FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Design and prototype implementation of data flow analysis of jobs in Matillion ETL for the Manta project
Student:	Illia Krauchenia
Supervisor:	Ing. Michal Valenta, Ph.D.
Study program:	Informatics
Branch / specialization:	Web and Software Engineering, specialization Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2022/2023

Instructions

The aim of this work is to design and implement a prototype of a functional module that performs syntactic and semantic analysis of tasks in the cloud tool Matillion ETL, and then uses its result for data flow analysis and generation of a graph that represents data flows. The design and implementation will ensure a seamless connection of the module to the Manta project.
Follow these steps:

1. Learn about the Manta project and Matillion ETL.
2. Design a module in the Manta project for Matillion ETL task processing. Use the existing project infrastructure.
3. Implement the prototype, properly document it, and test it.

Electronically approved by Ing. Michal Valenta, Ph.D. on 2 May 2022 in Prague.

Bachelor's thesis

DESIGN AND PROTOTYPE IMPLEMENTATION OF DATA FLOW ANALYSIS OF JOBS IN MATILLION ETL FOR THE MANTA PROJECT

Illia Krauchenia

Faculty of Information Technology
Department of Software Engineering
Supervisor: Ing. Michal Valenta, Ph.D.
December 17, 2022

Citation of this thesis: Krauchenia Illia. *Design and Prototype Implementation of Data Flow Analysis of Jobs in Matillion ETL for the Manta Project.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Contents

Acknowledgments **vi**

Declaration **vii**

Abstract **viii**

List of abbreviations **ix**

1 Introduction **1**
- 1.1 Motivation 1
- 1.2 Thesis goals 1
- 1.3 Thesis structure 2

2 Basic concepts and technologies **3**
- 2.1 Data lineage 3
- 2.2 Manta 3
  - 2.2.1 Manta Flow 4
  - 2.2.2 Data flow graph 4
- 2.3 Data warehouse 7
  - 2.3.1 Snowflake 7
- 2.4 ETL and ELT 7
  - 2.4.1 Matillion ETL 8
- 2.5 SQL 9
- 2.6 Metadata 9
- 2.7 Java 9
  - 2.7.1 Javadoc 10
- 2.8 IDE 10
- 2.9 Apache Maven 10
- 2.10 Spring 10
  - 2.10.1 Dependency injection 10
- 2.11 JUnit 11
- 2.12 JSON 11
  - 2.12.1 Jackson Streaming API 11
  - 2.12.2 ObjectMapper 11
- 2.13 Git 11
- 2.14 Jenkins 12
- 2.15 SonarQube 12

3 Analysis **13**
- 3.1 Matillion ETL 13
  - 3.1.1 Key elements 13
  - 3.1.2 Metadata export 16
- 3.2 Matillion scanner requirements 19
  - 3.2.1 Functional requirements 19

3.2.2 Non-functional requirements 20

4 Matillion scanner design **21**
4.1 Connector module 21
4.1.1 Manual input directory structure 22
4.1.2 Manual input files splitting 22
4.1.3 Module structure 23
4.2 Data Flow Generator module 26
4.2.1 Data flow graph nodes structure 26
4.2.2 Data flow analysis and graph generation 27

5 Matillion scanner prototype implementation **29**
5.1 Connector module 29
5.2 Data Flow Generator module 33

6 Matillion scanner prototype testing **37**
6.1 Connector module 37
6.2 Data Flow Generator module 39
6.3 Data flow visualization 39

7 Conclusion **41**

A More Matillion ETL data flow visualization examples **43**

Contents of enclosed media **51**

List of Figures

2.1 Manta Flow architecture [6] . . . 5
2.2 Data flow graph example in Manta Flow [8] . . . 6
2.3 ELT process [14] . . . 8

3.1 Matillion ETL user interface . . . 14
3.2 Matillion ETL export menu . . . 17
3.3 Matillion ETL manual export file root structure . . . 18
3.4 Matillion ETL manual export file component property structure . . . 18

4.1 Dependency diagram of the Matillion scanner modules . . . 21
4.2 Manual input directory structure . . . 22
4.3 Structure of the temporary directory for split manual input files . . . 22
4.4 Class diagram in the `manta-connector-matillion-model` submodule . . . 25
4.5 Data flow graph nodes structure . . . 26
4.6 Sequence diagram for the Data Flow Generator module . . . 27

6.1 Result data flow graph example . . . 39

A.1 Matillion ETL data flow visualization example No. 1 . . . 44
A.2 Matillion ETL data flow visualization example No. 2 . . . 45
A.3 Matillion ETL data flow visualization example No. 3 . . . 46

List of code listings

1 The `extract` method in the `ManualInputExtractor` class . . . 30
2 The `readEnvironmentContent` method in the `AbstractJsonSplitter` class . . . 31
3 The `writeToEnvironmentFile` method in the `OutputFileWriter` class . . . 31
4 The `read` method in the `JsonJobReader` class . . . 32
5 The `buildJobNode` method in the `MatillionGraphHelper` class . . . 33
6 The `analyze` method in the `ConvertTypeComponentAnalyzer` class . . . 34
7 The `resolveAllScalarVariables` method in the `VariablesResolver` class . . . 35
8 The `createSnowflakePlatformConnection` method in the `DatabaseConnector` class . . . 35
9 The `extractTest` method in the `ManualInputExtractorTest` class . . . 38
10 The `jsonSuccessTest` method in the `ManualInputFileHolderTest` class . . . 38
11 The `compareExpectedGraphToActual` method in the `DataflowEqualityTest` class 39

Here I would like to express gratitude to Ing. Michal Valenta, Ph.D. for great assistance and Ing. Petr Košvanec for valuable advice and guidance during the writing of this thesis and work at Manta. I am also incredibly grateful to my family, friends, and all the other people who have supported me in any way throughout my studies.

Declaration

In Prague on December 17, 2022 ....

Abstract

This work aims to design and implement a prototype of a functional module that performs analysis of data flows in the cloud tool Matillion ETL. The input for the analysis is the metadata describing the tool elements. Based on the analysis, the implemented prototype generates a graph that visualizes data flows. The design and implementation ensure a seamless connection of the prototype to the data lineage platform called Manta. The first part of the work represents a general analysis of Matillion ETL, its key elements, ways to export metadata, as well as the analysis of its format and structure. The work then continues with the design and implementation parts of the functional module prototype and ends with the evaluation and testing of its correct functionality. Examples of the resulting data flow graphs generated by the prototype can be found in the appendix.

Keywords module prototype, data lineage, data flow analysis and visualization, metadata, Matillion ETL, data warehouse, Manta, Java

Abstrakt

Tato práce si klade za cíl navrhnout a implementovat prototyp funkčního modulu, který provádí analýzu datových toků v cloudovém nástroji Matillion ETL. Vstupem pro analýzu jsou metadata, popisující elementy nástroje. Na základě analýzy implementovaný prototyp generuje graf, který vizualizuje datové toky. Návrh a implementace zajišťují bezproblémové připojení prototypu k platformě data lineage zvané Manta. První část práce představuje obecnou analýzu Matillion ETL, jeho klíčových prvků, způsobů exportu metadat, a také analýzu jejich formátu a struktury. Práce pak pokračuje částmi návrhu a implementace prototypu funkčního modulu a končí vyhodnocením a testováním jeho správné funkčnosti. Ukázky výsledných grafů datových toků vygenerovaných prototypem lze nalézt v příloze.

Klíčová slova prototyp modulu, datová linie, analýza a vizualizace datových toků, metadata, Matillion ETL, datový sklad, Manta, Java

List of abbreviations

ETL	Extract, Transform, Load
ELT	Extract, Load, Transform
UI	User Interface
SQL	Structured Query Language
JSON	JavaScript Object Notation
API	Application Programming Interface

Chapter 1

Introduction

This chapter introduces the motivation that influenced the choice of the topic of this thesis by its author, as well as the goals and chapter structure of this work.

1.1 Motivation

"*The world of data is almost unrecognizable even from as little as five years ago. In reality, most organizations today understand the value of storing and managing their data to optimize their performance and to remain competitive in their market space. We all recognize that better information leads to better decisions, and an effective data solution, along with a new data "culture" makes this possible. Most businesses have no shortage of data, but organizing that data for easy access and new insights is a challenge that requires more than just data storage in a warehouse.*" [1]

So what helps companies organize, manage, and analyze their data effectively nowadays? The answer is quite simple: there are various tools for this, such as Matillion ETL. This tool helps to gather data coming from different sources, transform it into the desired format, and put it in target repositories, making it ready for further use in business intelligence.

Nevertheless, often when working with Matillion ETL, as well as with other similar tools, the end user is also interested in information about the origin of data and the history of its transformations on the way to the target storage. The solution to this issue is provided by the Manta software, which is able to analyze data flows in various technologies working with data, including ETL tools, and construct a graph representing these flows.

Currently, Manta does not support the Matillion ETL technology, which is why it was chosen as the topic for this thesis. Another reason that influenced the choice of this topic by the author is the opportunity to work on the module, studying the Manta infrastructure and gaining real work experience.

1.2 Thesis goals

The main goal of this thesis is to design and implement a prototype of a functional module for the Manta project, which performs analysis of data flows in the Matillion ETL tool. Based on the analysis, the prototype must be able to generate a graph that visualizes data flows from their source locations to the destinations. The subgoals of the thesis are qualitative testing of the prototype and documentation of its source code. The goals of individual chapters of the thesis are presented in the next section.

1.3 Thesis structure

Starting with the next, the thesis is divided into the following chapters:

2. **“Basic concepts and technologies”** – describes the main concepts and technologies used within this work.
3. **“Analysis”** – serves to present a general analysis of Matillion ETL, its key elements, ways to export metadata, as well as the analysis of its format and structure.
4. **“Matillion scanner design”** – introduces the design of the module prototype. All the key algorithms and mechanisms can be found in this chapter.
5. **“Matillion scanner prototype implementation”** – based on the previous chapter, presents the implementation of the prototype, mainly focusing on the essential parts.
6. **“Matillion scanner prototype testing”** – shows the way of testing the implemented solution.
7. **“Conclusion”** – serves to sum up the results of the work and evaluate the achievement of the set goals.

In the appendix **“More Matillion ETL data flow visualization examples”** the reader can find examples of data flow graphs, generated by the implemented prototype and not included in the main part of the work.

Chapter 2

Basic concepts and technologies

This chapter is intended to describe all the essential concepts necessary to understand the analytical and practical parts of the work, as well as the technologies used to implement the prototype. The chapter begins with a description of the concept of data lineage and introduces the Manta software and its most significant concepts. Then the chapter proceeds to describe the concept of a data warehouse, also mentioning Snowflake, and ETL, ELT tools, and focuses on one of them, Matillion ETL, which the implemented prototype will be closely related to. The chapter also contains descriptions of the SQL language, the concept of metadata, the Java programming language and its Spring framework, an overview of the JSON data format and tools for processing it, and other valuable technologies used within this work.

2.1 Data lineage

"*Data lineage uncovers the life cycle of data – it aims to show the complete data flow*[1]*, from start to finish. Data lineage is the process of understanding, recording, and visualizing data as it flows from data sources to consumption. This includes all transformations the data underwent along the way – how the data was transformed, what changed, and why.*" [3]

Some of the benefits of the data lineage process for various companies or other users may be, for example, the ability to track, analyze, and control the processes of the data they work with. This allows them to make changes to their systems, migrate them, and predict the consequences with more confidence and less risk. This also facilitates the process of finding and correcting possible errors in systems, which, for example, may be associated with data breaches.

One of the tools that provide data lineage visualization is **Manta**.

2.2 Manta

Manta is a lineage platform allowing information users and system administrators to visualize the entire flow of data through corporate information systems – where it flows, what happens to it, and why. Based on the obtained metadata[2], the tool can visualize individual data flows. [5]

Manta currently supports a large number of connections, metadata extractions, and further data lineage construction from various tools and technologies working with data, such as

[1]Within this work, the data flow concept will mean "*the movement of data through a system comprised of software, hardware or a combination of both.*" [2]

[2]The concept of metadata can be understood as data that provide information about other data [4]. This term will be explained in the relevant "Metadata" section later.

databases, ETL and reporting tools, programming languages, and others. These processes are performed by **scanners** designed for a specific tool or technology.

2.2.1 Manta Flow

Manta Flow software is Manta's flagship product, enabling metadata extraction and analysis using the technology-specific scanners mentioned above. Scanners convert the received metadata into an internal model and then use it to generate **a data flow graph** used for data lineage visualization.

"*Manta Flow is made up of three major components:*

- ***Manta Flow CLI*** *– Java command-line application that extracts all scripts from source databases and repositories, analyzes them, sends all gathered metadata to the Manta Flow Server, and optionally, processes and uploads the generated export to a target metadata database.*
- ***Manta Flow Server*** *– Java server application that stores all gathered metadata in its metadata repository, transforms it to a format suitable for import to a target metadata database, and provides it to its visualization or third-party applications via API.*
- ***Manta Admin UI*** *(runs on the Manta Flow Service Utility application server) – Java server application providing a graphical and programming interface for the installation, configuration, updating, and overall maintenance of Manta Flow.*" [6]

Figure 2.1 illustrates the Manta Flow architecture, its components, and interactions with third-party resources [6]. As will be seen later, the modules of the prototype implemented in this work will perfectly fit into this architecture.

