

David Stefan Sandru

# Dynamic File Import module

**Bachelor's Thesis**

Graz University of Technology

Knowledge Technologies Institute  
Head: Univ.-Prof. Dr. Lindstaedt

Supervisor: Dipl.-Ing. Dr.techn. Kern

Graz, January 2021

This document is set in Palatino, compiled with [pdfL<sup>A</sup>T<sub>E</sub>X2e](#) and [Biber](#).

The L<sup>A</sup>T<sub>E</sub>X template from Karl Voit is based on [KOMA script](#) and can be found online: <https://github.com/novoid/LaTeX-KOMA-template>

## Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, \_\_\_\_\_

Date

\_\_\_\_\_  
Signature

## Eidesstattliche Erklärung<sup>1</sup>

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am \_\_\_\_\_

Datum

\_\_\_\_\_  
Unterschrift

---

<sup>1</sup>Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008



# Abstract

Since long ago when trading companies formed, there was always a need to be competitive, to gain any advantage possible, to get that deal and secure a resource. Today, huge companies have taken a foothold in an international market, competing not only with other large corporations, but also with small local enterprises. This ubiquitous pressure to be better intensifies the need for any advantage an enterprise, big or small, can get in this cutthroat age of globalization. There are many technologies available for companies to achieve a bleeding edge over the others and the Dynamic Import Module (DIM) presented in this document is one such tool.

In an age where trading and processing data is an everyday affair, many people are required to carry out mundane and mind numbing tasks such as manually typing files into databases or creating spreadsheets to crunch some numbers. The DIM aims to speed up and automate this process, freeing up time for people to spend their time on meaningful work and giving their company a lead in other areas which require human creativity.

The Dynamic Import Module is a lightweight software module for Extracting, Transforming and Loading (ETL) data from files into SQL databases. There are many different ETL and ELT applications available for a myriad of different programming languages and frameworks, but the Dynamic Import Module is a Grails 2.2 specific plugin and for this particular framework the first of its kind.

As of now, the DIM is able to extract data from DSV - which includes CSV, TSV and whitespace delimited files - PRN, simple XML, SpreadsheetML, simple JSON and JSON-like files like YAML. Further, it is possible to configure the DIM so that certain lines are skipped, that data is verified that it is of a certain data type and data can even be specified to be optional or required.

As for transforming the data, the DIM provides a suite of methods from simple string manipulations like appending and replacing substrings, to numerical operations like calculating a sum/mean and carrying out basic arithmetic operations. Not only that, the DIM is also able cache peculiar data and insert it into other data sets, or to create one-to-one and one-to-many relations between different database tables. Last but not least, all of the transformation methods can be applied universally for all data sets with exclusions if need be, or only selectively for special data sets.

Whatever the case, the extracted and possibly transformed data can then be loaded into the database in any representation the user needs it to be, as long as the equivalent domain classes exist within the application.

# Contents

<b>Abstract</b>	<b>v</b>
<b>List of Abbreviations</b>	<b>xviii</b>
<b>1. Introduction</b>	<b>1</b>
<b>2. Related work</b>	<b>3</b>
<b>3. Methods</b>	<b>6</b>
3.1. Concepts	6
3.1.1. Extraction	6
3.1.1.1. Fixed-width files	7
3.1.1.2. Delimiter separated value files (DSV)	8
3.1.1.3. Tag based files	9
3.1.1.3.1. JSON files	10
3.1.1.3.2. XML files	12
3.1.2. Transform and load	13
3.1.2.1. From extraction to transformation	14
3.1.2.2. Transformation and loading process	15
3.2. Implementation	18
3.2.1. Helpful technologies	18
3.2.2. Exceptions	19
3.2.3. Extraction	20
3.2.3.1. FileParsingService	20
3.2.3.2. The base parser class <b>DynamicParser</b>	23
3.2.3.3. ColumnWidthParser	29
3.2.3.4. TokenSplitParser	33
3.2.3.5. SimpleTagParser	37
3.2.3.6. SimpleXMLParser	55

## Contents

3.2.4.	Transformation	60
3.2.4.1.	Transformation routines	60
3.2.4.2.	Transformation procedures	62
3.2.4.3.	TransformationService	66
3.2.4.4.	Transformation methods	74
3.2.4.4.1.	appendString	76
3.2.4.4.2.	prependString	77
3.2.4.4.3.	trimField	77
3.2.4.4.4.	splitStringField	78
3.2.4.4.5.	concatenateFields	80
3.2.4.4.6.	calculateSum	81
3.2.4.4.7.	calculateMean	82
3.2.4.4.8.	arithmeticOperation	84
3.2.4.4.9.	unaryArithmeticOperation	86
3.2.4.4.10.	regexReplace	87
3.2.4.4.11.	setValueFromOptionalValues	88
3.2.4.4.12.	setTimestamp	89
3.2.4.4.13.	createRelation	90
3.2.4.4.14.	createOneToManyRelation	92
3.2.4.4.15.	cacheInfoForCrossProcedure	94
3.2.4.4.16.	crossCreateRelation	97
3.2.4.4.17.	crossSetValue	99
3.2.4.5.	Loading methods	100
3.2.4.5.1.	identityTransfer	100
3.2.4.5.2.	identityTransferAndSave	101
3.2.4.5.3.	saveAllObjects	103
3.2.5.	User Interface (UI)	105
3.2.5.1.	Extraction UI	105
3.2.5.1.1.	ColumnWidthParser UI	106
3.2.5.1.2.	TokenSplitParser UI	107
3.2.5.1.3.	SimpleTagParser UI	109
3.2.5.1.4.	SimpleXMLParser UI	111
3.2.5.2.	Transformation and loading UI	112
3.2.5.2.1.	TransformationRoutine UI	112
3.2.5.2.2.	TransformationProcedure UI	114



<b>4. Discussion</b>	<b>117</b>
4.1. Limitations	117
4.1.1. General	117
4.1.2. Extraction	119
4.1.2.1. SimpleTagParser	120
4.1.2.2. SimpleXMLParser	120
4.1.3. User Interface	121
4.2. Future Work	122
4.2.1. General	122
4.2.2. Extraction	123
4.2.2.1. TokenSplitParser	124
4.2.2.2. SimpleTagParser	124
4.2.2.3. SimpleXMLParser	125
4.2.3. Transformation	126
4.2.4. User Interface	127
<b>A. Getting started</b>	<b>131</b>
<b>B. Use cases and examples</b>	<b>134</b>
B.1. Extraction examples	134
B.1.1. Fixed-width file example (handled by the Column-WidthParser)	135
B.1.2. DSV examples (handled by the TokenSplitParser)	138
B.1.2.1. Comma delimited file example (CSV)	138
B.1.2.2. Tab delimited file example (TSV)	140
B.1.2.3. Whitespace delimited file example	142
B.1.3. Tag-based file examples (JSON/YAML) (handled by the SimpleTagParser)	144
B.1.3.1. YAML example (a tag based file format)	151
B.1.4. XML examples handled by the SimpleXMLParser	153
B.1.4.1. Regular XML examples	153
B.1.4.2. SpreadsheetML examples	155
B.2. Transformation examples	157
B.2.1. "Update properties" examples	157
B.2.2. "Notable objects" examples	157
B.2.3. appendString example	158
B.2.4. prependString	160

## Contents

B.2.5.	trimField	161
B.2.6.	splitStringField	163
B.2.7.	concatenateFields	165
B.2.8.	calculateSum	167
B.2.9.	calculateMean	169
B.2.10.	arithmeticOperation	171
B.2.11.	unaryArithmeticOperation	175
B.2.12.	regexReplace	176
B.2.13.	setValueFromOptionalValues	181
B.2.14.	setTimestamp	183
B.2.15.	createRelation	185
B.2.16.	createOneToManyRelation	186
B.2.17.	cacheInfoForCrossProcedure	187
B.2.17.1.	crossSetValue	187
B.2.17.2.	crossCreateRelation	189

# List of Figures

3.1.	Example of a typical fixed-width file parsable by the DIM. Source is <a href="http://dailydoseofexcel.com">dailydoseofexcel.com</a> <sup>2</sup> . . . . .	7
3.2.	Example of a typical CSV file parsable by the DIM. Source is <a href="http://spatialkey.com">spatialkey.com</a> <sup>3</sup> . . . . .	8
3.3.	A file using whitespace as a delimiter and has a variable amount of columns with variable data types . . . . .	9
3.4.	A simple JSON file parsable by the DIM. Source is <a href="http://json.org">json.org</a> <sup>4</sup> . . . . .	10
3.5.	A simple YAML file with repeating values parsable by the DIM. Source is <a href="http://ibm.com">ibm.com</a> <sup>5</sup> . . . . .	11
3.6.	A problematic example of a YAML file that can not be parsed properly because of the “skills” list. Source is <a href="http://ansible.com">ansible.com</a> <sup>6</sup> . . . . .	11
3.7.	A simple XML file parsable by the DIM. Source is <a href="http://w3schools.com">w3schools.com</a> <sup>7</sup> . . . . .	13
3.8.	A SpreadsheetML file parsable by the DIM. Source is <a href="http://etutorials.org">etutorials.org</a> <sup>8</sup> . . . . .	13
3.9.	Example data structure with entries from the fixed-width file example. Picture taken from IntelliJ IDEA. . . . .	14
3.10.	Rough outline of the DIM’s workflow. . . . .	17
3.11.	Example of parsers being saved in the same DB table ‘dynamic_parser’, because of the base class. Database showcased is MySQL. . . . .	26
3.12.	Showcase of a file that can make use of the data type priority and optional setting. . . . .	37
3.13.	Example pseudo-object with superTag “CD” and with fields being “TITLE”, “ARTIST”, “COUNTRY”, “COMPANY”, “PRICE” and “YEAR.” Source is <a href="http://w3schools.com">w3schools.com</a> <sup>9</sup> . . . . .	58
3.14.	Three example pseudo-objects with superTag being “Row”, startBuffer being “2”, and the excelTag being “Data”. Fields are irrelevant for SpreadsheetML and can be named arbitrarily. Source is <a href="http://etutorials.org">etutorials.org</a> <sup>10</sup> . . . . .	58

## List of Figures

3.15. A valid XML, not parseable by the SimpleXMLParser, as there are no repeating pseudo-objects with unique names. . . . .	59
3.16. UML diagram of the DIMs domain objects and how they are related to each other. . . . .	64
3.17. UI of the ColumnWidthParser . . . . .	106
3.18. UI of the TokenSplitParser . . . . .	107
3.19. UI of the TokenSplitParser . . . . .	109
3.20. UI of the SimpleXMLParser . . . . .	111
3.21. Initial transformation routine UI. . . . .	113
3.22. Transformation routine UI with parser and object class submitted. . . . .	113
3.23. Initial transformation procedure UI. . . . .	115
3.24. Transformation procedure UI with the routine and method submitted. . . . .	115
4.1. Problematic example from json.org. <sup>11</sup> . . . . .	121
A.1. Example of the ColumnWidthParser's tag library employed in a simple page used for testing. . . . .	132
A.2. The rendered tag library in the resulting page. . . . .	132
B.1. Example fixed-width file called "FixedWidthExample2.prn", with only the end of the file being displayed. Source is daily-doseofexcel.com <sup>12</sup> . . . . .	135
B.2. The resulting pseudo-object maps of the "FixedWidthExample2.prn" file, with two such objects on display. . . . .	136
B.3. A collage of a functioning configuration of the ColumnWidthParser for the "FixedWidthExample2.prn" file. . . . .	137
B.4. A large CSV file example called "FL_insurance_sample.csv". Source is spatialkey.com <sup>13</sup> . . . . .	138
B.5. The resulting pseudo-object maps of the "FL_insurance_sample.csv" file, with the first object on display. . . . .	138
B.6. A collage of a functional configuration of the TokenSplitParser for the CSV file "FL_insurance_sample.csv". . . . .	139
B.7. A simple nonsensical TSV file example called "output-onlinerandomtools.tsv". Source is onlinerandomtools.com <sup>14</sup> . . . . .	140

## List of Figures

B.8. The resulting pseudo-object maps of the “output-onlinerandomtools.tsv” file, with the first object on display. . . . .	140
B.9. A collage of a functional configuration of the TokenSplitParser for the TSV file “output-onlinerandomtools.tsv”. . . . .	141
B.10. A simple nonsensical whitespace delimited file example called “whitespace_delimited_file.txt”. . . . .	142
B.11. The resulting pseudo-object maps of the “whitespace_delimited_file.txt” file, with two notable objects on display. Entries 2 to 7 were omitted. . . . .	142
B.12. A collage of a functional configuration of the TokenSplitParser for the whitespace delimited file “whitespace_delimited_file.txt”. . . . .	143
B.13. A remarkable JSON file example called “all_inclusive_test.json”. . . . .	144
B.14. A collage of a functional configuration of the SimpleTagParser for the JSON file “all_inclusive_test.json”. . . . .	145
B.15. The resulting pseudo-object maps of the “all_inclusive_test.json” file, displaying all three object variations. . . . .	146
B.16. A simple JSON file showcasing the problem that can arise with optional values and no specified domain start/end-tag. . . . .	146
B.17. The resulting pseudo-object maps of the “all_inclusive_test.json” file, displaying all three object variations. . . . .	147
B.18. A collage of a functional configuration of the SimpleTagParser for the single line file “queries.txt”. . . . .	149
B.19. The resulting pseudo-object maps of the “queries.txt” file, displaying one such object. . . . .	150
B.20. Bonus figure of the “paragraphs” array with the delimiting character “ ” being replaced by the newline character. . . . .	150
B.21. A simple YAML file example called “yaml_exempl.yaml”. Source is <a href="#">ibm.com</a> <sup>15</sup> . . . . .	151
B.22. The resulting pseudo-object maps of the “yaml_exempl.yaml” file, displaying the two object variations. . . . .	151
B.23. A collage of a functional configuration of the SimpleTagParser for the YAML file “yaml_exempl.yaml”. . . . .	152
B.24. A simple XML file example called “cd_catalog.xml”. Source is <a href="#">w3schools.com</a> <sup>16</sup> . . . . .	153
B.25. The resulting pseudo-object maps of the “cd_catalog.xml” file, with the first object on display. . . . .	153

## List of Figures

B.26. A functional configuration of the SimpleXMLParser for the regular XML file "cd_catalog.xml". Fields are identical to the XML's pseudo-object tags. . . . .	154
B.27. A collage consisting of the beginning and ending of a SpreadsheetML file example called "spreadsheetML example.xml". Source is etutorials.org <sup>17</sup> . . . . .	155
B.28. The resulting pseudo-object maps of the "spreadsheetML example.xml" file, with the first and last objects on display. . . . .	155
B.29. A functional configuration of the SimpleXMLParser for the SpreadsheetML file "spreadsheetML example.xml". . . . .	156
B.30. Routine configuration. . . . .	158
B.31. Procedure definition of "appendString". . . . .	158
B.32. The corresponding loading procedure. . . . .	159
B.33. The domain class that is used for the load. . . . .	159
B.34. Resulting table of "appendString" and load in the database. . . . .	159
B.35. Procedure definition of "prependString". . . . .	160
B.36. Resulting table of conditional "prependString" and load in the database. . . . .	161
B.37. The altered file that is used for the showcase. . . . .	161
B.38. Procedure definition of "trimField". . . . .	162
B.39. Resulting table additional "trimField" and load in the database. . . . .	162
B.40. A collage of the loading procedure's configuration. . . . .	163
B.41. The routine configuration. . . . .	164
B.42. The extended domain class. . . . .	164
B.43. The procedure configuration for "splitStringField". . . . .	164
B.44. Resulting table of the "splitStringField" transformation and final load. . . . .	165
B.45. The routine configured with update properties . . . . .	166
B.46. Configuration of another loading procedure. . . . .	166
B.47. The procedure configuration for "concatenateFields". . . . .	166
B.48. Result of the "concatenateFields" procedure and a routine that updates already loaded object. . . . .	167
B.49. Domain class used for load. . . . .	167
B.50. The routine configuration. . . . .	168
B.51. Configuration of the "calculateSum" procedure. . . . .	168

B.52. The loading configuration executed after “calculateSum”.	168
B.53. Resulting table of the “calculateSum” procedure and final load.	169
B.54. Configuration of the “calculateMean” procedure.	170
B.55. Adapted loading configuration that is executed after “calculateMean”.	170
B.56. Resulting table of the “calculateMean” procedure and final load.	171
B.57. Configuration of the “arithmeticOperation” procedure.	172
B.58. Adapted loading configuration that is executed after “arithmeticOperation”.	172
B.59. Resulting table of the “+” “arithmeticOperation” procedure and final load.	173
B.60. Snippet of the resulting table using the “-” “arithmeticOpera- tion” procedure and final load.	173
B.61. Snippet of the resulting table using the “/” “arithmeticOpera- tion” procedure and final load.	174
B.62. Snippet of the resulting table using the “*” “arithmeticOpera- tion” procedure and final load.	174
B.63. Snippet of the resulting table using the “%” “arithmeticOpera- tion” procedure and final load.	174
B.64. Configuration of the “unaryArithmeticOperation”	175
B.65. Resulting table of the “unaryArithmeticOperation” procedure and final load.	175
B.66. Snippet of original table for comparison.	176
B.67. First procedure configuration for “regexReplace”.	177
B.68. Third procedure configuration for “regexReplace”.	177

## List of Figures

B.69. Second procedure configuration for "regexReplace". . . . .	177
B.70. Fourth procedure configuration for "regexReplace". . . . .	177
B.71. The domain class that is used for the load. . . . .	178
B.72. Configuration of the loading procedure. . . . .	178
B.73. Resulting table of all "regexReplace" procedures and final load. . . . .	179
B.74. The procedure configuration for "regexReplace". . . . .	180
B.75. Resulting table of yet another "regexReplace" procedure and final load. . . . .	180
B.76. Routine configured with "update properties". . . . .	181
B.77. The configuration of the "setValueFromOptionalValues" procedure. . . . .	182
B.78. Adapted loading configuration that is executed after "setValueFromOptionalValues". . . . .	182
B.79. Resulting table of the "setValueFromOptionalValues" proce- dure and final load. . . . .	182
B.80. New routine configuration for the domain class "RelationTestClass". . . . .	183
B.81. Configuration for the "setTimestamp" procedure. . . . .	183
B.82. Domain class that is introduced with "setTimestamp". . . . .	184
B.83. Loading configuration for "setTimestamp". . . . .	184
B.84. Resulting table of the "setTimestamp" procedure and final load. . . . .	184
B.85. Configuration of the "createRelation" procedure. . . . .	185
B.86. Inner join of the resulting "relation_test_class" table and the one to one relations to the "XML_CD_Catalog_Test" table it references. . . . .	185
B.87. Configuration of the "createRelation" procedure. . . . .	186
B.88. Inner join of the resulting "relation_test_class" table and the one to many relations to the "XML_CD_Catalog_Test" table. . . . .	186
B.89. Configuration of the "crossSetValue" procedure. . . . .	187



## List of Figures

B.90. The extended “relation_test_class” domain class. . . . .	188
B.91. The resulting “relation_test_class” table and the indirectly related “XML_CD_Catalog_Test” table. . . . .	188
B.92. Configuration of the “crossCreateRelation” procedure. . . . .	189
B.93. Inner join of the resulting “relation_test_class” table and the “XML_CD_Catalog_Test” table it references. . . . .	189

## List of Abbreviations

<b>API</b>	Application Programming Interface
<b>BLOB</b>	Binary Large Object
<b>CSS</b>	Cascading Style Sheets
<b>CSV</b>	Comma Separated Values
<b>DB</b>	DataBase
<b>DIM</b>	Dynamic Import Module
<b>DOM</b>	Document Object Model
<b>DSV</b>	Delimiter Separated Values
<b>ETL</b>	Extract Transform Load
<b>GORM</b>	Grails Object Relational Mapping
<b>ID</b>	IDentifier
<b>JPEG</b>	Joint Photographic Experts Group (picture format)
<b>JSON</b>	JavaScript Object Notation
<b>MVC</b>	Model View Controller
<b>OCR</b>	Optical Character Recognition
<b>OMR</b>	Optical Mark Recognition
<b>PDF</b>	Portable Document Format
<b>PNG</b>	Portable Network Graphics
<b>RAM</b>	Random Access Memory
<b>SQL</b>	Structured Query Language
<b>UI</b>	User Interface
<b>XML</b>	Extensible Markup Language
<b>XSD</b>	XML Schema Definition
<b>YAML</b>	YAML Ain't Markup Language



# 1. Introduction

Today, many different processes are being automated, ranging from automotive manufacturing to recommender systems on online shopping portals. Automation boosts efficiency, resulting in reduced costs and sped up processes while also reducing errors, which often occur when bored employees carry out menial tasks. Employees are now free to invest their time into creative processes, perpetuating unique projects and ideas for their respective companies. The Dynamic Import Module (DIM) on which this document is based on, is a tool employed to Extract, Transform and Load (ETL<sup>1</sup>) data from multiple file sources and aggregate such into a SQL database. The DIM's purpose is to support the automation of preexisting basic and repetitive tasks. Tasks that were occupying clerks with tedious data accumulation and number crunching. This automation would free up time for clerks to invest into more meaningful projects, giving their business an edge in certain areas.

The purpose of this paper is to be a documentation of the ideas behind the DIM, its features, how it was ultimately implemented and what the underlying limitations are. Chapter 2 takes a look at the current state of the Grails 2.2 plugins, making a comparison of what is available in this niche and how the DIM is able to fit in. To get a basic understanding on how the DIM is designed, it is recommended to visit the chapter 3.1, which not only elaborates on the file formats the DIM is able to parse, but also how all ETL components mesh together and what their fundamental building blocks are. If on the other hand some features are not fully understood and it is necessary to dive into the implementation itself, then this can be achieved via chapter 3.2. Here, all the notable code is displayed and annotated with explanations on their modus operandi, and info is given

---

<sup>1</sup>[https://www.sas.com/en\\_us/insights/data-management/what-is-etl.html](https://www.sas.com/en_us/insights/data-management/what-is-etl.html) (Last visited: 11.01.2021)

## 1. Introduction

on what limitations apply. The aforementioned implementation chapter 3.2 is further sectioned into three larger sub-chapters. The first sub-chapter 3.2.3 is responsible for the extraction block and explains how each parser approaches their respective file types. The second sub-chapter 3.2.4 refers to the transformation and loading block, in which all transformation and loading methods are listed and annotated. The final sub-chapter 3.2.5 gives a quick overview on how the user interface - responsible for handling the DIM's configuration - is composed and functions. Limitations and ideas for future implementations are highlighted in chapter 4, giving an overview on why using the DIM might be problematic in certain cases.

Last but not least are appendices A and B, with the former providing information on where to get the DIM and how to run it, whereas the latter has examples for each of the DIM's implemented features. These appendices can be ran over quickly and should suffice if the developer does not want to know about the DIM's inner workings, but just wants to get the software running as quickly as possible.

## 2. Related work

Grails has managed to become a popular framework for Java, with many developers employing it to quickly generate MVC web applications. This chapter gives a short summary of which plugins for this particular framework are available and relevant in regard to ETL. Further, a conclusion on how the DIM compares to said plugins is given and if the DIM actually provides any new functionality that is not covered by previous plugins.

Java itself enjoys the position of having a sizeable following and market penetration, giving it the benefit that most if not all feasible ideas were already implemented in some form or another. There is plenty of Java ETL software available, with JasperSoft ETL<sup>1</sup> and the Oracle Data Integrator<sup>2</sup> being just two examples. Still, the DIM has its niche as a Grails specific plugin and is therefore set side by side with other Grails plugins for this chapter. Please note that because the DIM is a Grails 2.2 plugin, only Grails plugins of version 1 and 2 were used for this status check.

As the name suggests, the “Grails Excel Import Plugin”<sup>3</sup> provides various methods to import not only Excel files, but also their counterpart [CSV](#). The “Grails Excel Import Plugin” is also able to handle type errors, check for empty cells and evaluate formulas. CSV is a common file format and another plugin handling the import of such is the “Grails CSV Plugin”<sup>4</sup>. This plugin is able to ignore nested commas inside quotes, acknowledge escaped tokens and multi-line quoted values. The CSV plugin also allows its user to specify the separation character, the escape character and the text

---

All sites were last visited: 11.01.2021

<sup>1</sup><https://community.jaspersoft.com/project/jaspersoft-etl>

<sup>2</sup><https://www.oracle.com/middleware/technologies/data-integrator.html>

<sup>3</sup><https://github.com/gpc/grails-excel-import>

<sup>4</sup><https://github.com/vsachinv/grails-csv>

## 2. Related work

encoding the file is using. Further, it is also possible to define how many lines should be skipped before the actual parsing begins.

Another plugin processing CSV is the “Bulk Data Imports Plugin”<sup>5</sup>, offering customization on how to import CSV. For example, the plugin allows the user to create associations between data, define what columns to be marshalled and set default values.

“The DB Stuff db->xml->db” plugin<sup>6</sup> facilitates importing XML data into databases (tested with MySQL and SQL servers). The plugin is able to create and populate schemata and to export back to XML.

“GORM for Riak”<sup>7</sup> is a JSON parsing plugin, which is able to load the extracted data into a distributed NoSQL database called Riak.

The “Extended Data Binding Plugin”<sup>8</sup> enables further configuration of the Grails databinder to be able to populate custom properties (e.g. calendar) into domain objects and load them then into the database. This is achieved by allowing the developer to create an interface and then populate the domain objects by simply setting the objects properties to the parameters passed to the controller, or as specified by the plugin, by calling “bind()” on the parameters. This process could be argued as a far fetched form of “extraction” of data in form of user generated input, which means this plugin theoretically handles extraction and loading.

Honorable mention of the “Xsd Reverse Engineering Plugin”<sup>9</sup>, which is technically not extracting data but metadata in form of XSD<sup>10</sup> and using the metadata to create GORM classes that in turn can be populated and loaded into databases.

In conclusion, there is no all-in-one ETL plugin available. Some plugins are able to extract and load, whereas others only extract or load. Most such software focuses on one or two file formats and further, no other plugin provides means to transform extracted data like the DIM does. On the other

---

<sup>5</sup><https://mvnrepository.com/artifact/org.grails.plugins/bulk-data-imports>

<sup>6</sup><https://github.com/9ci/grails-db-stuff>

<sup>7</sup><https://github.com/akhil/grails-gorm-riak>

<sup>8</sup><https://github.com/berngp/grails-extended-data-binding>

<sup>9</sup><https://github.com/aaronzirbes/grails-xsd-reverse-engineer>

<sup>10</sup>[https://www.w3schools.com/xml/xml\\_schema.asp](https://www.w3schools.com/xml/xml_schema.asp)

hand, the extraction plugins do a far better job at actually extracting the data and provide the user with more customization in parsing the files than the DIM does. Still, if a developer is looking for a lightweight all-in-one ETL plugin that quickly and easily allows to automate an ETL procedure, then the plugin of this document might be of help.



## 3. Methods

### 3.1. Concepts

This chapter explains the ideas surrounding the DIM and how and the involved technologies and their properties are handled in theory. Further, the design decisions behind the [implementation](#) of the DIM are listed and elaborated on in regard to how or why they were made.

#### 3.1.1. Extraction

As indicated by the title, this section will deal with the technologies and concepts around the DIM's data extraction. More specifically, the basic file formats it is able to parse and how the module takes each of the file format's properties into account. As of now, the DIM can parse following files:

- Fixed-width files
- DSV files (e.g. CSV, TSV)
- XML files (including SpreadsheetML)
- JSON and JSON-like files (e.g. YAML)

## 3.1. Concepts

### 3.1.1.1. Fixed-width files

Fixed-width files are tabular in design with columns and rows, with their defining characteristic being that every column in the file has its own fixed width across all rows in which the corresponding data is contained. The DIM takes this into consideration by requiring the stipulation of a start and end-index for each such column, allowing for a simple program that parses row for row via a single loop. Another inherent property is that these files can contain lines within that do not adhere to the fixed-width rule. Further, these files often contain some sort of meta-data in the beginning that gives information on the following data or the file itself. An example for such lines is demonstrated in the first few lines of figure 3.1. Those mentioned properties can be skipped via one of the [ColumnWidthParsers](#) features.

03/04/2013												Page	1
Period 01 Thru 03													
4:16 pm													
Company 200													
Entry	Per.	Post	Date	GL Account	Description	Srce.	Cflow	Ref.	Post	Debit	Credit	Alloc.	
16524	01		10/17/2012	3930621977	TXNFUES	S1	Yes	RHMWPCEP	Yes		5,007.10	No	
191675	01		01/14/2013	2368183100	OUNHQEX XUFQONY	S1	No		Yes		43,537.00	Yes	
191667	01		01/14/2013	3714468136	GHAKASC QHJXDFM	S1	Yes		Yes	3,172.53		Yes	
191673	01		01/14/2013	2632703881	PAHFSAP LUVIKXZ	S1	No		Yes	983.21		No	
80495	01		11/21/2012	2766389794	XDZANTV	S1	Yes	TGZGMOXG	Yes		903.78	Yes	
80507	01		11/21/2012	4609266335	BWVYEZL	S1	Yes	USUKVMZO	Yes		670.31	No	
80509	01		11/21/2012	1092717420	QJYPKVO	S1	No	DNUNTIAS	Yes		848.50	Yes	
80497	01		11/21/2012	3386366766	SOQLCHU	S1	Yes	BRHUMGJR	Yes		7.31	Yes	
191669	01		01/14/2013	5905893739	FYIWNKA QUAFDKD	S1	Yes		Yes	9,167.93		Yes	
191671	01		01/14/2013	2749355876	CEMJTLP NGFSEIS	S1	Yes		Yes	746.70		Yes	
191674	01		01/14/2013	4530359106	OTAVZGH ZUQFISZ	S1	Yes		No	7,035.74		Yes	
244819	01		02/04/2013	4679391677	EGHLQTI ABE	S1	Yes		No		89,947.13	No	
96062	01		11/30/2012	5996493062	KTSVTADFF EHEHFMX	S1	Yes	UBNQLRCC	Yes	7.10		Yes	
16527	01		10/17/2012	5595769375	ILCVJYC	S1	Yes	HCVZOUY	Yes		321.19	Yes	
191670	01		01/14/2013	1948028853	RFPDCWC UWODNIO	S1	Yes		No	9,293.80		No	
191672	01		01/14/2013	4938823703	CTMDXKP HXOKVFF	S1	Yes		No	175.00		Yes	
191668	01		01/14/2013	4207018603	DBZZULF QGDZQMD	S1	Yes		Yes	206.26		Yes	
ENDING BALANCE PERIOD 01										30,788.27	141,242.32		

Figure 3.1.: Example of a typical fixed-width file parsable by the DIM. Source is [dailydoseofexcel.com](http://dailydoseofexcel.com)<sup>1</sup>.

<sup>1</sup><http://dailydoseofexcel.com/archives/2013/04/12/sample-fixed-width-text-file/> (Last visited: 11.01.2021)

### 3. Methods

### 3.1.1.2. Delimiter separated value files (DSV)

Quite similar to the fixed-width files are delimiter-separated-value files. These DSV files are also tabular in design. A commonly used DSV file format on Windows is the CSV (Comma Separated Values) format. As implied by the name, for a CSV each value is separated by a comma. The first row of such a file usually consists of the columns' names, also separated by commas. This line has to be accounted for when parsing such a file, as it is not actual data but meta-data. The DIM does not consider the meta-data provided by the first line and sadly requires the user to specify their own definition, which can be identical. The CSV format restricts the use of commas appearing in values, as those would then be split into more values, conflicting with the amount of columns specified. Although there is software available that can consider the special case of nested commas, such software usually requires the values containing these commas to be encased by quotes, to avoid accidental splitting. An example of such software would be the "Grails CSV plugin" presented in the [related work](#) chapter. The DIM's implementation of a CSV parser, the [TokenSplitParser](#), is not able to handle any form of nested commas. Figure [3.2](#) represents a common CSV example that was used to test the DIM.

policyID	statecode	county	eq_site_limit	hu_site_limit	fl_site_limit	fr_site_limit	tiv_2011	tiv_2012	eq_sit_deductible	hu_site_deductible	fr_site_deductible	residential	masonry	wood		
119736	FL	CLAY COUNTY	498960	498960	498960	498960	792148.9	0	0	0	0.102261	-81.71177	Residential,Masonry	1		
448094	FL	CLAY COUNTY	1323276.3	1323276.3	1323276.3	1323276.3	1343816.57	0	0	0	0.063936	-81.70766	Residential,Masonry	3		
206893	FL	CLAY COUNTY	190724.4	190724.4	190724.4	190724.4	192476.78	0	0	0	0.089579	-81.70045	Residential,Wood	1		
333743	FL	CLAY COUNTY	0	79520.76	0	0	79520.76	86854.48	0	0	0	0.063236	-81.707703	Residential,Wood	3	
172534	FL	CLAY COUNTY	0	254281.5	0	254281.5	246144.49	0	0	0	0	0.060614	-81.702675	Residential,Wood	1	
785275	FL	CLAY COUNTY	0	515035.62	0	0	515035.62	884419.17	0	0	0	0.063236	-81.707703	Residential,Masonry	3	
995932	FL	CLAY COUNTY	0	19260000	0	0	19260000	20610000	0	0	0	0.102261	-81.713882	Commercial,Reinforced Concrete	1	
223488	FL	CLAY COUNTY	328500	328500	328500	328500	348374.25	0	0	0	0.102217	-81.707146	Residential,Wood	1		
433512	FL	CLAY COUNTY	315000	315000	315000	315000	315000	265821.57	0	0	0	0.18774	-81.704613	Residential,Wood	1	
142071	FL	CLAY COUNTY	705600	705600	705600	705600	101084.56	14112	35280	0	0	0.106268	-81.703571	Residential,Masonry	1	
253816	FL	CLAY COUNTY	831498	831498	831498	831498	3831498.3	831498.3	1117791.48	0	0	0	0.10216	-81.719444	Residential,Masonry	1

Figure 3.2.: Example of a typical CSV file parsable by the DIM. Source is [spatialkey.com](https://spatialkey.com)<sup>2</sup>.

