FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Design and implementation data flow analysis of jobs in IBM DataStage for Manta project
Student:	Vladyslav Zavirskyy
Supervisor:	Ing. Michal Valenta, Ph.D.
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of winter semester 2019/20

Instructions

The aim of this work is to design and implement a functional module prototype that performs syntactic and semantic analysis of tasks in IBM DataStage and then uses its result for data flow analysis. The module will be included in the Manta project.

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

References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague August 31, 2018

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Design and implementation data flow analysis of jobs in IBM DataStage for Manta project

Vladyslav Zavirskyy

Department of Software Engineering
Supervisor: Ing. Valenta Michal Ph.D.

May 15, 2019

Acknowledgements

I want to thank the Manta team, especially RNDr. Lukáš Hermann and Mgr. Jiří Toušek, and Ing. Michal Valenta, Ph.D. for invaluable assistance during writing this thesis.

Declaration

I hereby declare that I have authored this thesis independently, and that all sources used are declared in accordance with the "Metodický pokyn o etické přípravě vysokoškolských závěrečných prací".

I acknowledge that my thesis (work) is subject to the rights and obligations arising from Act No. 121/2000 Coll., on Copyright and Rights Related to Copyright and on Amendments to Certain Laws (the Copyright Act), as amended, (hereinafter as the "Copyright Act"), in particular § 35, and § 60 of the Copyright Act governing the school work.

With respect to the computer programs that are part of my thesis (work) and with respect to all documentation related to the computer programs ("software"), in accordance with Article 2373 of the Act No. 89/2012 Coll., the Civil Code, I hereby grant a nonexclusive and irrevocable authorization (license) to use this software, to any and all persons that wish to use the software. Such persons are entitled to use the software in any way without any limitations (including use for-profit purposes). This license is not limited in terms of time, location and quantity, is granted free of charge, and also covers the right to alter or modify the software, combine it with another work, and/or include the software in a collective work.

In Prague on May 15, 2019 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Zavirskyy, Vladyslav. *Design and implementation data flow analysis of jobs in IBM DataStage for Manta project.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

Abstrakt

Cílem této práce je návrh a implementace funkčního prototypu modulu, provádějícího syntaktickou a sémantickou analýzu úloh v IBM InfoSphere DataStage. Modul se používá pro analýzu datových toků a generaci grafu, který reprezentuje datove toky. Návrh a implementace podporují bezproblémové připojení modulu k projektu Manta. Práce obsahuje důkladnou analýzu nástroje IBM InfoSphere DataStage, návrhovou dokumentaci, implementovaný prototyp modulu a také testy, které zajišťují funkcionalitu modulu.

Klíčová slova IBM InfoSphere DataStage, původ dat, analýza datových toků, datové sklady, Java

Abstract

This work aims to design and implement a functional module prototype that performs syntactic and semantic analysis of tasks in IBM InfoSphere DataStage. The module provides data flow analysis and generation of the graph, which represents data flows. Design and implementation support the trouble-free connection of the module to the Manta project. The work contains an in-depth analysis of the IBM InfoSphere DataStage tool, design documentation,

implemented the module prototype and tests, which ensures module functionality.

Keywords IBM InfoSphere DataStage, Data Lineage, Data flow analyzing, Data warehouse, Java.

Contents

List of Figures

Introduction

Motivation and objectives

The amount of data we produce every day is truly mind-boggling. There are 2.5 quintillion bytes of data created each day at our current pace, but that pace is only accelerating with the growth of the Internet of Things. Over the last two years alone 90 percent of the data in the world was generated. [1]

Companies getting more and more data by the second, and it becomes difficult to control it manually without any tool, which would help manage it, so that is why specialized tools were made. Please note that information about DataStage in this paragraph is based on the source [2]. A possible tool is IBM InfoSphere DataStage, which is among leading platforms that integrate data across multiple enterprise systems. DataStage allows us to create complex tasks, such as collecting, integrating and transforming large volumes of data, with data structures ranging from the simple to the complex. It is frequently used in Data Warehousing projects to prepare the data for the generation of reports. DataStage supports a wide range of technologies for connection as data sources, such as Oracle, Db2, Teradata, Hive, and Hadoop and so on. If we have a task, to proceed the whole way of some data from the source to the destination for figure out why in the end, we got such a result, it is maybe challenging and worth much effort. Task became more complicated if the source of the data is one system and it reaches the destination after passing several different databases and data tools, which provide transformations over them. In such conditions idea has a tool, which would allow us to get a whole way of data and all transformation, which were made over them, in one place in a second and just by one click of a computer mouse, cannot seem unattractive.

That is what precisely the Manta project is doing, namely visualize data flows across many different technologies. Manta project does not support IBM InfoSphere DataStage at the moment, what is the reason for the choice of this bachelor thesis.

The result of the work would be a module, which will provide analysis of IBM InfoSphere tasks. Afterward, the module will be joined to the Manta project.

This work is divided into five chapters. The first chapter guides a reader through basic concepts that are essential to the understanding of the topic. In the second chapter, we will make a detailed analysis and also we will choose technologies for later implementation. After that, in chapter number three we will make the design of the module based on the analysis. In the last two chapters, we will talk about implementation, and it is testing.

The aim of the work

This work aims to design and implement a functional module prototype that performs syntactic and semantic analysis of tasks in IBM InfoSphere DataStage and then uses its result for data flow analysis. This aim is broken up into three smaller, namely analytical, designing and an implementation part.

In the analytical part, it is essential to become acquainted with IBM InfoSphere DataStage and Manta project. IBM InfoSphere DataStage has more than 20 years of developing, so it has powerful functionality, and we have to choose, which functions to support. It is critical to deeply understand the structure of the Manta project, for the successful joining of the new module.

In designing part based on the result obtained in the analytical part, we will make the design concept of the module for the Manta project, which will provide IBM DataStage task processing. During designing, we will use the Manta project structure.

In the implementation part based on the design made in the previous part, we will implement the module, document and test it.

Chapter 1

Basic concepts

1.1 Data lineage

Data Lineage is defined as a data lifecycle that includes the data's origins and where it moves over time. The ability to track, manage and view data lineage helps simplify tracking errors back to the data source, and it helps to debug the data flow process. By tracking and utilizing the data lineage information within the analytics process, organizations are better able to shorten the decision making process, enhance data loss prevention and enable more efficient and cost-effective compliance and auditing. [3]

For our purposes, we will divide data flows into two groups. The first group is direct flows, which are responsible for transferring particular information from one place to another. The second group is a so-called filter or indirect flows, which are responsible for influence on transferring information from one place to another. Typically it is about conditions in WHERE constructions or IF condition. For instance,

if we would like to provide a sort operation, we will specify sorting keys on which to perform the sort. A key is a column on which to sort the data, for example, if we have an age column we might specify that as the sort key to producing an numerical list of ages.

In this situation, the key column is the source of the filter flow, and the rest of the columns are the destination of filter flow.

1.2 Manta

Information in this section is based on these sources: [4], [5].

Manta Tools provide data lineage visualization and code analysis. Originally Czech startup recently expanded into the USA and Western Europe with their two core products:

- Manta Flow – creates a detailed visualization of their BI environment including multiple technologies and messy custom code.
- Manta Checker – automates code reviews and helps them quickly fix errors.

1.2.1 What is Manta Flow?

Manta Flow is the tool that allows us to analyze programming code (SQL, Java, ETL) automatically and following the describing of transformational logic contained in it. Software is unique through the ability to recognize also hard to read programming code. Thanks to this feature, it is can in a reasonable time (usually few hours) read the whole database at a rate of hundred thousand and few millions of data, and create from its well-arranged data lineage map. In practice it is mainly used for data warehouse optimization, cost reduction of software developing, impact analysis and also for environment documentation for the needs of regulatory agencies. We can see the result of Manta Flow tool analyzing in figure 1.1.

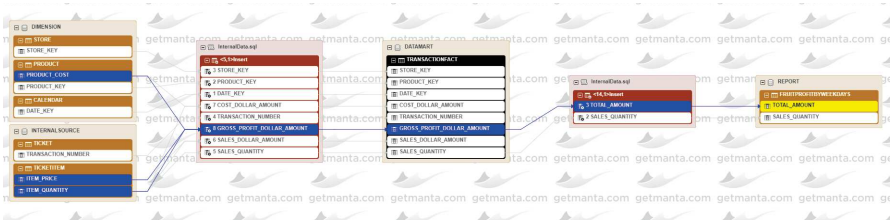

Figure 1.1: Example of Manta dataflow visualization

1.3 IBM InfoSphere DataStage

IBM InfoSphere DataStage (in the following merely the DataStage) is an ETL platform that integrates data across multiple enterprise systems. [2]

ETL is a type of data integration that refers to the three steps (extract, transform and load) used to blend data from multiple sources. It is often used to build a data warehouse. During this process, data is taken (extracted) from a source system, converted (transformed) into a format that can be analyzed, and stored (loaded) into a data warehouse or other system. Extract, load, transform (ELT) is an alternate but related approach designed to push processing down to the database for improved performance. [6]

DataStage extracts data, transform and load data from source to the target. The data sources might include sequential files, indexed files, relational databases, external data sources, archives and enterprise applications. DataStage

facilitates business analysis by providing quality data to help in gaining business intelligence. DataStage is used in a large organization as an interface between different systems. It takes care of the extraction, translation, and loading of data from the source to the target destination. [7]

DataStage has the following capabilities:

- *It can integrate data from the broadest range of enterprise and external data sources.*
- *It implements data validation rules.*
- *It is useful in processing and transforming large amounts of data.*
- *It uses a scalable parallel processing approach.*
- *It can handle complex transformations and manage multiple integration processes.*
- *Leverage direct connectivity to enterprise applications as sources or targets.*
- *Leverage metadata for analysis and maintenance.*
- *Operates in batch, real-time, or as a Web service.*

[7]

Chapter 2

Analysis

In this chapter, we will talk about the analysis, which took place on the start. The chapter is divided into five parts. In the first part will be discussed requirements analysis with functional and non-functional requirements. In the second part, we will choose technologies for further implementation, and after in the next part, we will talk about DataStage. In the fourth part, we will say some words about the Transformer stage, and after that, we will choose the format of exporting projects from DataStage. In the last two chapters, we will describe the structure of the exported XML file, and we will take a closer look at the Transformer stage.

2.1 Requirements analysis

Requirements Analysis is the process of defining the expectations of the users for an application that is to be built or modified. Requirements analysis involves all the tasks that are conducted to identify the needs of different stakeholders. Therefore requirements analysis means to analyze, document, validate and manage software or system requirements. High-quality requirements are documented, actionable, measurable, testable, traceable, helps to identify business opportunities, and are defined to a facilitate system design. [8]

2.1.1 Functional requirements

Functional requirements describe the services the system should provide. Sometimes the functional requirements state what the system should not do. Functional requirements can be high-level and general or detailed, expressing inputs, outputs, exceptions, and so on. [9, p. 47–48]

F1: Analysis and graph construction
The main functional requirement is that the module performs syntactic and semantic analysis of tasks in IBM DataStage and then uses its result for data flow

analysis. After the data flow analysis, the module will construct a DataStage part of the general Manta graph.

F2: Direct and filter flow

When the module performs analysis, it is getting data flows and distinguishes between direct and filter flows so that it will use the right type of edge during graph building. It is a vast difference between direct and filter flow, so for us, it is critical to identify the type of flow correctly.

F3: Export analyzing

The module analyzes jobs exported in DSX or XML format and maps valuable information into the inner structure for more convenient future analyzing.

F4: Log

In case of error during analyzing, the module will provide a meaningful message, which would help to understand what exactly happened.

2.1.2 Non-functional requirements

Non-functional requirements are imposed by the environment in which the system is to exist. These requirements could include timing constraints, quality properties, standard adherence, programming languages to be used, compliance with laws, and so on. [9, p. 47–48]

NF1: Performance

The module performs the analysis in a reasonable timeframe. Manta tools customers are companies with a massive amount of data, and that is why they are using Manta, so analysis should be provided as fast as possible.

NF2: Integration with Manta project

The module is supposed to be a part of the Manta project, so it should be able to be smoothly integrated into it. It can be achieved by implementing common interfaces, which are using for assembling manta modules.

NF3: Extensibility

Because of powerful DataStage functionality, now module does not support analysis of all functions which DataStage provides, so module should be easily extended in the future for supporting remaining functionality.

2.2 Used technologies

2.2.1 Integrated development environment

An integrated development environment (IDE) is a software suite that helps the programmer with software developing. During the software developing process, we are using a broad range of tools, such as text editor, compiler, version control and other.

IntelliJ IDEA by JetBrains was used for developing the module. It was chosen due to a wide range of development tools and plugins, which we can install to the IDE and efficiently work with them.

2.2.2 Java

The Java programming language is a general-purpose, concurrent, class-based, object-oriented language. It is designed to be simple enough that many programmers can achieve fluency in the language. The Java programming language is strongly typed. This specification clearly distinguishes between the compile-time errors that can and must be detected at compile time, and those that occur at run time. The Java programming language is a relatively high-level language. It includes automatic storage management, typically using a garbage collector, to avoid the safety problems of explicit deallocation. High-performance garbage-collected implementations can have bounded pauses to support systems programming and real-time applications. [10, p. 1]

Object-oriented programming lets us think in objects of the real world, which in turn helps us during the development. Using Java, we partly fulfill non-functional requirement number one because Java, base on the source [11], is modern and high-performance technology.