2.2.2 Data flow graph

The end result of the data flow analysis performed by the scanner is a directed graph that describes these data flows and consists of a set of vertices connected by directed edges. Manta distinguishes several types of vertices and edges, but only a subset of them, described below, will be emphasized in this work. [7]

- Vertex types:
  - **Resource** – indicates a certain technology that is familiar to Manta. For example Snowflake, Talend, Excel, etc.
  - **Node** – represents a concrete object participating in the data flow. The characteristic parameter of this object is "nodeType", which specifies its particular subtype and serves to distinguish it from other nodes of the same *resource*. Examples: database column, script, technology-specific object, etc.
  - **Attribute** – means additional information that can be attached to a vertex of type *node*. This can include, for example, an absolute path to a file on disk, a value indicator, an applied expression, etc. [7]
- Edge types:
  - **DirectFlow** – directed edge connecting two vertices of type *node* and representing the direct flow of data from the source to the destination node.
  - **FilterFlow** – directed edge connecting two vertices of type *node* and representing the indirect flow of data from the source to the destination node. The indirect data flow may appear, for example, after applying various expressions or conditions. [7]

Figure 2.1 Manta Flow architecture [6]

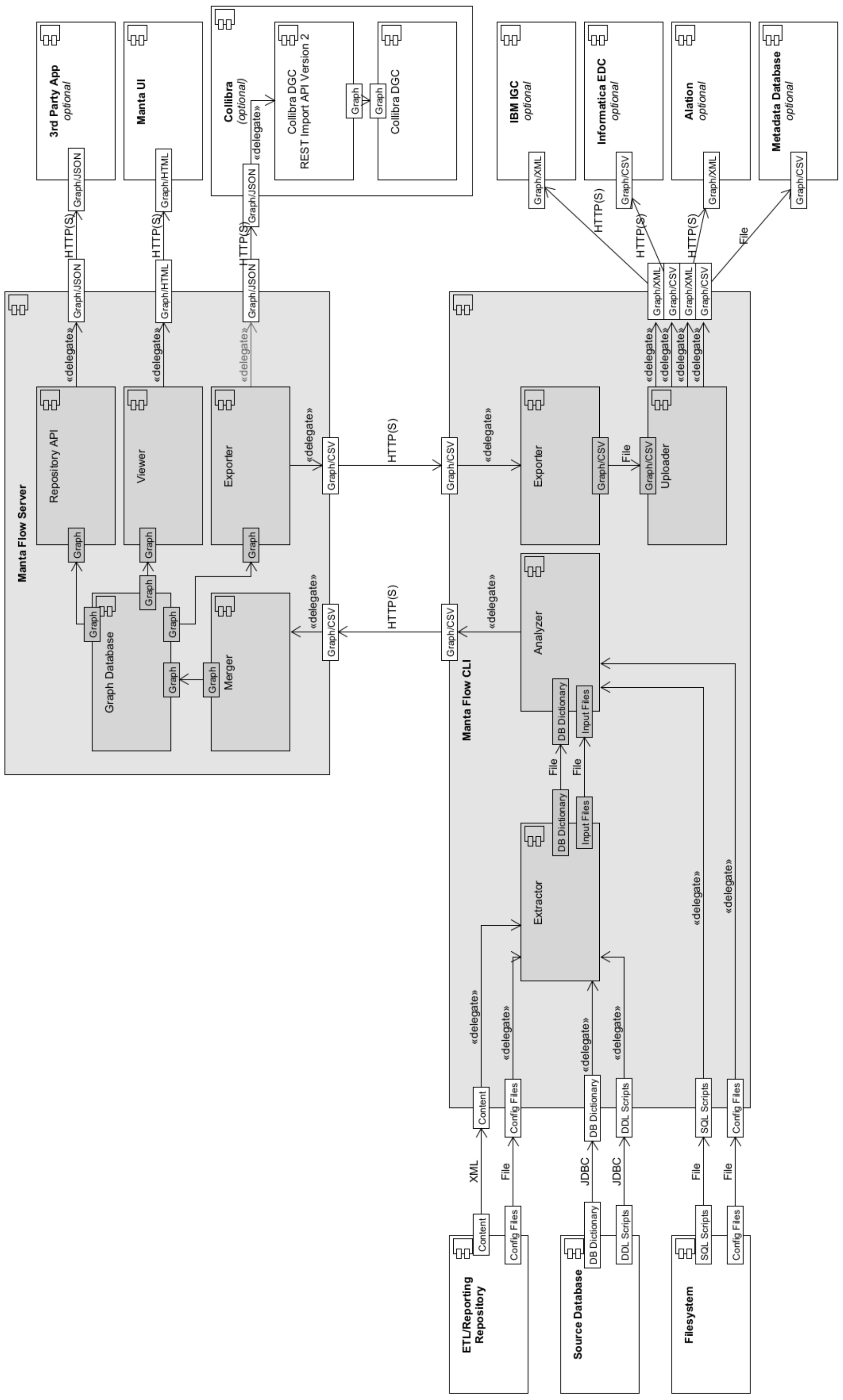
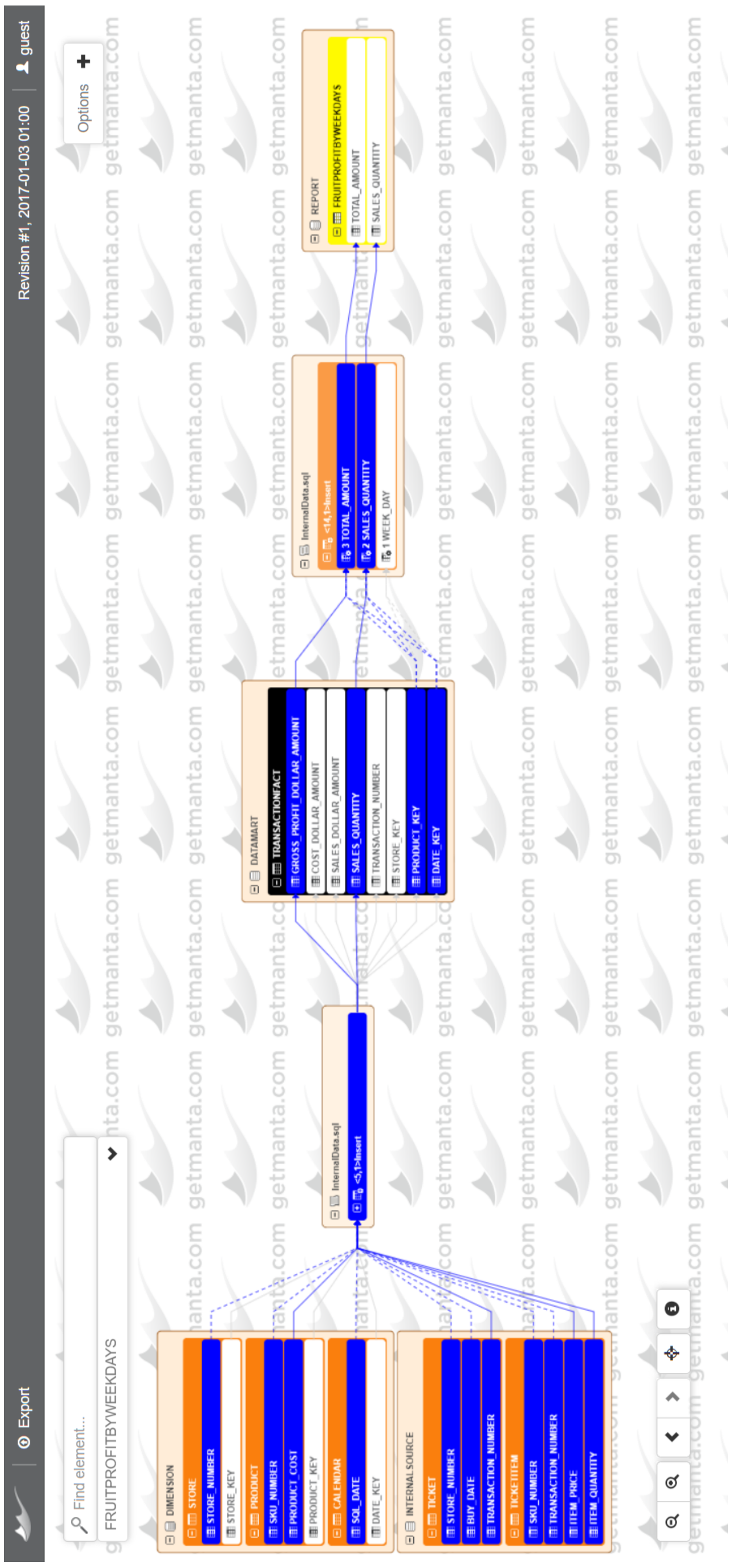


Figure 2.2 Data flow graph example in Manta Flow [8]



An example of the data flow graph can be seen in Figure 2.2 [8]. This example illustrates the column flow within specific objects of the Teradata technology from a data source on the left to a data destination on the right. Different elements of the graph can be colored differently depending on their characteristics, which helps the user to navigate more easily and quickly. The user can also select specific elements (highlighted in yellow in the example) and view their flow history (highlighted in blue).

2.3 Data warehouse

A data warehouse, [...] is a system that aggregates data from different sources into a single, central, consistent data store to support data analysis, data mining, artificial intelligence (AI), and machine learning. A data warehouse system enables an organization to run powerful analytics on huge volumes (petabytes) of historical data in ways that a standard database cannot. [9]

Cloud data warehouses (CDW), which store data in the cloud, deserve special mention. The "cloud" are servers that can be accessed over the Internet, and the data is stored on machines that are spread across the world. It allows organisations to provide applications that can be accessed by multiple people, globally. [10]

The complex process, without which data warehousing would be difficult, if not impossible, that brings the data together and makes it usable, is called **ETL**. This process will be explained in a bit, but for now, one particular data warehouse needs to be introduced in more detail.

2.3.1 Snowflake

Snowflake is one of the most popular cloud data warehouses. In this work, it will be mentioned a number of times, therefore, for a better understanding, it requires a separate description.

Designed with a patented new architecture to handle all aspects of data and analytics, Snowflake combines high performance, high concurrency, simplicity, and affordability at levels not possible with other data warehouses. [11]

"*Snowflake physically separates but logically integrates storage, compute, and services (like metadata and user management). Because each one of these components is separate, they can be expanded and contracted independently, enabling Snowflake to be more responsive and adaptable.*" [11]

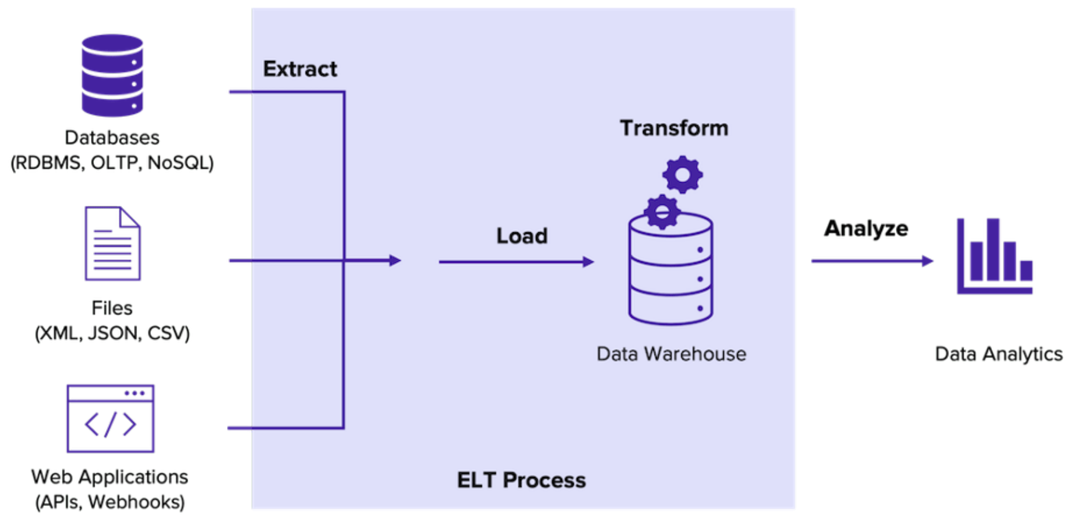
2.4 ETL and ELT

"*ETL, which stands for "extract, transform, load," are the three processes that, in combination, move data from one database, multiple databases, or other sources to a unified repository – typically a data warehouse. It enables data analysis to provide actionable business information, effectively preparing data for analysis and business intelligence processes.*" [12]

For a better understanding, it is worth looking at these processes more closely.

- **Extract** – the phase during which the raw data is pulled from one or multiple structured or unstructured sources to a staging area. These sources can be, for example, a database, flat file, API, email, or others. Extracted data may also be in several formats, such as relational database table(s), XML, JSON, etc.
- **Transform** – the phase when the raw data in the staging area might undergo various transformations to match the needs of an organization or other requirements. This phase may include filtering, cleansing, sorting, merging, and many other changes.
- **Load** – the last phase, in which the transformed business-ready data is moved from the staging area into a target data storage, becoming available for further analysis or other processing.

Figure 2.3 ELT process [14]



ETL has been around for a while, however, with cloud data warehouses coming into the picture, ELT (extract, load, transform) has emerged as the newer approach. "*The most obvious difference between ETL and ELT is the difference in order of operations. ELT copies or exports the data from the source locations, but instead of loading it to a staging area for transformation, it loads the raw data directly to the target data store to be transformed as needed.*" [13] ELT uses powerful data destination capabilities for the data transformations, also eliminating the need for its staging. ELT process diagram can be seen in Figure 2.3 [14].

Comparing the ETL and ELT processes is beyond the scope of this work, but it is worth noting that they both have their advantages and disadvantages and serve the same purpose, having a different implementation. The end user must determine which process is more profitable to use in a particular case.

In the past, organizations had to write their own code for these processes. Fortunately, there are now many open source, commercial, and cloud tools and services to choose from. One of them is **Matillion ETL**.

2.4.1 Matillion ETL

Matillion ETL is an ETL/ELT close source tool built specifically for cloud database platforms including Amazon Redshift, Google BigQuery, Snowflake, Microsoft Azure Synapse, and Delta Lake. It is a modern, browser-based UI, with powerful, push-down ETL/ELT functionality. [15] Although modern solutions are rapidly moving to ELT, "ETL" remains the common name for all such processes, which is why Matillion uses this designation.

As already mentioned, Matillion supports several cloud data warehouses, however, this work will be aimed only at the instance of the tool designated to use Snowflake as a target storage. The reason for this is the instance of the tool provided by the vendor, limited only to Snowflake. Nevertheless, the work will include laying the foundation for expansion and working with other warehouses in the future.

The Matillion team unveils a major release of Matillion ETL roughly once every eight weeks, and an interim release roughly once every two weeks, where required, to bring pivotal new features and improvements to the customers and to fix bugs where and when they arise. In this work, version 1.64.10 of the tool released on July 25, 2022, will be used, however, all newer versions should be compatible. [16]

For clarity, it should be mentioned that the prototype of the functional module that will be implemented in this work, in the context of the Manta project, will be called **the Matillion scanner prototype**.

All key elements in Matillion ETL will be introduced in detail later in the relevant "Analysis" chapter section.

2.5 SQL

"*Structured query language (SQL) is a programming language for storing and processing information in a relational database. A relational database stores information in tabular form, with rows and columns representing different data attributes and the various relationships between the data values. You can use SQL statements to store, update, remove, search, and retrieve information from the database. You can also use SQL to maintain and optimize database performance.*" [17]

In the context of this work, SQL language will be used when working with Snowflake. Although Snowflake has its own specific SQL dialect, it supports all of the most frequently used operations.

2.6 Metadata

"*Metadata is often simply described as data about data. A variety of definitions exist, and a simple Google search reveals many resources out there which describe in detail the nuts and bolts of metadata in different fields.*" [4] In this work, a more specific definition of metadata will be set as data that describes other data, their structure and characteristics, but not their content. For example, in the context of relational databases, information about schemas, tables, columns, but not about individual rows with content, will be considered metadata. Also worth mentioning is the context of ETL and ELT tools, which often provide metadata about the changes and transformations of the data they work with. The last mentioned context will play a very significant role in this work, when a lineage will be built based on the metadata obtained from Matillion ETL.

2.7 Java

The heart of the tech stack used during the implementation phase of this work is Java, "*a widely used object-oriented*[3] *programming language and software platform that runs on billions of devices, including notebook computers, mobile devices, gaming consoles, medical devices and many others. The rules and syntax of Java are based on the C and C++ languages.*

One major advantage of developing software with Java is its portability." [19] It is achieved by the fact that the Java compiler does not produce executable code, but the so-called bytecode – a highly optimized set of instructions designed to be executed in the Java runtime system, called Java Virtual Machine (JVM). The pure bytecode can then be executed on any system which supports JVM. [20]

Since its inception and until now, the language has been developing quite dynamically and has many versions. The last long-term support version (LTS) is Java 17, released in September 2021. [21] In this work, however, the LTS Java 11 version will be used, since the Manta project compiles and runs on it.

[3]"*Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which can contain data and code: data in the form of fields (often known as attributes or properties), and code, in the form of procedures (often known as methods).*" [18]

2.7.1 Javadoc

Javadoc is the most common and convenient tool for generating documentation from Java source code [22]. It will be used to document the module prototype code. The tool generates documentation by parsing special comments related to a particular element of the language, such as a class, method, attribute, etc.