Another DSV file the DIM was originally built for uses a whitespace as a delimiter. Even more peculiar, the mentioned file has a variable number of columns and the data types of the columns can be variable, i.e. they can be

<sup>2</sup><https://support.spatialkey.com/spatialkey-sample-csv-data/> (Last visited: 11.01.2021)

## 3.1. Concepts

floating point values or strings. Figure 3.3 is a good example for such a file and the [TokenSplitParser](#) is the implementation able to parse it.

```
Id-- Date---- desc-- ## 1 to 6 numbers or optionally a comment
1111 20181101 type1 01 943.23 02 403.91 03 1149.20 04 495.38 05 1001.49 06 259.75
1111 20181101 type2 01 502.08 02 440.03 03 694.12 04 272.13 05 439.41 06 42.96
1234 20181101 type1 01 870.34 02 947.29 03 1633.80 04 1222.15
1234 20181101 type2 01 592.12 02 464.46 03 907.61 04 580.85
2222 20181101 type1 01 1331.13 02 777.82 03 633.91
2222 20181101 type2 01 405.68 02 454.11 03 455.96
4588 20181101 type1 01 748.82 02 1312.00
4588 20181101 type2 01 874.31 02 827.15
5006 20181101 type1 01 1771.53 02 852.21 03 2830.33 04 922.68 05 2700.30 06 1111.33
5006 20181101 type2 01 479.05 02 351.01 03 1099.49 04 581.89 05 739.62 06 623.25
6504 20181101 type1 01 0.00 02 1029.81 03 0.00 04 1831.85 -- WOW-- !
6504 20181101 type2 01 0.00 02 293.36 03 0.00 04 651.05 -- WOW-- !
NEW STUFF
Id-- Date---- desc-- # numbers
1432 20181101 type3 1 10.51
1433 20181101 type3 1 83.34
1444 20181101 type3 1 34.75
```

Figure 3.3.: A file using whitespace as a delimiter and has a variable amount of columns with variable data types

### 3.1.1.3. Tag based files

Tag-based files are a little more difficult to parse than the tabular files detailed above. In tag-based files, the data is marked or encapsulated by tags, which simultaneously provide meta-information about their content, even if only the name of the object datum. There are plenty of tag-based file formats with JSON and XML being popular examples. The DIM provides two different implementations for each. The generic tag-based parser, called [SimpleTagParser](#), is able to parse JSON and JSON-like file variants. The other implementation, [SimpleXMLParser](#), is built only for XML and SpreadsheetML files and makes use of the Java XML library. These two parsers are both elaborated in their own separate sections, with each going into more detail on the two parsers, how specifically they differ from each other and how they handle their file formats' properties.

### 3. Methods

**3.1.1.3.1. JSON files** As mentioned before, JSON uses tags to mark their content/data. Start-tags or keys have to be strings, marked by quotes and are followed by a colon. JSON does not have end-tags in the sense XML does, although the implementation of the DIM, the [SimpleTagParser](#), uses commas, quotes and new line feeds as pseudo end-tags. Every object is marked with curly brackets and there is no limit on how many objects are nested within objects. JSON also allows for arrays, which are marked by square brackets and whose values are separated with commas. JSON has no mandatory regulation for line feeds, meaning files might be well-formed and have one tag-value-pair per line, but might also have multiple tags-value-pairs per line, possibly even whole files in a single line. Figure 3.4 shows an example of a typical JSON file.

This paragraph has quoted some content from w3schools<sup>3</sup>.

```
{
  "glossary": {
    "title": "example glossary",
    "GlossDiv": {
      "title": "S",
      "GlossList": {
        "GlossEntry": {
          "ID": "SGML",
          "SortAs": "SGML",
          "GlossTerm": "Standard Generalized Markup Language",
          "Acronym": "SGML",
          "Abbrev": "ISO 8879:1986",
          "GlossDef": {
            "para": "A meta-markup language, used to create markup languages such as DocBook.",
            "GlossSeeAlso": ["GML", "XML"]
          },
          "GlossSee": "markup"
        }
      }
    }
  }
}
```

Figure 3.4.: A simple JSON file parsable by the DIM. Source is json.org<sup>4</sup>.

---

<sup>3</sup>[https://www.w3schools.com/js/js\\_json\\_syntax.asp](https://www.w3schools.com/js/js_json_syntax.asp) (Last visited: 11.01.2021)

<sup>4</sup><https://json.org/example.html> (Last visited: 11.01.2021)

### 3.1. Concepts

```
nodes:
- name: controller
  description: Cloud controller node
  fqdn: controllername.company.com
  password: passw0rd
  identity_file: ~
  nics:
    management_network: eth0
    data_network: eth1
- name: kvm_compute
  description: Cloud KVM compute node
  fqdn: kvmcomputename.company.com
  password: ~
  identity_file: /root/identity.pem
  nics:
    management_network: eth0
    data_network: eth1
```

Figure 3.5.: A simple YAML file with repeating values parsable by the DIM. Source is [ibm.com](https://www.ibm.com/support/knowledgecenter/en/SST55W_4.3.0/liaca/liaca_cli_cloud_yaml_cfg.html)<sup>5</sup>.

```
# Employee records
- martin:
  name: Martin D'vloper
  job: Developer
  skills:
    - python
    - perl
    - pascal
- tabitha:
  name: Tabitha Bitumen
  job: Developer
  skills:
    - lisp
    - fortran
    - erlang
```

Figure 3.6.: A problematic example of a YAML file that can not be parsed properly because of the “skills” list. Source is [ansible.com](https://docs.ansible.com/ansible/latest/reference_appendices/YAMLSyntax.html)<sup>6</sup>.

The YAML file above can not be parsed properly as its list does not have a delimiter to the right of the value, which the DIM’s implementation requires.

---

<sup>5</sup>[https://www.ibm.com/support/knowledgecenter/en/SST55W\\_4.3.0/liaca/liaca\\_cli\\_cloud\\_yaml\\_cfg.html](https://www.ibm.com/support/knowledgecenter/en/SST55W_4.3.0/liaca/liaca_cli_cloud_yaml_cfg.html) (Last visited: 11.01.2021)

<sup>6</sup>[https://docs.ansible.com/ansible/latest/reference\\_appendices/YAMLSyntax.html](https://docs.ansible.com/ansible/latest/reference_appendices/YAMLSyntax.html) (Last visited: 11.01.2021)

### 3. Methods

**3.1.1.3.2. XML files** A similarity JSON and XML share is that they both use tags to annotate content/data. Contrary to JSON though, in XML each start-tag is obligated to have an opposing end-tag. More so, each end-tag has to appear in opposing order to its corresponding start-tag. The tag pairs are case sensitive and have to be properly nested. Each XML file must have a single root element, containing the actual contents of the file. The characters: <, >, &, ' and " are used by XML to define tags, their attributes and values. Thus, those characters can not be used for data/content and have to be replaced with their predefined entities. XML files have the ability to store meta-data in attributes and the values of those attributes can be different even for identically named tags, which is one of the main reasons why the DIM has a separate implementation for XML, the [SimpleXMLParser](#).

Another reason for the separation is the SpreadsheetML format, a combination of Excel and XML. SpreadsheetML files have a tabular format similar to Excel, but all the data is stored in tags conforming to XML standards and are therefore storable in XML. In SpreadsheetML the data is usually nested within the tags "row"->"cell"->"data", with the first appearing "row", similar to CSV/Excel, containing the names for the pseudo-columns. The tabular nature of these files means a slightly different approach to the standard XML parsing procedure is necessary, a requirement that is taken into account by the DIM. As of now, the DIM completely ignores attributes and their meta-data and wholly focuses on the tags and their encapsulated data. Figure 3.7 shows a simple XML file used as a CD catalog, whereas figure 3.8 shows a typical SpreadsheetML, about dinosaurs.

This paragraph has also quoted content from w3schools<sup>7</sup>.

---

<sup>7</sup>[https://www.w3schools.com/xml/xml\\_syntax.asp](https://www.w3schools.com/xml/xml_syntax.asp) (Last visited: 11.01.2021)

### 3.1. Concepts

```
<CD>
  <TITLE>Empire Burlesque</TITLE>
  <ARTIST>Bob Dylan</ARTIST>
  <COUNTRY>USA</COUNTRY>
  <COMPANY>Columbia</COMPANY>
  <PRICE>10.90</PRICE>
  <YEAR>1985</YEAR>
</CD>
<CD>
  <TITLE>Hide your heart</TITLE>
  <ARTIST>Bonnie Tyler</ARTIST>
  <COUNTRY>UK</COUNTRY>
  <COMPANY>CBS Records</COMPANY>
  <PRICE>9.90</PRICE>
  <YEAR>1988</YEAR>
</CD>
```

Figure 3.7.: A simple XML file parsable by the DIM. Source is w3schools.com<sup>8</sup>.

```
<Row ss:Index="3" ss:StyleID="s23">
  <Cell><Data ss:Type="String">ID Number</Data></Cell>
  <Cell><Data ss:Type="String">Critter</Data></Cell>
  <Cell><Data ss:Type="String">Price</Data></Cell>
  <Cell><Data ss:Type="String">Quantity</Data></Cell>
  <Cell><Data ss:Type="String">Total</Data></Cell>
</Row>

<Row>
  <Cell><Data ss:Type="Number">4627</Data><NamedCell ss:Name="ID"/></Cell>
  <Cell><Data ss:Type="String">Diplodocus</Data><NamedCell ss:Name="Critters"/></Cell>
  <Cell ss:StyleID="s22"><Data ss:Type="Number">22.5</Data><NamedCell ss:Name="Price"/></Cell>
  <Cell><Data ss:Type="Number">127</Data><NamedCell ss:Name="Quantity"/></Cell>
  <Cell ss:StyleID="s22" ss:Formula="=RC[-2]*RC[-1]"><Data ss:Type="Number">2857.5</Data></Cell>
</Row>
```

Figure 3.8.: A SpreadsheetML file parsable by the DIM. Source is etutorials.org<sup>9</sup>.

#### 3.1.2. Transform and load

Whereas the previous chapter brushes up on knowledge of already popular file formats and their properties, this chapter is where the actual original ideas of the DIM and the operating principles of the underlying data

<sup>8</sup>[https://www.w3schools.com/xml/xml\\_examples.asp](https://www.w3schools.com/xml/xml_examples.asp) (Last visited: 11.01.2021)

<sup>9</sup><https://etutorials.org/XML/xml+hacks/Chapter+3.+Transforming+XML+Documents/Hack+42+Create+and+Process+SpreadsheetML/> (Last visited: 11.01.2021)



### 3. Methods

pipeline are explained. Details are given on how the data is cached after a parser extracts it, how data is moved through the application and finally how the transformation and loading procedures are designed and applied onto it. The logic of the loading functionality is basically implemented as an extension to the transformation block. Consequently, their corresponding sub-chapters were consolidated into one bigger sub-chapter, further elaborated [later on](#).

#### 3.1.2.1. From extraction to transformation

Ideally, whenever a parser extracts data, said data should be stored in a structure that is easy to access and pass along through the application. For the DIM, the selected data structure combination consists of maps nested in lists. Each map represents a pseudo-object in a file, corresponding to a single line in a [DSV](#) or [fixed-width](#) file. In a [XML](#) or [JSON](#) this would be a specific parent tag, representing a pseudo-object with its properties. The list that contains all pseudo-object maps is basically a representation of its respective parsed file. Figure 3.9 shows an example of a data structure the DIM created at runtime.

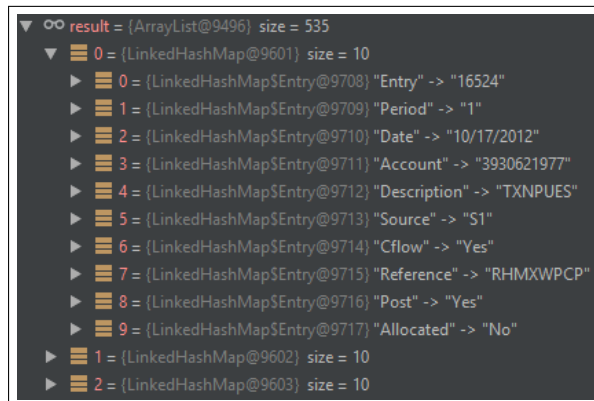


Figure 3.9.: Example data structure with entries from the fixed-width file example. Picture taken from IntelliJ IDEA.

Previously, a list of lists containing the data from all parsed files was kept, mostly for the purpose of debugging. Later on this implementation was

### 3.1. Concepts

dropped during a refactor, as keeping all data files stored in the main memory might risk overloading it, thus crashing the program. Now, whenever a file is successfully parsed, the extracted data is immediately passed on to the transformation block where the transformation is applied. The next file in the queue will be parsed only if the transformation of the previous data was successful. This is done to make sure that the data can be transformed as specified, i.e. the data is not incoherent in regard to its scheduled transformation and the result can be loaded without any faults. An exception arising during the transformation process execution results in the the whole program being stopped, the stacktrace being printed and an error message being returned. Each successfully parsed file is moved to a "parsed files" folder within the original file folder, a special case being the previously mentioned condition in which an exception is raised. Here, the current file and the other following files remain in the original folder, allowing the developer to resume the process after the faulty configuration is fixed. Unfortunately, the developer also has to manually clean up any leftover garbage data in the database, if a load already occurred before the exception was raised for the current file. A crutch for such a case would be the [timestamp transformation method](#) described in the implementation chapter. This method aids the clean-up of the database by annotating data with their creation date, which makes filtering the faulty data easier.

The DIM was designed with automation in mind and, combined with its dynamic nature, is why it is of importance to catch errors and general inconsistencies before extensive operations are applied, possibly even before faulty data is loaded into a database. Such problems are a general concern for ETL software and preventing them is a theme throughout this work.

#### 3.1.2.2. Transformation and loading process

After the extraction process successfully parses a file, its data is be passed on to the transformation block, which in turn executes transformation routines and their transformation procedures. Those routines and procedures are the building blocks of the transformation block, allowing the developer to apply multiple sequenced operations onto the data. The DIM defines each parser as responsible for a file, each transformation routine as responsible for the

### 3. Methods

object which is ultimately loaded into the database, and each transformation procedure as responsible for the transformation methods applied onto the parsed data. Each parser has many transformation routines and each transformation routine has many transformation procedures. All of them are ordered and stored in sorted sets, meaning there is a strict sequence for their execution. This ordering is important because each routine can build upon the results of previous routines, and each procedure can again build on previous procedures' results. Further, the loaded data from a previous file can be retrieved in a later procedure and be used to create relations between the current and the previous objects. Examples for such interactions can be found in the [addendum](#) of this document.

Transformation methods were briefly mentioned before and described as being properties of transformation procedures. Those methods are applied onto the parsed data and make up the core of the transformation process. The DIM provides many such methods with a varying degree of complexity and capability. A further elaboration on what transformation methods are available and how they are implemented can be found in their corresponding [implementation chapter](#).

Figure 3.10 mentions an object list that is initialized for each transformation routine and onto which some transformation methods are applied. These (domain) objects are not to be mistaken with the original maps resulting from a file parse. They can be of any class, specified in their respective transformation routine, and are usually only manipulated by loading methods and certain transformation methods that work directly with these objects. Further, loading methods are of the same design and stored in the same way as the transformation methods, making them indistinguishable at first sight. The object list itself is the foundation of the loading process and each of its objects are ultimately, after all object altering procedures were applied, loaded into the database.

Each successfully parsed and transformed file is moved into a “parsed file” folder, assuming that no exceptions of any kind were thrown. This is done to prevent a re-parsing of the same files over and over again by accident, a valid concern in case of automation. The folder for the finished files is hard coded and cannot be set in any way, other than altering the code itself.

### 3.1. Concepts

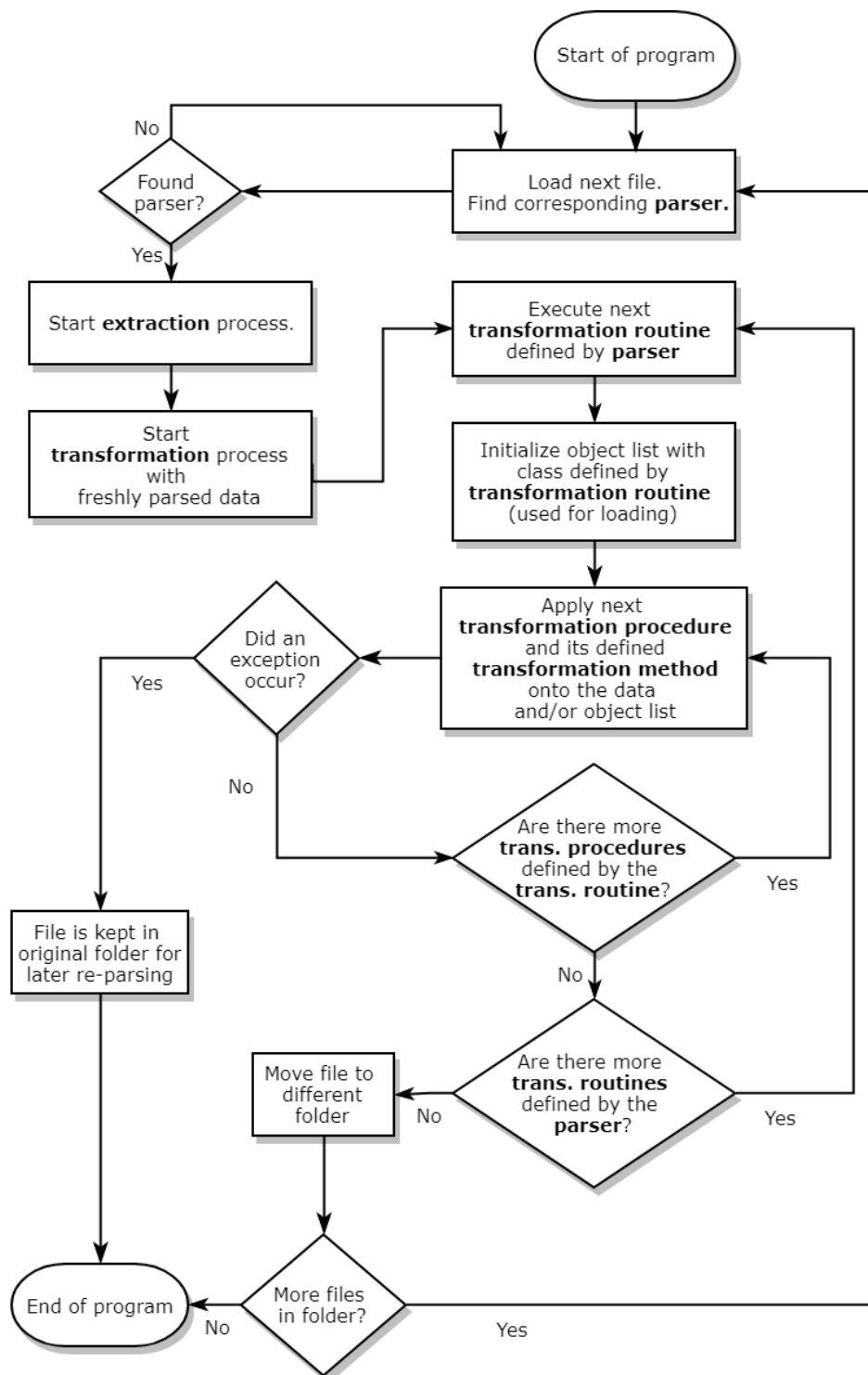


Figure 3.10.: Rough outline of the DIM's workflow.

### 3. Methods

## 3.2. Implementation

Whereas the previous chapter was all about the concepts and why certain design decisions were made, this chapter will elaborate on the actual implementation of those decisions, generally expand on how the code functions and where its limitations lie. Note that the code is clipped in a way to remove unnecessary elements that might distract from the actual program flow. Removed code consists of comments, variable initializations and methods that are self explanatory, e.g. `toString()`.

All methods, properties and classes introduced by the DIM are written in **bold** to help distinguish them from other third party classes and to help differentiate them from related nouns.

This chapter does not contain information related to the user interface, instead, visit the [UI chapter](#) for more information on it.

### 3.2.1. Helpful technologies

Here, a quick rundown of vital technologies is given that not only helped in creating the DIM, but made certain aspects of it possible in the first place.

A key technology frequently used by the DIM is Grails' own Object Relational Mapping or GORM<sup>10</sup> for short. It is built on the Hibernate<sup>11</sup> framework and is used by the DIM to persist/retrieve domain objects into/from the database, create relations between domain objects and also to establish the links between all of the DIM's components. I.e. relations between the parsers and their transformation routines, and transformation routines and their transformation procedures.

Grails allows for duck typing<sup>12</sup>, a form of dynamic typing that lives by the phrase: "If it walks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck.". Whereas static typing verifies at compilation if

---

<sup>10</sup><https://grails.github.io/grails2-doc/2.2.4/guide/single.html#GORM> (Last visited: 11.01.2021)

<sup>11</sup><https://hibernate.org/orm/> (Last visited: 11.01.2021)

<sup>12</sup><https://devopedia.org/duck-typing> (Last visited: 11.01.2021)

## 3.2. Implementation

an object has the correct type, duck typing examines at runtime if an object has the needed methods and properties. This feature is of big help when dealing with the domain object list, whose class is specified at runtime and is passed along in the transformation block. It also comes in handy when working with the polymorphous parsers that are all extended from the base parser class **DynamicParser**. GORM is able to persist such duck typed domain objects, in fact, GORM is even able to persist closures, which are Grails' own implementation of anonymous functions, a feature typically reserved to functional languages. Being able to store such methods is helpful as this means transformation procedures can be persisted and then later retrieved and executed at any time. More information on these interactions can be found in the [transformation procedure section](#).

Finally, JQuery<sup>13</sup> is used for handling the dynamic aspects of the user interface and for generally facilitating the use of JavaScript.

### 3.2.2. Exceptions

Before diving into the code, a description of the exceptions introduced by the DIM should be helpful. As of now there are only five different exceptions, with the first two handling the extraction block and the last three handling the transformation block:

**ParserInconsistentException** is thrown when the program detects conditions that are problematic. E.g. Multiple parsers were found for a single file and no indication which one is the correct one.

**ParserUnfitException** is thrown whenever a parser is not able to parse a file with its current configuration. E.g. a tabular file is parsed and there are not as many required columns contained in the file as specified in the parser's configuration.

**DatabaseSaveException** is raised during the loading procedures, whenever a load fails for whatever reason.

**IncorrectSpecificationException** is thrown whenever some property/field was specified incorrectly. E.g. string for a class name was given, but

---

<sup>13</sup><https://jquery.com> (Last visited: 11.01.2021)

### 3. Methods

class of that name can not be found in the database nor in the application.

**ValidationException** is thrown when the program detects a an inconsistency at runtime. E.g. searching the database for a unique object but getting multiple results.

#### 3.2.3. Extraction

This chapter begins with an explanation on how the files in a folder are handled, how their respective parser is recalled and, in further consequence, how the extracted data is passed on to the transformation block. Additionally, a detailed examination of the abstract base parser class' implementation, from which all available parsers are derived, is given. Lastly, the parsers' common features like their properties and methods are explained, and also how each parser handles the file formats for which they are responsible.

Annotating each parser's implementation and expanding on the ideas behind such provides considerably more information than what an average user might need, instead, the previous [concepts chapter](#) should suffice in giving a rough overview on each parser's handled file formats and its properties.

##### 3.2.3.1. FileParsingService

The **FileParsingService** file contains the code that is responsible for managing files and their respective parsers, and also for ultimately calling the transformation block for each extracted data list. There are only two static methods in the **FileparsingService**, with one of them being **parseAllFilesInDirectory()**, requiring only the directory file as an argument. This is the particular method that can be used by an automated job and is the method that gets the whole process of the DIM starting.

```
14 static def parseAllFilesInDirectory(File dir){
```

## 3.2. Implementation

```
38 dir.eachFile (FileType.FILES) { file ->
39     try {
40         DynamicParser parser = findCorrespondingParser(file)
41         if(parser) {
42             ArrayList<Map<String, Object>> result = parser.parse(file)
43             TransformationService.transformAndLoadData(result, parser)
44             parsedFilesCounter++
45
46             File finished_dir = new File((file.absolutePath - file.name) +
47                 "parsed files")
48             finished_dir.mkdir()
49             if(!file.renameTo(new File(finished_dir, file.getName())))
50                 throw new FileSystemException("Could not move file
51                     '${file.getName()}'! Is another process accessing it? Is a
52                     file with the same name already in the parsed files folder?")
53         }
54         else {
55             println "No parser found for file ${file.name}"
56         }
57     }
58     catch(Exception e) {
59         resultMessage = e.toString()
60         e.printStackTrace()
61         httpCode = 500
62         return [httpCode, resultMessage]
63     }
64 }
65
66 if(httpCode == 200)
67     resultMessage = "${parsedFilesCounter} out of ${fileCount} files were
68         parsed and transformed"
69
70 println resultMessage
71
72 return [httpCode, resultMessage]
73 }
```

A loop iterates through the directory and executes for each file, but excludes other folders contained within the original folder. Note that all files, which are to be parsed, are required to be stored locally on the device the DIM runs on. If files are available, the method **findCorrespondingParser()** then fetches the parser specified for its respective file. Assuming only a single parser matching the file was found, the parser's own overloaded **parse()**



### 3. Methods

function is then called in line 42, with the file as an argument. The result of `parse()` is then passed on to the transformation block in line 43. If a file has no corresponding parser it is skipped but left in the original folder, so that the developer can configure a parser for it in case of oversight. Each successfully parsed and transformed file is moved to the hard coded “parsed files” folder to prevent re-parsing them all over again. If a file could not be moved for whatever reason then an exception is thrown. Two possible reasons for an unmovable file could be, for example, a file of the same name already being in the “parsed files” folder, or that a file is already opened in another program. Whatever the case, the “parsed files” folder will be automatically created on the condition that it does not exist yet and will be placed in the current working directory that was originally passed as an argument. A http code of “200” meaning OK, in combination with a small message indicating the number of processed files, will be returned whenever a parser successfully processes all the files for which it is responsible. Note that if an exception is thrown during the transformation process, the program will ultimately come to a halt here in the extraction block. Further, instead of the code for “OK”, an error code is returned and accompanied by an error message detailing the exception. The error message is additionally printed to the console and the error code in such a case can be either “400” for “bad request”, or “500” for “internal error”. An exception here does not shut down the whole application running on the server, but only halts the DIM’s parsing or transformation process.

The aforementioned `findCorrespondingParser()` method takes a file as an argument and finds its corresponding parser, if one is available for it.

```
71  static DynamicParser findCorrespondingParser(File file) throws
    ParserInconsistentException
72  {
73      DynamicParser parser = null
74
75      DynamicParser.getAll().each {
76          if (it.appliesTo(file))
77              if (parser == null)
78                  parser = it
79              else
80                  throw new ParserInconsistentException("Multiple parsers found for
```

## 3.2. Implementation

```
81         a file. Which one to use? Parsers are: ${it.name} and  
82         ${parser.name}")  
83     }  
84     return parser  
}
```

It does this by iterating through all retrieved (Dynamic-)parsers, with each having the method **appliesTo()** implemented by their abstract base class **DynamicParser**, exhibited in the following section. If a single applicable parser is found, it is stored in a local variable and returned later. Contrary, if multiple parsers are found, an exception will be thrown. The reasoning for the exception is that the program can, in its current state, not determine which parser is the desired one. Further, allowing the DIM to parse the same files with multiple different parsers would most likely generate faults and inconsistencies, as doubling identical data would probably disturb many of the following transformation processes, which are built upon the requirement that such data is unique.

### 3.2.3.2. The base parser class **DynamicParser**

The **DynamicParser** is the abstract base class from which each parser's implementation is derived from. As such, each parser has a few common properties and methods, which will be listed here.

```
6 abstract class DynamicParser {  
7  
8     String name  
9     String description  
10    String selectorName  
11    AllowedFiletype selectorFileType  
12  
13    SortedSet<TransformationRoutine> routines = new  
14        TreeSet<TransformationRoutine>()  
15    SortedSet<DynamicParserEntry> entries = new TreeSet<DynamicParserEntry>()  
16  
17    static hasMany = [entries: DynamicParserEntry, routines:  
18        TransformationRoutine]
```

### 3. Methods

The string properties **name** and **description** are both used to differentiate between parsers. Whereas **name** is used later on in the UI, **description** is not used by the DIM itself and can only be stumbled upon in the database or during debugging, thus making this property more of a luxury than necessity.

The properties **selectorName** and **selectorFileType** on the other hand are used in the previously mentioned **appliesTo()** method, elaborated a bit later, to retrieve the matching parsers for their corresponding files and are therefore required to be set by the developer. The property **selectorFileType** is a simple enumeration of type “AllowedFileType”, containing the following allowed file types as of now:

- .txt
- .csv
- .prn (fixed-width file)
- .json
- .xml
- .yaml

The file types are rather unimportant, as each file can be potentially renamed to a text file. Nevertheless, it was implemented as it can be of use for the identification of a file’s respective parser.

Foreshadowed by the [concepts chapter](#) are the parsers’ transformation routines, with which they have a one to many relationship. Those routines are being kept in a sorted set as they have to apply their transformation in a strict predetermined order, allowing them to build and extend upon previous transformations.

Another sorted set called **entries** is used by the parsing process for identifying and extracting values within files. This set consists of **DynamicParserEntries**, which are elaborated in the following [DynamicParserEntry](#) subsection.

The **DynamicParser** concludes with a static **hasMany** property that is used by GORM to define one-to-many, or many-to-many relations. The latter can be done by using **hasMany** with both related domain classes, but keep in mind that the DIM does not directly support such a relation.

## 3.2. Implementation

```
18 static constraints = {
19     entries( validator: { entries ->
20         entries.every() { entry -> entries.field.findAll{ entry.field == it
21             }.size() == 1 }
22     })
23     selectorName( validator: { selectorName, obj ->
24         boolean isNoFileCollision = true
25         ((List<DynamicParser>)getAll()).each{
26             if(obj.id != it.id && (it.selectorName.contains(selectorName) ||
27                 selectorName.contains(it.selectorName))
28                 && obj.selectorFileType.toString() ==
29                 it.selectorFileType.toString())
30                 isNoFileCollision = false
31         }
32         return isNoFileCollision
33     })
34 }
```

Last but not least are the two constraints that are verified whenever a save occurs. The first validator specified for the entries makes sure that each **field** is unique for all **entries** of this parser, as these **fields** are used as map keys. The second validator, that is applied for the **selectorName**, ascertains that each file has just a single parser and not more. This particular constraint is debated in the [limitations chapter](#), as it prohibits some interesting opportunities.

```
34 abstract ArrayList<Map<String, Object>> parse(File file) throws
    ParserUnfitException, ParserInconsistentException
```

As implied by the nature of a parser, the class **DynamicParser** contains the abstract method **parse()** that has to be implemented by each derived parser and is the method responsible for processing a file.

```
36 boolean appliesTo(File file){
37     return file.name.toLowerCase().contains(selectorFileType.toString()) &&
38         file.name.contains(selectorName)
39 }
```

Then there is the already revealed method **appliesTo()**, a one liner of a method that affirms if a file is of a certain file type. Such is the case if

### 3. Methods

the name contains a sub-string from the **selectorFileType** enumeration and also if it contains the **selectorName** sub-string. E.g. a file's name is "CD\_catalog.xml" and **appliesTo()** examines if the name contains the **selectorName** "catalog" and the **selectorFileType** ".xml". Multiple parsers will be fetched for a particular file, if the **selectorName** and **selectorFileType** set by the developer are not unique together, which can not actually happen due to the constraints and assertions in place.

```
40  static boolean
    checkIfStrictParsingNeeded(SortedSet<TransformationRoutine> routines,
    String line){
41  boolean is_needed = true
42
43  routines.each{ routine ->
44    routine.procedures.each {
45      it.notable_objects.each {
46        if (line.contains(it.value) && procedure.is_repetitive) {
47          is_needed = false
48        }
49      }
50    }
51  }
52  return is_needed
53 }
```

Finally, the abstract class finishes with the static function **checkIfStrictParsingNeeded()**. This helper function is used by most parsers to verify during the parsing process, if a value that is not of a specified **dataType** actually needs an exception raised, or if the value is excluded later on in the transformation process and thus can be just ignored. How this exclusion is conducted during the transformation process, is elaborated later on in the [transformation chapter](#).

description	name	select	selector_name	class
parser for some products DSV	DSV products	TXT	Products	extraction.TokenSplitParser
Parser for a simple CD catalog XML	CD catalog parser	XML	cd_catalog	extraction.SimpleXMLParser
FixedWidthExample	Fixed-width example	TXT	FixedWidthExample	extraction.ColumnWidthParser

Figure 3.11.: Example of parsers being saved in the same DB table 'dynamic\_parser', because of the base class. Database showcased is MySQL.

### DynamicParserEntry

Each derivation of the **DynamicParser** has multiple **DynamicParserEntries**, which are used for identifying values in pseudo-objects within the files. Each **DynamicParserEntry** contains meta information on how to extract their respective values, thus making these **entries** a core component of the parsing process. Whereas all parsers try to handle each file on a pseudo-object to pseudo-object basis, the **entries** attempt to iterate over each of the pseudo-object's values. This operating principle is why the **DynamicParserEntry** also implements the "Comparable" interface, to ensure a correct sequential iteration over these values.

```
5 class DynamicParserEntry implements Comparable{  
6  
7     String field  
8     boolean optional = false  
9     EntryDatatype dataType
```

The class begins with a property called **field**, a string that is used as a map-key for a map mentioned in the previous [concepts chapter](#). Thus, this string can not be left null, can not be empty and has to be unique for all other **entries** of their respective parser. The constraint validation for the latter can be found in a previous code snippet of the parent class **DynamicParser**, shown in [lines 19 to 21](#).

Continuing on with the **DynamicParserEntry**'s **optional** property, a property used by most parsers except for the XML parser. Whenever this property is set to true, the extracted value must occur where it is specified to be, and has to be of the specified data type. Contrary, if set to false, both those requirements are waived and won't raise an exception.

All of the DIM's parsers are designed to parse one type of pseudo-object and their related values. The values have to be primitive and can not be further nested within pseudo-objects, as it is possible with JSON and XML files.

### 3. Methods

The property **dataType** is an enumeration that was created for the purpose of restricting which primitive data types a pseudo-object's values can have, and consists of following items as of now:

- Boolean
- Float
- Integer
- Long
- String

Whenever a value is extracted, it is parsed to one of these primitive data types, or in case of a string, left as is. Note that this data type conversion is actually not mandatory, as later during the loading process, each primitive value can be automatically cast and converted into the primitive data type of the respective domain object's property. It is still recommended to specify the **dataType** property though, as exceptions raised here during the parsing can bring attention to unexpected file properties or possible faults in a parser's configuration. The original plan was to also have a separate type for the date, but because of the various different formats and complexity it brings, the idea was scrapped and instead such a field should be declared as a plain string.

```
29 Boolean checkType(String type) throws ParserUnfitException{
30     if(dataType == EntryDatatype.INTEGER){
31         try{
32             Integer.parseInt(type)
33             return true
34         }
35         catch(NumberFormatException e){
36             if(optional)
37                 return false
38
39             throw new ParserUnfitException("Expected entry of type 'integer' in
40                 file", e.cause)
41         }
42     }
43 }
```

The method **checkType()** does just that, confirm if a value is parsable to the data type specified by the **dataType** property. The code snippet from lines 30 to 41 only shows one such if-else block, but the logic does repeat

## 3.2. Implementation

for all five primitive data types. String is the last case and does nothing if applicable, making it the most lenient of the types. This method also takes the **optional** property into account, in case a value does not appear. E.g. instead of the value '1234.5', only an empty string '' appears, with latter not being parsable to float. By setting **optional** to true, an exception for such a case is prevented. On the other hand, the **optional** property can be a liability if garbage data is parsed and it is required to be of a certain data type, as **optional** defuses the strict data type check.

```
85 def parseField(String value){  
86     if(dataType == EntryDatatype.INTEGER){  
87         return Integer.parseInt(value)
```

The last noteworthy method is **parseField()**, a method that applies the appropriate **parseTo()** method to a value and returns one of the possible primitives, or just the original string. The previous code snippet also only shows the first block and repeats again for all types. Note that this method does not take many different possible numeral representations into account, but follows the same rules each primitive data types “parseTo” method follows. E.g. a “123,456,789” string is not parsable to integer because of the commas. Another problematic example is the european localization that interchanges the use of commas and periods, creating another numeral representation which might not be parsable depending on the machine the DIM runs on. All this has to be considered by the developer themselves, as the DIM does not take such variations into account. A workaround can be implemented with a [certain transformation method](#), in combination with the fact that the transformation block tries to cast each value to the domain object property’s data type in which the value will be saved. See the [addendum](#) for an example on such a workaround.