2.2.3 Maven

Apache Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting, and documentation from a central piece of information. [12] During module developing maven was used for dependency management, build automation and parser generation.

2.2.4 JUnit

JUnit is a simple framework to write repeatable tests. It is an instance of the xUnit architecture for unit testing frameworks. [13] JUnit allows instantly to make sure that the program works properly and deliver a working solution.

2.2.5 Spring

The Spring Framework provides a complete programming and configuration model for modern Java-based enterprise applications - on any deployment platform. A key element of Spring is infrastructural support at the application level: Spring focuses on the "plumbing" of enterprise applications so that teams can focus on application-level business logic, without unnecessary ties to specific deployment environments. [14]

Spring test module makes beans from a spring context accessible for JUnit tests – it is required; otherwise, we cannot provide any test.

2.2.6 ANTLR 3

ANTLR 3 - ANother Tool for Language Recognition - is a tool that is used in the construction of formal language software tools such as translators, compilers, recognizers and, static/dynamic program analyzers. ANTLR 3 uses a top-down parser for a subset of context-free languages. It parses the input from Left to right, performing the Leftmost derivation of the sentence. ANTLR 3 is a compiler generator, and it is used to generate the source code for language recognizers, analyzers, and translators from language specifications. ANTLR 3 takes as its input a grammar - an accurate description of a language augmented with semantic actions - and generates source code files and other auxiliary files. The target language of the generated source code in our case is Java.

ANTLR 3 reads a language description file called grammar and generates several source code files and other auxiliary files. ANTLR generates these tools:

- *A Lexer: This reads an input character or byte stream (i.e., characters, binary data and other), divides it into tokens using patterns we specify, and generates a token stream as output.*

- *A Parser: This reads a token stream generated by a lexer, and matches phrases in language via the rules (patterns) we specify, and typically performs some semantic action for each phrase (or sub-phrase) matched. Each match generates an Abstract Syntax Tree for additional processing.*

[15]

One of the DataStage transformations is the Transformer stage, and this stage allows us to create transformations to apply to data. Transformations are described by simple expression language, which we are analyzing using parser generated by ANTLR 3.

2.2.7 SLF4J/log4j

The Simple Logging Facade for Java (SLF4J) serves as a simple facade or abstraction for various logging frameworks (e.g., java.util.logging, logback, log4j) allowing the end user to plug in the desired logging framework at deployment time. [16] As a logging framework, we are using log4j. These tools help us to complete functional requirement number four.

2.2.8 dom4j

dom4j is an easy to use, open source library for working with XML, XPath, and XSLT on the Java platform using the Java Collections Framework and with full support for DOM, SAX and JAXP. [17] This tool allows us to fulfill the functional requirement number three.

2.2.9 XPath

XPath (XML Path Language) is a language for addressing parts of an XML document, designed to be used by both XSLT and XPointer. The primary purpose of XPath is to address parts of an XML document. In support of this primary purpose, it also provides basic facilities for manipulation of strings, numbers, and booleans. XPath uses a compact, non-XML syntax to facilitate the use of XPath within URIs and XML attribute values. XPath operates on the abstract, logical structure of an XML document, rather than its surface syntax. XPath gets its name from its use of a path notation as in URLs for navigating through the hierarchical structure of an XML document. [18]

2.3 DataStage

DataStage is part of a broader suite of products called InfoSphere Information Server. We will take a look at the architecture of Information Server 11.7, the latest version at the time of writing.

2.3.1 DataStage Architecture

The architecture is service-oriented, enabling IBM InfoSphere Information Server to work within evolving enterprise service-oriented architectures. A service-oriented architecture also connects the individual suite product modules of InfoSphere Information Server. [19]

A service-oriented architecture is essentially a collection of services. These services communicate with each other. The communication can involve either simple data passing, or it could involve two or more services coordinating some activity. Some means of connecting services to each other is needed. [20]

The figure 2.1 shows the InfoSphere Information Server architecture. Figure source is [19]

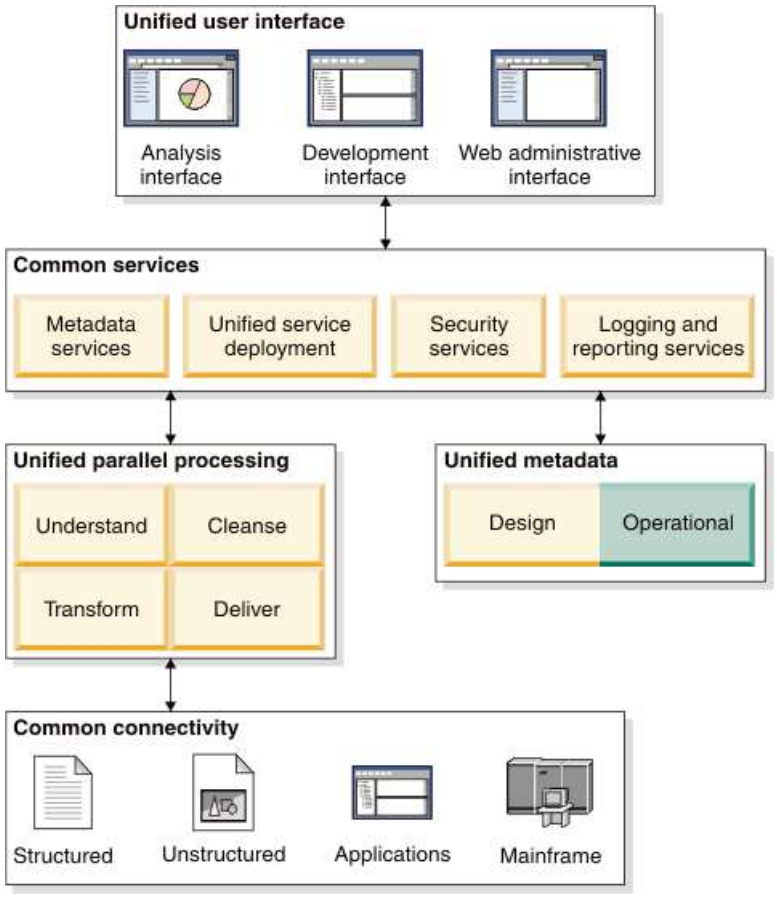

Figure 2.1: InfoSphere Information Server high-level architecture

Information Server is divided into modules - logical tiers. Each tier includes a subgroup of the components that make up InfoSphere Information Server. The tiers provide services, job execution, and storage of metadata and other data for product modules. Information Server consists of the next tiers: client, services, engine, repository, and microservices tier. [21] In figure 2.2 we can see tier relationships. The source of the figure is [22].

Client programs on the client tier communicate primarily with the services tier. Many services within the services tier communicate with agents on the engine tier. DataStage clients also communicate with the engine tier. ODBC drivers on the engine tier communicate with external databases. [22]

Info server provides three client tools for working with DataStage part of Information Server, namely:

- Administrator Client: It is used for administration tasks, so it allows us to create and delete new projects and change its properties.

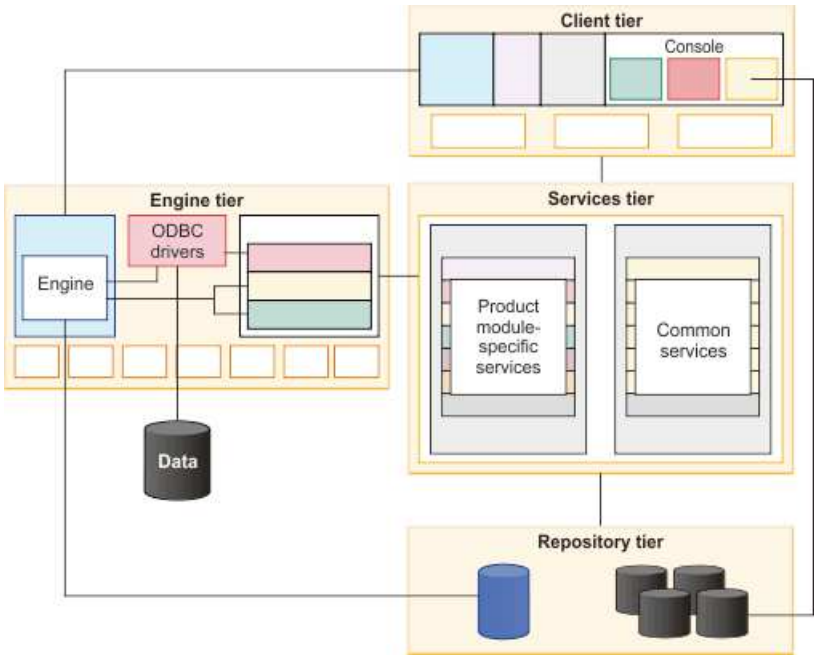

Figure 2.2: Tier relationships

- Designer Client: The Designer client is a graphical tool for job designing, which allows us to create different types of jobs. In figure 2.3 we can see the canvas for job designing.
- DataStage Flow Designer: It is a web-based version of DataStage Designer Client, but it is a new tool, at the time of writing, which does not support all functions, unlike Designer client. So for jobs creating, during the writing of this work, was chosen Flow Designer.
- Director Client: It is used to schedule, execute and monitor jobs.

2.3.2 Components

The main element in DataStage is a Job, which describes dataflow from a source to a target. DataStage offers us next types of jobs:

- Parallel job – newer and more effective, runs parallel. Parallel's job analyzing is the main functionality of the module, which will be a result of this bachelor thesis.
- Sequence job – allows us to use parallel and server jobs as stages
- Server job – older and less efficient, runs in sequence. Customers do not use last time, so not as interesting for us as a Parallel job.

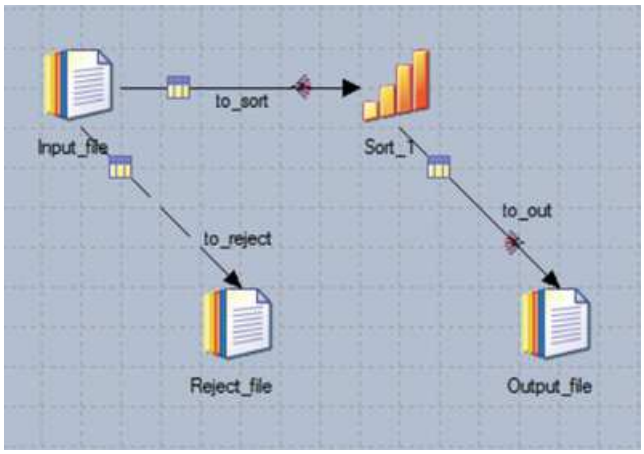

Figure 2.3: DataStage Designer Client canvas for a job-creating

Unlike Parallel job stages, Server job stages do not have built-in partitioning and parallelism mechanism for extracting and loading data between different stages. [23]

The job consists of stages and links; using them, we can create complex tasks for extracting, transforming and loading data. Mostly each type of job has it is own unique stages, because of different approaches and functionals, but there are some common stages in Parallel and Sequence jobs. Usually, each stage has at least one input and one output link, but there are stages that have more or fewer links.

Stages and links can be grouped in a shared container. Instances of the shared container can then be reused in different parallel jobs. We can also define a local container within a job — this groups stages and links into a single unit, but can only be used within the job in which it is defined. [24, p. 23]

Each stage has a set of predefined and editable properties that tell it how to perform or process data. We can see example of stage properties in figure 2.4. Properties might include the file name for the Sequential File stage, the columns to sort, the transformations to perform, and the database table name for the DB2 stage. These properties are viewed or edited using stage editors. Stages are added to a job and linked together using the Designer. [24, p. 36]

Stages have stage properties, which cover the whole stage, and link properties, which covers only data going through the link. For instance, a sort stage, which allows us to perform complex sorting over the data, has only stage properties where we can define key columns for sorting, sort order and other properties related to sorting. Oracle connector, which allows us to read or load data from Oracle database, in addition to stage properties, where we are defining the server, user name, and password for connecting to a database, has input or output links (depends if the stage is used as a source or as a tar-

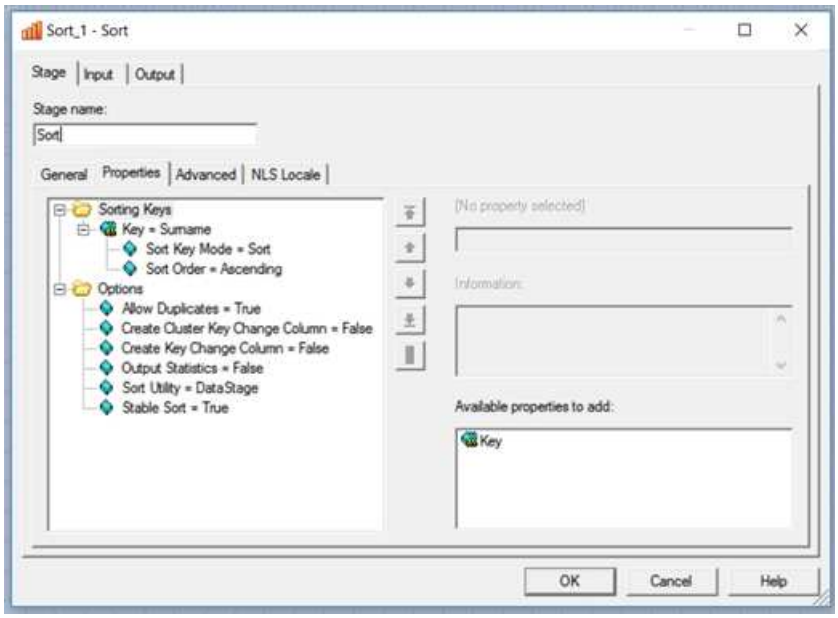

Figure 2.4: Example of stage properties

get). Each link has it is own properties, where we can define SQL statements or table names, which will be used during data extraction or loading.