2.8 IDE

An integrated development environment (IDE) is software for building applications that combines common developer tools into a single graphical user interface. An IDE usually consists of at least a source code editor, build automation tools, and a debugger, and may also include an integrated version control system, compiler, interpreter, and other tools. [23]

IntelliJ IDEA is an IDE for JVM languages designed to maximize developers productivity. It does the routine and repetitive tasks for them by providing clever code completion, static code analysis, refactorings, and has many other useful features. [24] This IDE was chosen for development of the module prototype based on the author's previous experience with it and its convenience.

2.9 Apache Maven

Apache Maven is a powerful project management software widely used in Java development that is based on POM (project object model). POM is the fundamental unit of work in Maven. It is an XML file that contains information about the project, its default values, and configuration details for building it. [25]

Maven will ease the process of building the implemented module prototype project by automating it, performing dependency management, and loading the required libraries.

2.10 Spring

Spring is the most popular open source framework for enterprise Java. Developers all over the world use Spring to create reliable, high-quality, and easily testable code. The framework is divided into a number of separate modules, and one of them, called Core module, provides key parts of the framework including **Dependency injection**. [26]

2.10.1 Dependency injection

Dependency Injection (DI) is a design pattern and a fundamental aspect of the Spring framework, through which the Spring mechanism, called IoC[4] Container, "injects" objects into other objects or "dependencies", and makes them ready for use. Simply put, this allows objects to be loosely coupled and moves the responsibility of managing them onto the container. [28]

In fact, Spring has a special name for the objects mentioned above. It calls them **Beans**[5].

Spring provides several DI configuration techniques, and it is important to mention that during the implementation part of this work, XML-based beans configuration will be used because of the general approach in Manta scanners.

[4]Inversion of control (IoC) is a design principle which helps to invert the control of object creation [27]. Dependency Injection implements this principle.

[5]Any standard Java plain class can be a Spring Bean if it is configured to be initialized via container by providing configuration metadata information. [28]

2.11 JUnit

"*JUnit is a unit testing*[6] *open-source framework for the Java programming language. Java developers use this framework to write and execute automated tests. In Java, there are test cases that have to be re-executed every time a new code is added. This is done to make sure that nothing in the code is broken.*" [29]

The latest version of this framework called JUnit 5 will be used to increase the reliability of the Matillion scanner.

2.12 JSON

"*JSON, or JavaScript Object Notation, is a format used to represent data. It was introduced in the early 2000s as part of JavaScript and gradually expanded to become the most common medium for describing and exchanging text-based data. Today, JSON is the universal standard of data exchange. It is found in every area of programming, including front-end and server-side development, systems, middleware, and databases.*" [30]

As will be seen later, the metadata that the scanner will work with will be in the JSON format, so it is necessary to get acquainted with the tools for working with it effectively in Java, such as **the Jackson Streaming API** and **ObjectMapper**.

2.12.1 Jackson Streaming API

The Jackson Streaming API processes JSON content as discrete events, therefore it allows to parse huge JSON documents without loading their whole content into memory at once. It is the most efficient way to process JSON data and has the lowest memory and processing overhead, but it comes with a cost: is not the most convenient way to process JSON. The code that comes out of working with the API is verbose, repetitive, and tedious to write. However, in combination with another tool called `ObjectMapper`, working with the API becomes much more enjoyable. [31]

2.12.2 ObjectMapper

The `ObjectMapper` class provided by the Jackson library is the simplest way to parse JSON with Jackson. It can parse JSON from a string, stream, or file and create a Java object or object graph representing the parsed JSON. Parsing JSON into Java objects is also known as deserializing Java objects from JSON. [32]

2.13 Git

Git is an open source, distributed, and the most commonly used version control system that keeps track of changes made to files, so any record of what has been done to them is available, as well as reverting to a specific version. Git also makes collaboration easier, allowing changes by multiple people to all be merged into one source. A complete copy of all project files and the entire revision history are located locally on each user's computer in a place called Git repository. [33]

[6]Unit testing can be understood as testing small parts of code, or units, such as, for example, individual class methods.

2.14 Jenkins

Jenkins is an open-source automation tool built for continuous integration purposes. It is used to build and test software projects continuously, making it easier for developers to integrate changes into a project and making it easier for users to obtain a fresh build. It also allows continuous software delivery by integrating with a large number of testing and deployment technologies. [34]

2.15 SonarQube

SonarQube is an open-source platform for continuous inspection of code quality. It performs automatic reviews with static analysis of code to detect bugs, code smells, vulnerabilities, code duplication, and many others flaws in numerous programming languages, including Java. [35]

Chapter 3

Analysis

In this chapter, the analysis of the Matillion ETL tool will be made, mainly focusing on its key elements involved in data flow, as well as the way metadata can be exported from the tool and its characteristics. Further, the chapter will introduce the functional and non-functional requirements for the Matillion scanner prototype that will be implemented.

3.1 Matillion ETL

The information in this section is based on the official Matillion ETL documentation [36].

Matillion ETL is purpose-built for such cloud database platforms as Snowflake, Amazon Redshift, Google BigQuery, Microsoft Azure Synapse, and Delta Lake. Although each of these products contains specific features and functionalities for the analytics platform they support, users interact with Matillion ETL in the same way. Despite the ETL abbreviation in the name, Matillion is an ELT tool, which means that the data loading process precedes its transformation.

In order to have a better idea of the instrument, its browser-based user interface is presented in Figure 3.1.

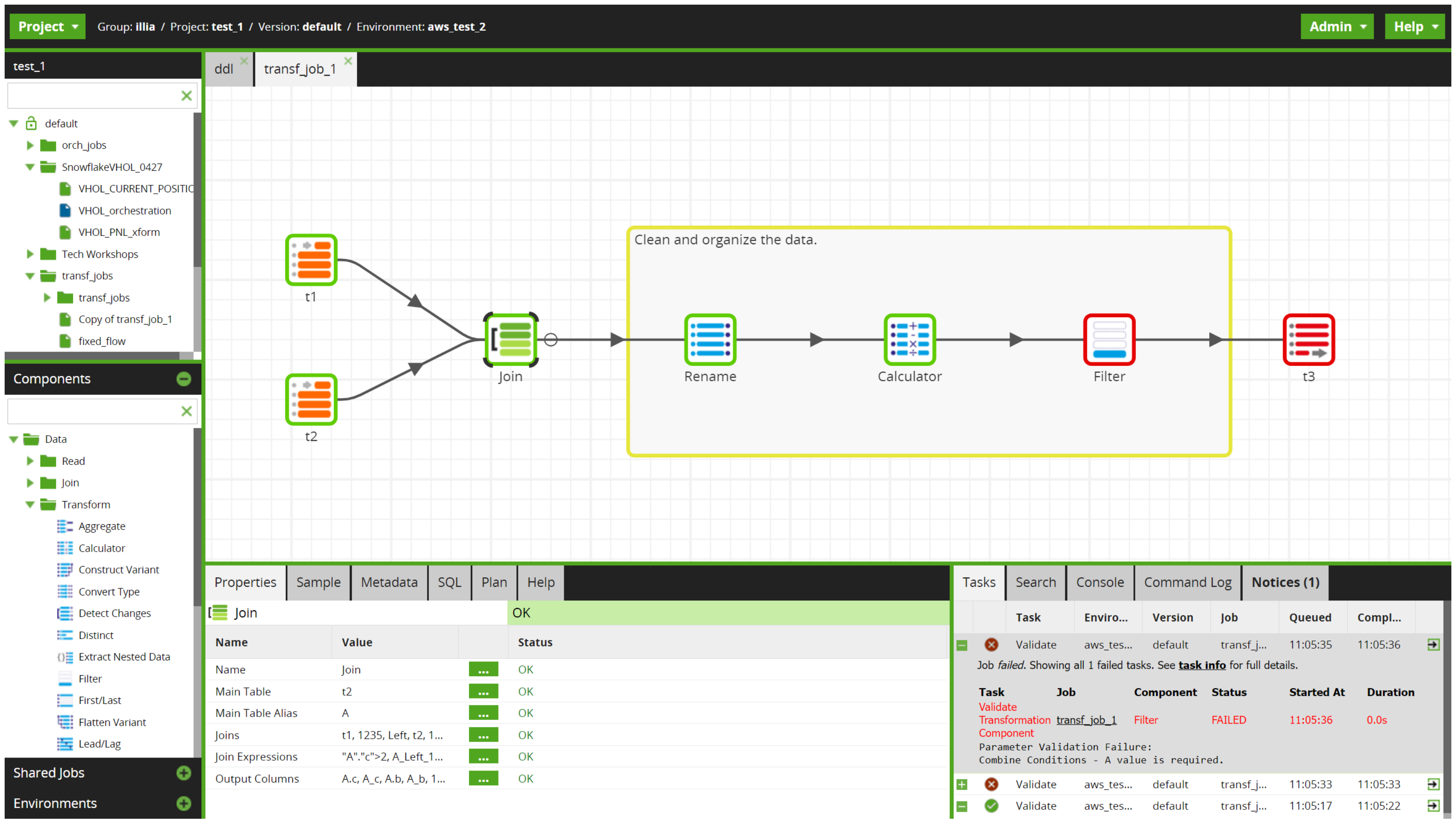
3.1.1 Key elements

Matillion ETL has several key elements that the user mainly interacts with and that play an important role in the data flow. Let's look at the description of each of them separately.

3.1.1.1 Project

To get started in Matillion ETL, the user must first create a project. A project is a logical grouping of configuration settings and resources, such as **jobs**. To create a project, the user must specify its name and the name of the project group to which it will belong. Thus, projects are logically grouped into project groups. Next, each new project will be assigned a version named "default". Subsequently, the user can create other versions as the project changes. Versioning supports many use cases, but perhaps the most salient is to be able to capture development at a point in time to designate as "live", "production", etc. Also, this feature allows to make changes to the project without fear of breaking it, since it is always possible to revert to an older version.

Figure 3.1 Matillion ETL user interface



Join OK

Name	Value	Status
Name	Join	OK
Main Table	t2	OK
Main Table Alias	A	OK
Joins	t1, 1235, Left, t2, 1...	OK
Join Expressions	"A"."c">2, A_Left_1...	OK
Output Columns	A.c, A_c, A.b, A_b, 1...	OK

Task	Enviro...	Version	Job	Queued	Compl...
Validate	aws_tes...	default	transf_j...	11:05:35	11:05:36

Job *failed*. Showing all 1 failed tasks. See **task info** for full details.

Task	Job	Component	Status	Started At	Duration
Validate Transformation Component	transf_job_1	Filter	FAILED	11:05:36	0.0s

```
Parameter Validation Failure:
Combine Conditions - A value is required.
```

Task	Enviro...	Version	Job	Queued	Compl...
Validate	aws_tes...	default	transf_j...	11:05:33	11:05:33
Validate	aws_tes...	default	transf_j...	11:05:17	11:05:22

3.1.1.2 Environment

An environment in Matillion ETL describes which sets of target cloud database platform credentials to use for a connection. Multiple environments can be configured at the project level, meaning different users can use different environments in the same project. When a user runs a job, it runs within the environment currently in use by that user.

3.1.1.3 Job

Jobs are Matillion ETL's main way of designing, organising, and executing workflows. The most common usage of Matillion ETL is to build strings of configured **components** inside a job and then run that job to accomplish a desired task, such as extracting, loading, or transforming data. Each job belongs to a specific version of the project and is located in a particular project job folder.

There are two main flavours of jobs:

- **Orchestration** – primarily concerned with DDL[1] statements (especially creating, dropping, and altering resources), extracting data from external sources, and loading them to the target cloud platform.
- **Transformation** – used for transforming data that already exists in the target warehouse tables, getting it ready for analysis. This includes filtering data, changing data types, removing rows, and many other transformations.

It is worth mentioning that Matillion ETL has a special feature called "Shared Jobs" that allows users to embed jobs into each other as components. Working with this feature is out of the scope of this thesis due to its complexity.

3.1.1.4 Component

Each component is named according to its particular function: from moving data into file storage to performing custom calculations on table columns. The area in which components are laid out within a job is called the canvas. Components come in two types, depending on the type of its parent job: orchestration and transformation. Each component has a unique set of user-configurable properties that effect its function, such as connection string, source column names, filter condition, column mapping, and many more. Changes made to these properties will not affect the properties of any other component, even components of the same type. It should also be noted that the configurable properties for the same component type may vary for different target platforms of Matillion ETL.

Matillion ETL currently supports more than 150 components and its developers often add new ones in updates, depending on the needs of customers. It should be clarified that the components have a different usage frequency. The list of the most frequently used components includes "Database Query", "Join", "Table Input", "Filter", "Rename", and some other components. The Matillion scanner prototype will not support all the components as it would be very time consuming and out of the scope of this work, but only some, including those mentioned above.

3.1.1.5 Connector

Components should be linked to one another in the order in which they will need to be executed. This is achieved using so-called connectors. Depending on its type, the source component can be connected to the target component(s) 1:1 or 1:N (using multiple output connectors).

[1]A data definition language (DDL) is a language used to create and modify the structure of objects in a database. These database objects include views, schemas, tables, indexes, etc. [37]

There are several types of orchestration component connectors, such as, for example, successful, unsuccessful, unconditional, and others. Depending on the source orchestration component execution result, the appropriate connector type will be selected to continue the data flow. Transformation components, in turn, have only one unnamed type of connectors.

3.1.1.6 Variable

Matillion ETL supports several types of variables that can be used in all sorts of properties and expressions to allow the user to pass and centralize configuration:

- **Job variables** – name-value pairs that are defined within the scope of a single job and cannot be used outside of it, that is, in other jobs. These variables will override any environment variables of the same name within that specific job.
- **Grid variables** – a special type of job variables. Represent two-dimensional arrays that hold scalar values in named columns. These variables can be used in many places where lists of data need to be passed around.
- **Environment variables** – name-value pairs, which have global scope and can be used everywhere, unlike job variables. Environment variable values are defined for each project environment separately.

When using job and environment variables, their declaration must have the following format: `${<variable_name>}`. For instance, if the user wants to use the job variable named "path", it must be entered as follows: `${path}`. Grid variables are applied through the user interface using the "Use Grid Variable" option in some properties configuration.

