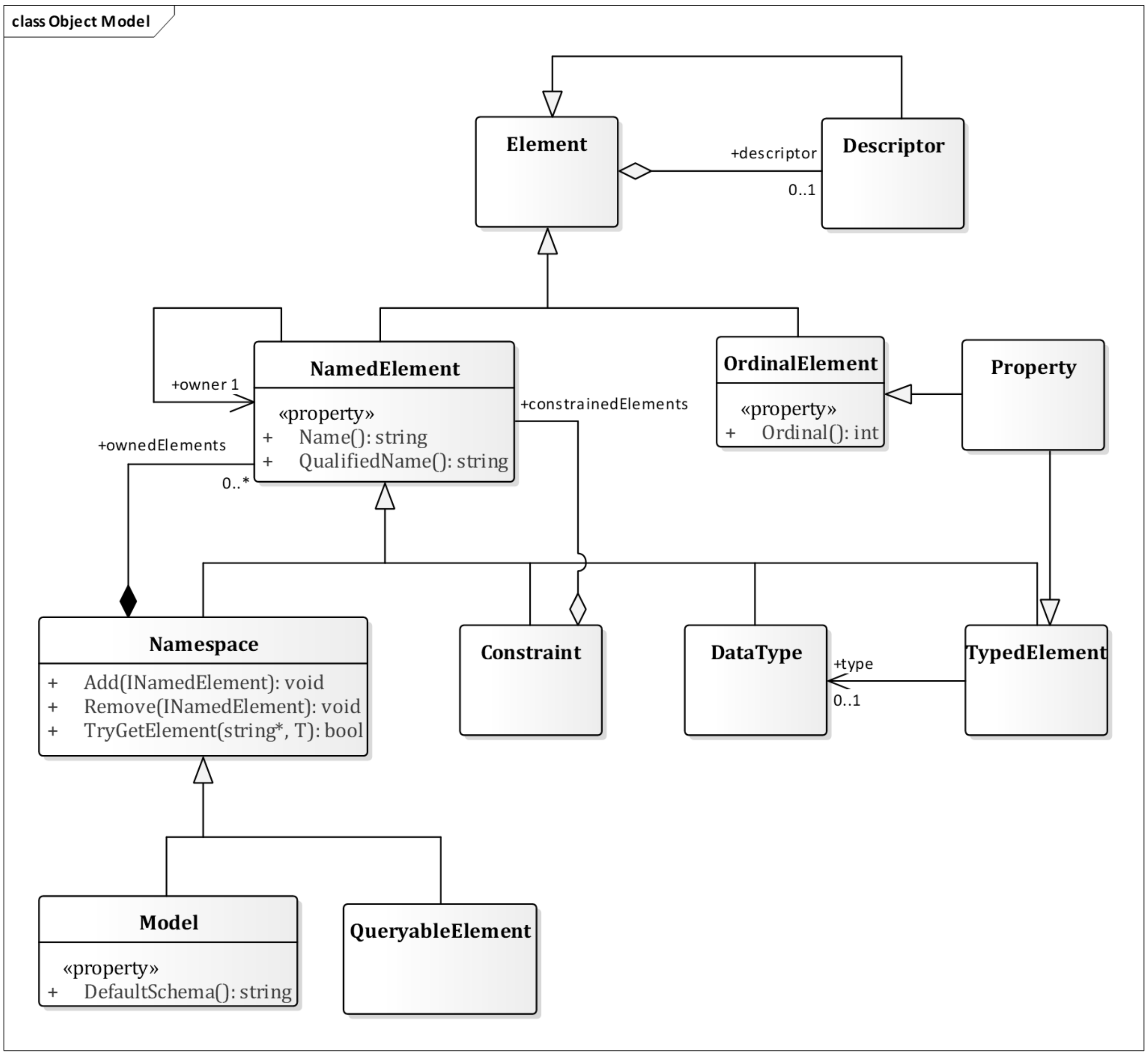


Figure 3.2 Abstract Syntax for Object Model Package

The *Named Element*, *Namespace*, *Data Type*, *Constraint*, and *Property* semantics in the Object Model package are the pillars of CM. Note that the Object Model package represents a subset of the MOF Model and is used as the base metamodel.

The main goal that is pursued with the Object Model package is to provide a familiar, common set of metamodel concepts, relationships, and constraints for defining mainly object-oriented data sources. These concepts create an environment wherein the focal point is their purposes. The primary role of the Object Model package is to make clear why some commonly used concepts, such as Table and Column, supports object-oriented concepts like Entity and Attribute. Therefore, if an object-oriented data source cannot be visualized or created through the Relational package, the Object Model package can be used.

3.2.1.2 Data Types Package

The notion of *data type* – a named type that consists of a set of allowed data values – is central to most modern programming languages and database systems. The Data Types package provides the infrastructure required to support the definition of primitive, parametric and complex data types. The semantics in the Data Types package are as follows (see Figure 3.3):

Parametric Type. A data type – parameterized over type – that is determined at runtime.

Complex Type. A data type for defining structured data in the form of one or more properties, each of which has a data type and describes an instance of a data type. A complex type is a *Namespace*; as such, it has the capability of owning *Named Elements*. However, some boundaries must be taken into consideration like *Complex Types* can only exist as a type; *Constraints* cannot be defined inside of *Complex Types*; and *Complex Types* cannot participate in relationships. The primary, advanced feature of a *Complex Type* is to include a set of methods.

Table Type. A complex data type for defining the structured data of a specific *table*.

JSON Type*.* A complex data type for accessing data in a JavaScript Object Notation (JSON) document. The *JSON Type* provides several advantages over storing JSON-format strings in a string *Property*, such as automatic validation of JSON documents stored in a JSON *Property*, and optimized storage format.

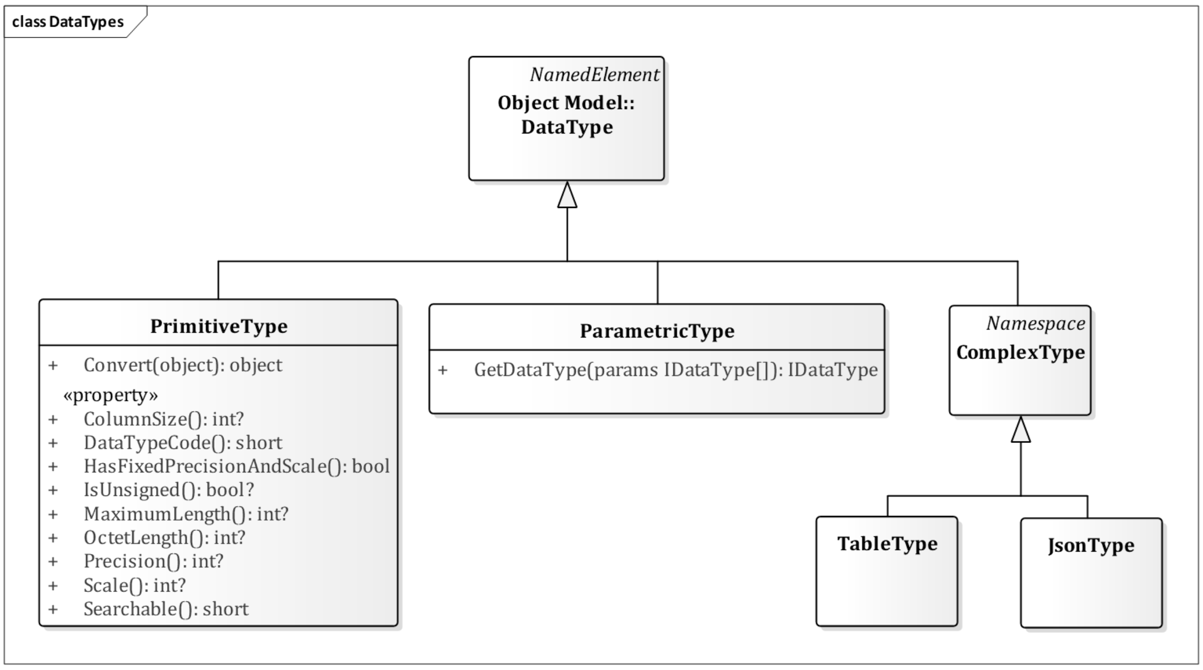


Figure 3.3 Abstract Syntax for Data Types Package

Primitive Type (from MOF). A predefined data type without any substructure. Within the boundaries imposed by CB Server, we defined the *primitive data types*. A *primitive data type* definition requires information, such as *precision*, *scale*, *maximum length*, *octet length*, and/or *data type code*; which is used to inform about how to handle with a *Typed Element* based on its data *type*. Additionally, we defined a noteworthy method for converting a given value object to the *common language runtime* (CLR) type.

The Data Types package provides a familiar, common set of concepts for defining both object-oriented and relational data types. In its first definition, we only considered primitive data types – *decimal*, *Int16*, *Int32*, *Int64*, *UInt16*, *UInt32*, *UInt64*, *Single*, *char*, *string*, *DateTime*, *Boolean*, *double*, *byte array*, and *byte*. Then, we introduced the parametric data type as a consequence of having *Typed Elements* whose output data type depends on the input; thus, the type could be determined at runtime as desired. Along with the parametric data type, we introduced a “new” primitive type for generalization of all other types: *Any Type*, which is the type to which all other types conform. Next, we introduced the complex data types as a result of having *Typed Elements* whose *type* represents a *table type* or a *JSON type*. Both complex data types do not have defined methods. Finally, we introduced a “new” primitive type: *signed byte*, which was missing.

3.2.1.3 Relational Package

Relational database concepts can be described by the Relational package. The metamodel for the Relational package, as the name suggests, supports the description of SQL-based relational databases, including its object-oriented extensions.

Though the Relational package is adequate for the definition of relational data, it is unlikely that it can hold a complete description of any commercial relational database data model. This is because the metamodel supports the logical aspects – tables, views, columns, procedures, functions, and so on – but not the physical aspects – file locations, attributes, indexes, and similar characteristics – of relational databases. These aspects of relational databases are not commonly the kind of information that needs to be defined. The semantics in the Relational package, which satisfies FR-6, are as follows (see Figure 3.4):

Schema. A logical grouping of database objects. It is the conceptual organization of the relational database. A schema is *metadata*. It may own *tables*, *views*, *functions*, *data types*, and *procedures*.

Table. A logical structure identifiable within the schema. A table may own *columns* and *constraints*.

View. A virtual table who provides an alternative way of looking at the data in one or more tables. A *view* can be used to perform joins and simplify multiple tables into a single virtual table and hide complexity. A view may own *columns*.

Procedure. A namespace that accepts input parameters and possibly will return one or more values. A procedure is mainly used to perform an action. A procedure may own *parameters*.

Function. A namespace that accepts input parameters and must return one or more values. A function may own *parameters*.

Column. A property owned by a *table* or *view*.

Parameter. A *property* owned by a *function* or *procedure*. The *direction* property specifies whether a value is passed into, out of, or both into and out of the owning *function* or *procedure*. Thru the *direction* property, FR-3 is fulfilled.

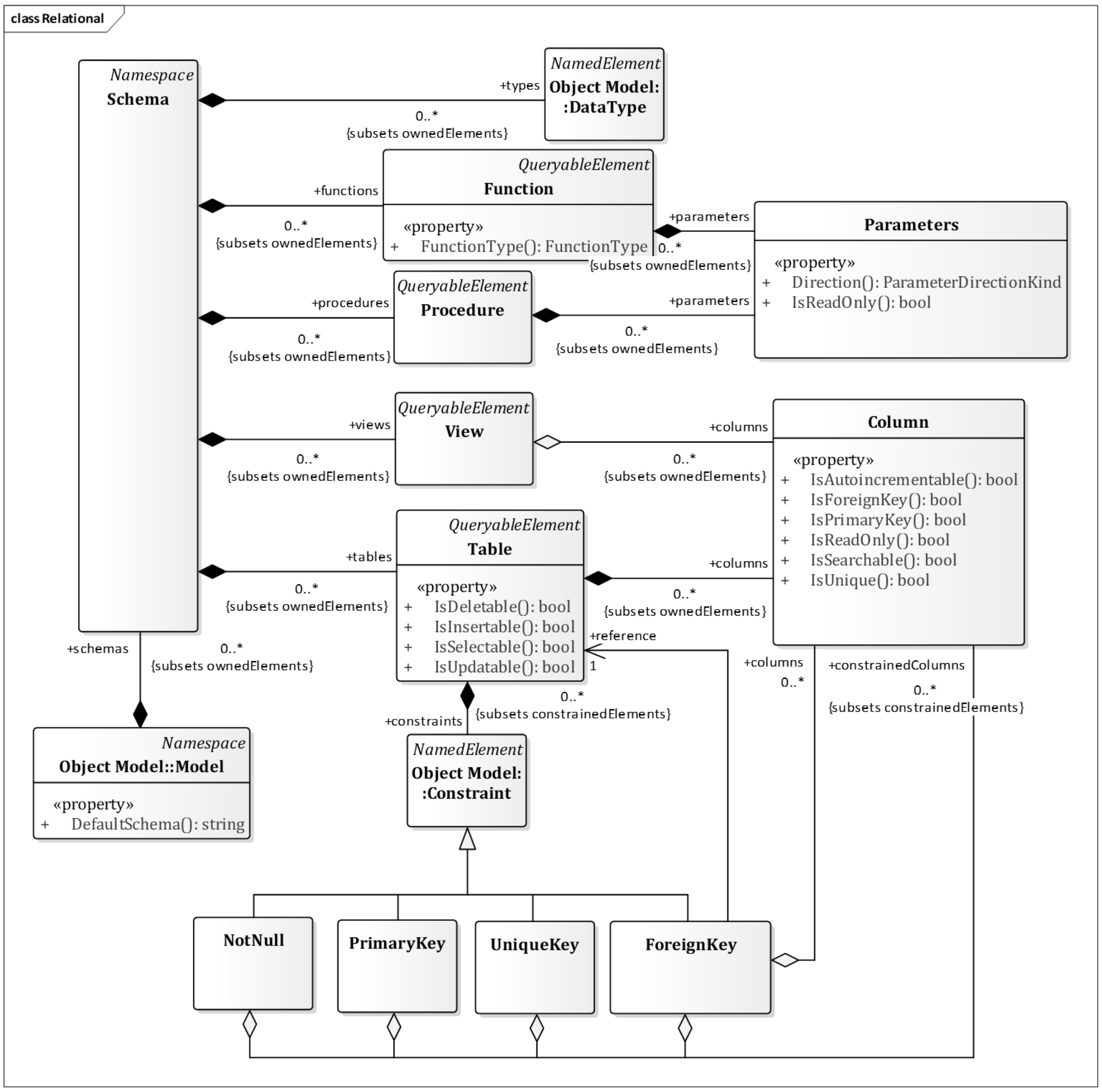


Figure 3.4 Abstract Syntax for Relational package

Not Null. Ensures that a *constrained element* has no NULL value.

Primary Key. Enforces data integrity and ensures unique identification.

***Unique Key*.** Ensures that all values of a constrained element are dissimilar.

Foreign Key. Enforces data integrity and establishes a link between two *Named Elements*.

The Relational package provides a familiar, common set of concepts for defining relational databases. The envisioned outcome is to empower the definition of relational and object-oriented data sources through the Relational package.

3.2.1.4 Collections Package

Several modern programming languages and DBMSs offer means of creating and manipulating collections of related objects. A program may create and reference many objects. A database may hold a greater collection of objects. To provide this significant characteristic in an effective and consistent manner, the Collections package provides the infrastructure required to support the definition of collections of related objects which can be individually accessed by index or key. The semantics in the Collections package, which satisfies FR-12, are presented in Figure 3.5.

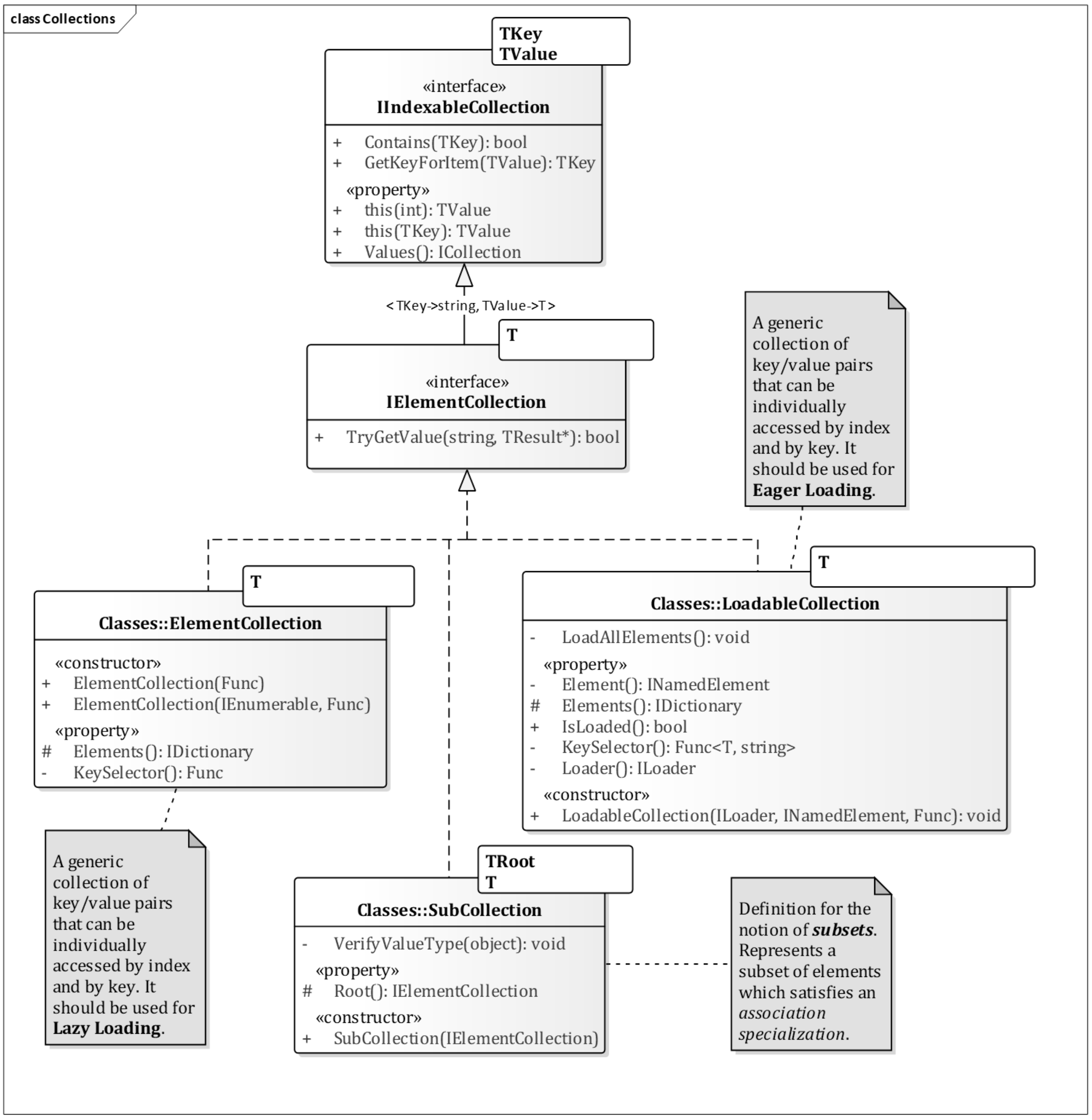

class Collections
TKey
TValue
«interface»
IIndexableCollection
+ Contains(TKey): bool
+ GetKeyForItem(TValue): TKey
«property»
+ this(int): TValue
+ this(TKey): TValue
+ Values(): ICollection
< TKey->string, TValue->T >
T
«interface»
IElementCollection
+ TryGetValue(string, TResult*): bool
A generic collection of key/value pairs that can be individually accessed by index and by key. It should be used for Eager Loading.
T
Classes::ElementCollection
«constructor»
+ ElementCollection(Func)
+ ElementCollection(IEnumerable, Func)
«property»
# Elements(): IDictionary
- KeySelector(): Func
T
Classes::LoadableCollection
- LoadAllElements(): void
«property»
- Element(): INamedElement
# Elements(): IDictionary
+ IsLoaded(): bool
- KeySelector(): Func<T, string>
- Loader(): ILoader
«constructor»
+ LoadableCollection(ILoader, INamedElement, Func): void
A generic collection of key/value pairs that can be individually accessed by index and by key. It should be used for Lazy Loading.
TRoot
T
Classes::SubCollection
- VerifyValueType(object): void
«property»
# Root(): IElementCollection
«constructor»
+ SubCollection(IElementCollection)
Definition for the notion of subsets. Represents a subset of elements which satisfies an association specialization.

Figure 3.5 Abstract Syntax for Collections package

Any modern object-oriented programming language provides this as well. So, what features does Collections package offer us? The major feature of Collections package is to provide fast search based on key and type, and index-based access; wherein the main enhancement is performance and ease of use. Additionally, the Collections package provides *eager* and *lazy loading* semantics to load related data. By means of *lazy load*, the performance improvements are significant if the initialization of the object is costly. See Appendix B.2.1 for a detailed explanation of this solution.

3.2.2 Designing the Command

As described throughout the previous chapters, when a user inputs an SQL statement or query through a client application, the SQL statement is collected by the CB Server for processing in a parse tree after the SQL statement has been evaluated according to the rules of a formal grammar. A parse tree is a data structure that represents a parsed statement. The parsed statement is expected to be formally defined into a formal model. As a result, a formal definition of the *semantics* of SQL queries has to be well-defined. To achieve this, we had to extract and consolidate several major concepts found in the SQL Grammar (see Appendix C) and use them as the basis for our formal model.

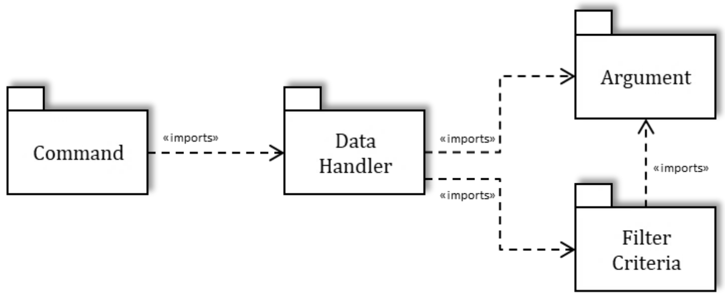


Figure 3.6 Package Diagram - Command packages

The Command, which satisfies FR-1, has a modular, or *packages*, architecture built on *T-SQL* basis that describes the parts from which an SQL statement is constructed and how those parts are arranged. The Command is organized in four substantial, understandable *packages*: **Command**, **Data Handler**, **Argument**, and **Filter criteria** as shown in a package diagram, Figure 3.6. Each package covers different parts of an SQL statement. This is shown by the following queries.

Q_1 **SELECT DISTINCT** <*argument*>
FROM <*reference table*>
WHERE <*filter criteria*>
GROUP BY <*argument*>
HAVING <*filter criteria*>
ORDER BY <*argument*>
LIMIT <*argument*> **OFFSET** <*argument*>

Q_2 **SELECT TOP** *5* <*argument*>
FROM <*reference table*>
WHERE NOT EXISTS <*subquery*>

3.2.2.1 Command Package

The Command package contains the basic concepts used to describe an SQL statement in accordance with T-SQL, which is a Microsoft's extension of SQL Language and expands on the SQL standard. This standard was selected because of its widespread acceptance in the database field and because it is the standard already used by the CB Server.

Although the Command package is suitable for the definition of the *semantics* of SQL statements, it is unlikely that it can hold a complete description of ANSI/ISO SQL standard. This is because Command package supports concepts which are consistent with the SQL grammar defined by CB Server. The core semantics in the Command package are as follows (see Figure 3.7):

> ***Command***. An SQL command that specifically relates to data handling. Some of these commands perform data definition functions; and other data manipulation functions. We only consider the *data manipulation commands* that allow inserting, updating, deleting, and retrieving data from the *data source*.
>
> ***Statement***. An SQL statement queries or manipulates data in an existing *data source*. A *Statement* is a *Command*. As such, it has the capability of handling data.
>
> ***Data Handler***. A data handler consists of a chain of sequential processing handlers arranged so the information flows in these handlers from the *sink* calling for data until the *source* between directly connected handlers. We follow the *Pipes and Filters* style – *pull* flow variant. The aspects described here are discussed in greater detail in the subsequent section. Note that a data handler is a data structure that represents a parsed statement. Every single *Data Handler* will be owned by some *Statement*.

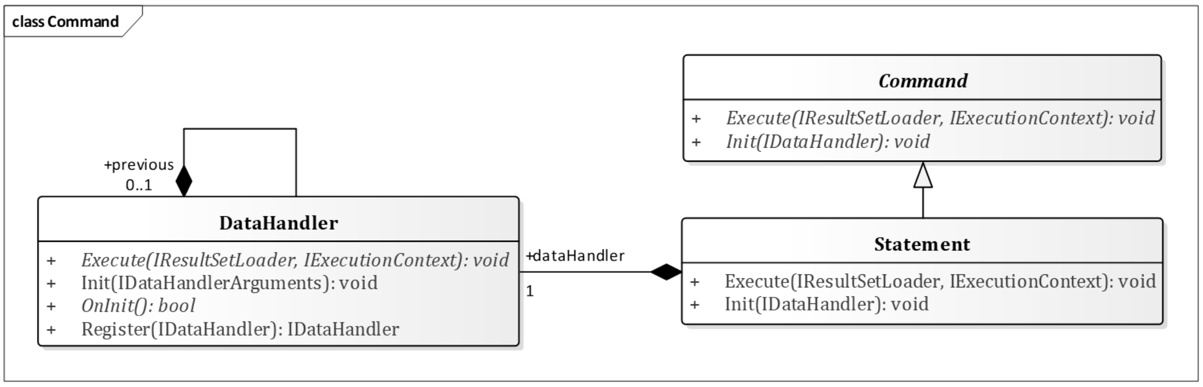

class Command
Command
+ Execute(IResultSetLoader, IExecutionContext): void
+ Init(IDataHandler): void
+previous
0..1
DataHandler
+ Execute(IResultSetLoader, IExecutionContext): void
+ Init(IDataHandlerArguments): void
+ OnInit(): bool
+ Register(IDataHandler): IDataHandler
+dataHandler
1
Statement
+ Execute(IResultSetLoader, IExecutionContext): void
+ Init(IDataHandler): void

Figure 3.7 Abstract Syntax for Command Package

3.2.2.2 Data Handler Package

The Data Handler package provides the infrastructure required to support the definition of a parsed statement – that is, the parts from which it is constructed and how those parts are arranged. We follow the *Pipes and Filters* style. This architectural style was selected because of its independence and flexibility. The *Pipes and Filters* style supports three ways by which information flows between directly connected handlers: push, pull, and pull/push. We selected the *pull* flow variant. The rationale behind the selection of the *pull* flow variant was as follows: 1) a request may generate large amount of data; 2) different sources may exist for a request (e.g. *Join* between two or more *sources*); and 3) the *handler* receives information by generating a request in which the information flows from the *sink* calling for data until the *source*. See Appendix B.1.1 for a detailed explanation of this solution. The core semantics in the Data Handler package are as follows (see Figure 3.8):

Data Source. A data handler in the chain of processing handlers. The responsibility of a source element is to provide input data.

Data Filter. A data handler in the chain of processing handlers that encapsulates a processing step (algorithm). The responsibilities of a filter element are: 1) get input data, 2) perform one or more operations on its input data, and 3) supply output data. The basic activities of a filter element, often combined in a single filter, are as follows:

1. Enrich input data;
2. Refine the input data (e.g. filter out “uninteresting” input, sort input);
3. Transform input data (e.g. perform calculation).