### 3.2.3.3. ColumnWidthParser

The **ColumnWidthParser** is the most straightforward of all the parsers and is responsible for [fixed-width files](#).

```
5 class ColumnWidthParser extends DynamicParser{
```



### 3. Methods

```
6
7  static hasMany = [linesToIgnore: String]
```

The **ColumnWidthParser** expands the base parser by only a single “has-Many” property called **linesToIgnore**, a list of strings that are used for identifying lines that have to be skipped during the parsing process. As already touched on in the concepts chapter, fixed-width files can have lines that do not contain actual data, but meta-data about the files themselves. These lines have to be detected and skipped by the parser, which is done via exactly this property.

```
13  @Override
14  ArrayList<Map<String, Object>> parse(File file) throws
    ParserUnfitException, ParserInconsistentException {
15  ArrayList<Map<String, Object>> allObjects = new ArrayList<Map<String,
    String>>()
16
17  file.eachLine{ line ->
18  boolean skip = false
19  for(String forbiddenLine in linesToIgnore){
20  if(line.contains(forbiddenLine) || line.trim().isEmpty()){
21  skip = true
22  break
23  }
24  }
```

This parser’s implementation of the **parse()** function begins by iterating over each line of the tabular file and determining if the line contains any of all the defined **linesToIgnore** sub-strings, or if the line is empty, bar leading and trailing whitespaces. The line is skipped from further processing, if any of these conditions apply. Do note that the program reads the file line for line, allowing the parse of huge files. Another peculiarity to watch out for is that the “contains()” method is case sensitive, a feature that can be of help in certain circumstances. E.g. the fixed-width file, illustrated in figure 3.1 in the concepts chapter, has description or reference columns, which sometimes contain values with a “PM” sub-string in uppercase within them. This would have been a potential problem if “contains()” was not case sensitive and the user chose to exclude the timestamp by identifying it with “pm”. The given example is not realistic though, as using “pm” to

## 3.2. Implementation

exclude the timestamp line is prone to skipping other possibly relevant lines.

```
26     if(!skip) {
27         Map<String, Object> objectMap = [:]
28         boolean finishedLine = false
29
30         entries.eachWithIndex { entry_it, entry_index ->
31             String value
32
33             if(finishedLine)
34                 return
35
36             if(entry_it.columnEnd - 1 >= line.length()) {
37
38                 boolean isLastNonOptional = true
39                 entry_index = entry_index + 1 > entries.size() ? entry_index + 1
40                     : entry_index
41                 entries.toList().subList(entry_index, entries.size()).each {
42                     check_entry_it ->
43                     if (!check_entry_it.optional)
44                         isLastNonOptional = false
45                 }
46
47                 if (!isLastNonOptional)
48                     throw new ParserUnfitException("Cannot parse file, because a
49                         line ends before the end should be reached! I.e. when an
50                         entry is already at the end of a line but there are
51                         following non-optional entries specified for the parser.")
52             }
53             else {
54                 value = line.substring(entry_it.columnStart - 1,
55                     line.length()).trim()
56             }
57
58             finishedLine = true
59         }
60         else
61             value = line.substring(entry_it.columnStart - 1,
62                 entry_it.columnEnd - 1).trim()
```

Assuming a valid line was detected, the method then iterates over all **ColumnWidthEntries** and clips each value from in between its defined **start-** and **end-index**. Whenever a line ends before the **entry's end-index** is reached, the program will have to further verify if there are any non-optional entries

### 3. Methods

following the current **entry** and, if such a condition emerges, throw an exception. Contrary, if there are no following **entries**, or all the following **entries** are **optional**, the value will be clipped from the defined **start-index** until the end of the line, and any following **optional entries** are dismissed.

```
56         boolean isParseable = false
57         try {
58             isParseable = entry_it.checkType(value)
59         }
60         catch (ParserUnfitException e) {
61             if(checkIfStrictParsingNeeded(this.routines, value))
62                 throw new ParserUnfitException(e.getMessage() + " (field
63                     called: '" + entry_it.field + "', value was: '" +
64                         entry.toString() + "' )", e.cause)
65         }
66
67         if(isParseable)
68             objectMap.put(entry_it.field, entry_it.parseField(entry))
69     }
70
71     }
72     return allObjects
73 }
```

At any time a value is successfully extracted, the **parse()** method will validate that such is of its specified **dataType**. If this is not the case and the **entry** is non-**optional**, an exception is thrown within the **checkType()** method and immediately caught in the calling method **parse()**. In such an event, the value which raised an exception is examined by the **checkIfStrictParsingNeeded()** method and, if the value is not excluded by the transformation block, the exception is re-thrown with an extended error message. On the other hand if the value is parsable, or not parsable and **optional**, it is put into the data map with the key being the **entry**'s specified **field**. Note that it is recommended to have the class's properties to be assigned with default values - assuming there are such **optional entries** for a file.

### ColumnWidthEntry

Additionally to being a derivation of the basic **DynamicParserEntry**, the **ColumnWidthEntry** adds two more properties. The **columnStart** is used to mark a column's beginning in a row, whereas the **columnEnd** marks an ending. Both of those properties can not be null and have to be greater or equal to one, as is verified by simple constraints not shown in the following listing.

```
4 class ColumnWidthEntry extends DynamicParserEntry{
5
6     Integer columnStart
7     Integer columnEnd
```

One property defining the width of each column could have been enough, but because it should be easier for the developer to just open a text editor and read off the two column indices for each column. Further, two separate indices for each column are unmistakable in regard to a columns positioning, which resulted in the decision for two separate properties.

#### 3.2.3.4. TokenSplitParser

The **TokenSplitParser** is responsible for [DSV files](#) and all of its variants like the CSV and TSV.

```
5 class TokenSplitParser extends DynamicParser{
6
7     String token
8
9     static hasMany = [linesToIgnore: String]
```

There are two properties extended to the base class by the **TokenSplitParser**. The first property being the **token** string, used for splitting each line into separate values. The other property is the static hasMany **linesToIgnore** sub-string list, that is used to detect and skip unneeded lines, similar to how the **ColumnWidthParser** does it.

### 3. Methods

```
16  @Override
17  ArrayList<Map<String, Object>> parse(File file) throws
    ParserUnfitException, ParserInconsistentException {
18      ArrayList<Map<String, Object>> allObjects = new ArrayList<Map<String,
        String>>()
19
20      file.eachLine{ line ->
21          boolean skip = false
22          for(String forbiddenLine in linesToIgnore){
23              if(line.contains(forbiddenLine) || line.trim().isEmpty()){
24                  skip = true
25                  break
26              }
27          }
28      }
```

The **TokenSplitParser** also begins its **parse()** method by reading the file line for line and determining if a line is empty, or if it contains one of the **linesToIgnore**, in which case it is skipped. Note that because the program iterates over each line without loading the whole file, it is possible to parse files of larger size. As was also mentioned in the **ColumnWidthParser** section, the **contains()** method is case-sensitive, meaning the same ramifications apply here as well.

```
30      def splitLine = line.split("\\s*" + token + "\\s*")
31
32      Map<String, Object> objectMap = [:]
33      Map<Integer, ArrayList<Object>> objectMapHelperMap = [:]
34
35      entries.each { entry_it ->
36          if ((splitLine.size() - 1) >= entry_it.splitIndex) {
37              def value = splitLine[entry_it.splitIndex]
38
39              boolean isParseable
40              try {
41                  isParseable = (entry_it.checkType(value))
42              }
43              catch (ParserUnfitException e) {
44                  if(checkIfStrictParsingNeeded(this.routines, value))
45                      throw e
46              }
47          }
48      }
```

The **parse()** method proceeds by splitting the line via a regular expression,

## 3.2. Implementation

as is shown in line 30, whenever a valid line of data appears. All the regex does is match the **token** plus any leading and trailing whitespaces. E.g. “1 , 2 , 3” would be an acceptable line. This regex does work in combination with whitespace and tabs - i.e. TSV files can be parsed without a hassle. The whitespace token can be defined as “ ” and a tab token has to be defined as “\t” in the [UI](#).

After splitting each line with the **token**, the program then iterates over all **entries**, verifying that each **splitIndex** is still smaller than the number of values a split line results in. If that is the case, then the next step would be to parse the corresponding value, which is accessed via a **entry**’s **splitIndex** in the split line, to its specified **dataType**. An exception will be thrown in the **checkType()** method, if a value is not parsable and non-optional. The exception is then immediately caught and another verification step is done via the **checkIfStrictParsingNeeded()** method, which affirms that the **entry** is actually needed by the transformation process. In the event that this is true, the exception is re-thrown with additional information.

```
48         if (isParseable) {
49             if (objectMapHelperMap.get(entry_it.splitIndex) == null) {
50                 objectMapHelperMap.put(entry_it.splitIndex, [entry_it.field,
51                     entry_it.dataType, entry_it.parseField(value)])
52             } else if (objectMapHelperMap.get(entry_it.splitIndex)[1] ==
53                 EntryDatatype.STRING && entry_it.dataType !=
54                 EntryDatatype.STRING) {
55                 objectMapHelperMap.put(entry_it.splitIndex, [entry_it.field,
56                     entry_it.dataType, entry_it.parseField(value)])
57             } else if (entry_it.dataType != EntryDatatype.STRING &&
58                 objectMapHelperMap.get(entry_it.splitIndex)[1] !=
59                 EntryDatatype.STRING) {
60                 throw new ParserInconsistentException("Parser is not
61                     deterministic! Multiple options to parse a value")
62             } else if (entry_it.dataType ==
63                 objectMapHelperMap.get(entry_it.splitIndex)[1]) {
64                 throw new ParserInconsistentException("Parser is not
65                     deterministic! Multiple options to parse a value")
66             }
67         }
68     }
69 }
```

### 3. Methods

Any value that is parsable to its specified **dataType** is stored within a helper map. The **TokenSplitParser** has a more sophisticated approach to extracting and adding values to the data map, as it is possible for two different **entries** to occupy same **splitIndex**. For this to work, one **entry** is required to be a number type (e.g. float, int), whereas the other must be a string. The **TokenSplitParser** considers both for the data map, but gives priority to the number if it is valid - i.e. the number value is not empty and parsable. An exception will be thrown if multiple valid number values are detected. Figure 3.12 showcases lines which apply for the above explained data type priority selection.

```
61     objectMapHelperMap.each{ k, v ->
62         objectMap.put((String)v[0], (Object)v[2])
63     }
64
65     allObjects.add(objectMap)
```

Whatever the case, the value with higher priority remains in the helper map and is then passed to the actual data map. Whenever a line is parsed successfully, its corresponding data map is added to the data-map list that is ultimately returned by **parse()**.

#### TokenSplitEntry

The **TokenSplitParsers** own **DynamicParserEntry**, the **TokenSplitEntry**, only adds a single previously mentioned property called **splitIndex**, an integer.

```
4 class TokenSplitEntry extends DynamicParserEntry{
5
6     Integer splitIndex
```

This property is used to access the desired value in the list of values, where the latter is the result of splitting a line with the parser specified **token**.

## 3.2. Implementation

```
5006 20181101 type1 01 1771.53 02 852.21 03 2830.33 04 922.68 05 2700.30 06 1111.33
5006 20181101 type2 01 479.05 02 351.01 03 1099.49 04 581.89 05 739.62 06 623.25
6504 20181101 type1 01 0.00 02 1029.81 03 0.00 04 1831.85 -- WOW-- !
6504 20181101 type2 01 0.00 02 293.36 03 0.00 04 651.05 -- WOW-- !
```

Figure 3.12.: Showcase of a file that can make use of the data type priority and optional setting.

### 3.2.3.5. SimpleTagParser

The **SimpleTagParser** is a lazy implementation of parser that deals with JSON and JSON-like files and parses such files line-for-line. This approach allows for a basic implementation that is not only able to parse simple JSON files, but also a multitude of simple JSON-like file formats. XML files can in theory also be parsed by this parser, but the DIM provides its own implementation of a XML parser and it is therefore not recommended to use this particular parser for XML files. A consequence of the fundamental generalization built into this parser is, that it is not able to parse more complex JSON or JSON-like files and attempting such might lead to errors and undesired results. E.g. It is possible to parse a simple YAML file, but it is not possible to parse a YAML that has arrays nested within objects. The **SimpleTagParser** has multiple properties with each dealing with certain aspects of different files. Some of those properties are not required to be set, but nonetheless help in properly parsing files, if they are set.

```
7 class SimpleTagParser extends DynamicParser{
8
9   String domainStartTag = null
10  String domainEndTag = null
11  int nestingLevel = -1
12
13  static constraints = {
14    name(blank: false)
15    domainStartTag(blank: false, nullable: true)
16    domainEndTag(blank: false, nullable: true, validator: { val, obj ->
17      return (val == null && obj.domainStartTag == null) || (val != null &&
18        obj.domainStartTag != null)
19    })
20  }
```



### 3. Methods

The **domainStartTag** and **domainEndTag** are two such properties that do not have to be set but are recommended to if possible. Those two tags are used by the parser to detect when a pseudo-object begins and ends. In the latter case, the parser wraps up the current object-map by saving it in the object-map-list and initializing a new object-map. Both of those tags must be specified together, or not at all.

The last property introduced by the **SimpleTagParser** itself is the **nestingLevel**, which is another property that does not have to be set but is recommended. Both of the domain tags have to be set accordingly, if a **nestingLevel** is specified, as the **nestingLevel** is dependent on such and can not function without them. The **nestingLevel** property can be used to filter unwanted values from parent pseudo-objects within the files, but is usually used to tell the parser how deeply nested the wanted pseudo-object is - i.e. how many **domainStartTags** will preface the first valid value. Each value that appears beneath the specified **nestingLevel** is not taken into consideration. Those three properties are used in combination to correctly identify pseudo-objects in JSON and JSON-like files, in case that some properties might not appear or are reordered for each pseudo-object. Examples for such can be found in the addendum. The program will try to determine the value for the **nestingLevel** on its own, in case that only the domain tags are set, but do keep in mind that this feature can lead to undesirable results.

An example configuration of those properties for figure 3.4 would be:

- **domainStartTag** = "{"
- **domainEndTag** = "}"
- **nestingLevel** = 1

This setting would ensure that all values within all nested pseudo-objects are properly extracted, with 'title:' being the first key containing a valid value.

```
21 @Override
22 ArrayList<Map<String, Object>> parse(File file) throws
    ParserUnfitException, ParserInconsistentException {
```

The **parse()** method of the **SimpleTagParser** is the arguably most complex implementation of all the parsers. Tabular file formats, which are dealt

### 3.2. Implementation

with by the **ColumnWidthParser** and the **TokenSplitParser**, are usually straightforward and limited in their design, allowing for simple parsing programs. Contrary, tag-based parsers have to take many more possibilities into consideration as their inherent nature allows for more liberty in how each pseudo-object is defined. These file formats have no restriction how many elements each pseudo-object can have, which names those elements can have, how many lists each pseudo-object can contain or how many pseudo-objects can be nested within each other. The **SimpleXMLParser** is using a built-in java library that simplifies the parser's implementation, but because the **SimpleTagParser** tries to unite many JSON-like formats under its umbrella, the need to reinvent the wheel prevails again.

```
44     for(int j = i + 1; j < entries.size(); j++)
45         if(entries[j].startTag.contains(entries[i].startTag))
46             throw new ParserInconsistentException("Simpler start-tag appeared
                in entry-list that will match tags which are matched by a
                later defined and more complex start-tag!")
```

The **parse()** method starts off by verifying that all simpler **entries** occur later in the **entry**-list. The reasoning here is that a tag can be matched by multiple **entries** and therefore the more complex **entries** must have priority in such, to ensure a correct parsing process.

```
49     if(file.readLines().size() == 1 && selectorFileType ==
        AllowedFiletype.JSON)
50         input = new JSONObject(file.readLines()[0]).toString(2)
51     else if(file.readLines().size() == 1 && selectorFileType !=
        AllowedFiletype.JSON) {
52         if (entries.endTag.contains(null) && entries.size() > 1)
53             throw new ParserInconsistentException("We have a file which only
                has a single line and the parser is defined with more than one
                entry, " +
54                 "but there are entries with no endTag!")
55
56         singleLineFile = true
57     }
```

Another few checks are carried out by the program, with the first one affirming if the file only contains a single line and is a JSON. If so, the JSON file is then “prettified” into a multi-line file with a single tag per

### 3. Methods

line, a format preferred by the parser. Note that this prettifying can reorder tags/keys and their values within their respective pseudo-objects, which might result in different **endTags** being required. E.g. JSON values at the end of pseudo-objects are without commas.

If a file has its whole content in a single line, is not a JSON, and the respective parser has **entries** without an **endTag** specified, then parsing such a file is not possible and an exception is thrown. Contrary, if every **entry** does have an **endTag**, then the parser is capable of extracting values from such a file and a boolean reflecting this circumstance is set.

```
59     entries.each{
60         if(it.endTag == null && it.arraySplitTag == null)
61             patterns.put((DynamicParserEntry)it, Pattern.compile("\\Q" +
62                 it.startTag + "\\E(.*)"))
63         else if(it.arraySplitTag != null){
64             patternArray.put((DynamicParserEntry)it, Pattern.compile("\\Q" +
65                 it.startTag + "\\E(.*)?\\Q" + it.endTag + "\\E"))
66         }
67         else{
68             patterns.put((DynamicParserEntry)it, Pattern.compile("\\Q" +
69                 it.startTag + "\\E(.*)?\\Q" + it.endTag + "\\E"))
70             patternsNoEndTag.put((DynamicParserEntry)it, Pattern.compile("\\Q" +
71                 it.startTag + "\\E(.*)"))
72         }
73     }
```

Another preparation, that is done before the actual parsing process, is the compilation of multiple regex patterns, which are used for matching and extracting values from their key-value pair combinations. It is possible for certain pairs to not have an end-tag, for values to span over multiple lines - assumed by the parser to be strings - or to be arrays, latter which can also stretch over multiple lines. Whatever the case, there is a pattern pre-compiled and available for matching just that.

The “\\Q” and “\\E” markers tell the regex engine to take the provided text as a literal string and to not interpret any possible regex commands into it. E.g. JSON’s curly brackets, which are used as markers for whenever a pseudo-object begins or ends, are also employed as regex commands. The pattern, consisting of the **startTag** appended with “(.\*)”, and optionally

## 3.2. Implementation

the **endTag**, signifies that this is a group of an unspecified number of variable characters enclosed by the specified tags, which are to be extracted and returned. If the pattern is extended with the **endTag**, then a question mark is inserted into the previous identifier, resulting in the following regex `"(. *?)"`. The question mark conveys the regex engine to be non-greedy, i.e. telling the engine to stop at the first and not the last end-tag, as it would be the case without a question mark. The non-greedy indicator should only be relevant for single-line files, but keep in mind that this means that values that do contain their end-tags within them, will not be identified correctly and extracted incompletely. Whatever the case, each pattern is saved in a map with their keys being the respective **SimpleTagEntry**.

```
71 input.eachLine { line, line_index ->
```

As previously mentioned and reflected in the code snippet above, the **SimpleTagParser** skims through its files with a non-flexible line-for-line loop, not looking back or ahead in any way. The code from the lines 72 to 169 is elaborated later on in this chapter, as this block is only executed after certain conditions are met during the file parsing process.

```
170     if(!singleLineFile && domainStartTag != null &&
171         line.contains(domainStartTag))
172         nestedCounter++
173     else if(!singleLineFile && domainEndTag != null &&
174         line.contains(domainEndTag)) {
175         if(nestingLevelTemp == nestedCounter && objectMap.size() != 0) {
176             if(objectMap.size() != entries.size()){
177                 entries.each {
178                     if(!it.optional && objectMap.get(it.field) == null)
179                         throw new ParserUnfitException("Required entry '" + it.field
180                             + "' for object does not appear in the file above line:
181                             " + line_index + "!")
182                 }
183             }
184             allObjects.add(objectMap)
185             objectMap = [:]
186
187             if(nestingLevel == -1)
188                 nestingLevelTemp = -1
```

### 3. Methods

```
186     }
187     else if(nestingLevelTemp > nestedCounter && objectMap.size() != 0)
188         throw new ParserInconsistentException("Parser is below the
            defined 'nested level'. This happens if there is a
            'sub-domain' in the beginning of the domain and the user
            specified no 'nesting-level'. Or if the user specified a
            'nesting-level' higher than it actually is.")
189
190     nestedCounter--
191
192     return
193 }
```

Before iterating over all **entries** and finding the applicable pattern, the program first examines if the current line contains any **domainStartTags** or **domainEndTags**. If the start of such a pseudo-object was detected, a counter responsible for designating the current nesting level is incremented by one. The opposite is done when a domain ends, but there are some additional conditions that have to be resolved. Whenever a **domainEndTag** appears, **parse()** has to verify that the values appeared in the predetermined nesting depth and that no non-**optional entries** were left out. Only when all those conditions are met, can the current object-map be saved and a new map be created. In the event that **nestingLevel** was not set by the developer, the corresponding temporary variable will be reset for the next pseudo-object. This is done every time a new object-map is created and allows each pseudo-object to have a different self determined nesting level, assuming it was not specified.

```
195     for(int entry_index = 0; entry_index < this.entries.size();
196         entry_index++){
197         def entry_it = entries.get(entry_index)
198
199         if(entry_it.arraySplitTag != null)
200             m = patternArray.get(entry_it).matcher((String)line)
201         else
202             m = patterns.get(entry_it).matcher((String)line)
203
204         if(entry_it.endTag != null && entry_it.arraySplitTag == null)
205             m_Multi = patternsNoEndTag.get(entry_it).matcher((String)line)
```

Next up for the **parse()** method is to iterate through all **entries** in the order

## 3.2. Implementation

they were saved in. The program fetches the standard pattern or the array pattern, depending on if the current **entry** has an **arraySplitTag** set. There is also a pattern responsible for tags spanning over multiple lines, which is basically the standard pattern with its end-tag being omitted. The multi-line pattern does only work if such a key-value pair has an end-tag specified and if the value is not part of an array, as the latter would imply that array separator tokens might be strewn around.

```
207     if(!singleLineFile){
208         if(entry_it.arraySplitTag == null && m.find()) {
209             String matchedLine = m.group(1).trim()
210
211             if(domainStartTag == null && objectMap.get(entry_it.field) !=
                null){
212                 if(objectMap.size() != entries.size()){
213                     entries.each {
214                         if(!it.optional && objectMap.get(it.field) == null)
215                             throw new ParserUnfitException("Required entry '" +
                                it.field + "' for object does not appear in the file
                                above line: " + line_index + "!")
216                     }
217                 }
218                 allObjects.add(objectMap)
219                 objectMap = [:]
220
221                 if(nestingLevel == -1)
222                     nestingLevelTemp = -1
223             }
224
225             try {
226                 entry_it.checkType(matchedLine)
227             }
228             catch (ParserUnfitException e) {
229                 if(checkIfStrictParsingNeeded(this.routines, matchedLine))
230                     throw new ParserUnfitException(e.message + " parsed by
                                SimpleTagParser: " + this.name, e.cause)
231             }
232
233             objectMap.put(entry_it.field, entry_it.parseField(matchedLine))
234
235             if(nestingLevelTemp == -1)
236                 nestingLevelTemp = nestedCounter
```

### 3. Methods

```
237  
238         break  
239     }
```

Assuming the file does not consist of only a single line, the current **entry** has no **arraySplitTag** defined, and the pattern of the **entry** successfully matches the current line, which can be a pattern either with or without an **endTag**, then the value is extracted and is cached in a local variable. Contrary, if the program detects that an **entry** was already used to extract a value for the current object map and there is no indicator for when a pseudo-object begins or ends, i.e. no domain tags are set, then it will assume that the current object-map is finished. In such an event, the old object-map is saved, a new map is created, and the **nestingLevel** is again reset if it was automatically determined. Before the old object-map is finally discarded, the program has to affirm that no non-**optional entries** are missing from it. This just explained code snippet from line 211 to 223 repeats throughout the **parse()** method each time a new object-map is created and will be omitted from further elaborations and snippets.

Nevertheless, each extracted value is parsed to its specified primitive data type and saved in the current respective object-map. Subsequently, the temporary nesting level counter is set to the current nesting level, if not set - i.e. no domain tags and **nestingLevel** specified by developer - as this must be the level where the values are occurring. This check is also always done when a field is put into the object-map and will be excluded from the following code snippets of this section.

A break is executed, whenever an **entry**'s determined value was successfully extracted and the end of the corresponding if-block was reached. This break is done to prevent further **entries** from being applied onto the current, already processed line.

```
240         else if(entry_it.arraySplitTag == null && (entry_it.endTag !=  
241             null || domainEndTag != null) != null && m_Multi.find()){  
242             if(nestingLevelTemp == -1)  
243                 nestingLevelTemp = nestedCounter  
244  
245             multilineEntry = entry_it
```

## 3.2. Implementation

```
246     multilineConcat = new StringBuilder("")
247     multilineConcat.append(m_Multi.group(1))
248     break
249 }
```

The program ends up in this particular else-if-branch whenever the pattern matching values with start-tags and end-tags does not apply, but there was a **endTag** property specified. This condition implies that a value spanning over multiple lines must have begun and the appropriate measures have to be adopted. A multi-line entry can only be a string, as splitting a numeral into multiple lines would be nonsensical. The if-clause, responsible for validating the data type, is deferred to the block responsible for the ensuing lines of the multi-line entry, as it would be possible that a number has its end-tag replaced by a **domainEndTag**, thus retroactively nullifying the perceived multi-line condition.

Back to the issue at hand, a temporary entry, indicating the multi-line condition, is set and a StringBuilder is created to concatenate the current line with all its following lines. Another break is executed here as the **entry**, fit for extracting the current value, was identified.

```
250     else if(entry_it.arraySplitTag != null){
251         if(line.contains(entry_it.startTag) &&
           line.contains(entry_it.endTag)) {
252             if (m.find()) {
253                 String matchedLine = m.group(1).trim()
254
255                 arrayStringBuilder = new StringBuilder()
256                 matchedLine.split(entry_it.arraySplitTag).each {
257                     try {
258                         entry_it.checkType(it)
259                     }
260                     catch (ParserUnfitException e) {
261                         if(checkIfStrictParsingNeeded(this.routines, it))
262                             throw new ParserUnfitException(e.message + " parsed by
SimpleTagParser: " + this.name, e.cause)
263                     }
264                 }
265
266                 if(arrayStringBuilder.toString().isEmpty())
267                     arrayStringBuilder.append(it)
268                 else
269                     arrayStringBuilder.append("|" + it)
270             }
271         }
272     }
```



### 3. Methods

```
283     }
284     objectMap.put(entry_it.field, arrayStringBuilder.toString())
285     arrayStringBuilder = null
286
287     if(nestingLevelTemp == -1)
288         nestingLevelTemp = nestedCounter
289
290     break
291 }
292 }
293 }
```

The last branch, of the three branches accounting for tag variations, deals with the appearance of arrays. This case further diverges into two possibilities. Either a complete array appeared within a single line, or only the array's start-tag appeared and the remaining elements are distributed over multiple subsequent lines. The code being displayed in the snippet above is handling the former case.

Whenever a complete array is matched, its **startTag** and **endTag** is removed, the array is split via the specified **arraySplitTag**, and the values are cast back to string before concatenating and separating such by a vertical line "|". Originally, each matched array would get its own collection into which each parsed value was put into, but this was later removed and replaced with a simple "StringBuilder", as allowing collections in the DIM's data pipeline was deemed too complex for now. Therefore, the original code responsible for parsing each value to its specified type is still left in place as a data type validation.

The omitted lines from 253 to 268 contain another check, verifying if the current object-map is complete and a new one is needed.

```
294     else if(line.contains(entry_it.startTag)) {
295         arrayStringBuilder = new StringBuilder()
296         arrayEntry = entry_it
297
298         String matchedLine = line
299             .replaceAll(Pattern.quote(entry_it.startTag)+"(.*)", "\\$1")
300             .replaceAll("(.*)" + Pattern.quote(entry_it.arraySplitTag), "\\$1")
301             .trim()
302     }
```

## 3.2. Implementation

```
303     try {
304         if(!matchedLine.isEmpty())
305             entry_it.checkType(matchedLine)
306     }
307     catch (ParserUnfitException e) {
308         if(checkIfStrictParsingNeeded(this.routines, matchedLine))
309             throw new ParserUnfitException(e.message + " parsed by
310                 SimpleTagParser: " + this.name, e.cause)
311     }
312     if(!matchedLine.isEmpty()){
313         arrayStringBuilder.append(matchedLine)
314     }
315     break
316 }
317 }
318 }
319 }
```

The second array representation, in which an array's elements are distributed over multiple lines, can be identified when only the array's **startTag** appeared within the line, missing the required **endTag**. In such an event, the parser again takes the appropriate measures by setting a variable, which is responsible for reflecting this condition. Then, the **startTag** is removed from the lead of the currently matched line, whereas the optional **arraySplitTag** is cut from the rear, latter only being done if the first array-value appeared in the same line as its array's **startTag**. Another data type check is performed and, in case it is cleared, the value appended to the "StringBuilder".

```
320     else{
321         for(int i = 0; m.find(); i++){
322             String matchedLine = m.group(1).trim()
323
324             if(i == maxSize){
325                 allObjects.add([:])
326             }
327
328             if(entry_it.arraySplitTag != null){
329                 arrayStringBuilder = new StringBuilder()
330                 matchedLine.split(entry_it.arraySplitTag).each{
331                     try {
332                         entry_it.checkType(matchedLine)
```

### 3. Methods

```
333     }
334     catch (ParserUnfitException e) {
335         if(checkIfStrictParsingNeeded(this.routines, matchedLine))
336             throw new ParserUnfitException(e.message + " parsed by
337                                     SimpleTagParser: " + this.name, e.cause)
338     }
339
340     if(arrayStringBuilder.toString().isEmpty())
341         arrayStringBuilder.append(it)
342     else
343         arrayStringBuilder.append("|" + it)
344
345     }
346     allObjects[i].put(entry_it.field,
347                     arrayStringBuilder.toString())
348     arrayStringBuilder = null
349 }
350 else{
351     try {
352         entry_it.checkType(matchedLine)
353     }
354     catch (ParserUnfitException e) {
355         if(checkIfStrictParsingNeeded(this.routines, matchedLine))
356             throw new ParserUnfitException(e.message + " parsed by
357                                     SimpleTagParser: " + this.name, e.cause)
358     }
359
360     allObjects[i].put(entry_it.field,
361                     entry_it.parseField(matchedLine))
362 }
363
364     if(maxSize <= i)
365         maxSize = i + 1
366 }
```

There is another feature built into the **SimpleTagParser** that was foreshadowed via the “singleLineFile” boolean variable. As implied, the **SimpleTagParser** is able to parse files that consist only of a single-line, though such is done sloppily and therefore it is recommended to avoid this feature if other means are available. In contrast to the line-for-line approach, the single-line parsing process does not take the **nestingLevel**, the **domainStartTag**,

### 3.2. Implementation

or the **domainEndTag** into consideration. Further, it is required that each pseudo-object always contains all of its specified key-value pairs, even if the values are possibly empty, as the **optional** property is not employed in any way. The parser will mismatch the key-value pairs with the wrong pseudo-object-maps, if said pairs only appear sporadically. The line-for-line approach puts object-maps into the object-map-list when the current map is perceived as complete, which is the case when an entry collision is detected, the map has as many key-value pairs as there are **entries**, or when the file ends. Contrary, the single-line approach dynamically adds new maps to the list for every new key-value pair matched by the regex engine. When parsing a single-line file, the parser does not iterate over each line and then each **entry**, but instead iterates over each **entry** and then over each of the **entry**'s pattern matched key-value pairs, until there are no more matches left. The first such extracted value is put into the first map, the second value into the second map, and so on. In the event that the **entry** has an **arraySplitTag** set, the parser will try to split the value via this tag. The from the split resulting values' data types are again verified, before said values are concatenated and separated by a "|". The variable "maxSize" is used to keep track of how many object-maps were added to the object-map list and is used to ascertain if the current pattern matches more key-value pairs than there are object-maps, in which case a new map has to be added. The "maxSize" variable should only be incremented by the first executed **entry**, but the program does not prohibit a later **entry** from matching additional key-value pairs, most likely creating new mismatched pseudo-object-maps. The parser in its current state can not detect such a circumstance and will not throw an exception of any kind if it does arise, which is another reasons why this feature is not recommended.

```
367     if(multilineEntry == null && !singleLineFile && domainStartTag ==  
368         null && objectMap.size() == entries.size()){  
369         allObjects.add(objectMap)  
370         objectMap = [:]  
371  
372         if(nestingLevel == -1)  
373             nestingLevelTemp = -1  
374     }  
}
```

### 3. Methods

At the end of each iteration, another check is performed to examine if the current map is complete. This is again done by affirming that each **entry** has its own field in the object-map, in which case a new map is created and the old map once again added to the object-map list. This check is only relevant for the line-for-line parsing process, as the single-line approach is complete in itself.

```
376     if(!singleLineFile && domainStartTag == null && objectMap.size() != 0) {
377         if(objectMap.size() != entries.size()) {
378             entries.each {
379                 if (!it.optional && objectMap.get(it.field) == null)
380                     throw new ParserUnfitException("Required entry '" + it.field +
381                                             "' for object does not appear near the end of the file!")
382             }
383
384             allObjects.add(objectMap)
385         }
386
387         return allObjects
388     }
```

Even if file was parsed successfully, there might still be a complete map cached in the local variable that has yet to be added to the object-map list. This can happen if there are no domain tags specified, which means the parser will only save a map when an **entry** collision is detected. This condition also arises whenever the cached map has less key-value entries than there are parser **entries** specified, with the exception that the missing parser **entries** are **optional**, which ultimately makes the map valid.

```
72     if(multilineEntry != null){
73
74         if((domainEndTag != null && (line.contains(domainEndTag)) ||
75             line.contains(multilineEntry.endTag))) {
76             String extractedVal
77
78             if (domainEndTag != null && line.contains(domainEndTag))
79                 extractedVal = line.subSequence(0,
80                     (int)line.indexOf(domainEndTag)).trim()
81             else
82                 extractedVal = line.subSequence(0,
```

## 3.2. Implementation

```

      (int)line.indexOf(multilineEntry.endTag)).trim()
81
82      if(!extractedVal.isEmpty() && multilineEntry.dataType !=
        EntryDataType.STRING)
83          throw new ParserUnfitException("A multi-line spanning value was
            detected, but the corresponding entry indicates a non-string
            data type, which is not allowed!")
84
85      multilineConcat.append(extractedVal)
86
101      String multilineResult = multilineConcat.toString()
102      objectMap.put(multilineEntry.field, multilineResult)
103      multilineEntry = null
104
105      if(nestingLevelTemp == -1)
106          nestingLevelTemp = nestedCounter
107      }
108      else{
109          multilineConcat.append(line.trim())
110          return
111      }
112  }
```

Mentioned previously in the file-reading-loop section, the code from lines 72 to 168 can only be executed after at least a line has been parsed and one of two notable conditions has arisen. The program initiates this special procedure whenever a multi-line-array condition or multi-line-string condition is detected. Lines 72 to 112 deal with the multi-line-string event and ascertain for each file-line if they contain a tag marking the end, which can be either the specified **endTag**, or even the **domainEndTag**. Either way, the rest, including the **endTag** or **domainEndTag**, is cut from the tail of the line - with the **endTag** having priority in case of both - and the result is being appended to its corresponding multi-line map entry. Line 82 contains the aforementioned check that was missing from the previously elaborated multi-line detection clause. An exception is thrown if a value is possibly a multi-line, but not a string, and is also not a numeral with a missing **endTag** or **domainEndTag**. The omitted lines from 86 to 100 contain another entry collision detection that creates a new map, if no domain tags are set. Finally, the completely concatenated multi-line string is put into the object-map with the key being the **entry**'s specified **field**. Note that this if-branch has no return, as it partially deals with the **domainEndTag**, but does not confirm if

### 3. Methods

the **domainEndTag** completes the current pseudo-object map. Such is done later on in the loop and is the reason why the return statement, a pseudo “continue” statement in Grails’ each-loops, is dropped here.