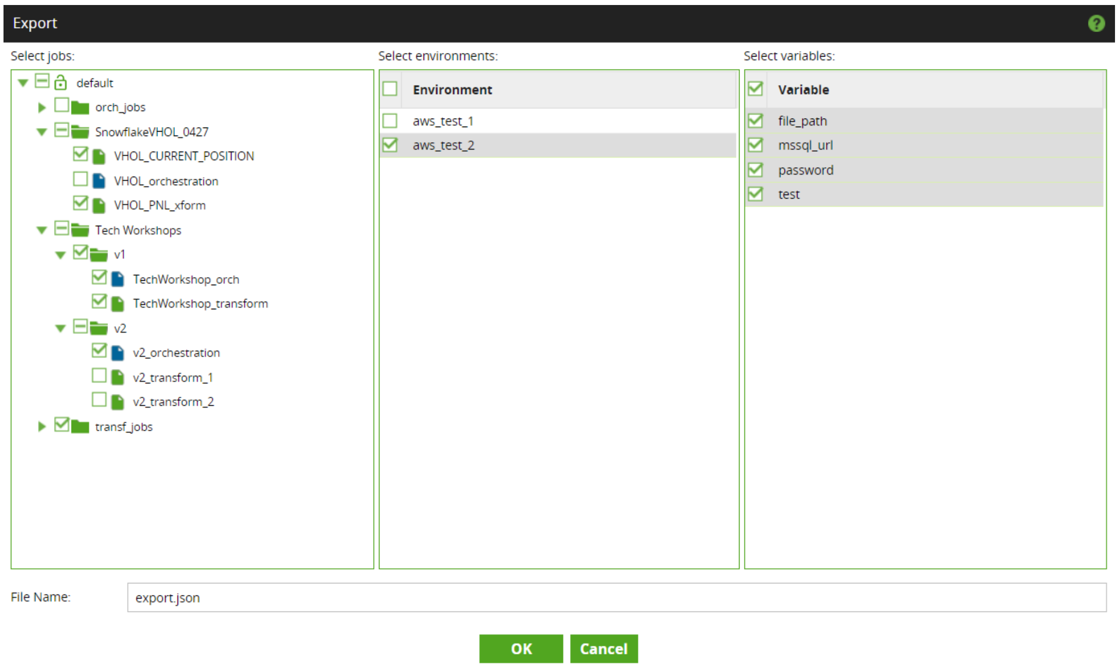
3.1.2 Metadata export

Project information, such as its jobs, environments, and variables, can be exported from one Matillion ETL instance to be imported into another, for example, for the migration or sharing purposes. However, for the Matillion scanner, this function can be used as an excellent source of metadata.

One way of accomplishing the export is through the Matillion ETL API, which provides several endpoints for this. The advantage of this method is that the export will be done automatically, however, for this the user will have to go through a series of configurations related to providing credentials to create a connection to the Matillion ETL server and specifying the target projects elements to be exported. The responses from the API come in JSON format.

The tool also provides a second export method with its user interface. The Figure 3.2 shows a special menu for exporting jobs, environments, and variables belonging to a particular tool instance project. It allows the user to export them as a single file, which can later be transferred to the Matillion scanner for data flow analysis. This manual export method has a number of disadvantages, such as, for example, shifting the responsibility for exporting metadata to the user, who will also be forced to work with the file system. On the other hand, this method allows the user to describe the scanner exactly what needs to be analyzed and provide everything necessary for this. From the implementation point of view, the manual export method is more profitable, because working on it will take less time than working on the export method via the API, which will allow investing more time and focusing on the implementation of data flow analysis. That is why, within this work, the Matillion scanner prototype will only support the manual export method. Nevertheless, the prototype will be aimed at adding API export support in the future and will contain a good basis for this. The only question that needs to be answered is: does the manual export metadata contain enough information to track the data lineage?

Figure 3.2 Matillion ETL export menu



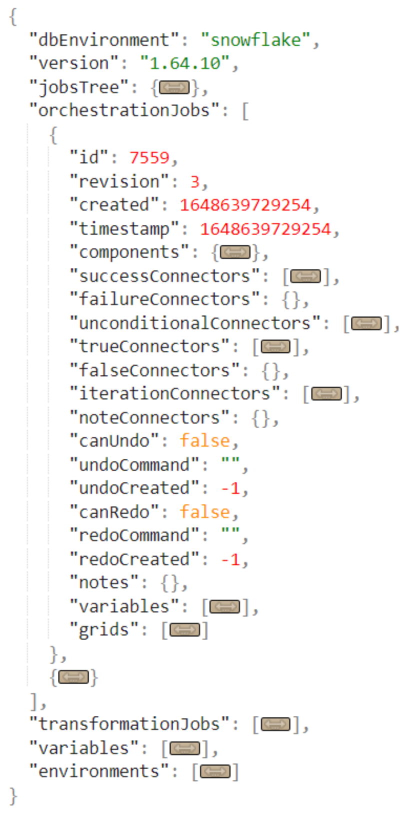
3.1.2.1 Manual export files format and structure

Manual export files have JSON format. Although the structure of these files was created and intended for internal use in Matillion ETL, they are suitable for achieving the goals of this work. It is worth mentioning once again that each export file belongs to some project and describes its elements.

An example of the root structure of each file can be seen in Figure 3.3. This structure represents an object containing the following elements:

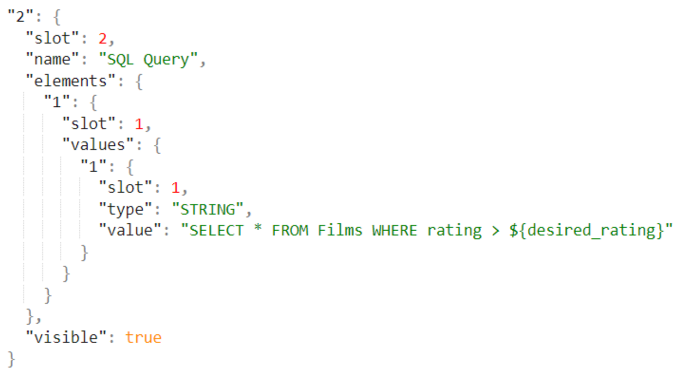
1. **dbEnvironment** – name of the cloud storage dedicated to the given Matillion ETL instance.
2. **version** – Matillion ETL instance version.
3. **jobsTree** – a special object that stores basic information about exported jobs, namely their identifiers, names, types, descriptions, and internal project folders where these jobs are located.
4. **orchestrationJobs** – a list of exported orchestration job objects, where each object contains detailed information about the job, namely its identifier, components, connectors of different types, variables, and other elements not related to the data flow.
5. **transformationJobs** – a list of exported transformation job objects, having a similar structure as the previous orchestration job objects list, except that each transformation job object contains only one type of connectors.
6. **variables** – a list of exported environment variable definition objects. Each object stores complete information about the variable, such as, for example, its name and type.
7. **environments** – a list of exported environment objects. Each object stores information about the environment, namely its name, connection credentials, and the values of environment variables relevant to this environment.

Figure 3.3 Matillion ETL manual export file root structure



```
{
  "dbEnvironment": "snowflake",
  "version": "1.64.10",
  "jobsTree": {↔},
  "orchestrationJobs": [
    {
      "id": 7559,
      "revision": 3,
      "created": 1648639729254,
      "timestamp": 1648639729254,
      "components": {↔},
      "successConnectors": [↔],
      "failureConnectors": {},
      "unconditionalConnectors": [↔],
      "trueConnectors": [↔],
      "falseConnectors": {},
      "iterationConnectors": [↔],
      "noteConnectors": {},
      "canUndo": false,
      "undoCommand": "",
      "undoCreated": -1,
      "canRedo": false,
      "redoCommand": "",
      "redoCreated": -1,
      "notes": {},
      "variables": [↔],
      "grids": [↔]
    },
    {↔}
  ],
  "transformationJobs": [↔],
  "variables": [↔],
  "environments": [↔]
}
```

Figure 3.4 Matillion ETL manual export file component property structure



```
"2": {
  "slot": 2,
  "name": "SQL Query",
  "elements": {
    "1": {
      "slot": 1,
      "values": {
        "1": {
          "slot": 1,
          "type": "STRING",
          "value": "SELECT * FROM Films WHERE rating > ${desired_rating}"
        }
      }
    }
  },
  "visible": true
}
```

Digging deeper into the structure of the export files, it can be seen that they contain detailed information about each job component, including all of its user-configurable properties. Figure 3.4 shows an example of the property structure of a component that executes an SQL query entered by the user.

From what was written above, it can be summed up that manual export files provide enough information needed to track the data lineage. The Matillion scanner will have to be able to load, parse them, and store all the information needed for the data flow analysis in an internal representation. However, there are also several important issues that need to be addressed, namely:

- The size of individual manual export files can be gigantic (hundreds of megabytes), so when they are loaded, there may not be enough memory.
- The project group name, project name, and project version name parameters are missing in the export files. It is not possible to determine which version of a project the exported jobs in the files belong to.
- Incorrectly configured jobs, their component properties, environments, and other elements may still be successfully exported for data flow analysis.

3.2 Matillion scanner requirements

For a better understanding of the scanner's capabilities, it is necessary to define the requirements for it and divide them into functional and non-functional types.

3.2.1 Functional requirements

Functional requirements define what precisely a software must be able to do and how a system must respond to inputs. They define the software's goals, meaning that the software will not work if these requirements are not met. [38] The following functional requirements have been established for the Matillion scanner:

- F1: The scanner will be able to efficiently load JSON files manually exported from Matillion ETL by the user, who must place them in a special folder structure expected by the scanner.
- F2: The scanner will extract the necessary metadata from the provided files and convert it into an internal convenient model.
- F3: Based on the extracted metadata, the scanner will perform data flow analysis and graph generation.
- F4: The scanner will support analysis of all major tool elements, including both orchestration and transformation Matillion ETL jobs. In the context of a Matillion ETL project, each job provided by the user will be analyzed within each provided Matillion ETL environment. As mentioned earlier, at the moment the analysis will only be supported for at least ten of the most frequently used Matillion ETL components and with Snowflake as the target platform.
- F5: The generated data flow graph will have a rational nodes structure that describes the analyzed elements and their relationships. The graph nodes will contain all relevant characteristics and attributes.
- F6: The scanner will be able to work with incorrectly configured tool elements, filling in a possibly missing lineage using a special deduction mechanism, which will be presented in the next chapter.
- F7: In case of any error or other illegal state, the scanner will log a meaningful message.

3.2.2 Non-functional requirements

Non-functional requirements are the constraints or the requirements, such as scalability, maintainability, performance, portability, security, reliability, and many others, imposed on a system. They specify the quality attribute of the software. [39] The Matillion scanner should satisfy the following non-functional requirements:

- N1: The scanner will be easily extensible to add new functionality that was beyond the scope of this work, namely support for the remaining Matillion ETL target cloud database platforms (Amazon Redshift, Google BigQuery, Microsoft Azure Synapse, and Delta Lake), metadata extraction via the Matillion ETL API, support for all remaining relevant components of the tool.
- N2: The scanner will be seamlessly connected to the Manta project, so it must follow the module structure common to Manta scanners, as well as Manta naming conversions. The scanner will use common technologies developed or used by Manta, and only third-party libraries approved for use by Manta.
- N3: Metadata extraction, data flow analysis, and graph generation processes must be performed by the scanner in a reasonable amount of time, with a limit of five seconds per one Matillion ETL job analyzed within one tool environment.
- N4: The scanner code will be well readable, all its essential public and protected class methods must be properly documented using Javadoc.
- N5: The scanner must be qualitatively tested using various types of tests, including unit, integration, functional, and end-to-end. SonarQube tool should be used to evaluate the quality of the written code.

Chapter 4

Matillion scanner design

In this chapter, the design of the Matillion scanner will be introduced, focusing on the two essential modules that it consists of. The relationships and logic of these modules and their submodules, as well as the main algorithms and mechanisms of the scanner can be found here.

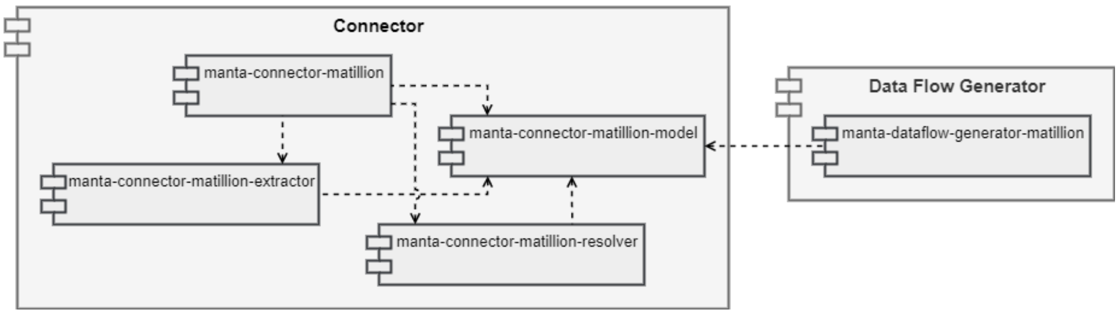
Integration with the Manta project implies that the Matillion scanner design should follow the module structure common to other Manta scanners. Therefore, the scanner will consist of two main modules called Connector and Data Flow Generator. The dependency diagram of these modules and their submodules is shown in Figure 4.1. The result of their joint and coordinated work will be a graph representing data flows in Matillion ETL.

4.1 Connector module

The Connector module will serve to load the input JSON files, extract the necessary metadata from them, and transform this metadata into an internal model that the Data Flow Generator module[1] will work with.

Before talking about the structure of this module and the functionality of its submodules, it is necessary to familiarize yourself with the strategy for working with input files.

Figure 4.1 Dependency diagram of the Matillion scanner modules


Connector
manta-connector-matillion
manta-connector-matillion-model
manta-connector-matillion-extractor
manta-connector-matillion-resolver
Data Flow Generator
manta-dataflow-generator-matillion

[1]The Data Flow Generator module will provide data flow analysis and graph generation. It will be introduced later in a separate section of this chapter.

Figure 4.2 Manual input directory structure

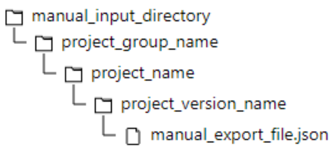
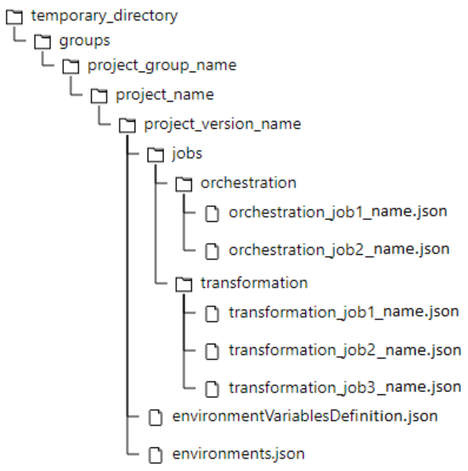


Figure 4.3 Structure of the temporary directory for split manual input files


temporary_directory
groups
project_group_name
project_name
project_version_name
jobs
orchestration
orchestration_job1_name.json
orchestration_job2_name.json
transformation
transformation_job1_name.json
transformation_job2_name.json
transformation_job3_name.json
environmentVariablesDefinition.json
environments.json

4.1.1 Manual input directory structure

After manually exporting JSON files from Matillion ETL, the user will need to transfer them to the scanner. At this stage, the problem mentioned above, related to the absence of project group name, project name, and project version name parameters in the export files, can be solved.

To do this, the user will have to place the files in the directory structure shown in Figure 4.2. As can be seen from the figure, the user will need to create three directories, each of which will be placed in the previous one. The directory names will match the missing parameters. The exported file with metadata will be placed in the last directory, allowing the scanner to determine which version of the project the exported jobs in the file belong to. This structure will also allow the scanner to perform data flow analysis of all jobs available in the user's Matillion ETL instance.

Another problem, also mentioned above, related to the extremely large size of individual manual export files, still remains. The next section is devoted to its solution.

4.1.2 Manual input files splitting

The size of the exported metadata in a single input file cannot be controlled, so there is no guarantee that the user's environment has enough memory to process it. The solution to this could be to split the files into smaller ones and use them as a metadata source for the scanner.