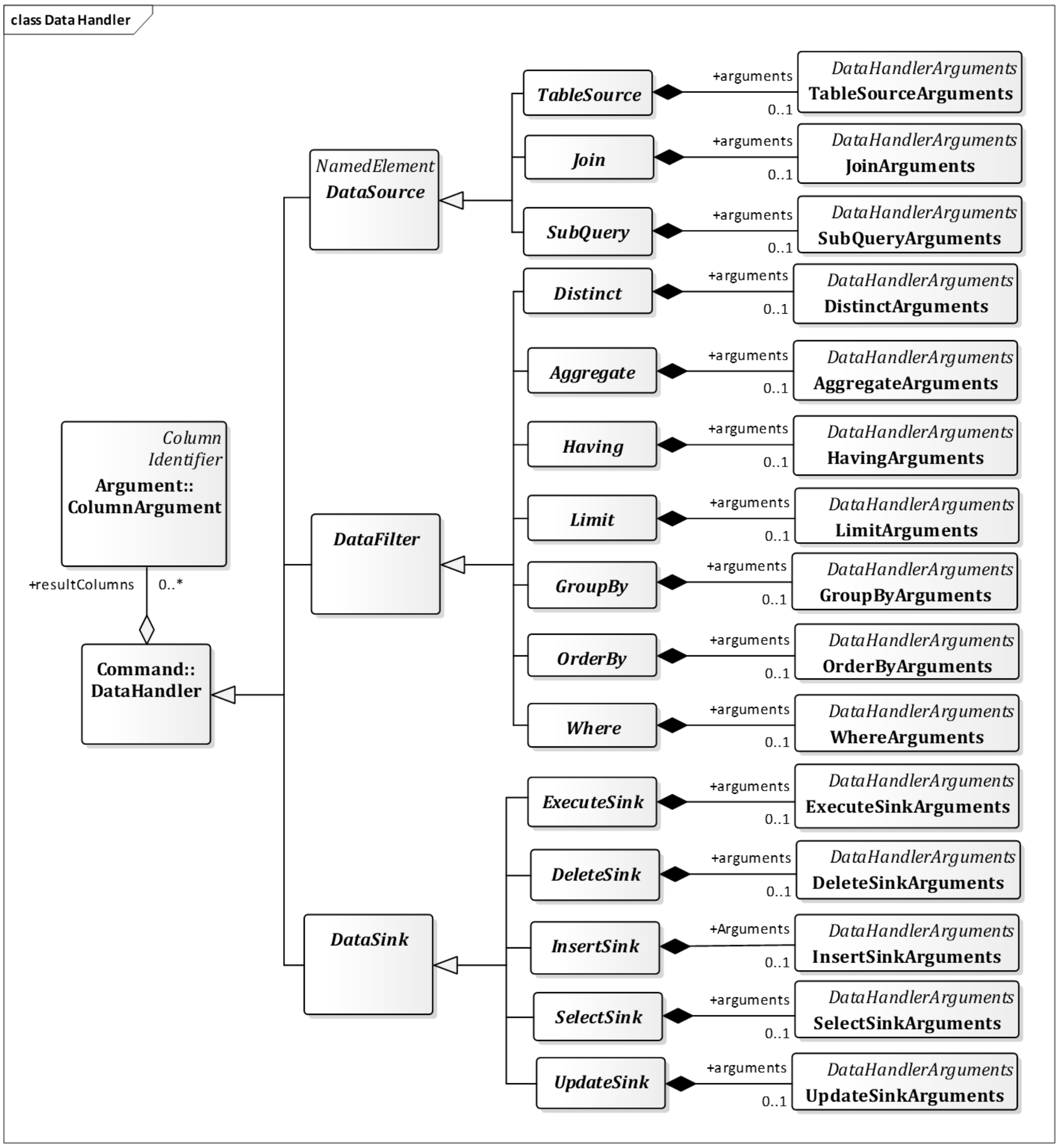


Figure 3.8 Abstract Syntax for Data Handler Package

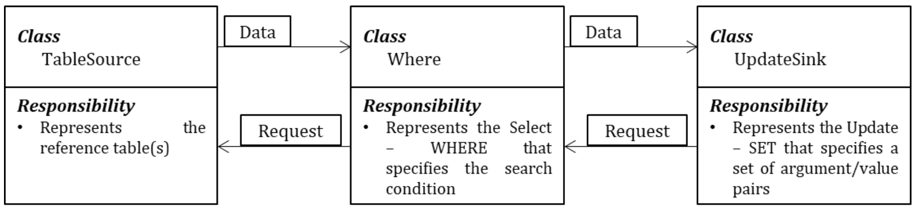
Data Sink. A data handler in the chain of processing handlers. The responsibility of a sink element is to consume the output data (that is, the result of the complete computation).

The SQL data manipulation commands – JOIN, WHERE, ORDERBY, SELECT, UPDATE, and so forth (see Figure 3.8) – are widely understood and will not be further discussed here. See Appendix D.1 for a complete definition of the abstract syntax for Data Handler package.

To further illustrate how can classes represent the parts from which an SQL statement is constructed and how those classes can be arranged, two concrete examples are given below in which the Class-Responsibility-Collaborator (CRC) model is used.

For example, for the following SQL statement, Figure 3.9 is given below.

Q$_1$ **UPDATE** Customers
SET Country= 'Portugal', City = 'Lisbon'
WHERE CustomerId= 1

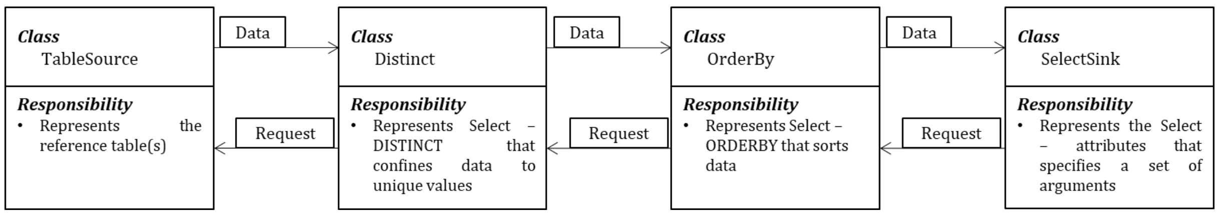


Class
TableSource
Responsibility
• Represents the reference table(s)
Data
Request
Class
Where
Responsibility
• Represents the Select – WHERE that specifies the search condition
Data
Request
Class
UpdateSink
Responsibility
• Represents the Update – SET that specifies a set of argument/value pairs

Figure 3.9 Transformation of Q$_1$ into a chain of sequential processing handlers

In yet another example, for the following SQL statement, Figure 3.10 is given below.

Q$_2$ **SELECT DISTINCT** *
FROM Customers
ORDER BY Country **ASC**



Class
TableSource
Responsibility
• Represents the reference table(s)
Data
Request
Class
Distinct
Responsibility
• Represents Select – DISTINCT that confines data to unique values
Data
Request
Class
OrderBy
Responsibility
• Represents Select – ORDERBY that sorts data
Data
Request
Class
SelectSink
Responsibility
• Represents the Select – attributes that specifies a set of arguments

Figure 3.10 Transformation of Q$_2$ into a chain of sequential processing handlers

We have defined that the data handler's state is captured and externalized so that it can be set only by the Server Layer and accessed by the Plugin Layer. As a result, we end up simplifying the data handler implementation details.

3.2.2.3 Argument Package

The notion of *data* – raw facts of interest to the user – is central to all modern programming languages and database systems. In fact, in database systems, at the intersection of every database table's column and row is a specific data item called a *value* from which we can extract meaningful relationships and trends. The Argument package provides the basic concepts used to describe the various kinds of values, as well as expressions and functions in a parsed statement. Remember that *functions* examine data and calculate a value based on the data, while *expressions* are an ordered combination of data values and operations that can be evaluated to produce a single value. The semantics in the Argument package are as follows (see Figure 3.11):

Argument. The foundation for all other modelling concepts in the *Argument* package. An argument is a typed element that may represent a *value*.

Identifier. An argument that may uniquely identify a *value*. SQL Identifiers are often used to represent aliases, constants, variables, column name and so forth.

Constant. An identifier that may represent a *value*. Logically enough, the *value* of a *constant* never changes. In some SQL statements a numeric, string or date data value has to be expressed in text form. For instance, in the following SQL statement the value for each column is specified in the **VALUES** clause:

```
Q1      INSERT INTO Department (DepartmentID, Name, GroupName, ModifiedDate)
        VALUES (115, 'Engineering', ‘Development’, '2018-05-13 14:15:07.000')
```

Constant data values are also used in *expressions*, such as in the following SQL statement:

```
Q2      SELECT *
         FROM [Product]
         WHERE [Price] > (2 * [Weight]) + 5
```

Variable. An identifier that may represent a *value*. Logically enough, the *value* of a *variable* may change. For instance, in the following SQL statement a numeric value is stored in a *variable* named *@price*:

```
Q1      UPDATE Product
          SET Price= @price
          WHERE ProductId= 151
```

Column Argument. An identifier that may represent a column-reference. A column contains values, one in each row of a table. SQL statements often refer to such values. For instance, in the following SQL statement:

```
Q1      SELECT Price
          FROM Product
          WHERE Size = ‘M’
```

Size is a column-reference. This reference contains the value 'M'. *Price* is also a column-reference, but its value is unknown until the previous SQL statement executes.

Column references are also used in *aggregate function*, such as in the following SQL statement:

```
Q2      SELECT MAX (Price)
          FROM Product
```

Price is a column-reference. When a column-reference is used in an *aggregate function*, the column of the SQL statement results in a calculated column and hence has *no column name* to be used for the specification.

Column Source. A column-reference whose values come directly from the database.

Column Projection. A column-reference whose values are calculated from the stored data values – so-called *calculated column*.

Valued Function. An argument that may represent a *function* reference. A valued function is a simple to complex operation.

Aggregate Function. A valued function that may represent a *value*. Aggregate functions are used to get information from several rows, process that information in some way (e.g. complex calculation) and deliver a single-row value. COUNT, AVG, MAX, MIN, and SUM are well-known SQL aggregate functions. For instance, in the following SQL statement the value is the number of rows with non-null values for the table *Product*:

```
Q1      SELECT COUNT (*)
          FROM Product
```

Scalar-Valued Function. A valued function that may represent a *value*. Scalar-valued functions are used to calculate a scalar (single) value. For instance, in the following SQL statement the *Price* of each *Product* is calculated as a percentage of its *Weight*:

Q_1 **SELECT** Size, Price, (Price/Weight) * 100
FROM [Product]
WHERE [Weight] > 0

The division (X / Y) and the multiplication (X * Y) are scalar-valued functions (so-called *expressions*).

Scalar-valued functions represent several kinds of operations (not only arithmetic operations), as such in the following SQL statement:

Q_2 **SELECT** *
FROM [Product]
WHERE [Price] > CEIL([Weight]) + 88.99

The addition (X + Y) and CEIL(Z) are scalar-valued functions.

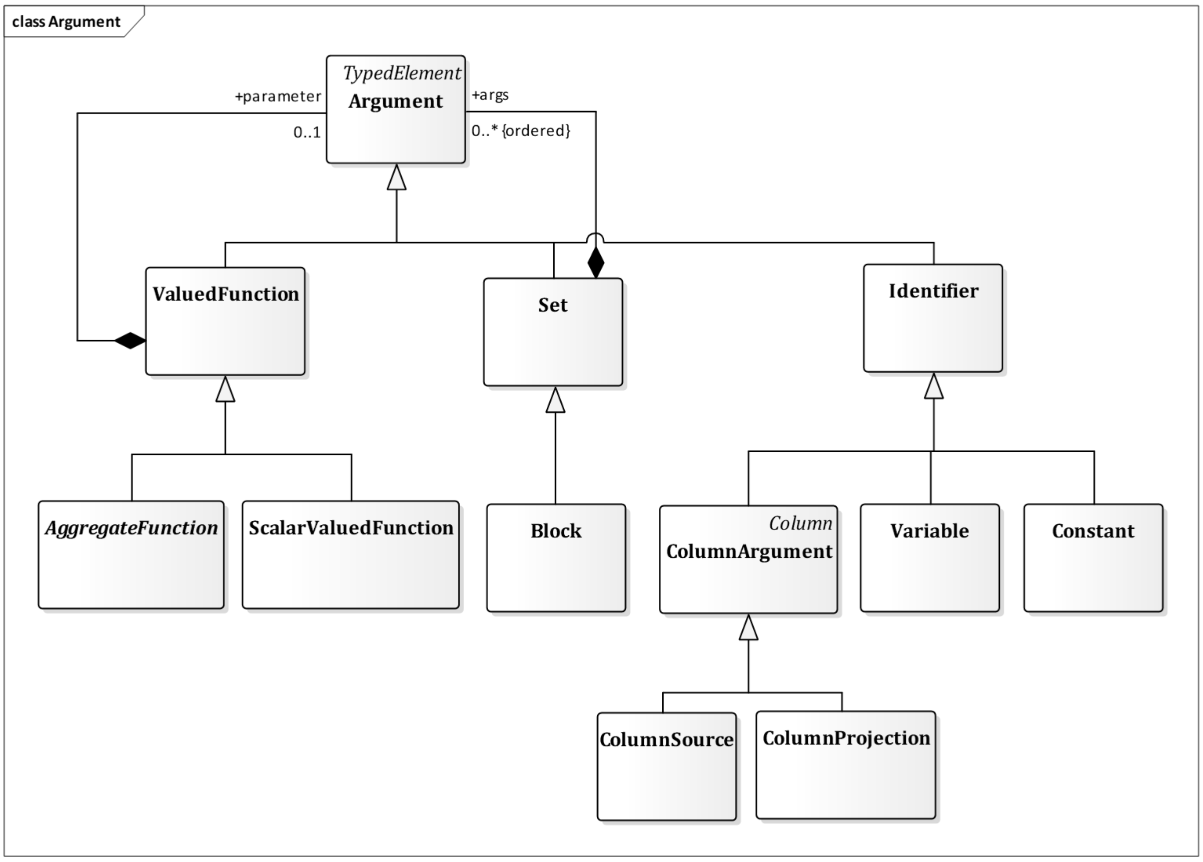


Figure 3.11 Abstract Syntax for Argument Package

Set. An argument that may represent a set of values. For instance, in the following SQL statement the search criterion tests whether a data value matches one of a set of target values:

```
Q1      SELECT *
          FROM [Customers]
          WHERE [Country] IN ('Germany', 'Portugal', 'France')
```

Block. A set whose values are enclosed in parentheses.

The Argument package provides a familiar, common set of concepts for describing the basic elements of the SQL language, such as constants, variables, expressions, functions, and so on. *Functions* can be parsed directly in a hierarchical data structure, which follows the *Composite Pattern* described in Appendix B.2.2. The tree-structure of *functions* is provided by the composite association between *Function* and *Argument* in which the result of one function may act as an actual parameter to its containing function. For example, the expression *(Price+5)*(Weight/10)* can be represented in a functional notation as follows:

Mult(**Add**(Price,5), **Div**(Weight,10))

3.2.2.4 Filter Criteria Package

The Filter Criteria package provides the basic concepts used to describe a rich set of search criteria (so-called *predicates* in the ANSI/ISO standard) that allow specifying a condition that is "true", "false", or "unknown" about a given row or group of data. Simple SQL search criteria can be combined to form more complex ones, which in turn may themselves be compound search criteria. To accomplish this, we follow the *Composite Pattern*. This design pattern was selected because of its ability to represent part-whole hierarchies by composing objects into tree structures. See Appendix B.2.2 for a detailed explanation of this solution. The core semantics in the Filter Criteria package are as follows (see Figure 3.12):

Filter Criteria. The foundation for all other modelling concepts in the *Filter Criteria* package. A *search condition* that specifies a condition that is "true", "false", or "unknown" about a given row or group of data. The reflexive composite association from *Filter Criteria* to itself allows *Filter Criteria* to own *Filter Criteria*. This association allows *Filter Criteria* to be organized in hierarchical, or tree-like, arrangements wherein the parent *Filter Criteria* is said to own its child *Filter Criteria*.

Binary Criteria. A *filter criterion* that combines two search criteria.

Unary Criteria. A *filter criterion* that negates a search criterion.

Comparison Criteria. A *filter criterion* that compares multiple *arguments*. Remember that an argument can be as simple as a column name or a constant, or they can be expressions and functions. Comparison criteria are often used to represent *basic predicate*, *quantified predicate*, *BETWEEN predicate*, *IN predicate*, and so forth. For instance, in the following SQL statements the ANY is used in conjunction with one of the six SQL comparison operators (=, <>, <, <=, >, >=) to compare a value with a set of values (note that a *Basic Comparison Criteria* is said to own its child *Quantifier Criteria*):

```
Q1    SELECT *
       FROM [Product]
       WHERE [ProductID] = ANY (SELECT [ProductID] FROM [ProductDetails])
```

In yet another example, in the following SQL statement the BETWEEN is used to check whether a data value lies between two specified values (note that BETWEEN is an *Extended Comparison Criteria*):

```
Q2    SELECT *
       FROM [Product]
       WHERE [ReleaseDate] BETWEEN '2011-06-1' AND '2018-05-1'
```

Block. A unary criterion whose *search criteria* are enclosed in parentheses. Parentheses are often used to build more complex search criteria. For instance, in the following SQL statement the parentheses are used to ensure portability, increase the readability of the statement, remove any possible ambiguity and make programmatic SQL statements easier to maintain:

```
Q1    SELECT *
       FROM [Product]
       WHERE ([Price] > 100.5)
        OR (Weight IS NOT NULL AND Weight > 10)
```

Parentheses are used to group the search criteria so that SQL process the search criteria in the expected order. Search criteria within parentheses are evaluated first. If the order is not specified by parentheses, NOT has the highest precedence, followed by AND and then OR.

It is worthy of note that when an SQL statement is collected by the CB Server for processing in a parsed statement, CB Server always adds parentheses even when they are not required (in the *search criteria*).

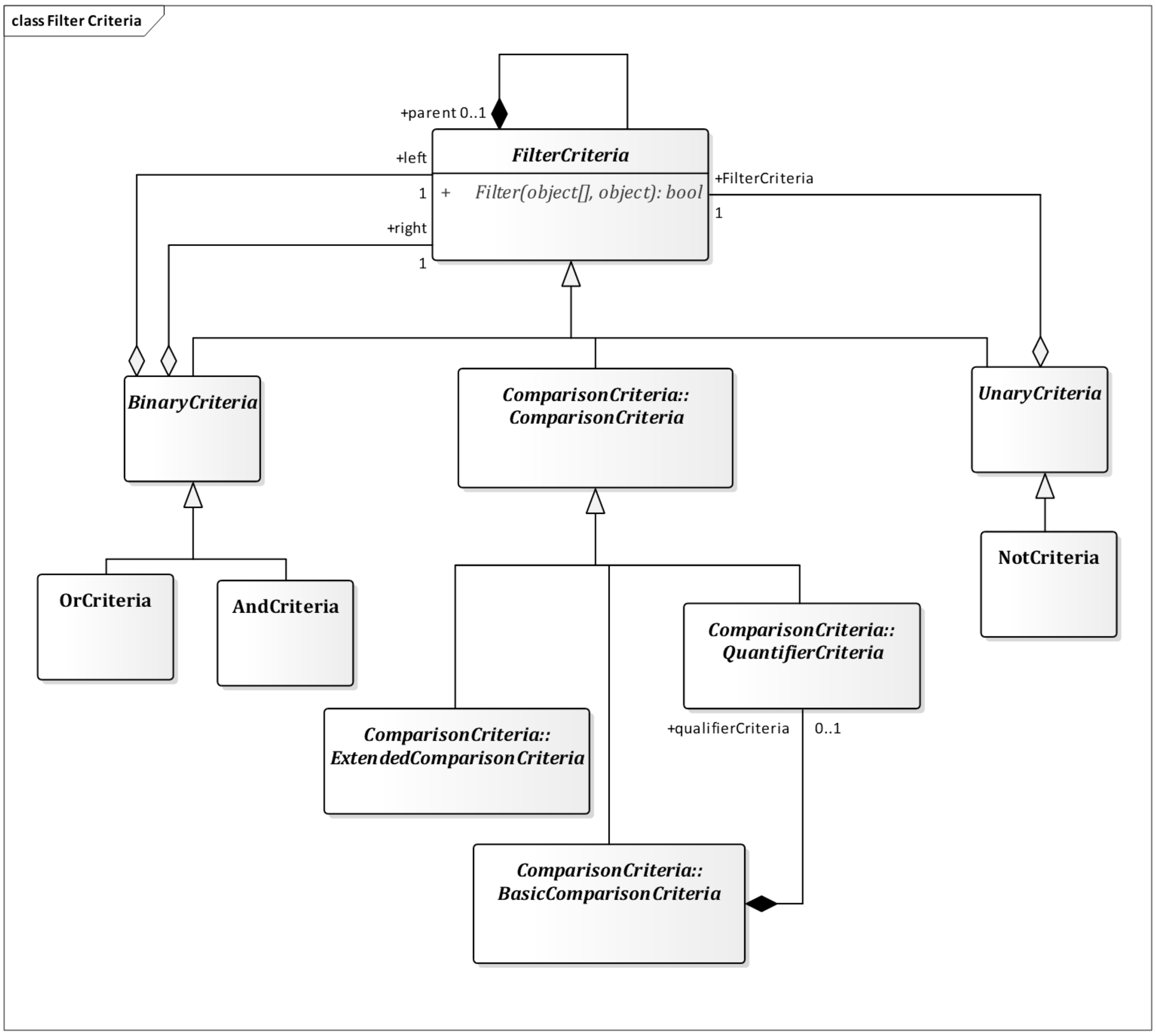


Figure 3.12 Abstract Syntax for Filter Criteria Package

The SQL search conditions – IN, EXISTS, IS NULL, LIKE, and so on – are widely understood and will not be further discussed here. See Appendix D.2 for a complete definition of the abstract syntax for Filter Criteria package.

To further illustrate how a search criterion can be parsed directly in a hierarchical data structure, a concrete example is given below. For example, for the following SQL statement, Figure 3.13 is given below. Remember that search criteria follow the WHERE clause.

Q$_1$ **SELECT** *
FROM [Product]
WHERE ([Name] **LIKE** '%st%')
OR (Weight >=10 **AND** Weight **IN** ('XL', 'M'))

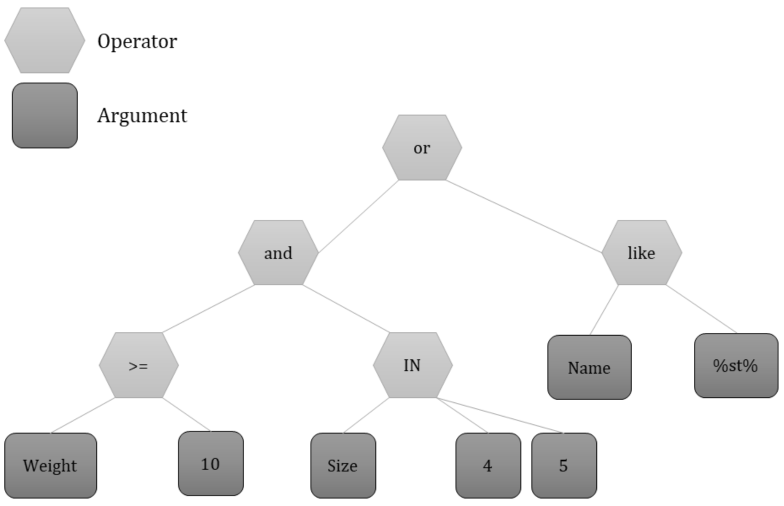


Figure 3.13 The tree-structure of search criterion

3.2.3 Designing the Connector

As previously stated, the real purpose of this dissertation is to develop an SDK for the creation of connectors to integrate with CB Server that can be used with minimal technical support by the developers. In fact, creating connectors can be challenging. As a result, a set of subroutine definitions, interfaces, classes, common structures, utilities, and helpers has to be defined to simplify the process of creating connectors. To accomplish this, we had to identify various major concepts from which to analyse, gather, and model the requirements.

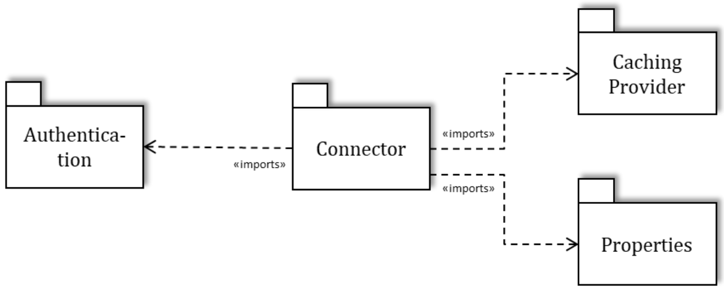


Figure 3.14 Package Diagram - Connector Packages

The Connector has a modular, or *packages*, architecture organized in four small, understandable *packages*: **Connector**, **Properties**, **Authentication**, and **Caching Provider** as shown in a package diagram, Figure 3.14. Each package covers different aspects of the creation of “new” connectors to integrate with CB Server. The Connector package, which depends upon *Plugin* package[4], defines all basic constructs required to specify the backbone of a functional connector. The Properties package provides the structure required to support the definition of connector properties which hold configuration information. The Authentication package contains several kinds of user authentication (OAuth 1.0, OAuth 2.0, and Basic Authentication). The Caching Provider package provides the basic concepts underlying in-memory cache that allow improving the performance and scalability by reducing the effort required to generate content.

We present here only a very small overview of the Connector whose complete definition can be found in Appendix D.3.

Figure 3.15 illustrates an overview of a functional connector as a realization of *Connector Interface*. A functional connector is modelled as a *relational database* – a collection of both data and metadata. As well, a functional connector has to provide the result of the complete computation of a *SQL data manipulation command*. Because Connect Bridge Server includes most of the major components of DBMS – that is, a data dictionary, a query language, and so forth – it is defined as a *simulated* database system (DBMS).

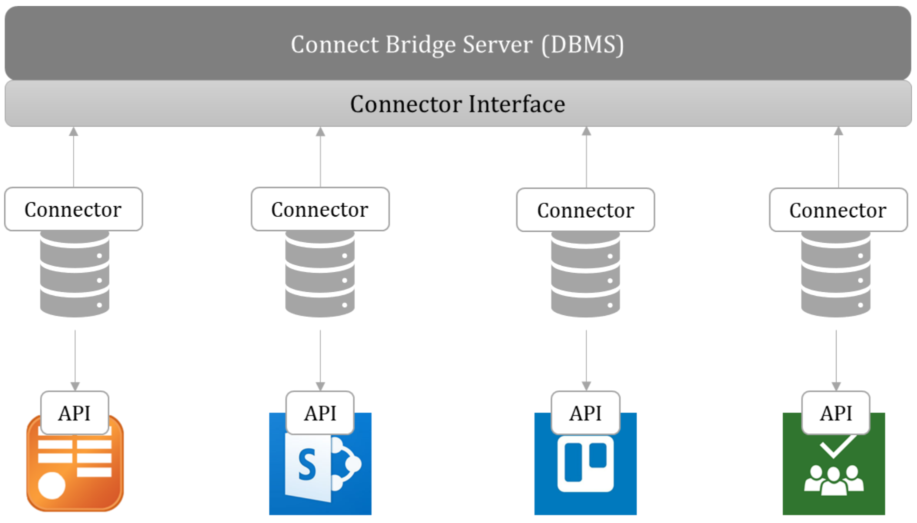


Figure 3.15 Overview of a functional connector

[4] The *Plugin package* is part of an existing standard library and provides the *Plugin* interface which contains properties and methods required to specify a “old” functional connector.

3.2.4 Designing the Exceptions

Error handling is one of the most neglected aspects of programming. An experience shared by almost every computer user is that programs fail with a cryptic, nonsensical, idiotic, useless, plainly wrong, or even missing error message. In fact, correct and complete reporting of error information can make the difference between spending 5 minutes or a week on finding the cause of an error. Thus, adequate treatment of errors become the key factor for the usability of a software product. Consequently, it is usually a good idea to produce custom exceptions so that computer users see (by the exception type) if the exception was raised by our code or by foreign code, and more importantly, since the exceptions are ours, we are able to change their definition to include additional information (e.g. error code, message, and so on). To accomplish this, we had to identify as much as possible information that might be relevant in an error situation.

The Exceptions, which satisfies FR-4, has a modular, or *packages*, architecture organized in one package called Exceptions. The Exceptions package provides a rich set of custom exceptions that is ideal for both CB Server and Connector.

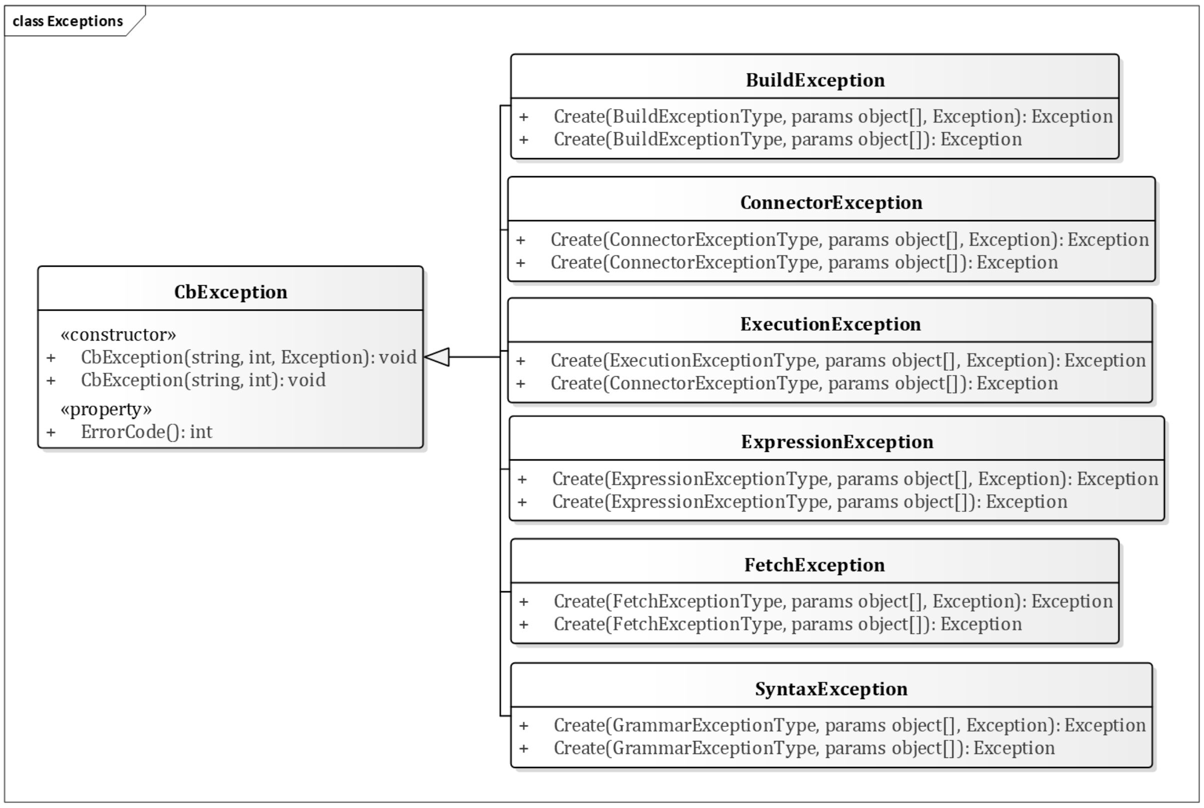

class Exceptions
CbException
«constructor»
+ CbException(string, int, Exception): void
+ CbException(string, int): void
«property»
+ ErrorCode(): int
BuildException
+ Create(BuildExceptionType, params object[], Exception): Exception
+ Create(BuildExceptionType, params object[]): Exception
ConnectorException
+ Create(ConnectorExceptionType, params object[], Exception): Exception
+ Create(ConnectorExceptionType, params object[]): Exception
ExecutionException
+ Create(ExecutionExceptionType, params object[], Exception): Exception
+ Create(ConnectorExceptionType, params object[]): Exception
ExpressionException
+ Create(ExpressionExceptionType, params object[], Exception): Exception
+ Create(ExpressionExceptionType, params object[]): Exception
FetchException
+ Create(FetchExceptionType, params object[], Exception): Exception
+ Create(FetchExceptionType, params object[]): Exception
SyntaxException
+ Create(GrammarExceptionType, params object[], Exception): Exception
+ Create(GrammarExceptionType, params object[]): Exception

Figure 3.16 Abstract Syntax for Exceptions Package

3.2.5 Summary

In this section, we presented a clear understanding of the architecture's key aspects, including the rationale and options considered, such as choosing design patterns and architectural styles. Through architecture decisions and design rationale, we are capable of documenting traceability between requirements and technical implementation.