Stage property values can be set manually, or we can use a predefined job parameter. Job parameters allow us to modify the value of a parameter, such as a file path, server name, connection string, password, and after then all jobs in production will run with new values without the need of recompiling them. We can see job parameters in figure 2.5 This feature is highly useful; it simplifies job managing and provides an easy way to change stage properties in the case of a mistake.

In figure 2.6, we can see an example of using job parameter FILE_PATH for providing File property in the Sequential File stage. Two hashes should surround the name of the parameter. So parameter name should be provided in the next format: #FILE_PATH#.

Each link contains columns. Columns have the following parameters:

- Column name – The name of the column.
- Key – Indicates whether the column is part of the primary key.
- SQLType – The SQL data type.
- Extended – Allows us to specify column type more precisely.
- Length – The data precision.
- Scale – The data scale factor.

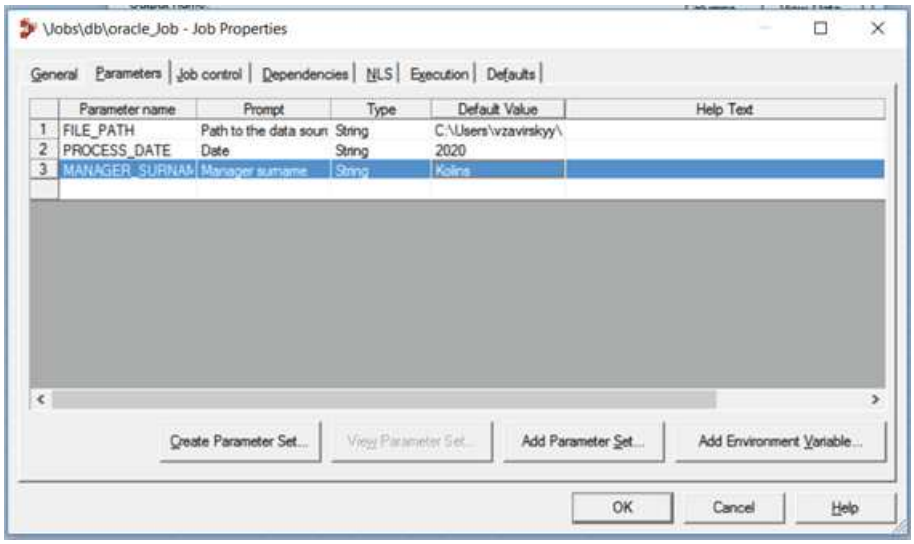

Figure 2.5: Job parameters

- Nullable – Indicates whether the column can contain a null value.
- Description – Describes the purpose of the column .

In figure 2.7, we can see the "Runtime column propagation" checkbox. It is responsible for enabling runtime column propagation. We can define part of our schema and specify that, if our job encounters extra columns that are not defined in the metadata when it runs, it will adopt these extra columns and propagate them through the rest of the job. It is known as runtime column propagation.[25] This feature may be extremely useful during job designing, but it was decided not to include this to the module, due to the complexity of it.

2.3.3 Supported jobs and stages

Parallel jobs were chosen as most relevant at this moment, due to the interesting of customers to them, so mainly we will support them. Parallel jobs have ten different groups of stages:

- General – annotations, containers, and links
- Data Quality – MDM(Mobile device management) connector
- Database – connectors to different databases
- Development/Debug – tools for simplifying the development process
- File – connectors to different files

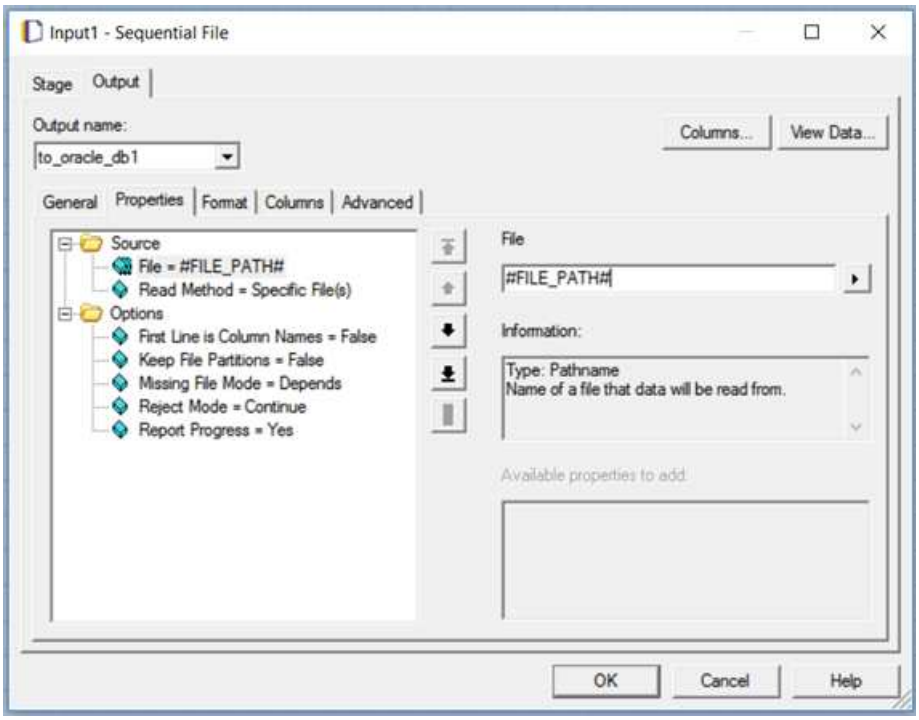

Figure 2.6: Using job parameter FILE_PATH in Sequential File stage

- Hive – Hive metastore import connector
- Packs – connectors to third side tools as Saleforce or Essbase
- Processing – stages which are responsible for letter T in ETL acronym, namely data transformer tools
- Real-Time – stages which perform real-time data integration
- Restructure – stages for data flow restructuring

General, Database, File, and Processing groups were chosen for analyzing as the most relevant. Let us take a closer look at the content of each group.

General group:

- Annotation – It allows us to add some notes on a job canvas.
- Container – It allows us group stage and links.
- Description annotation – The same as Annotation, but can be added to the canvas just once.
- Link – It is the connection between stages.

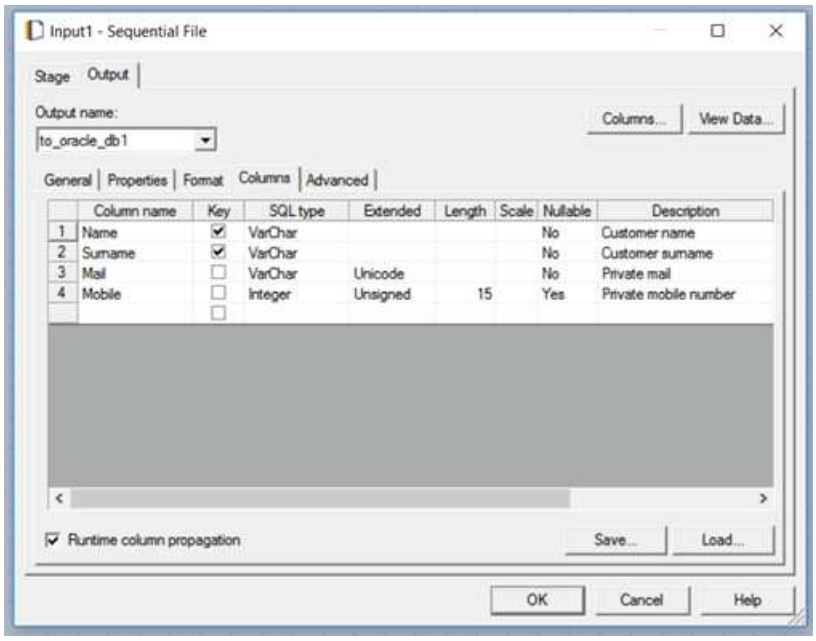

Figure 2.7: Example of columns in output link

Names of stages make the clear purpose of each stage, but for some of them we will make notes. Information about stages in Database and File groups based on the source [26]

Database group:

- Classic Federation
- Cognos TM1 Connector
- DB2 Connector
- Distributed Transaction – *Stage provides the capability to run distributed transactions, with IBM WebSphere MQ as the transaction manager, and IBM DB2 connector, Terradata connector, ODBC connector, or the Oracle connector as the resource manager.*
- DRS Connector – *It is a relatively new stage that offers better functionality and performance than older connectors, so it is preferable to use it. It supports connection to IBM Db2, Oracle, and ODBC data sources.*
- Greenplum Connector
- HBase Connector

- Hive Connector
- Informix CLI
- Informix Enterprise
- Informix Load
- iWay Enterprise
- JDBC Connector
- Netezza Connector
- ODBC Connector
- Oracle Connector
- SQLServer Enterprise
- Stored Procedure – *It allows us to execute stored procedures from MSSQL Server, Oracle, Db2, Sybase and Teradata.*
- Sybase Enterprise
- Sybase IQ 12 Load
- Sybase OC
- Teradata Connector

File group:

- Amazon S3
- Complex Flat File
- Data Set – *It is an internal data representation of data in DataStage. Parallel jobs use data sets to manage data within a job. We can think of each link in a job as carrying a data set.*
- External Source – *It allows us to read data that is output from one or more source programs. The stage calls the program and passes appropriate arguments.*
- External Target – *The same as External Source, but write data to the input of some program.*
- File Connector – *The stage that allows us to read data from or write data to a file set. It allows us to read files from and write files to a local file system on the engine tier or a Hadoop Distributed File System (HDFS) by using the WebHDFS API or the HttpFS API.*

- File Set – *DataStage can generate and name exported files, write them to their destination, and list the files it has generated in a file whose extension is, by convention, .fs. The data files and the file that lists them are called a file set.*
- Lookup File Set – *The same as File Set, but modified for better performance when connected with Lookup stage.*
- Sequential File
- Unstructured Data – It uses Excel files as the source or target.
- z/OS File – It allows us to use as a source or a target file from z/OS, operation system for mainfraimes from IBM.

Information in processing group is based on the source [27].
Processing group:

- Aggregator – *It classifies data rows from a single input link into groups and computes totals or other aggregate functions for each group.*
- Bloomfilter – *Bloom filter operator allows us to lookup incoming keys efficiently against previous values.*
- Change Apply – *It takes the change data set, which contains the changes in the before and after data sets, from the Change Capture stage and applies the encoded change operations to a before data set to compute an after data set.*
- Change Capture – *The stage compares two data sets and makes a record of the differences.*
- Checksum – *Use the Checksum stage to generate a checksum value from the specified columns in a row and add the checksum to the row.*
- Compare – *This stage performs a column-by-column comparison of records in two presorted input data sets.*
- Compress – *This stage uses the UNIX compress or GZIP utility to compress a data set.*
- Copy – *The Copy stage copies a single input data set to several output data sets.*
- Data Masking – *Use the Data Masking stage to mask sensitive data that must be included for analysis, in research, or for the development of new software.*

- Decode – *It decodes a data set using a UNIX decoding command, such as gzip, that we supply.*
- Difference – *It performs a record-by-record comparison of two input data sets, which are different versions of the same data set designated the before and after data sets.*
- Encode – *It encodes a data set using a UNIX encoding command, such as gzip, that we supply.*
- Expand – *Stage uses the UNIX uncompress or GZIP utility to expand a data set.*
- External Filter – *The External Filter stage is a processing stage that allows us to specify a UNIX command that acts as a filter on the data we are processing.*
- Filter – *This stage transfers, unmodified, the records of the input data set which satisfy the specified requirements and filters out all other records.*
- FTP Enterprise – *This stage invokes an FTP client and transfers the files to and from a remote host.*
- FTP Plug-in – *The FTP Plug-in stage provides users with rapid and efficient remote file access using existing FTP servers on remote platforms.*
- Funnel – *The Funnel stage is a processing stage that copies multiple input data sets to a single output data set.*
- Generic – *The Generic stage is a processing stage that allows us to call an Orchestrate operator from within a stage and pass it options as required.*
- Join – *It performs join operations on two or more data sets input to the stage and then outputs the resulting data set.*
- Lookup – *It is used to perform lookup operations on a data set read into memory from any other Parallel job stage that can output data.*
- Merge – *The Merge stage combines a master data set with one or more update data sets.*
- Modify – *The Modify stage alters the record schema of its input data set.*
- Pivot – *It maps sets of columns in an input table to a single column in an output table.*
- Pivot Enterprise – *The Pivot Enterprise stage is a processing stage that pivots data horizontally and vertically.*

- Remove Duplicates – *The Remove Duplicates stage takes a single sorted data set as input, removes all duplicate rows, and writes the results to an output data set.*
- Slowly Changing Dimension – *The stage reads source data on the input link, performs a dimension table lookup on the reference link, and writes data on the output link.*
- Sort – *The stage is used to perform complex sort operations.*
- Surrogate Key Generator – *The stage is a processing stage that generates surrogate key columns and maintains the key source.*
- Switch – *The stage takes a single data set as input and assigns each input row to an output data set based on the value of a selector field.*
- Transformer – *Stage allows us to create transformations to apply to data. These transformations can be simple or complex and can be applied to individual columns in data. Transformations are specified using a set of functions.*
- Wave Generator – *DataStage wave generator is used for handling the job. There is a need to determine end-of-wave or end-of-data marker on every interval for some stages to start processing the data — for example, the Sort stage and the Aggregator stage. Sort stage cannot work when the data is still incoming data. To handle this, an end of wave marker is generated on every interval. End-of-wave indicates that the data can be pushed to the next level.*

2.3.4 Transformer stage

The Transformer stage is a powerful tool, which allows us using expression language to create data transformations. In figure 2.8, we can see transformer stage editor. This stage can replace any other processing stage, because of a large number of different functions.