To achieve this, each export file must be divided into logical independent units that can be used as separate inputs for data flow analysis, such as jobs. Each job JSON object located in the source export file will now represent a single file named after the contained job and placed in a specially created temporary directory structure that extends the previous one, as shown in Figure 4.3. The lists of exported environment and environment variable JSON objects will now also be separate files with fixed names.

4.1.3 Module structure

The Connector module will consist of four submodules, following the common design of other Manta scanners. All of them will be introduced in separate sections below, in order of how the Matillion scanner will work with the inputs.

4.1.3.1 Extractor submodule

Within this work, the `manta-connector-matillion-extractor` submodule, or simply Extractor, will serve to load manually exported files, split them into smaller parts, and save them in a created temporary directory for later use. The design of this submodule, however, should consider adding support for metadata extraction via the Matillion API in the future, in which the received response will be processed in the same way as manual input, that is, split into smaller parts and stored in a similar temporary directory structure. For this, two interfaces named `IExtractor` and `ISplitter` were created.

`IExtractor` declares only one method called `extract`, which performs the extraction and runs the splitting of the extracted data, the type of which depends on the type of extraction. This method must be overridden in classes that implement this interface, such as `ManualInputExtractor` and `ApiExtractor`. The first class is used to “extract” manual input files, meaning reading them and passing to an instances of `ManualInputFileHolder`, each of which stores the input file in form of `java.io.File` object and other supporting information, such as project group name, project name, and project version name received from the input file path. The `extract` method in `ApiExtractor` has been left unimplemented, but will be used for the connection to the Matillion API and extraction through it in future versions of the scanner.

Each `ManualInputFileHolder` object is then passed for splitting to a concrete class that implements `ISplitter` interface, named `ManualInputJsonSplitter`. This class overrides a single method defined in `ISplitter` called `split`, which takes the extracted data as a parameter, splits it, and continuously saves the ready split parts of it in a created temporary directory. The class uses a combination of the Jackson Streaming API and `ObjectMapper` to parse JSON, which ensures efficient iteration over the contents of each input file. Another class named `ApiJsonSplitter` that implements `ISplitter` interface will later be used in the same way to split the JSON content of the API response.

4.1.3.2 Reader submodule

Split files containing the necessary metadata will be read in the `manta-connector-matillion` submodule. The main class involved in this process is called `MatillionInputReader`. Manta expects this class to extend `AbstractFileInputReader`, which is the common parent for classes that provide reading of primary input files in various Manta scanners. To extend this class, `MatillionInputReader` must implement its abstract method called `readFile`. Manta calls this method in a loop, passing each primary input file as a parameter to it. As mentioned earlier, for the Matillion scanner, this input file will be the job file that appeared in the temporary directory after the splitting process. The scanner reads the given job file, as well as the relevant files with environments and environment variables definition metadata in the specified overridden method, using instances of the created `JsonJobReader`, `JsonEnvironmentsReader`, and `JsonEnvironmentVariablesDefinitionReader` classes. These classes use the combination of the Jackson Streaming API and `ObjectMapper` again to parse JSON content and convert it into an internal model. After that, the `readFile` method returns a scanner-specific object representing a single input for data flow analysis, whose class, in case of the Matillion scanner, must implement the `IDataflowInput` interface, which will be discussed in more detail in the next section.

4.1.3.3 Model submodule

The `manta-connector-matillion-model` submodule will be intended to declare the internal representation of the metadata received by the scanner. For this, a special structure of Java interfaces and enumerations was created. Its most important elements can be seen in the class diagram shown in Figure 4.4. The concrete classes that implement these interfaces will be located in another submodule, which will be discussed in the next section. Such logic distribution allows to separate the processes of filling the internal model with metadata from working with it during the data flow analysis.

As already mentioned, the single input for data flow analysis in the Matillion scanner is an object whose class implements the `IDataflowInput` interface. This object stores a single job, as well as a list of environments within which this job will be analyzed. The object also stores the project group name, project name, and project version name values, needed to create the correct node structure of the generated data flow graph.

The `IJob` interface declares methods that provide access to various properties of a job, such as its internal identifier, type, variables, and the components and their connectors contained in it. Two interfaces named `IOrchestrationJob` and `ITransformationJob` extend `IJob` and represent concrete jobs types.

Each component type has its own interface that provides access to its unique set of user-configurable properties, as well as common properties for all types of components. Although the configurable properties for the same component type may vary across different Matillion ETL target platforms, the component interface will contain methods to retrieve all possible properties, so that only the relevant ones can be used. The `IUnknownOrchestrationComponent` and `IUnknownTransformationComponent` interfaces, which do not provide access to any unique properties, were created for unsupported components. In the future, when adding support for a new component, its special interface will be added. The `IDatabaseConnectiveComponent` interface was added in order to separate such components that work with databases (for example, extract data from them). During the data flow analysis, such components will need to be approached separately, which will be discussed later in the relevant “Connection to databases” section.

The interfaces `IOrchestrationJob`, `ITransformationJob`, `IOrchestrationComponent`, and `ITransformationComponent` may seem superfluous, but several properties can be obtained through them that are not currently used, but might be needed in the future in further development of the scanner. Also, there is still a possibility that the structure of the exported metadata may change with the latest versions of the Matillion ETL, or new properties will be added that can later be obtained through these interfaces and used during the data flow analysis performed by the scanner.

The `IEnvironment` interface declares methods that provide credentials needed by the scanner to connect to the target cloud storage that the user’s Matillion ETL instance is working with, as well as environment variables with values specific to this environment. As already known, the implemented prototype will only work with Snowflake environments, using `ISnowflakeEnvironment`, but in the future it will be possible to add new interfaces for other cloud platforms.

4.1.3.4 Resolver submodule

The Java classes located in the `manta-connector-matillion-resolver` submodule will implement the interfaces of the Model submodule. Objects of most of these classes will be instantiated by the `ObjectMapper` instance in the Reader submodule’s `readFile` method mentioned earlier. This means that the `ObjectMapper` will transform the JSON content of the input files into concrete Java objects. To make this possible, the class of the created object must declare attributes with names and types identical to the JSON fields mapped in them, or use the special `@JsonProperty` Java annotation.

Figure 4.4 Class diagram in the `manta-connector-matillion-model` submodule

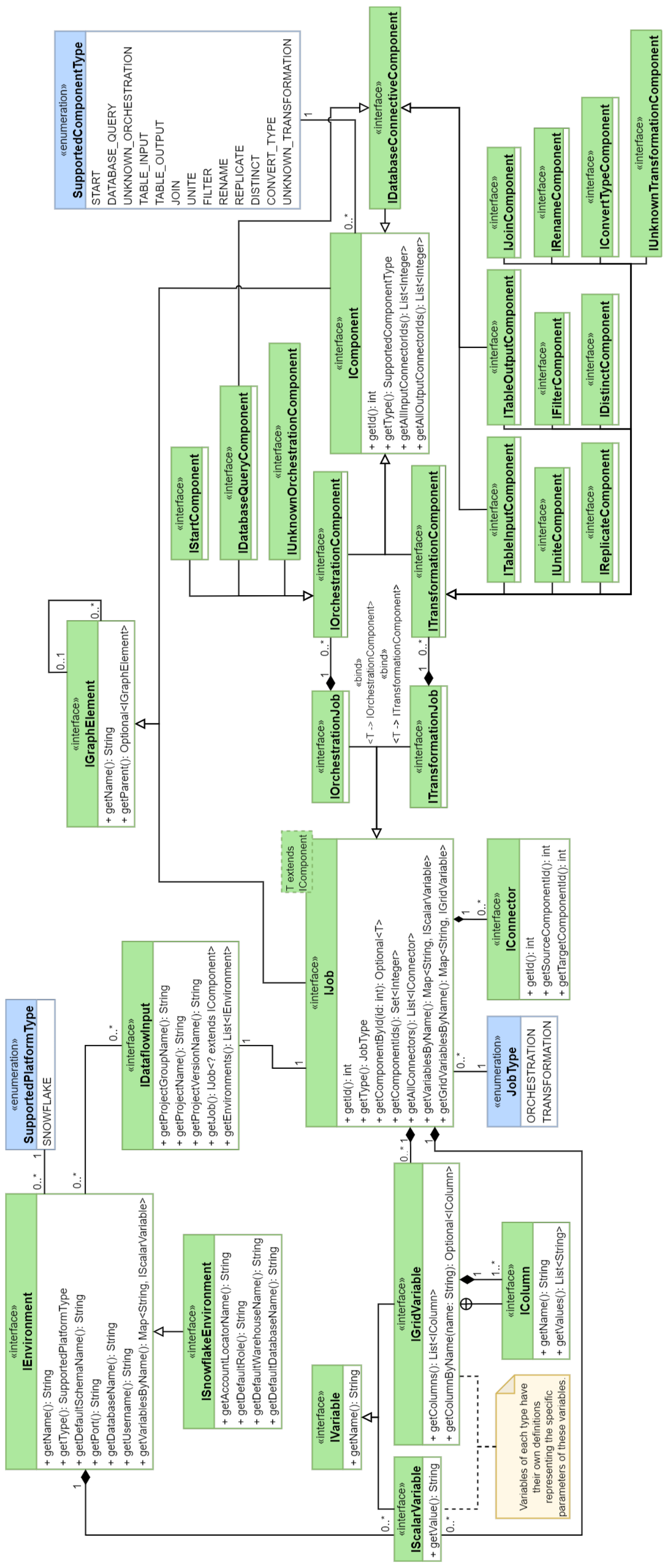
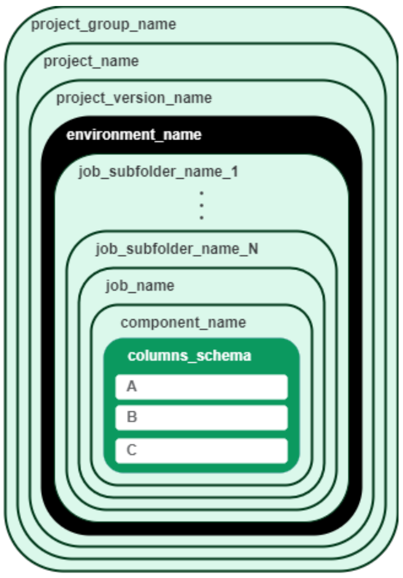

«enumeration» SupportedComponentType
START
DATABASE_QUERY
UNKNOWN_ORCHESTRATION
TABLE_INPUT
TABLE_OUTPUT
JOIN
UNITE
FILTER
RENAME
REPLICATE
DISTINCT
CONVERT_TYPE
UNKNOWN_TRANSFORMATION
«interface» IDatabaseConnectiveComponent
«interface» IComponent
+ getId(): int
+ getType(): SupportedComponentType
+ getAllInputConnectorIds(): List<Integer>
+ getAllOutputConnectorIds(): List<Integer>
«interface» IJoinComponent
«interface» IRenameComponent
«interface» IConvertTypeComponent
«interface» IUnknownTransformationComponent
«interface» ITableOutputComponent
«interface» IFilterComponent
«interface» IDistinctComponent
«interface» ITableInputComponent
«interface» IUniteComponent
«interface» IReplicateComponent
«interface» IStartComponent
«interface» IDatabaseQueryComponent
«interface» IUnknownOrchestrationComponent
«interface» IOrchestrationComponent
«interface» ITransformationComponent
«interface» IGraphElement
+ getName(): String
+ getParent(): Optional<IGraphElement>
0..1
0..*
«interface» IOrchestrationJob
«interface» ITransformationJob
«bind» <T -> IOrchestrationComponent>
«bind» <T -> ITransformationComponent>
T extends IComponent
«interface» IJob
+ getId(): int
+ getType(): JobType
+ getComponentById(id: int): Optional<T>
+ getComponentIds(): Set<Integer>
+ getAllConnectors(): List<IConnector>
+ getVariablesByName(): Map<String, IScalarVariable>
+ getGridVariablesByName(): Map<String, IGridVariable>
«interface» IConnector
+ getId(): int
+ getSourceComponentId(): int
+ getTargetComponentId(): int
«enumeration» JobType
ORCHESTRATION
TRANSFORMATION
«enumeration» SupportedPlatformType
SNOWFLAKE
«interface» IDataflowInput
+ getProjectGroupName(): String
+ getProjectName(): String
+ getProjectVersionName(): String
+ getJob(): IJob<? extends IComponent>
+ getEnvironments(): List<IEnvironment>
«interface» IEnvironment
+ getName(): String
+ getType(): SupportedPlatformType
+ getDefaultSchemaName(): String
+ getPort(): String
+ getDatabaseName(): String
+ getUsername(): String
+ getVariablesByName(): Map<String, IScalarVariable>
«interface» ISnowflakeEnvironment
+ getAccountLocatorName(): String
+ getDefaultRole(): String
+ getDefaultWarehouseName(): String
+ getDefaultDatabaseName(): String
«interface» IVariable
+ getName(): String
«interface» IGridVariable
+ getColumns(): List<IColumn>
+ getColumnByName(name: String): Optional<IColumn>
«interface» IColumn
+ getName(): String
+ getValues(): List<String>
1..*
«interface» IScalarVariable
+ getValue(): String
Variables of each type have their own definitions representing the specific parameters of these variables.

Figure 4.5 Data flow graph nodes structure



4.2 Data Flow Generator module

The purpose of the Data Flow Generator module will be to perform data flow analysis of input metadata represented by the internal model. The module will work with interfaces declared in the Connector Model submodule, without dealing with their concrete implementations, which creates a high level of abstraction and makes the processes of filling the model and working with it independent of each other. Based on the analysis, the result data flow graph will be generated in this module.

Before studying the key algorithms of the module, it is necessary to familiarize yourself with the structure of graph nodes, which is the subject of the next section.