In summary, the CBDK API is organized into 14 small, understandable separate packages as shown in Figure 3.17. Because understanding the main ideas of each package was our primary concern, the package descriptions had been simplified and focus on their central ideas.

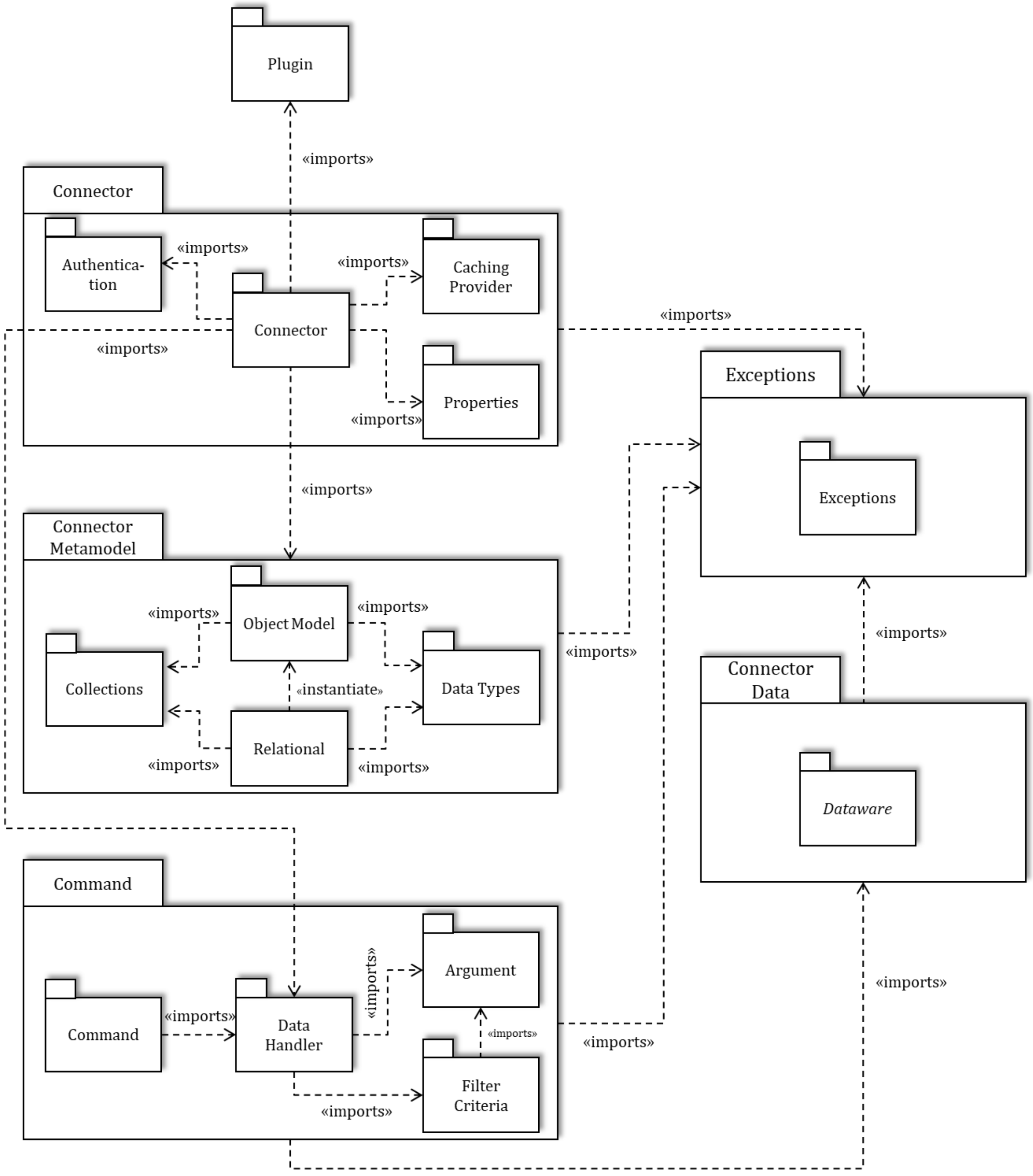


Figure 3.17 Package Diagram – Package structure of the API

In Figure 3.17, the content and interrelationships of 14 packages are shown. To promote further understandability, the packages are organized into five large, compound *packages*. Figure 3.17 contains 15 packages not 14 as previously mentioned; this is because *Plugin* package is not part of the CBDK API. Note that the *Dataware* package, which satisfies FR-5, FR-7 and FR-9, is not described in this section; this is because it is not required to understand other packages and because it was designed by a team member.

3.3 IMPLEMENTATION

The architectural design presented in section 3.2 was implemented. This section presents a brief overview of the implementation. Our solution was implemented using C# programming language with .NET Framework 4.6.1. and the Visual Studio environment. The source code is under a non-disclosure agreement to safeguard a company's proprietary information.

Figure 3.18 depicts our solution which: (a) supports an open and flexible architecture to make easier the development of new connectors to integrate with CB Server; and (b) provides some artefacts/tools which facilitate the development process.

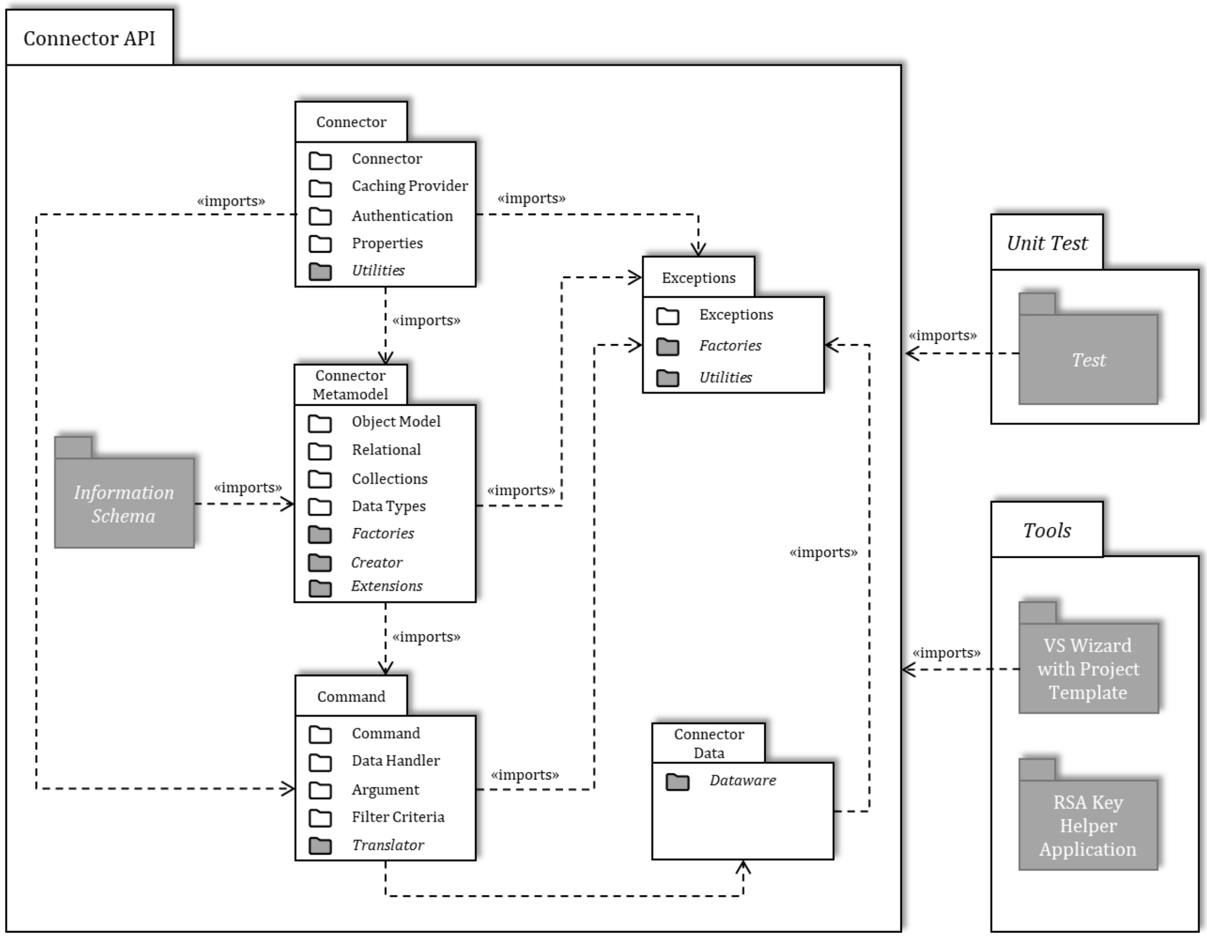


Figure 3.18 Package diagram of the full system (overview of the implementation)

Our solution consists of the following components (those visualized in grey were defined during the implementation):

– The **Information Schema** (from *Connector API*) which is a complete ANSI/ISO SQL standard definition for INFORMATION_SCHEMA. The information schema contains the *system tables*. The *system tables* hold the database's *metadata*. The definition of the information schema is based on SQL Server Schema Collections, which is detailed in Appendix G.

- The **Connector Metamodel** (from *Connector API*) which is a complete meta-level representation for modelling relational and non-relational data sources as a database. The **Connector Metamodel** realises the *connector metamodel* design presented in section 3.2.1. Furthermore (within *Factories* package), we follow the *Abstract Factory Pattern*. This design pattern was selected because of its ability to create families of related objects without specifying their concrete classes. See Appendix B.2.3 for a detailed explanation of this solution. Additionally (within *Creator* package), we follow the *Builder Pattern*. This design pattern was selected because of its ability to separate the construction of a *model* from its representation and because it allows instantiating a *model* by means of *eager loading*. See Appendix B.2.4 for a detailed explanation of this solution.

- The **Connector Data** (from *Connector API*) for creating *tables of data* and checking data consistency. A *table of data* represents a database *result set*, which is usually generated by executing a statement that queries the database. A *table of data* consists of a set of rows, each row having the same set of columns and at the intersection of every column and row is a specific data item called a *value*.

- The **Command** (from *Connector API*) realises the *command* design presented in section 3.2.2.

- The **Connector** (from *Connector API*) realises the *connector* design presented in section 3.2.3. In addition (within *Utilities* package), we provide helper methods for encrypting/decrypting, checking and parsing input data.

- The **Exceptions** (from *Connector API*) realises the *connector* design presented in section 3.2.4.

- The **Test** (from *Unit Test*) for validating the source code in the *Connector API*. The validation consists of validating individual units of source code through unit testing so that we could improve the software quality.

– The **Visual Studio (VS) Wizard with Project Template** tool (from *Tools*) tool for generating the skeleton of a functional connector exploiting the API. A Windows Form is displayed to gather user input (i.e. name, description, author, version, and connector properties) and to generate public/private *Rivest–Shamir–Adleman* (RSA) key pair.

– The **RSA Key Generator** tool (from *Tools*) for generating a public/private RSA key pair.

3.3.1 Connector API

The *Connector API* was implemented following the architectural presented in section 3.2. We started by developing the source code conventionally (i.e. through writing the code by hand). For this, the following implementation-level rules were applied:

Rule1. Primitive data types are directly mapped to corresponding primitive data types in the programming language.

Rule2. Enumeration is mapped to the corresponding enumeration in the programming language.

Rule3. For all classes in the source model, a C# *interface* was created, respecting the inheritance and relationships that are defined in the source model. In this sense, the interface hierarchy reflects the class hierarchy defined in the source model.

Rule4. The class attributes of the type *primitive type* are mapped to *properties* with same *definition* (i.e. *name* and *data type*) in the corresponding interface.

Rule5. The class attributes that are *sets* (usually, as result of *associations*), are mapped to *properties* in the corresponding interface where the type is mapped to collection types in the programming language or custom collection types, which take advantage of *Generics* to enforce type compliance at compile time.

In total, the architectural design involves 15 primitive data types, 11 enumerations, 117 classes and 47 associations.

For example, for the following fragment of the source model (considering the Named Element class), the C# interface is given below (by applying **Rule1**, **Rule3** and **Rule4**).

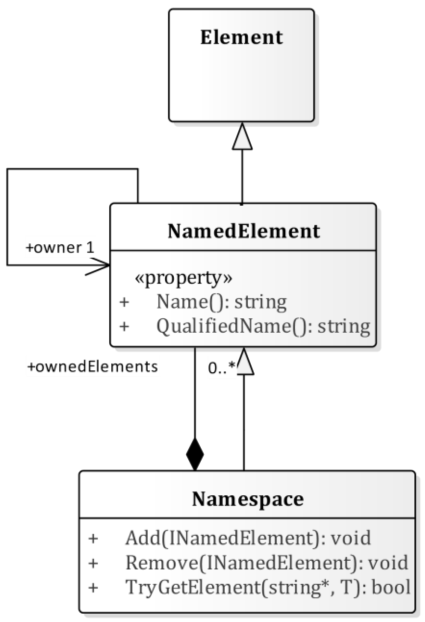

Element
NamedElement
«property»
+ Name(): string
+ QualifiedName(): string
+owner 1
+ownedElements
0..*
Namespace
+ Add(INamedElement): void
+ Remove(INamedElement): void
+ TryGetElement(string*, T): bool

Figure 3.19 Fragment of the source model – Named Element

```
public interface INamedElement : IElement
{
    string Name { get; }
    string QualifiedName { get; }
    INamedElement Owner { get; set; }
}
```

In yet another example, for the following fragment of the source model (considering the Schema class), the corresponding source code is given below (by applying **Rule5**).

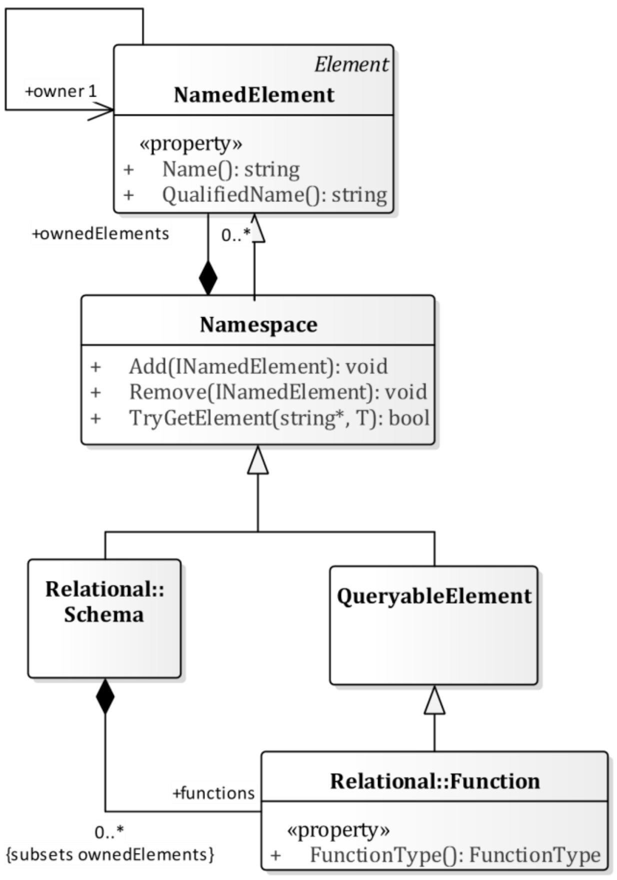


Figure 3.20 Fragment of the source model – Schema

```
internal class Schema : Namespace, ISchema
{
    public Schema(string name) : base(name)
    {
        ...
        Functions = new SubCollection<INamedElement, IFunction>(Elements);
    }
    public sealed override IElementCollection<INamedElement> Elements { get; }
    public IElementCollection<IFunction> Functions { get; }
    ...
}
```

In Figure 3.20, the notation *functions 0..** {**subsets** *ownedElements*} means that the set of elements for the property *functions* subsets the elements for the property *ownedElements*. But C# programming language has no definition for the notion of **subsets**. For this, our solution comprehends the *SubCollection* type (in *Collections Package*), which represents a subset of elements. For instance, the set of elements for the property *functions* will contain all the *functions* within the set of the elements of the property *ownedElements*.

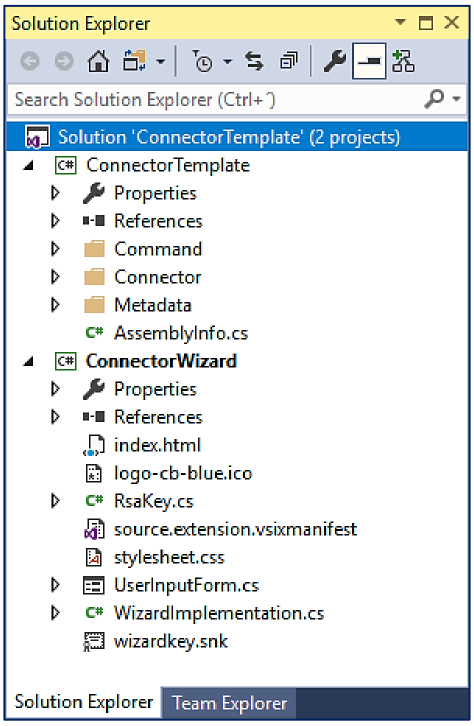
Note that the relationships previously mentioned are established in the class *constructor*.

The definition of the remaining interfaces and their realizations in classes is a long repetition of the procedures previously described.

3.3.2 Tools

The Connector API allows settling an open and flexible infrastructure to make easier the development of new connectors to integrate with CB Server. We briefly present a set of tools around the Connector API developed which facilitate the development process by exploiting the Connector API facilities.

The VS Wizard with Project Template, which is depicted in Appendix E.1, provides a starting point for developers to start creating new connectors to integrate with CB Server. The project template provides the files that are required for creating a new connector, includes standard assembly references, and sets default project properties and compiler options. The VS Wizard opens a Windows Form before the project is created. The form allows computer users to insert a custom parameter value that is added to the source code during project creation. To accomplish this, we implemented the `IWizard` interface. Figure 3.21 illustrates the project structure.



```
Solution 'ConnectorTemplate' (2 projects)
  ConnectorTemplate
    Properties
    References
    Command
    Connector
    Metadata
    AssemblyInfo.cs
  ConnectorWizard
    Properties
    References
    index.html
    logo-cb-blue.ico
    RsaKey.cs
    source.extension.vsixmanifest
    stylesheet.css
    UserInputForm.cs
    WizardImplementation.cs
    wizardkey.snk
```

Figure 3.21 The project structure

The RSA Key Generator, which is depicted in Appendix E.2, allows computer users to generate public/private RSA key pair to the console or to the XML file using `System.Security.Cryptography`. The console application includes a simple menu and requests the connector name.

3.4 SUMMARY

In this chapter, we presented a clear understanding of the overall process, including the rationale and decisions made in the course of this dissertation work. The overall process turned out to be *iterative*: that is, as the requirements changed or as requirements problems were encountered, the system design or implementation had to be reworked and retested.

All selected software requirements were satisfied. The system design was originally established from the software requirements, without considering the constraints imposed by CB Server. Consequently, there are some functionalities presented on the final product that are not supported by the CB Server.

Furthermore, on the way to help novice or expert developers, we produced a variety of documents: a step-by-step guide for creating a connector; a Sandcastle API specification; and Enterprise Architect (EA) class diagrams. Thus, NFR-1 and NFR-2 were fulfilled.

In short, the final product overcomes the problems previously identified, including several “new” concepts and constructs.

Chapter 4

Case Studies

This chapter presents two examples and case studies to analyse the correctness, feasibility, completeness, and accessibility of the API. While the first case study presents a connector for a relational data source (Microsoft SQL Server), the second case study presents a connector for a non-relational data source (OData). The outcome consists of a list of issues found in the API and a set of proposals for improvement.

4.1 Microsoft SQL Server

Microsoft SQL Server (MSSQL) is Microsoft's relational database management system which manages the database structure and controls access to the relational data stored in the database [50]. MSSQL supports ANSI/ISO SQL standard, explicitly T-SQL, which is a Microsoft's extension of SQL Language and expands on the SQL standard.

4.1.1 Overview

The real purpose of this case study was twofold: 1) to analyse the correctness, feasibility, completeness, and accessibility of the API for **relational data sources**, and 2) because CB Platform is SQL-compliant, to validate the functionality of the "new" CB Platform. The output of this work consists of the MSSQL connector, as well as a list of issues and proposals for improvement.

The development was done conventionally (that is, through writing the code by hand). The software was developed in C# with the .NET framework 4.6.1. Besides the .Net framework two important larger libraries were used:

– **MG.dll**. Connector library required for the creation of connectors to integrate with CB Server.

– **Microsoft.SqlServer.Scripting.dll**[5]. SQL Server Management Objects (SMO) library required to execute scripts.

The remaining code was written from scratch.

For developing the MSSQL connector, we set up a development environment that is made up of at least a SQL Server[6] instance installed on the local machine, including the AdventureWorks2016[7] sample database, and a CB Platform instance installed on the local machine, including a license for connector development.

4.1.2 Implementation Details

The development of MSSQL connector was simple and straightforward. This is because MSSQL connector is a SQL-based relational database for a relational data source. The overall process of developing MSSQL connector exploiting the API consists of the following steps:

1. Define the backbone of MSSQL connector.
2. Definition of the metamodel for MSSQL concepts.
    a. Get the *metadata* (i.e., tables, views, columns, parameters, etc.).
    b. Convert MSSQL data types to API data types.
3. Generate a SQL statement to execute against the MSSQL and retrieve data.

Step 1. resulted in the backbone of the MSSQL connector. We started by defining the name, description, author, version, and public/private key, and then we defined the properties. To minimize complexity only one property was defined:

– ***Connection String***, which specifies information about a MSSQL database and the means of connecting to it.

Step 2. lead to the instantiation of the connector's metamodel by means of *lazy loading*.

[5] https://www.nuget.org/packages/Microsoft.SqlServer.Scripting/
[6] https://go.microsoft.com/fwlink/?linkid=875802
[7] https://www.microsoft.com/en-us/download/details.aspx?id=49502

Step 3. We started by defining a *data structure* containing the parts from which an SQL statement is constructed. The *data structure* is provided to all handlers. Each handler fills the *data structure* according to its information. We then generate the SQL statement according to the *data structure* to execute against MSSQL and retrieve data. Figure 4.1 illustrates the basic workflow here described.

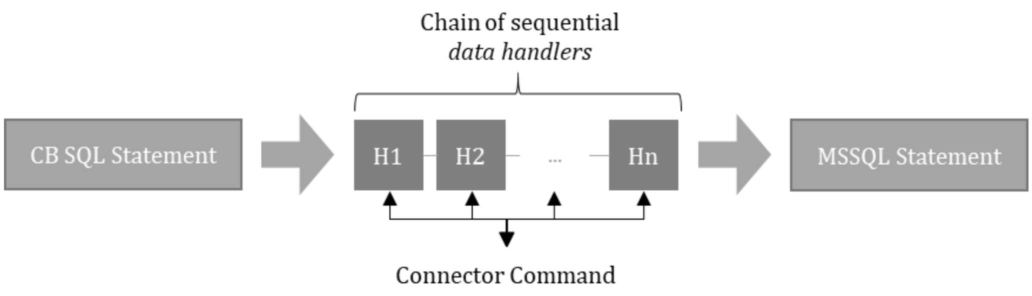
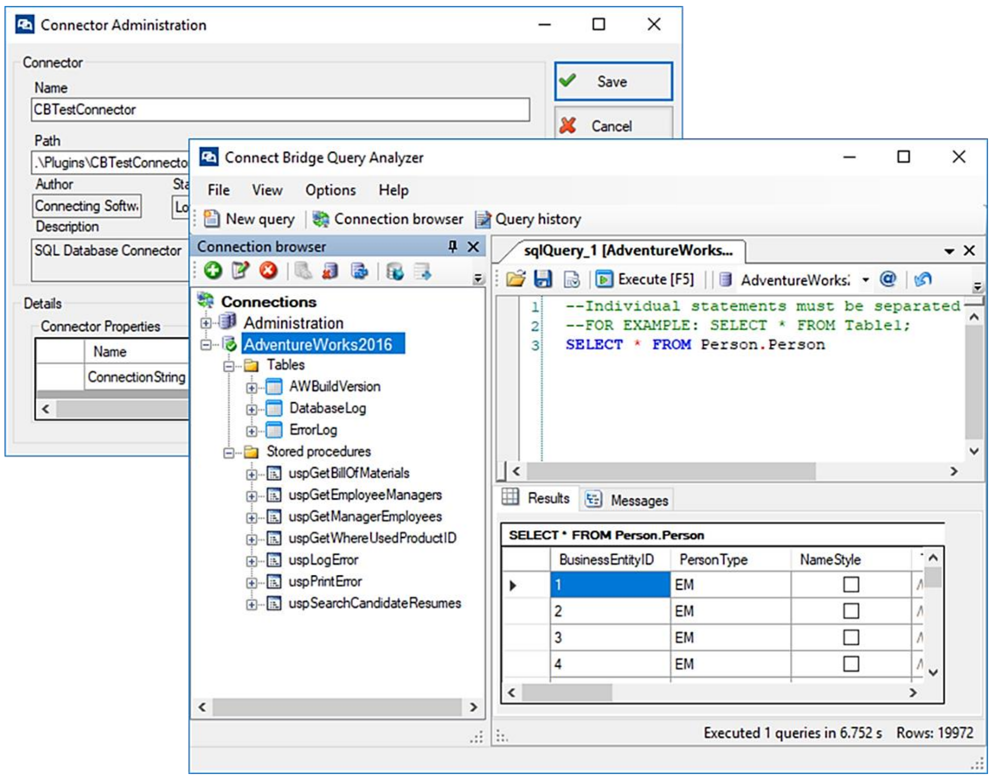


Figure 4.1 Basic Workflow in MSSQL connector

4.1.3 Analysis and Discussion

Figure 4.2 depicts the final result of the MSSQL connector.