The last else-branch deals with the case that the end of the multi-line value was not reached yet. In that event, the current line is simply trimmed and appended to the current “StringBuilder” instance.

```
113     else if(arrayEntry != null){
114         String matchedLine
115         if(line.contains(arrayEntry.endTag)){
116             matchedLine = line.replaceAll("(.*)" +
                Pattern.quote(arrayEntry.endTag), "\\$1").trim()
132         try {
133             if(!matchedLine.isEmpty())
134                 arrayEntry.checkType(matchedLine)
135         }
136         catch (ParserUnfitException e) {
137             if(checkIfStrictParsingNeeded(this.routines, matchedLine))
138                 throw new ParserUnfitException(e.message + " parsed by
                SimpleTagParser: " + this.name, e.cause)
139         }
140
141         if(!matchedLine.isEmpty())
142             arrayStringBuilder.append("|" + matchedLine)
143
144         objectMap.put(arrayEntry.field, arrayStringBuilder.toString())
145         arrayStringBuilder = null
146         arrayEntry = null
147
148         if(nestingLevelTemp == -1)
149             nestingLevelTemp = nestedCounter
150     }
```

On the other hand, if a multi-line array was found then, similarly to the multi-line string, the parser has to account for two possible variations appearing in the following lines. Either the next line contains another lone array element, or the end of the array is reached. The first if-branch shown in the listing above handles the latter of those two cases and also accounts for a value being in the same line as the **endTag**. Here, the rest, including the **endTag**, is cut from the tail of the line, before once again confirming

### 3.2. Implementation

that there are no entry collisions within the current object-map. The entry collision is done in the omitted lines 111 to 125. A new map is created in the event of an entry collision, whereas the old map is then added to the object-map list. Another data type check is done before the extracted value is concatenated to the builder and separated by a vertical line “|”. Finally, the complete string is put into the current object-map.

```
151     else{
152         matchedLine = line.replaceAll("(\\s*)" +
153             Pattern.quote(arrayEntry.arraySplitTag), "\\s$1").trim()
154         try {
155             arrayEntry.checkType(matchedLine)
156         }
157         catch (ParserUnfitException e) {
158             if(checkIfStrictParsingNeeded(this.routines, matchedLine))
159                 throw new ParserUnfitException(e.message + " parsed by
160                     SimpleTagParser: " + this.name, e.cause)
161         }
162
163         if(arrayStringBuilder.toString().isEmpty())
164             arrayStringBuilder.append(matchedLine)
165         else
166             arrayStringBuilder.append("|" + matchedLine)
167     }
168     return
169 }
```

The last if-branch deals with lone array elements that are strewn between the **startTag** and **endTag**, with the premise that only one value occurs per line. It is assumed that the token splitting the array, the **arraySplitTag**, is always to the right of such values. This means that the **SimpleTagParser** is not able to parse files whose arrays are mutli-line and whose delimiting token is put to the left of their corresponding elements. This block concludes with a last data type check, before concatenating the value and again separating it with a vertical line “|”.



### 3. Methods

#### SimpleTagEntry

The **SimpleTagParser**'s own **entry** is the **SimpleTagEntry**, extending the base class with the aforementioned **startTag**, **endTag**, and **arraySplitTag** properties.

```
3 class SimpleTagEntry extends DynamicParserEntry{
4
5     String startTag
6     String endTag = null
7     String arraySplitTag = null
8
9     static constraints = {
10         startTag(blank: false, nullable: false)
11         endTag(blank: false, nullable: true)
12         arraySplitTag nullable: true, validator: { val, obj ->
13             return (val != null && obj.endTag != null) || val == null
14         }
15     }
```

The **startTag** always has to be set, as the parser can not parse any tag-based files otherwise. The **endTag** on the other hand is not strictly necessary, but might be needed depending on the file's structure. If the **endTag** is set but the file has none, then the parser will keep looking for the **endTag** in the following lines and unnecessarily append content, instead of stopping at the end of the line. Contrary, specifying a configuration without **endTags**, even though a file does contain such, will result in those tags not being cut from the extracted value. Further, this circumstance will throw an exception if the entry is specified as being a number. E.g. the program tries to parse the extracted data "11231.3453," into a floating point number, which is invalid specifically because of the comma at the end. Similarly, the **arraySplitTag** will always need to have its **entry**'s respective **startTag** and **endTag** specified, or the parser will not be able to identify the start and end of these arrays.

The [addendum A](#) contains examples for different JSON and some JSON-like files, with illustrations on how to configure the DIM accordingly.

### 3.2.3.6. SimpleXMLParser

The **SimpleXMLParser** is a lazy implementation of a [xml and spreadsheetML](#) parser. Unlike other parsers, the **SimpleXMLParser** does build upon existing libraries to parse its files and employs the help of Javax's XML parser package<sup>14</sup>. This package converts a XML file into a DOM document object, allowing for easy access of the values nested within the XML files.

```

10 class SimpleXMLParser extends DynamicParser{
11
12     String superTag = null
13     String excelTag = null
14     Integer startBuffer = 0
15     Integer endBuffer = 0
16
17     static constraints = {
18         superTag nullable: true
19         excelTag nullable: true, validator: { val, obj ->
20             return (val == null && obj.startBuffer == 0 && obj.endBuffer == 0) ||
21                 (val != null && obj.startBuffer > 0)
22     }

```

Before explaining the new properties introduced by the **SimpleXMLParser**, note that the original **field** property has a innate special functionality for this particular parser, when dealing with normal XML, not SpreadsheetML. The **field** variable is used to identify the child-tags nested within the pseudo-object tags. Figure 3.13 shows an example for a XML pseudo-object.

There are four new properties introduced by the **SimpleXMLParser**, with the first one being the **superTag**. The **superTag** is a string property used for identifying pseudo-objects in a XML file, in which the **entries** can in, further consequence, identify and extract the values.

The **excelTag** is needed, in addition to the **superTag**, when dealing with SpreadsheetML. Whereas in normal XML the **field** of each **entry** is used to access the tag of the same name and the values within, in SpreadsheetML each value is encapsulated by the same generic tag. This generic tag has to

<sup>14</sup><https://docs.oracle.com/javase/7/docs/api/javax/xml/parsers/package-summary.html> (Last visited: 11.01.2021)

### 3. Methods

be specified for the **excelTag** property, so that the values can be extracted properly.

Next are the two integers, **startBuffer** and **endBuffer**, which detail how many leading and trailing pseudo-objects, consisting only of meta-data, have to be skipped. These two properties were originally intended to be used only for SpreadsheetML, but were later on implemented for normal XML files too, as a quality-of-life feature.

```
27  @Override
28  ArrayList<Map<String, Object>> parse(File file) throws
    ParserUnfitException, ParserInconsistentException {
32      DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance()
33      DocumentBuilder dBuilder = dbFactory.newDocumentBuilder()
34      Document doc = dBuilder.parse(file)
35      doc.getDocumentElement().normalize()
36
37      NodeList nList = doc.getElementsByTagName(superTag)
```

The implementation of the **parse()** method here begins by initializing a DOM document object, that is used to access the pseudo-objects and their data within the XML. Before extracting said data, the file is normalized to ensure a proper extraction of whole values, in the event that tags and their values occur over multiple lines. Finally, the initialization concludes by fetching a list of all pseudo-objects, identified via the **superTag** property.

```
39      if(startBuffer >= nList.getLength() - endBuffer)
40          throw new ParserUnfitException("The startBuffer and
            endBuffer-configuration for the file: '" + file.getName() + "'
            result in no objects being parsed and passed on!")
41
42      for (int i = startBuffer; i < nList.getLength() - endBuffer; i++) {
43
44          Node nNode = nList.item(i)
45
46          if(nNode.getNodeType() == Node.ELEMENT_NODE) {
47
48              Element eElement = (Element) nNode
49              objectMap = [:]
50
51              entries.eachWithIndex{ entry_it, entry_index ->
```

## 3.2. Implementation

```
52         if(excelTag)
53             objectMap.put(entry_it.field, entry_it.parseField(
54                 eElement.getElementsByTagName(excelTag)
55                     .item(entry_index).getTextContent()))
56         else
57             objectMap.put(entry_it.field, entry_it.parseField(
58                 eElement.getElementsByTagName(entry_it.field)
59                     .item(0).getTextContent()))
60     }
61
62     allObjects.add(objectMap)
```

A check is done to validate that the **startBuffer** is not larger than the number of all available pseudo-objects, minus the **endBuffer**, else an exception is thrown. Assuming the buffers were set correctly, the process then begins to iterate over all pseudo-objects while making sure that each such pseudo-object is actually an element node. Now, there are two approaches to extracting the values. In case that the current file is a SpreadsheetML, denoted by the parser's **excelTag**, then the parser accesses the value via the **excelTag** and the index of the **entry** the parser has in the current iteration. E.g. the second **entry** of the **entries**-list has an index of '2', therefore the second element of a SpreadsheetML pseudo-object corresponds to the second **entry** from all the **entries**. On the other hand, if the file is a regular XML, then the value can be accessed by its unique name, specified by the **field** property of each **entry**. Whatever the case, the extracted value is then saved in the object-map, with the key being the **field** property, and the object-map is then added to the object-map list.

Note that the **SimpleXMLParser** assumes that each XML has a repeatably occurring pseudo-object, that always has the same elements. The **optional** property is completely ignored here. Further, the **superTag** is a property of the parser itself, not the **entries**, meaning only one unique **superTag** is considered per parser. Thus, parsing such a XML file with different non-repeating elements as displayed in figure 3.15, does not return desirable results.

### 3. Methods

#### SimpleXMLElement

The **SimpleXMLElement** does not extend its base class, the **DynamicParserEntry**, in any form.

```
<CD>
  <TITLE>Empire Burlesque</TITLE>
  <ARTIST>Bob Dylan</ARTIST>
  <COUNTRY>USA</COUNTRY>
  <COMPANY>Columbia</COMPANY>
  <PRICE>10.90</PRICE>
  <YEAR>1985</YEAR>
</CD>
```

Figure 3.13.: Example pseudo-object with superTag “CD” and with fields being “TITLE”, “ARTIST”, “COUNTRY”, “COMPANY”, “PRICE” and “YEAR.” Source is w3schools.com<sup>15</sup>.

```
<Row>
  <Cell ss:StyleID="s23"><Data ss:Type="String">Sales for:</Data></Cell>
  <Cell ss:StyleID="s21"><Data ss:Type="DateTime">2004-01-01T00:00:00.000</Data><NamedCell ss:Name="Date"/></Cell>
</Row>

<Row ss:Index="3" ss:StyleID="s23">
  <Cell><Data ss:Type="String">ID Number</Data></Cell>
  <Cell><Data ss:Type="String">Critter</Data></Cell>
  <Cell><Data ss:Type="String">Price</Data></Cell>
  <Cell><Data ss:Type="String">Quantity</Data></Cell>
  <Cell><Data ss:Type="String">Total</Data></Cell>
</Row>

<Row>
  <Cell><Data ss:Type="Number">4627</Data><NamedCell ss:Name="ID"/></Cell>
  <Cell><Data ss:Type="String">Diplodocus</Data><NamedCell ss:Name="Critters"/></Cell>
  <Cell ss:StyleID="s22"><Data ss:Type="Number">22.5</Data><NamedCell ss:Name="Price"/></Cell>
  <Cell><Data ss:Type="Number">127</Data><NamedCell ss:Name="Quantity"/></Cell>
  <Cell ss:StyleID="s22" ss:Formula="=RC[-2]*RC[-1]"><Data ss:Type="Number">2857.5</Data></Cell>
</Row>
```

Figure 3.14.: Three example pseudo-objects with superTag being “Row”, startBuffer being “2”, and the excelTag being “Data”. Fields are irrelevant for SpreadsheetML and can be named arbitrarily. Source is etutorials.org<sup>16</sup>.

<sup>15</sup>[https://www.w3schools.com/xml/xml\\_examples.asp](https://www.w3schools.com/xml/xml_examples.asp) (Last visited: 11.01.2021)

<sup>16</sup><https://etutorials.org/XML/xml+hacks/Chapter+3.+Transforming+XML+Documents/Hack+42+Create+and+Process+SpreadsheetML/> (Last visited: 11.01.2021)

## 3.2. Implementation

```
<widget>
  <debug>on</debug>
  <window title="Sample Konfabulator Widget">
    <name>main_window</name>
    <width>500</width>
    <height>500</height>
  </window>
  <image src="Images/Sun.png" name="sun1">
    <hOffset>250</hOffset>
    <vOffset>250</vOffset>
    <alignment>center</alignment>
  </image>
  <text data="Click Here" size="36" style="bold">
    <name>text1</name>
    <hOffset>250</hOffset>
    <vOffset>100</vOffset>
    <alignment>center</alignment>
    <onMouseUp>
      sun1.opacity = (sun1.opacity / 100) * 90;
    </onMouseUp>
  </text>
</widget>
```

Figure 3.15.: A valid XML, not parseable by the SimpleXMLParser, as there are no repeating pseudo-objects with unique names.

### 3. Methods

#### 3.2.4. Transformation

With the extraction part taken care of by the previous sub-chapter, what follows next is the implementation of the transformation and additionally, because of how the DIM is designed, the loading part of this ETL module. Transformation routines and their underlying procedures are the fundamental building blocks of the transformation and loading block. These routines and procedures are executed in a predetermined sequence, essentially applying transformation methods onto the data and ultimately loading the transformed data into the database.

The following sections will detail the implementation of those **TransformationRoutines**, **TransformationProcedures** and the procedures' **transformation\_methods**.

##### 3.2.4.1. Transformation routines

Whenever data is extracted and passed on to the transformation block, the corresponding **TransformationRoutines** are fetched and executed. Each **DynamicParser** has one or more of those routines stored in a sorted set and executes each routine in an ascending order denoted by the routines' **order\_ids**.

```
5 class TransformationRoutine implements Comparable{
6
7     Integer order_id = 0
8     String target_object
9     boolean to_update = false
10    Map<String, String> update_properties = [:]
11
12    SortedSet<TransformationProcedure> procedures = new
        TreeSet<TransformationProcedure>()
13
14    static hasMany = [procedures: TransformationProcedure]
15
16    static constraints = {
17        target_object nullable: false
18        to_update validator: { val, obj ->
```

## 3.2. Implementation

```
19     return (!val && obj.update_properties.isEmpty()) || (val &&  
20         !obj.update_properties.isEmpty())  
21 }
```

Whereas each parser is responsible for a file and the file's pseudo-objects, the routine is responsible for each domain object that is ultimately persisted into the database. This domain object is reflected by the **target.object** string property, an identifier that is used at runtime to retrieve the domain object class and create instances of such. This **target.object** property can not be null, as the loading process depends on it. This string can be either just the class name, or the class name with its package prepended. It is recommended that the full class name including the package is used to ensure a unique identification. The UI is displaying and using the full name, including the package.

Two properties that are used in combination are the **to.update** boolean and the **update\_properties** map, with the latter being composed of key-and value-strings. The keys are used to access the parser created pseudo-object maps, whose entries are keyed with the **field** specified in each **DynamicParserEntry**. The values on the other hand are used to identify the **target.object**'s properties, with the object's class being set by the **TransformationRoutine**. Both of those properties are used by the transformation block when creating instances of the **target.object**. Whenever this feature is configured, the transformation block will try to retrieve objects whose properties' values match the given values of the pseudo-object maps. The program will instantiate new objects if no matching object can be found, or if the feature is not in use. Whatever the case, both of those properties have to be set together or not at all, as is reflected in their respective constraint.

The last property is a one-to-many relation between the **TransformationRoutines** and the **TransformationProcedures**, latter being detailed in the next section. Each routine can possess multiple procedures that are sequentially executed during the transformation process.



### 3. Methods

#### 3.2.4.2. Transformation procedures

To recap, each **DynamicParser** is responsible for a certain file type and its pseudo-objects it extracts, each **TransformationRoutine** is responsible for the domain objects that are ultimately persisted in the database, and each **TransformationProcedure** is responsible for the **transformation\_method** that is applied onto the data and all of its needed meta-data. Each **DynamicParser** has one or more **TransformationRoutines**, and each routine has one or more **TransformationProcedures** that are executed. Similar to how the parser enacts each routine in a predetermined sequence, each routine's **TransformationProcedure** is also performed in a strictly defined ascending order, which is again denoted by the procedure's own **order\_id** property.

```
5 class TransformationProcedure implements Comparable{
6
7     Integer order_id = 0
8     Closure transformation_method
9     Boolean is_repetitive = true
10    Map<String, String> notable_objects = [:]
11    List<DynamicParserEntry> created_entries = []
12    Boolean temporary = false
13
14    SortedSet<ParamEntryWrapper> parameterWrappers = new
        TreeSet<ParamEntryWrapper>()
15
16    static hasMany = [parameterWrappers: ParamEntryWrapper, created_entries:
        DynamicParserEntry]
```

The frequently mentioned **transformation\_methods** are anonymous functions, or “closures” as called in Grails, which are executed during the transformation process to modify or load data, therefore being a central part of the transformation block. Through this property the respective anonymous functions can be called, applying the desired modification onto the data. The notable part here is that such closures can be persisted in the database as BLOBs, a trait that is reflected in the respective static “mapping” property. This feature allows the DIM to be only configured once and then be employed repetitively and automatically wherever and whenever. Please note that what is persisted is not the executable code per se, but the identification that is resolved by Grails.

### 3.2. Implementation

Next are the two properties **is\_repetitive** and **notable\_objects**, former being a boolean, latter being a map consisting of string keys and values. The keys of the **notable\_objects** are used to access the values of the pseudo-object map created by the parser, whereas the values are literal values specified by the developer and used for finding matching pseudo-objects. Those two properties are used in tandem to provide info for the transformation block on how to handle peculiar data, if at all. In case that **is\_repetitive** is true, each pseudo-object will be transformed as long as it does not have the same values as specified by the **notable\_objects** map. Contrary, if **is\_repetitive** is set to false, then only the pseudo-objects matching **notable\_objects** will be transformed. This feature can be used to sift unwanted data before transforming and loading such, but also to look for a special datum and apply [special transformation methods](#) onto such.

The **DynamicParserEntry**-list, called **created\_entries**, is a list used exclusively by the user interface. This list is set and expanded whenever a **TransformationProcedure** is created whose **transformation\_method** adds key-value pairs to the already parser created pseudo-object.

The boolean called **temporary** is another internally used property, that is not intended to be used by the developer and is therefore not included in the UI. This boolean is used by a [certain transformation method](#) that injects new **TransformationProcedures** into the current **TransformationRoutine** at runtime. Those injected procedures are, as the property implies, temporary and will be removed after they were applied on the data.

The last property in the **TransformationProcedure** class is the **parameter-Wrappers** set and serves as the argument that is passed to its respective **transformation\_method**. This sorted set can consist of up to three **ParamEntryWrappers**, but can also be void of any wrappers in certain cases. The **ParamEntryWrapper** class is devoid of properties, bar another sorted set that is made up of the **ParamEntry** class. The intention behind the **ParamEntryWrappers** is to break up multiple arguments, the **ParamEntries**, into separate coherent sets, so that the concerned **transformation\_method** is able to differentiate between the otherwise indistinguishable groups of such entries. Additionally, the separate **ParamEntryWrapper** class is also a solution for persisting the arguments' structure can be persisted properly.

### 3. Methods

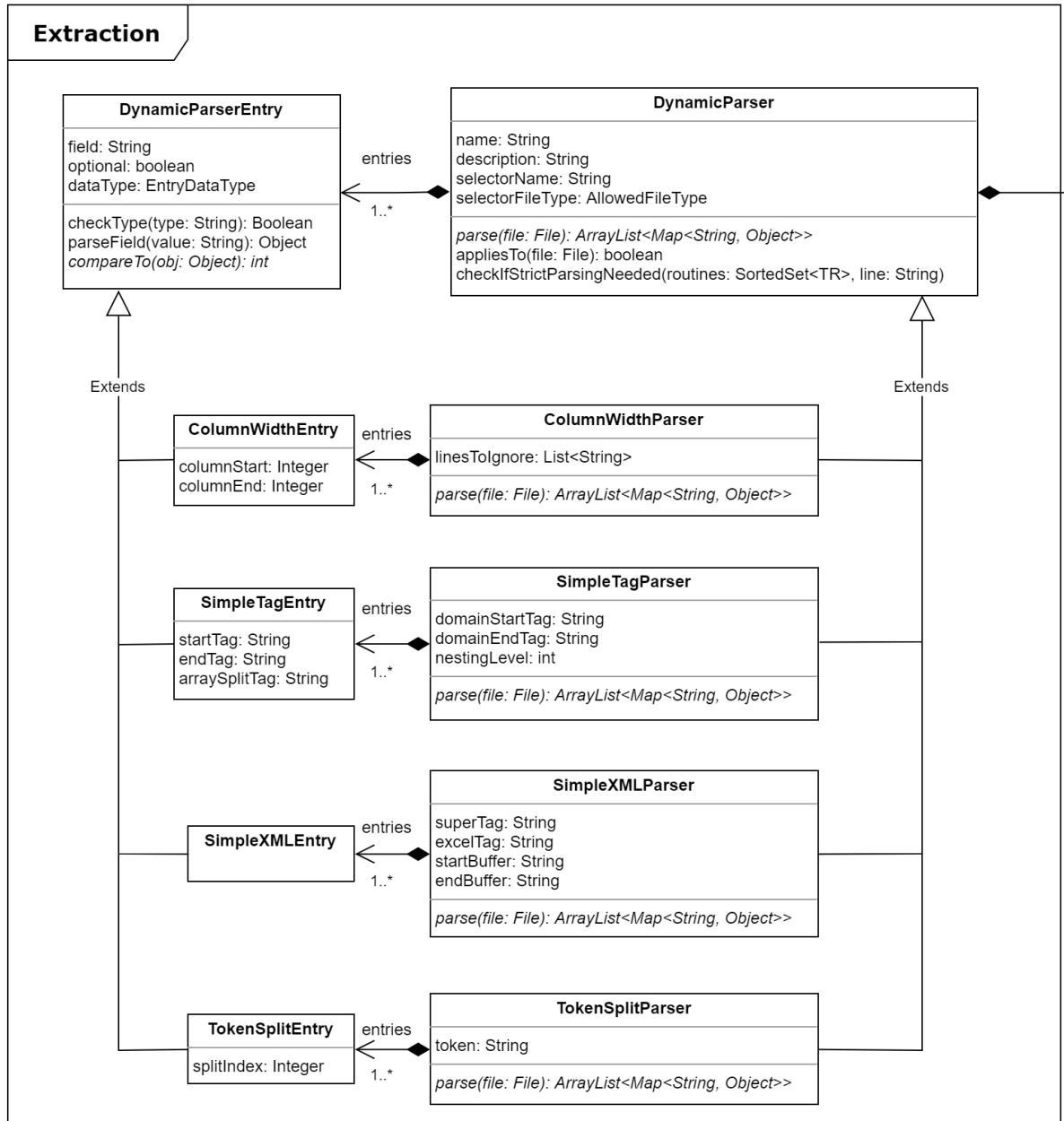
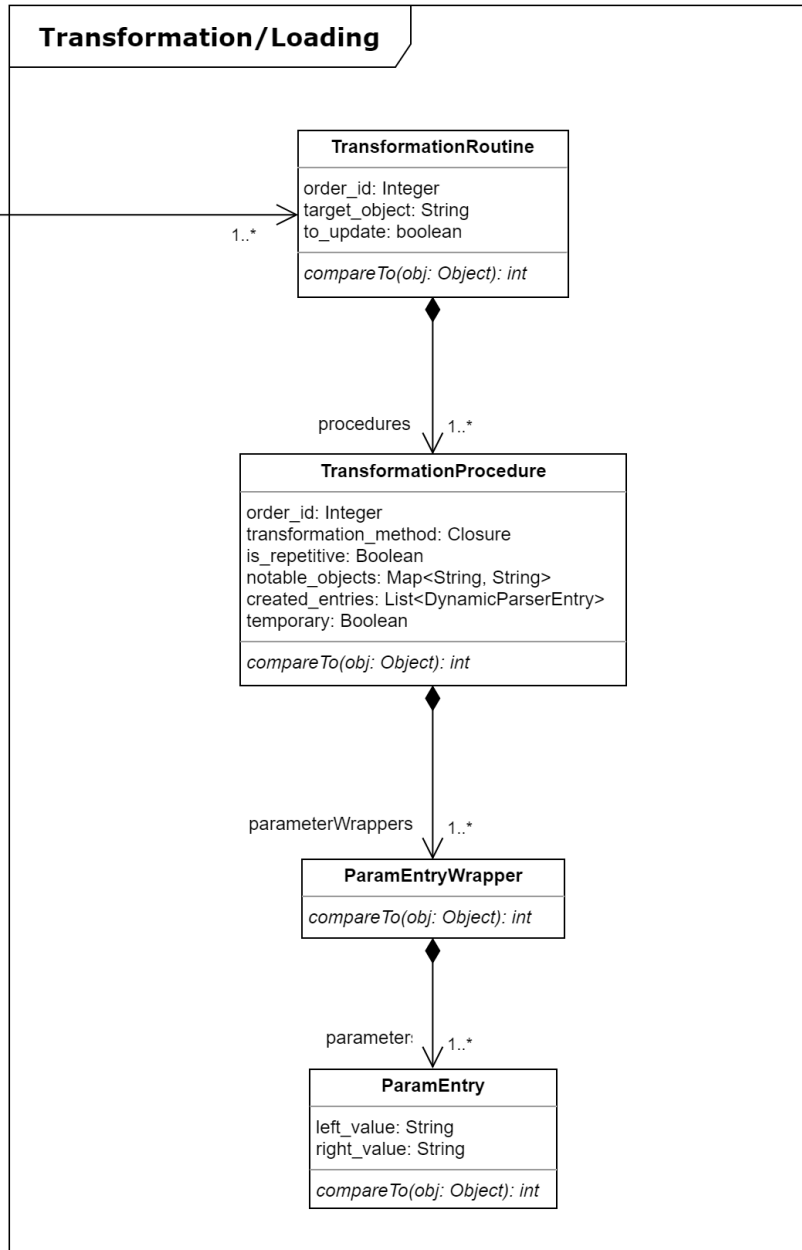


Figure 3.16.: UML diagram of the DIMs domain objects and how they are related to each other.

### 3.2. Implementation



### 3. Methods

#### 3.2.4.3. TransformationService

Similar to how the **FileParsingService** manages the extraction process, so does the **TransformationService** manage the transformation and, additionally, the loading process. For the DIM in particular this means that this service is responsible for executing each routine, each routine's procedures and finally the procedure's underlying **transformation\_methods** onto the data.

The heart of the **TransformationService** is the static method **transformAndLoadData()**. This function is automatically called each time a file was successfully parsed and applies the desired transformation onto the data, which is provided by the aforementioned extraction block during the call to the transformation block. The **TransformationService** does not only manage the transformation, but also contains all provided transformation methods that can be cached in the **TransformationProcedure**'s property of the same name. Those aforementioned methods are fully elaborated in the following section, as this section itself only deals with the management and its helper-functions. A quick rundown on how the **TransformationService** and the **FileParsingService** function is shown in figure 3.10.

```
11 class TransformationService {
12
13     static def transformAndLoadData(ArrayList<Map<String,Object>> data,
14                                     DynamicParser parser) throws Exception{
15
16         parser.routines.each { routine ->
17             ArrayList<Object> objects = new ArrayList<Object>()
18
19             data.each { objects.add(null) }
20
21             for (int procedure_index = 0; procedure_index <
22                 routine.getProcedures().size(); procedure_index++) {
23                 List<TransformationProcedure> procedures =
24                     routine.getProcedures().asList()
25                 TransformationProcedure procedure
26
27                 if(procedures.contains(true))
28                     procedure = procedures[procedures.indexOf(true)]
29                 else
```

## 3.2. Implementation

```
27     procedure = procedures[procedure_index]
28
29     data.eachWithIndex { parse_object, object_index ->
```

The method parameters of **transformAndLoadData()** consist of the actual parsed data - i.e. a file's aforementioned pseudo-objects and the actual parser responsible for such. This circumstance reflects that even after the parsing process has concluded, the parser is still in some form responsible for its generated data and has to give the transformation block additional information on how such is to be handled. The **transformAndLoadData()** method begins by iterating over each of the parser's specified **TransformationRoutines**, of which each in turn creates an empty object list, with the size being equal to the size of the pseudo-object-map list. A trait revealing that the DIM assumes a one to one relationship between the pseudo-objects and the domain objects, meaning each pseudo-object datum generates its own domain object, that is consequently transformed and loaded into the database. After creating the domain object list, the method then begins to iterate over each of the routine's specified **TransformationProcedures**. Due to a [certain transformation method](#), that is able to inject temporary procedures into the routine at runtime, a short if-else is needed to prioritize the aforementioned temporary procedure, in the event that such was created. This is the only intermission in this otherwise straightforward approach, that then continues to iterate over the data, beginning the third and final nested loop.

```
30     boolean is_notable_line = false
31     procedure.notable_objects.each {
32         if (parse_object[it.key] ==
33             parseToCorrespondingType(parse_object[it.key], it.value)) {
34             is_notable_line = true
35         }
36     }
37
38     if ((procedure.is_repetitive && !is_notable_line) ||
39         (!procedure.is_repetitive && is_notable_line)) {
```

Foreshadowed by the **notable\_objects** property, the developer is able to specify key-value pairs that, when equivalent to a datum's own key-value pairs, indicate which data requires special handling. Whenever such an

### 3. Methods

unusual datum is detected, the program then has to either ignore or process such, depending on the context that is denoted by the procedure's **is\_repetitive** boolean.

```
38      Class target_class = getClassFromString(routine.target_object)
39      StringBuilder propertiesErrorString = new StringBuilder()
40
41      if (routine.to_update) {
42          List found_objects = target_class.findAll {
43              routine.update_properties.each {
44                  propertiesErrorString.append(" " + it.value + "->" +
45                      parse_object[it.key] + ";")
46                  eq it.value, parse_object[it.key]
47              }
48          }
49
50          if (found_objects.size() > 1) {
51              StringBuilder updateCriteriaErrorString = new StringBuilder()
52              updateCriteriaErrorString.append("Problematic object is of
53                  type '${target_class.name}' with its compared properties
54                  being values:${propertiesErrorString.toString()}")
55
56              throw new ValidationException("Found multiple objects in
57                  database which apply for the 'update' criteria! " +
58                  "Update criteria (or a combination of it) must be unique
59                  to each object! " +
60                  updateCriteriaErrorString.toString())
61          }
62
63          objects[object_index] = found_objects[0]
64
65          if (objects[object_index] == null)
66              objects[object_index] = target_class.newInstance()
67
68          procedure.transformation_method(procedure, parse_object,
69              objects[object_index])
70      }
71  }
```

In line 38, the routine's **target\_object**, the domain object class specified in the routine, is resolved from the string and stored in a local "class" variable. If the developer wishes for each respective database object to be

### 3.2. Implementation

updated, instead of being appended to its respective database table, then this can be done via the **to.update** property examined in line 41. Here, the method fetches an object instance of the previously resolved target class from the database. Values from the routine's **update\_properties** are referring to domain object properties' names, whereas keys are used for accessing values from the parsed data, for the comparison in the database search query. An exception will be thrown if multiple objects match this criteria, as the intent is to only update a single object, not multiple, meaning the update configuration is not strict enough. A new object will be instantiated whenever there is no **update\_criteria** defined, or whenever the database does not return a fitting object for a update query.