We can declare and use our variables within a Transformer stage. Such variables are accessible only from the Transformer stage in which they are declared.

Stage variables can be used as follows:

- *Expressions can assign them values.*
- *They can be used in expressions which define an output column derivation.*
- *Expressions evaluating a variable can include other variables or the variable being evaluated.*

[28]

Transformer stage supports expressions from simple to complex. An example of simple expression is sum of two input columns:

```
Link_name.Column_1 + Link_name.Column_2
```

and an example of complex expression is if then else expression:

```
If ((max(Link.Column_1, Link.Column_2) + 2) > 5)
Then Link.Column_3 Else Link.Column_1.
```

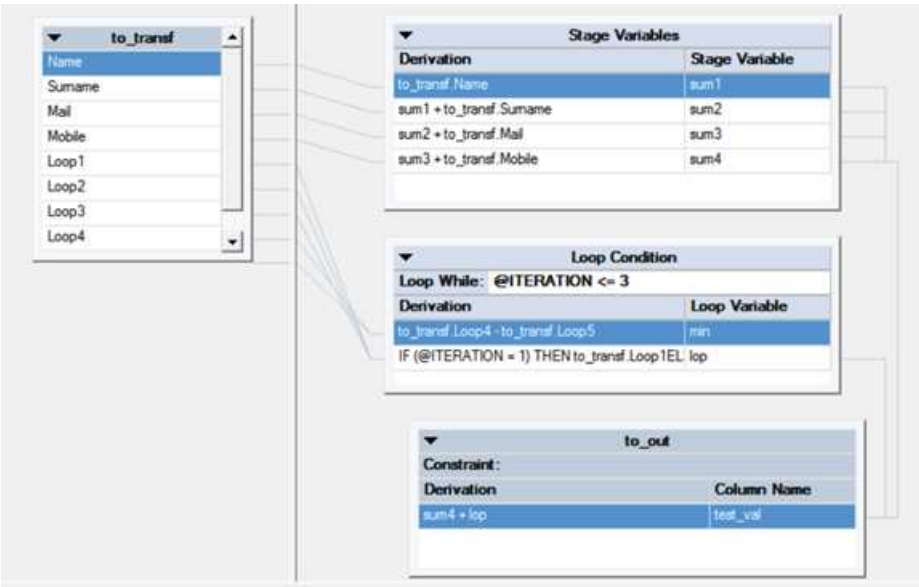

Figure 2.8: Transformer stage editor

2.4 Possible input file formats for analyzing

There are two options available to us in which format export projects from DataStage, namely DSX and XML. We will speak more about the advantages and disadvantages of each of them below.

2.4.1 Dsx

Dsx is a file format used in IBM products. Structure of the file little like XML structure. In XML each element begins with start-tag <element>and ends with end-tag <element/>.

In DSX another approach is taken, the file contains construction looks as follows BEGIN ELEMENT and END ELEMENT with all metadata inside written in this way: name “value”

In figure 2.9, we can see an example of an element in DSX format. This example describes a column in the output link one of the stages.

```
BEGIN DSSUBRECORD
    Name "Name"
    SqlType "12"
    Precision "0"
    Scale "0"
    Nullable "0"
    KeyPosition "1"
    DisplaySize "0"
    Group "0"
    SortKey "0"
    SortType "0"
    AllowCRLF "0"
    LevelNo "0"
    Occurs "0"
    PadNulls "0"
    SignOption "0"
    SortingOrder "0"
    ArrayHandling "0"
    SyncIndicator "0"
    PadChar ""
    ExtendedPrecision "1"
    TaggedSubrec "0"
    OccursVarying "0"
    PKeyIsCaseless "0"
    SCDPurpose "0"
END DSSUBRECORD
```

Figure 2.9: Part of the dsx export file

IBM does not provide any documentation, libraries or tools for DSX format. There are few open source solutions, namely parsers written in Python, Scala, Perl, and ANTLR4 grammar – individuals write them.

2.4.2 XML

Extensible Markup Language (XML) is a simple, very flexible text format derived from SGML (ISO 8879). Originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere. [29]

XML is an open source, well-known technology with good documentation and plenty of tried and trusted open source libraries written in Java and other popular programming languages. Text editors such as Notepad++ allows us to browse XML files comfortably so that data mining will be more comfortable

and faster. Also, I had some excellent experience with parsing and creating XML files.

2.4.3 Choosing a format for analyzing

Following a detailed analysis of the two approaches, as a format for exporting projects from DataStage, was chosen XML. Thanks to a large number of open source libraries written in Java, good documentation and my previous experience with this technology, the choice seems obvious.

2.5 Export file structure

Here we will talk about the structure of the XML file received after project export. Based on the selected jobs and stages for supporting, we will provide an analysis of the following elements.

2.5.1 Link Structure

When we add a link between two stages on the canvas in the Designer client, example of this we can see in figure 2.10, in the XML export, it will add output and input links and connect them in a specific way. The XML structure of the link is shown in figure 2.11. In the following text, we will use the term "Corresponding output link", which means the output link of the source stage. We should notice, that Output Link and Input Link have different Identifiers, but they have the same Name as on the canvas.

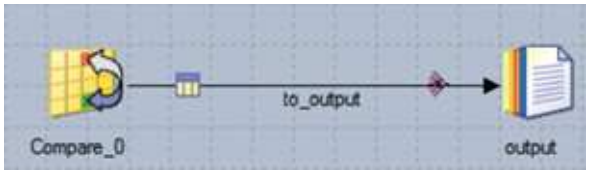

Figure 2.10: Example of the link

2.5.2 Root Element Structure

Root element is DSExport, inside we can find element Header, which has as attributes following values CharacterSet, ExportingTool, ToolVersion, ServerName, ToolInstanceID, Date, Time and ServerVersion. The header does not contain any other elements inside. Together with the Header element, inside are placed Job elements, which contain descriptions of jobs.

Job element as attributes has the following values Identifier, DateModified, and TimeModified. The identifier is the name of the job, which was chosen by the user.

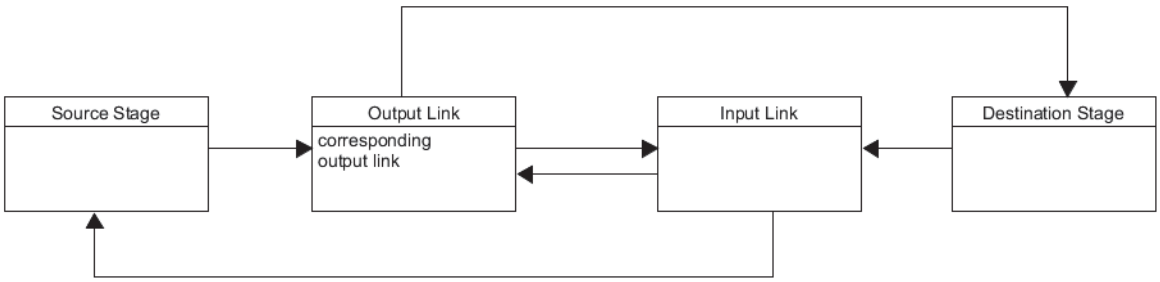

Figure 2.11: Structure of the link in XML

In figure 2.12 we can see an example of the root structure. For illustrative purposes, Header has fewer attributes, than it is.

```
<DSExport>
    <Header CharacterSet="CP1250" ExportingTool="IBM InfoSphere DataStage Export" ToolVersion="8"
    <Job Identifier="seqFileJob" DateModified="2019-04-26" TimeModified="23.27.31">
    <Job Identifier="transformFilesJob" DateModified="2018-05-1" TimeModified="11.20.14">
</DSExport>
```

Figure 2.12: Content of the root element of the exported file

Example of job elements we can see in figure 2.13. Job elements have the following attributes:

- Identifier – unique identifier within a job. DataStage generates this identifier
- Type – show type of the element, it is the purpose
- Readonly – indicates if the element is read-only

```
<Job Identifier="try" DateModified="2019-04-26" TimeModified="23.27.31">
    <Record Identifier="ROOT" Type="JobDefn" Readonly="0">
    <Record Identifier="V0" Type="ContainerView" Readonly="0">
    <Record Identifier="V0S0" Type="CustomStage" Readonly="0">
    <Record Identifier="V0S0P1" Type="CustomOutput" Readonly="0">
    <Record Identifier="V0S2" Type="CustomStage" Readonly="0">
    <Record Identifier="V0S2P1" Type="CustomInput" Readonly="0">
</Job>
```

Figure 2.13: Example of the Job element content

2.5.3 JobDefn Structure

It contains job definition. The only useful data for us from this element are job parameters. Job parameters have the following properties:

- Name – A unique name chosen by the user.
- Prompt – The text which is using during a job run for prompting.
- Default – The value of the parameter.
- HelpTxt – It is help text for cases, when the job parameters are specified at runtime.
- ParamType – A type of the parameter. The available types are: String, Encrypted, Integer, Float, Pathname, List, Date and Time.
- ParamLenght – The length of the parameter.
- ParamScale – The scale of the parameter.

For our purposes, the only useful information for us is Name and Default values. Job parameters are disposed of in the XML structure, which we can see in figure 2.14.

```
<Collection Name="Parameters" Type="Parameters">
   <SubRecord>
      <Property Name="Name">FILE_PATH</Property>
      <Property Name="Prompt">Path to the source file</Property>
      <Property Name="Default">C:\Users\vzavirskyy\Desktop\filter\input1.txt</Property>
      <Property Name="HelpTxt">It is the file path to the data source</Property>
      <Property Name="ParamType">4</Property>
      <Property Name="ParamLength">0</Property>
      <Property Name="ParamScale">0</Property>
   </SubRecord>
   <SubRecord>
      <Property Name="Name">SERVER_NAME</Property>
      <Property Name="Prompt">Name of the server</Property>
      <Property Name="Default">localhost</Property>
      <Property Name="HelpTxt">Name of the local server</Property>
      <Property Name="ParamType">0</Property>
      <Property Name="ParamLength">0</Property>
      <Property Name="ParamScale">0</Property>
   </SubRecord>
</Collection>
```

Figure 2.14: Example of job parameters structure

2.5.4 CustomStage Structure

The element is used for describing all types of stages, except for the Transformer stage. Example of the structure we can see in figure 2.15. For our purposes useful information from the element are:

- Name – A unique name chosen by the user.
- InputPins, OutputPins – There are identifiers of input and output links. Using these identifiers, we can bind stages with each other.
- StageType – A type of the stage. Using this we can understand which type of analysis we should provide.

- Collection – It contains properties of the stage. These properties are useful during stage analysis.

```
<Record Identifier="V0S69" Type="CustomStage" Readonly="0">
   <Property Name="Name">Checksum_69</Property>
   <Property Name="NextID">3</Property>
   <Property Name="InputPins">V0S69P1</Property>
   <Property Name="OutputPins">V0S69P2</Property>
   <Property Name="StageType">PxChecksum</Property>
   <Property Name="AllowColumnMapping">0</Property>
   <Collection Name="Properties" Type="CustomProperty">
      <SubRecord>
         <Property Name="Name">comp_mode</Property>
         <Property Name="Value">\(20)</Property>
      </SubRecord>
   </Collection>
   <Property Name="NextRecordID">0</Property>
</Record>
```

Figure 2.15: Example of CustomStage structure

2.5.5 TransformerStage Structure

The element is used for describing the Transformer stage. The example of the structure we can see in figure 2.16. It contains the same useful information for analysis as CustomStage, but with two additions:

- StageVars – stage variables
- LoopVars – loop variables

2.5.6 CustomInput, StdPin, TrxInput Structure

CustomInput, StdPin, and TrxInput are input links for different stages. CustomInput is used with the most stages, StdPin is used with Generic and Wave Generator stages, and TrxInput is the special input link for the Transformer stage. Their structures we can see in figures 2.17, 2.18, 2.19. Input links do not contain any columns.