Connector

Name: CBTestConnector

Path: .\Plugins\CBTestConnecto

Author: Connecting Softw

Description: SQL Database Connector

Details

Connector Properties

Name
ConnectionString

Connection browser

- Connections
  - Administration
  - AdventureWorks2016
    - Tables
      - AWBuildVersion
      - DatabaseLog
      - ErrorLog
    - Stored procedures
      - uspGetBillOfMaterials
      - uspGetEmployeeManagers
      - uspGetManagerEmployees
      - uspGetWhereUsedProductID
      - uspLogError
      - uspPrintError
      - uspSearchCandidateResumes

sqlQuery_1 [AdventureWorks...

```
--Individual statements must be separated
--FOR EXAMPLE: SELECT * FROM Table1;
SELECT * FROM Person.Person
```

SELECT * FROM Person.Person

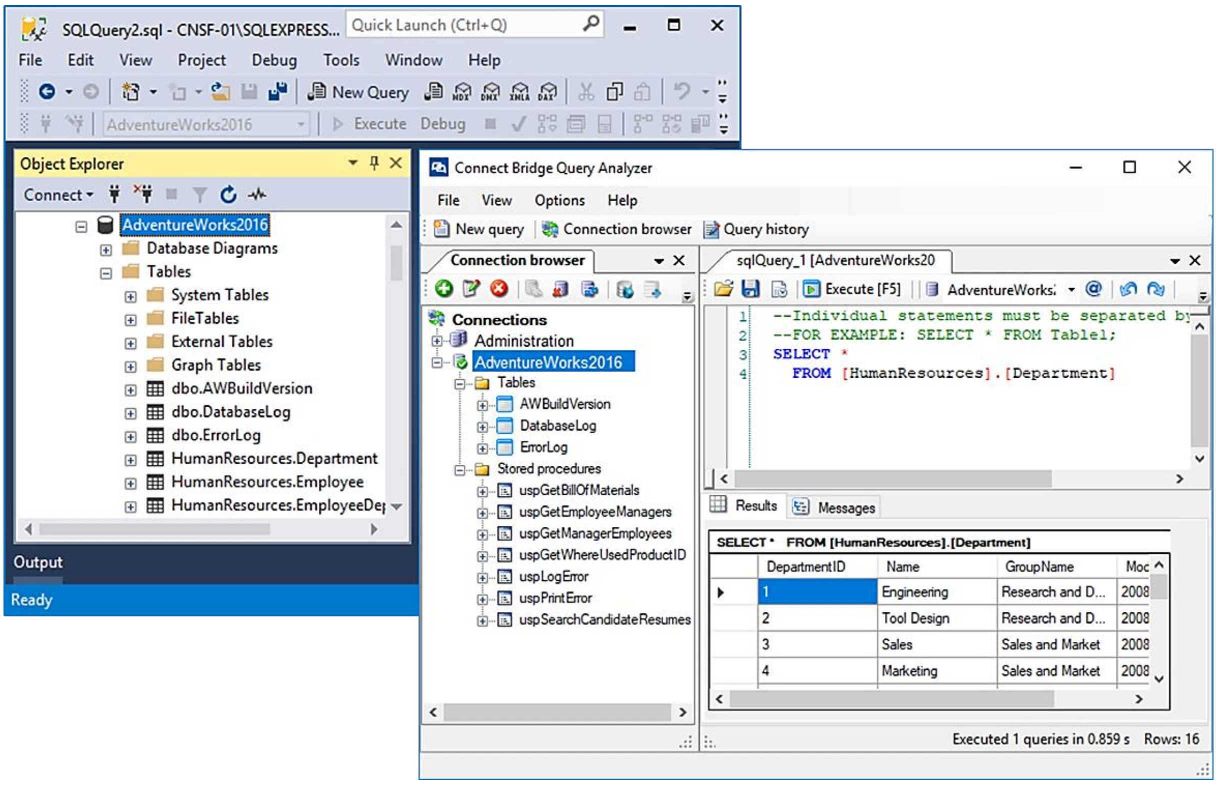
BusinessEntityID	PersonType	NameStyle
1	EM	☐
2	EM	☐
3	EM	☐
4	EM	☐

Executed 1 queries in 6.752 s Rows: 19972

Figure 4.2 A screenshot of the MSSQL connector

As a result of the development of MSSQL connector, the following issues and proposals for enhancement were encountered:

1. The system can only show Tables (of the default schema) and Stored Procedures in the Connection browser as shown in Figure 4.3. This situation is problematic because users will not possess means of knowing about all other data elements.



```
--Individual statements must be separated by
--FOR EXAMPLE: SELECT * FROM Table1;
SELECT *
  FROM [HumanResources].[Department]
```

SELECT * FROM [HumanResources].[Department]

DepartmentID	Name	GroupName	Moc
1	Engineering	Research and D...	2008
2	Tool Design	Research and D...	2008
3	Sales	Sales and Market	2008
4	Marketing	Sales and Market	2008

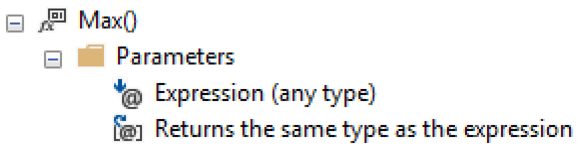
Figure 4.3 Comparison between MSSQL and MSSQL connector metadata

2. The system still needs to improve compatibility with T-SQL (on the server side). Several SQL search conditions, such as EXISTS, ALL, and ANY, as well as subqueries, are not supported.

3. The system needs to provide the ability to invoke a table-valued function in a FROM clause as shown in the following SQL statement:

   SELECT * **FROM** [dbo].[ufnGetContactInformation] (1)

4. The API needs to provide more kinds of data types, such as *Any Type* and *Parametric Type*. For instance, in the following definition of the *MAX* function:

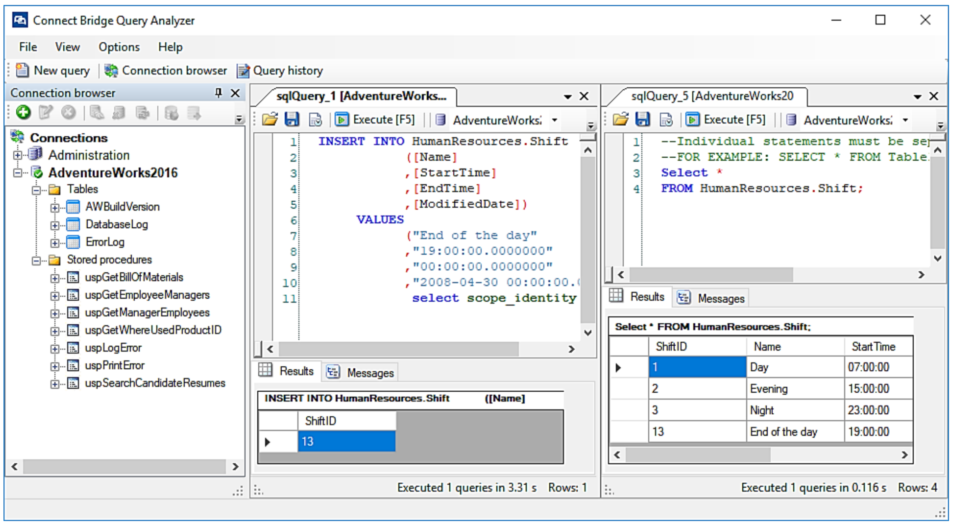


Expression is an input parameter of the type *any type*, while the *return* parameter is of the type *parametric type*; this is because the return parameter depends upon the type of the *Expression*.

5. The API fails to convert some MSSQL data types to API data types, and vice-versa.
6. The API should include some system functions (e.g. SCOPE_IDENTITY).

Our findings suggest that: (a) most of the fundamental issues from the previous CB Platform still remain unresolved; and (b) the API requires better testing so that errors do not pass unnoticed. The **issues** labelled **2** and **3** confirm that most of the functionality supported by MSSQL is not supported by the "new" CB Platform, while **issue** labelled **5** confirms that the API generates unexpected errors.

With regards to our findings, we had to make some improvements in both API and CB Server. First, considering **enhancement** labelled **4**, we introduced the parametric data type along with the primitive data type *Any Type*. Second, considering **issue** labelled **5**, we made corrections in the *Convert* method for several primitive data types (e.g. string, byte array), and hence the unit tests were improved. Third, considering **enhancement** labelled **6**, we introduced the SCOPE_IDENTITY as system function in the *Information Schema* (see Figure 4.4), which satisfies FR-10. Fourth, considering **issues** labelled **2** and **3**, a team member made some modifications in the CB Server to improve compatibility with T-SQL since the CB Server has limited access. Finally, because a "new" web-based interface was being developed to replace CB Query Analyser and CB Administration Tool, the **issue** labelled **1** was left unresolved.



```
INSERT INTO HumanResources.Shift
           ([Name]
           ,[StartTime]
           ,[EndTime]
           ,[ModifiedDate])
     VALUES
           ("End of the day"
           ,"19:00:00.0000000"
           ,"00:00:00.0000000"
           ,"2008-04-30 00:00:00.(
            select scope_identity
```

INSERT INTO HumanResources.Shift ([Name]

	ShiftID
▶	13

Executed 1 queries in 3.31 s Rows: 1

```
--Individual statements must be se
--FOR EXAMPLE: SELECT * FROM Table
Select *
FROM HumanResources.Shift;
```

Select * FROM HumanResources.Shift;

	ShiftID	Name	StartTime
▶	1	Day	07:00:00
	2	Evening	15:00:00
	3	Night	23:00:00
	13	End of the day	19:00:00

Executed 1 queries in 0.116 s Rows: 4

Figure 4.4 System function SCOPE_IDENTITY

4.2 Open Data Protocol

Open Data Protocol (OData) is an ISO/IEC approved, OASIS standard for building and consuming RESTful APIs [51]. The protocol provides a uniform way to describe both the *metadata* (i.e., a machine-readable description of the data model exposed by a data provider) and the data (i.e. sets of data entities and the relationships between them). Additionally, the protocol supports the editing and querying of data according to the metadata. An *OData metadata document* is a representation of a service's data model exposed for client consumption. The data exposed by an OData service is described in terms of the *Entity Data Model* (EDM). The core concepts in the EDM are *entities*, *entity sets*, *singletons, relationships*, and *operations*.

Entities are instances of entity types (e.g. *Person*, *Photo*, etc.). *Entity types* are nominal structured types with a key. Entity types support inheritance from other entity types. Entities consist of named properties and may include relationships with other entities.

Entity sets are named collections of entities (e.g. *People* is an entity set containing *Person* entities).

Singletons are named single entities (e.g. *Me* is a singleton containing *Person* entity).

Relationships have a name and are used to navigate from an entity to related entities. Relationships are represented in entity types as *navigation properties*.

Relationships may be addressed directly through a navigation link representing the relationship itself. Each relationship has a cardinality. Relationships may include *binding information* to bind the entity to existing entities.

Operations allow the execution of custom logic on parts of a data model. *Actions* allow side effects (e.g. an action may be used to extend CRUD operations or to invoke custom operations). *Functions* do not allow side effects. Actions and functions may include parameters and return type.

4.2.1 Overview

The real purpose of this case study was to analyse the correctness, feasibility, completeness, and accessibility of the API for **non-relational data sources**. The output of this work consists of the OData connector, as well as a list of issues and proposals for improvement.

The development was done conventionally (that is, through writing the code by hand). The software was developed in C# with the .NET framework 4.6.1. Besides the .Net framework three important larger libraries were used:

– **MG.dll**. Connector library required for the creation of connectors to integrate with CB Server.

– **Microsoft.OData.Edm.dll**[8]. Microsoft OData library required to represent, construct, parse, serialize and validate Entity Data Models.

– **Newtonsoft.Json.dll**[9]. JSON library required to serialize and deserialize JSON.

The remaining code was written from scratch.

For developing the OData connector, we set up a development environment that is made up of at least a CB Platform instance installed on the local machine, including a license for

[8] https://www.nuget.org/packages/Microsoft.OData.Edm/
[9] https://www.nuget.org/packages/Newtonsoft.Json/

connector development, a Postman[10] instance installed on the local machine, and several OData APIs (e.g. Trippin[11], Microsoft Graph[12], and Microsoft Dynamics CRM).

4.2.2 Implementation Details

The development of OData connector was more complicated. This is because OData connector is a SQL-based relational database for a non-relational data source (e.g. relationships, complex types, XML parser, JSON parser, etc.). The overall process of developing OData connector exploiting the API consisted of the following steps:

1. Define the backbone of OData connector.
2. Definition of the metamodel for OData concepts.
    a. Get the *metadata* (i.e., *entities*, *entity sets*, *operations*, etc.).
    b. Convert OData data types to API data types.
3. Generate an URL to execute against the OData service and retrieve data.
4. Deserialize JSON data into *Result Set*.

Step 1. resulted in the backbone of the OData connector. We started by defining the name, description, author, version, and public/private key, and then we defined the properties. To control how the OData connector models OData APIs as a database, the following properties were defined:

– ***Service Root URL***, which identifies the root URL of an OData service.
– ***Use Simple Names***, which specifies restrictions on OData's Name attribute.
– ***Supports Expand***, which specifies restrictions on OData's *$expand* query option.
– ***Maximum Expansion Depth***, which specifies restrictions on OData's *$expand* query option.

[10] https://www.getpostman.com/
[11] https://services.odata.org/V4/TripPinServiceRW/$metadata
[12] https://graph.microsoft.com/v1.0/$metadata

To authenticate to an OData service, several connection properties were defined to support various kinds of user authentication: OAuth 1.0, OAuth 2.0, Basic Authentication, and No Authentication.

Step 2. lead to the instantiation of the connector's metamodel. We begin by deserializing the *OData metadata document* into an EDM object, and then we automatically derived from the EDM object the following data elements:

- ***Schema***, which describes the *entity model* exposed by an OData service.
- ***Table***, which describes *entity sets*, *singletons* and *relationships*. To accomplish this, the following implementation-level rules were applied:

  Rule1. An entity set or singleton results in a table with the same name. All named properties are represented as columns (no additional information).

  Rule2. A *relationship* (i.e., *navigation property*) with *binding information* results in a table named in the format *ParentTable_Target*. Only the keys are represented as columns (no additional information).

  Rule3. A *relationship* (i.e., *navigation property*) without *binding information* results in a table named in the format *_ParentTable_NavigationProperty*. All named properties of the related entity and the key of the *parent table* are represented as columns.

- ***Stored Procedure***, which describes unbounded or bounded actions. To accomplish this, the following implementation-level rules were applied:

  Rule1. An *unbounded action* results in a procedure with the same name. All *parameters* are represented as input parameters and the *return type* as a return parameter.

  Rule2. A *bounded action* results in a procedure named in the format *ActionNameForBindingParameter*. All *parameters* (except the *binding parameter*) and key of the *binding parameter* are represented as input parameters and the *return type* as a return parameter.

- ***Function***, which describes unbounded or bounded functions.

– ***Data Type***, which describes EDM primitive types, Enumeration types, Collection types, Complex types, JSON types or Table types.

See Appendix F for a detailed explanation of this solution.

Because we found problems related to performance and memory consumption for Microsoft Graph and Microsoft Dynamics CRM, we decided to instantiate the connector's metamodel by means of *eager* and *lazy loading*.

Step 3 and **step 4**. We started by defining a *data structure* containing the parts from which an OData URL is constructed (i.e., *$filter*, *$expand, $top, $select*, etc.). The *data structure* is provided to all handlers. Each handler fills the *data structure* according to its information. We then generate the OData URL according to the *data structure* to execute against the OData service and retrieve data (in JSON format). Figure 4.5 illustrates the basic workflow here described. The retrieved data is deserialized into *Result Set* using *JsonTextReader*.

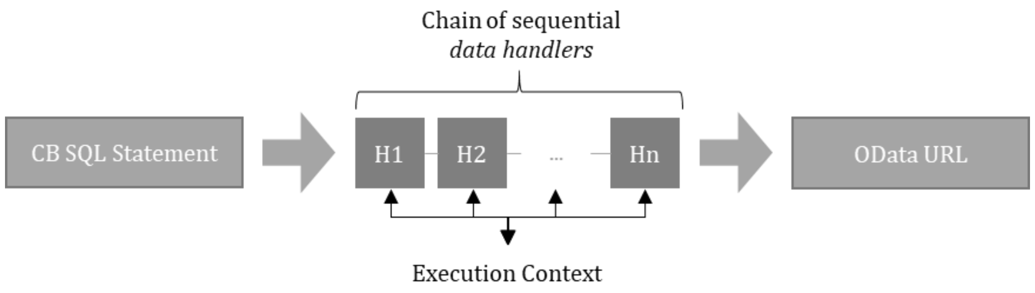
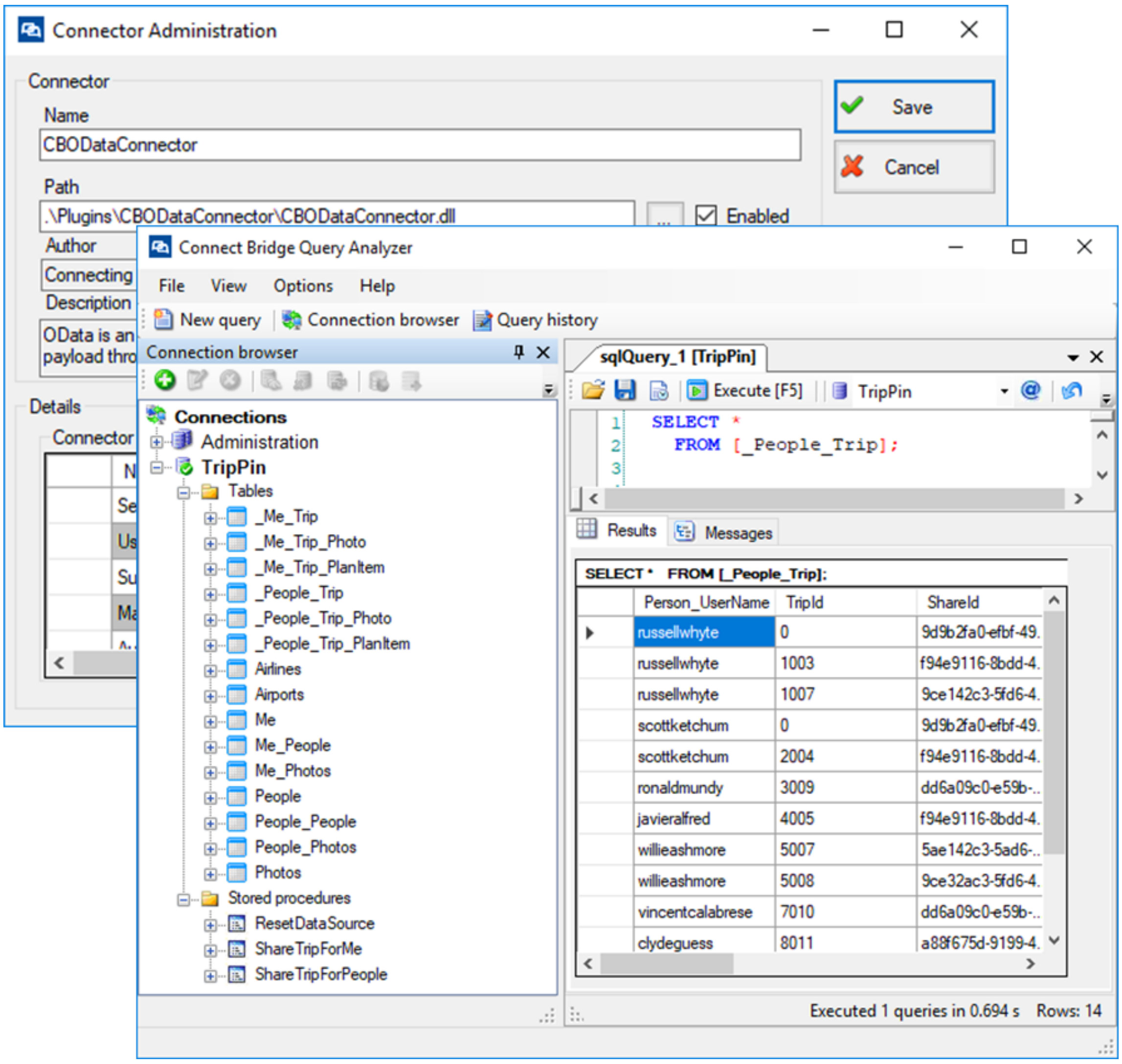


Figure 4.5 Basic Workflow in OData connector

4.2.3 Analysis and Discussion

Figure 4.6 depicts the final result of the OData connector.



```
SELECT *
  FROM [_People_Trip];
```

SELECT * FROM [_People_Trip];

Person_UserName	TripId	ShareId
russellwhyte	0	9d9b2fa0-efbf-49.
russellwhyte	1003	f94e9116-8bdd-4.
russellwhyte	1007	9ce142c3-5fd6-4.
scottketchum	0	9d9b2fa0-efbf-49.
scottketchum	2004	f94e9116-8bdd-4.
ronaldmundy	3009	dd6a09c0-e59b-..
javieralfred	4005	f94e9116-8bdd-4.
willieashmore	5007	5ae142c3-5ad6-..
willieashmore	5008	9ce32ac3-5fd6-4.
vincentcalabrese	7010	dd6a09c0-e59b-..
clydeguess	8011	a88f675d-9199-4.

Executed 1 queries in 0.694 s Rows: 14

Figure 4.6 A screenshot of the OData connector

As a result of the development of MSSQL connector, the following issues and proposals for enhancement were encountered:

1. The system still needs to improve compatibility with T-SQL (on the server side).
2. The API should include facilities for XML and JSON deserialization.
3. The API should provide facilities for supporting several types of user authentication: OAuth 1.0, OAuth 2.0, and Basic Authentication.
4. The system needs to provide the ability to store *connection properties* on server side (e.g. access token, refresh token, expiration date, etc.).
5. The API needs to provide facilities for generating documentation.

6. The API needs to provide more kinds of data types, such as *JSON Type* and *Table Type*. For instance, in the following definition of the *assignLicence* procedure:

   ```
   <Action Name="assignLicense" IsBound="true">
     <Parameter Name="bindingParameter" Type="user" Nullable="false"/>
     <Parameter Name="addLicenses" Type="Collection(assignedLicense)" Nullable="false"/>
     <Parameter Name="removeLicenses" Type="Collection(Edm.Guid)" Nullable="false"/>
     <ReturnType Type="microsoft.graph.user"/>
   </Action>
   ```

   addLicenses is an input parameter of the type *collection* of *complex type*, while the *return* parameter is of the type *entity type*. A *collection* of *complex type* may be represented as *JSON Type*, which can include a set of methods (e.g. *json_value*, *json_query*, *json_exists*, *is_json*, *json_contains*, etc.). An *entity type* may be represented as *Table Type*.

Our findings suggest that: (a) some of the fundamental issues from the previous case study still remain unresolved; and (b) the API fails to provide facilities for various activities and tasks in the software development process.

With regards to our findings, we had to make some improvements in both API and CB Server. First, considering **enhancement** labelled **3**, we introduced facilities for supporting several types of user authentication (OAuth 1.0, OAuth 2.0, and Basic Authentication). Additionally, we made available *OAuthConnector* class for OAuth 1.0 and 2.0 to generate login URL and to define call-back URL which will be called by third-party services on successful authentication. Second, considering **enhancement** labelled **6**, we introduced the *JSON Type* along with the *Table Type* as *Complex Types*; however, both complex types do not have defined methods. Third, considering **issue** labelled **1** and **enhancement** labelled **4**, a team member made some modifications in the CB Server to improve compatibility with T-SQL and to support saving *connection properties* on server side since the CB Server has limited access. Finally, because **enhancements** labelled **2** and **5** were placed in a low priority, they were left unresolved.

Chapter 5

Evaluation

After several years of progress in software engineering, there is still poor API design. Henning stated in [52] that the major reason is that "*it is very easy to create a bad API and rather difficult to create a good one*". In fact, we all recognize a good API when we get to use one; a good API is pleasant and intuitive to use, is well documented, can be found and memorized easily, and provides *abstractions* to reduce further complexity. However, we do not yet possess metrics to measure the quality of an API. API quality encompasses several aspects, such as correctness, completeness, usability, performance and so forth. Our focus here is only on API usability. The best well-known definition of usability is perhaps the one in [53] which defines usability as "*the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use*". However, usability is not simply "*better*" or "*worse*"; so, how do we analyse the usability of an API?

Several existing approaches of human-computer interaction (HCI) can be applied to evaluate the usability of an API. One of the well-known frameworks is GOMS. John and Kieras stated in [54] that GOMS "*is useful to analyse the knowledge of how to do a task in terms of **g**oals, **o**perators, **m**ethods, and **s**election rules*". Briefly, *goals* represent the user's goals; *operators* are the basic actions that the user performs to accomplish its goals; *methods* are sequences of sub-goals and operators; and *selection rules* are the reason why users choose one method to use in a certain context. The weakness of GOMS framework is the complexity of predicting the detailed time of every single task, as well as the lack of support for notational issues. Green and Petre [55] proposed a different approach, named *cognitive dimensions framework*, as a "*broad-brush evaluation technique for interactive devices and for non-interactive notations*". Unlike many other approaches, the cognitive dimensions framework focuses on assessing the overall usability of the design. Other approaches emphases on software quality metrics. A software quality metric is defined as a "*function whose inputs are software data and whose output is a single numerical value* *that can be interpreted as the degree to which software possesses a given attribute that affects its quality*” [56].

In fact, the existing approaches related to API usability are heterogeneous in terms of goals, scope, and audience, and the use of software quality metrics to measure API usability is focused only on quantifiable characteristics rather than usability aspects [57]. To overcome this problem, E. Mosqueira-Rey et al. in “*A systematic approach to API usability: Taxonomy-derived criteria and a case study*” proposed an approach to API Usability based on a comprehensive model of usability and context-of-use.

This chapter introduces the methodological approach applied to evaluate the usability of the API. Multiple case study works as the main evaluation method, combining several HCI methods. The case study consists of three evaluation phases – a hands-on workshop, a heuristic evaluation and subjective analysis. Moreover, results collected from all phases of the evaluation are presented and further discussed. In addition, the context of the study is discussed to help withdraw conclusion from the results.

5.1 METHODOLOGY

Our methodological approach for evaluating API usability combines several usability evaluation methods – observations, heuristic evaluation, questionnaires and interviews. Each method has different strong and weak points. It combines several HCI methods to take full advantage of the strong points of each method. This combination provides an opportunity to get a broad view of the usability of the API to be evaluated. These may be not only problems and flaws in the source code, but also conceptual and runtime problems as well as findings related to user experience (UX). The methodology is the first step in the direction of defining a structured process to accomplish these goals.

5.1.1 Roles

In our methodological approach we identify the following roles:

– ***Expert Developers***, Computer science engineers knowledgeable in the application domain with experience in the programming language of the evaluated API. They should be fully familiar with the previous API[13]. The expert developers are used in the hands-on workshop (1st phase of the evaluation process), heuristic evaluation (2nd phase) and subjective analysis (3rd phase).

- ***Novice Developers***, Computer science engineers with experience in software development. They should be familiar with the programming language of the evaluated API, but they should not be familiar with the application domain. The novice developers participate in the hands-on workshop (1st phase of the evaluation process), heuristic evaluation (2nd phase) and subjective analysis (3rd phase).
- ***Evaluators***, Computer science engineers who collect and analyse findings in each phase of the evaluation. They actively participate in the hands-on workshop and they also conduct the interviews.

5.1.2 Process Overview

Figure 5.1 illustrates the process of our evaluation approach addressing the usability of the API. The process consists of five phases: (1) planning; (2) workshop with novice and expert developers, where the questions, problems, and potentials are assessed; (3) heuristic evaluation; (4) subjective analysis, where questions and subjective opinions are evaluated; and (5) the final analysis. The three different evaluation methods are conducted independently, and the subjective analysis is done right after the workshop to avoid the repetition of an introduction part to the API.

[13] MG Framework

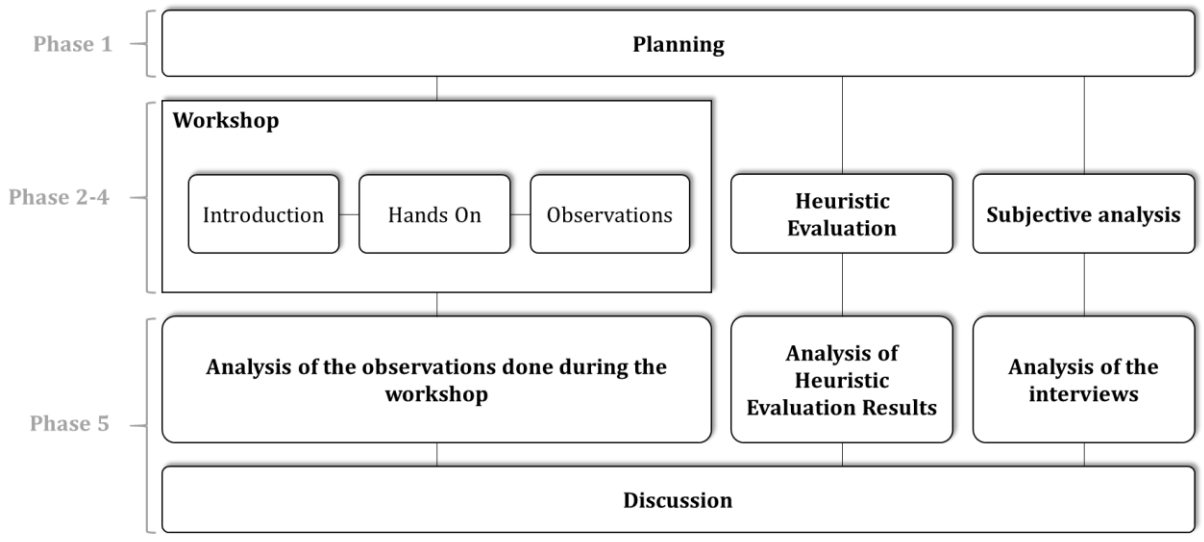


Figure 5.1 Methodological Approach

Phase 1: Planning. In the *planning* phase, the evaluators define a few novice and expert developers to be involved in the evaluation process. Evaluators also decide the objectives and identify relevant parts of the API to be evaluated, as well as details of the next phases.

Phase 2: The Hands-on Workshop takes approximately 3-5 days and consists of two parts – introduction, and hands-on project. In the introduction, the application domain of the API is presented as well as the API itself. After that novice and expert developers are asked to implement a *connector* defined by the evaluators using the API. They are also asked to note problems they struggle with. Evaluators are also present during the workshop. The role is to observe, note usability problems and provide help when someone is struggling with a problem for too long. A demographic questionnaire and an informed consent have to be signed by them at the beginning of the workshop.

Phase 3: The heuristic evaluation is conducted after the *hands-on workshop*. Novice and expert developers receive a list of heuristics used to find, analyse, and categorize problems identified with the API – API Usability questionnaire. The outcome of the heuristic evaluation consists of a list of strong and weak points of the usability of the API and a set of proposals for improvement. After the evaluation, the findings are collected and analysed.

Phase 4: Subjective Analysis is conducted right after the *heuristic evaluation* to try to clarify any questions that developers could have had when filling the API Usability questionnaire, and to obtain subjective opinions regarding their individual experience of the workshop. The interviews are recorded to allow future analysis.