Line 63 is significant, as this is where the previously persisted anonymous closure is called. I.e. this line calls the specified transformation method, finally enacting the actual purpose of the transformation block. It does not matter which transformation method is applied, as each such method will alter at least one and at most two of the given arguments. The first of those two alterable arguments is the respective pseudo-object map, or datum, that is created by the parser and which is the argument the transformation methods are responsible for. The second argument, that can be manipulated, is the object instance, whose class is given by the routine via the **target.object** property. Object instances are mostly only handled by the loading methods, that are also implemented within the transformation block and whose signature is indistinguishable from the transformation methods.

```
67     if (procedure.temporary) {
68
69         def wrapsToDelete = new ArrayList<ParamEntryWrapper>()
70         procedure.parameterWrappers.each { wrap ->
71             if ((TransformationProcedure.findAll("from
72                 TransformationProcedure where ? in
73                 elements(parameterWrappers)", [wrap]).size()) == 1) {
74                 def paramsToDelete = new ArrayList<ParamEntry>()
75                 wrap.parameters.each { paramsToDelete.add(it) }
76                 wrap.parameters.clear()
77                 paramsToDelete.each { it.delete(flush: true) }
78                 wrapsToDelete.add(wrap)
79             }
80         }
81     }
```



### 3. Methods

```
79         wrapsToDelete.each { wrap ->
80             procedure.parameterWrappers.remove(wrap)
81             wrap.delete(flush: true)
82         }
83
84         routine.procedures.remove(procedure)
85         routine.save(flush: true)
86
87         procedure.delete(flush: true)
88         procedure_index--
89     }
90 }
91 }
92 }
93 }
```

Each temporary procedure is discarded by the **transformAndLoadData()** method after its execution. Here, all of the method's parameters are deleted, the method removed from the routine, and the procedure loop index is decremented so it continues on from where it left off, before the temporary procedure was executed.

```
95     static def getClassFromString(String className) throws
96         IncorrectSpecifcationException{
97         def grailsApplication = Holders.getGrailsApplication()
98         Class object = grailsApplication.getDomainClass(className)?.clazz
99
100         if(object == null)
101             object = grailsApplication.domainClasses.find { it.clazz.simpleName
102                 == className }?.clazz
103
104         if(object == null)
105             throw new IncorrectSpecifcationException("The specified
106                 classname-string does not match any domain classes")
107
108         return object
109     }
```

Next up, after the main method, are the helper methods. These helper methods are used not only by the main managing method, but some are also used by the transformation methods themselves. The **getClassFromString()** method is one such method that is used universally, as it is the

### 3.2. Implementation

helper method that is responsible for resolving the class name strings and returning the respective class objects. “Class” objects contain meta information on the domain object’s classes and can be used to create instances of these objects. This method accepts one of two class-string definitions as its argument; Either the full name of the class including the package it is contained in, or just the lone name of the class itself. The former variant is preferential to the latter to avoid any potential mix ups. In the event that the class string was misspecified, an exception will be thrown to indicate that no class of such name could be found.

```
108 static def parseToCorrespondingType(def obj, String fieldToParse){
109     if(obj instanceof Float)
110         try{
111             return Float.parseFloat(fieldToParse)
112         }
113         catch(Exception e){
114             throw new IncorrectSpecificationException("Specified entry cannot
115                 be parsed to the data type of the object it is assigned to.")
116         }
117
118     return fieldToParse.toString()
119 }
120
121 catch(Exception e){
122     throw new IncorrectSpecificationException("Specified entry cannot
123         be parsed to the data type of the object it is assigned to.")
124 }
125
126 else
127     return fieldToParse
128 }
```

The **parseToCorrespondingType** method is very similar to the **DynamicParserEntry**’s **parseField()** method, in that it also parses a value to one of the allowed primitive data types. The difference here being that **parseToCorrespondingType** actually uses the given object’s property, in which the value will be saved in, to identify the primitive data type the value has to be parsed to. Another peculiarity is that **parseToCorrespondingType** allows the use of double, whereas **parseField()** only accounts for float when dealing with floating point values. The double was later added to the DIM’s transformation data types due to the fact that Java’s float might allow larger numerals than the respective database’s float, thus requiring the larger

### 3. Methods

double data type for the domain object's properties as a workaround.

This particular method, just like **parseField()**, does not take many different numeral representations into account and follows the same rules each primitive data types "parseTo" method follows. E.g. a "123,456,789" string is not parsable to integer due to the commas. Another problematic example would be the european localization that swaps commas with periods and vice versa, a condition that might not be parsable depending on the localization of the device the DIM runs on. All of this has to be taken into consideration by the developer themselves. A possible workaround can be implemented via a [certain transformation method](#), in combination with the fact that the transformation block will ultimately try to parse each value to its object property's data type, in which the value is to be saved.

Back to the issue at hand, the displayed code above only shows the first if-else-branch, but the method continues on for each possible data type. Data types are limited to the following as of now:

- boolean
- double
- float
- integer
- long
- string

Note that double is not implemented in the extraction block, only the transformation block.

```
155 static def checkIfParamsSetCorrectly(String methodName,
    TransformationProcedure procedure) throws
    IncorrectSpecificationException{
156 if(methodName != "cacheInfoForCrossProcedure" &&
    procedure.parameterWrappers.size() !=
    MethodInfo.getWrapperCount(methodName))
157 throw new IncorrectSpecificationException("Found
    ${procedure.parameterWrappers.size()} instead of " +
158 "${MethodInfo.getWrapperCount(methodName)} parameter-wrappers in
    method: '${methodName}'")
159
160 ArrayList<ParamEntryWrapper> wrappers =
    procedure.parameterWrappers.asList()
161 for(int i = 0; i < procedure.parameterWrappers.size(); i++) {
```

## 3.2. Implementation

```
162
163     if (!MethodInfo.isRepeatable(methodName, i) &&
164         wrappers[i].parameters.size() > 1) {
165         throw new IncorrectSpecificationException("Found a parameterlist of
166             size bigger than 1 in wrapper #${i} " +
167             "of method: '${methodName}' in
168             '${TransformationRoutine.find{procedures{id == 1}}}'" +
169             ", even though it is a non-repeatable parameter type!")
170     }
171     else if(wrappers[i].parameters.size() == 0) {
172         throw new IncorrectSpecificationException("Found a parameter-list
173             of size 0 in wrapper #${i} of method: '${methodName}' " +
174             "in '${TransformationRoutine.find{procedures{id == 1}}}'")
175     }
176
177     if(MethodInfo.getSecondClassPropertiesPosition(methodName) != null &&
178         i == 0
179         && (wrappers[i].parameters.first().right_value == "null"
180             || wrappers[i].parameters.first().right_value == ""
181             || wrappers[i].parameters.first().right_value == null))
182         throw new IncorrectSpecificationException("Found an empty classname
183             the right value of wrapper #0 of method: '${methodName}' " +
184             "in '${TransformationRoutine.find{procedures{id == 1}}}'")
185     }
186 }
```

Last of the helper methods is **checkIfParamsSetCorrectly()**. It is used at the beginning of each transformation method, to validate that the polymorphous and arbitrary parameters, given to the method's as an argument, are set correctly. This method makes use of an enumeration called **MethodInfo**. **MethodInfo** contains various hard coded meta information on each transformation method and is used extensively by the UI to assert and guide a proper configuration of each transformation methods.

Considering that the UI already does the majority of the creation validation, the checks here are simple and more of a "just in case", than necessity. The first check affirms that each wrapper does not have multiple parameters, if such a wrapper is not specified as such. Although the wrappers can contain more than one parameter if specified as "repeatable", implying that each additional parameter is used for the same purpose, this notation is misleading, as there are transformation methods that require multiple

### 3. Methods

distinct and differently used parameters in the same wrapper.

The next verification is fairly simple, as it only makes sure that each wrapper is not devoid of any parameters.

Finally, a more specific check confirms that whenever a wrapper must contain a class name, which has to be in the right value of the first parameter, that this class name is not empty or any variation of “null”.

#### 3.2.4.4. Transformation methods

Further down, but still contained within the **TransformationService**, are the respective implementations of all the transformation methods. The DIM’s transformation methods are all uniform in design, with each such method having the same method parameters and no explicit return values, thus making their name the only outward characteristic on which they can be differentiated. Leading the method parameters is the **Transformation-Procedure**, which is responsible for supplying the executed **transformation\_method** with all its needed meta-data and arguments, latter which appears in the form of **ParamEntryWrappers** and **ParamEntries**. Further method parameters include the aforementioned current datum, onto which the transformation is applied, and the domain object instance, onto which the load is executed.

This chapter not only gives a short description on what each transformation method does, but also details what arguments each transformation method requires and how they are composed, illustrated via tables. The tables have a certain layout, giving information on the parameters’ structure. All gray fields are **ParamEntries** and each new **ParameterWrapper** is indicated by a double horizontal line. Further, vertical dots in a field indicate that the previous **ParamEntry** can be repeated any number of times. Note that the **ParamEntries** are pair tuples, having a left and right value of type string. There is no legitimate reason for the use of this particular data structure. Moreover, the first few implemented transformation methods worked well with such, ultimately resulting in the **ParamEntries** being kept in that form. Examples on how each transformation and loading method can be configured, are shown in their respective [addendum](#).

### 3.2. Implementation

**General limitations of the transformation methods:** As of now, transformation methods are only able to handle primitive data types. Further, of all possible primitive data types, the DIM only accounts for float, integer, long, boolean, and string, with the latter technically not being a primitive data type, but being handled as such. The DIM and all of its transformation methods are not built for working with objects whose properties are collections of any kind, with an exception being the “hasMany” property, that can be only set by a [single specific relational transformation method](#). The DIM is also not able to create or update properties that are objects themselves, as the transformation methods’ design is built and restricted around the single object class that is defined by the respective routine’s **target\_object** property.

Many transformation methods do not affirm themselves, whether a key delivers a valid value from the pseudo-object-map, as it is assumed that the following operations working with said value would throw an exception anyways. The DIM was designed to catch inconsistencies as early as possible, even to throw exceptions that are would not necessarily stop the program. This is done to bring attention to any possible shortcomings in the configuration of the transformation methods, hopefully preventing errors and misparsed or mistransformed data later on during automated jobs. Some logical assertions were omitted from the runtime of the transformation block, as it is assumed that the developer uses the respective UI to configure the DIM, which in turn contains multiple such assertions to catch any configuration inconsistencies at the time of creation.

Any transformation method working with numerals, casts those numerals to float, as this is the most lenient data type that works with any operation - e.g. division - and can be cast back to other data types with mostly acceptable precision. The drawback due to this is the floating point error, a ubiquitous problem that should be taken into account by the developer. In case that the provided precision is too inaccurate, it is advised to use the DIM only as an extraction and loading tool, and to apply the necessary transformations later on via external manipulation. I.e. creating an ELT application, with the DIM only handling the Extraction and Loading part.

All transformation methods that work with strings, assume for all given values to be strings by default and do not restrict for such in any way. I.e.

### 3. Methods

some methods might throw an exception if a certain operation is applied on non-strings, whereas others might silently convert them to strings. Some other transformation methods do examine if a value is null and, if such is the case, skip the value. Such is necessary because the UI might deliver a null-value in certain corner cases, a circumstance that might have to be accounted for by the developer by, for example, transforming or replacing such null-values with a default value.

**3.2.4.4.1. appendString** appends each **ParamEntry**'s **right\_value** to its respective value in the pseudo-object map.

left_value	right_value
Map key for value to which the <b>right_value</b> is appended	Value that is appended
:	:

```
183 static def appendString(TransformationProcedure procedure, Map<String,  
    Object> datum, Object object_instance){  
184     checkIfParamsSetCorrectly("appendString", procedure)  
185  
186     def parameters = procedure.parameterWrappers.first().parameters  
187  
188     parameters.each{ parameter ->  
189         if(parameter.left_value != null && parameter.left_value != "" &&  
            parameter.left_value != "null" && datum[parameter.left_value] !=  
            null)  
190             datum[parameter.left_value] = datum[parameter.left_value] +  
                parameter.right_value  
191     }  
192 }
```

The method iterates over all its repeatable **ParamEntries** and appends each string to the corresponding datum's accessed value. Note that the method does nothing if a specified key is not valid. Additionally, because of the use of the '+' operator, any datum containing a number is implicitly cast to string during the concatenation.

**3.2.4.4.2. prependString** prepends each **left.value** to its counterpart value in the pseudo-object-map. This method takes the same arguments as **appendStrings**, but transposes those arguments to imply a prepending instead.

left.value	right.value
Value that is prepended	Map key for value to which the <b>left.value</b> is prepended
:	:

```

194 static def prependString(TransformationProcedure procedure, Map<String,
    Object> datum, Object object_instance){
195     checkIfParamsSetCorrectly("prependString", procedure)
196
197     def parameters = procedure.parameterWrappers.first().parameters
198
199     parameters.each{ parameter ->
200         if(parameter.right_value != null && parameter.right_value != "" &&
            parameter.right_value != "null" && datum[parameter.right_value]
            != null)
201             datum[parameter.right_value] = parameter.left_value +
                datum[parameter.right_value]
202     }
203 }
```

This method too does nothing if a specified key is not valid and also, because of the '+' operator, casts any datum containing a number implicitly to string during the concatenation.

**3.2.4.4.3. trimField** can be used to trim any pseudo-object-map values of any leading or trailing spaces. Usually this method is not necessary, as most parsers already trim their extracted values, but was nonetheless included into the transformation repertoire just in case.

left.value	right.value
Map key for value to be trimmed	Map key for value to be trimmed
:	:



### 3. Methods

```
205 static def trimField(TransformationProcedure procedure, Map<String,  
    Object> datum, Object object_instance){  
206     checkIfParamsSetCorrectly("trimField", procedure)  
207  
208     def parameters = procedure.parameterWrappers.first().parameters  
209     parameters.each{ parameter ->  
210         if(parameter.left_value != null && parameter.left_value != "" &&  
            parameter.left_value != "null" && datum[parameter.left_value] !=  
            null)  
211             datum[parameter.left_value] = datum[parameter.left_value].trim()  
212         if(parameter.right_value != null && parameter.right_value != "" &&  
            parameter.right_value != "null" && datum[parameter.right_value]  
            != null)  
213             datum[parameter.right_value] = datum[parameter.right_value].trim()  
214     }  
215 }
```

The code iterates over each **ParamEntry** and trims each value, if it is set. The **ParamEntry** tuple does not have to be filled, i.e. only the left or right value can be set. It is assumed that the specified data contains strings, otherwise an exception will be thrown.

**3.2.4.4.4. splitStringField** is used for splitting a string, retrieved from the pseudo-object-map, into multiple smaller strings. Split strings are then saved back into new or already existing pseudo-object-map entries. The token or Regex, that is used for the split, has to be specified in the first **ParamEntry** on the right value. Note that this method uses Java's own "split()"<sup>17</sup> method, which means a regular expression (regex) can be given to split the string instead of just a simple token. A good use case for this transformation method would be the "array string values" generated by the **SimpleTagParser**.

---

<sup>17</sup>[https://docs.oracle.com/javase/7/docs/api/java/lang/String.html#split\(java.lang.String\)](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html#split(java.lang.String)) (Last visited: 11.01.2021)

### 3.2. Implementation

left.value	right.value
Map key for value to be split	Token or regex that is used for the split
New or existing map key into which a single split result will be saved in	Index that is used for accessing one value from the split result
:	:

```
217 static def splitStringField(TransformationProcedure procedure,
218   Map<String, Object> datum, Object object_instance) {
219   checkIfParamsSetCorrectly("splitStringField", procedure)
220
221   def param = procedure.parameterWrappers.first().parameters.first()
222   def mapping = procedure.parameterWrappers.asList()[1].parameters
223   def line_to_split = datum[param.left_value]
224   def split_lines = line_to_split.split("\\s*" + param.right_value + "\\s*")
225
226   mapping.each{
227     def index = Integer.parseInt(it.right_value)
228
229     if(index < split_lines.size())
230       datum[it.left_value] = split_lines[index]
231   }
```

The implementation iterates over the second wrapper's repeatable parameters and assigns each split result, that is accessed via a specified index, to its corresponding map entry. It is recommended to define more indices, than there are values expected to be delivered by the split, as the method does not consider any values beyond the specified indices. Contrary, if there are more indices than split values, then those extra indices are simply omitted. No alterations are applied onto the original string, used for the split. The "split()" method's argument already has regex in place to remove leading and trailing spaces, which should be taken into consideration by the developer in case that additional regex is required. It is assumed, that the data to be split, will be a string, otherwise an exception will be raised.

### 3. Methods

**3.2.4.4.5. concatenateFields** does the exact opposite of **splitStringField**, by concatenating multiple pseudo-object-map values and then saving the result into a new or already existing pseudo-object-map entry.

left_value	right_value
New map key where the result will be saved	Optional string that is put in between each concatenation
Map key where value for concatenation is fetched from	Map key where value for concatenation is fetched from
:	:

```
233 static def concatenateFields(TransformationProcedure procedure,
234                               Map<String, Object> datum, Object object_instance) {
235     checkIfParamsSetCorrectly("concatenateFields", procedure)
236
237     String fieldname =
238         procedure.parameterWrappers.first().parameters.first().left_value
239     String mid_string =
240         procedure.parameterWrappers.first().parameters.first().right_value
241     def fields_to_append =
242         procedure.parameterWrappers.asList()[1].parameters
243
244     StringBuilder result = new StringBuilder("")
245     fields_to_append.eachWithIndex { it, index ->
246         if (it.left_value != null && it.left_value != "" && it.left_value !=
247             "null" && datum[it.left_value] != null) {
248             if (!result.toString().isEmpty())
249                 result.append(mid_string)
250
251             result.append(datum[it.left_value])
252         }
253         if (it.right_value != null && it.right_value != "" && it.right_value
254             != "null" && datum[it.right_value] != null) {
255             if (!result.toString().isEmpty())
256                 result.append(mid_string)
257
258             result.append(datum[it.right_value])
259         }
260     }
261
262     datum[fieldname] = result.toString()
263 }
```

## 3.2. Implementation

The implementation iterates through the second wrapper's repeatable parameters and appends a value if a key is specified for such, while also prepending the **right\_value** of the first parameter if the "StringBuilder" is not empty anymore. The aforementioned **right\_value** of the first parameter is used here as an optional delimiter and can be left empty if not needed. The result of the iteration is then saved in a map entry with a key specified by the **left\_value** of the first parameter of the first wrapper. An assertion examines each key to make sure it returns a valid value in the map - i.e. any value not null.

**3.2.4.4.6. calculateSum** is the first transformation methods that applies arithmetic operations onto numerical data. As implied by the name, this method calculates the sum over an arbitrary number of pseudo-object-map values. The result is saved as a float in a specified new or already existing map entry. Any output float will be cast to a primitive data type later on during the load, depending on the corresponding object property's data type. Note that the floating point error is a potential issue here.

left_value	right_value
New map key where the result will be saved	Multiplier that is applied onto each value/result (default is 1)
Map key where value for summation is fetched from	Map key where value for summation is fetched from
:	:

```
259 static def calculateSum(TransformationProcedure procedure, Map<String,
    Object> datum, Object object_instance) {
260     checkIfParamsSetCorrectly("calculateSum", procedure)
261
262     String multiplier =
        procedure.parameterWrappers.first().parameters.first().right_value
263     String fieldName =
        procedure.parameterWrappers.first().parameters.first().left_value
264     def fields_to_aggregate =
        procedure.parameterWrappers.asList()[1].parameters
265     ArrayList<Float> converted_fields = new ArrayList()
266     float result = 0
```

### 3. Methods

```
267     fields_to_aggregate.each{
268         if(it.left_value != "" && it.left_value != null && it.left_value !=
269            "null" && datum[it.left_value] != null)
270             converted_fields.add(
271                 Float.parseFloat(datum[it.left_value].toString()))
272         if(it.right_value != "" && it.right_value != null && it.right_value
273            != "null" && datum[it.right_value] != null)
274             converted_fields.add(
275                 Float.parseFloat(datum[it.right_value].toString()))
276     }
277     converted_fields.each{
278         result += (it * ((multiplier == null || multiplier.equals("")) ||
279             Float.parseFloat(multiplier) == 0) ? 1.0f :
280             Float.parseFloat(multiplier)))
281     }
282     datum[fieldName] = result
283 }
```

Before executing the actual calculation, the program makes sure that each of the developer set values is valid, i.e. the keys are valid and return a result. Further, each value is converted to float, which is done by casting the object to string - regardless if the object is already an actual float - and then parsing it back to float. An exception is thrown if this is not possible. In case of success however, the float value is cached for later. The calculation itself applies a multiplier to each value, if such a multiplier was set. Whatever the case, the result is saved in a specified new or already existing map entry.

**3.2.4.4.7. calculateMean** applies a mean calculation over multiple pseudo-object-map values. The result is saved as a float in a specified new or pre-existing map entry. Resulting floats can be cast to any type later during the load, depending on the corresponding object properties data type. Like most methods dealing with floats, the floating point error is also a potential problem here.

### 3.2. Implementation

left_value	right_value
New map key where the result will be saved	Multiplier that is applied onto each value/result (default is 1)
Map key where value for the mean calculation is fetched from	Map key where value for the mean calculation is fetched from
:	:

```

284 static def calculateMean(TransformationProcedure procedure, Map<String,
    Object> datum, Object object_instance) {
285     checkIfParamsSetCorrectly("calculateMean", procedure)
286
287     String multiplier =
        procedure.parameterWrappers.first().parameters.first().right_value
288     String fieldName =
        procedure.parameterWrappers.first().parameters.first().left_value
289     def fields_to_aggregate =
        procedure.parameterWrappers.asList()[1].parameters
290     ArrayList<Float> converted_fields = new ArrayList()
291     float result = 0
292
293     fields_to_aggregate.each{
294         if(it.left_value != "" && it.left_value != null && it.left_value !=
            "null" && datum[it.left_value] != null)
295             converted_fields.add(
296                 Float.parseFloat(datum[it.left_value].toString()))
297         if(it.right_value != "" && it.right_value != null && it.right_value
            != "null" && datum[it.right_value] != null)
298             converted_fields.add(
299                 Float.parseFloat(datum[it.right_value].toString()))
300     }
301
302     float count = 0
303     converted_fields.each{
304         result += (it * ((multiplier == null || multiplier.equals("")) ||
            Float.parseFloat(multiplier) == 0) ? 1.0f :
            Float.parseFloat(multiplier)))
305         count++
306     }
307     result /= count
308
309     datum[fieldName] = result
310 }

```

### 3. Methods

The implementation is almost identical to **calculateSum**, again beginning by trying to cast each valid numeral value to float and storing them into a map. During the calculation, the method applies a multiplier - if such is specified - onto each value. In contrast to **calculateSum**, **calculateMean** keeps track of a counter variable, that is incremented for each value, and then divided by the result to calculate the mean, instead of the sum.

This method also concludes by saving the result in a specified new or already existing map entry.

**3.2.4.4.8. arithmeticOperation** applies one of five possible binary arithmetic operations onto the data. Available operations choosable by the developer consist of the following:

- (+) - Addition
- (-) - Subtraction
- (\*) - Multiplication
- (/) - Division
- (%) - Remainder

Any selected operation is then applied onto the two operands, both of which can be either a value from the pseudo-object-map, or a developer input value. Whatever the operation, the result is then saved in a specified new or already existing map entry.

left_value	right_value
New map key where the result will be saved	Binary operator (+, -, *, /, %)
Left operand which can be a map key or user input number	Right operand which can be a map key or user input number

```
312 static def arithmeticOperation(TransformationProcedure procedure,
313                               Map<String, Object> datum, Object object_instance) {
314     checkIfParamsSetCorrectly("arithmeticOperation", procedure)
315
316     String operator =
        procedure.parameterWrappers.first().parameters.first().right_value
    String fieldName =
        procedure.parameterWrappers.first().parameters.first().left_value
```

## 3.2. Implementation

```
317 def operands =  
    procedure.parameterWrappers.asList()[1].parameters.first()  
318  
319 Object left_operand  
320 Object right_operand  
321  
322 if(datum[operands.left_value] != null)  
323     left_operand = datum[operands.left_value]  
324 else  
325     left_operand = Float.parseFloat(operands.left_value)  
326  
327 if(datum[operands.right_value] != null)  
328     right_operand = datum[operands.right_value]  
329 else  
330     right_operand = Float.parseFloat(operands.right_value)  
331  
332 if((datum[operands.left_value] != null && datum[operands.left_value]  
    instanceof String) || (datum[operands.right_value] != null &&  
    datum[operands.right_value] instanceof String))  
333     throw new ValidationException("Found a 'String' object in the  
        'unaryOperation' transformation method. Unary arithmetic  
        operations can only applied on numbers.")  
334  
335 if(operator == "+")  
336     datum[fieldName] = left_operand + right_operand  
337 else if(operator == "-")  
338     datum[fieldName] = left_operand - right_operand  
339 else if(operator == "*")  
340     datum[fieldName] = left_operand * right_operand  
341 else if(operator == "/")  
342     datum[fieldName] = left_operand / right_operand  
343 else if(operator == "%")  
344     datum[fieldName] = left_operand % right_operand  
345 }
```

The implementation begins by ascertaining if any of the operands are map keys, or if they are just simple numerals. This is done by trying the **ParamEntries** values as map keys first and, if they do deliver a valid value, setting those values as the operands. Contrary, whenever the map returns a null value for an attempted key, the method will try to cast the attempted key to float and use it as an operand instead. A final check is done to verify that the pseudo-object-map values used as operands are not strings, before



### 3. Methods

carrying out the desired operation and saving the result in a specified map entry, new or preexisting.

**3.2.4.4.9. unaryArithmeticOperation** applies a unary operation onto the data. As of now the selection contains the following unary operations:

- (++) - Incrementation
- (--) - Decrementation
- (-) - Negation

Each selected operation is then applied onto its paired value from the pseudo-object-map.

left_value	right_value
Map key for operand	Unary operator (++ , -- , -)
:	:

```
347 static def unaryArithmeticOperation(TransformationProcedure procedure,
348   Map<String, Object> datum, Object object_instance) {
349   checkIfParamsSetCorrectly("unaryArithmeticOperation", procedure)
350   def parameters = procedure.parameterWrappers.first().parameters
351   parameters.eachWithIndex{ parameter, parameter_index ->
352     if(datum[parameter.left_value] instanceof String)
353       throw new ValidationException("Found a 'String' object in the
354         'unaryOperation' transformation method. Unary arithmetic
355         operations can only applied on numbers.")
356     if(parameter.right_value == "++" && datum[parameter.left_value] !=
357       null)
358       datum[parameter.left_value]++
359     else if(parameter.right_value == "--" && datum[parameter.left_value]
360       != null)
361       datum[parameter.left_value]--
362     else if(parameter.right_value == "-" && datum[parameter.left_value]
363       != null)
364       datum[parameter.left_value] = -datum[parameter.left_value]
365   }
366 }
```

## 3.2. Implementation

The implementation affirms that the relevant pseudo-object-map values are not strings, before applying the designated unary operations onto the data, altering the original values. Keys returning null values result in the corresponding data being skipped.

**3.2.4.4.10. regexReplace** allows the developer to replace all regex matched substrings with a specified string. This method uses Javas own `replaceAll()`<sup>18</sup> method, thus having the same behaviour.

left_value	right_value
New map key where the result will be saved	Map key for data onto which <code>replaceAll()</code> is applied on
Regular expression	Replacement string

```
365 static def regexReplace(TransformationProcedure procedure, Map<String,
    Object> datum, Object object_instance) {
366     checkIfParamsSetCorrectly("regexReplace", procedure)
367
368     String fieldName =
        procedure.parameterWrappers.first().parameters.first().left_value
369     String fieldToFetch =
        procedure.parameterWrappers.first().parameters.first().right_value
370     String regexPattern =
        procedure.parameterWrappers.first().parameters.first().left_value
371     String replacement =
        procedure.parameterWrappers.first().parameters.first().right_value
372
373     if(datum[fieldToFetch] != null)
374         datum[fieldName] = datum[fieldToFetch].replaceAll(regexPattern,
            replacement)
375 }
```

The implementation is lean, only calling “`replaceAll()`” onto the data and saving the result in a specified new or already existing pseudo-object-map entry.

---

<sup>18</sup>[https://docs.oracle.com/javase/7/docs/api/java/lang/String.html#replaceAll\(java.lang.String,%20java.lang.String\)](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html#replaceAll(java.lang.String,%20java.lang.String)) (Last visited: 11.01.2021)

### 3. Methods

Note that any datum whose key returns a null value, is skipped. Additionally, the “replacement” value here is not quoted, meaning slashes “\” and dollar signs “\$” have special meaning and have to be escaped by the developer themselves, if they are to be taken literal.

**3.2.4.4.11. setValueFromOptionalValues** looks through each specified pseudo-object-map entry and puts the first value deemed valid into a specified new or already existing map entry. This transformation method is similar to **concatenateFields**, but differs in the way that each map key is first examined if it returns a valid value. Further, **setValueFromOptionalValues** immediately returns after the first valid value is discovered, making it an ideal method for consolidating multiple optional map entries of the same archetype. A good application for this method would be the optional entries generated by the **TokenSplitParser**.

left.value	right.value
New map key where the result will be saved	NOT IN USE
Map key for potential value	Map key for potential value
:	:

```
377 static def setValueFromOptionalValues(TransformationProcedure procedure,
378   Map<String, Object> datum, Object object_instance) {
379   checkIfParamsSetCorrectly("setValueFromOptionalValues", procedure)
380
381   def fieldName =
382     procedure.parameterWrappers.first().parameters.first().left_value
383   def fields = procedure.parameterWrappers.asList()[1].parameters
384
385   fields.each{
386     if(datum[it.left_value] != "" && datum[it.left_value] != null &&
387       datum[it.left_value] != "null") {
388       datum[fieldName] = datum[it.left_value]
389       return
390     }
391     else if(datum[it.right_value] != "" && datum[it.right_value] != null
392       && datum[it.right_value] != "null") {
```

## 3.2. Implementation

```
389     datum[fieldName] = datum[it.right_value]
390     return
391 }
392 }
393 }
```

This method also allows its second **ParamEntryWrapper** to have repeatable **ParamEntries**. Whenever a valid value within the pseudo-object-map is found, the program saves it in a specified new or already existing map entry and exits the method. A valid value is any value not equal to null, not an empty string or the string “null”. Values are being examined from left to right, thus giving the left value priority over the right value.

**3.2.4.4.12. setTimestamp** assigns the current timestamp into a developer designated object property. The purpose of this method is to give the developer a property that can be used to later filter incorrectly parsed or transformed data from the database, based on the time the data was imported. This can be especially useful when the DIM is used in an automated process. Note that **setTimestamp** is the first transformation method that directly modifies the object instance, which is later loaded into the database. Data is not loaded into the database by this method itself, instead **setTimestamp** has to be used in combination with the appropriate loading method for any changes to be persisted.

left_value	right_value
Property name of routine-defined object in which timestamp is set	NOT IN USE

```
395 static def setTimestamp(TransformationProcedure procedure, Map<String,
396     Object> datum, Object object_instance) {
397     checkIfParamsSetCorrectly("setTimestamp", procedure)
398
399     def params = procedure.parameterWrappers.first().parameters.first()
400
401     if(object_instance.metaClass.getProperties().find{ class_it ->
        class_it.name == params.left_value } == null)
        throw new IncorrectSpecificationException("Can not find class with
            name '" + params.left_value + "'!")
}
```

### 3. Methods

```
402     object_instance[params.left_value] = new Timestamp(new Date().getTime())
403 }
404
```

The timestamp that is placed into the object's property is of the data type "Timestamp"<sup>19</sup>, a characteristic that should to be accounted for by the developer when planning said objects.

**3.2.4.4.13. createRelation** is able to establish a unidirectional relationship between two objects in the form of a foreign key. The parameters for the search query - used to find an applicable object - are composed of the names of an object's properties and keys for the pseudo-object-map values. An object will be fetched from the database if their properties values match the values from the pseudo-object-maps. Note that the pseudo-object-map values must be unique together, i.e. they must always deliver a single object from the database query. A good such value would be an identification number of any kind, or even a string that is a primary key for such object. Another possibility is a combination of properties that, together, always create a compound key that uniquely identifies an object.

The object, in which the foreign key is created, must be the routine-defined object. Otherwise, there is no restriction with what object the relation is established with, as long as the class is specified within the application and present within the database. Only the routine-defined object will know of this unidirectional relation. Further, this transformation method is only able to create one-to-one relations and will throw an exception if it is attempted to create a one-to-many relation. Finally, the developer has to affirm that the routine-defined object has a member variable of the appropriate object class, else the program will raise an exception.

---

<sup>19</sup><https://docs.oracle.com/javase/7/docs/api/java/sql/Timestamp.html> (Last visited: 11.01.2021)

### 3.2. Implementation

left_value	right_value
Property name of routine-defined object in which the relation is persisted	Name of the class of which an instance is found and stored within the routine-defined object
Property name of object that is used for search query	Map key for value that is used to find object instance
:	:

```

406 static def createRelation(TransformationProcedure procedure, Map<String,
    Object> datum, Object object_instance) {
407     checkIfParamsSetCorrectly("createRelation", procedure)
408
409     def params = procedure.parameterWrappers.first().parameters.first()
410     def mapping = procedure.parameterWrappers.asList()[1].parameters
411
412     Class target_class = getClassFromString(params.right_value)
413     StringBuilder propertiesErrorString = new StringBuilder()
414
415     List found_objects = target_class.findAll{
416         mapping.each{
417             eq it.left_value, datum[it.right_value]
418             propertiesErrorString.append(" " + it.left_value + "->" +
                datum[it.right_value] + ";")
419         }
420     }
421
422     if(found_objects.size() > 1){
423         StringBuilder updateCriteriaErrorString = new StringBuilder()
424         updateCriteriaErrorString.append("Problematic object is of type
            '${target_class.name}' with its compared properties being
            values:${propertiesErrorString.toString()}")
425
426         throw new ValidationException("Found multiple objects in database
            which apply for the 'update' criteria! " +
            "Update criteria (or a combination of it) must be unique to each
            object! " + updateCriteriaErrorString.toString())
427     }
428
429     if(object_instance.metaClass.getProperties().find{ class_it ->
        class_it.name == params.left_value} == null)
430

```

### 3. Methods

```
431         throw new IncorrectSpecificationException("Can not find class with  
         name '" + params.left_value + "'!")  
432  
433         object_instance[params.left_value] = found_objects[0]  
434     }
```

The method employs the help of **getClassFromString()** to resolve the required meta-class from the parameter string. This meta-class is then used, with the developer specified parameters, to create a search query via GORM, which searches for an applicable preexisting database object. In case that multiple objects apply, an exception is thrown to notify the user that the query is not specific enough. If the query retrieves not more than one, it is saved in the routine-defined object's specified property. Do note that the query result can comprise null too, i.e. no object was found, resulting in the foreign key being set to null.

**3.2.4.4.14. createOneToManyRelation** is able to create a unidirectional one-to-many relation between the routine-defined object and multiple other objects, for which the search query applies. The parameters for the query are made up of names for the object's properties and keys for the pseudo-object-map values. Each object is retrieved if their properties' values match the values from the pseudo-object-maps. Unlike **createRelation**, this method is lenient by design and throws no exception if multiple objects match the search query criteria, as this is the methods intended use.

The object in question, with which the relation is created, must be the routine-defined object. There is no other restriction on which objects the relation can be established with, as long as the class is specified within the application and present in the database. Only the routine-defined object will know of those unidirectional relations. Finally, the property in which the result will be saved in has to be the GORM related "hasMany" property, which is used to indicate such one-to-many relations.