All of these links have the same data, which we need during the analysis:

- Name – A unique name of a link within the job.
- Partner – It keeps identifiers of the source stage and the corresponding output link.

```
<Record Identifier="V0S151" Type="TransformerStage" Readonly="0">
   <Property Name="Name">Transformer_151</Property>
   <Property Name="NextID">3</Property>
   <Property Name="InputPins">V0S151P1</Property>
   <Property Name="OutputPins">V0S151P2</Property>
   <Collection Name="MetaBag" Type="MetaProperty">
   <Property Name="ValidationStatus">0</Property>
   <Property Name="StageType">CTransformerStage</Property>
   <Property Name="BlockSize">0</Property>
   <Property Name="SKKeySourceType">file</Property>
   <Collection Name="StageVars" Type="StageVar">
      <SubRecord>
         <Property Name="Name">stageVar1</Property>
         <Property Name="SqlType">12</Property>
         <Property Name="Precision">255</Property>
         <Property Name="ColScale">0</Property>
         <Property Name="ExtendedPrecision">0</Property>
      </SubRecord>
   </Collection>
   <Collection Name="LoopVars" Type="StageVar">
      <SubRecord>
         <Property Name="Name">loopvar</Property>
         <Property Name="SqlType">12</Property>
         <Property Name="Precision">255</Property>
         <Property Name="ColScale">0</Property>
         <Property Name="ExtendedPrecision">0</Property>
      </SubRecord>
   </Collection>
   <Property Name="StageVarsMinimised">0</Property>
   <Property Name="LoopVarsMaximised">1</Property>
   <Property Name="MaxLoopIterations">0</Property>
</Record>
```

Figure 2.16: Example of TransformerStage structure

```
<Record Identifier="C2P1" Type="StdPin" Readonly="0">
   <Property Name="Name">DSLink2</Property>
   <Property Name="Partner">V0S126|V0S126P2</Property>
   <Property Name="LinkType">1</Property>
   <Collection Name="MetaBag" Type="MetaProperty">
</Record>
```

Figure 2.17: Example of StdPin

2.5.7 CustomOutput, StdOut, TrxOutput Structure

It is the same situation with these elements as with the previous point, but there are output links. Also unlike the previous point, output links contain columns. The only important change, in comparison with the previous point, is the availability of columns, so we go straight to the column structure. We can see an example of the structure in figure 2.20. The structure can be a little bit different depending on the stage from which link comes out, but the main properties are next:

```
<Record Identifier="V0S102P1" Type="CustomInput" Readonly="0">
   <Property Name="Name">DSLink104</Property>
   <Property Name="Partner">V0S103|V0S103P1</Property>
   <Property Name="LinkType">1</Property>
   <Property Name="ConditionNotMet">fail</Property>
   <Property Name="LookupFail">fail</Property>
   <Collection Name="MetaBag" Type="MetaProperty">
   <Property Name="TransactionSize">0</Property>
   <Property Name="TXNBehaviour">0</Property>
   <Property Name="EnableTxGroup">0</Property>
   <Property Name="LinkMinimised">0</Property>
</Record>
```

Figure 2.18: Example of CustomInput

```
<Record Identifier="V0S151P1" Type="TrxInput" Readonly="0">
   <Property Name="Name">DSLink3</Property>
   <Property Name="Partner">C2|C2P2</Property>
   <Property Name="LinkType">1</Property>
   <Collection Name="MetaBag" Type="MetaProperty">
   <Property Name="MultiRow">0</Property>
   <Property Name="LinkMinimised">0</Property>
</Record>
```

Figure 2.19: Example of TrxInput

- Name – The name of the column chosen by the user.
- SqlType – SQL type chosen by user. There are 32 different types of variables in DataStage.
- ExtendedPrecision – It gives us further control over data type. For instance, if SqlType is Integer than as ExtendedPrecision value, we can choose Unsigned. User chooses value.
- Precision – The data precision value, which is chosen by user.
- Scale – The data scale factor chosen by user.
- Nullable – Indicates whether the column can contain null values. User chooses value.
- Derivation – Provides us information from which column was the column derived. If the column was created in this stage than Derivation will be null. If column value was derived from other stage and transferred to this stage by some link, than Derivation value will have next format: <Input link name>.<Column name in the source stage>. Also if the column was created using some function than the value will have next format: <Function name>() or <Function name>(<Input link name>.<Column name in the source stage>). Here less than (<) and

greater than (>) symbols means to start and end of a string of characters.

```
<Collection Name="Columns" Type="OutputColumn">
   <SubRecord>
      <Property Name="Name">Name</Property>
      <Property Name="SqlType">1</Property>
      <Property Name="Precision">0</Property>
      <Property Name="Scale">0</Property>
      <Property Name="Nullable">0</Property>
      <Property Name="KeyPosition">0</Property>
      <Property Name="DisplaySize">0</Property>
      <Property Name="Group">0</Property>
      <Property Name="SortKey">0</Property>
      <Property Name="SortType">0</Property>
      <Property Name="AllowCRLF">0</Property>
      <Property Name="LevelNo">0</Property>
      <Property Name="Occurs">0</Property>
      <Property Name="PadNulls">0</Property>
      <Property Name="SignOption">0</Property>
      <Property Name="SortingOrder">0</Property>
      <Property Name="ArrayHandling">0</Property>
      <Property Name="SyncIndicator">0</Property>
      <Property Name="PadChar"/>
      <Property Name="ExtendedPrecision">0</Property>
      <Property Name="TaggedSubrec">0</Property>
      <Property Name="OccursVarying">0</Property>
      <Property Name="PKeyIsCaseless">0</Property>
      <Property Name="SCDPurpose">0</Property>
   </SubRecord>
</Collection>
```

Figure 2.20: Example of column structure

Chapter 3

Design

In this chapter, we will talk about the design of the module. This chapter is based on the previous chapter, where was performed a detailed analysis.

The module is divided into four parts:

- Connector module
- Connector model module
- Connector resolver module
- Dataflow generator module

We can see relations between modules in figure 3.1. Manta element on the figure is not one module, but several different modules from the Manta project. Such a strong dependency is required by non-functional requirement number two.

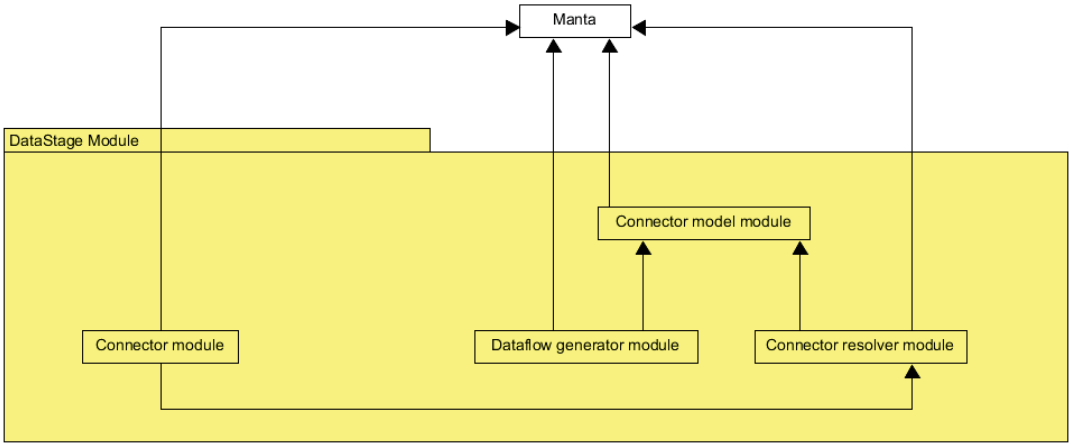

Figure 3.1: DataStage module structure

In figure 3.2, we can see the sequence diagram, which shows us communication between Manta and DataStage module. Firstly Manta calls method readFile(File file) from connector module, more specifically DataStageXmlReader class. This method reads an XML file into the memory, and then passes it to the connector resolver module, specifically Project class constructor, which transforms XML into an internal representation. After that, the readFile method returns the IProject object. Manta calls dataflow generator module, more specifically DataStageDataflowTask class, which provides analysis of the internal representation of the XML and generates graph.

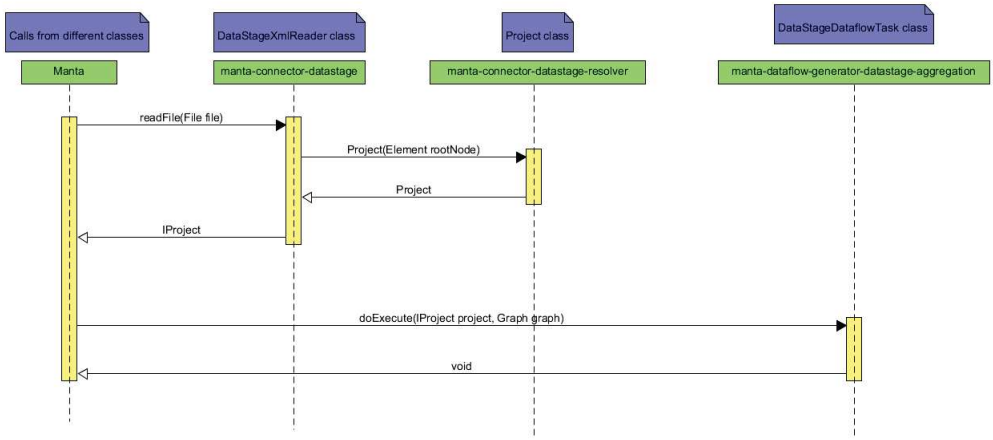

Figure 3.2: Sequence diagram of calls from Manta to DataStage module

3.1 Connector module

The module read an XML input and then pass the XML structure loaded into the memory to connector resolver module. Resolver module returns an object, which implements interface IProject with all essential data from XML. Module contains the only class DataStageXmlReader which extends AbstractFileInputReader abstract class from the Manta project. Extending this abstract class allows us to fulfill non-functional requirement number two.

3.2 Connector model module

It contains definitions of interfaces and enums for stage and variable types. The module provides a level of abstraction between Manta and connector resolver and dataflow generator modules. The IProject interface now is implemented just by Project class, which read data from XML to the classes, but it is possible to create one more implementation, which will read data from

the dsx file. Using the IProject interface allows us to fulfill non-functional requirement number three partly. Also, there are IParallelJob, IServerJob, and ISequenceJob interfaces, but just IParallelJob has a proper implementation because it was chosen as the most relevant at this time. So in the future, if necessary, we can extend module with further analyzers.

3.3 Connector resolver module

This module provides transformation of the XML export file from DataStage into internal representation for more convenient further analysis. This module provides syntactic and semantic analysis of the XML file, so it is allows us partly fulfill functional requirement number one. Syntactic analysis is provided using the dom4j library, and semantic analysis is implemented in the module. It allows us to fulfill functional requirement number three.

The module contains Java classes that implement interfaces from the connector model module. The main class, which encapsulates all the other, is Project class. In figure 3.3, we can see the structure of class encapsulations.

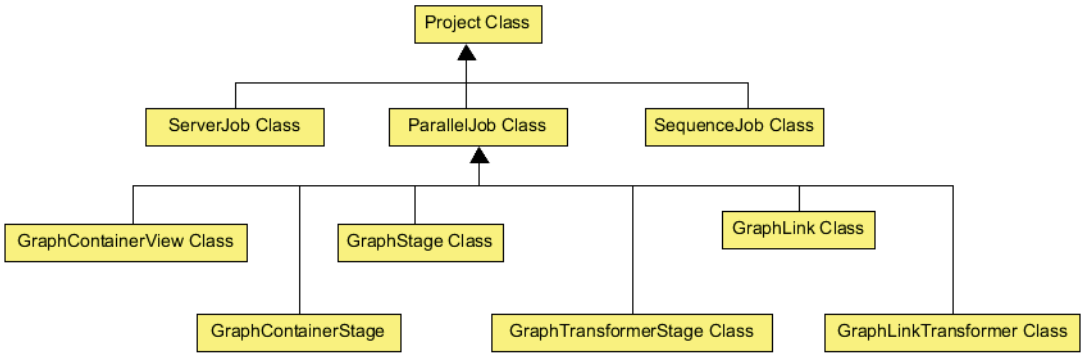

Figure 3.3: Structure of class encapsulations

Encapsulate classes into each other allows us access data necessary for analysis and graph generation in a more convenient way. The main reason for using this additional layer with the internal representation of the export file is the possibility to add support of other input files, without having to write one more analyzer. So, for instance, we can have as input ten different types of files or any other sources, which we transform to internal representation and use the only analyzer.

In figure 3.4, we can see interfaces inheritance hierarchy diagram. IElement is a common interface for all other interfaces. It merely returns the name of the element, what is common functionality for all elements. For information about the functionality of the rest of the interfaces, please refer to the source code.

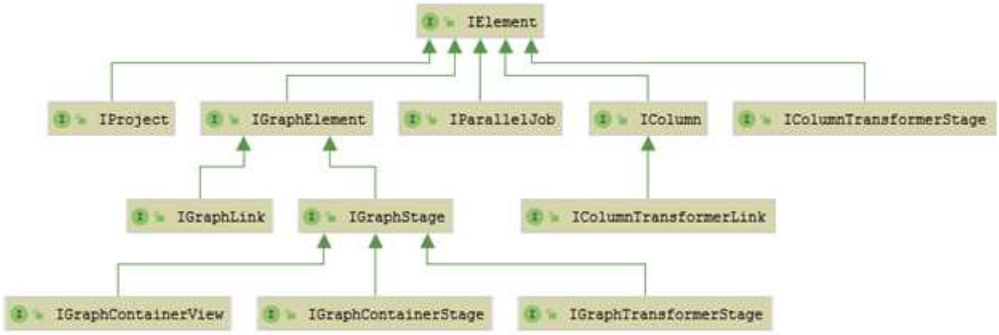

Figure 3.4: Interfaces inheritance hierarchy diagram

3.4 Dataflow generator module

The purpose of the module is analyzing the internal representation of the XML file received from the connector resolver module, and then the generation of the graph based on the previous analysis, so this module allows us fulfill functional requirement number one. Also, this module fulfill functional requirement number two, because during the analysis we distinguishes between direct and filter flows so we will use the right type of edge during graph building.