Phase 5: Discussion. After collecting all materials, evaluators analyse notes, logs, and questionnaires. They identify and summarize usability problems. The output of the methodology is a set of recommendations for improving the evaluated API.

5.2 CASE STUDY

Case studies are one of the several research approaches within scientific research. Other research approaches include archive research, survey research and experiments [58]. Amongst existing research approaches, case studies are considered the preferred approach for answering "*how?*" and "*why?*" questions, although they are less useful for investigating "*what?*" and "*how much?*" questions.

Yin in [58] defines case study as follows:

> *"an empirical inquiry that investigates a contemporary phenomenon within its* ***real-life context****; when the boundaries between phenomenon and context are not clearly evident; and in which* ***multiple sources of evidence are used****"*

This definition helps us to understand clearly how case studies differ from other research approaches. Experiments intentionally divide a phenomenon from its context and often the context is "*controlled*" by the laboratory environment; while case studies focus on the phenomenon in its real-life context. Survey research can deal with phenomenon and context, but their capability to do so is limited. Archive research can deal with phenomenon and context but focus on non-contemporary phenomena.

Within case study research it is possible to differentiate between single and multiple case studies [58]. While a single case study relies on one single case (as the name suggests), multiple case studies rely on the investigation of several cases. Obviously, investigating an issue in more than one context/case is typically better than basing results on just one single case. However, there are occasions when a single case study is quite sufficient (e.g. for investigating extreme or unique cases).

Multiple case studies are considered more appealing as their results are more robust. However, conducting multiple case studies can require extensive resources and time, as well as considerable thought on which cases to select. Yin in [58] advises that "*multiple* *cases*" should be regarded as "*multiple experiments*" and not "*multiple respondents in a survey*", and so *replication logic* and not *sampling logic* should be used for the selection of cases. The main idea behind *replication logic* is that based on one's theory one expects that the phenomenon differs if the situations change or the same phenomenon occurs in the same situations. That is, representativeness is not the criteria for the selection of cases [59], rather the choice of each case should be made such that it either: predicts similar results for predictable reasons (i.e., literal replication); or predicts contrary results for predictable reasons (i.e., theoretical replication).

Because phenomenon and context are not always distinguishable in real-life situations, case studies offer a useful approach for combining several sources of evidence, such as observations, documents and archives, and interviews.

5.2.1 Study Setup

Following the methodology described in section 5.1 we set up a multiple case study to obtain results from the different evaluation methods. The goal was to explore different HCI methods to find usability problems in the API. We conducted a workshop (see section 5.1 – Phase 2) to get in-depth information about the actual usability of the API. A heuristic evaluation (see section 5.1 – Phase 3) to obtain findings identified by novice and expert developers. Additionally, we interviewed each developer (see section 5.1 – Phase 4) to gather qualitative data about the usability of the API. Both qualitative and quantitative data were gathered for analysis as a source for supporting evidence for potential usability problems.

The hands-on workshop had the primary goal to evaluate the usability of the API in its real-life context. The workshop started with a *setup session* that had the purpose to explain the particular requirements and objectives of the workshop. During this session, novice and expert developers obtained all the resources necessary to develop with the API and also got an introduction to the documentation of the API. In addition, they were given a demographic questionnaire that focuses on the programming experience.

The *setup session* was followed by an in-depth presentation providing developers with insights about the objectives of the workshop and the functional parts of the API (overall concepts).

After the theoretical introduction, novice and expert developers were asked to start developing their own *hands-on project*. They were provided with ideas that they could "*easily*" implement – Google Calendar, Google Drive, Gmail, and OPC Unified Architecture (UA). The developers were asked to use the documentation and templates available and to note all the problems, misunderstandings, potential bugs, and errors they identified. They were provided with six task scenarios defined by evaluators to cover as much of the API as possible (Table 5.1). In addition, they were given an *After-Scenario Questionnaire* developed by Lewis [60] to assess their subjective satisfaction regarding each task.

Table 5.1 List of six tasks for the workshop participants

Task No.	Description
Task 1	Define connector properties (e.g. username, password, etc.)
Task 2	Check the connection with the data provider
Task 3	Create session
Task 4	Obtain metadata and Generate metamodel (e.g. tables, columns, etc.)
Task 5	Simple SQL Select
Task 6	Return data

Only two developers (all men, mean age 23) participated in a five-day workshop. Pre-requirements for the developers participating have been defined through required prior knowledge in programming C#, SQL and the Visual Studio environment.

The heuristic evaluation was conducted after the workshop to identify usability problems of the API. The heuristic evaluation involves 42 heuristics based on API design guidelines identified by E. Mosqueira-Rey et al. [57]. These heuristics were selected as they summarize existing guidelines available in the literature on API usability but also cover other usability aspects that have been neglected by the literature. Table 5.2 gives a short description of the first-level attributes based on [57]. The heuristic evaluation required to analyse the API itself in consort with the documentation. The findings were collected by the evaluators. The heuristics were classified according to the following categories: *Yes* (it is fulfilled), *No* (it is not fulfilled), *Partially* (it is only in part fulfilled), and *Not Applicable* (it is not possible to apply the heuristic for numerous reasons).

Table 5.2 The first-level attributes for the API evaluation

Attribute	Description
Knowability	The API should be easy to understand, learn and remember. This attribute is subdivided into *clarity*, *consistency, memorability*, and *helpfulness.*
Operability	The API should provide the necessary functionalities to implement the tasks intended by the user. This attribute is subdivided into *completeness*, *precision, universality*, and *flexibility.*
Efficiency	The API should produce appropriate results in return for the resources that are invested. This attribute is subdivided into efficiency *in human effort*, *in task execution, in tied up resources*, and *in economic costs.*
Robustness	The API should resist error and adverse situations. This attribute is subdivided into robustness *to internal error*, *to improper use, to third-party abuse*, and *to environment problems.*
Safety	The API should avoid risk and damage derived from its use. This attribute is subdivided into *user safety*, *third-party safety,* and *environment safety.*
Subjective satisfaction	The API should produce feelings of pleasure and interest in users. This attribute is subdivided into *interest* and *aesthetics*.

The subjective analysis has been done after the heuristic evaluation (on the day after). Each developer had an interview with the evaluator to try to clarify any question that they could have had when filling the API usability questionnaire. Throughout the interviews, several usability aspects were addressed such as learnability, understandability, helpfulness misconceptions, errors, perception of the API, documentation, and user experience. The notes that have been given by the developers during the workshop, as well as the ASQ outcomes, act as a base for the interviews.

5.2.2 Analysis

In this section, we present the results of the analysis of the three different parts of the evaluation of the API.

Hands-on workshop. During the hands-on workshop different types of materials have been collected – notes, logs, and questionnaires. The collected material provided us with insights into understanding the usability problems of the API and the documentation.

Two participants were asked to fill out an *After-Scenario Questionnaire* (ASQ) for each of the six task scenarios to assess three important aspects of user satisfaction: ease of task completion, time to complete a task, and adequacy of support information (documentation). Each of these three aspects was measured based on a seven-point Likert Scale with the left and right anchor being "*Strongly Disagreed*" and "*Strongly Agreed*", respectively. We computed the results from the ASQ for each role, individually. Figure 5.2 shows the results of the ASQ for the expert developer. Results reveal that most of the tasks were perceived as easy to solve, except task 5.

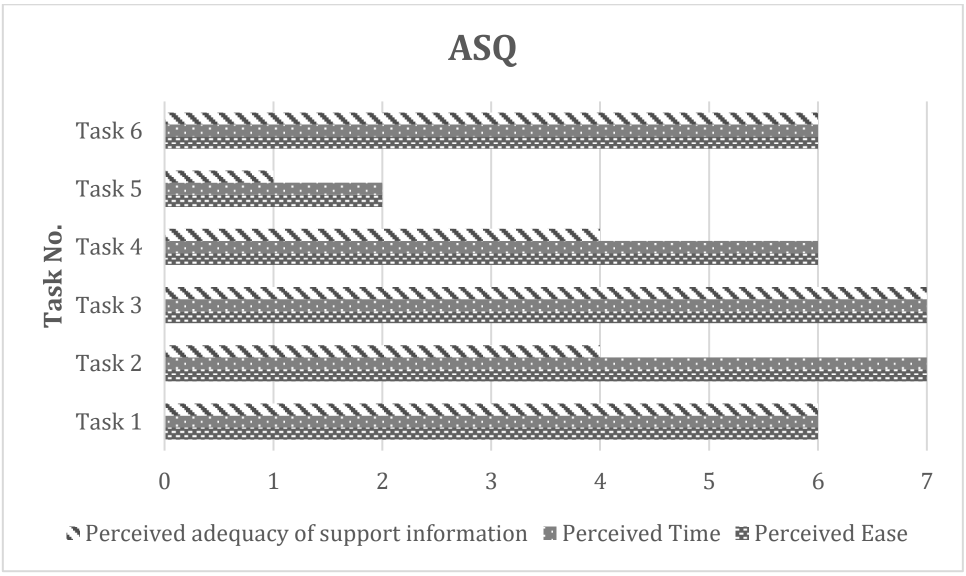


Figure 5.2 Results of the ASQ – Expert Developer

Figure 5.3 shows the results of the ASQ for the novice developer. Results reveal that the novice developer found tasks 1, 2 and 5 as difficult to solve.

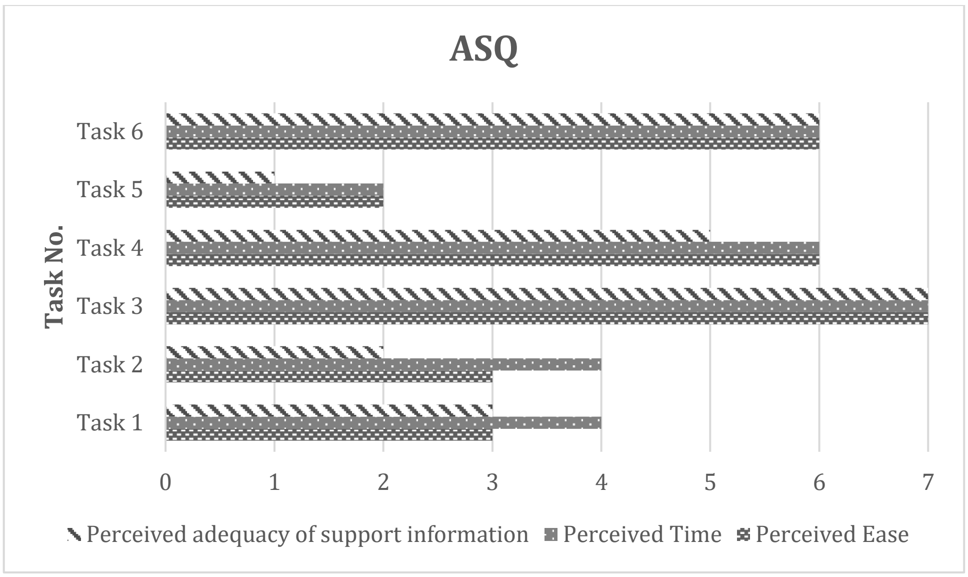


Figure 5.3 Results of the ASQ – Novice Developer

An interesting finding is that tasks perceived as difficult have associated documentation problems. In addition, both novice and expert developers perceived task 5 as difficult.

The issues identified in the workshop were mainly related to documentation issues, to the perception of names, to undetected errors, and to the unnecessary complexity. In the course of the workshop, evaluators had to provide further examples and explanations to help developers.

Heuristic evaluation. Two participants were asked to fill out the API Usability questionnaire. They inspected several classes and interfaces in 14 packages, and two sample projects using the API. The duration of one inspection varied between two to three hours. Heuristic evaluation explores all parts of the API equally.

The findings were collected and then analysed by the evaluators. The results of analysing the 42 heuristics were computed for each role, individually. Figure 5.4 shows the results of analysing the 42 heuristics for the expert developer. The problems that we found in the API were mostly from the knowability attribute: it was not easy to understand what the code did (KCS-2), and it was not clear what classes and methods developers need to use to perform some tasks (KCF-3). Also, the API was difficult to use because it was poorly documented (KHS-1), did not include only relevant information (KHS-2) and did not provide code samples for the most common scenarios (KHS-5).

We found also problems in other categories like operability – the API did not provide the functionalities necessary to implement common tasks (OC-1); robustness – the API did not use exception to expose potential errors (RI-1) – and finally, subjective satisfaction – using the API was not satisfying (SI-1).

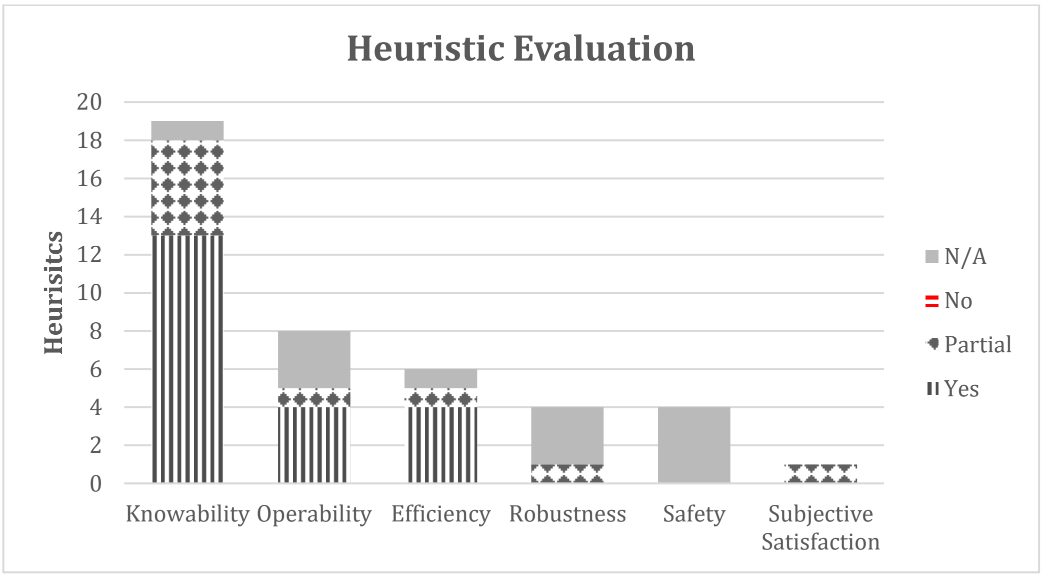


Figure 5.4 Results of the heuristic evaluation (N/A: Not Applicable) – Expert Developer

Figure 5.5 shows the results of analysing the 42 heuristics for the novice developer. Most of the problems that we found in the API were from the knowability (identical problems listed above).

We found also problems in other categories like operability – the API did not provide the functionalities necessary to implement common tasks (OC-1) and the API did not contain backwards compatibility deprecating properties (OC-2); and efficiency – the user needed to put too much effort when implementing a task (EH-1).

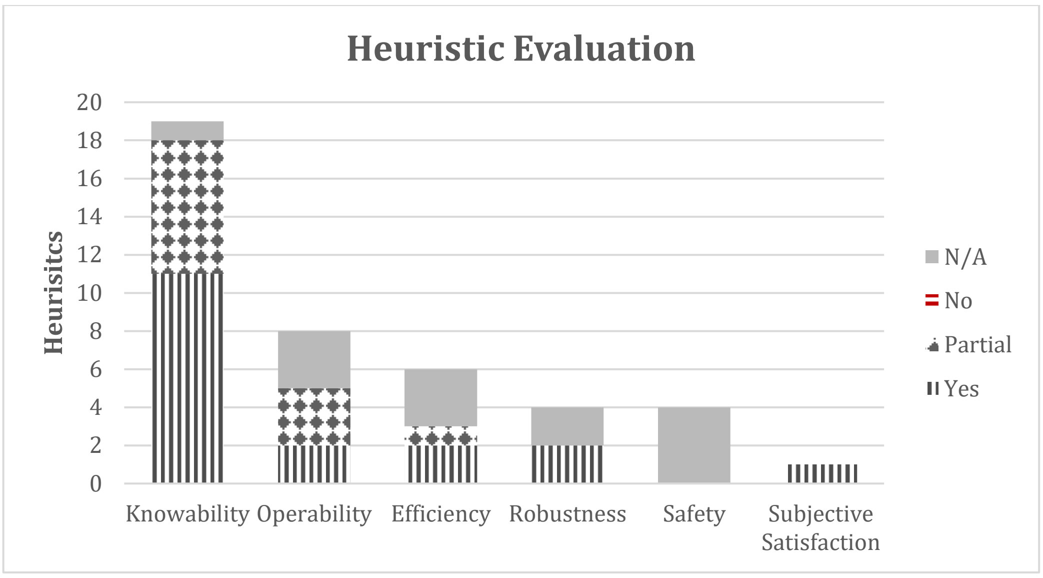


Figure 5.5 Results of the heuristic evaluation (N/A: Not Applicable) – Novice Developer

Subjective analysis. The interviews with novice and expert developers were conducted on the day after the workshop. The interviews allowed us to gain valuable insights into the main problems that the developers struggled with. Again, the largest number of problems in the API was mainly from the knowability attribute. Participants mentioned that the API was poorly documented – code snippets, source code documentation, sample projects, FAQs, and in-depth description of main concepts (e.g. metamodel, data handler, etc.) – and also the API does not expose vulnerabilities that allow users to make errors – lacks validation and compile-time errors. These problems are the most crucial to be corrected as they represent obstacles. Such obstacles can then easily discourage a novice developer from using the API.

Besides usability problems, interviews revealed the subjective opinion of the developers of the evaluated API. The expert developer liked the structure of the API and considered the API easier to learn and understand than the previous API. The novice developer negatively commented on the structure (mainly, the *chain of sequential processing handlers*). Both developers agreed that documentation needs to be improved.

5.2.3 Discussion

Before we discuss the findings from each part of the methodology, we will point out the context of the study that will help us draw conclusions from the results.

Each applied method has its strong and weak points and focuses on different areas of problems. A combination of the selected methods resulted in a more complete view of the overall usability. Combining such specific methods hence allows addressing related usability problems with an API in an appropriate way.

Despite heuristic evaluation involves a small group of *usability experts* analysing an API and examining its compliance with a predefined set of heuristics, in our methodological approach, novice and expert developers performed the heuristic evaluation. As stated by Molich and Jeffries in [61], heuristic evaluation can be applied by "*someone without particular knowledge of usability engineering*". Hertzum and Jacobsen also stated in [62] that "*any computer professional should be able to apply heuristic evaluation, but the informality of the method leaves much to the evaluator*". In fact, levels of expertise and experience of individual *evaluators* have evident influences on usability evaluation results – the so-called *evaluator effect* [62]. We tried to reduce the *evaluator effect* during the hands-on workshop by providing an in-depth introduction to the documentation of the API and to its functional parts (main concepts).

Furthermore, in our methodological approach, we included a small sample of participants because 1) the case-study requires extensive resources and time, and 2) our supervisor agreed that a small sample would be sufficient to draw useful and reliable conclusions. Due to the sample size, our findings have limited generalizability.

The findings were different from each part of the methodology. Whereas heuristic evaluation provided formal and detailed descriptions of problems in the API, the workshop focusses on usability problems regarding concept and structure of the API. During the workshop, we could reveal more runtime problems that are not obvious and thus cannot be found in the heuristic evaluation. Interviews not only provided a deeper understanding of these problems but also revealed the subjective opinions to the API. An excerpt of the findings is shown in Table 5.3.

Table 5.3 Findings (HE = Heuristic evaluation, W = workshop, SA = subjective analysis)

Finding	Phase
…	
Class: Model	HE

Finding: Interfaces are used as arguments of functions that expect a specific data type to work correctly. *Heuristic*: KCE-2	
Class: Model *Finding:* constructor requires default schema name; but the schema is not automatically generated *Heuristic*: KCF-1	HE
Class: *OAuthConnector* *Finding:* *PluginPropertyList* should be hidden and marked as deprecated. *Heuristic*: OC-2	HE
Class: *DataHandler* *Finding:* Missing mechanisms to access connector properties. *Heuristic*: OC-1	HE
The use of the data type Any Type resulted in runtime errors (columns were not presented in the CB Query Analyser).	W
Tables accept several columns with same ordinal number.	W
There are missing helpers for XML/JSON parser.	SA
There is missing some validation at compile time.	SA
I'm not happy using the API mainly due to poor documentation which leads to frustration.	SA
It would be good to have a step-by-step guide.	SA
The metamodel part is simple and pleasing	SA
The data handlers are very difficult to understand without proper examples and diagrams.	SA
…	

After all, we decided to modify the API focusing mainly on the documentation problems that were easy to solve at this stage of development. Nonetheless, problems in other categories were left for future work.

Chapter 6

Conclusions and Future Work

This chapter presents the main conclusions about the work done and some challenges of this dissertation, as well as a few ideas for future work that must be done in order to have a fully functional SDK for the creation of connectors to integrate with CB Server which can be used with minimal technical support by the developers.

6.1 Conclusions

In this dissertation, we have presented an SDK for the creation of connectors to integrate with CB Server which accelerates deployment, ensures best practices and simplifies the various activities and tasks in the software development process. The SDK offers a public, simple API together with a set of tools exploiting the API. The key requirements of the API are: to store a collection of both data and metadata for relational and non-relational data sources and to allow inserting, updating, deleting, and retrieving data from the data source.

For this, we first had to align the connector metamodel with current versions of the MOF, exploiting the MDD metamodeling approach. The connector metamodel is a metamodel of a relational data model. We introduced several concepts and constructs for the alignment. We believe that the proposed concepts and constructs could be useful for modelling metadata for relational and non-relational data sources. A formal definition of the semantics of SQL queries has been implemented to allow inserting, updating, deleting, and retrieving data from the data source, but limited to SQL data manipulation commands (that is, SELECT, UPDATE, DELETE, INSERT, and EXEC). Additionally, the backbone of a functional connector has also been defined, as well as a rich set of custom exceptions including additional information (e.g., error code, message, and so forth). Furthermore, a set of utilities has been developed to help developers in the software development process.

To prove the adequacy of our approach, we presented two examples and case studies. The first case study presents a connector for a relational data source (MS SQL Server). The second case study presents a connector for a non-relational data source (OData). The outcome was a list of issues found in the API and a set of proposals for improvement. Based on the results we had to make some improvements in both API and CB Server.

To evaluate the usability of the API a methodology based on several usability evaluation methods has been developed. Combining the HCI methods is important as it allows us to get a broad view of problems covering several aspects of API usability. We applied this method in a multiple case study. The applied methodology generated insights based on an inspection method, a user test, and interviews. We identify not only problems and flaws in the source code, but also runtime, structural and documentation problems, as well as problems related to user experience. Based on the findings we decided to modify the API focusing mainly on the documentation problems.

Although the API is targeted to the connector development process, we think that our approach can be applied to any other formal method to develop software around the fields of relational databases, data integration, and data modelling.

6.2 CHALLENGES

This dissertation presented many challenges. First, we had to study the basic workflow in CB Platform so that the "new" SDK overcomes the problems identified in the previous SDK. For this, we had to develop a simple connector without prior knowledge, and to read some internal documentation that were missing many details and information. Second, we had to provide a public API so that external developers, in a directed or undirected way, can build connectors that the company is unable to develop itself. For this, from a technical standpoint, the system architecture, including its interface to external developers, is the central, critical pillar and serious mistakes will probably cause significant dissatisfaction and rejection among the external developers. The challenge here is to provide a stable and easy to use and learn interface between the CB platform and the external developers that evolves in an expected way; without prior experience in system architecture. Finally, we had to develop tools and documentation to help developers in the software development process. For this, we had to study how to use wizards with project templates and how to generate documentation automatically with Sandcastle.

6.3 FUTURE WORK

Future work will include the development of new functionalities in the API such as a *connector wrapper* for "old" connectors to exploit as much as possible the "new" API, and standard implementation for all data handlers to emulate the pipeline. Furthermore, we aim to solve some problems found in the applied methodology described in Section 5 to evaluate the usability of the API, namely problems and flaws in the source code (e.g. validations) and structural problems.

Chapter 7

References

[1] P. Ziegler and K. R. Dittrich, "Data Integration — Problems, Approaches, and Perspectives," in *Conceptual Modelling in Information Systems Engineering*, Springer, Berlin, Heidelberg, 2007, pp. 39–58.

[2] E. M. Zaman, "Information Integration for Heterogeneous Data Sources," *IOSR J. Eng.*, vol. 02, no. 04, pp. 640–643, Apr. 2012.

[3] T. M. Connolly and C. E. Begg, *Database Systems: A Practical Approach to Design, Implementation, and Management*. Pearson Education, 2005.

[4] C. Coronel and S. Morris, *Database Systems: Design, Implementation, & Management*. Cengage Learning, 2016.

[5] A. G. Taylor, *SQL for dummies*, 5th ed. Hoboken, NJ: Wiley, 2003.

[6] E. Eessaar, "Using Metamodeling in Order to Evaluate Data Models," in *Proceedings of the 6th Conference on 6th WSEAS Int. Conf. On Artificial Intelligence, Knowledge Engineering and Data Bases - Volume 6*, Stevens Point, Wisconsin, USA, 2007, pp. 181–186.

[7] R. Ramakrishnan and J. Gehrke, *Database Management Systems*, 2nd ed. New York, NY, USA: McGraw-Hill, Inc., 2000.

[8] J. F. Courtney, D. B. Paradice, K. L. Brewer, and J. C. Graham, *Database Systems For Management, 3rd Edition*, 3rd ed. 2010.

[9] E. F. Codd, "Data Models in Database Management," in *Proceedings of the 1980 Workshop on Data Abstraction, Databases and Conceptual Modeling*, New York, NY, USA, 1980, pp. 112–114.

[10] R. Elmasri and S. Navathe, *Fundamentals of database systems*, 6th ed. Boston: Addison-Wesley, 2011.

[11] S. B. Navathe, “Evolution of Data Modeling for Databases,” *Commun ACM*, vol. 35, no. 9, pp. 112–123, Sep. 1992.

[12] E. F. Codd, *The Relational Model for Database Management: Version 2*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1990.

[13] E. F. Codd, “Extending the Database Relational Model to Capture More Meaning,” *ACM Trans Database Syst*, vol. 4, no. 4, pp. 397–434, Dec. 1979.

[14] J. R. Groff, P. N. Weinberg, and A. Oppel, *SQL The Complete Reference, 3rd Edition*. McGraw Hill Professional, 2008.

[15] M. Völter, T. Stahl, J. Bettin, A. Haase, and S. Helsen, *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2013.

[16] R. Picek and V. Strahonja, “Model Driven Development - Future or Failure of Software Development?,” in *In IIS’07: 18th International Conference on Information and Intelligent Systems*, 2007.

[17] B. Selic, “Model-driven development: its essence and opportunities,” in *Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC’06)*, 2006, pp. 7 pp.-.

[18] S. J. Mellor, A. N. Clark, and T. Futagami, “Model-Driven Development,” *IEEE Softw.*, p. 5, 2003.

[19] C. Lange, M. Chaudron, and J. Muskens, “In Practice: UML Software Architecture and Design Description,” *IEEE Softw*, vol. 23, no. 2, pp. 40–46, Mar. 2006.

[20] A. W. Brown, J. Conallen, and D. Tropeano, “Introduction: Models, Modeling, and Model-Driven Architecture (MDA),” in *Model-Driven Software Development*, Springer, Berlin, Heidelberg, 2005, pp. 1–16.

[21] R. Soley, “Model Driven Architecture.” OMG, Nov-2000.

[22] “In practice: UML software architecture and design description - IEEE Journals & Magazine.” [Online]. Available: https://ieeexplore.ieee.org/document/1605177/. [Accessed: 19-Jul-2018].

[23] J. Ludewig, “Models in software engineering – an introduction,” *Softw. Syst. Model.*, vol. 2, no. 1, pp. 5–14, Mar. 2003.

[24] J. Favre, “Megamodeling and etymology - a story of words: From MED to MDE via MODEL in five milleniums,” presented at the In Dagstuhl Seminar on Transformation Techniques in Software Engineering, number 05161 in DROPS 04101. IFBI, 2005.

[25] J. Bezivin and O. Gerbe, “Towards a precise definition of the OMG/MDA framework,” in *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, 2001, pp. 273–280.

[26] E. Seidewitz, “What models mean,” *IEEE Softw.*, vol. 20, no. 5, pp. 26–32, Sep. 2003.

[27] “MDA Guide Version 1.0.1.” OMG, Jun-2003.

[28] J.-M. Favre, “Foundations of Model (Driven) (Reverse) Engineering - Episode I: Story of The Fidus Papyrus and the Solarus,” presented at the Dagsthul Seminar On Model Driven Reverse Engineering, 2004.

[29] A. G. Kleppe, J. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[30] J. Favre, “Foundations of meta-pyramids: Languages vs. metamodels – episode II: Story of thotus the baboon,” in *Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl*, 2005.

[31] C. Atkinson and T. Kuhne, “Model-driven development: a metamodeling foundation,” *IEEE Softw.*, vol. 20, no. 5, pp. 36–41, Sep. 2003.

[32] J. Bézivin, “In Search of a Basic Principle for Model Driven Engineering,” *Novatica/Upgrade*, vol. 5, 2004.

[33] J. Bezivin, “Model Engineering: From Principles to Platform,” in *WIT-Kolloquium*, Technical University of Vienna, 2005.

[34] “OMG Meta Object Facility (MOF) Core Specification.” OMG, Nov-2016.

[35] “Meta Object Facility.” OMG, Apr-2002.