### 3.2. Implementation

left_value	right_value
Property name of routine-defined object in which the relation is saved	Name of the class of which an instance is found and stored within the routine-defined object
Property name of object that is used for search query	Map key for value that is used to find object instance
:	:

```

436 static def createOneToManyRelation(TransformationProcedure procedure,
437     Map<String, Object> datum, Object object_instance) {
438     checkIfParamsSetCorrectly("createOneToManyRelation", procedure)
439
440     def params = procedure.parameterWrappers.first().parameters.first()
441     def mapping = procedure.parameterWrappers.asList()[1].parameters
442
443     Class target_class = getClassFromString(params.right_value)
444     StringBuilder propertiesErrorString = new StringBuilder()
445
446     List found_objects = target_class.findAll{
447         mapping.each{
448             eq it.left_value, datum[it.right_value]
449             propertiesErrorString.append(" " + it.left_value + "->" +
450                 datum[it.right_value] + ";")
451         }
452     }
453
454     if(object_instance.metaClass.getProperties().find{ class_it ->
455         class_it.name == params.left_value } == null)
456         throw new IncorrectSpecificationException("Can not find class with
457             name '" + params.left_value + "'!")
458
459     object_instance[params.left_value] = found_objects
460 }

```

Here, the method begins by determining a meta-class from the string and building a search query with it. The query is built with the parameters and retrieves all applicable objects. The object list is then saved in the routine object's specified property, concluding this method. Note that this method can deliver an empty list.



### 3. Methods

**3.2.4.4.15. `cacheInfoForCrossProcedure`** is able to cache specific data and create a temporary procedure with the cached data becoming part of the procedures parameters. In other words, this method is used to filter desired data from a file and introduce this data into another datum in whichever form needed. E.g. a file contains all of a companies employees, but there is a spokesperson for each certain employee group who has to be marked down as such for each employee. As of now there are only two such “cross-methods”, that can be temporarily created and injected into the program flow by **`cacheInfoForCrossProcedure`**. These two “cross-methods”, called **`crossSetValue`** and **`crossCreateRelation`**, will be elaborated in the following paragraphs. Setting the “cross-methods” aside, the second **`ParamEntry-Wrapper`** of this method is optional and only used by **`crossCreateRelation`**. In contrast to other methods, this method has configuration restrictions. The first such restriction requires its parent procedure to not be repetitive and further, that the user specifies a datum that is unique - i.e. there is only a single datum per file for which this method is executed. The then executed temporary “cross-method” does not have any of those restrictions and can be either repetitive, non-repetitive, and additionally skip, or filter certain data.

left_value	right_value
Name of temporary transformation method that will be created	Boolean for “is_repetitive” (“true” / “false”)
Key for “notable_objects”	Value for “notable_objects”
⋮	⋮
Left parameter value for temporary procedure	Right parameter value for temporary procedure
⋮	⋮
Property name of routine-defined object	Map key for value that is cached for temporary procedure
⋮	⋮

```
458 static def cacheInfoForCrossProcedure(TransformationProcedure procedure,
459                                     Map<String, Object> datum, Object object_instance) {
    checkIfParamsSetCorrectly("cacheInfoForCrossProcedure", procedure)
```

## 3.2. Implementation

```
460
461 def params_procedure =
    procedure.parameterWrappers.first().parameters.asList()
462
463 def methodname = params_procedure[0].left_value
464 MethodClosure method = TransformationService.&"$methodname"
465
466 def runtime_params = procedure.parameterWrappers.asList()
    [MethodInfo.getWrapperCount(methodname)].parameters.asList()
467 SortedSet<ParamEntryWrapper> new_wraps = new
    TreeSet<ParamEntryWrapper>()
468 ArrayList<ParamEntry> params_for_new_wrap = new ArrayList<ParamEntry>()
469 ParamEntryWrapper new_runtime_param_wrap
470 Map<String, String> notable_objects = [:]
471
472 TransformationRoutine tr = TransformationRoutine.find("from
    TransformationRoutine where ? in elements(procedures)", [procedure])
473
474 if(tr.procedures.temporary.contains(true))
475     throw new ValidationException("'cacheInfoForCrossProcedure' is trying
        to create another temporary procedure even though such already
        exists for this routine. This means the specified data-set is not
        unique!")
476
477 for(int i = 1; i < params_procedure.size(); i++)
478     notable_objects.put(params_procedure[i].left_value,
        params_procedure[i].right_value)
479
480 runtime_params.each{
481     params_for_new_wrap.add(new ParamEntry(it.left_value,
        datum[it.right_value].toString()))
482 }
483
484 params_for_new_wrap.each{it.save(flush: true)}
485
486 new_runtime_param_wrap = new ParamEntryWrapper(params_for_new_wrap)
487 new_runtime_param_wrap.save(flush: true)
488
489 new_wraps.add(new_runtime_param_wrap)
490 if(procedure.parameterWrappers.size() == 3)
491     new_wraps.add(procedure.parameterWrappers.asList()[1])
492
493 TransformationProcedure tp = new TransformationProcedure(order_id: -1,
    transformation_method: method,
```

### 3. Methods

```
494         is_repetitive:
495             Boolean.parseBoolean((params_procedure[0].right_value as
496                 String)), temporary: true, notable_objects: notable_objects)
497         tp.parameterWrappers = new_wraps
498         tp.save(flush: true)
499         tr.procedures.add(tp)
500         tr.save(flush: true)
501
502         if(tr.hasErrors()) {
503             StringBuilder errorString = new StringBuilder()
504             tr.errors.allErrors.each { errorString.append(it.toString() + ";") }
505             throw new DatabaseSaveException("There was an error while loading the
506                 data into the database. Maybe some constraints are not
507                 fulfilled?\n" + errorString.toString())
508         }
509     }
```

This methods implementation is dedicated to creating a temporary procedure and making sure that such is successfully inserted into the current runtime process. It begins by affirming that no other identical temporary procedure exists currently, as this would indicate that the developer specified data properties are not unique. Assuming there exists no such procedure, it then proceeds by creating the procedures **notable\_objects** map, whose values are stored in this method's first wrapper.

Next up are the runtime generated parameters, the defining characteristic which makes this transformation method stand out from the other methods. Those generated parameters are built out of the parameters of the last wrapper. The **ParamEntries'** **left\_values** are just passed on, whereas the **right\_values** are used as keys for the pseudo-object-map. The method passes the optional parameters on to the temporary procedure - if they exist - and ultimately creates the new procedure with all its required properties. Additionally, the unique **temporary** property set to "true", to indicate that this procedure is to be removed after its execution. Note that the **order\_id** is set to "-1", as temporary procedures have priority in their execution, regardless of their **order\_id**. A final check is performed to confirm if the procedure could not be saved within its transformation routine for any reason whatsoever.

Note that the method immediately persists each finished object and does not roll back such if an exception occurs during its execution.

**3.2.4.4.16. crossCreateRelation** creates a one-to-one relation between the routine-defined object and another object; the latter being identified and retrieved from the database via precached data from the **cacheInfoForCrossProcedure** method. Multiple object instances of the routine-defined class, previously created by the transformation block, can have a relation to the same database fetched object. Objects that were previously filtered out by the “notable\_objects” mechanism will not have such a relation created. The **crossCreateRelation** method should not be used on its own, as it is recommended to use the corresponding UI to configure the **cacheInfoForCrossProcedure**. The UI is built around the “cross-methods”, listing all available selections and configurations for such.

Similar to how **createRelation** operates, this method’s object query can also return a null pointer if no matching object is found; thus setting the object’s property in which the relation is created, to null. The parameters for the search query have to be unique, i.e. they do not return more than one object or an exception will be thrown. Finally, only the routine-defined object will know of this relation.

left_value	right_value
Property name of routine-defined object in which the relation is saved	Name of the class of which an instance is found and stored within the routine-defined object
Property name of object that is used for search query	Value for property that is set by <b>cacheInfoForCrossProcedure</b>
⋮	⋮

```

507  static def crossCreateRelation(TransformationProcedure procedure,
    Map<String, Object> datum, Object object_instance) {
508      checkIfParamsSetCorrectly("crossCreateRelation", procedure)
509
510      def params = procedure.parameterWrappers.first().parameters.first()
511      def search_params = procedure.parameterWrappers.first().parameters
512
513      Class target_class = getClassFromString(params.right_value)
514
515      Object instance = target_class.newInstance()
516      StringBuilder propertiesErrorString = new StringBuilder()
517

```

### 3. Methods

```
518 List found_objects = target_class.findAll {
519     search_params.each {
520         if (it.right_value != null && it.right_value != "" &&
521             it.right_value != "null") {
522             eq it.left_value,
523                 parseToCorrespondingType(instance[it.left_value],
524                     it.right_value)
525             propertiesErrorString.append(" " + it.left_value + "->" +
526                 it.right_value + ";")
527         }
528         else
529             new ValidationException("Right value of parameter of method
530                 'crossCreateRelation' has no values set!")
531     }
532 }
533
534 if (found_objects.size() > 1) {
535     StringBuilder updateCriteriaErrorString = new StringBuilder()
536     updateCriteriaErrorString.append("Problematic object is of type
537         '${target_class.name}' with its compared properties being
538         values:${propertiesErrorString.toString()}")
539
540     throw new ValidationException("Found multiple objects in database
541         which apply for the 'search' criteria! " +
542         "Update criteria (or a combination of it) must be unique to each
543         object! " + updateCriteriaErrorString.toString())
544 }
545
546 if(object_instance.metaClass.getProperties().find{ class_it ->
547     class_it.name == params.left_value } == null)
548     throw new IncorrectSpecificationException("Can not find class with
549         name '" + params.left_value + "'!")
550
551 object_instance[params.left_value] = found_objects[0]
```

The **crossCreateRelation** method also makes use of **getClassFromString()** to create a meta-class from the parameter string. Said meta-class is then used, with part of the developer specified parameters and part of the cached data parameters from **cacheInfoForCrossProcedure**, to create a search query, which finds an applicable existing object in the database. The **right values** of the parameters have to be cast back to their corresponding type, which is done via another helper-method and a temporary object instance. An excep-

## 3.2. Implementation

tion is thrown if the database query returns multiple objects, as the method's purpose is to only create a one-to-one relation, not one-to-many. Assuming the query results in exactly one object, it is then saved in the routine-defined object's specified property. A valid query result can comprise null too, i.e. no object was found and thus no relation will be created.

**3.2.4.4.17. crossSetValue** puts pre-cached data into each routine-defined object; all of which was filtered and prepared beforehand by the **cacheInfoForCrossProcedure**. Similar to the **crossCreateRelation** method, the pre-cached datum will not change during the execution and will be copied into each of the objects by default. If desired, objects can again be filtered out beforehand with the "notable\_objects" mechanism, so that the cached data is not stored into such.

This method is also a "cross-method". Like all other "cross-methods", it should not be used on its own, but rather in conjunction with the recommended UI to configure the **cacheInfoForCrossProcedure** method, which in turn prompts the "cross-method" selection with all its available options.

left_value	right_value
Property name of routine-defined object	Value for property that is set by <b>cacheInfoForCrossProcedure</b>

```
543 static def crossSetValue(TransformationProcedure procedure, Map<String,
    Object> datum, Object object_instance) {
544     checkIfParamsSetCorrectly("crossSetValue", procedure)
545
546     def value_map = procedure.parameterWrappers.first().parameters
547
548     value_map.each{
549         if(it.right_value != null && it.right_value != "" && it.right_value
            != "null"){
550             if(object_instance.metaClass.getProperties().find{ class_it ->
                class_it.name == it.left_value } == null)
551                 throw new IncorrectSpecificationException("Can not find class
                    with name '" + it.left_value + "'!")
552
553             object_instance[it.left_value] =
                parseToCorrespondingType(object_instance[it.left_value],
                    it.right_value)
```

### 3. Methods

```
554     }  
555     else  
556         new ValidationException("Right value of parameter of method  
557                                 'crossAddValue' has no values set!")  
558     }
```

Every value that is deemed valid is parsed to its corresponding property's data type, in which it is ultimately saved.

#### 3.2.4.5. Loading methods

As of now, the DIM only provides three loading methods. Loading methods **identityTransfer** and **saveAllObjects** are to be used together, optionally after other transformation methods were executed. The **identityTransferAnd-Save** is a shortcut that can be used if the developer desires to entirely skip the transformation functionality of this ETL module; alternatively it can also be used in together after some transformation methods. There is no loading method that is able to filter which objects are persisted and whatnot, as there are already plenty of tools to filter pseudo-objects and the in parallel created objects. I.e. if the developer does not wish for certain data to be loaded, he has to filter such beforehand in the extraction or transformation block.

**3.2.4.5.1. identityTransfer** loads data from the pseudo-object maps into domain objects that can be persisted into the database. This method is not a loading method per se, as no objects are persisted by it, but is classified as one as it performs the beforehand necessary work so that the domain objects can be properly loaded. Due to this, **identityTransfer** must be used in combination with but before the **saveAllObjects** method, to actually persists the objects.

## 3.2. Implementation

left.value	right.value
Property name of routine-defined object	Map key for value saved into property
:	:

```
560 static def identityTransfer(TransformationProcedure procedure,
    Map<String, Object> datum, Object object_instance) {
561   checkIfParamsSetCorrectly("identityTransfer", procedure)
562
563   def IO_map = procedure.parameterWrappers.first().parameters
564
565   IO_map.each{ mapping ->
566     if(object_instance.metaClass.getProperties().find{ class_it ->
        class_it.name == mapping.left_value} == null)
567       throw new IncorrectSpecificationException("Can not find class with
        name '" + mapping.left_value + "'!")
568
569     if(datum[mapping.right_value] != null)
570       object_instance[mapping.left_value] =
        parseToCorrespondingType(object_instance[mapping.left_value],
        datum[mapping.right_value].toString())
571   }
572 }
```

Here, each valid value is parsed to the data type of the routine-defined object's property, then stored in such.

**3.2.4.5.2. identityTransferAndSave** loads data from the pseudo-object-maps into the domain objects, then tries to persist such into the database. This method can be used on its own and without any previous transformation methods, depending on if the developer only wants to extract data and load it into the database. It can also be used in combination with any previous transformation methods, but has to be the last executed method if all the changes are to be kept.



### 3. Methods

left.value	right.value
Property name of routine-defined object	Map key for value saved into property
:	:

```

574 static def identityTransferAndSave(TransformationProcedure procedure,
    Map<String, Object> datum, Object object_instance) {
575     checkIfParamsSetCorrectly("identityTransferAndSave", procedure)
576
577     def IO_map = procedure.parameterWrappers.first().parameters
578
579     IO_map.each{ mapping ->
580         if(object_instance.metaClass.getProperties().find{ class_it ->
            class_it.name == mapping.left_value} == null)
581             throw new IncorrectSpecificationException("Can not find class with
                name '" + mapping.left_value + "'!")
582
583         if(datum[mapping.right_value] != null)
584             object_instance[mapping.left_value] =
                parseToCorrespondingType(object_instance[mapping.left_value],
                    datum[mapping.right_value].toString())
585     }
586
587     object_instance.save(flush: true)
588
589     if(object_instance.hasErrors()) {
590         StringBuilder errorString = new StringBuilder()
591         object_instance.errors.allErrors.each {
592             errorString.append(it.toString() + ";") }
593         throw new DatabaseSaveException("There was an error while loading the
            data into the database. Maybe some constraints are not
            fulfilled?\n" + errorString.toString())
594     }
595 }

```

Each valid value is parsed to the property's data type, then assigned to said object's property. In case that all values were successfully saved, the method then tries to persist the routine-defined object into the database. In case of failure, an exception is thrown. The flush flag is set here for the "save()" method, meaning objects are persisted immediately, instead of being buffered and then persisted at the end of the program. This has

## 3.2. Implementation

some notable implications. First, if any exception occurs, be it constraint or validation exceptions, it is detected immediately, but all previously successfully persisted objects will not be rolled back and are kept as is in the database. Second, because of the immediate flush, the loading is generally slower than it would be by being buffered and persisted.

**3.2.4.5.3. saveAllObjects** persists each existing routine-defined object, all of which are created in parallel with the pseudo-object-maps, into the database. This method can be used on its own, but will have no effect without calling at least **identityTransfer** beforehand. It is recommended to always use **saveAllObjects** last, so that all changes to the objects are loaded into the database. An exception to this is the other loading method **identityTransferAndSave**, in which case a redundant **saveAllObjects** call can be skipped.

left.value	right.value
NOT IN USE	

```
596 static def saveAllObjects(TransformationProcedure procedure, Map<String,
    Object> datum, Object object_instance) {
597     checkIfParamsSetCorrectly("saveAllObjects", procedure)
598
599     object_instance.save(flush: true)
600
601     if(object_instance.hasErrors()) {
602         StringBuilder errorString = new StringBuilder()
603         object_instance.errors.allErrors.each {
604             errorString.append(it.toString() + ";") }
605         throw new DatabaseSaveException("There was an error while loading the
            data into the database. Maybe some constraints are not
            fulfilled?\n" + errorString.toString())
606     }
607 }
```

Each domain object instance is persisted into the database via GORM's "save()" method. Additionally, the flush flag is used to immediately apply the changes to the database and to detect complications, in case that an

### 3. Methods

object could not be saved for whatever reason. The flush flag means that in case of an exception, like a validation or constraint exception during the save, the previously successfully saved objects would not be rolled back and removed from the database. Further, because there is no buffering for the load, the program will be less performant than without a flush flag.

### 3.2.5. User Interface (UI)

In contrast to the previous chapters, the UI chapter takes a different approach, by not displaying the code of the implementation. Instead, a rough overview is given on how each different UI component functions and which inconsistencies they catch, be it client side or server side. This is done in an attempt to cut down on more bloating of this already large document. Further, the UI does not add any new ETL functionality per se, but instead makes sure that the existing components are configured correctly even with limited understanding of the DIM's inner workings.

#### 3.2.5.1. Extraction UI

Each of the parsers' UIs is based on two basic templates called "parser" and "entries". Those templates are modelled after the base parser class **DynamicParser** and its **DynamicParserEntries**, thus having the same structure. E.g. each **DynamicParserEntry** has a string property called "field", each UI entry likewise has a text field for "field". **DynamicParser** has a string property called "name", parser UI likewise has a text field for "name". Each parser's UI extends the base UI templates, similar to how the parser extends the base parser class.

The following two basic templates add following elements to each parser's UI:

For the parser itself:

- **Parsername** as a text field.
- **Parserdescription** as a text field.
- **Filename substring** as a text field.
- **File type** as a drop-down menu (e.g. txt, xml, json).

For each of the parser's entries:

- **Field name** as a text field (used as a map key).
- **Data type** as a drop-down menu (e.g. Float, String, etc.).
- **Optional** indicator as a checkbox.

### 3. Methods

#### 3.2.5.1.1. ColumnWidthParser UI

The first element added by the **ColumnWidthParser**'s own UI is the “**substring to ignore**” list, which is used to skip lines that contain certain unwanted substrings. These substrings are each displayed as a text fields and can be added repeatedly with the “add substring to ignore” button.

Each entry is now extended by two additional elements consisting of a “Column start” and “Column end”. Those two elements are number fields, requiring their input to be strictly numeral in nature. Both fields are used together to identify the columns for their respective entry.

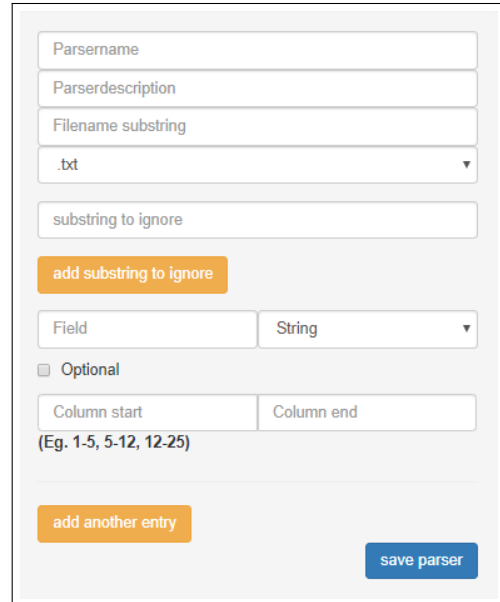
The image shows a web form for configuring a ColumnWidthParser. It includes input fields for 'Parse name', 'Parser description', and 'Filename substring' (with a dropdown menu currently showing '.txt'). Below these is a 'substring to ignore' text field and an 'add substring to ignore' button. Further down is a 'Field' input field, a 'String' dropdown menu, and an 'Optional' checkbox. At the bottom of this section are 'Column start' and 'Column end' input fields, with a note '(Eg. 1-5, 5-12, 12-25)' below them. At the very bottom are 'add another entry' and 'save parser' buttons.

Figure 3.17.: UI of the ColumnWidthParser

#### Client side verification

- **Name** can not be empty.
- **Filename substring** can not be empty.
- Each **field** can not be empty.
- Each **column start** can not be empty, negative and must be a number.
- Each **column end** can not be empty, negative and must be a number.
- Each **column start** has to be smaller than their counterpart **column end**.
- Each **column start** has to be bigger than the previous entry's **column end**.

## 3.2. Implementation

### Server side verification

Note that the controller stops the “substrings to ignore” creation loop the moment an empty or non-existent line appears. I.e. if multiple fields are filled but a single one in between is empty, the lines following the empty line will be skipped.

- No other parser exists that already handles files of the specified **file-name substring** and **file type**.
- At least one **entry** was specified for the parser.
- The **parser** can be loaded into the database (no errors during save).
  - The parser’s **name** can not be null or empty.
  - Each of the parser’s **entries** has a unique **field** property.
- Each **entry** can be loaded into the database (no errors during save).
  - **Field** can not be null or empty.
  - **Data type** can not be null.
  - **Column start and end** can both not be null and have to be  $\geq 1$ .

#### 3.2.5.1.2. TokenSplitParser UI

The **TokenSplitParser**’s UI adds a **separation token** text field, that is required for splitting each line. This text field can be just a single white space.

Similar to the **ColumnWidthParser**, this parser also has a **substrings to ignore** functionality that can be used to skip lines containing certain substrings.

Each entry now has an additional **field index** number field appended, that is used as an index to access one of the split lines.

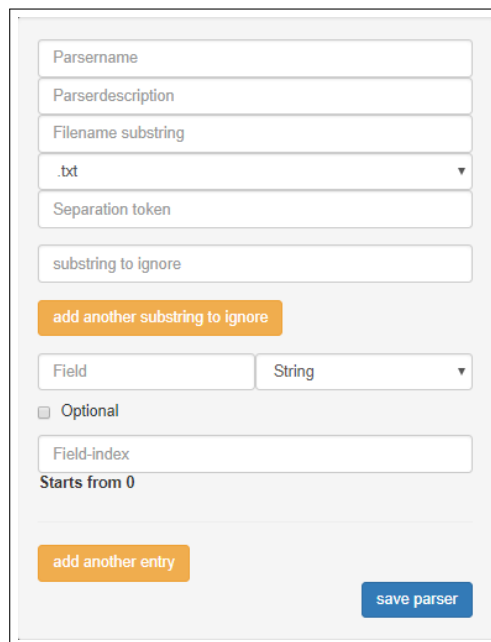
The image shows a web form for configuring a TokenSplitParser. It includes input fields for 'Parse name', 'Parser description', 'Filename substring', 'Separation token', and 'substring to ignore'. There is an orange button 'add another substring to ignore' below the 'substring to ignore' field. Below that is a 'Field' input field and a 'String' dropdown menu. An 'Optional' checkbox is followed by a 'Field-index' input field and the text 'Starts from 0'. At the bottom, there is an orange button 'add another entry' and a blue button 'save parser'.

Figure 3.18.: UI of the TokenSplitParser

### 3. Methods

#### Client side verification

- **Name** can not be empty.
- **Filename substring** can not be empty.
- **Separation token** can not be empty (white space is allowed).
- Each **field** can not be empty.
- Each **field index** can not be empty.

#### Server side verification

Identical to the **ColumnWidthParser** creation logic, the **substrings to ignore** loop in the controller also stops saving these lines the moment an empty or null line appears; regardless if there are further lines in the array.

- No other parser exists that already handles files of the specified **file-name substring** and **file type**.
- At least one **entry** was specified for the parser.
- The **parser** itself can be saved into the database (no errors during save).
  - The parser's **name** can not be null or empty.
  - The parser's **separation token** can not be null or empty.
  - Each of the parser's **entries** has a unique **field** property.
- Each **entry** can be saved into the database (no errors during save).
  - **Field** can not be null or empty.
  - **Data type** can not be null.
  - **Field index** can not be null and has to be  $\geq 0$ .

### 3.2.5.1.3. SimpleTagParser UI

This parser adds a few more new elements to the UI compared to other parsers. There are three elements for the parser itself, which consist of the **domain start tag** and **domain end tag**, two text fields that are used together for detecting when-ever a pseudo-object begins or ends within a file. The **nesting level** is a purely numeral field, also intended to be used with the aforementioned domain tags. Together those three properties can be used in combination or not at all, with the **nesting level** not being strictly required even when using only the domain tags. It is recommended though to fill out these fields if possible.

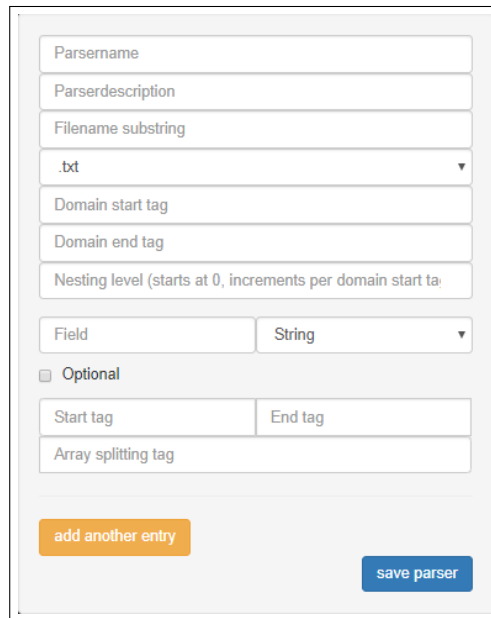


Figure 3.19.: UI of the TokenSplitParser

The entries too have gotten three new elements in total, called **start tag**, **end tag**, and **array splitting tag**. All three of those new entry tags are text fields.

### Client side verification

- **Name** can not be empty.
- **Filename substring** can not be empty.
- **Domain start tag** and **domain end tag** both have to be set, or left empty.
- **Nesting level** can only be set with the domain tags, or not at all.
- Each **field** can not be empty.
- Each **start tag** can not be empty.



### 3. Methods

- Each **array split tag** can only be set in combination with their respective start and end tags or not at all.
- Each **start tag** can not be already covered by a previous more simple **start tag**.

#### Server side verification

- No other parser exists which already handles files of the specified **filename substring** and **file type**.
- At least one **entry** was specified for the parser.
- The **parser** itself can be saved into the database (no errors during save).
  - The parser's **name** can not be null or empty.
  - Each of the parser's **entries** has a unique **field** property
  - The **domain start tag** can not be empty, but can be null.
  - The **domain end tag** has to be set in tandem with the **domain start tag**, or not set at all.
- Each **entry** can be saved into the database (no errors during save).
  - **Field** can not be null or empty.
  - **Data type** can not be null.
  - **Start tag** can not be empty or null.
  - **End tag** can not be empty, but can be null.
  - **Array split tag** can only be set if both **start tag** and **end tag** are also set, otherwise must be null.

## 3.2. Implementation

### 3.2.5.1.4. SimpleXMLParser UI

Four new elements are added to the basic parser UI by the **SimpleXMLParser**. The **super tag** is a text field intended to be used as a pseudo-object identifier in an XML. The **excel tag** is necessary when parsing SpreadsheetML, which encapsulate their data into a “data tag”. Finally, the two numerical fields, both specifying how many unnecessary pseudo-objects are at the beginning or end of the SpreadsheetML file. Unnecessary in the sense that they contain meta-data, not actual data.

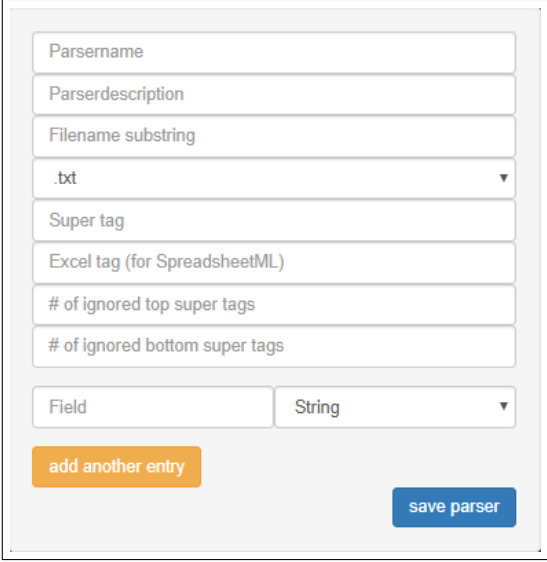
The image shows a web form for configuring a SimpleXMLParser. It contains several input fields: 'Parsername', 'Parserdescription', 'Filename substring', a dropdown menu currently showing '.txt', 'Super tag', 'Excel tag (for SpreadsheetML)', '# of ignored top super tags', and '# of ignored bottom super tags'. Below these is a 'Field' input field and a dropdown menu currently showing 'String'. At the bottom, there are two buttons: an orange 'add another entry' button and a blue 'save parser' button.

Figure 3.20.: UI of the SimpleXMLParser

Otherwise, the **SimpleXMLParser** does not add any new fields to the parser template, on the contrary, it removes the **optional** checkbox. Note that this parser requires its **field** values to be the exact same as their corresponding tags' names in the XML file; this only applies for regular XML, not SpreadsheetML.

### Client side verification

- **Name** can not be empty.
- **Filename substring** can not be empty.
- **Super tag** can not be empty.
- Each **field** can not be empty.

### Server side verification

- No other parser exists which already handles files of the specified **filename substring** and **file type**.

### 3. Methods

- At least one **entry** was specified for the parser.
- The **parser** itself can be saved into the database (no errors during save).
  - The parser's **name** can not be null or empty.
  - Each of the parser's **entries** has a unique **field** property.
  - **Super tag** can not be null or empty.
  - **Excel tag** can not be empty, but can be null.
  - Both **ignored super tag** fields can not be negative (must be  $\geq 0$ ).
- Each **entry** can be saved into the database (no errors during save).
  - **Field** can not be null or empty.
  - **Data type** can not be null.

#### 3.2.5.2. Transformation and loading UI

##### 3.2.5.2.1. TransformationRoutine UI

The UI for the routines is fairly simple, as they only need information on which object class they will applied on and what their parser will be. Illustration for the UI is shown in figure 3.21. Both those required properties are selectable via drop-down menus, limiting possible inputs to sensible options. After a decision was made on a target-object class and parser, the UI will dynamically load in the update properties as displayed in figure 3.22. Here, an **update existing** checkbox can be filled to tell the routine if it should retrieve and update objects of the target class from the database. This is under the assumption that these objects exist, if not, a new instance will be created. If on the other hand the checkbox is not checked, the transformation block will just append the domain objects to the existing table. Note that the **update existing** checkbox has to be used together with the **properties of object to update** selects.

The **update properties** are two drop-down menus, that depend on the previously selected parser and target object's properties. These drop-downs contain options reflecting the previous selections, restricting the user input to sensible options for the context. As many update properties as necessary can be appended to the UI, to ensure a database retrieval of at most a single

## 3.2. Implementation

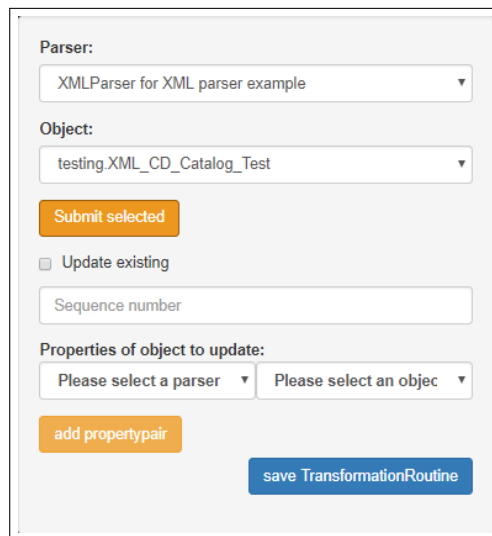
object. It is also acceptable if no object matches the query and the result is null.

The **sequence number** is a numerical field and can be left empty, in which case the controller allocates the first free number, starting from zero, to the routine.



The image shows a web form for creating a new transformation routine. It contains two dropdown menus: 'Parser:' with the placeholder text 'Please select a parser...' and 'Object:' with the placeholder text 'Please select a target object...'. Below these is an orange button labeled 'Submit selected'.

Figure 3.21.: Initial transformation routine UI.



The image shows the same web form after the 'Parser' and 'Object' fields have been populated. The 'Parser:' dropdown now shows 'XMLParser for XML parser example' and the 'Object:' dropdown shows 'testing.XML\_CD\_Catalog\_Test'. An orange 'Submit selected' button is present. Below it is a checkbox labeled 'Update existing' which is currently unchecked. Underneath is a text input field for 'Sequence number'. Further down is a section titled 'Properties of object to update:' containing two dropdown menus: 'Please select a parser' and 'Please select an objec'. Below these is an orange 'add propertypair' button and a blue 'save TransformationRoutine' button.

Figure 3.22.: Transformation routine UI with parser and object class submitted.

### Client side verification

- **Parser** can not be null and has to be selected.
- **Target object** can not be null and has to be selected.
- **Update existing** can not be checked if no update properties were selected.

### Server side verification

- **Sequence number** has to be unique for all routines of this parser.

### 3. Methods

- **Update property key** has to be a valid object property.
- **Update property value** has to be a valid parser map entry.
- **Update properties** have to be set if the **update existing** checkbox was set.
- **Transformation routine** was able to be saved (no errors during save).
  - **Target object** can not be null.
  - **Update properties** can not be empty if **update existing** is true.
- The **parser** and its new relation with the **routine** was successfully persisted.

#### 3.2.5.2.2. TransformationProcedure UI

Similar to the routines, the UI for the procedures is also in need of some information before the rest of the UI is dynamically loaded in. Procedures need different parameters, depending on which routine they are part of and what their transformation/loading method will be. The routine specifies the domain class the procedure will work with, whereas the transformation/loading method gives info on what parameters are required. See figure 3.23 for the initial UI.

Depending on the submitted information, the UI can extend considerably. The example shown in figure 3.24 is the most complex variant. Note how in this illustration a new field, called “Crossmethod”, was inserted. Cross methods are only displayed when selecting the particular “cacheInfoFor-CrossProcedure” transformation method, as they are methods created at runtime by it. All of these elements are drop-down menus and restrict the user input to reasonable options.