3.4.1 Graph

We get an instance of the Graph class from the Manta and then fill it with nodes and edges during project analyzing. The graph in Manta is a non-simple directed graph in which loops and multiple edges between any two vertices are permitted. This graph has a slightly different structure than the usual graph, namely nodes may contain other nodes. Using them, we create such a structure, as we can see in figure 3.5. Name and Surname are column nodes, File stage and Sort stage are nodes responsible for the representation of stages in the graph. The Job node represents a particular job. A DataStage node represents the DataStage tool.

3.4.2 Workflow

The main class, which serves as a communication layer between Manta and this module, is DataStageDataflowTask. The class extends AbstractGraphTask, an abstract class from the Manta project, so extending this abstract class allows us to fulfill non-functional requirement number two.

In figure 3.6, we can see a simplified sequence diagram of the module, with all main method calls. DataStageDataflowTask calls method buildGraphHelper from GraphHelperBuilder class, which implements builder pattern,

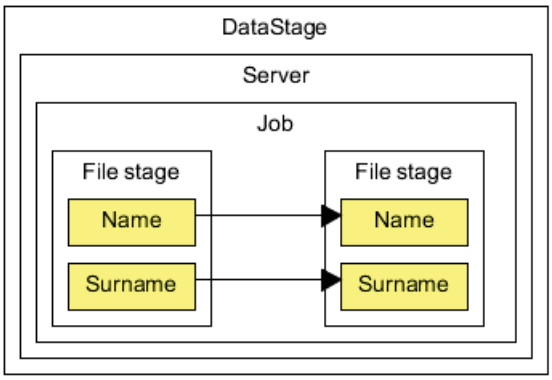

Figure 3.5: Example of the graph structure

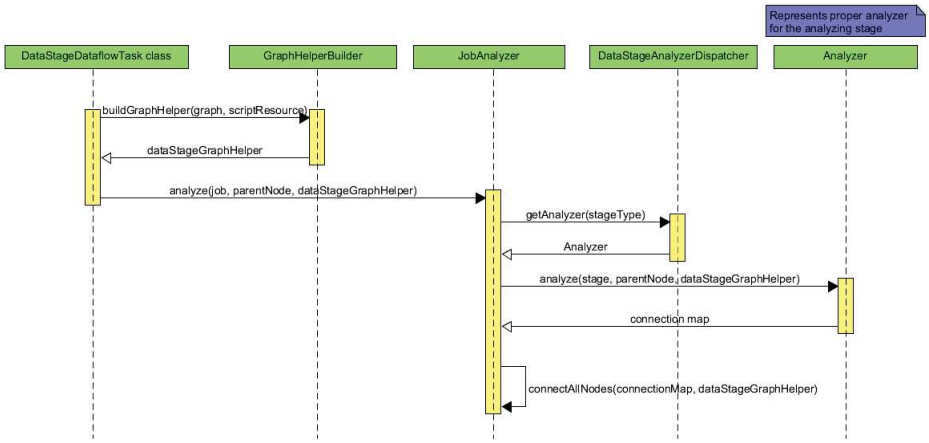

Figure 3.6: Sequence diagram of the generator module

with graph received from Manta and DataStage script resource. Script resource is an instance of ResourceImpl class, which represents a source of nodes and edges in the graph. The buildGraphHelper method returns an instance of DataStageGraphHelper class, which extends AbstractGraphHelper from the Manta, so this is one more thing which allows us to fulfill non-functional requirement number two. Builder construct instance with all objects needs for further graph building, namely DataflowQueryService, NodeCreator and graph objects, all these objects were gotten from Manta. DataflowQueryService serves to connect to different databases, such as Oracle, Netezza, Db2 and others. NodeCreator makes it easier to create file and table nodes with the path structure. For instance, we have file text.txt with the following path C:/Users/username/files/text.txt, and in the graph, we will have the following structure as in figure 3.7. NodeCreate parses path to the file and creates a

proper structure.

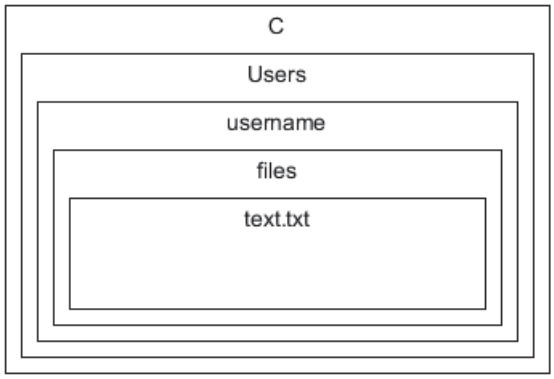

Figure 3.7: Path structure C:/Users/username/files/text.txt

DataStageGraphHelper class implements a facade pattern and simplify graph building. DataStageDataflowTask pass analyzing of each job from the export file, to the JobAnalyzer class. JobAnalyzer calls a getAnalyzer method from DataStageAnalyzerDispatcher, which returns proper analyzer for the analyzing stage. After that, JobAnalyzer calls the analyze method from the analyzer, which returns the connection map. Based on the structure of links, which we can see in figure 2.11, we will provide late binding of stages, in other words, we will add edges between columns in the stages on the end of job analyzing. During stage analyzing we cannot instantly add an edge between two nodes in the graph, because of the lack of guarantees that the source stage has already been analyzed. As an option, we can provide topological sort above stages and analyze them one by one, but it is no need complication, and there are no advantages of this solution. So we apply another approach to this problem, more specifically, we will analyze stages one by one and gradually collect information need for late binding and then, on the end of job analyzing, we will connect all columns at once. JobAnalyzer class calls method connectAllNodes from itself and provides late binding of all columns.

3.4.3 Stage analyzers

In figure 3.8 we can see the class diagram of analyzers. Each analyzer class has the following name format: *Analyzer, where * is the name of the analyzing stage, so, for instance, sort stage analyzer has the next name: SortAnalyzer. For illustrative purposes, there are elements in the figure with three dots, which means, that there are more classes inherit parent class. Common abstract class for all analyzers is AbstractStageAnalyzer, it contains commons methods, which are used during analyzing of any stage. All processing stages analyzers inherit AbstractStageAnalyzer, they do not use any specific

methods, so it is no need to provide a particular abstract parent for them. AbstractChangeAnalzer is an abstract class parent for ChangeApplyAnalyzer and ChangeCaptureAnalyzer processing analyzers, these two analyzers have many methods in common, so it was decided to extract common methods to separate abstract class. AbstractFileConnectorAnalyzer is an abstract parent class for all file analyzers, and AbstractDatabaseConnectorAnalyzer is an abstract parent class for all database connector analyzers. In the case of the meeting stage, which we do not support, UnknowComponentAnalyzer will provide analysis. This analyzer connects all input columns with all output columns. There are 32 processing analyzers, 11 file connector analyzers, and 22 database connector analyzers.

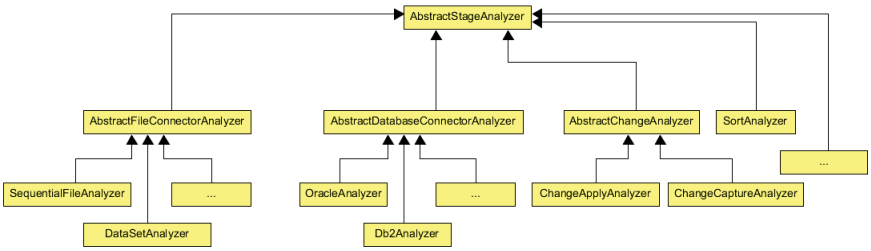

Figure 3.8: Class Diagram of analyzers

3.4.4 Transformer stage analyzer

During the transformer stage analyzing we are challenged to analyze expressions for determining data flows. For this purpose, we are using parser generated by ANTLR 3. The Expression format was found in DataStage documentation [30], but it was not ready for use in ANTLR. It contains left recursion, and it has another format that needs ANTLR. Based on this format, it was created lexer and parser grammars. ANTLR used both of them during parser generation. The Transformer stage analyzer passes an expression to the TransformerExpressionResolver class for analyzing. This class, in turn, use classes generated by ANTLR 3, namely TransformerLexer and TransformerParser classes, for expression analyzing. It is possible to use predefined functions in expressions and depends on the function, parameters that we pass can have direct or filter flow. In this work, all parameters, which we pass are considered as direct flows. TransformerExpressionResolver returns variable and source column dependencies for resolving. For instance, we have the following expression: Link.Name + username_variable, we get from the expression Link.Name source column and internal variable username_variable. We can instantly use source column and add it to the map for the late binding, but in the case of a variable, first, we have to get source columns on which it depends, and then we can add them to the map.

3.4.5 Database connectors analyzers

Most of the database connectors have properties in XML format. It does not matter if we export jobs in XML or DSX format, in any case, database properties such as server name, SQL query and others will be located in XML structure. For extracting useful data from the inner XML, we are using the XmlParser class. It extracts properties need for connecting to the database, table names and SQL queries. For connecting to databases, we are using DataflowQueryService class; this class provides us Manta. If Manta knows about the database, in other words, Manta has already analyzed the database; then we can pass the name of the table or SQL query to DataflowQueryService, and it will return part of the graph with the database. Because of the complexity of the solution, in this work, we do not support cases when the database connector stage serves as the destination of the data flow and, uses SQL query for insertion of the data, all other cases are supported. So we support the cases when the stage is the source, and we use SQL query or table name to get data and when the stage is a destination, and we use table name to get data.

Chapter 4

Implementation

In this chapter, we will talk about the implementation part of the work. Due to the massive amount of code, we will discuss only essential parts of the implemented module. We will not discuss the connector model module, because there are not any critical implementation moments. In the implementation, we are using the most effective approaches and algorithms to fulfill non-functional requirement number one.

In each class, where logging is necessary, we get Logger object from the LoggerFactory and then use this object for logging info, warn and error messages to the log. LoggerFactory is the class from the slf4j library. Logging allows us to fulfill functional requirement number 4. Bellow, in the source code 4.1, we can see logger creating and process of the logging error message.

4.1 Connector module

The only class in this module is DataStageXmlReader. In the source code 4.2, we can see the method code. Method readFile at first reads a file into memory calling a readStream method from the XmlStreamReader class (inputReader is an instance of XmlStreamReader class), which returns an instance of Element class from the dom4j library. Then pass this instance to the constructor of the Project class, from the connector resolver module, and then returns it to the caller.

```
private static final Logger log
   = LoggerFactory.getLogger(DataflowTaskTest.class);
...
log.error("Could not get a job identifier. Job reading was stopped.");
```

Source code 4.1: Logger creation and error logging

```
protected IProject readFile(File file) throws IOException {
   if (!file.canRead()) {
      throw new IOException("It is no input file for reading Data
          Stage project on the way :"
            + file.getAbsolutePath());
   }

   log.info("Start analyzing the file : {}", file.getAbsolutePath());

   try (FileInputStream inputStream =
       FileUtils.openInputStream(file)) {
      Element rootNode = (Element)inputReader.readStream(inputStream,
          file.getName());

      return new Project(rootNode);
   } catch (IOException | IllegalArgumentException e) {
      throw new IOException("An error has occurred, when trying to
          read the file : "
            + file.getAbsolutePath(), e);
   }
}
```

Source code 4.2: Source code of the readFile method

4.2 Connector resolver module

Each class in this module is responsible for analysis of some part of the XML file. Objects receive in the constructor, as a parameter, instance of Element class from the dom4j library, which contains part of XML needed for the analysis and construction of the object. The Project class receives an Element object, which represents the tree structure of the XML file. To obtain the necessary data from the tree, we are using XPath expressions, which allows us to get the necessary data effectively. Method selectSingleNode from the Element class, receive as a parameter XPath expression and returns an instance of the Node class. Also, the Element class has method selectNodes, which returns a list of nodes. Method getStringValue called on the instance of Node class returns us the value of the node. We can see the source code of the Project class constructor in source code 4.3 and GraphLink class constructor in source code 4.4.

```
public Project(Element rootNode) throws IllegalArgumentException,
    NullPointerException {
   if(rootNode == null) {
      throw new NullPointerException("Root of XML file can not be
          Null");
   }

   Element nodeProjectName = (Element)
       rootNode.selectSingleNode("Header");
   if(nodeProjectName == null) {
      throw new IllegalArgumentException("Unexpected XML structure.
          Can not find header");
   }

   Attribute atr = nodeProjectName.attribute("ServerName");
   if(atr == null) {
      throw new IllegalArgumentException("Unexpected XML structure.
          Can not find 'ServerName' attribute");
   }
   serverName = atr.getStringValue();

   List<Element> listOfJobs = rootNode.selectNodes("Job");
   for (Element job : listOfJobs) {
      addJob(job);
   }
}
```

Source code 4.3: Source code of the Project class constructor

```
GraphLink(Element xmlNode, AbstractElement parent) {
   super(xmlNode, parent);

   Node partnersNode =
       xmlNode.selectSingleNode("Property[@Name='Partner']");
   if(partnersNode != null) {
      String[] inputPinsArray =
          partnersNode.getStringValue().split(Pattern.quote("|"));
      partnerList = Arrays.asList(inputPinsArray);
   }

   addColumns(xmlNode);
}
```

Source code 4.4: Source code of the GraphLink class constructor

```
protected void doExecute(IProject project, Graph graph) {

   DataStageGraphHelper gh = builder.buildGraphHelper(graph,
       getScriptResource());

   Node projectRoot = gh.addNode(project.getName(),
       NodeType.DATASTAGE_SERVER, null,
         getScriptResource());

   for(IParallelJob job : project.getParallelJobs()) {
      log.info("START PARALLEL JOB ANALYZING : {}", job.getName());
      try {
         jobAnalyzer.analyzeParallelJob(job, projectRoot, gh);
      } catch (Throwable ex) {
         log.error("Unsuccessful parallel job analyzing. Job name : "
             + job.getName(), ex);
      }
      log.info("END PARALLEL JOB ANALYZING : {}", job.getName());
   }

}
```

Source code 4.5: Source code of doExecute method from the DataStageDataflowTask class

4.3 Dataflow generator module

4.3.1 Workflow

Analyzing the internal representation of the XML export file, namely instance of the class, which implements the IProject interface, starts in the doExecute method from the DataStageDataflowTask class. The method we can see in source code 4.5. The caller of the method is Manta, and it passes to the method as parameter instances of the project and the graph. Now the method provides analyzing of the parallel job; it does not support server and sequence job analyzing. Call of this method is located in the try catch block because we want to keep it analyzing jobs despite problems with analyzing some of them. The method pass is analyzing of each job to the analyzeParallelJob method from the JobAnalyzer class.