[36] B. Hailpern and P. Tarr, "Model-driven development: The good, the bad, and the ugly," *IBM Syst. J.*, vol. 45, no. 3, pp. 451–461, 2006.

[37] M. Fowler, "Language Workbenches: The Killer-App for Domain Specific Languages?" [Online]. Available: https://martinfowler.com/articles/languageWorkbench.html. [Accessed: 23-Jul-2018].

[38] "About OMG," *About OMG*, 2018. [Online]. Available: https://www.omg.org/about/. [Accessed: 09-Jul-2018].

[39] J. Pike, "We Set the Standard," p. 3.

[40] E. Keremitsis and I. J. Fuller, "HP distributed smalltalk: a tool for developing distributed applications," vol. 46, pp. 85–92, Apr. 1995.

[41] J. Poole, D. Chang, D. Tolbert, and D. Mellor, *Common Warehouse Metamodel*. John Wiley & Sons, 2002.

[42] B. Selic, "What's New in UML 2.0? Technical report IBM Rational Software," IBM Corporation, Apr. 2005.

[43] U. Aßmann, S. Zschaler, and G. Wagner, "Ontologies, Meta-models, and the Model-Driven Paradigm," in *Ontologies for Software Engineering and Software Technology*, Springer, Berlin, Heidelberg, 2006, pp. 249–273.

[44] "UML 2.2 - UML Infrastructure Specification." OMG, Feb-2009.

[45] M. J. Emerson, J. Sztipanovits, and T. Bapty, "A MOF-Based Metamodeling Environment.," *J UCS*, vol. 10, pp. 1357–1382, Jan. 2004.

[46] A. Butković Tomac and D. Tomac, "Metadata Interchange in Service Based Architecture," *J. Inf. Organ. Sci.*, vol. 27, no. 2, pp. 73–79, Dec. 2003.

[47] I. Sommerville, *Software Engineering*, 8th ed. Pearson Education, 2007.

[48] C. Bock *et al.*, "UML 2.5 - UML Specification." OMG, Mar-2015.

[49] T. Cowling, "Model-driven development and the future of software engineering education," in *2013 26th International Conference on Software Engineering Education and Training (CSEE T)*, 2013, pp. 329–331.

[50] K. Delaney, A. Machanic, P. S. Randal, K. L. Tripp, C. Cunningham, and B. Nevarez, *Microsoft SQL Server 2008 Internals*. Microsoft Press, 2009.

[51] "OData - the Best Way to REST." [Online]. Available: https://www.odata.org/. [Accessed: 12-Aug-2018].

[52] M. Henning, "API Design Matters," *Commun ACM*, vol. 52, no. 5, pp. 46–56, May 2009.

[53] ISO 9241-11, "Ergonomic Requirements for Office Work with Visual Display Terminals (VDTs)," in *Part 11: Guidance on usability*, Genova: International Organization for Standardization, 1998.

[54] B. E. John and D. E. Kieras, "Using GOMS for User Interface Design and Evaluation: Which Technique?," *ACM Trans Comput-Hum Interact*, vol. 3, no. 4, pp. 287–319, Dec. 1996.

[55] T. R. G. Green and M. Petre, "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework," *J. Vis. Lang. Comput.*, vol. 7, no. 2, pp. 131–174, Jun. 1996.

[56] "IEEE Standard for a Software Quality Metrics Methodology," *IEEE Std 1061-1992*, p. 3, 1993.

[57] E. Mosqueira-Rey, D. Alonso-Ríos, V. Moret-Bonillo, I. Fernández-Varela, and D. Álvarez-Estévez, "A systematic approach to API usability: Taxonomy-derived criteria and a case study," *Inf. Softw. Technol.*, vol. 97, pp. 46–63, May 2018.

[58] R. K. Yin, *Case Study Research: Design and Methods*. Sage Publications, 1994.

[59] R. E. Stake, *The Art of Case Study Research*. Sage Publications, 1995.

[60] J. R. Lewis, "IBM Computer Usability Satisfaction Questionnaires: Psychometric Evaluation and Instructions for Use (IBM Technical Report 54.786)," *Int. J. Hum.-Comput. Interact.*, pp. 57–78, 1995.

[61] R. Molich and R. Jeffries, "Comparative Expert Reviews," in *CHI '03 Extended Abstracts on Human Factors in Computing Systems*, New York, NY, USA, 2003, pp. 1060–1061.

[62] M. Hertzum and N. E. Jacobsen, “The Evaluator Effect: A Chilling Fact about Usability Evaluation Methods,” p. 17.

[63] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.

[64] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[65] M. Fowler, *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[66] “SQL Server Schema Collections,” *Microsoft Docs*. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/sql-server-schema-collections. [Accessed: 15-Jun-2018].

APPENDIX A

UML concepts and drawing conventions

This appendix uses a simple restaurant model to summarize the UML concepts and drawing conventions used in the descriptions of the packages (Figure A.1). As Enterprise Architect (EA) was used for these descriptions, the UML concepts and drawing conventions shown here are related with EA convections and drawing concepts.

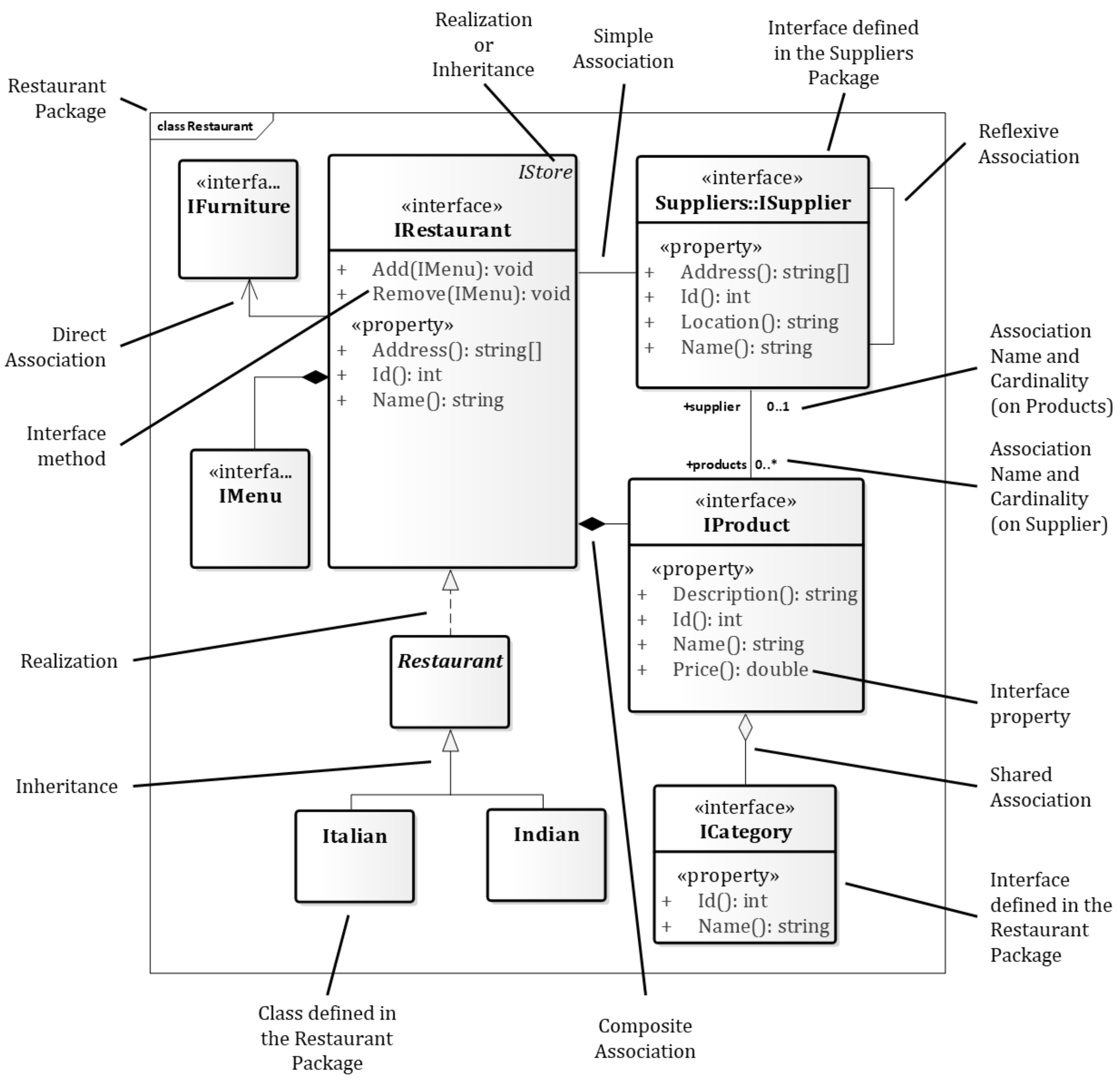


Figure A.1 A simple restaurant model illustrating how packages are described

Appendix B

Architectural Styles and Design Patterns

This appendix reviews the main *architecture styles* and *design patterns* used throughout this dissertation.

B.1 Architectural Styles

An architectural style characterizes a family of related systems in terms of a pattern of structural organizations [63]. More precisely, an architectural style defines a *vocabulary* of elements – components and connector types – together with a set of rules – or *constraints* – on how that vocabulary is used, and semantic assumptions about that vocabulary.

There are several commonly used architectural styles, such as: objects, pipes and filters, implicit invocation, layering, repositories, interpreters, and process control. Our focus here is only on *pipes and filters* style.

B.1.1 Pipes and Filters

In [63], Garlan and Shaw describe the *Pipes and Filters* style as follows:

> *"In a pipe-and-filter style each component has a set of inputs and a set of outputs. A component reads streams of data on its inputs and produces streams of data on its outputs. […] Hence components are termed* ***filters****. The connectors of this style serve as conduits for the streams, transmitting outputs of one filter to another. Hence the connectors are termed* ***pipes****. […] filters must be independent entities: in particular, they should not share state with other filters. […] filters do not know the identity of their upstream and downstream filters."*

In other words, a *Pipes and Filters* style consists of several components – named *filters* – and connectors – named *pipes*. Each *filter* processes a local and incremental data transformation on the data received on its inputs and sends the results on its outputs. Each *pipe* represents a sequential flow of data between two filters, in which the output stream of a filter flows to the input of the next filter. *Filters* can be characterized by their input/output behaviour: **source filters** produce a stream without any input; **transform filters** consume an input stream and produce an output stream; and **sink filters** consume an input stream but do not produce any output. Figure B.1 illustrates this style.

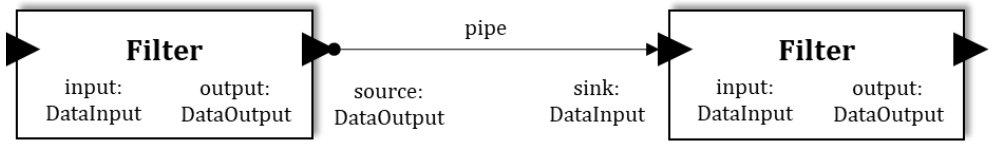


Figure B.1 Pipes and Filters

The pipe and filter style has several good properties that make it attractive and efficient for numerous applications. It is relatively simple to describe, understand and implement. It is also quite intuitive and allows to model systems while preserving the flow of data. Because of the interaction modalities between filters, complex systems described in terms of data flowing from filter to filter are easily understandable as a series of local transformations. The implementation details of each filter are irrelevant to understand the overall system, as long as a logical definition of their behaviour is specified (input, transformation and output). The localization and isolation of transformations facilitates design, implementation, maintenance and evolution. Additionally, filters can be implemented and tested individually. Furthermore, because filters are independent entities, filters allow reusability and interoperability, as well as parallel and distributed processing, which contribute to efficiency and scalability.

Nonetheless, it also has their shortcomings and limitations. Because the pipes are first-in-first-out (FIFO), the overall throughout of a Pipes and Filters system is imposed by the transmission rate of the slowest filter in the system. If filters independence provides a natural design for parallel and distributed processing, the pipes impose arbitrary transmission bottlenecks that make it non-optimal. It also can lead to massive/expensive data copying, loss of performance and increased complexity.

Common specializations of this style include: (1) *pipelines*, which restricts the topologies to linear sequences of filters; (2) *bounded pipes*, which restrict the amount of data that can reside on a pipe; and (3) *typed pipes*, which require that the data passed between two filters have a well-defined type.

The *pipeline* approach is based on the pure form of *Pipes and Filters* style, in which each filter has only a single input stream and a single output stream. Within *pipeline* approach, there are three *alternatives* to control data flowing from filter to filter: (1) *push-flow*, in which each filter of the pipeline *pushes* data in a downstream; (2) *pull-flow*, in which each filter of the pipeline *pulls* the data from an upstream; and (2) *push/pull-flow*, in which each filter of the pipeline may *pull* data from an upstream and *push* transformed data in a downstream.

B.2 Design Pattern

In [64], Gamma et al. stated that: "*a design pattern **names**, **abstracts**, and **identifies** the key aspects of a common design structure that make it useful for creating a **reusable** object-oriented design*". Several ways in which design patterns can affect the way we design object-oriented software include:

– A common design vocabulary for designers to communicate, document, and explore design alternatives at a higher level of abstraction so that software complexity is reduced.

– A flexible and reusable base of experience and knowledge.

– A target for reducing the amount of refactoring.

Design patterns describe a problem which occurs repeatedly in our environment, and then describe the solution to that problem so that we can use this solution, without doing it same way twice.

We describe design patterns using a consistent format. Each pattern consists of three essential parts: the pattern name, an abstract description, and the consequences of using the design pattern to a system's architecture.

B.2.1 Lazy Load Pattern

The *Lazy Load Pattern* is a *behavioural pattern* in which an object, that does not contain all the data, encapsulates all the information needed to get it [65]. There are four main ways we can implement *Lazy Load*:

- **Lazy initialization**, which is the simplest approach. The basic idea is that every access to the field checks first to see if it's null. If so, it calculates the value of the field before returning the field. A **consequence** of using the *Lazy initialization* is:
  - It tends to force a dependency between the object and the *database*.
- **Virtual proxy**, which is an object that provides a surrogate or placeholder for another object that should be in the field but doesn't contain actually anything. Only when one of its methods is called does it load the correct object. Some **consequences** of using the *Virtual Proxy* are:
  - It can hide the fact that an object resides in a different address space.
  - It can perform optimizations such as creating an object on demand.
  - Both protection proxies and smart references allow additional housekeeping tasks when an object is accessed.
- **Value holder**, which is an object that wraps another object. To get the underlying object we ask the value holder for its value, but only on the first access does it actually load. A **consequence** of using the *Value holder* is:
  - It loses the explicitness of strong typing.
- **Ghost**, which is a real object in a partial state. In essence, a ghost is an object where every field is lazy-initiated, or as a virtual proxy, where the object is its own virtual proxy. In fact, a ghost doesn't need to be completely "empty"; for instance, it may make sense to load data that is quick to get and commonly used when we load the ghost.

A **consequence** of using the *Lazy Load Pattern* is:

1. It improves performance when the initialization of the object is costly – often, loading one object has the effect of loading a huge number of related objects; to

solve this problem, *Lazy Load Pattern* interrupts this loading process for the moment, leaving a marker in the object structure so that if the data is needed it can be loaded only when it is used.

B.2.2 Composite Pattern

The *Composite Pattern* is a *structural pattern*, which deals with the composition of classes or objects. More precisely, this design pattern composes "*into tree structures to represent part-whole hierarchies [...]; lets clients treat individual objects and compositions of objects uniformly*" [64].

Some **consequences** of using the *Composite Pattern* are:

1. It defines class hierarchies consisting of primitive objects and composite objects.
2. It makes the client simple.
3. It makes it easier to add new types of components.
4. It can make software design overly general.

B.2.3 Abstract Factory Pattern

The *Abstract Factory Pattern* is a *creational pattern*, which abstracts the process of object creation. More precisely, this design pattern provides "*an interface for creating families of related or dependent objects without specifying their concrete classes*" [64].

Some **consequences** of using the *Abstract Factory Pattern* are:

1. It isolates concrete classes.
2. It makes exchanging product families easy.
3. It promotes consistency among products.
4. Supporting new kinds of products is difficult.

B.2.4 Builder Pattern

The *Builder Pattern* is a *creational pattern*, which abstracts the process of object creation. More precisely, this design pattern separates "*the construction of a complex object from* *its representation so that the same construction process can create different representations*" [64].

Some **consequences** of using the *Builder Pattern* are:

1. It lets us vary a product's internal representation.
2. It isolates code for construction and representation.
3. It gives us finer control over the construction process.

APPENDIX C

SQL Grammar

This appendix describes the **SQL grammar**.

Table C.1 Syntax of SQL Queries

	QUERY BLOCK
1	<QUERY> = **SELECT** [**ALL** | **DISTINCT**] <SELECT LIST> <FRCLAUSE> [<WHCLAUSE>] [<GBCLAUSE>] [<HCLAUSE>] [OBCLAUSE]
2	<SELECT LIST> = list of <SELECT ELEMENT>
3	<SE.ECT ELEMENT> = <COL OR VAL> | <FUNC SPEC>
4	<COL OR VAL> = <correlation name><column name> | <literal>
5	<FUNC SPEC> = <COUNT FUNCTION SPEC> | <AGGR FUNCTION SPEC>
6	<COUNT FUNCTION SPEC> = <DISTINCT COUNT FUNCTION> | **COUNT (*)**
7	<AGGR FUNCTION SPEC> = <DISTINCT AGGR FUNCTION> | <ALL AGGR FUNCTION>
8	<DISTINCT COUNT FUNCTION> = **COUNT** (**DISTINCT** <correlation name><column name>)
9	<DISTINCT AGGR FUNCTION> = <AGGR FUNCTION NAME> (**DISTINCT** <correlation name><column name>)
10	<ALL AGGR FUNCTION> = <AGGR FUNCTION NAME> ([**ALL**] <correlation name><column name>)
11	<AGGR FUNCTION NAME> = **AVG** | **MAX** | **MIN** | **SUM**
12	<FRCLAUSE> = FROM <TABLE REFERENCE> [. <TABLE REFERENCE>]
13	<TABLE REFERENCE> = <table name><correlation name>
14	<WHCLAUSE> = **WHERE** <WHERE SEARCH COND>
15	<GBCLAUSE> = **GROUP BY** <correlation name><column name>
16	<HCLAUSE> = **HAVING** <HAVING SEARCH COND>
17	<OBCLAUSE> = **ORDER BY** <correlation name><column name> [**ASC** | **DESC**]
	WHERE SEARCH CONDITION
18	<WHERE SEARCH COND> = “Boolean Expression of” <WPRED>
19	<WPRED> = <SIMPLE PRED> | <COMPLEX PRED>
20	<SIMPLE PRED> = <COL OR VAL> <comp op><COL OR VAL>
21	<COMPLEX PRED> = <ANY QUATIFIED PRED>|<ANY QUANTIFIED AFPRED>|<ALL QUANTIFIED PRED>| |<ALL QUANTIFIED AFPRED>|<COMPLEX IN PRED>|<COMPLEX IN AFPRED>|<COMPLEX NOT IN PRED>| |<COMPLEX NOT IN AFPRED>|<EXISTS PRED>|<COMPLEX COMP PRED>|<COMPLEX COMP AFPRED>
22	<ANY QUATIFIED PRED> = <COL OR VAL><comp op>**ANY**<SUBQ>
23	<ANY QUATIFIED AFPRED> = <COL OR VAL><comp op>**ANY**<AFSUBQ>
24	<ALL QUATIFIED PRED> = <COL OR VAL><comp op>**ALL**<SUBQ>
25	<ALL QUATIFIED AFPRED> = <COL OR VAL><comp op>**ALL**<AFSUBQ>
26	<COMPLEX IN PRED> = <COL OR VAL>**IN**<SUBQ>
27	<COMPLEX IN AFPRED> = <COL OR VAL>**IN**<AFSUBQ>
28	<COMPLEX NOT IN PRED> = <COL OR VAL>**NOT IN**<SUBQ>
29	<COMPLEX NOT IN AFPRED> = <COL OR VAL>**NOT IN**<AFSUBQ>
30	<EXISTS PRED> = **EXISTS**<SUBQ>
31	<COMPLEX COMP PRED> = <COL OR VAL><comp op><SUBQ>
32	<COMPLEX COMP AFPRED> = <COL OR VAL><comp op><AFSUBQ>
	HAVING SEARCH CONDITION
33	<HAVING SEARCH COND> = “Boolean Expression of” <HPRED>

34	<HPRED> = <HSIMPLE PRED>|<HCOMPLEX PRED>|<HACOL PRED>|<HAFUNC PRED>|<HACOMPLELX PRED>
35	<HSIMPLE PRED> = <SIMPLE PRED>
36	<HCOMPLEX PRED> = <COMPLEX PRED>
37	<HACOL PRED> = <FUNC SPEC><comp op><COL OR VAL>
38	<HAFUNC PRED> = <FUNC SPEC><comp op><FUNC SPEC>
39	<HACOMPLEX PRED> = <AFANY QUANTIFIED PRED>|<AFANY QUANTIFIED AFPRED>|<AFALL QUANTIFIED PRED>| |<AFALL QUANTIFIED AFPRED>|<AFCOMPLEX IN PRED>|<AFCOMPLEX IN AFPRED>|<AFCOMPLEX NOT IN PRED>| |<AFCOMPLEX NOT IN AFPRED>|<AFCOMPLEX COMP PRED>|<AFCOMPLEX COMP AFPRED>
40	<AFANY QUATIFIED PRED> = <FUNC SPEC><comp op>**ANY**<SUBQ>
41	<AFANY QUATIFIED AFPRED> = < FUNC SPEC><comp op>**ANY**<AFSUBQ>
42	<AFALL QUATIFIED PRED> = <FUNC SPEC><comp op>**ALL**<SUBQ>
43	<AFALL QUATIFIED AFPRED> = <FUNC SPEC><comp op>**ALL**<AFSUBQ>
44	<AFCOMPLEX IN PRED> = <FUNC SPEC>**IN**<SUBQ>
45	<AFCOMPLEX IN AFPRED> = <FUNC SPEC>**IN**<AFSUBQ>
46	<AFCOMPLEX NOT IN PRED> = <FUNC SPEC>**NOT IN**<SUBQ>
47	<AFCOMPLEX NOT IN AFPRED> = <FUNC SPEC>**NOT IN**<AFSUBQ>
48	<AFCOMPLEX COMP PRED> = <FUNC SPEC><comp op><SUBQ>
49	<AFCOMPLEX COMP AFPRED> = <FUNC SPEC><comp op><AFSUBQ>
	SUBQUERY BLOCK
50	<SUBQ> = **SELECT** [**ALL** | **DISTINCT**] <COL OR VAL><FRCLAUSE> [<WHCLAUSE>] [<GBCLAUSE>] [<HCLAUSE>] [OBCLAUSE]
51	<AFSUBQ> = **SELECT** [**ALL** | **DISTINCT**] <FUNC SPEC><FRCLAUSE> [<WHCLAUSE>] [<GBCLAUSE>] [<HCLAUSE>] [OBCLAUSE]

APPENDIX D

Architectural Design

This appendix presents absent parts of the architectural design presented in section 3.2.

D.1 Data Handler Package

This appendix shows the abstract syntax for Data Handler package.