The dynamically loaded part of the UI consists of the **sequence number**, the **repetitive procedure** checkbox, the **notable parsing objects**, and the method parameters, with latter being polymorphic in nature. The example shown in 3.24 is rather peculiar, as the method parameters contain another transformation procedure definition.

### 3.2. Implementation

**Notable parsing objects** are used to filter, or skip, parser created pseudo-object maps. The keys of the **notable parsing objects** are drop-down menus of the aforementioned map entries, whereas the values are text fields with no restriction on what the user puts in.

Last are the multiform transformation method parameters. These can be simple text fields, drop-downs for the parser generated map keys, drop-downs for object properties, or drop-downs for object classes, depending on the transformation method and its required parameters. A hidden enumeration, called **MethodInfo**, helps building the UI according to the selected procedures, restricting user input to sensible options and preventing any inconsistencies. This enumeration will not be elaborated on further as it just consists of hard coded maps and helper methods, but can be viewed in the code's repository if desired. The UI also automatically adds any method-created map entries to the list of such, resulting in those new map entries also being displayed in the respective drop-downs. An example for that can be found in the [transformation method addendum](#).

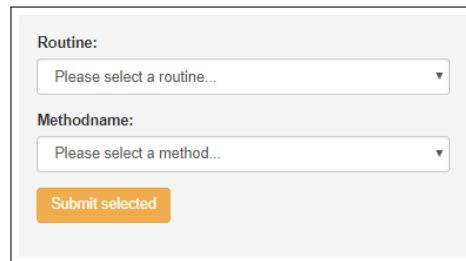
The image shows a web form titled 'Initial transformation procedure UI'. It contains two dropdown menus: 'Routine:' with the placeholder text 'Please select a routine...' and 'Methodname:' with the placeholder text 'Please select a method...'. Below these is an orange button labeled 'Submit selected'.

Figure 3.23.: Initial transformation procedure UI.

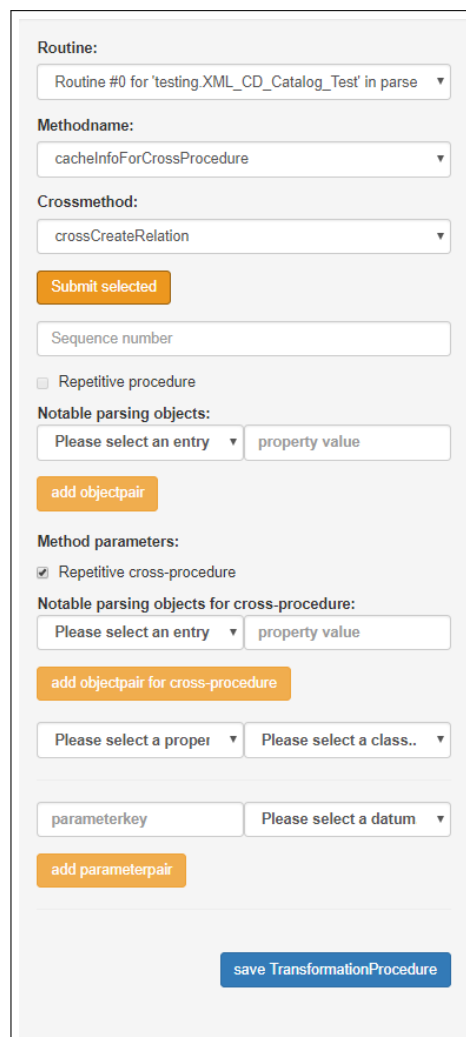
The image shows a more complex web form titled 'Transformation procedure UI with the routine and method submitted'. It contains several sections: 'Routine:' with a dropdown showing 'Routine #0 for 'testing.XML\_CD\_Catalog\_Test' in parse'; 'Methodname:' with a dropdown showing 'cacheInfoForCrossProcedure'; 'Crossmethod:' with a dropdown showing 'crossCreateRelation'; an orange button 'Submit selected'; a text field 'Sequence number'; a checkbox 'Repetitive procedure'; 'Notable parsing objects:' with a dropdown 'Please select an entry' and a text field 'property value', followed by an orange button 'add objectpair'; 'Method parameters:' with a checked checkbox 'Repetitive cross-procedure'; 'Notable parsing objects for cross-procedure:' with a dropdown 'Please select an entry' and a text field 'property value', followed by an orange button 'add objectpair for cross-procedure'; two dropdowns 'Please select a proper' and 'Please select a class..'; a text field 'parameterkey' and a dropdown 'Please select a datum'; an orange button 'add parameterpair'; and a blue button 'save TransformationProcedure' at the bottom.

Figure 3.24.: Transformation procedure UI with the routine and method submitted.

### 3. Methods

#### Client side verification

The client side verification of the transformation procedure is minimal, as most inputs are drop-downs that only display reasonable options.

- **Routine** can not be null and has to be selected.
- **Transformation method** can not be null and has to be selected.
- If **transformation method** is “cacheInfoForProcedure”, then **Cross-method** can not be null and has to be selected.

#### Server side verification

- **Sequence number** has to be unique for all procedures of this routine.
- **Notable parsing objects** must be set for procedure, if it is specified as **non-repetitive**.
- **Notable parsing objects for cross-procedure** must be set, if **crossprocedure** is specified as **non-repetitive**.
- All **parameters** and their **wrappers** were saved successfully (no errors during save).
  - The **right value** can be empty or null, as long as the **left value** is not also null.
- There was at least one **parameter** specified, if required by the method.
- **Procedure** was persisted successfully.
- **Routine** and its relation to the new **procedure** was persisted successfully.

For more information on parameter wrappers, visit the [procedure section](#) in the transformation chapter.

## 4. Discussion

As a conclusion for the main document, the discussion focuses on the two less positive aspects of the DIM. The first section, limitations, deals with the question: “What is the DIM not able to do?”, elaborating on how the design decisions or current implementation limit the DIM in certain areas. On the other hand the second section, future work, expands the previous question by with a futuristic perspective: “What could the DIM do in the future that it is not able to do now?”. Here, a multitude of potential ideas are listed, all of which were brought up during the development process, but were ultimately dropped, as they were not feasible for the initial release.

### 4.1. Limitations

#### 4.1.1. General

Every time an exception occurs during a process of the DIM, be it during a parse, transformation, or loading, the program stops and prints the stack trace into the standard output. Additionally a small error message and HTTP status code are returned by the parsing function, instead of the usual success code and message. The implication here for the parsing process is that if files are parsed and processed automatically, during a reoccurring job or similar, the moment a file is unparsable, the DIM will try to parse the same file in the same way every time, generating the same exception. There are two options to dealing with any exceptions occurring during the parsing process. Either the overlying program handles the files - which are causing the exceptions - on its own. The second approach requires the user themselves to be on stand-by and to either tune the parser, so no exception



#### 4. Discussion

occurs, or to just remove the file causing the exception.

The same applies for the transformation process. Any exception thrown during a transformation, results in it ultimately being caught in the overlying parsing process, halting the parsing/transformation process. This exceptional program ending is accompanied with a print of the stack trace, a returned error message and a HTTP status code. The remedy here is similar to the parsing exception's, in that the problem causing files have to be moved from the parsing folder, but the transformation routine and procedure have to be tuned, instead of the parser, so that the transformation can be applied successfully.

More problematic are exceptions occurring during the load. Apart from automatically moving the file or manually fixing the loading procedure when an exception occurs, the database also has to be validated. The dilemma here is that any successfully loaded data is not rolled back when an exception occurs. I.e. anything persisted remains persisted, reparsing and loading the same file multiple times to reproduce a bug will result in redundant data being persisted each time.

The only proper way to configure a parser, transformation, and loading program, is via the UI, a tedious process that is prone to mistakes by user input. As of now there is no safe way to extend, edit, or even delete said configurations, except by manipulating the database into which they are persisted. This requires the manipulator to be experienced with the DIM and its inner working.

Even though the DIM was designed with automation in mind, it is not optimized for bulk data import. The DIM was only tested for the import of a couple of files at times and never in combination with a huge amount of files or large files in general. Employing the DIM in such a case should be possible, but will most likely not be the solution to time critical applications, at least not without further optimization to the code.

Every parser, every transformation, and every loading procedure has to be deterministic. An exception is thrown whenever a situation arises in which multiple options are valid. There is no mechanism built into the DIM that allows the user to select the correct option of all available options at runtime, or even an option to tell the DIM to just select the first choice. This

## 4.1. Limitations

is intended by design, but an argument can be made to at least include the latter mentioned configuration option.

### 4.1.2. Extraction

As of now, all parsers are implemented without any specialized third party libraries. Whereas the XML parser is using the Java API for XML processing, the other parsers are built with standard Java and Groovy functionality. With the exception of the antithetically named **SimpleTagParser**, all parsers are rather simple in execution and built to parse basic text-based files, which is fine for most use cases, like fixed width or comma separated files. Problems can arise for more complex or niche files, like more complicated JSON or XML files.

Files are only differentiated based on their name and the filename extension. Parsers do not use any meta data contained within files for discernment, meaning the expectation lies on the user to provide files with distinct names.

The **SimpleXMLParser** and the **SimpleTagParser** are both only able to parse a single type of object, which was defined in their configuration. As of now, these two parsers are not able to parse multiple other objects further nested within the original objects. There is an exception though, it is possible to parse sub-objects within objects, if every such instance occurs only once within each object and if their content tags are unique in respect to the parent object and to other sub-objects within the parent object. An exemplary JSON file with a parsable sub-object can be seen [here](#). Note that the sub-object “GlossDef” itself is not acknowledged in the parser’s configuration and its content tags are parsed as if they were properties of the original parent object. A JSON file containing an array of sub-objects, whose content is only parsable as a list of strings, can be found in the [addendum](#). In theory, a file could be parsed multiple times, with different parsers, to properly parse its sub objects, with their relations being added during the transformation block. Although this solution requires the concerning files to be renamed after a parse is over, to enable the next parser to only select the already parsed files for the next run. This renaming is required, as

## 4. Discussion

the DIM does not allow different parsers to parse the same files. The DIM does not have a file renaming feature, meaning logic outside of the DIM is necessary.

### 4.1.2.1. SimpleTagParser

As described in the paragraph before, the **SimpleTagParser** is only able to parse the properties of sub-objects nested within the actual object, when certain conditions are met. The problem herein lies, if no “nestinglevel” is specified and the first property to be parsed is within such a sub-object, then the parser will only detect all other values within this depth. This issue arises because the parser will set the “nestinglevel” to the depth at which the first valid value to be parsed occurs, if it was not set by the user, which is why a “nestinglevel” should always be specified whenever there are sub-objects that can be parsed.

The **SimpleTagParser** is able to parse single-lined files, but this is generally not recommended. If the file is a JSON file, the parser prettifies the file before attempting a parse, which is also not recommended. During a prettify, the order of the properties might be reordered, messing up the current configuration of the parser and resulting in a bad parse. E.g. if an “endtag” is specified with a “,” for a value, the comma might be missing in the prettified JSON if the value was rearranged to the end of the object. Thus, it is recommended to prettify the JSON files beforehand and then configure the parser accordingly.

There is no mechanism differentiating between multiple tags of the same name. If an object contains multiple such tags, only the value of the last occurrence is kept. Lastly, the parser is not able to distinguish multiple “domaintags” per line, with the exception being the single-lined files parsing mode.

### 4.1.2.2. SimpleXMLParser

The **SimpleXMLParser** expects every named property tag from the parser’s configuration to occur in every object, even if the tags have no content. If

## 4.1. Limitations

the property tag is missing from an object, an exception will be thrown.

Figure 4.1 on the right shows an example in which each pseudo object of the “widget” root-tag has a different name and differently named value tags. The **SimpleXMLParser** only allows for the definition of a single supertag, with the options in this case being either “window”, “image”, or “text”. This is an unsatisfactory configuration, because no matter which of the pseudo object is named as a supertag, a sizeable part of the information is lost. A different approach would be to specify the “widget” root-tag for the supertag and make use of the parser’s lax behaviour, which is able to parse further nested values. This way, more of the value tags can be parsed, but problems arise with the identically named value tags “name”, “hOffset”, “vOffset”, and “alignment”. In those cases only one of these identical tags each can be parsed.

```
<widget>
  <debug>on</debug>
  <window title="Sample Konfabulator Widget">
    <name>main_window</name>
    <width>500</width>
    <height>500</height>
  </window>
  <image src="Images/Sun.png" name="sun1">
    <hOffset>250</hOffset>
    <vOffset>250</vOffset>
    <alignment>center</alignment>
  </image>
  <text data="Click Here" size="36" style="bold">
    <name>text1</name>
    <hOffset>250</hOffset>
    <vOffset>100</vOffset>
    <alignment>center</alignment>
    <onMouseUp>
      sun1.opacity = (sun1.opacity / 100) * 90;
    </onMouseUp>
  </text>
</widget>
```

Figure 4.1.: Problematic example from [json.org](https://json.org/example.html).<sup>1</sup>

The takeaway here is that the **SimpleXMLParser** only allows for the configuration of a single pseudo-object, restricting parsable XML files to a simple repetitive format.

### 4.1.3. User Interface

Many elements of the different UIs’ taglibs have identical IDs or names, meaning taglibs for different parsers can not be mixed within the same page without breaking controller and client logic.

As of now there is no easy-to-use API and the UI is the only proper method to define parsers, transformation, and loading routines in a controlled

---

<sup>1</sup><https://json.org/example.html> (Last visited: 11.01.2021)

## 4. Discussion

environment. The drawback is that while the UI is easy to use, it can be very tedious to put in the configuration of more complex ETL logic. If an input was made in error, the go-to way of fixing a configuration is by either manually editing the database itself, or by writing a program with access to GORM.

## 4.2. Future Work

### 4.2.1. General

Many of the tag-oriented parsers, like the **SimpleXMLParser** or the **SimpleTagParser**, are only able to parse a single configured object within those files. They are, with some exceptions, not able to parse sub-objects nested within the original object, much less able to properly store the sub-objects in separate tables and create relations. Further, these parsers do not allow for the configuration of multiple different objects per file and the DIM itself does not allow for multiple parsers to be applicable for a file, meaning more complex files, with multiple objects containing data, can not be properly parsed. Allowing for a more dynamic configuration, in which multiple objects and sub-objects can be specified per parser, would remedy such.

Similarly to how the **SimpleXMLParser** and the **SimpleTagParser** are not able to parse sub-objects, their ability to parse lists or arrays is also minimal. Whereas the **SimpleTagParser** has a limited ability to parse simple arrays, the **SimpleXMLParser** does not possess such an ability currently.

The DIM does not offer any mechanisms to help the user, if data was imported in an undesired way. In such a case, the user has to manually amend the database and filter out the wrongly imported data themselves. A quick solution would be to let the DIM manage a table, containing information on which table was appended with what data at what time; basically just timestamping each object in a separate table to allow for quick filtering by time.

The DIM was never properly tested and optimized for a huge number of files or huge files in general, which is antithetical to its intended use as

## 4.2. Future Work

an automation tool. Depending on the severeness of the performance loss caused by the DIM under real-world conditions, optimization work might turn out to be a priority.

Each datum persisted into the database is flushed immediately from the Hibernate session. Flushing has an effect on performance, as the program has to wait during each save for the storage medium to complete the database transfer. Buffering data and persisting it as batches would improve performance.

Implementing unit, integration, and functional tests<sup>2</sup> would ensure that the DIM stays functionally the same with future updates/refactors, also guaranteeing backwards compatibility, aside from different Grails and Java versions.

### 4.2.2. Extraction

When thinking of future work for an ETL module, what often comes to mind is more document types the module should be able to parse. A ubiquitous but very diverse file format is PDF. PDF can contain data in any form or structure and might be readable by machine or just by human, making it a file format that can be hard to interpret. Depending on the file, it might contain data as a text, or encoded as figures. In the latter case, a dedicated OCR software library is needed, inflating the cost of such a parser implementation. There is well-known OCR software out there, but it is not infallible and might produce flawed results, depending on the file it is applied on. Lastly, PDF files can contain data in any manner, possibly not structured at all; all in all making the PDF format difficult to uniformly process and store, but nonetheless a possible future prospect.

Integrating OCR libraries would allow to parse files that are usually not readable by machine, including the previously mentioned PDF, to Word format. Possibly even scanned documents resulting in picture formats like JPEG and PNG. A combination of OCR and OMR libraries would even allow for automatic parsing and evaluation of checklists.

---

<sup>2</sup><https://grails.github.io/grails2-doc/2.2.4/guide/testing.html> (Last visited: 11.01.2021)

## 4. Discussion

The excel file format is also a widely used format for the easy storage and manipulation of data. These files are similar in structure to CSV files and thus easy to iterate over, making them a possible candidate for a future parser implementation. Problems arise when interpreting their cells' contents as they can be non-intuitive at first sight, with a cells possibly containing formulas or some other unconventional types.

Most files, parsable by the DIM, have formats that are well defined and easy to interpret. This includes files like XML variations, CSV, fixed-width, and JSON files. In theory, these files could be analyzed automatically by the DIM and all-encompassing parser configurations could be generated. All work left over for the user to do, would be to specify the filename sub-string, for which the self-generated parsers would apply. To cut down on processing time, options might be given the user to allow for the removal of unnecessarily parsed data from the parsers' configurations, so that only the desired data is parsed. For XML, the automatic detection would need to consider any data type meta-data, if available. For the tabular formats, like CSV and fixed-width files, the table headers have to be considered.

### 4.2.2.1. TokenSplitParser

The **TokenSplitParser** is a general purpose parser, that is able to parse CSV files, although not as proficient as a dedicated third party library. Proper usage of such a library, like for example the Apache Commons CSV<sup>3</sup>, would fix some of the parser's lacking features. E.g. the parser is not able to consider quote surrounded commas nested within values and instead erroneously splits up such values too. Alternatively, these missing features could be implemented themselves into the parser, as they are straightforward.

### 4.2.2.2. SimpleTagParser

Files consisting of a single line are, as of now, parsable by the **Simple-TagParser**. This feature contains many inconsistencies and is generally not

---

<sup>3</sup><https://commons.apache.org/proper/commons-csv/> (Last visited: 11.01.2021)

## 4.2. Future Work

recommended to use. Future work here would encompass replacing this feature with logic that properly splits or prettifies such single-lined files, into multi-lined and well-defined files, allowing for a proper parse by the base functionality. This feature would have to be able to make a distinction between different file types, e.g. handling JSON differently than, say, YAML.

JSON files are handled by the general purpose **SimpleTagParser**. They can be sufficiently parsed, but the parser itself does not follow the official JSON specifications and is therefore not able to fully parse more complex files. E.g. arrays are only partially parsable, objects nested within the original objects can only be evaluated partially and so on. Here, the implementation of a standalone JSON parser, with help of third party libraries, would remedy this limitation and allow for apt parsing of all JSON files, regardless of complexity. A proper implementation of a JSON parser means the UI, for the JSON parser, would also need to account for the added flexibility it would bring. Similarly, the transformation logic block would need adaptations to properly process different objects with variable content, fed back from a single parser.

### 4.2.2.3. SimpleXMLParser

XML files can be arbitrarily complex, with the **SimpleXMLParser** being designed for more simple variants. XML files can contain infinitely nested objects, with their tags' attributes also containing information, all of which can not be parsed as of now. The **SimpleXMLParser** can only partially parse the values of sub-objects, can not parse attributes at all, and is not able to parse more than one specified object type. All features that are a necessity for a proper dynamic XML parser. Future work here would encompass the option to define as many objects as necessary, with as much content as necessary, regardless if nested within other objects and with consideration how often these objects appear. Further, the ability to define if values are to be read from a tag's content itself, or from one or more the attributes. Additionally, the UI and the transformation block would both need adaptations to account for the additional list of attributes and the multiple differently structured objects, now returned by a single parser.



## 4. Discussion

### 4.2.3. Transformation

Certain transformation methods retrieve objects from the database and use them to create relations with routine-defined objects. This retrieval is based on a simple equality comparison that is examining if strings are identical in content. Here, the implementation of an optional “contains” check - whether a string contains a subsequence equal to another string - would allow for more flexibility in the selection of such objects.

Whenever certain transformation methods apply data manipulation, the underlying datum is usually converted to string and - depending on if the method is numerical in nature - possibly parsed back from string into a float for a calculation. Most transformation methods handle/convert the data as/into strings, for simplicity's sake, even though all parsed data is stored as “objects” internally and could be anything from float to string. One of the implications here is that information can be lost when parsing the data from and back into string, a problem exacerbated by floating point data types. Proper data type handling would avert any inconsistencies arising because of such.

A more ambiguous but nonetheless ubiquitous data type is the date<sup>4</sup> format. As of now, the DIM is not able to handle the date format or any of its variations like e.g. the gregorian calendar<sup>5</sup>, much less able to alter date data. The only option to store and process dates is via the string format, an improper solution if true date-time functionality is needed. There are two possible approaches where a date functionality could be implemented. On one hand, whenever a parsers extracts a date value from a file, the value could then be converted into the date format and passed on to the transformation block. This requires a proper data type handling in the transformation block, as described in the previous paragraph. On the other hand, the date values extracted by the parsers could be kept as is and passed on to the transformation block as strings, with an optional transformation method then converting the desired values into the date format before saving them in the routine-defined objects.

---

<sup>4</sup><https://docs.oracle.com/javase/7/docs/api/java/util/Date.html> (Last visited: 11.01.2021)

<sup>5</sup><https://docs.oracle.com/javase/7/docs/api/java/util/GregorianCalendar.html> (Last visited: 11.01.2021)

Lastly, the transformation block offers a repertoire of transformation methods that should be sufficient for basic data-management needs, but will be shorthanded for more complex demands. New transformation methods can be easily added to the transformation service and only require some knowledge on how the data is stored. Additionally, meta-information for the UI has to be extended, if need be.

### 4.2.4. User Interface

Parsers, transformation routines, and transformation procedures can not be deleted or edited via the DIM. These actions must be performed manually in the database itself, which requires in-depth knowledge and can be cumbersome. A proper UI implementation of edit/delete for each such domain object would be a convenient addition to the DIM.

Similarly, the DIM will return an error message whenever an attempt is made to create a parser for which another parser exists, that handles the same files. Here, the user has to manually edit the database and drop the already existing parser beforehand. A possible solution to this dilemma is to drop the previous parser and all its related objects, with previous explicit confirmation from the user. This can also be indirectly fixed with the edit functionality described in the previous paragraph. The same “drop similar previous” feature might be nice to have for transformation routines and procedures, but raises some questions. Unlike parsers, there is no creation restriction or criteria on when routines and procedures can be considered similar. Further, routines and procedures can simply be created anew whenever, if previous configurations are deemed insufficient.

Identifiers provided for each value during a parser’s configuration are used as string keys for the values in a map. This means that if the user puts in the same identifier for multiple values, only the last value will be stored in the map under the identification. To solve this issue, the implementation of a collision detection before persistence would be helpful.

The UI often employs “select” elements for the method parameters, depending on what transformation method is used. These selects can either show a domain object’s properties, or map keys resulting from a file’s parse.

#### 4. Discussion

Nevertheless, the UI does not give information on what a select is depicting, which can be confusing without any clues.

No UI element has any styles or classes set by default. There is Javascript functionality provided by the DIM, that will set all elements' classes for the Bootstrap 3 framework, if imported to the page by the user. This Javascript solution is inelegant, as it is inefficient at setting all elements' classes, in addition to containing selectors for all elements of all UIs. A better approach would be to give all taglibs an option that, if specified, renders the elements accordingly with their default Bootstrap 3 CSS classes.

## appendix



# Appendix A.

## Getting started

The original code can be found in its git repository [here](#).

There is a SQL file that can be used to remove all DIM related tables created by it, which can be found [here](#).

Another SQL query that helps visualizing the relation between all routines, procedures and their parameters can be found [here](#).

### A.1. How to employ the DIM

Each of the DIM's components can be configured via their own UI, of which each makes sure that the user can not submit any potential inconsistencies. The DIM provides tag libraries for each of the components, allowing for an easy integration into already existing web applications. Following is a list of all the DIM's components and how their respective tag libraries are used:

- **Extraction:**

- ColumnWidthParser → `<dim:createColumnWidthParser/>`
- TokenSplitParser → `<dim:createTokenSplitParser/>`
- SimpleTagParser → `<dim:createSimpleTagParser/>`
- SimpleXMLParser → `<dim:createSimpleXMLParser/>`

- **Transformation/Loading:**

- TransformationRoutine → `<dim:createTransformationRoutine/>`

## Appendix A. Getting started

- TransformationProcedure → <dim:createTransformationProcedure/>

```
<?> page contentType="text/html; charset=UTF-8" ?>
<html>
  <head>
    <title>create CWP</title>
    <link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
          integrity="sha384-BVYiisSiFeKldGmJRAkyCuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u"
          rel="stylesheet" crossorigin="anonymous">
  </head>
  <body style="...">
    <div class="col-lg-3 center-block">
      <dim:createColumnWidthParser />
    </div>
  </body>
  <g:javascript src="setBootstrapClasses.js"/>
</html>
```

Figure A.1.: Example of the ColumnWidthParser’s tag library employed in a simple page used for testing.

To get the Bootstrap 3 look, it is necessary to import the Bootstrap 3.3<sup>1</sup> CSS and the “setBootstrapClasses” Javascript into the page, latter assigns all corresponding Bootstrap CSS classes to their DIM related elements. A showcase for using bootstrap with the DIM is shown in figure A.1.

Feel free to copy and adapt the Javascript if another CSS framework or style is preferred.

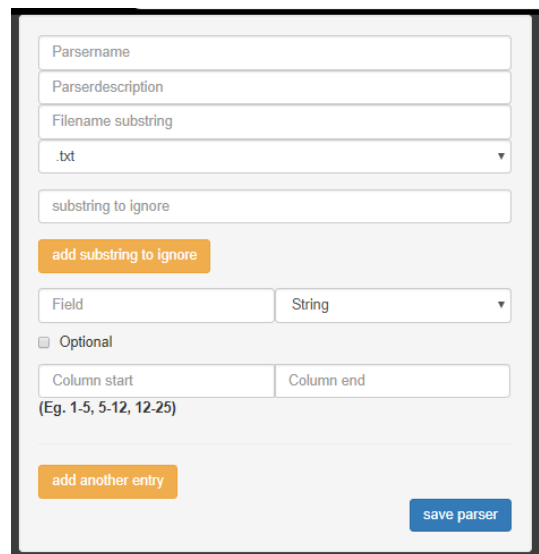


Figure A.2.: The rendered tag library in the resulting page.

<sup>1</sup><https://getbootstrap.com/docs/3.3/> (Last visited: 11.01.2021)

### A.1. How to employ the DIM

After all the desired parsers, transformation routines and procedures were configured accordingly and loaded into the database, all that is left is to begin the DIM's overarching execution program. There are two approaches on how to boot up the DIM. The first method involves using the file selection tag library and manually putting in the directory path for the folders, each time the DIM should be started. The second method can be achieved by directly calling the **FileParsingService**'s main method with the file path, possibly in an automated environment.

For the former variant, make use of the tag library called: `<dim:fileSelection/>`. The taglib's rendering results in a simple text field and button, which should look akin to this, if the CSS is enabled:



To start the parsing and transformation process manually, insert the folder path - where the files for the parse currently reside - and push the button.

Alternatively, it is possible to directly call the DIM's **parseAllFilesInDirectory()** method itself to launch the program with it. This method requires the directory file, in which the files reside, as an argument.



## Appendix B.

### Use cases and examples

Use cases presented in this chapter aim to give a quick and rough overview on how to use the different ETL aspects of this module. Snapshots can be seen displaying examples for sophisticated files, some corner cases, and how to configure the DIM for each whilst also showcasing the basics. The majority of each subchapter will be made up of figures, illustrating what the developer might encounter themselves during a use case. Although sparse, text will be present if there are situations that are not clearly evident at first sight. Note that the all the figures were made with the UI using Bootstrap 3.3 as a CSS framework, with the suggested Javascript setting the classes. The UI might look different depending on the application employing the DIM.

#### B.1. Extraction examples

Each parser configuration example consists of three figures. A snippet of the file that is parsed, the corresponding configuration needed, and a snippet of the delivered result, the generated pseudo-object maps. The complete configuration depicted might be a bit confusing, as there are no labels and all the placeholders that give information are replaced by the actual values. Thus it is recommended to integrate and run the plugin's UI beforehand to have a rough idea of what values each parser's configuration needs.

### B.1.1. Fixed-width file example (handled by the ColumnWidthParser)

03/04/2013										Page 12
Period 01 Thru 03										
4:16 pm										
Company 200										
Entry	Per.	Post Date	GL Account	Description	Src.	Cflow	Ref.	Post	Debit	Credit Alloc.
281241	03	02/18/2013	5958787235	VJX VYRCHRJ RXB TY TP	S1	Yes	TOCG	Yes	43.45	Yes
284329	03	02/19/2013	2584327630	TKQ JXP ISKV CEIF	S1	Yes	FWMA	Yes	1.89	No
281604	03	02/18/2013	5475117985	QFR RPN KSECD QJRA	S1	Yes	ZTHE	Yes		66,233.44
278436	03	02/15/2013	2226484431	MSJ NBZ NIXS	S1	Yes	BMCN	Yes	97,818.54	Yes
281239	03	02/18/2013	5661794541	LJB XE PQ USKWBJT MNE UF	S1	Yes	EBVO	Yes		45,379.28
281603	03	02/18/2013	3394861553	GVQ AEP ZKFLY GOAO	S1	Yes	IIBY	Yes	92,299.81	Yes
284331	03	02/19/2013	5660526806	FDN QLT CYTM GSVJ	S1	Yes	SRJF	Yes	12,298.17	Yes
261098	03	02/08/2013	5649580335	BMS FFB	S1	Yes		No		72,855.65
261096	03	02/08/2013	4732911590	DHV UYQ	S1	Yes		Yes		618.88
294832	03	02/25/2013	2068136558	QLGZIUJ PY WTU SZL	S1	No	PJLT	Yes		63,683.68
ENDING BALANCE PERIOD 03									21,739,353.22	12,804,234.52
BALANCE									33,417,988.84	176,367,802.34
End of Report										

Figure B.1.: Example fixed-width file called “FixedWidthExample2.prn”, with only the end of the file being displayed. Source is [dailydoseofexcel.com](http://dailydoseofexcel.com)<sup>1</sup>.

The example shown in the figure above is the only fixed-width file shown in this chapter as it showcases most features of the **ColumnWidthParser**. This file in particular has a couple of properties that have to be considered by the DIM:

- A line that only contains a timestamp. This line is filtered via the “:” character, which is risky but works for this specific file format, as this character only occurs here. Using “pm” would be dangerous, as this could change to “am” in another file.
- Two of the later lines only have values in the “Description”, “Debit”, and “Credit” columns, which is why all other lines have to be optional.

<sup>1</sup><http://dailydoseofexcel.com/archives/2013/04/12/sample-fixed-width-text-file/> (Last visited: 11.01.2021)

## Appendix B. Use cases and examples

- The “Alloc.” field has another requirement on why it has to be optional, even though strings can be empty by default. This is because the **ColumnWidthParser** throws an exception if a non-optional field is missing during the parsing process. In the example file, the lines are sometimes cut short and end before the “alloc” column would be parsed. I.e. the line width is smaller than the specified range for the “alloc” column, resulting in an exception when non-optional.
- The end of this particular file has a line indicating its end, which can easily be overlooked.
- Finally, “Debit” and “Credit” both use a representation that indicates large numbers with commas, which is not supported by the floats “parseFloat()” method, thus requiring to be saved as a string and transformed via [other means](#) later in the process if a float is required.

The corresponding configuration can be found on the following page.

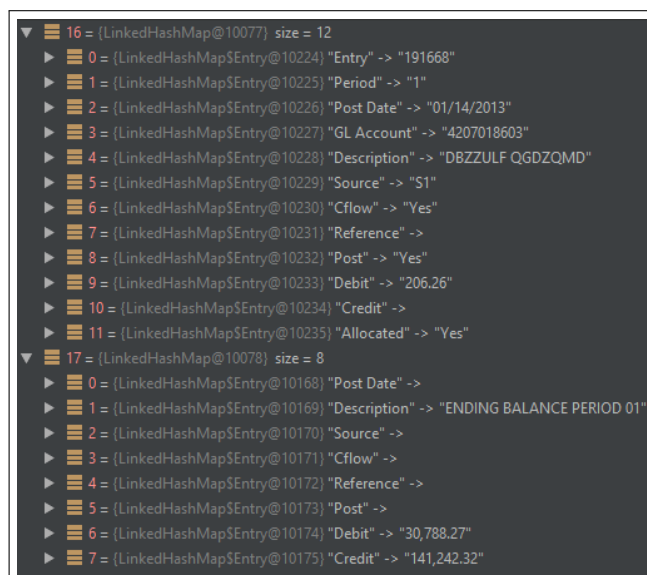


Figure B.2.: The resulting pseudo-object maps of the “FixedWidthExample2.prn” file, with two such objects on display.

## B.1. Extraction examples

Example parser for a fixed-width-file		Description	String
Fixed-width file parser configuration example		<input type="checkbox"/> Optional	
FixedWidthExample		38	64
.pm			
Page	Source	String	
Period	<input checked="" type="checkbox"/> Optional		
:	64	69	
Company			
Entry	Cflow	String	
-----	<input checked="" type="checkbox"/> Optional		
=====	69	74	
End of Report			
add substring to ignore		Reference	String
Entry	Integer	<input checked="" type="checkbox"/> Optional	
<input checked="" type="checkbox"/> Optional	74	83	
1	9		
(Eg. 1-5, 5-12, 12-25)			
Period	Integer	<input checked="" type="checkbox"/> Optional	
<input checked="" type="checkbox"/> Optional	83	87	
9	13		
Post Date	String	<input checked="" type="checkbox"/> Optional	
<input checked="" type="checkbox"/> Optional	87	107	
13	25		
GL Account	Long	<input checked="" type="checkbox"/> Optional	
<input checked="" type="checkbox"/> Optional	107	125	
25	38		
Allocated	String	<input checked="" type="checkbox"/> Optional	
<input checked="" type="checkbox"/> Optional	126	133	

Figure B.3.: A collage of a functioning configuration of the ColumnWidthParser for the "FixedWidthExample2.prm" file.