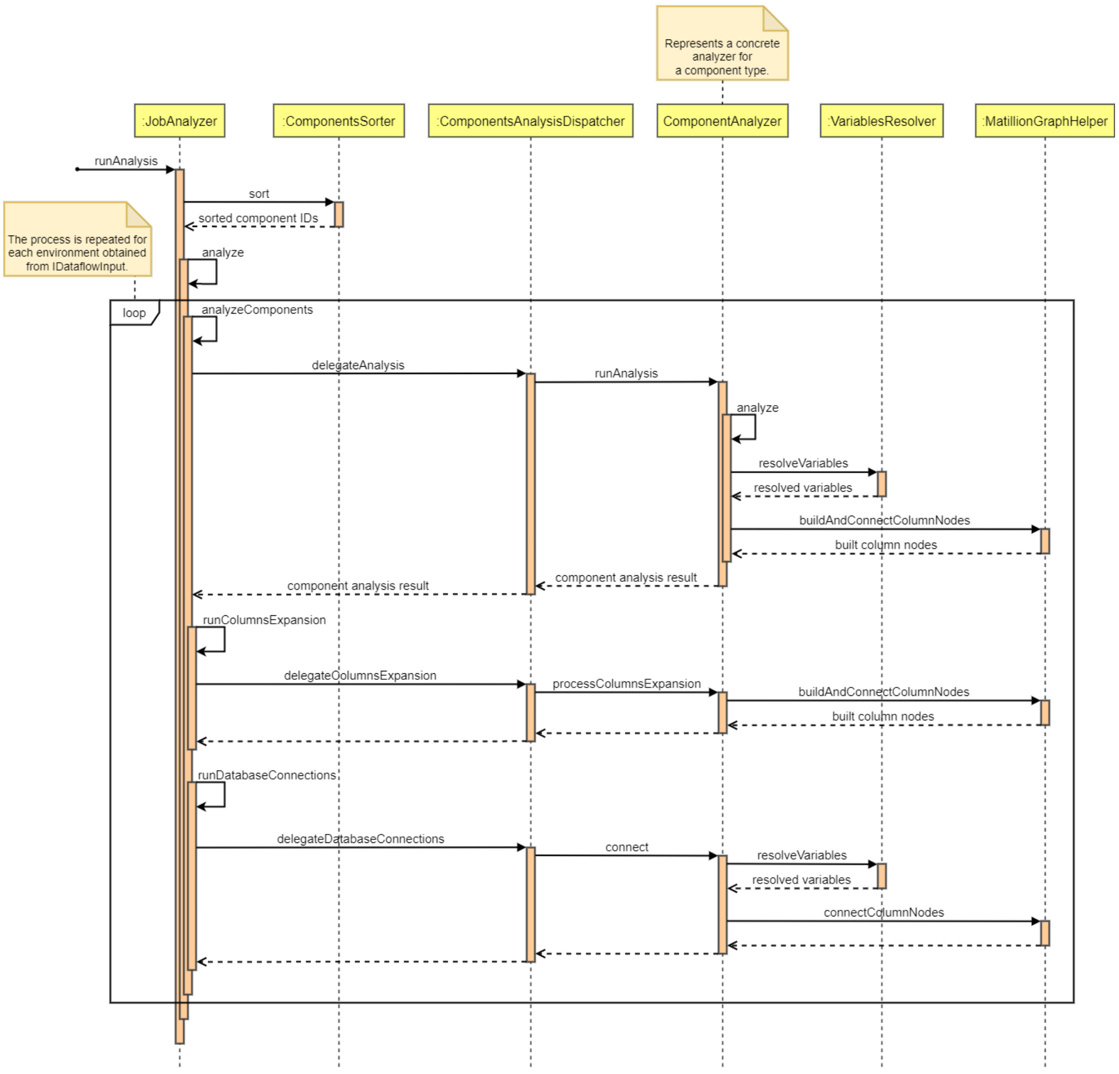
4.2.1 Data flow graph nodes structure

The nodes structure of the generated data flow graph is specific to each Manta scanner, since each of them works with different key objects. The structure created for the Matillion scanner can be seen in Figure 4.5. Each node has a type and is characterized by the name of the object it represents, and may have a corresponding set of attributes that describe that object.

The three root nodes generated by the Matillion scanner represent, respectively, a group of projects, a project that belongs to this group, and its version. Next comes the environment node, which describes the environment, within which the job was analyzed. The environment node has an attribute indicating the type of target platform inherent in this environment, for example, "Snowflake". A job also has its own node, which is preceded by a hierarchy of nodes representing the internal folder structure of the tool in which the given job is located. The component node has many attributes that basically describe the configurable properties of that component. At the end of the hierarchy are nodes that describe the individual columns that the corresponding component works with. These columns could have been created in this component, or extracted from some data source and passed to this component by the previous ones. To group all column nodes, the parent node with the constant name `columns_schema` was created.

As a result, such a structure of nodes is able to describe all the key elements in Matillion and their relationship in the context of scanner analysis.

Figure 4.6 Sequence diagram for the Data Flow Generator module



4.2.2 Data flow analysis and graph generation

Data flow analysis of a single job that was passed through the `IDataflowInput` interface starts an object of the `JobAnalyzer` class via the method called `runAnalysis`, which can be seen in the sequence diagram in Figure 4.6. Job analysis consists in analyzing its individual components and studying the flow of columns within them. The problem is that the received metadata about each component does not contain information about all the columns flowing through it, but only about those that the component somehow works with (for example, creates or modifies them). Therefore, the components should be analyzed in topological order, passing known columns to each other. This process has been named “columns propagation”.

Objects of classes that extend the abstract class `AbstractComponentAnalyzer` are responsible for analyzing components of concrete types, since analysis is always type-specific. For example, `JoinComponentAnalyzer` analyzes components of type “Join”, etc. Analyzers examine the influence of their components on the columns flowing through them, and based on this gradually generate a data flow graph. For easier graph manipulation, the `MatillionGraphHelper` instance is used, which has a variety of methods that allows to create and connect nodes. It is also necessary to keep in mind the variables of several types that can be used in the components configuration. The `VariablesResolver` instance serves to resolve them.

The results of component analysis are stored in objects of class `ComponentAnalysisResult`. They contain information about the component columns, whether the analysis was successful, and other related values. Columns propagation works by passing the analysis results of previous components to the analyzers of the following ones.

4.2.2.1 Columns expansion

As mentioned earlier, there is a problem with incorrectly configured jobs, their component properties, environments, and other elements passed for data flow analysis. The columns expansion mechanism is intended to solve this problem.

The mechanism starts after the columns propagation phase, when all the information that could be obtained from the input metadata has been processed, and based on this, a data flow graph has been generated. At the same time, the results of the analysis of all components were returned, containing the statuses of whether they were successful. The analysis of a component is considered unsuccessful if its analyzer noticed its incorrect configuration, or the analysis of the previous component was unsuccessful. If so, this means that there is a possibility that part of the lineage is missing from the data flow graph.

Unlike the propagation mechanism, the expansion mechanism starts from the last component in the topological sort and ends with the first one. It works with `columnExpansionMappings` saved in every unsuccessfully analyzed component result. These mappings are processed by concrete component analyzers and contain information about the columns to be expanded, namely the names of their graph nodes to be created and the already existing nodes to which they should be connected.

As a result of this “deduction”, the output graph is filled with possibly missing nodes.

4.2.2.2 Connection to databases

The last step in data flow analysis, which goes after creating all possible columns, is connecting to databases and creating their special nodes in the data flow graph. This is achieved by calling the `connect` method of the database connective component analyzers, which, in turn, call the methods of the internal Manta tool called `DataflowQueryService`. This tool is able to create nodes for columns that come from databases based on the provided connection properties, SQL query, or other parameters known to the scanner from the input metadata. After the nodes are created, the scanner can ask the tool to merge them with the output graph.

Chapter 5

Matillion scanner prototype implementation

The purpose of this chapter is to present the implementation of the Matillion scanner prototype based on the design discussed in the previous chapter. Due to the huge amount of source code, the focus will be on only the essential and most interesting parts of the Connector and Data Flow Generator modules.

5.1 Connector module

As already known, the first step when working with manually provided JSON files is to load ("extract") them from the input directory, split into smaller parts, and store them in a temporary directory, which is done in the Extractor submodule.

Extraction of the input files is done within the `extract` method of the `ManualInputExtractor` class object, as shown in Listing 1. The method creates an instance of the `ManualInputReader` class, which is responsible for loading files from the input directory. This class extends the Manta's `AbstractFileInputReader`, which declares methods for input files processing, such as `canRead` and `read`. For them to work correctly for the purposes of the Matillion scanner, the `ManualInputReader` takes the path to the input directory as a constructor parameter and specifies the characteristics of the files of this directory that should be read, such as their extension type being JSON. Each input file (its `java.io.File` representation) is then stored in an instance of `ManualInputFileHolder` that gets the necessary supporting information from the file path and throws the `WrongManualInputFilePathException` in case of incorrect file placement by the user.

The input file processing is then continues in the split phase. An example of the split can be found in Listing 2 that introduces the `readEnvironmentContent` method of the `AbstractJsonSplitter` class. This method was created in order to provide the splitting of the `variables` and `environments` lists of JSON objects of the input file into separate files called `environmentVariablesDefinition.json` and `environments.json` respectively. The method works with `JsonParser` and `JsonGenerator` instances from the Jackson Streaming API, which are respectively used to iterate over the JSON content and copy and write it to the `ByteArrayOutputStream`. Splitting of the jobs JSON content in general works in a similar way, but with some additional functionality not relevant to this section.

Finally, the split parts are written to the output files in a temporary directory, an example of which can be seen in Listing 3. Note that this step may fail in case of any error related to working with the file system, such as, for example, access denial or a file that already exists in the desired path.

```java
@Override
public boolean extract(SupportedPlatformType platformType, OutputFileWriter outputFileWriter) {

    LOGGER.info("Start of manual input extraction.");

    boolean isExtracted = false;
    try (ManualInputReader inputReader = new ManualInputReader(inputDirectory)) {

        File currentInputFile = null;
        while (inputReader.canRead()) {

            try {
                currentInputFile = inputReader.read();
                ManualInputFileHolder inputFileHolder =
                        new ManualInputFileHolder(currentInputFile, serverName);
                LOGGER.info(
                        "Start of manual input file extraction: [{}].",
                        inputFileHolder.getFilePath()
                );
                if (!isExtracted) {
                    isExtracted = runSplit(platformType, inputFileHolder, outputFileWriter);
                    continue;
                }
                runSplit(platformType, inputFileHolder, outputFileWriter);

            } catch (IOException | IllegalStateException e) {
                LOGGER.log(Categories
                        .extractionErrors()
                        .failedToExtractManualInputFile()
                        .filePath(inputReader.getInputName())
                        .catching(e)
                );
            } catch (WrongManualInputFilePathException e) {
                LOGGER.log(Categories
                        .extractionErrors()
                        .wrongManualInputFilePath()
                        .filePath(currentInputFile.getPath())
                );
            }
        }
    }
    return isExtracted;
}
```

Code listing 1 The `extract` method in the `ManualInputExtractor` class

```java
private void readEnvironmentContent(JsonParser jsonParser,
                                    OutputFileWriter outputFileWriter,
                                    String projectGroupName,
                                    String projectName,
                                    String projectVersionName,
                                    String outputEnvironmentFileName) throws IOException {

    // Check that the cursor points to the JSON start array "[" token
    if (jsonParser.getCurrentToken() != JsonToken.START_ARRAY) {
        throw new IllegalStateException();
    }

    ByteArrayOutputStream byteArrayOutputStream = new ByteArrayOutputStream();
    try (JsonGenerator jsonGenerator = createJsonGenerator(byteArrayOutputStream)) {

        jsonGenerator.copyCurrentEvent(jsonParser);
        while (jsonParser.nextToken() != JsonToken.END_ARRAY) {

            if (jsonParser.getCurrentToken() != JsonToken.START_OBJECT) {
                throw new IllegalStateException();
            }
            copySingleObject(jsonParser, jsonGenerator);
        }
        jsonGenerator.copyCurrentEvent(jsonParser);
    }

    // In order not to create an environment file with empty content "[]"
    if (byteArrayOutputStream.size() != 2) {
        outputFileWriter.writeToEnvironmentFile(
                projectGroupName,
                projectName,
                projectVersionName,
                outputEnvironmentFileName,
                SupportedFileExtension.JSON,
                byteArrayOutputStream
        );
    }
}
```

Code listing 2 The `readEnvironmentContent` method in the `AbstractJsonSplitter` class

```java
public void writeToEnvironmentFile(String projectGroupName,
                                   String projectName,
                                   String projectVersionName,
                                   String fileName,
                                   SupportedFileExtension fileExtension,
                                   ByteArrayOutputStream byteArrayOutputStream) {

    Optional<File> optionalEnvironmentFile = createFile(
            Path.of(
                    outputDirectory.getAbsolutePath(),
                    FileNames.GROUPS_DIRECTORY_NAME,
                    projectGroupName,
                    projectName,
                    projectVersionName,
                    fileName + fileExtension.getName()
            )
    );
    optionalEnvironmentFile.ifPresent(file -> writeToFile(file, byteArrayOutputStream));
}
```

Code listing 3 The `writeToEnvironmentFile` method in the `OutputFileWriter` class

```
@Override
public IJob<? extends IComponent> read(File jobFile) throws IOException {

    try (JsonParser jsonParser = createJsonParser(jobFile)) {

        // Check that the next token the cursor points to is the JSON start object "{"
        if (jsonParser.nextToken() != JsonToken.START_OBJECT) {
            throw new IllegalStateException();
        }

        JobType jobType = getJobType(jobFile.getParentFile().getName());
        switch (jobType) {
            case ORCHESTRATION:
                return getObjectMapper().readValue(jsonParser, OrchestrationJob.class);
            case TRANSFORMATION:
                return getObjectMapper().readValue(jsonParser, TransformationJob.class);
            default:
                throw new IllegalArgumentException(
                    String.format("Unknown job type: [%s].", jobType)
                );
        }

    } catch (IllegalStateException | JsonMappingException e) {
        LOGGER.log(Categories
                .inputStructureErrors()
                .wrongJobFileJsonStructure()
                .filePath(jobFile.getPath())
                .catching(e)
        );
        throw e;
    } catch (IOException e) {
        LOGGER.log(CommonErrors
                .IOErrors()
                .cannotReadFile()
                .file(jobFile.getPath())
                .catching(e)
        );
        throw e;
    }
}
```

Code listing 4 The `read` method in the `JsonJobReader` class

Listing 4 presents the `read` method of the `JsonJobReader` class, located in the Reader submodule. The method takes the job JSON file that appeared in the temporary directory after the split process, represented by an instance of `java.io.File`, and returns the internal object representation of this job. To do this, the `readValues` method of the `ObjectMapper` class is called, which is able to work with Jackon's `JsonParser` that provides access to the JSON content of the job file. However, this method must know exactly the desired class of the object being created, and takes this information as its second argument.

The Resolver submodule contains basically only plain Java classes that implement the interfaces declared in the Model submodule. These classes are primarily instantiated by the `ObjectMapper` instance in the Reader submodule, that is why they contain attributes whose names match the JSON fields in the input files or are marked with the `@JsonProperty` Java annotation. The last important thing to say is that the internal model cannot be changed after it has been created, that is, the interfaces in Model do not provide any setter method that can be used during data flow analysis, and the return values of its getter methods are immutable.

```
public Node buildJobNode(IJob<? extends IComponent> job, Node parentNode) {

    Node parentJobFolderNode = parentNode;
    for (String jobFolder : job.getPath()) {
        parentJobFolderNode = addNode(jobFolder, NodeType.MATILLION_JOB_FOLDER, parentJobFolderNode);
    }
    Node jobNode = addNode(job.getName(), job.getNodeType(), parentJobFolderNode);
    if (StringUtils.isNotBlank(job.getDescription())) {
        jobNode.addAttribute(NodeAttributesNames.DESCRIPTION, job.getDescription());
    }
    return jobNode;
}
```

Code listing 5 The `buildJobNode` method in the `MatillionGraphHelper` class

5.2 Data Flow Generator module

One of the most significant classes in the Data Flow Generator module is called `MatillionGraphHelper`. An object of this class manipulates the output flow graph using various methods for creating and connecting nodes. The class extends `AbstractGraphHelper` provided by Manta. The method used to create a structure of nodes, relevant to the analyzed job, can be seen in Listing 5. Each graph node can be created via the inherited `addNode` method that takes the name of the node to be created, its type, and a reference to the parent node as arguments. After that, if necessary, numerous attributes can be added to the created node object using its `addAttribute` method, to which the attribute name and value must be passed. The last two inherited methods of the `MatillionGraphHelper` class, which should be introduced in this section, are called `addDirectFlow` and `addFilterFlow`. These methods take the source and target nodes as arguments and connect them using the corresponding edge type.