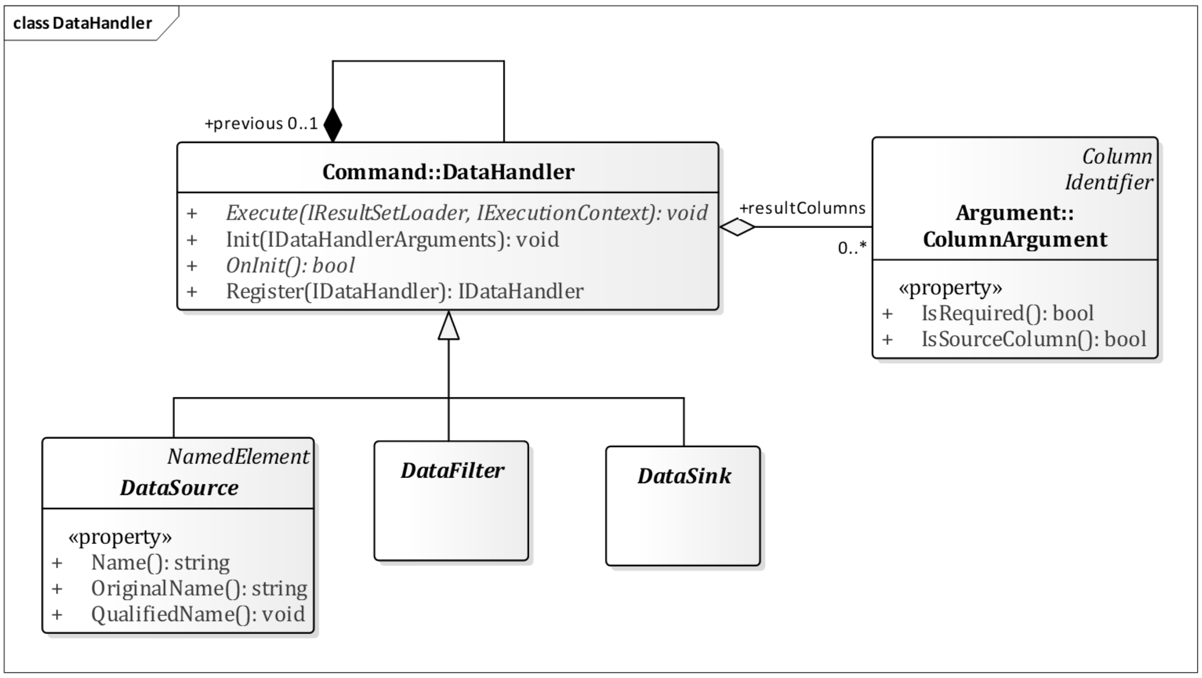


Figure D.1 Abstract syntax for Data Handler package – Data Handler

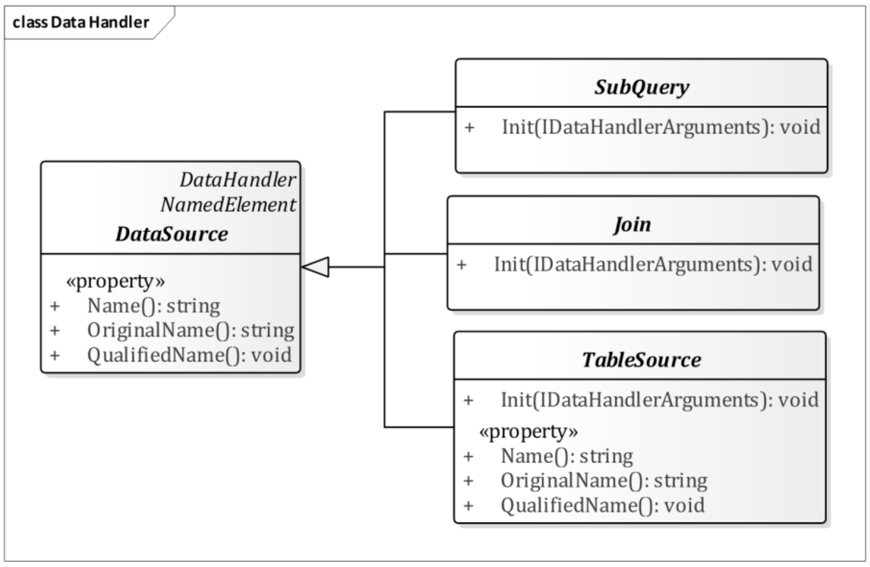


Figure D.2 Abstract syntax for Data Handler package – Data Source

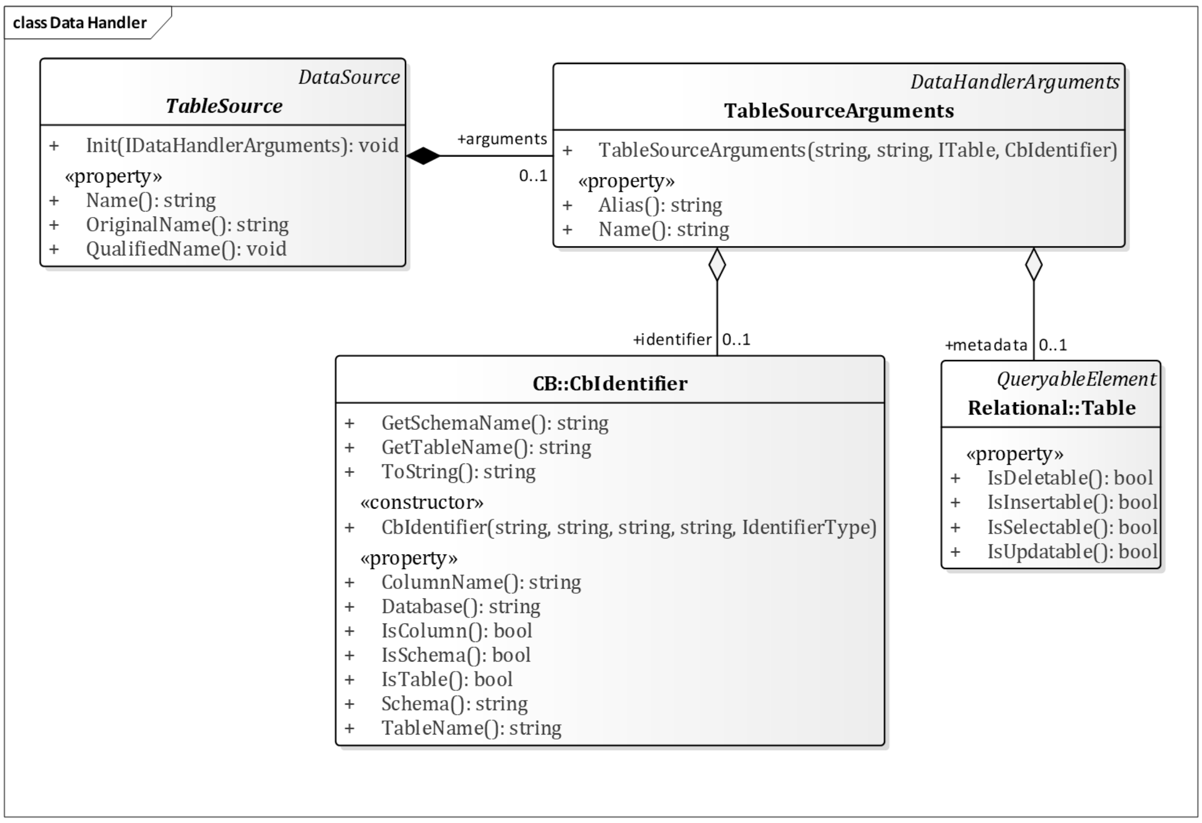


Figure D.3 Abstract syntax for Data Handler package – Table Reference

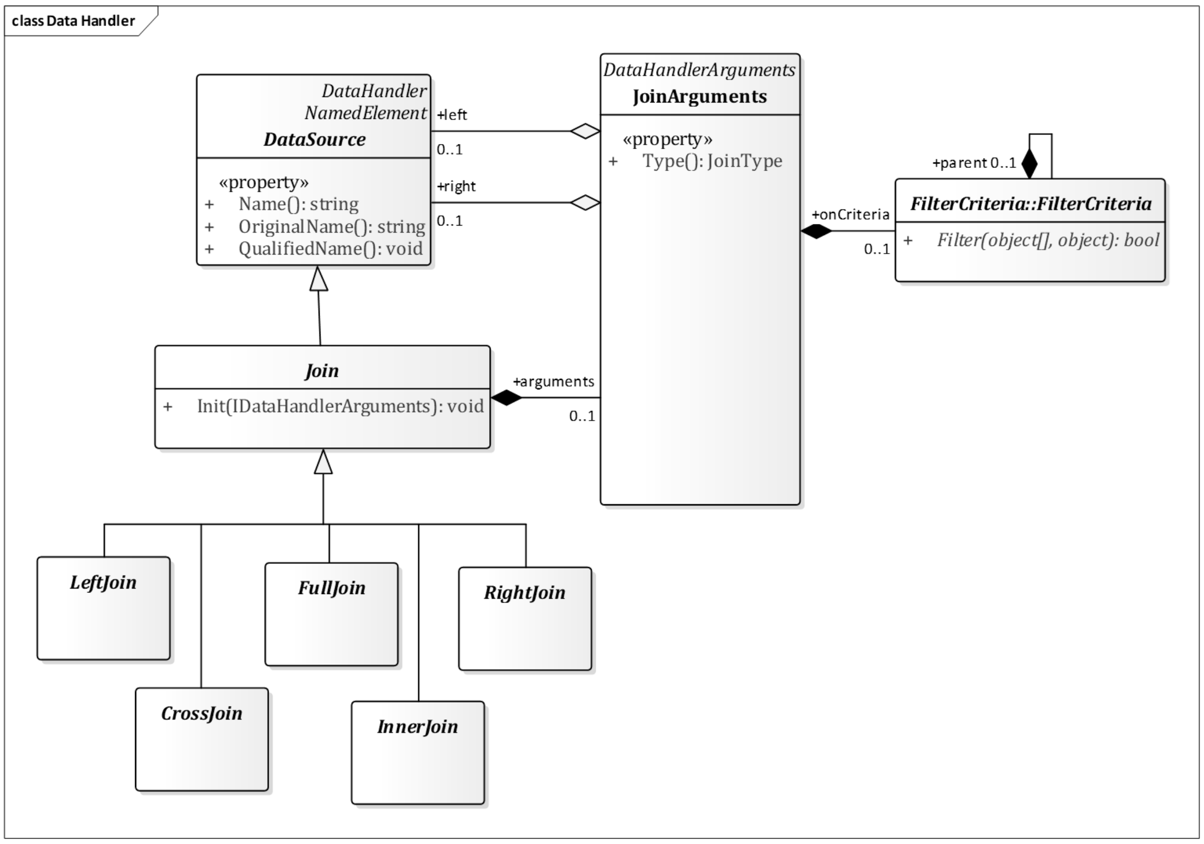


Figure D.4 Abstract syntax for Data Handler package – Join

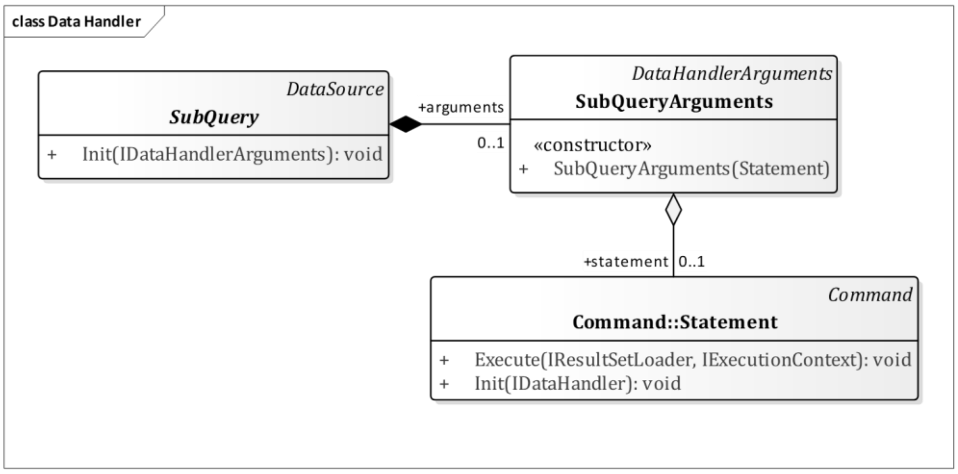

class Data Handler
DataSource
SubQuery
+ Init(IDataHandlerArguments): void
+arguments
0..1
DataHandlerArguments
SubQueryArguments
«constructor»
+ SubQueryArguments(Statement)
+statement
0..1
Command
Command::Statement
+ Execute(IResultSetLoader, IExecutionContext): void
+ Init(IDataHandler): void

Figure D.5 Abstract syntax for Data Handler package – Subquery

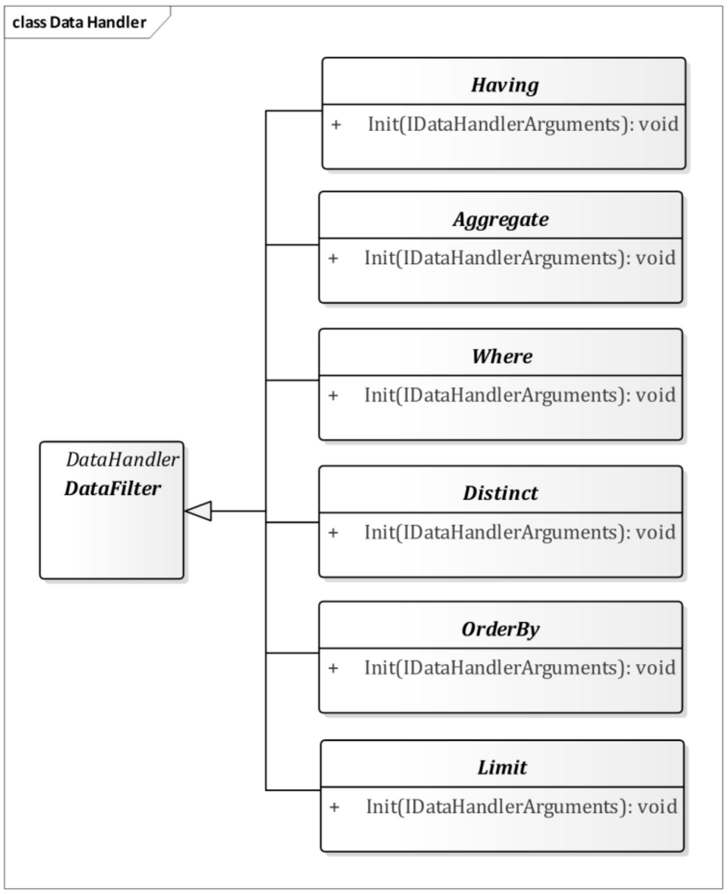

class Data Handler
Having
+ Init(IDataHandlerArguments): void
Aggregate
+ Init(IDataHandlerArguments): void
Where
+ Init(IDataHandlerArguments): void
Distinct
+ Init(IDataHandlerArguments): void
OrderBy
+ Init(IDataHandlerArguments): void
Limit
+ Init(IDataHandlerArguments): void
DataHandler
DataFilter

Figure D.6 Abstract syntax for Data Handler package – Data Filter

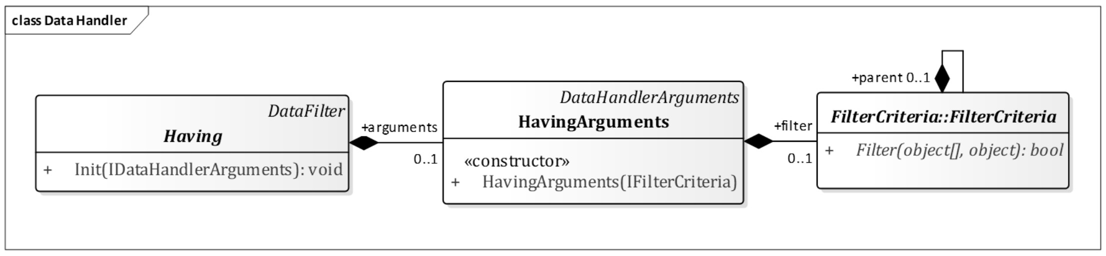

class Data Handler
DataFilter
Having
+ Init(IDataHandlerArguments): void
+arguments
0..1
DataHandlerArguments
HavingArguments
«constructor»
+ HavingArguments(IFilterCriteria)
+filter
0..1
+parent 0..1
FilterCriteria::FilterCriteria
+ Filter(object[], object): bool

Figure D.7 Abstract syntax for Data Handler package – Having

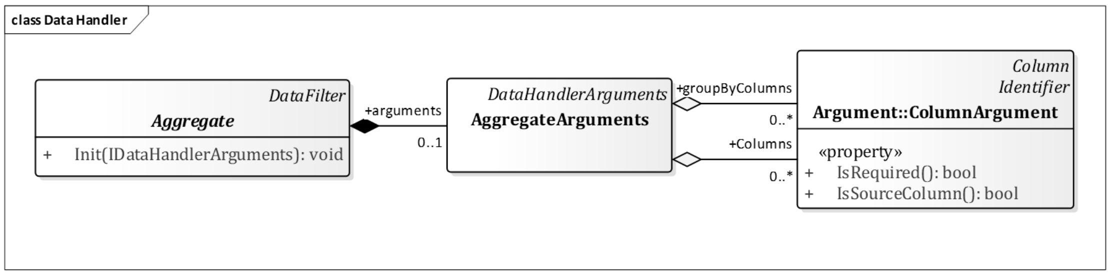


Figure D.8 Abstract syntax for Data Handler package – Having

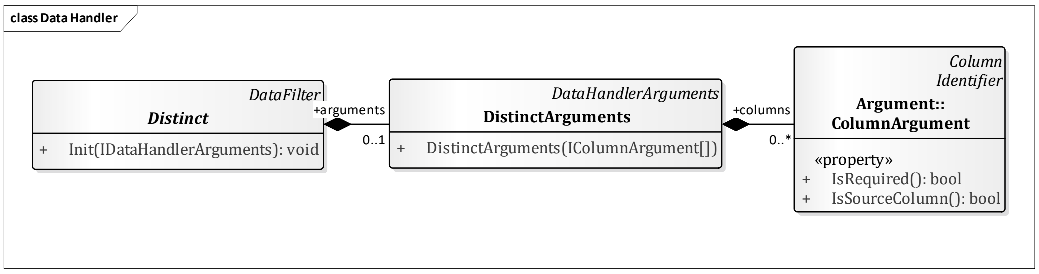


Figure D.9 Abstract syntax for Data Handler package – Distinct

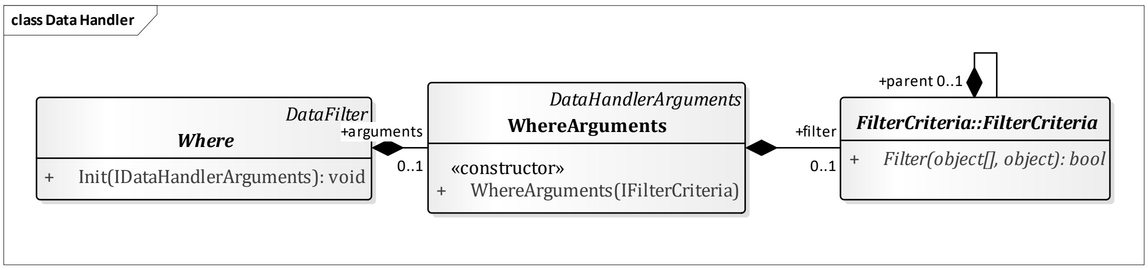

class Data Handler
DataFilter
Where
+ Init(IDataHandlerArguments): void
+arguments
0..1
DataHandlerArguments
WhereArguments
«constructor»
+ WhereArguments(IFilterCriteria)
+filter
0..1
+parent 0..1
FilterCriteria::FilterCriteria
+ Filter(object[], object): bool

Figure D.10 Abstract syntax for Data Handler package – Where

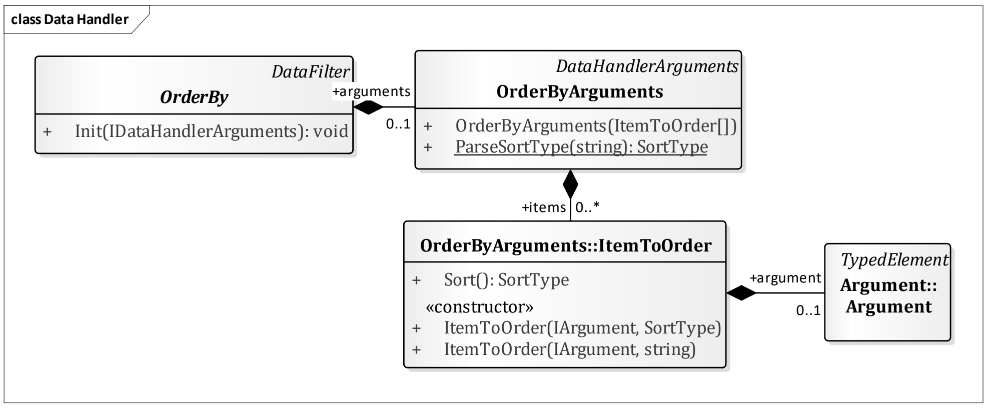

class Data Handler
DataFilter
OrderBy
+ Init(IDataHandlerArguments): void
+arguments
0..1
DataHandlerArguments
OrderByArguments
+ OrderByArguments(ItemToOrder[])
+ ParseSortType(string): SortType
+items 0..*
OrderByArguments::ItemToOrder
+ Sort(): SortType
«constructor»
+ ItemToOrder(IArgument, SortType)
+ ItemToOrder(IArgument, string)
+argument
0..1
TypedElement
Argument:: Argument

Figure D.11 Abstract syntax for Data Handler package – Order By

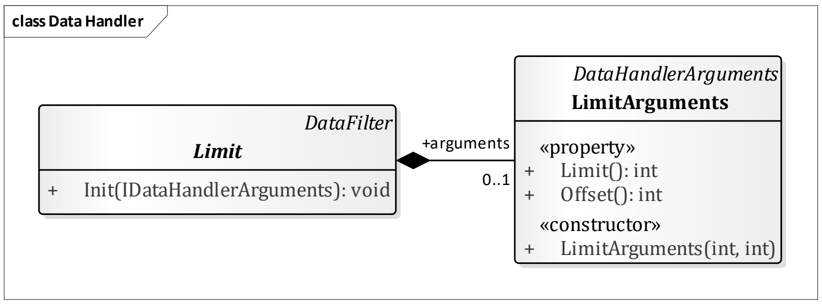


Figure D.12 Abstract syntax for Data Handler package – Limit and Offset

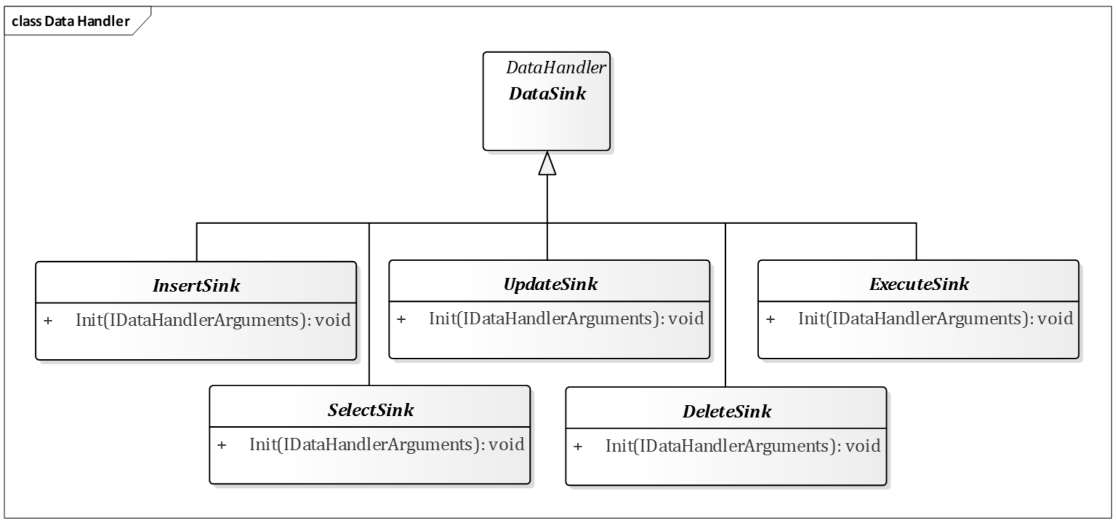

class Data Handler
DataHandler
DataSink
InsertSink
+ Init(IDataHandlerArguments): void
UpdateSink
+ Init(IDataHandlerArguments): void
ExecuteSink
+ Init(IDataHandlerArguments): void
SelectSink
+ Init(IDataHandlerArguments): void
DeleteSink
+ Init(IDataHandlerArguments): void

Figure D.13 Abstract syntax for Data Handler package – Data Sink

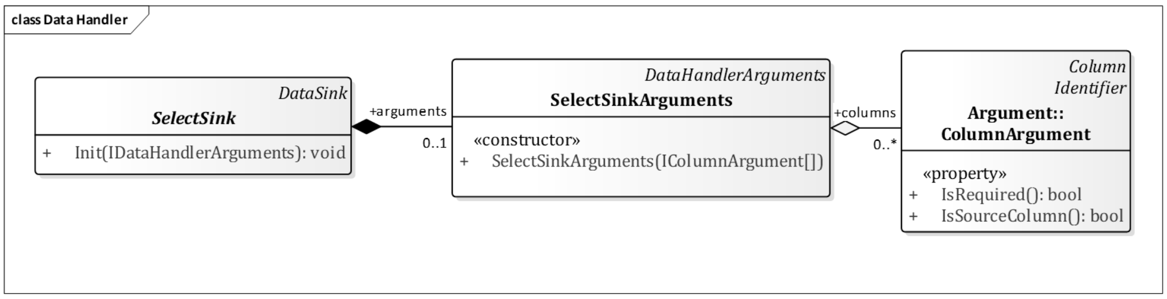


Figure D.14 Abstract syntax for Data Handler package – Select

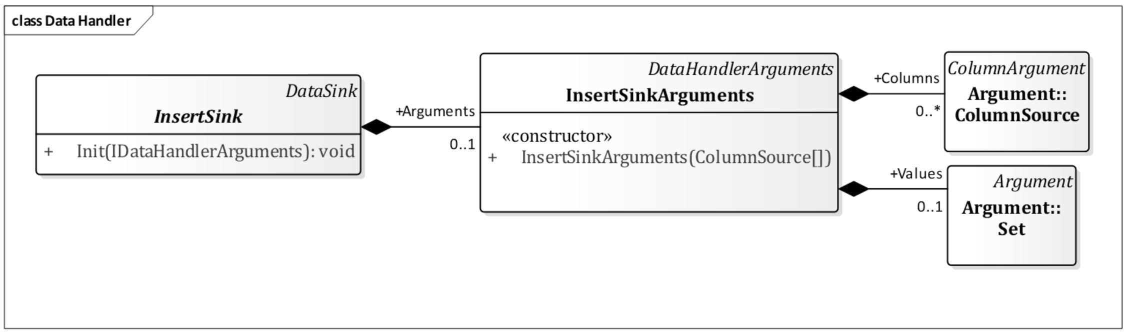


Figure D.15 Abstract syntax for Data Handler package – Insert

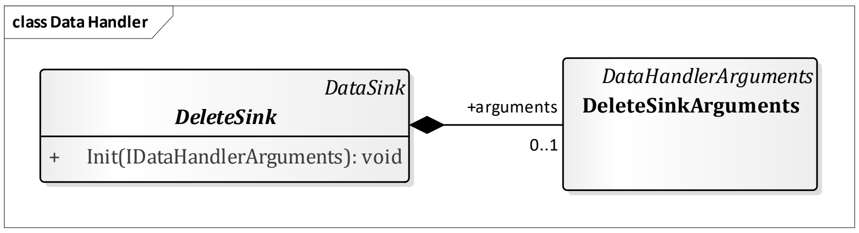


Figure D.16 Abstract syntax for Data Handler package – Delete

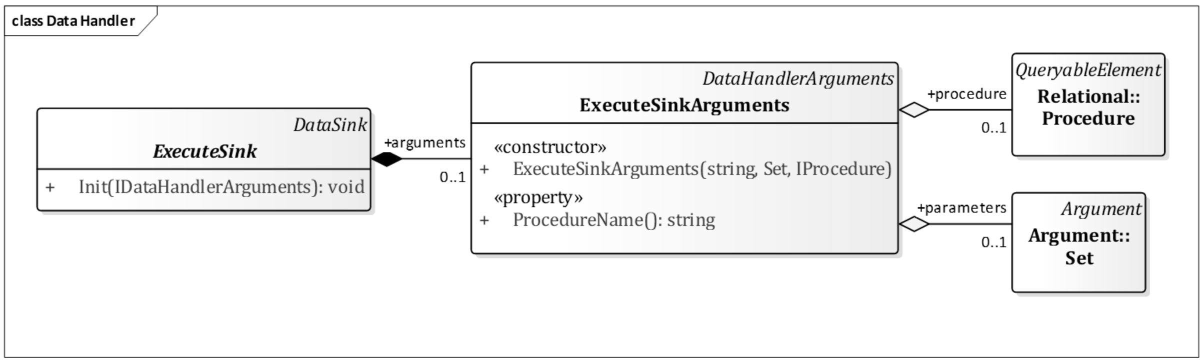


Figure D.17 Abstract syntax for Data Handler package – Execute (EXEC)

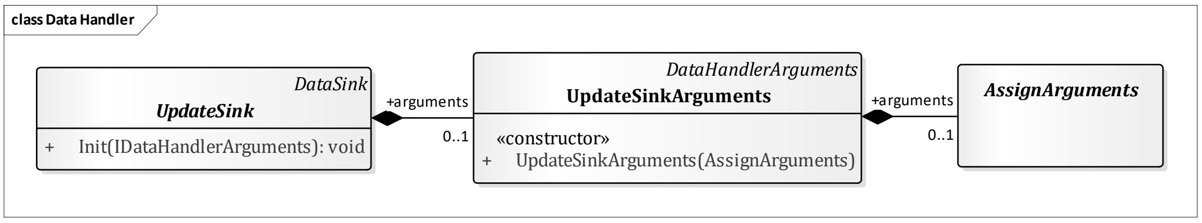


Figure D.18 Abstract syntax for Data Handler package – Update

D.2 Filter Criteria Package

This appendix shows the abstract syntax for Filter Criteria package.

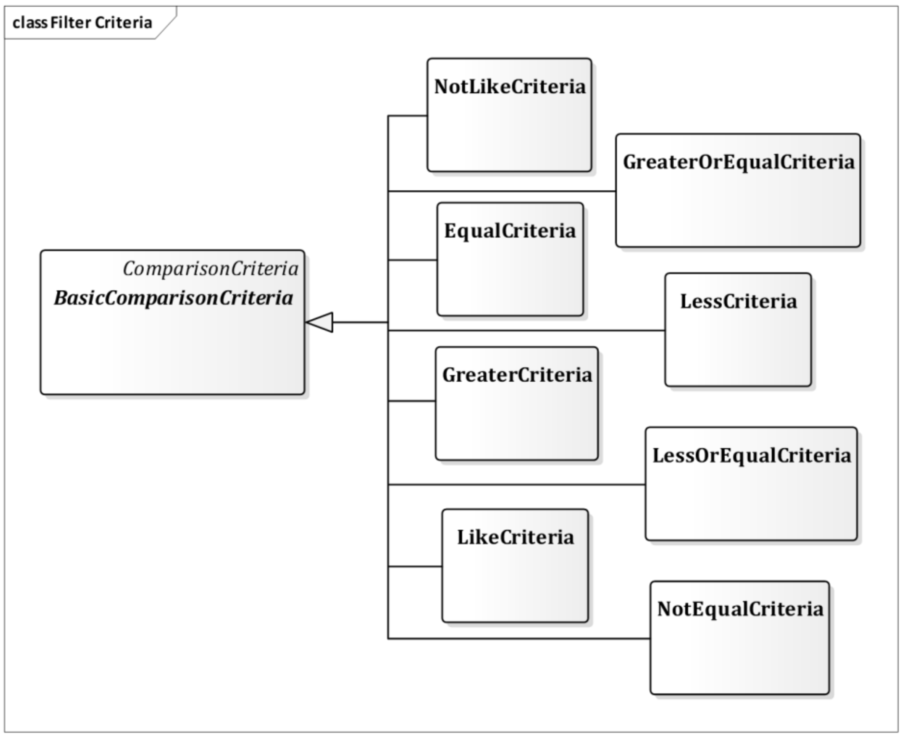


Figure D.19 Abstract syntax for Filter Criteria package – Basic Comparison Criteria

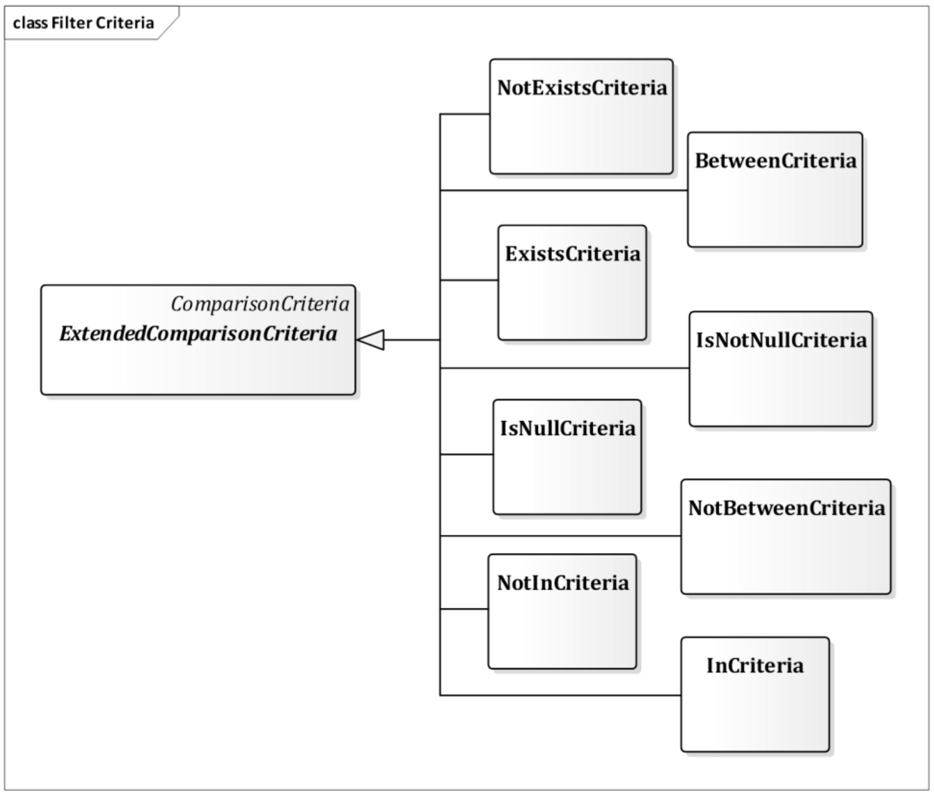


Figure D.20 Abstract syntax for Filter Criteria package – Extended Comparison Criteria

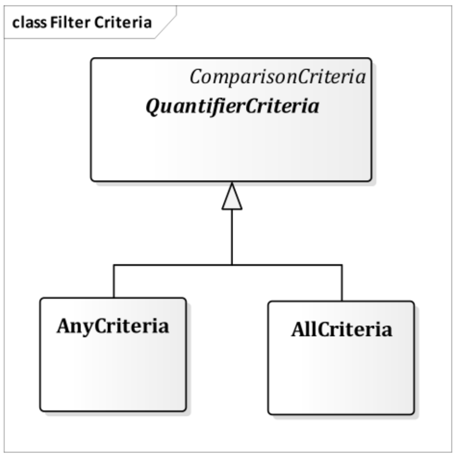


Figure D.21 Abstract syntax for Filter Criteria package – Quantifier Criteria

D.3 Connector Package

This appendix shows the abstract syntax for Connector package.