Method analyzeParallelJob provides analyzing of parallel jobs. We can see the method in source code 4.6. In the first instance, it creates a node in the graph, which represents analyzing the job. After that, it creates a map, which will be later used for binding of the stages. The map contains the destination column node as a key, and instance of FlowConnector class as a value. FlowConnector class contains all the necessary information for binding stages. In the source code 4.7 we can see the content of FlowConnector class

```
public void analyzeParallelJob(IParallelJob job, Node parentNode,
    DataStageGraphHelper gh) {
   Node jobRoot = gh.addNode(job.getName(),
       NodeType.DATASTAGE_PARALLEL_JOB, parentNode);

   MultiValuedMap<Node, FlowConnector> connectionMap = new
       ArrayListValuedHashMap<>();

   for (IGraphStage stage : job.getStagesById().values()) {
      log.info("START STAGE ANALYZING: {}", stage.getName());
      try {
         connectionMap.putAll(analyzerDispatcher
         .getAnalyzer(stage.getStageType()).analyze(stage, jobRoot,
             gh));
      } catch (Throwable ex) {
         log.error("Unsuccessful stage analyzing : {}",
             stage.getName(), ex);
      }
      log.info("END STAGE ANALYZING: {}", stage.getName());
   }

   connectAllNodes(connectionMap, gh);
}
```

Source code 4.6: Method analyzeParallelJob from JobAnalyzer class

```
private IGraphStage sourceStage;    // source stage where start of
    the edge is located
private String    sourceColumnName; // name of start column of the
    edge in the source stage
private String    linkName;         // name of input link to
    analyzing stage
private Edge.Type type;             // type of edge Direct or Filter
```

Source code 4.7: Source code of the FlowConnector class

with descriptions of its content.

After that, we get an analyzer using the getAnalyzer method from the instance of the DataStageAnalyzerDispatcher class. The source code of the method we can see below in the source code 4.8. We are looking for a proper analyzer by type of the stage; in case of lack of analyzer, the method returns a particular analyzer for unknown or not supported stages.

We got an analyzer, so we call the analyze method from it. The method returns a map with the necessary information for late binding, and then we merge this with that map we defined previously. When all stages have been analyzed, we call a connectAllNodes method, which connects stages.

```
AbstractStageAnalyzer getAnalyzer(StageType stageType) {
   AbstractStageAnalyzer analyzer = analyzers.get(stageType);
   if(analyzer == null) {
      analyzer = analyzers.get(StageType.UNKNOWN);
   }

   return analyzer;
}
```

Source code 4.8: Method getAnalyzer from DataStageAnalyzerDispatcher class

```
// based on knowledge that sort stage has only one input and output
    link
// we can find source stage and input link immediately
IGraphStage sourceStage = getSourceStage(stage);
String inputLinkName = getFirstInputLinkName(stage);
if(sourceStage == null) {
   log.error("It is not any input flow to the stage : {}, in the job
       : {}\n " +
             "Stage can not be analyzed, so graph can be inaccurate",
         stage.getName(), stage.getParent().getName());
   return result;
}
```

Source code 4.9: Calling the getSourceStage method during analyzing of the sort stage

4.3.2 Structure of analyzers

Here we will talk about an example of analyzing some of the stages, more specifically analyzing of sort stage. We will not discuss each analyzer, because the main idea in all analyzers is the same. Analyzing each stage starts with adding a node to the graph, which represents the analyzing stage; for this, we are using the buildNode method from the instance of DataStageGraphHelper class. After we are using the getSourceStage method if we know for sure, that stage could have the only one input link. Otherwise, we are using the indSourceStage method for each input link. We can see the calling of the getSourceStage method in source code 4.9.

For sort stage analyzing, it is necessary to know which columns were used as key columns. We get key columns from the stage in the next format:

```
\(2)\(2)0\(1)\(3)key\(2)Surname\(2)0\(1)\(3)\(3)asc\desc\(2)asc
\(2)0\(1)\(3)\(3)sorted\clustered\(2) \(2)0
```

The only useful information for us from this string is Surname. The same

```
// go through every column and create node of proper type and add it
    to the graph
for (IGraphLink outputLink : stage.getOutputPinsList()) {
   for (IColumn column : outputLink.getColumnsList()) {
      Node columnNode = gh.addNode(column.getName(),
          NodeType.COLUMN_FLOW, sortStageNode);

      // if source column string is not null, then add direct flow
      String sourceColumnName =
          getSourceColumnName(column.getSourceColumn());

      if(sourceColumnName != null) {
         result.put(columnNode, new FlowConnector(sourceStage,
             sourceColumnName, inputLinkName, DIRECT));

         // add filter flow to the graph
         // connect every key column from source stage with every no
             key column in destination
         for(String key : keyColumnsSet) {
            if(!sourceColumnName.equals(key)) {
               result.put(columnNode, new FlowConnector(sourceStage,
                   key, inputLinkName, FILTER));
            }
         }
      }
   }
}
```

Source code 4.10: Going through output links during analyzing of sort stage

ugly format of properties is contained in XML or dsx file. AbstractStageAnalyzer abstract class, which is parent class for all analyzers, contains particular methods to handle such formats.

We have gathered all the necessary information for sort stage analyzing, so now we are going through every output link and creating nodes, which represents columns in the stage. We can this process bellow in source code 4.10. If the column was not created in the analyzing stage, then it has the derived value, which has such format: Link.Column or Function(Link.Name). The Function is the name of the function, which was used to get the value of the column, the Link is the name of the input link to the analyzing stage, and the Column is the name of the origin column, from which it was derived, from source stage. We get the origin column name using the getSourceColumnName method. After that, we add all necessary information for later binding of the direct and filter flow edges to the resulting map, which we return at the end of the analyzing. For more information about each of the analyzers, please refer to the source code.

```
public Dependencies getDependencies(String expression) throws
    ParseException {
   CommonTree tree = getTree(expression);
   return parseTree(tree, Edge.Type.DIRECT);
}
```

Source code 4.11: Method getDependencies source code from TransformerExpressionResolver class

4.3.3 Transformer stage analyzer

In the transformer stage analyzer, the main idea of analyzing is the same as in all other analyzers, which we have discussed in the previous subchapter, so this why we will focus on specific moments of the transformer stage analyzer.

The Transformer analyzer firstly resolving inner variables expressions, namely loop and stage variables, and then resolve output column expressions. It passes the resolving task to the getDependencies method from the TransformerExpressionResolver class. In the source code 4.11, we can see the source code of the method. In the beginning, it gets an abstract syntax tree from the expression and then passes it to the parseTree method for providing tree parsing.

In source code 4.12, we can see the source code of the getTree method. Firstly we create a char stream from the analyzing expression; then we are using lexer for parsing char stream and from getting tokens. After we use the parser, which analyzes a stream of tokens and creates the tree from it. In case of any problem during analyzing, we are throwing an exception with problem expression for later logging of the error. ANTLR generated TransformerLexer and TransformerParser classes.

Method parseTree uses recursion for analyzing the tree. In source codes 4.13 and 4.14 bellow, we can see grammar definition of If_Then_Else_Expression and Link_And_Name elements, also we can see parts of the source code of the corresponding method in source codes 4.15 and 4.16. Based on the type of node, we provide proper action. At the moment we do not support determining the type of the flow of function parameters by the name of the function. Link_And_Name and Variable nodes are terminal nodes, and we add their values to the dependencies with the type of flow; all the other nodes we are passing to the next level of recursion.

After that, when we get all dependencies, we add them to the map for the late binding of the stages. Link_And_Name dependencies are added instantly, Variable dependencies firstly resolved and then added to the map. For resolving we are using the resolver method, which we can see in source code 4.17. To prevent stack overflow due to infinite recursion, we set the maximal depth of the recursion on the start and then reduce it with each call of it. For instance, we have the MAIN variable which depends on two other variables

```
private CommonTree getTree(String expression) throws ParseException {
   CharStream charStream = new ANTLRStringStream(expression);

   TransformerLexer lexer = new TransformerLexer(charStream);
   CommonTokenStream tokens = new CommonTokenStream(lexer);

   TransformerParser parser = new TransformerParser(tokens);
   try {
      return (CommonTree) parser.start_rule().getTree();
   } catch (RecognitionException e) {
      throw new ParseException(MessageFormat
            .format("Cannot parse expression \"{0}\"", expression),
                e.line);
   }
}
```

Source code 4.12: The source code of the getTree method

```
if_then_else_expression :
   IF expression THEN expression ELSE expression;
```

Source code 4.13: If_Then_Else_Expression part of the grammar parser definition

```
Link_And_Name :
   LINK_AND_NAME
      -> ^(Link_And_Name LINK_AND_NAME);
```

Source code 4.14: Link_And_Name part of the grammar parser definition

```
case TransformerParser.If_Then_Else_Expression: {
   if(tree.getChildCount() >= 2) {
      dependencies.merge(parseTree((CommonTree)tree.getChild(0),
         Edge.Type.FILTER));
      dependencies.merge(parseTree((CommonTree)tree.getChild(1),
         type));
   }
   if(tree.getChildCount() == 3) {
      dependencies.merge(parseTree((CommonTree)tree.getChild(2),
         type));
   }
   break;
}
```

Source code 4.15: If_Then_Else_Expression part of the source code of the parseTree method

```
case TransformerParser.Link_And_Name: {
   if(tree.getChildCount() == 1) {
      String linkAndName = tree.getChild(0).getText();
      dependencies.addDependence(linkAndName, type,
          Dependencies.DependenceType.SOURCE_COLUMN);
   }
   break;
}
```

Source code 4.16: Link_And_Name part of the source code of the parseTree method

FIRST and SECOND. The MAIN expression may look like FIRST + SECOND. Variable FIRST has the following format: Link1.Column1 and variable SECOND has this: Link2.Column2. So, as a result, variable MAIN depends on Link1.Column1 and Link2.Column2 columns.

4.3.4 Database connector analyzer

The main idea of database connectors analyzing try to get nodes for connecting to database from dataflow query service or create them by ourselves. In source code 4.18, we can see part of the source code of the proceedConnectionAsSource method from the AbstractDatabaseConnectorAnalyzer class. If dataflow query service returns us nodes, which represent database table or view, then we can connect these nodes with the columns in the analyzing stage. Otherwise, we construct a similar structure by ourselves and fills it with all the information about the connection we have. This structure will represent a database table, but in fact, it is just a dummy, but it allows for the user to understand more or less data flow. If we do not know some information, such as the server or schema name, then we use some stub values. As a result, we will get such a structure as in figure 4.1.

```
private Dependencies resolver(Map<String, Dependencies>
    stageAndLoopVariables, String dependenceName,
                    Edge.Type dependenceFlowType, int recDeep) {

   Dependencies result = new Dependencies();

   Dependencies dependencies =
       stageAndLoopVariables.get(dependenceName);
   dependencies.getDependencies().forEach(dependence -> {
      switch (dependence.getDependenceType()) {
         case VARIABLE: {
            if(recDeep > 0) {
               result.merge(
                     resolver(stageAndLoopVariables,
                         dependence.getName(), dependenceFlowType,
                           recDeep - 1));
            }
            break;
         }
         case SOURCE_COLUMN: {
            result.addDependence(
                  dependence.getName(), dependenceFlowType,
                      Dependencies.DependenceType.SOURCE_COLUMN);
            break;
         }
         default: {
            log.error("Unknown type of dependence type was meet.
                Analyzing can be inaccurate!");
         }
      }
   });

   return result;
}
```

Source code 4.17: The source code of the resolver method

```
Graph dbGraph = new GraphImpl(null);
for (Map.Entry<SqlType, String> entry : sqlStatements.entrySet()) {
    SqlType sqlType = entry.getKey();
    String sql = entry.getValue();
    try {
        gh.queryService().getDataFlow(null,
                stage.getParent().getName() + "." + stage.getName(),
                "Query:" + stage.getName(),
                sql, connection, dbGraph);
    } catch (RuntimeException e) {
        log.error("Failed to parse SQL query : \"{}\", type of query :
            \"{}\", in the stage \"{}\"",
                sqlType, sql, stage.getName(), e);
    }
}

if (dbGraph.isEmpty()) {
    proceedManually(stage, connection, gh, stageColumns,
        analyzeDirection, tableNames);
    return;
}
```

Source code 4.18: Getting of database nodes

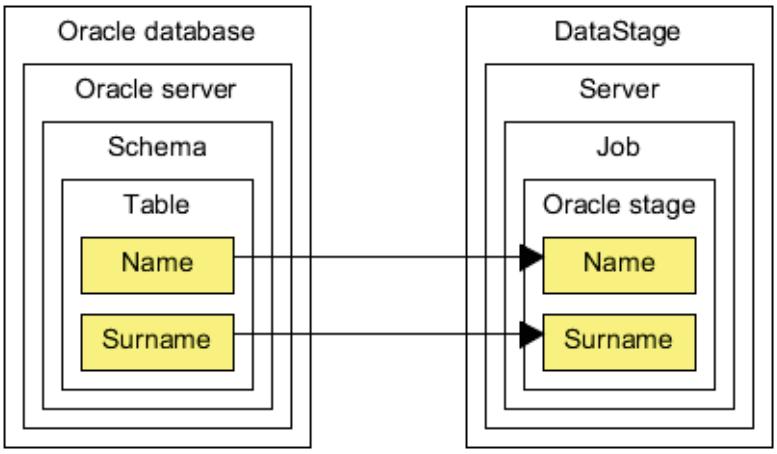

Figure 4.1: Example of connection to Oracle database

Chapter 5

Testing

In this chapter, we will talk about the testing of the module. For providing automated tests, we are using the JUnit library. Tests are run with every Maven build or manually on demand. Tests ensure that the module works properly. The module has a large number of different tests so that we will discuss them briefly.