The core data flow analysis logic resides in component analyzers. Listing 6 introduces the `analyze` method of the `ConvertTypeComponentAnalyzer` as an example. The "Convert Type" component does nothing else but convert type of columns flowing through it. This means that the appropriate attribute should be added to the node of column whose type has changed. Besides, the column nodes of the preceding components should be copied into the columns schema relevant to the component being analyzed and connected with edges of direct flow type.

During analysis, the component analyzer collects the names of columns that may be expanded. As already mentioned in 4.2.2, the received metadata about each component does not contain information about all the columns flowing through it, but only about those that the component somehow works with. That is why the `IConvertTypeComponent` interface provides only a single method that enables to get all the columns whose type has changed, along with the corresponding type conversion in form of a string, suitable to be added as an attribute of the column node. In the given example, if the analyzer has not found the name of such column using columns propagation, it marks this column name as a candidate for expansion and fills the `columnExpansionMappings` if the analysis result of the given component was not successful.

It is important to mention that component analyzers make extensive use of the `VariablesResolver` instance to resolve variables of different types. Its method called `resolveAllScalarVariables` is shown in Listing 7 as an example. The method takes a string with scalar variable declarations as a parameter and returns the same string, but with the declarations replaced by their actual values.

Another important thing to introduce is the way to connect to databases in database connective component analyzers. They all use an instance of a helper class named `DatabaseConnector`, which works directly with `DataflowQueryService` and has many methods for processing database nodes. However, to establish a connection, the analyzer must first create an instance of a class that implements the `Connection` interface using known credentials, as shown in Listing 8.

```java
@Override
protected ComponentAnalysisResult<IConvertTypeComponent> analyze(
    IConvertTypeComponent component,
    Node componentNode,
    IEnvironment environment,
    List<ComponentAnalysisResult<? extends IComponent>> previousComponentAnalysisResults,
    VariablesResolver variablesResolver,
    MatillionGraphHelper graphHelper
) {
    Node columnsSchemaNode = graphHelper.buildColumnsSchemaNode(componentNode);
    ComponentAnalysisResult<IConvertTypeComponent> result = createEmptyAnalysisResult(
            component,
            componentNode,
            previousComponentAnalysisResults,
            1,
            columnsSchemaNode,
            variablesResolver
    );

    Map<String, String> columnTypeConversionsByColumnName =
            getColumnTypeConversionsByColumnName(
                    component.getColumnTypeConversionsHolder(), variablesResolver
            );
    Set<String> possibleColumnNamesToExpand =
        new HashSet<>(columnTypeConversionsByColumnName.keySet());
    for (ComponentAnalysisResult<? extends IComponent> previousComponentAnalysisResult :
            previousComponentAnalysisResults) {

        if (!previousComponentAnalysisResult.isSuccess()) {
            result.setIsSuccess(false);
        }
        for (Node sourceColumnNode : previousComponentAnalysisResult.getColumnNodes()) {

            Node columnNode = graphHelper.copyAndConnectSourceColumnNode(
                    sourceColumnNode,
                    columnsSchemaNode,
                    TransformationClassification.NON_TRANSFORMING
            );
            String columnName = columnNode.getName();
            if (columnTypeConversionsByColumnName.containsKey(columnName)) {
                columnNode.addAttribute(
                        NodeAttributesNames.TYPE_CONVERSION,
                        columnTypeConversionsByColumnName.get(columnName)
                );
                possibleColumnNamesToExpand.remove(columnName);
            }
            result.addColumnNode(columnNode);
        }
    }

    processPossibleColumnExpansionMappings(
            possibleColumnNamesToExpand, columnsSchemaNode, result, graphHelper
    );

    checkEmptyColumnNodes(result);
    return result;
}
```

Code listing 6 The `analyze` method in the `ConvertTypeComponentAnalyzer` class

```java
public String resolveAllScalarVariables(String stringWithScalarVariableDeclarations,
                                        int componentId,
                                        String componentName) {

    // Scalar variable declaration must be in the ${variableName} format
    // with certain additional rules
    Matcher matcher = Pattern.compile("\\$\\{([A-Za-z_$][A-Za-z0-9_$]*)}")
        .matcher(stringWithScalarVariableDeclarations);
    StringBuilder stringBuilder = new StringBuilder();
    while (matcher.find()) {

        String scalarVariableName = matcher.group(1);
        Optional<String> optionalScalarVariableValue =
                getScalarVariableValue(scalarVariableName, componentId, componentName);
        if (optionalScalarVariableValue.isPresent()) {
            matcher.appendReplacement(stringBuilder, optionalScalarVariableValue.get());
        }
        else {
            LOGGER.log(Categories
                    .variableResolvingErrors()
                    .failedToResolveScalarVariable()
                    .scalarVariableName(scalarVariableName)
            );
        }
    }
    matcher.appendTail(stringBuilder);
    return stringBuilder.toString();
}
```

Code listing 7 The `resolveAllScalarVariables` method in the `VariablesResolver` class

```java
public Connection createSnowflakePlatformConnection(String accountLocatorName,
                                                    String warehouseName,
                                                    String databaseName,
                                                    String schemaName,
                                                    String username) {

    String serverName = Objects.nonNull(accountLocatorName) && Objects.nonNull(warehouseName) ?
        String.format(
                "jdbc:snowflake://%s.snowflakecomputing.com/?warehouse=%s",
                accountLocatorName,
                warehouseName
        ) : null;
    return new ConnectionImpl(
            getPlatformTypeName(SupportedPlatformType.SNOWFLAKE),
            accountLocatorName,
            serverName,
            databaseName,
            schemaName,
            username
    );
}
```

Code listing 8 The `createSnowflakePlatformConnection` method in the `DatabaseConnector` class

Chapter 6

Matillion scanner prototype testing

In this chapter, the way of testing the implemented Matillion scanner prototype will be introduced, mainly focusing on key test cases. The chapter first presents automated tests of the Connector and Data Flow Generator modules, and then continues with result data flow visualization examples.

6.1 Connector module

To present the Connector module automated JUnit tests, it is better to start with those that are located in the Extractor submodule, which handles manual input files. Listing 9 shows a test that covers the entire process of "extracting" files, that is, reading, splitting, and saving parts of it to the output temporary directory. The test first calls the `extract` method of the `ManualInputExtractor`, which performs the mentioned steps using predefined manual input and temporary output directory paths. After that, the test compares the pre-filled expected output temporary directory with the actual one that appeared during the extraction process, iterating over each file and its contents.

Another noteworthy test of the Extractor submodule is shown in Listing 10. This test checks the correctness of the `ManualInputFileHolder` class logic by asserting the expected attribute values of its object. The test class, however, also has another methods that mainly check for `WrongManualInputFilePathException` to be thrown if the input file is not placed correctly by the user.

The tests located in the Reader submodule serve to verify the correct reading of split files and filling in the internal model in the Resolver submodule. For example, the `JsonJobReaderTest` class contains methods for testing the `read` method of the `JsonJobReader` class by calling it and asserting the expected properties of read jobs of both orchestration and transformation types. The `JsonEnvironmentsReaderTest` and `JsonEnvironmentVariablesDefinitionReaderTest` tests work similarly. Special mention deserves a large number of component tests located along the path `src/test/java/eu/profinit/manta/connector/matillion/component`. These tests were created for each component type separately in order to verify their resolved unique properties.

To summarize, the Connector module was tested with a total of 40 passed unit tests.

```
@Test
void extractTest() {

    LOGGER.debug("Extracting manual input files.");
    manualInputExtractor.extract(platformType, outputFileWriter);

    Path expectedOutputDirectory = Path.of(System.getProperty("user.dir"))
            .resolve("src/test/resources/expected/output/" + serverName);
    try {
        Files.walkFileTree(expectedOutputDirectory, new SimpleFileVisitor<>() {

            @Override
            public FileVisitResult visitFile(Path expectedFilePath,
                                             BasicFileAttributes attrs)
                    throws IOException {

                if (!Files.isDirectory(expectedFilePath)) {
                    assertTrue(expectedFileTest(expectedFilePath));
                }
                return FileVisitResult.CONTINUE;
            }
        });

    } catch (IOException e) {
        fail(e);
    }
}
```

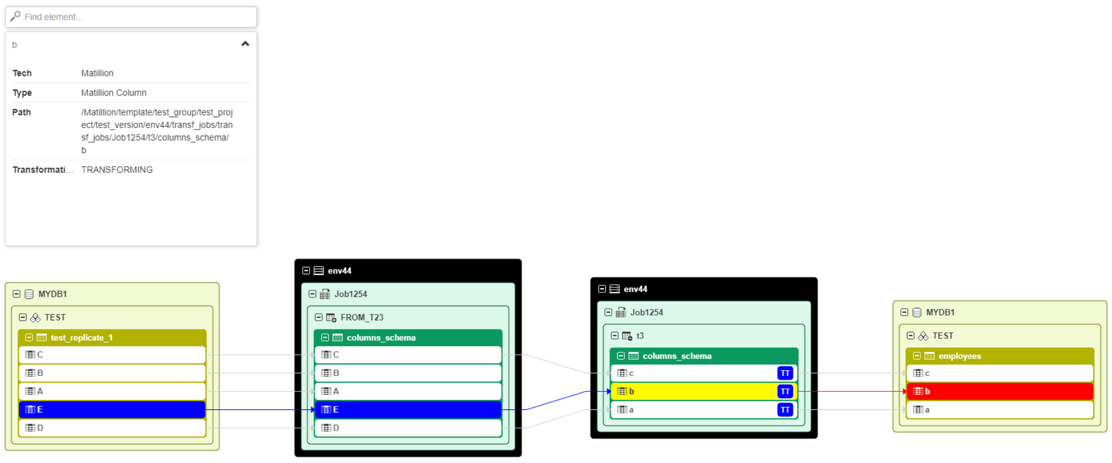
Code listing 9 The `extractTest` method in the `ManualInputExtractorTest` class

```
@Test
void jsonSuccessTest() {

    try {
        File inputFile = new File("server/group/project/version/export.json");
        ManualInputFileHolder manualInputFileHolder =
                new ManualInputFileHolder(inputFile, "server");

        assertEquals(inputFile, manualInputFileHolder.getFile());
        assertEquals("export.json", manualInputFileHolder.getFileName());
        assertEquals(
            SupportedFileExtension.JSON, manualInputFileHolder.getFileExtension()
        );
        assertEquals("group", manualInputFileHolder.getProjectGroupName());
        assertEquals("project", manualInputFileHolder.getProjectName());
        assertEquals("version", manualInputFileHolder.getProjectVersionName());

    } catch (WrongManualInputFilePathException e) {
        fail(e);
    }
}
```

Code listing 10 The `jsonSuccessTest` method in the `ManualInputFileHolderTest` class

```
@ParameterizedTest(name = "{index}: sourceFile: {0}")
@MethodSource("collectResolvedDataFlowInputs")
void compareExpectedGraphToActual(Path expectedGraphFilePath,
                                  IDataflowInput dataFlowInput) throws IOException {

    Graph outputGraph = new GraphImpl(null);
    dataFlowTask.execute(dataFlowInput, outputGraph);
    GraphEqualityUtil.checkGraphsEqual(
            outputGraph, getExpectedGraphFile(expectedGraphFilePath), TestMode.COMPARE
    );
}
```

Code listing 11 The `compareExpectedGraphToActual` method in the `DataflowEqualityTest` class


Find element...

b

Tech	Matillion
Type	Matillion Column
Path	/Matillion/template/test_group/test_project/test_version/env44/transf_jobs/transf_jobs/Job1254/t3/columns_schema/b
Transformati...	TRANSFORMING

Figure 6.1 Result data flow graph example

6.2 Data Flow Generator module

The Data Flow Generator module is mainly tested using the common Manta utility for testing the output data flow graph named `GraphEqualityUtil`. The utility provides a single method called `checkGraphsEqual` that takes the generated output graph as the first argument, writes it to a plain file of a special format, and possibly compares this file with the expected graph file passed as the second argument. The use of this utility can be seen in Listing 11, which shows a method that executes data flow analysis and is called several times with different data flow inputs. In general, this way of testing allows to add as many data flow inputs as necessary, covering each analysis case.

The module also contains the `VariablesResolverTest` class, which verifies that variables of each type are resolved correctly, primarily by mocking their properties.

The Data Flow Generator module was tested using 29 unit tests in total.

6.3 Data flow visualization

The essential test, which shows that the scanner prototype actually works and is successfully integrated into the Manta project, is the simple resulting visualization example presented above in Figure 6.1. Note that more such examples can be found below in the appendix named **"More Matillion ETL data flow visualization examples"**.

Chapter 7
Conclusion

The main goal of this thesis was to design and implement a prototype of a functional module for the Manta project, which performs analysis of data flows in the Matillion ETL tool based on exported metadata describing tool elements. After analysis, the implemented prototype generates a graph that visualizes these data flows. To achieve this goal, the thesis was divided into several logical parts that were preceded by a description of the basic concepts and technologies that the work operates on.

The first part (chapter “Analysis”) was devoted to the analysis of Matillion ETL, getting acquainted with its key elements and ways to export metadata, as well as its characteristics. Within this part were also defined the requirements for the implemented prototype and the functionality it should support.

In the second part (chapter “Matillion scanner design”), the design of the solution was presented, which included a description of the two modules of the implemented prototype and key algorithms for metadata processing, data flow analysis, and graph generation. The design was made based on the previous analysis and ensures a seamless connection of the implemented module to the Manta project, following its existing infrastructure.

The third part (chapters “Matillion scanner prototype implementation” and “Matillion scanner prototype testing”) introduced the implementation of the prototype based on its design and the way of testing it. The result module was also qualitatively documented using Javadoc, fulfilling the subgoals of this thesis.

As expected, the current state of the prototype does not allow it to be released as part of the Manta software due to its limited functionality. However, in the course of working on this thesis, a good foundation was created that makes the prototype easily extensible.