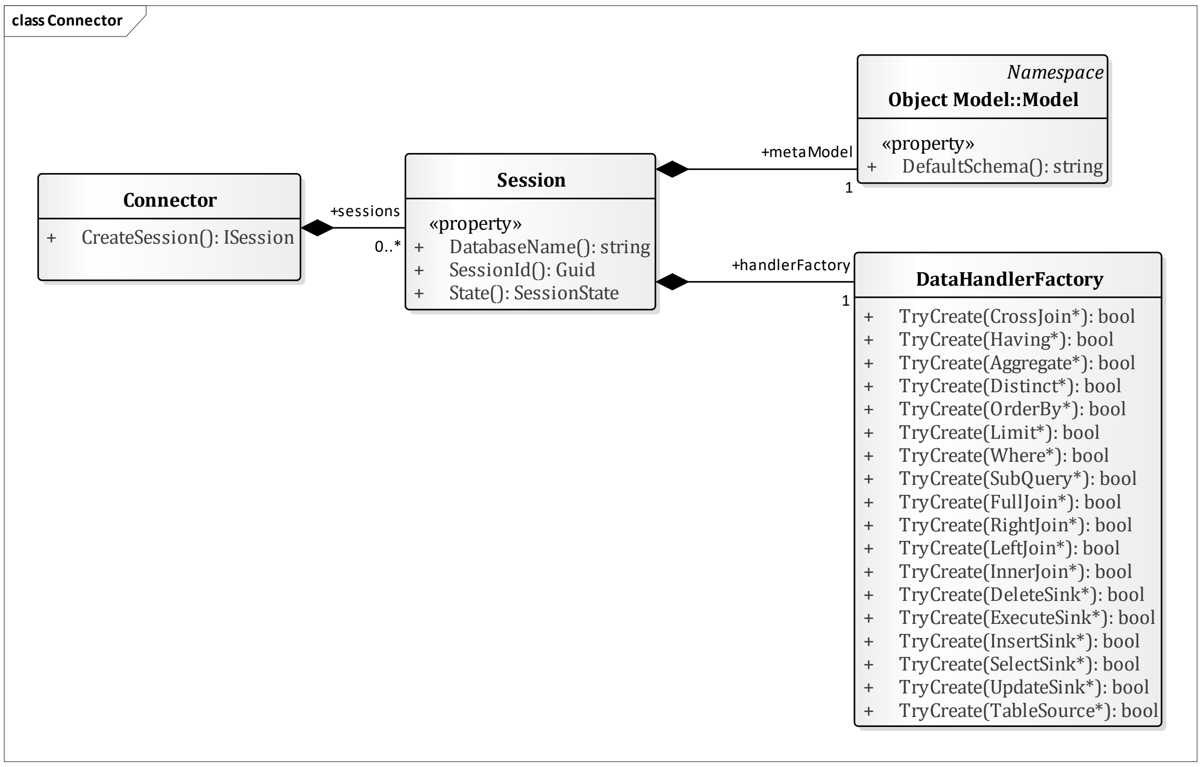
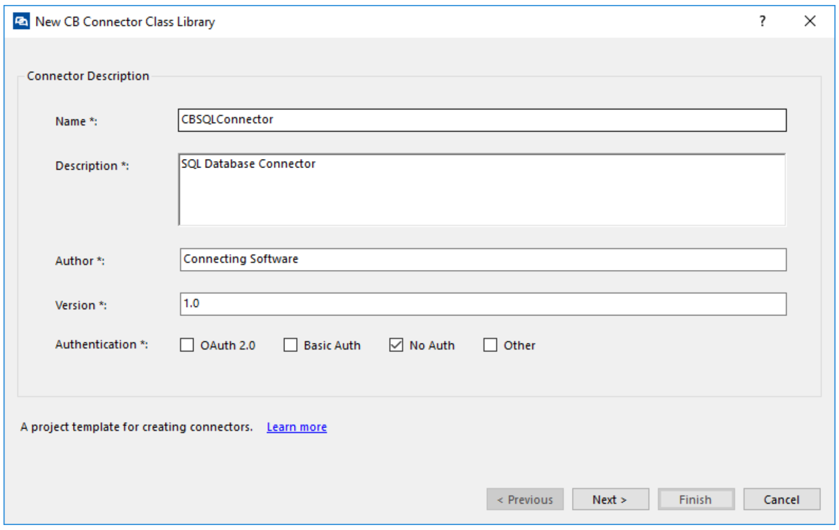

class Connector
Connector
+ CreateSession(): ISession
+sessions
0..*
Session
«property»
+ DatabaseName(): string
+ SessionId(): Guid
+ State(): SessionState
+metaModel
1
Namespace
Object Model::Model
«property»
+ DefaultSchema(): string
+handlerFactory
1
DataHandlerFactory
+ TryCreate(CrossJoin*): bool
+ TryCreate(Having*): bool
+ TryCreate(Aggregate*): bool
+ TryCreate(Distinct*): bool
+ TryCreate(OrderBy*): bool
+ TryCreate(Limit*): bool
+ TryCreate(Where*): bool
+ TryCreate(SubQuery*): bool
+ TryCreate(FullJoin*): bool
+ TryCreate(RightJoin*): bool
+ TryCreate(LeftJoin*): bool
+ TryCreate(InnerJoin*): bool
+ TryCreate(DeleteSink*): bool
+ TryCreate(ExecuteSink*): bool
+ TryCreate(InsertSink*): bool
+ TryCreate(SelectSink*): bool
+ TryCreate(UpdateSink*): bool
+ TryCreate(TableSource*): bool

Figure D.22 Abstract syntax for Connector package

APPENDIX E Tools

This appendix shows the final result of the tools presented in section 3.3.2.

E.1 VS Wizard with Project Template

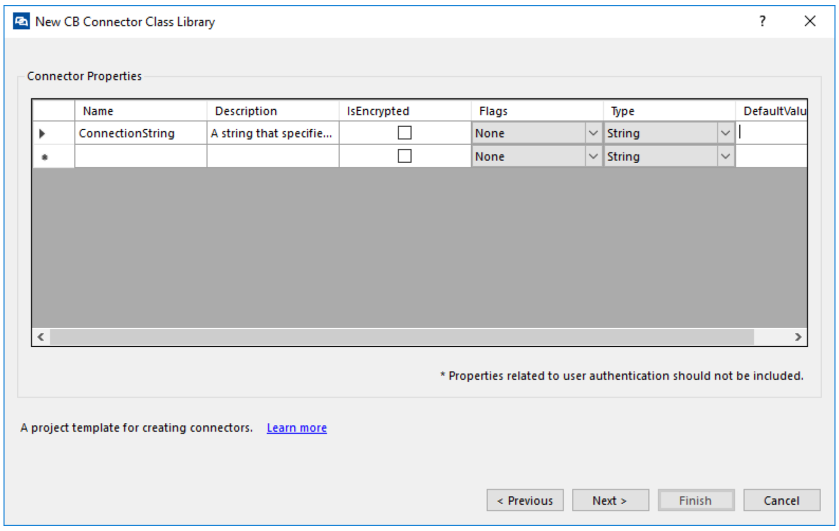

New CB Connector Class Library

Connector Description

Name *:	CBSQLConnector
Description *:	SQL Database Connector
Author *:	Connecting Software
Version *:	1.0
Authentication *:	☐ OAuth 2.0 ☐ Basic Auth ☑ No Auth ☐ Other

A project template for creating connectors. Learn more

Figure E.1 Connector Description


New CB Connector Class Library

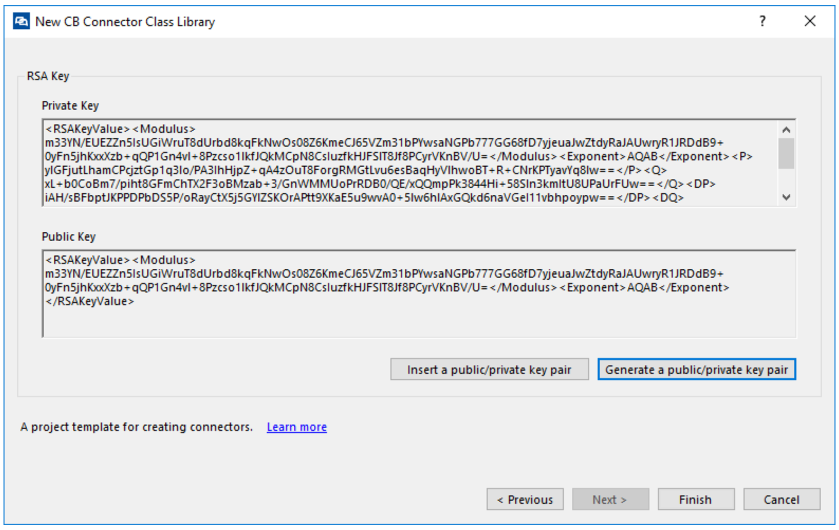
Connector Properties

	Name	Description	IsEncrypted	Flags	Type	DefaultValu
▶	ConnectionString	A string that specifie...	☐	None	String	
*			☐	None	String	

* Properties related to user authentication should not be included.

A project template for creating connectors. Learn more

Figure E.2 Connector Properties


New CB Connector Class Library

RSA Key

Private Key

```
<RSAKeyValue><Modulus>
m33YN/EUEZZn5IsUGiWruT8dUrbd8kqFkNwOs08Z6KmeCJ65VZm31bPYwsaNGPb777GG68fD7yjeuaJwZtdyRaJAUwryR1JRDdB9+
0yFn5jhKxxXzb+qQP1Gn4vl+8Pzcso1lkfJQkMCpN8CsIuzfkHJFSlT8Jf8PCyrVKnBV/U=</Modulus><Exponent>AQAB</Exponent><P>
ylGFjutLhamCPcjztGp1q3lo/PA3IhHjpZ+qA4zOuT8ForgRMGtLvu6esBaqHyVlhwoBT+R+CNrKPTyavYq8lw==</P><Q>
xL+b0CoBm7/piht8GFmChTX2F3oBMzab+3/GnWMMUoPrRDB0/QE/xQQmpPk3844Hi+58Sln3kmItU8UPaUrFUw==</Q><DP>
iAH/sBFbptJKPPDPbDS5P/oRayCtX5j5GYlZSKOrAPtt9XKaE5u9wvA0+5lw6hlAxGQkd6naVGeI11vbhpoypw==</DP><DQ>
```

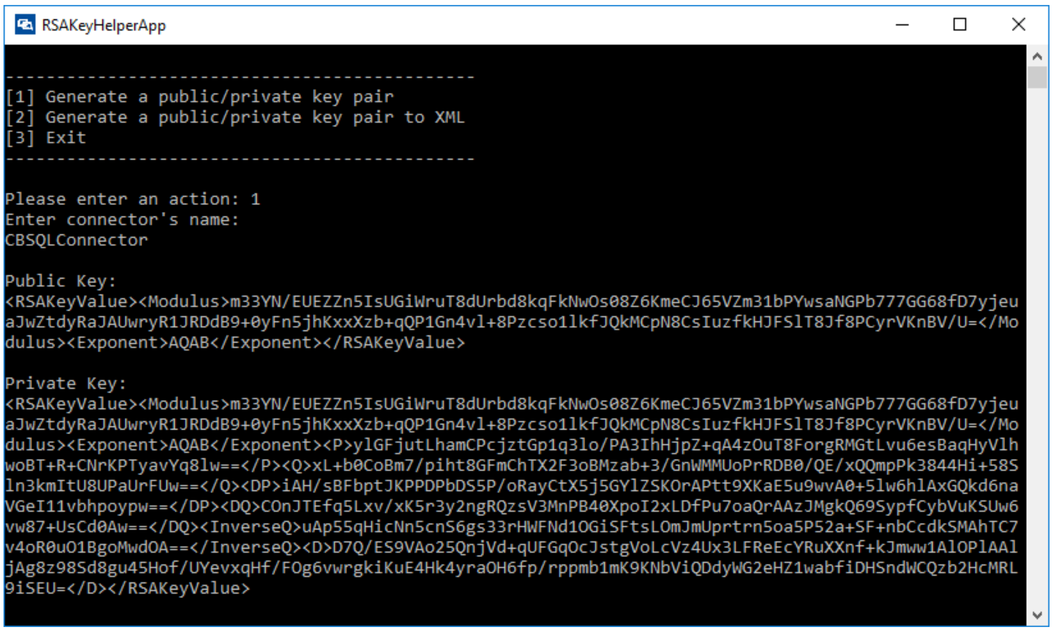
Public Key

```
<RSAKeyValue><Modulus>
m33YN/EUEZZn5IsUGiWruT8dUrbd8kqFkNwOs08Z6KmeCJ65VZm31bPYwsaNGPb777GG68fD7yjeuaJwZtdyRaJAUwryR1JRDdB9+
0yFn5jhKxxXzb+qQP1Gn4vl+8Pzcso1lkfJQkMCpN8CsIuzfkHJFSlT8Jf8PCyrVKnBV/U=</Modulus><Exponent>AQAB</Exponent>
</RSAKeyValue>
```

Insert a public/private key pair
Generate a public/private key pair
A project template for creating connectors. Learn more
< Previous
Next >
Finish
Cancel

Figure E.3 Public/Private RSA key pair

E.2 RSA Key Generator


RSAKeyHelperApp

```
-----------------------------------------------
[1] Generate a public/private key pair
[2] Generate a public/private key pair to XML
[3] Exit
-----------------------------------------------

Please enter an action: 1
Enter connector's name:
CBSQLConnector

Public Key:
<RSAKeyValue><Modulus>m33YN/EUEZZn5IsUGiWruT8dUrbd8kqFkNwOs08Z6KmeCJ65VZm31bPYwsaNGPb777GG68fD7yjeu
aJwZtdyRaJAUwryR1JRDdB9+0yFn5jhKxxXzb+qQP1Gn4vl+8Pzcso1lkfJQkMCpN8CsIuzfkHJFSlT8Jf8PCyrVKnBV/U=</Mo
dulus><Exponent>AQAB</Exponent></RSAKeyValue>

Private Key:
<RSAKeyValue><Modulus>m33YN/EUEZZn5IsUGiWruT8dUrbd8kqFkNwOs08Z6KmeCJ65VZm31bPYwsaNGPb777GG68fD7yjeu
aJwZtdyRaJAUwryR1JRDdB9+0yFn5jhKxxXzb+qQP1Gn4vl+8Pzcso1lkfJQkMCpN8CsIuzfkHJFSlT8Jf8PCyrVKnBV/U=</Mo
dulus><Exponent>AQAB</Exponent><P>ylGFjutLhamCPcjztGp1q3lo/PA3IhHjpZ+qA4zOuT8ForgRMGtLvu6esBaqHyVlh
woBT+R+CNrKPTyavYq8lw==</P><Q>xL+b0CoBm7/piht8GFmChTX2F3oBMzab+3/GnWMMUoPrRDB0/QE/xQQmpPk3844Hi+58S
ln3kmItU8UPaUrFUw==</Q><DP>iAH/sBFbptJKPPDPbDS5P/oRayCtX5j5GYlZSKOrAPtt9XKaE5u9wvA0+5lw6hlAxGQkd6na
VGeI11vbhpoypw==</DP><DQ>COnJTEfq5Lxv/xK5r3y2ngRQzsV3MnPB40XpoI2xLDfPu7oaQrAAzJMgkQ69SypfCybVuKSUw6
vw87+UsCd0Aw==</DQ><InverseQ>uAp55qHicNn5cnS6gs33rHWFNd1OGiSFtsLOmJmUprtrn5oa5P52a+SF+nbCcdkSMAhTC7
v4oR0uO1BgoMwdOA==</InverseQ><D>D7Q/ES9VAo25QnjVd+qUFGqOcJstgVoLcVz4Ux3LFReEcYRuXXnf+kJmww1AlOPlAAl
jAg8z98Sd8gu45Hof/UYevxqHf/FOg6vwrgkiKuE4Hk4yraOH6fp/rppmb1mK9KNbViQDdyWG2eHZ1wabfiDHSndWCQzb2HcMRL
9iSEU=</D></RSAKeyValue>
```

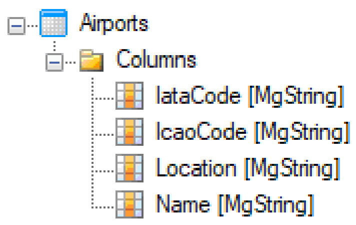
Figure E.4 RSA Key Generator

APPENDIX F
Open Data Protocol

This appendix shows how metamodel concepts are defined following the implementation-level rules presented in section 4.2.2.

For instance, for the following XML fragment, the table definition is given below (by applying **Rule1**).

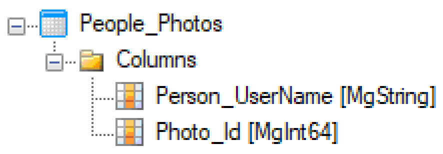
```
<EntityType Name="Airport"><Key><PropertyRef Name="IcaoCode"/></Key>
<Property Name="IcaoCode" Type="Edm.String" Nullable="false"/>
<Property Name="Name" Type="Edm.String" Nullable="false"/>
<Property Name="IataCode" Type="Edm.String" Nullable="false"/>
<Property Name="Location" Type="AirportLocation" Nullable="false"/>
</EntityType>
<EntitySet Name="Airports" EntityType="Airport"/>
```



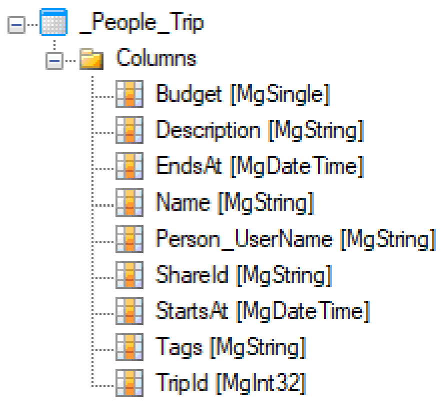
```
Airports
  Columns
    IataCode [MgString]
    IcaoCode [MgString]
    Location [MgString]
    Name [MgString]
```

For instance, for the following XML fragment, two table definitions are given below (by applying **Rule2** and **Rule3**, respectively).

```
<EntityType Name="Person" OpenType="true"> <Key> <PropertyRef Name="UserName"/> </Key>
  <Property Name="UserName" Type="Edm.String" Nullable="false">...</Property>
  <Property Name="FirstName" Type="Edm.String" Nullable="false"/>
  <Property Name="LastName" Type="Edm.String" Nullable="false"/>
  <Property Name="Emails" Type="Collection(Edm.String)"/>
  <Property Name="AddressInfo" Type="Collection(Location)"/>
  <Property Name="Gender" Type="PersonGender"/>
  <Property Name="Concurrency" Type="Edm.Int64" Nullable="false">...</Property>
  <NavigationProperty Name="Friends" Type="Collection(Person)"/>
  <NavigationProperty Name="Trips" Type="Collection(Trip)" ContainsTarget="true"/>
  <NavigationProperty Name="Photo" Type="Photo"/>
</EntityType>
<EntitySet Name="People" EntityType="Person">
  <NavigationPropertyBinding Path="Friends" Target="People"/>
  <NavigationPropertyBinding Path="Photo" Target="Photos"/>
</EntitySet>
```



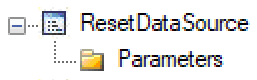
```
People_Photos
  Columns
    Person_UserName [MgString]
    Photo_Id [MgInt64]
```



```
_People_Trip
  Columns
    Budget [MgSingle]
    Description [MgString]
    EndsAt [MgDateTime]
    Name [MgString]
    Person_UserName [MgString]
    ShareId [MgString]
    StartsAt [MgDateTime]
    Tags [MgString]
    TripId [MgInt32]
```

For instance, for the following XML fragment, the procedure definition is given below (by applying **Rule1**).

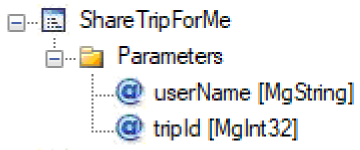
```
<Action Name="ResetDataSource"/>
```



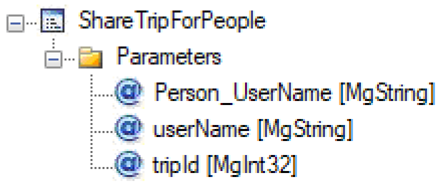
```
ResetDataSource
  Parameters
```

For instance, for the following XML fragment, two procedure definitions are given below (by applying **Rule2**).

```
<Action Name="ShareTrip" IsBound="true">
  <Parameter Name="person" Type="Person" Nullable="false"/>
  <Parameter Name="userName" Type="Edm.String" Nullable="false"/>
  <Parameter Name="tripId" Type="Edm.Int32" Nullable="false"/>
</Action>
```



```
ShareTripForMe
  Parameters
    userName [MgString]
    tripId [MgInt32]
```



```
ShareTripForPeople
  Parameters
    Person_UserName [MgString]
    userName [MgString]
    tripId [MgInt32]
```

APPENDIX G
Schema Collections

This appendix describes the schema collections for **SQL Server** from Microsoft [66].

Table G.1 Schema collections for MS SQL Server

MS SQL Server		
	Name	**Description**
Databases	DATABASE_NAME	Name of the database.
	DBID	Database Identifier.
	CREATE_DATE	Creation data of the database.
Foreign Keys	CONSTRAINT_CATALOG	*Catalog* the constraint belongs to.
	CONSTRAINT_SCHEMA	Schema that contains the constraint.
	CONSTRAINT_NAME	Constraint Name.
	TABLE_CATALOG	Table Name constraint is part of.
	TABLE_SCHEMA	Schema that contains the table.
	TABLE_NAME	Table Name.
	CONSTRAINT_TYPE	Type of constraint.
	IS_DEFERRABLE	Specifies whether the constraint is deferrable.
	INITIALLY_DEFERRED	Specifies whether the constraint is initially deferrable.

MS SQL Server		
	Name	**Description**
Indexes	CONSTRAINT_CATALOG	*Catalog* that index belongs to.
	CONSTRAINT_SCHEMA	Schema that contains the index.
	CONSTRAINT_NAME	Name of the index.
	TABLE_CATALOG	Table name the index is associated with.
	TABLE_SCHEMA	Schema that contains the table the index is associated with.
	TABLE_NAME	Table Name.
	INDEX_NAME	Index Name.
	TYPE_DESC	The type of the index.
IndexColumns	CONSTRAINT_CATALOG	*Catalog* that index belongs to.
	CONSTRAINT_SCHEMA	Schema that contains the index.
	CONSTRAINT_NAME	Name of the index.
	TABLE_CATALOG	Table name the index is associated with.
	TABLE_SCHEMA	Schema that contains the table the index is associated with.
	TABLE_NAME	Table Name.
	COLUMN_NAME	Column name the index is associated with.
	ORDINAL_POSITION	Column ordinal position.
	KEYTYPE	The type of object.
	INDEX_NAME	Index Name.

MS SQL Server		
	Name	**Description**
Procedures	SPECIFIC_CATALOG	Specific name for the *catalog*.
	SPECIFIC_SCHEMA	Specific name for the schema.
	SPECIFIC_NAME	Specific name for the *catalog*.
	ROUTINE_CATALOG	*Catalog* that stored procedure belongs to.
	ROUTINE_SCHEMA	Schema that contains the stored procedure.
	ROUTINE_NAME	Name of the stored procedure.
	ROUTINE_TYPE	The type of procedure: PROCEDURE or FUNCTION.
	CREATED	Time the procedure was created.
	LAST_ALTERED	The last time the procedure was modified.
Procedure Parameters	SPECIFIC_CATALOG	*Catalog* name of the procedure for which this is a parameter.
	SPECIFIC_SCHEMA	Schema that contains the procedure for which this parameter is part of.
	SPECIFIC_NAME	Name of the procedure for which this parameter is a part of.
	ORDINAL_POSITION	Ordinal position of the parameter starting at 1.

MS SQL Server		
	Name	**Description**
	PARAMETER_MODE	Returns IN if an input parameter, OUT if an output parameter, and INOUT if an input/output parameter.
	IS_RESULT	Indicates if result of the procedure is a function.
	AS_LOCATOR	Indicates if parameter was declared as locator.
	PARAMETER_NAME	Name of the parameter.
	DATA_TYPE	System-supplied data type.
	CHARACTER_MAXIMUM_LENGTH	Maximum length, in characters, for binary or character data types.
	CHARACTER_OCTET_LENGTH	Maximum length, in bytes, for binary or character data types.
	COLLATION_CATALOG	*Catalog* name of the collation of the parameter.
	COLLATION_SCHEMA	Always returns NULL.
	COLLATION_NAME	Name of the collation of the parameter.
	CHARACTER_SET_CATALOG	*Catalog* name of the character set of the parameter.
	CHARACTER_SET_SCHEMA	Always returns NULL.
	CHARACTER_SET_NAME	Name of the character set of the parameter.
	NUMERIC_PRECISION	Precision of approximate numeric data, exact

MS SQL Server		
	Name	**Description**
		numeric data, integer data, or monetary data.
	NUMERIC_PRECISION_RADIX	Precision radix of approximate numeric data, exact numeric data, integer data, or monetary data.
	NUMERIC_SCALE	Scale of approximate numeric data, exact numeric data, integer data, or monetary data.
	DATETIME_PRECISION	Precision in fractional seconds if the parameter type is datetime or small-datetime.
Tables	TABLE_CATALOG	*Catalog* of the table.
	TABLE_SCHEMA	Schema that contains the table.
	TABLE_NAME	Table Name.
	TABLE_TYPE	Type of table.
Columns	TABLE_CATALOG	*Catalog* of the table.
	TABLE_SCHEMA	Schema that contains the table.
	TABLE_NAME	Table Name.
	COLUMN_NAME	Column Name.
	ORDINAL_POSITION	Column identification number.
	COLUMN_DEFAULT	Default value of the column.
	IS_NULLABLE	Nullability of the column.
	DATA_TYPE	System-supplied data type.

MS SQL Server		
	Name	**Description**
	CHARACTER_MAXIMUM_LENGTH	Maximum length, in characters, for binary data, character data, or text and image data.
	CHARACTER_OCTET_LENGTH	Maximum length, in bytes, for binary data, character data, or text and image data.
	COLLATION_CATALOG	Indicates the database in which the *collation* is located – *master* (if column is character data or text data type).
	CHARACTER_SET_CATALOG	Indicates the database in which the *character set* is located – *master* (if column is character data or text data type).
	CHARACTER_SET_SCHEMA	Always returns NULL.
	CHARACTER_SET_NAME	If this column is character data or text data type, Returns the unique name for the character set.
	NUMERIC_PRECISION	Precision of approximate numeric data, exact numeric data, integer data, or monetary data.
	NUMERIC_PRECISION_RADIX	Precision radix of approximate numeric data,

MS SQL Server		
	Name	**Description**
		exact numeric data, integer data, or monetary data.
	NUMERIC_SCALE	Scale of approximate numeric data, exact numeric data, integer data, or monetary data.
	DATETIME_PRECISION	Subtype code for datetime.
	IS_FILESTREAM	Indicates if the column has FILESTREAM attribute.
	IS_SPARSE	Indicates if the column is a sparse column.
	IS_COLUMN_SET	Indicates if the column is a column set column.
Users	UID	Unique identifier in the database.
	USER_NAME	Unique username or group name in the database.
	CREATEDATE	Date the account was added.
	UPDATEDATE	Date the account was last changed.
Views	TABLE_CATALOG	*Catalog* of the view.
	TABLE_SCHEMA	Schema that contains the view.
	TABLE_NAME	View name.
	CHECK_OPTION	Type of WITH CHECK OPTION
	IS_UPDATABLE	Indicates if the view is updatable.
ViewColumns	VIEW_CATALOG	*Catalog* of the view.

MS SQL Server		
	Name	**Description**
	VIEW_SCHEMA	Schema that contains the view.
	VIEW_NAME	View name.
	TABLE_CATALOG	*Catalog* of the table that is associated with this view.
	TABLE_SCHEMA	Schema that contains the table that is associated with this view.
	TABLE_NAME	Name of the table that is associated with the view.
	COLUMN_NAME	Column name.
UserDefinedTypes (UDT)	ASSEMBLY_NAME	The name of the file for the assembly.
	UDT_NAME	The class name for the assembly.
	VERSION_MAJOR	Major Version Number.
	VERSION_MINOR	Minor Version Number.
	VERSION_BUILD	Build Version Number.
	VERSION_REVISION	Revision Version Number.
	CULTURE_INFO	The culture information associated with the UDT.
	PUBLIC_KEY	The public key used by the assembly.
	IS_FIXED_LENGTH	Indicate if length of type is always same as *MAX_LENGTH*.
	MAX_LENGTH	Maximum length of type in bytes.
	CREATE_DATE	The date the assembly was created/registered.

MS SQL Server		
	Name	**Description**
	PERMISSION_SET_DESC	The friendly name for the permission-set/security-level for the assembly.