5.1 Connector module

It is the only test class in this module, namely DataStageXmlReaderTest. There are two tests in this module, and both of them test the readFile method from the DataStageXmlReader class. The first one tests if readFile has not a null return value when we provide a path to the existing file. The second one tests if a method throws IOException when we provide a path to a nonexistent file.

5.2 Connector model module

It is the only test class in this module, namely VariableTypeTest, with one test. It tests if the getVariableType method from enum VariableType determines the type of variable correctly.

5.3 Connector resolver module

There are three classes with tests in this module, namely ColumnTest, GraphStageTest, and ProjectTest. Each of them is responsible for testing a particular functionality. ColumnTest class has the following tests:

- testConstructorOk Here we are testing the correctness of Column constructor. We provide a part of the XML file tree, which is necessary

for Column constructor working, and then we are checking if properties from the XML, have been read properly.

- testGetNullableType This test method controls the correctness of the getNullableType method from the NullableEnum class. We supply the method with different values and check the output.

GraphStageTest class has the only test method. The method ensures the correctness of GraphStage class constructor. We give the class constructor the proper part of the XML tree, and then we test if attributes have been read right.

ProjectTest class has 6 test methods. Two of them are testing the correctness of the getJobType method from the Project class. The remaining tests control the work of the Project constructor.

Class contains following tests:

- getJobTypeTestOk
- getJobTypeTestFail
- testSurrogateKeyStageOk
- testSwitchStageOk
- testTransformerStageOk
- testContainerStageOk

5.4 Dataflow generator module

Each analyzer or helper, which has some functionality to test, has its test class, which ensures functionality. We will discuss only the most exciting of them.

TransformerExpressionResolverTest class provides tests of the Transfirmer-ExpressionResolver class, more specifically getDependencies method. There are 17 tests, which ensure the functionality of the method. We supply it with potentially troublesome expressions.

DataflowTaskTest class provides tests of each stage, which we support. Firstly we provide manual testing and visualize graph using the Graphviz library. An example of visualization we can see in figure 5.1. We compare our expectations with visualization, and if they match, then we provide an export of the graph to the text file, which will serve as the expectation file. After that, we can automate our test. Each time we run the test, we export the graph to the text file and compare it with expectation file, if there are the same, it means that the analyzer works appropriately. Otherwise, we can visualize expected and actual text files and compare them.

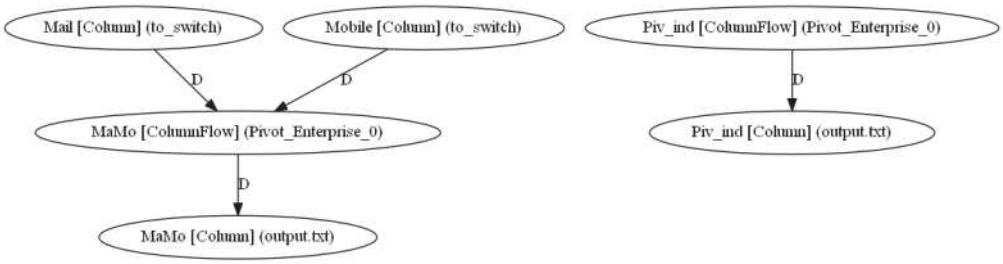

Figure 5.1: Graph visualization

Conclusion

This work aimed to design and implement a functional module prototype that performs syntactic and semantic analysis of tasks in IBM DataStage and then uses its result for data flow analysis. This aim was broken up into three smaller, namely analytical, designing and an implementation part.

In the analytical part, we had to become acquainted with IBM InfoSphere DataStage and Manta project. Also, choose the functionality to support. I got acquainted with IBM InfoSphere DataStage, it is functional and also with the Manta project and its structure. After that, the essential functions to support were chosen. Between types of jobs were chosen the parallel task, so-called Parallel Jobs, as the most exciting and relevant at this time, because this type is used now the most. Between stages were chosen all Processing stages, which provide different transformations over data, for instance, sorting, filtering and copying data. All Database connector stages were chosen to support; they provide connections to different databases, such as Oracle, Db2, Teradata and so on. Also, all file connector stages were chosen; they are using different types of files as data sources. So the analytical part was successfully done, and all parts of it were completed.

In designing part based on the result obtained in the analytical part, we had to make a design concept of the module for the Manta project, which will provide IBM DataStage task processing. During designing, we had to use the Manta project structure. The design concept was made based on previous analysis and design ensures trouble-free joining of the module to the Manta project.

In the implementation portion based on the design made in the previous part, we had to implement the module, document and test it. The module was implemented, documented and successfully tested. The module provides syntactic and semantic analysis; after that result of the analysis is used during the generation of the data flow graph. The module contains documentation in Javadoc format and more than 100 tests, which test all components of the module and ensure its functionality.

Following functional is not supported by the module:

- Server and Sequence Jobs
- Stages that do not belong to Processing, Database connectors or File connector stages. The local container stage is also supported.
- Determining the type of flow in the functions in Transformer stage expressions.
- Analyzing insert SQL statements in Database connectors.

In case of need, it is possible to expand the module with the functionality listed above.

Bibliography

[1] Marr, B. How Much Data Do We Create Every Day? The Mind-Blowing Stats Everyone Should Read. Mar 2019, [online] [online] [Accessed 2019-05-14]. Available from: https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/#2f73744c60ba

[2] IBM InfoSphere DataStage. May 2019, [online] [Accessed 2019-05-14]. Available from: https://www.ibm.com/us-en/marketplace/datastage

[3] Hoang, N. Data Lineage Helps Drive Business Value. Aug 2018, [online] [Accessed 2019-05-14]. Available from: https://www.trifacta.com/data-lineage/

[4] Manta Tools. [online] [Accessed 2019-05-14]. Available from: http://www.czechstartups.org/en/startups/manta-tools/

[5] Sedlák, J. V Dejvicích roste další velká česká softwarová věc. Feb 2015, [online] [Accessed 2019-05-14]. Available from: https://connect.zive.cz/clanky/v-dejvicich-roste-dalsi-velka-ceska-softwarova-vec/sc-320-a-177282

[6] What Is ETL? [online] [Accessed 2019-05-14]. Available from: https://www.sas.com/en_us/insights/data-management/what-is-etl.html

[7] DataStage Tutorial: Beginner's Training. [online] [Accessed 2019-05-14]. Available from: https://www.guru99.com/datastage-tutorial.html

[8] Requirements Analysis - Requirements Analysis Process,Techniques. Apr 2019, [online] [Accessed 2019-05-14]. Available from: https://reqtest.com/requirements-blog/requirements-analysis/

[9] Laplante, P. A. *What every engineer should know about software engineering*. CRC, 2007.

[10] Gosling, J. *The Java language specification: Java SE 8 edition*. Addison-Wesley, 2014.

[11] Java High-Performance JVM. [online] [Accessed 2019-05-14]. Available from: https://jrebel.com/rebellabs/10-reasons-why-java-rocks-more-than-ever-part-5-high-performance-jvm/

[12] Maven. [online] [Accessed 2019-05-14]. Available from: https://maven.apache.org/index.html

[13] JUnit 4. [online] [Accessed 2019-05-14]. Available from: https://junit.org/junit4/

[14] Spring Projects. [online] [Accessed 2019-05-14]. Available from: https://spring.io/projects/spring-framework

[15] Five minute introduction to ANTLR 3. [online] [Accessed 2019-05-14]. Available from: https://theantlrguy.atlassian.net/wiki/spaces/ANTLR3/pages/2687102/FiveminuteintroductiontoANTLR3

[16] Simple Logging Facade for Java (SLF4J). [online] [Accessed 2019-05-14]. Available from: https://www.slf4j.org/

[17] dom4j. dom4j/dom4j. [online] [Accessed 2019-05-14]. Available from: https://github.com/dom4j/dom4j/wiki

[18] XML Path Language (XPath)Version 1.0. [online] [Accessed 2019-05-14]. Available from: https://www.w3.org/TR/1999/REC-xpath-19991116/#section-Introduction

[19] Information Server architecture and concepts. [online] [Accessed 2019-05-14]. Available from: https://www.ibm.com/support/knowledgecenter/SSZJPZ_11.7.0/com.ibm.swg.im.iis.productization.iisinfsv.overview.doc/topics/cisoarchoverview.html

[20] Barry, D. K. Service-Oriented Architecture (SOA) Definition. [online] [Accessed 2019-05-14]. Available from: https://www.service-architecture.com/articles/web-services/service-oriented_architecture_soa_definition.html

[21] Tiers and components. [online] [Accessed 2019-05-14]. Available from: https://www.ibm.com/support/knowledgecenter/SSZJPZ_11.7.0/com.ibm.swg.im.iis.productization.iisinfsv.overview.arch.doc/topics/wsisinst_pln_configurations.html

[22] Tier relationships. [online] [Accessed 2019-05-14]. Available from: https://www.ibm.com/support/knowledgecenter/SSZJPZ_11.7.0/com.ibm.swg.im.iis.productization.iisinfsv.overview.arch.doc/topics/wsisinst_arch_tierrelationships.html

[23] Difference between server jobs and parallel jobs in Datastage. [online] [Accessed 2019-05-14]. Available from: https://learndwconcepts.blogspot.com/2012/02/difference-between-server-jobs-and.html

[24] Alur, N.; Takahashi, C.; et al. *IBM Infosphere Datastage Data Flow and Job Design*. IBM, International Technical Support Organization, 2008, [online] [Accessed 2019-05-14]. Available from: https://www.redbooks.ibm.com/redbooks/pdfs/sg247576.pdf

[25] Runtime column propagation. [online] [Accessed 2019-05-14]. Available from: https://www.ibm.com/support/knowledgecenter/en/SSZJPZ_11.7.0/com.ibm.swg.im.iis.ds.parjob.dev.doc/topics/c_deeref_Runtime_Column_Propagation.html

[26] Supported connectors. [online] [Accessed 2019-05-14]. Available from: https://www.ibm.com/support/knowledgecenter/en/SSZJPZ_11.7.0/com.ibm.swg.im.iis.connect.nav.doc/containers/cont_iisinfsrv_connect.html

[27] Processing Data. [online] [Accessed 2019-05-14]. Available from: https://www.ibm.com/support/knowledgecenter/en/SSZJPZ_11.7.0/com.ibm.swg.im.iis.ds.parjob.dev.doc/topics/processingdata.html

[28] Defining local stage variables. [online] [Accessed 2019-05-14]. Available from: https://www.ibm.com/support/knowledgecenter/SSZJPZ_11.7.0/com.ibm.swg.im.iis.ds.parjob.dev.doc/topics/t_deeref_Defining_Local_Stage_Variables_transformer_stage.html

[29] Extensible Markup Language (XML). [online] [Accessed 2019-05-14]. Available from: https://www.w3.org/XML/

[30] Expression format. [online] [Accessed 2019-05-14]. Available from: https://www.ibm.com/support/knowledgecenter/SSZJPZ_11.7.0/com.ibm.swg.im.iis.ds.parjob.dev.doc/topics/r_deeref_Expression_Format_transformer_stage.html

Appendix A

Acronyms

- **AST** - Abstract Syntax Tree
- **XML** – Extensible markup language
- **ETL** – Extract Transform Load
- **IDE** – Integrated Development Environment
- **SQL** – Structured Query Language
- **PL/SQL** – Procedural Language/Structured Query Language
- **CSV** – Comma-separated values

Appendix B

Contents of CD

- `readme.txt` ..... the file with CD contents description
- `src` ..... the directory of source codes
  - `module` ..... the directory of DataStage module
  - `thesis` ..... the directory of LaTeX source codes of the thesis
    - `figures` ..... the thesis figures directory
    - `*.tex` ..... the LaTeX source code files of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the Bachelor thesis in PDF format