Appendix A

More Matillion ETL data flow visualization examples

Figure A.1 Matillion ETL data flow visualization example No. 1

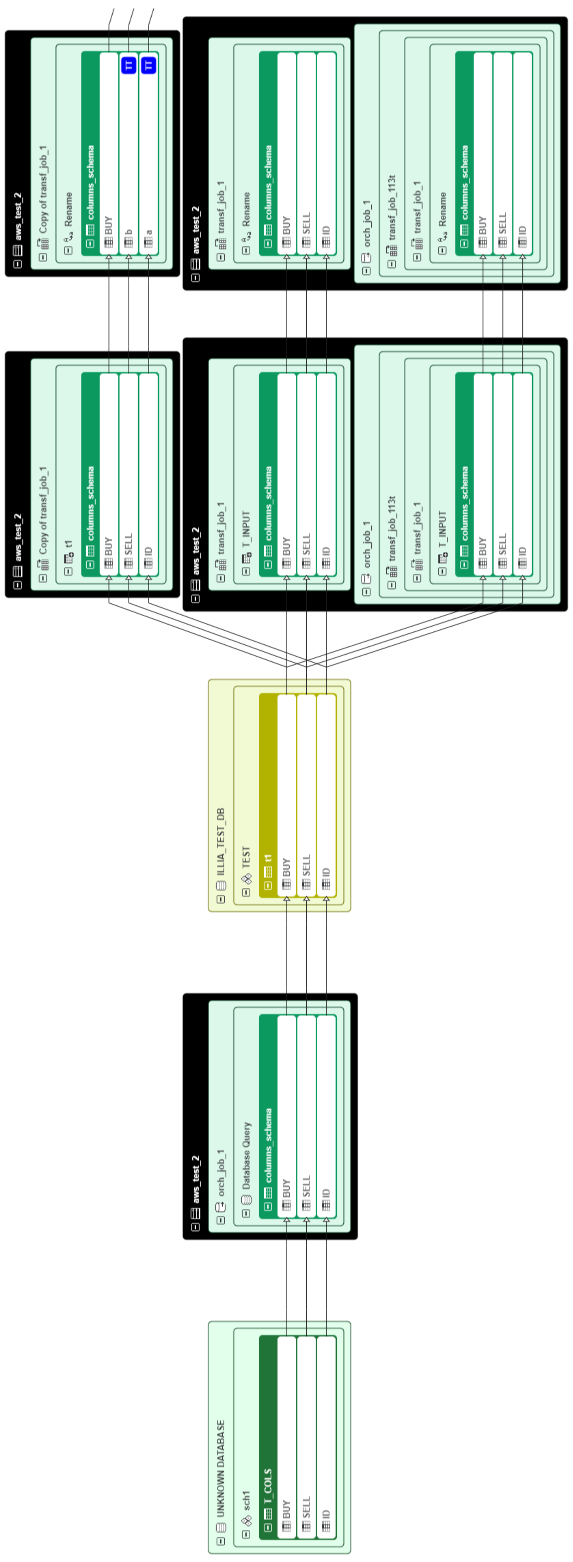


Figure A.2 Matillion ETL data flow visualization example No. 2

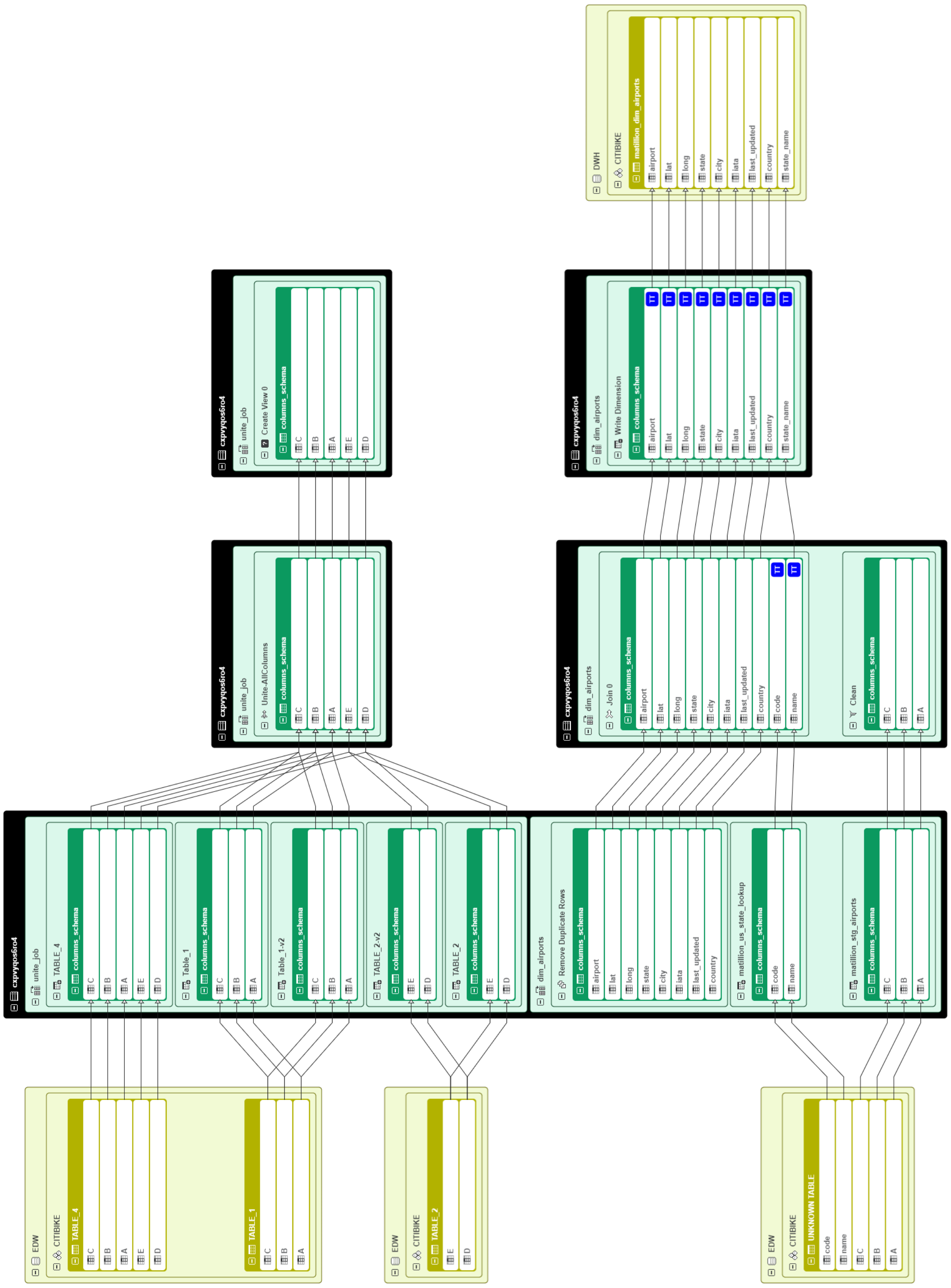
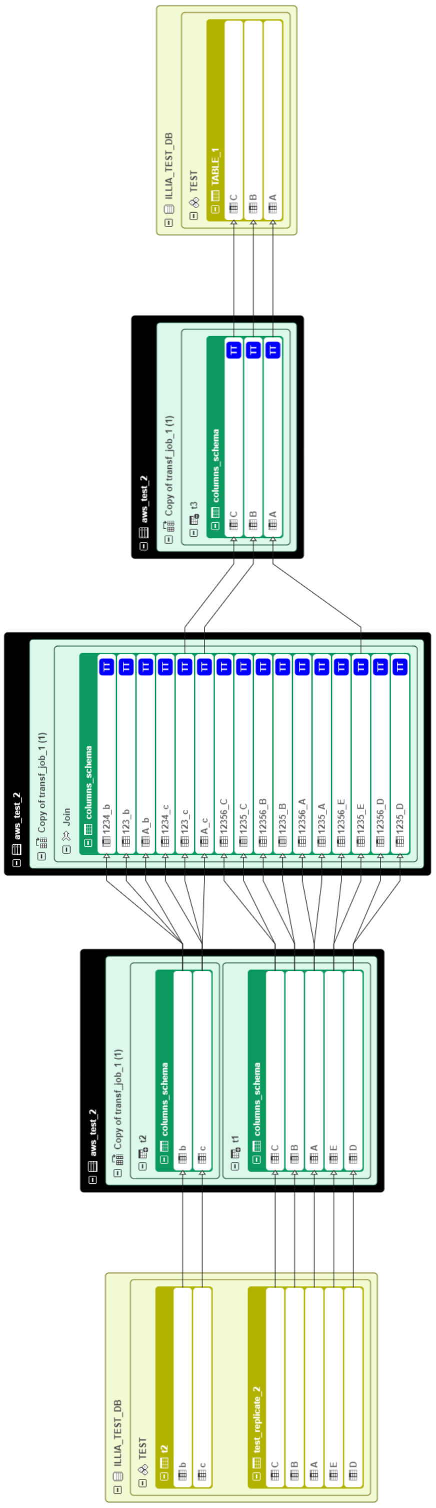


Figure A.3 Matillion ETL data flow visualization example No. 3



Bibliography

1. PETERMAN, Mark. *What is ETL and why do I need it?* [online]. [N.d.]. [visited on 2022-11-25]. Available from: `https://blog.csgsolutions.com/what-is-etl`.
2. *What is Dataflow?* [online]. techopedia, 2017 [visited on 2022-11-22]. Available from: `https://www.techopedia.com/definition/6743/dataflow`.
3. *What is Data Lineage?* [online]. imperva, 2022 [visited on 2022-11-22]. Available from: `https://www.imperva.com/learn/data-security/data-lineage/`.
4. REESE, Eric. *All You Need to Know About Metadata* [online]. opendatasoft, 2020 [visited on 2022-11-22]. Available from: `https://www.opendatasoft.com/en/blog/all-you-need-to-know-about-metadata`.
5. *Manta (firma)* [online]. Wikimedia Foundation, 2021 [visited on 2022-11-22]. Available from: `https://cs.wikipedia.org/wiki/MANTA_(firma)`.
6. HERMANN, Lukáš; ULRYCH, Jan; MORAVEC, Jakub; HOLLINGER, Rebecca. *Manta Flow Architecture* [online]. Manta Confluence (internal documentation), 2022 [visited on 2022-11-22]. Available from: `https://mantatools.atlassian.net/wiki/spaces/MTKB/pages/70230122/MANTA+Flow+Architecture`.
7. KOŠVANEC, Petr. *Analýza datových toků v reportovacích nástrojích*. 2019. MA thesis. Faculty of Information Technology, Czech Technical University in Prague.
8. *Data Lineage Done Right* [online]. 2022. [visited on 2022-11-22]. Available from: `https://getmanta.com/`.
9. *What is a Data Warehouse?* [online]. IBM, 2020 [visited on 2022-11-23]. Available from: `https://www.ibm.com/cloud/learn/data-warehouse`.
10. *Cloud data warehouses* [online]. [N.d.]. [visited on 2022-11-23]. Available from: `https://documentation.matillion.com/docs/2852129`.
11. *What is a Data Warehouse?* [online]. 2022. [visited on 2022-11-23]. Available from: `https://www.snowflake.com/data-cloud-glossary/data-warehousing/`.
12. *What is ETL (Extract, Transform, Load)?* [online]. [N.d.]. [visited on 2022-11-23]. Available from: `https://www.snowflake.com/guides/what-etl`.
13. *What is ETL (Extract, Transform, Load)?* [online]. IBM, 2020 [visited on 2022-11-23]. Available from: `https://www.ibm.com/cloud/learn/etl`.
14. RANAWAKA, Malsha. *ETL vs ELT: The Difference is in the How* [online]. Panoply, 2022 [visited on 2022-11-23]. Available from: `https://blog.panoply.io/etl-vs-elt-the-difference-is-in-the-how`.

15. *Matillion ETL Product Overview* [online]. [N.d.]. [visited on 2022-11-23]. Available from: `https://documentation.matillion.com/docs/1975061`.
16. *Matillion ETL Releases Overview* [online]. [N.d.]. [visited on 2022-11-23]. Available from: `https://documentation.matillion.com/docs/1799483`.
17. *What is SQL?* [online]. Amazon, [n.d.] [visited on 2022-11-23]. Available from: `https://aws.amazon.com/what-is/sql/`.
18. *Object-oriented programming* [online]. Wikimedia Foundation, 2022 [visited on 2022-11-23]. Available from: `https://en.wikipedia.org/wiki/Object-oriented_programming`.
19. *What is Java?* [online]. IBM, 2019 [visited on 2022-11-23]. Available from: `https://www.ibm.com/cloud/learn/java-explained`.
20. SCHILDT, Herbert. Java's Magic: The Bytecode. In: *Java: The complete Reference*. Eleventh. McGraw-Hill Education, 2019. ISBN 978-1-26-044024-9.
21. *Java version history* [online]. Wikimedia Foundation, 2022 [visited on 2022-11-23]. Available from: `https://en.wikipedia.org/wiki/Java_version_history`.
22. *Javadoc* [online]. Wikimedia Foundation, 2022 [visited on 2022-11-23]. Available from: `https://en.wikipedia.org/wiki/Javadoc`.
23. *What is an IDE?* [online]. 2019. [visited on 2022-11-24]. Available from: `https://www.redhat.com/en/topics/middleware/what-is-ide`.
24. *IntelliJ IDEA overview* [online]. 2022. [visited on 2022-11-24]. Available from: `https://www.jetbrains.com/help/idea/discover-intellij-idea.html`.
25. PORTER, Brett; ZYL, Jason van; LAMY, Olivier. *Welcome to Apache Maven* [online]. 2022. [visited on 2022-11-24]. Available from: `https://maven.apache.org/`.
26. PANKAJ. *Spring Framework* [online]. DigitalOcean, 2022 [visited on 2022-11-24]. Available from: `https://www.digitalocean.com/community/tutorials/spring-framework`.
27. *Difference between IOC and Dependency Injection in Spring.* [online]. 2020. [visited on 2022-11-24]. Available from: `https://www.tutorialspoint.com/difference-between-ioc-and-dependency-injection-in-spring`.
28. BAELDUNG. *Spring Dependency Injection* [online]. 2022. [visited on 2022-11-24]. Available from: `https://www.baeldung.com/spring-dependency-injection`.
29. GABA, Ishan. *What Is JUnit: An Overview of the Best Java Testing Framework* [online]. Simplilearn, 2022 [visited on 2022-11-24]. Available from: `https://www.simplilearn.com/tutorials/java-tutorial/what-is-junit`.
30. TYSON, Matthew. *What is JSON? The Universal Data Format* [online]. InfoWorld, 2022 [visited on 2022-11-24]. Available from: `https://www.infoworld.com/article/3222851/what-is-json-a-better-format-for-data-exchange.html`.
31. MOLIN, Cassio Mazzochi. *Combining the Jackson Streaming API with ObjectMapper for Parsing JSON* [online]. 2019. [visited on 2022-11-24]. Available from: `https://cassiomolin.com/2019/08/19/combining-jackson-streaming-api-with-objectmapper-for-parsing-json/`.
32. JENKOV, Jakob. *Jackson ObjectMapper* [online]. 2019. [visited on 2022-11-24]. Available from: `https://jenkov.com/tutorials/java-json/jackson-objectmapper.html`.
33. *What is Git and Why Should You Use it?* [online]. 2022. [visited on 2022-11-24]. Available from: `https://www.nobledesktop.com/learn/git/what-is-git`.
34. *What is Jenkins?* [online]. 2022. [visited on 2022-11-24]. Available from: `https://www.edureka.co/blog/what-is-jenkins/`.

35. *SonarQube* [online]. Wikimedia Foundation, 2022 [visited on 2022-11-24]. Available from: `https://en.wikipedia.org/wiki/SonarQube`.
36. *Matillion ETL Documentation* [online]. [N.d.]. [visited on 2022-11-24]. Available from: `https://documentation.matillion.com/docs`.
37. *What is Data Definition Language (DDL)?* [online]. 2022. [visited on 2022-11-24]. Available from: `https://www.techopedia.com/definition/1175/data-definition-language-ddl`.
38. KOMPANIETS, Anastasia. *Functional vs. Non-Functional Requirements: Why Are Both Important?* [online]. [N.d.]. [visited on 2022-11-24]. Available from: `https://www.uptech.team/blog/functional-vs-non-functional-requirements`.
39. *Non-functional Requirements in Software Engineering* [online]. 2022. [visited on 2022-11-24]. Available from: `https://www.geeksforgeeks.org/non-functional-requirements-in-software-engineering/`.

Contents of enclosed media

```
readme.txt ........................ a brief description of the media content
src ........................................... the source codes of the work
  impl .............................. the directory of implementation source codes
  thesis ........................... the directory of LaTeX source codes of the work
text ............................................... the text of the work
  thesis.pdf ..................... the text of the work in the PDF format
```