## Appendix B. Use cases and examples

### B.1.2. DSV examples (handled by the TokenSplitParser)

#### B.1.2.1. Comma delimited file example (CSV)

```
policyID,statecode,county,eq_site_limit,hu_site_limit,fl_site_limit,fr_site_limit,tiv_2011,tiv_2012,eq_site_deductible,hu_site_deductible,fl_site_deductible,fr_site_deductible,point_latitude,point_longitude,line,construction,point_granularity
119736,FL,CLAY COUNTY,498960,498960,498960,498960,792148.9,0,9979.2,0,0,30.102261,-81.711777,Residential,Masonry,1
448094,FL,CLAY COUNTY,1322376.3,1322376.3,1322376.3,1322376.3,1438163.57,0,0,0,30.063936,-81.707664,Residential,Masonry,1
206893,FL,CLAY COUNTY,190724.4,190724.4,190724.4,190724.4,192476.78,0,0,0,30.089579,-81.700455,Residential,Wood,1
333743,FL,CLAY COUNTY,0,79520.76,0,0,79520.76,86854.48,0,0,0,30.063236,-81.707703,Residential,Wood,3
172534,FL,CLAY COUNTY,0,254281.5,0,254281.5,246144.49,0,0,0,30.060614,-81.702675,Residential,Wood,1
785275,FL,CLAY COUNTY,0,515035.62,0,0,515035.62,884419.17,0,0,0,30.063236,-81.707703,Residential,Masonry,3
995932,FL,CLAY COUNTY,0,19260000,0,0,19260000,20610000,0,0,0,30.102226,-81.713882,Commercial,Reinforced Concrete,1
223488,FL,CLAY COUNTY,328500,328500,328500,328500,348374.25,0,16425,0,0,30.102217,-81.707146,Residential,Wood,1
433512,FL,CLAY COUNTY,315000,315000,315000,315000,265821.57,0,15750,0,0,30.118774,-81.704613,Residential,Wood,1
142071,FL,CLAY COUNTY,705600,705600,705600,705600,1010842.56,14112,35280,0,0,30.100628,-81.703751,Residential,Masonry,1
253816,FL,CLAY COUNTY,831498.3,831498.3,831498.3,831498.3,1117791.48,0,0,0,30.10216,-81.719444,Residential,Masonry,1
894922,FL,CLAY COUNTY,0,24059.09,0,0,24059.09,33952.19,0,0,0,30.095957,-81.695099,Residential,Wood,1
422834,FL,CLAY COUNTY,0,48115.94,0,0,48115.94,66755.39,0,0,0,30.100073,-81.739822,Residential,Wood,1
582721,FL,CLAY COUNTY,0,28869.12,0,0,28869.12,42826.99,0,0,0,30.09248,-81.725167,Residential,Wood,1
842700,FL,CLAY COUNTY,0,56135.64,0,0,56135.64,50656.8,0,0,0,30.101356,-81.726248,Residential,Wood,1
874333,FL,CLAY COUNTY,0,48115.94,0,0,48115.94,67905.07,0,0,0,30.113743,-81.727463,Residential,Wood,1
```

Figure B.4.: A large CSV file example called “FL\_insurance\_sample.csv”. Source is [spatialkey.com](https://support.spatialkey.com/spatialkey-sample-csv-data/)<sup>2</sup>.

```
result = (ArrayList@8391) size = 36634
  0 = (HashMap@9679) size = 18
    0 = (HashMap$Entry@9786) "policyID" -> "119736"
    1 = (HashMap$Entry@9787) "statecode" -> "FL"
    2 = (HashMap$Entry@9788) "county" -> "CLAY COUNTY"
    3 = (HashMap$Entry@9789) "eq_site_limit" -> "498960.0"
    4 = (HashMap$Entry@9790) "hu_site_limit" -> "498960.0"
    5 = (HashMap$Entry@9791) "fl_site_limit" -> "498960.0"
    6 = (HashMap$Entry@9792) "fr_site_limit" -> "498960.0"
    7 = (HashMap$Entry@9793) "tiv_2011" -> "498960.0"
    8 = (HashMap$Entry@9794) "tiv_2012" -> "792148.9"
    9 = (HashMap$Entry@9795) "eq_site_deductible" -> "0.0"
    10 = (HashMap$Entry@9796) "hu_site_deductible" -> "9979.2"
    11 = (HashMap$Entry@9797) "fl_site_deductible" -> "0.0"
    12 = (HashMap$Entry@9798) "fr_site_deductible" -> "0.0"
    13 = (HashMap$Entry@9799) "point_latitude" -> "30.10226"
    14 = (HashMap$Entry@9800) "point_longitude" -> "-81.71178"
    15 = (HashMap$Entry@9801) "line" -> "Residential"
    16 = (HashMap$Entry@9802) "construction" -> "Masonry"
    17 = (HashMap$Entry@9803) "point_granularity" -> "1"
  1 = (HashMap@9680) size = 18
  2 = (HashMap@9681) size = 18
```

Figure B.5.: The resulting pseudo-object maps of the “FL\_insurance\_sample.csv” file, with the first object on display.

<sup>2</sup>[support.spatialkey.com/spatialkey-sample-csv-data/](https://support.spatialkey.com/spatialkey-sample-csv-data/) (Last visited: 11.01.2021)

## B.1. Extraction examples

Example parser for a CSV file		hu_site_limit	Float	fl_site_deductible	Float
CSV file parser configuration example		<input type="checkbox"/> Optional		<input type="checkbox"/> Optional	
FL_insurance_sample		4		11	
.csv		fl_site_limit	Float	fr_site_deductible	Float
policyID,statecode,county		<input type="checkbox"/> Optional		<input type="checkbox"/> Optional	
add another substring to ignore		5		12	
policyID	Integer	fr_site_limit	Float	point_latitude	Float
<input type="checkbox"/> Optional		<input type="checkbox"/> Optional		<input type="checkbox"/> Optional	
0		6		13	
Starts from 0					
statecode	String	tiv_2011	Float	point_longitude	Float
<input type="checkbox"/> Optional		<input type="checkbox"/> Optional		<input type="checkbox"/> Optional	
1		7		14	
county	String	tiv_2012	Float	line	String
<input type="checkbox"/> Optional		<input type="checkbox"/> Optional		<input type="checkbox"/> Optional	
2		8		15	
eq_site_limit	Float	eq_site_deductible	Float	construction	String
<input type="checkbox"/> Optional		<input type="checkbox"/> Optional		<input type="checkbox"/> Optional	
3		9		16	
		hu_site_deductible	Float	point_granularity	Integer
		<input type="checkbox"/> Optional		<input type="checkbox"/> Optional	
		10		17	

Figure B.6.: A collage of a functional configuration of the TokenSplitParser for the CSV file “FL\_insurance\_sample.csv”.

## Appendix B. Use cases and examples

### B.1.2.2. Tab delimited file example (TSV)

Here we have a lightweight example displaying how to configure the `TokenSplitParser` for a TSV file. The tab delimiter has to be specified as “\t” in its corresponding field.

```
left town composition different fix disease wheel running
stems spend double caught giant tea get nodded
two perfect fog fifty paid chosen next fast
correct writer railroad dress source jump cattle earlier
missing gray bowl as select condition became felt
taken bone industrial lonely able busy various quick
gulf figure indeed exactly depth that pride system
men then over brought even means property myself
burst exclaimed clear cap empty invented modern visitor
vote cent group dear doctor gave kids composed
struggle crew usual sail excited cold her town
as attached school rising longer unusual higher bowl
pet house college brain giving shoe excited have
correct route land tobacco method disappear factor factor
owner angry rays market actual upon thing almost
attempt someone choice missing start journey curve surprise
thousand distant official doctor birth travel finally poor
```

Figure B.7.: A simple nonsensical TSV file example called “output-onlinerandomtools.tsv”. Source is [onlinerandomtools.com](https://onlinerandomtools.com)<sup>3</sup>.

```
▼ ∞ result = {ArrayList@9630} size = 100
  ▼ 0 = {LinkedHashMap@9757} size = 8
    ▶ 0 = {LinkedHashMap$Entry@9864} "Field 0" -> "left"
    ▶ 1 = {LinkedHashMap$Entry@9865} "Field 1" -> "town"
    ▶ 2 = {LinkedHashMap$Entry@9866} "Field 2" -> "composition"
    ▶ 3 = {LinkedHashMap$Entry@9867} "Field 3" -> "different"
    ▶ 4 = {LinkedHashMap$Entry@9868} "Field 4" -> "fix"
    ▶ 5 = {LinkedHashMap$Entry@9869} "Field 5" -> "disease"
    ▶ 6 = {LinkedHashMap$Entry@9870} "Field 6" -> "wheel"
    ▶ 7 = {LinkedHashMap$Entry@9871} "Field 7" -> "running"
    ▶ 1 = {LinkedHashMap@9758} size = 8
```

Figure B.8.: The resulting pseudo-object maps of the “output-onlinerandomtools.tsv” file, with the first object on display.

<sup>3</sup><https://onlinerandomtools.com/generate-random-tsv> (Last visited: 11.01.2021)

## B.1. Extraction examples

Example parser for a TSV file	
TSV file parser configuration example	
output-onlinerandomtools	
.tsv	
\t	
substring to ignore	
add another substring to ignore	
Field 0	String
<input type="checkbox"/> Optional	
0	
Starts from 0	
Field 1	String
<input type="checkbox"/> Optional	
1	
Field 2	String
<input type="checkbox"/> Optional	
2	
Field 3	String
<input type="checkbox"/> Optional	
3	
Field 4	String
<input type="checkbox"/> Optional	
4	
Field 5	String
<input type="checkbox"/> Optional	
5	
Field 6	String
<input type="checkbox"/> Optional	
6	
Field 7	String
<input type="checkbox"/> Optional	
7	

Figure B.9.: A collage of a functional configuration of the TokenSplitParser for the TSV file “output-onlinerandomtools.tsv”.



## Appendix B. Use cases and examples

### B.1.2.3. Whitespace delimited file example

This whitespace file is an interesting example, as each of the float numbers are optional. Further, a comment can appear, replacing a float number.

```
Id-- Date---- desc- ## 1 to 6 numbers or optionally a comment
1111 20181101 type1 01 943.23 02 403.91 03 1149.20 04 495.38 05 1001.49 06 259.75
1111 20181101 type2 01 502.08 02 440.03 03 694.12 04 272.13 05 439.41 06 42.96
1234 20181101 type1 01 870.34 02 947.29 03 1633.80 04 1222.15
1234 20181101 type2 01 592.12 02 464.46 03 907.61 04 580.85
2222 20181101 type1 01 1331.13 02 777.82 03 633.91
2222 20181101 type2 01 405.68 02 454.11 03 455.96
4588 20181101 type1 01 748.82 02 1312.00 -- WOW-- !
4588 20181101 type2 01 874.31 02 827.15 -- WOW-- !
5006 20181101 type1 01 1771.53 02 852.21 03 2830.33 04 922.68 05 2700.30 06 1111.33
5006 20181101 type2 01 479.05 02 351.01 03 1099.49 04 581.89 05 739.62 06 623.25
6504 20181101 type1 01 0.00 02 1029.81 03 0.00 04 1831.85 -- WOW-- !
6504 20181101 type2 01 0.00 02 293.36 03 0.00 04 651.05 -- WOW-- !
NEW STUFF
Id-- Date---- desc-- # number
1432 20181101 type3 1 10.51
1433 20181101 type3 1 83.34
1444 20181101 type3 1 34.75
```

Figure B.10.: A simple nonsensical whitespace delimited file example called “whitespace\_delimited\_file.txt”.

```
▼ result = (ArrayList@9664) size = 15
  ► 0 = (HashMap@9795) size = 9
  ▼ 1 = (HashMap@9796) size = 9
    ► 0 = (HashMap$Entry@9921) "ID" -> "1111"
    ► 1 = (HashMap$Entry@9922) "Date" -> "20181101"
    ► 2 = (HashMap$Entry@9923) "Description" -> "type2"
    ► 3 = (HashMap$Entry@9924) "Number 1" -> "502.08"
    ► 4 = (HashMap$Entry@9925) "Number 2" -> "440.03"
    ► 5 = (HashMap$Entry@9926) "Number 3" -> "694.12"
    ► 6 = (HashMap$Entry@9927) "Number 4" -> "272.13"
    ► 7 = (HashMap$Entry@9928) "Number 5" -> "439.41"
    ► 8 = (HashMap$Entry@9929) "Number 6" -> "42.96"
  ▼ 7 = (HashMap@9802) size = 6
    ► 0 = (HashMap$Entry@9906) "ID" -> "4588"
    ► 1 = (HashMap$Entry@9907) "Date" -> "20181101"
    ► 2 = (HashMap$Entry@9908) "Description" -> "type2"
    ► 3 = (HashMap$Entry@9909) "Number 1" -> "874.31"
    ► 4 = (HashMap$Entry@9910) "Number 2" -> "827.15"
    ► 5 = (HashMap$Entry@9911) "Comment 2" -> "WOW--"
  ► 8 = (HashMap@9803) size = 9
```

Figure B.11.: The resulting pseudo-object maps of the “whitespace\_delimited\_file.txt” file, with two notable objects on display. Entries 2 to 7 were omitted.

## B.1. Extraction examples

Do note that the field for the separation token (a.k.a. delimiter) here is not empty, but actually a whitespace character “`␣`”.

Whitespace delimited file parser	Number 2	Float	Comment 2	String
An example of a parser that deals with whites	<input checked="" type="checkbox"/> Optional		<input checked="" type="checkbox"/> Optional	
whitespace	6		8	
.txt				
Id-- Date----	<input checked="" type="checkbox"/> Optional		<input checked="" type="checkbox"/> Optional	
NEW STUFF	8		10	
<a href="#">add another substring to ignore</a>				
ID	Integer	Number 4	Float	Comment 4
<input type="checkbox"/> Optional		<input checked="" type="checkbox"/> Optional		<input checked="" type="checkbox"/> Optional
0		10		12
Starts from 0				
		Number 5	Float	Comment 5
Date	String	<input checked="" type="checkbox"/> Optional		<input checked="" type="checkbox"/> Optional
<input type="checkbox"/> Optional		12		14
1				
		Number 6	Float	Comment 6
Description	String	<input checked="" type="checkbox"/> Optional		<input type="checkbox"/> Optional
<input type="checkbox"/> Optional		14		16
2				
		Comment 1	String	
		<input checked="" type="checkbox"/> Optional		
Number 1	Float	6		
<input type="checkbox"/> Optional				
4				

Figure B.12.: A collage of a functional configuration of the TokenSplitParser for the whitespace delimited file “`whitespace_delimited_file.txt`”.

### B.1.3. Tag-based file examples (JSON/YAML) (handled by the SimpleTagParser)

The SimpleTagParser is a more sophisticated parser with many features. For that, a file was prepared showcasing following problems:

- Some numerals, in this case floats, have no end tag, but only a domain-end-tag that can mark the end of such.
- Each array has a different representation per object. The first such array even contains other objects, which can not be parsed properly, but still extracted as a string for later transforming.
- There are multi-line strings here for the tag “type”.
- The last pseudo-object is missing the optional “boolValue”.
- Some objects have a different permutation than the rest.

```
{
  "id": "environment07/wtB",
  "type": "tinkerforge/motion_detector/motion_detected",
  "arrivalTime": 1493721461.6555664,
  "payload": {
    "boolValue": true,
    "timestamp": 1493721461.1656731
  }
  "array": [
    {"value": "New", "onclick": "CreateNewDoc()"},
    {"value": "Open", "onclick": "OpenDoc()"},
    {"value": "Close", "onclick": "CloseDoc()"}
  ]
}
{
  "payload": {
    "boolValue": false,
    "timestamp": 1234567861.656731
  }
  "id": "environment03/wtB",
  "type": "tinkerforge/
  motion_detector/
  motion_detected",
  "array": [1,2,3,4],
  "arrivalTime": 1234567861.664
}
{
  "id": "environment05/wtB",
  "type": "tinkerforge/motion_detector/motion_detected",
  "arrivalTime": 8978674756.6456464,
  "payload": {
    "timestamp": 9876543223.656731
  }
  "array": [1,
2,
3,
4]
}
```

Figure B.13.: A remarkable JSON file example called “all\_inclusive\_test.json”.

## B.1. Extraction examples

Json parser example		arrivalTime	Float
Parser for a JSON example file that showcase		<input type="checkbox"/> Optional	
all_inclusive_test		"arrivalTime":	
.json		,	
{		Array splitting tag	
}			
1			
id	String	boolValue	Boolean
<input type="checkbox"/> Optional		<input checked="" type="checkbox"/> Optional	
"id": "	",	"boolValue":	
Array splitting tag		,	
		Array splitting tag	
type	String	timestamp	Float
<input type="checkbox"/> Optional		<input type="checkbox"/> Optional	
"type": "	",	"timestamp":	
Array splitting tag		Endtag	
		Array splitting tag	
		array	String
		<input type="checkbox"/> Optional	
		"array": [	
		]	
		,	

Figure B.14.: A collage of a functional configuration of the SimpleTagParser for the JSON file "all\_inclusive\_test.json".

## Appendix B. Use cases and examples

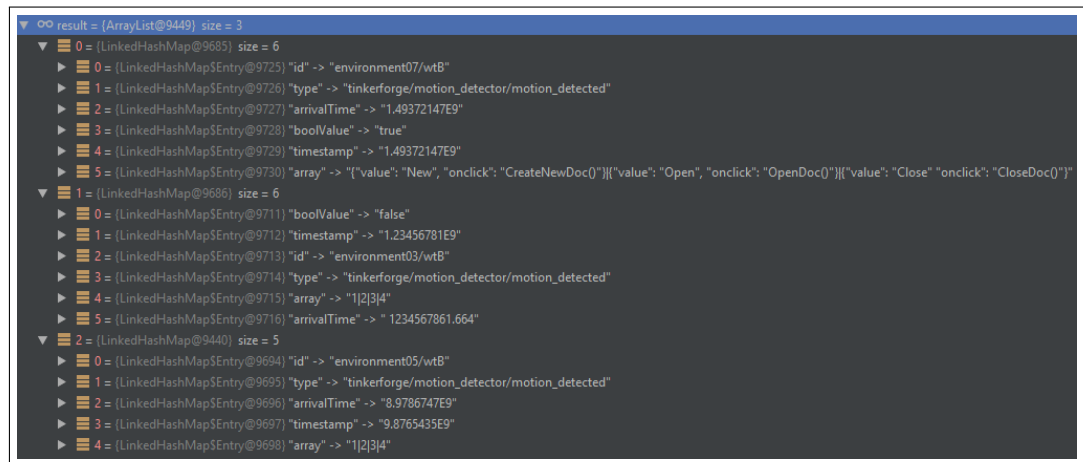


Figure B.15.: The resulting pseudo-object maps of the “all\_inclusive\_test.json” file, displaying all three object variations.

### Potential problems without domain tags

Beside the example above, in which some numerals need the domain-end-tag to mark their end, there is another problem that can arise with optional values and no set domain-start/end-tags. An example for such is illustrated below.

```
{
  "timestamp": "2016-07-26 22:03:10.464"
}

{
  "custom_value": "blah specialcase",
  "timestamp": "2016-07-26 22:03:13.557"
}

{
  "custom_value": "blah 1",
  "timestamp": "2016-07-26 22:03:13.758"
}
```

Figure B.16.: A simple JSON file showcasing the problem that can arise with optional values and no specified domain start/end-tag.

This example has an optional “custom\_value” that does not have to appear in each pseudo-object.

## B.1. Extraction examples

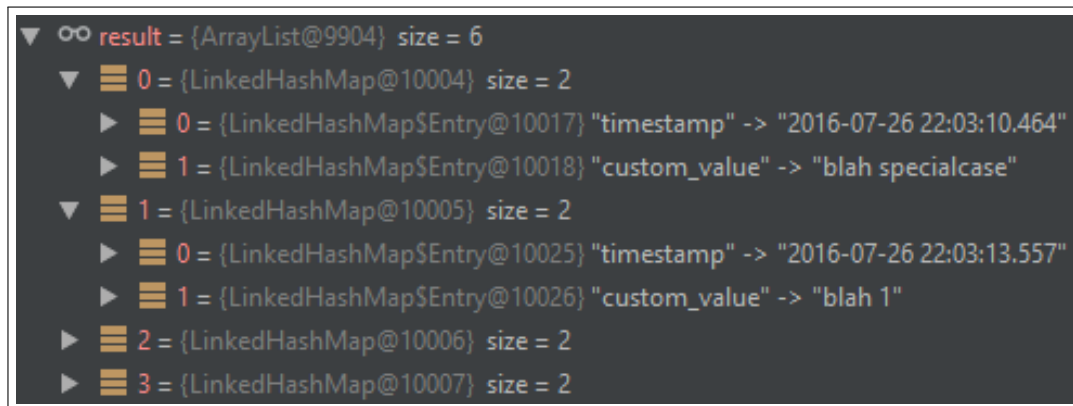


Figure B.17.: The resulting pseudo-object maps of the “all\_inclusive\_test.json” file, displaying all three object variations.

Note how the “custom\_value” of each pseudo-object is wrongfully shifted to the previous pseudo-object, because of how the **SimpleTagParser** evaluates an object’s end without additional info, thus demonstrating why domain tags are recommended to be set if possible.

## Appendix B. Use cases and examples

### Tag based file with a single line

The **SimpleTagParser** is also able to parse files that only have a single line. The use of this feature is not recommended, but might be necessary in some cases. Do keep in mind that the parser prettifies such single-lined files beforehand, if they are JSON. This prettifying puts each key-value pair into a single line, at the same time these key-value pairs might be reordered within their respective pseudo-objects, meaning previously valid end-tags might now be missing. E.g. numerals usually have a comma marking their end, except if there is a following domain end-tag “}” in which case the comma is missing.

There will be no illustration for the problem mentioned above, but instead a regular example on how to configure the DIM for a single-lined text file. This file is secretly a JSON file, but was renamed to be a simple text file so that the parser is deceived for this particular use-case. It can not be properly shown in a figure due to its wide and sizeable nature. Instead, a copy is available under its link [here](#). The “queries.txt” file is composed of multiple articles from [wikipedia.org](#)<sup>4</sup>, [warfarehistorian.blogspot.de](#)<sup>5</sup>, [habsburger.net](#)<sup>6</sup> and [britannica.com](#)<sup>7</sup>.

---

<sup>4</sup><https://www.wikipedia.org/> (Last visited: 11.01.2021)

<sup>5</sup><https://warfarehistorian.blogspot.de/> (Last visited: 11.01.2021)

<sup>6</sup><https://www.habsburger.net//> (Last visited: 11.01.2021)

<sup>7</sup><https://www.britannica.com/> (Last visited: 11.01.2021)

## B.1. Extraction examples

Single line file parser	url	String
Configuration example for a single line tag-based file		
queries	<input type="checkbox"/> Optional "url": " " , Array splitting tag	
.txt		
Domain start tag		
Domain end tag	selectedText String	
Nesting level (starts at 0, increments per domain start tag)	<input type="checkbox"/> Optional "selectedText": " " , Array splitting tag	
lang	String	
<input type="checkbox"/> Optional "lang": " " , Array splitting tag		
task_name	String	
<input type="checkbox"/> Optional "task_name": " " , Array splitting tag		
task_id	Integer	
<input type="checkbox"/> Optional "task_id": " , Array splitting tag		
timestamp	Long	
<input type="checkbox"/> Optional "timestamp": " , Array splitting tag		
query	String	
<input type="checkbox"/> Optional "query": " " , Array splitting tag		
id	Integer	
<input type="checkbox"/> Optional "id": " } Array splitting tag		

Figure B.18.: A collage of a functional configuration of the SimpleTagParser for the single line file “queries.txt”.



## Appendix B. Use cases and examples

```
▶ 90 = (LinkedHashMap@9726) size = 9
▶ 91 = (LinkedHashMap@9727) size = 9
▶ 92 = (LinkedHashMap@9728) size = 9
▼ 93 = (LinkedHashMap@9729) size = 9
  ▶ 0 = (LinkedHashMap$Entry@9978) "lang" -> "all"
  ▶ 1 = (LinkedHashMap$Entry@9979) "task_name" -> "T.A3,en"
  ▶ 2 = (LinkedHashMap$Entry@9980) "task_id" -> "6"
  ▶ 3 = (LinkedHashMap$Entry@9981) "timestamp" -> "1394192881852"
  ▶ 4 = (LinkedHashMap$Entry@9982) "url" -> "http://warfarehistorian.blogspot.de/"
  ▶ 5 = (LinkedHashMap$Entry@9983) "selectedText" -> "This elite brigade of frogmen, sailors, and naval commandos most renowned for their role in the rescue of the Italian 10th MAS during the Battle of the Strait of Otranto in 1941."
  ▶ 6 = (LinkedHashMap$Entry@9984) "paragraphs" -> "Odd Fighting Units: Italian Frogmen and the Human Torpedoes, 1940-1943|The Decima Flottiglia MAS (Decima Flottiglia Mezzi d'Assalto) known as La Decima or Artist Depiction of Italian Regia Marina Frogmen, c.1941|When examining the impact that the Italian 10th MAS had during its years of service"
  ▶ 7 = (LinkedHashMap$Entry@9985) "query" -> "italian frogman"
  ▶ 8 = (LinkedHashMap$Entry@9986) "id" -> "443"
▶ 94 = (LinkedHashMap@9730) size = 9
```

Figure B.19.: The resulting pseudo-object maps of the “queries.txt” file, displaying one such object.

```
Odd Fighting Units: Italian Frogmen and the Human Torpedoes, 1940-1943
The Decima Flottiglia MAS (Decima Flottiglia Mezzi d'Assalto) known as La Decima or
Artist Depiction of Italian Regia Marina Frogmen, c.1941
When examining the impact that the Italian 10th MAS had during its years of service
```

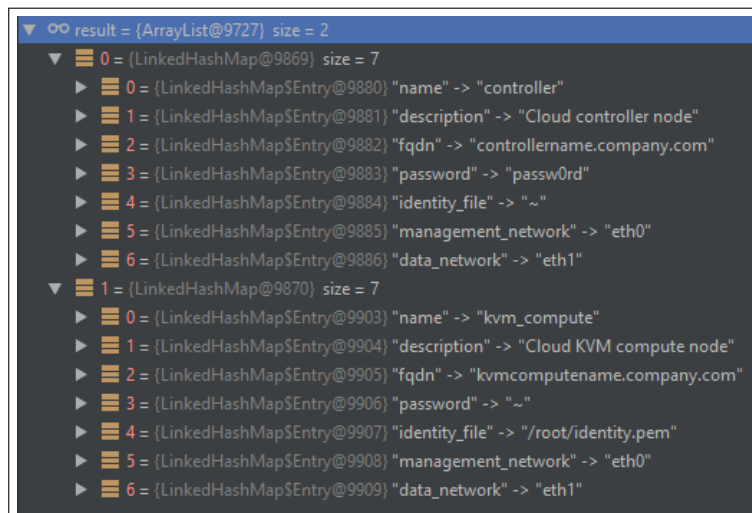
Figure B.20.: Bonus figure of the “paragraphs” array with the delimiting character “|” being replaced by the newline character.

## B.1. Extraction examples

### B.1.3.1. YAML example (a tag based file format)

```
nodes:
- name: controller
  description: Cloud controller node
  fqdn: controllername.company.com
  password: password
  identity_file: ~
  nics:
    management_network: eth0
    data_network: eth1
- name: kvm_compute
  description: Cloud KVM compute node
  fqdn: kvmcomputename.company.com
  password: ~
  identity_file: /root/identity.pem
  nics:
    management_network: eth0
    data_network: eth1
```

Figure B.21.: A simple YAML file example called “yaml exampl.yaml”. Source is [ibm.com](https://www.ibm.com/support/knowledgecenter/SST55W_4.3.0/liaca/liaca_cli_cloud_yaml_cfg.html)<sup>8</sup>.



```
▼ result = (ArrayList@9727) size = 2
  ▼ 0 = (LinkedHashMap@9869) size = 7
    ▶ 0 = (LinkedHashMap$Entry@9880) "name" -> "controller"
    ▶ 1 = (LinkedHashMap$Entry@9881) "description" -> "Cloud controller node"
    ▶ 2 = (LinkedHashMap$Entry@9882) "fqdn" -> "controllername.company.com"
    ▶ 3 = (LinkedHashMap$Entry@9883) "password" -> "password"
    ▶ 4 = (LinkedHashMap$Entry@9884) "identity_file" -> "~"
    ▶ 5 = (LinkedHashMap$Entry@9885) "management_network" -> "eth0"
    ▶ 6 = (LinkedHashMap$Entry@9886) "data_network" -> "eth1"
  ▼ 1 = (LinkedHashMap@9870) size = 7
    ▶ 0 = (LinkedHashMap$Entry@9903) "name" -> "kvm_compute"
    ▶ 1 = (LinkedHashMap$Entry@9904) "description" -> "Cloud KVM compute node"
    ▶ 2 = (LinkedHashMap$Entry@9905) "fqdn" -> "kvmcomputename.company.com"
    ▶ 3 = (LinkedHashMap$Entry@9906) "password" -> "~"
    ▶ 4 = (LinkedHashMap$Entry@9907) "identity_file" -> "/root/identity.pem"
    ▶ 5 = (LinkedHashMap$Entry@9908) "management_network" -> "eth0"
    ▶ 6 = (LinkedHashMap$Entry@9909) "data_network" -> "eth1"
```

Figure B.22.: The resulting pseudo-object maps of the “yaml exampl.yaml” file, displaying the two object variations.

<sup>8</sup>[https://www.ibm.com/support/knowledgecenter/SST55W\\_4.3.0/liaca/liaca\\_cli\\_cloud\\_yaml\\_cfg.html](https://www.ibm.com/support/knowledgecenter/SST55W_4.3.0/liaca/liaca_cli_cloud_yaml_cfg.html) (Last visited: 11.01.2021)

## Appendix B. Use cases and examples

YAML exampl parser		password	String
For the tag based file format called yaml, as an example		<input type="checkbox"/> Optional	
yaml exampl		password:	Endtag
.yaml		Array splitting tag	
Domain start tag			
Domain end tag		identity_file	String
Nesting level (starts at 0, increments per domain start tag)		<input type="checkbox"/> Optional	
name	String	identity_file:	Endtag
<input type="checkbox"/> Optional		Array splitting tag	
name:	Endtag		
Array splitting tag			
		management_network	String
<input type="checkbox"/> Optional		<input type="checkbox"/> Optional	
description	String	management_network:	Endtag
<input type="checkbox"/> Optional		Array splitting tag	
description:	Endtag		
Array splitting tag			
		data_network	String
<input type="checkbox"/> Optional		<input type="checkbox"/> Optional	
fqdn	String	data_network:	Endtag
<input type="checkbox"/> Optional		Array splitting tag	
fqdn:	Endtag		
Array splitting tag			

Figure B.23.: A collage of a functional configuration of the SimpleTagParser for the YAML file “yaml-exampl.yaml”.

### B.1.4. XML examples handled by the SimpleXMLParser

#### B.1.4.1. Regular XML examples

```
<?xml version="1.0" encoding="UTF-8"?>
<CATALOG>
  <CD>
    <TITLE>Empire Burlesque</TITLE>
    <ARTIST>Bob Dylan</ARTIST>
    <COUNTRY>USA</COUNTRY>
    <COMPANY>Columbia</COMPANY>
    <PRICE>10.90</PRICE>
    <YEAR>1985</YEAR>
  </CD>
  <CD>
    <TITLE>Hide your heart</TITLE>
    <ARTIST>Bonnie Tyler</ARTIST>
    <COUNTRY>UK</COUNTRY>
    <COMPANY>CBS Records</COMPANY>
    <PRICE>9.90</PRICE>
    <YEAR>1988</YEAR>
  </CD>
</CATALOG>
```

Figure B.24.: A simple XML file example called “cd\_catalog.xml”. Source is w3schools.com<sup>9</sup>.

```
▼ result = {ArrayList@9635} size = 26
  ▼ 0 = {LinkedHashMap@9766} size = 6
    ► 0 = {LinkedHashMap$Entry@9799} "TITLE" -> "Empire Burlesque"
    ► 1 = {LinkedHashMap$Entry@9800} "ARTIST" -> "Bob Dylan"
    ► 2 = {LinkedHashMap$Entry@9801} "COUNTRY" -> "USA"
    ► 3 = {LinkedHashMap$Entry@9802} "COMPANY" -> "Columbia"
    ► 4 = {LinkedHashMap$Entry@9803} "PRICE" -> "10.9"
    ► 5 = {LinkedHashMap$Entry@9804} "YEAR" -> "1985"
  ► 1 = {LinkedHashMap@9767} size = 6
  ► 2 = {LinkedHashMap@9768} size = 6
  ► 3 = {LinkedHashMap@9769} size = 6
```

Figure B.25.: The resulting pseudo-object maps of the “cd\_catalog.xml” file, with the first object on display.

<sup>9</sup>[https://www.w3schools.com/xml/xml\\_examples.asp](https://www.w3schools.com/xml/xml_examples.asp) (Last visited: 11.01.2021)

## Appendix B. Use cases and examples

Note that the **SimpleXMLParser** is the only parser requiring its fields to have the exact same name as the tags in the XML file. A requirement only expected by regular XML files. The “ExcelTag” and the “ignored Super-/bottom-tags” are not in use for regular XML, but only for SpreadsheetML, with an example of such being shown in the following section.

XML parser example	
An exemplary parser for simple XML files with repe	
cd_catalog	
.xml ▼	
CD	
Exceltag (for SpreadsheetML)	
# of ignored top supertags	
# of ignored bottom supertags	
TITLE	String ▼
ARTIST	String ▼
COUNTRY	String ▼
COMPANY	String ▼
PRICE	Float ▼
YEAR	Integer ▼

Figure B.26.: A functional configuration of the SimpleXMLParser for the regular XML file “cd\_catalog.xml”. Fields are identical to the XML’s pseudo-object tags.

## B.1. Extraction examples

### B.1.4.2. SpreadsheetML examples

```
<Worksheet ss:Name="Sheet1">
  <Table ss:ExpandedColumnCount="5" ss:ExpandedRowCount="12"
    x:FullColumns="1" x:FullRows="1">
    <Column ss:AutoFitWidth="0" ss:Width="73.5"/>
    <Column ss:AutoFitWidth="0" ss:Width="96.75"/>
    <Column ss:Index="5" ss:AutoFitWidth="0" ss:Width="56.25"/>

    <Row>
      <Cell ss:StyleID="s23"><Data ss:Type="String">Sales
        for:</Data></Cell>
      <Cell ss:StyleID="s21"><Data ss:Type="DateTime">
        2004-01-01T00:00:00.000</Data><NamedCell ss:Name="Date"></Cell>
    </Row>

    <Row ss:Index="3" ss:StyleID="s23">
      <Cell><Data ss:Type="String">ID Number</Data></Cell>
      <Cell><Data ss:Type="String">Criter</Data></Cell>
      <Cell><Data ss:Type="String">Price</Data></Cell>
      <Cell><Data ss:Type="String">Quantity</Data></Cell>
      <Cell><Data ss:Type="String">Total</Data></Cell>
    </Row>

    <Row>
      <Cell><Data ss:Type="Number">4627</Data><NamedCell
        ss:Name="ID"/></Cell>
      <Cell><Data ss:Type="String">Diplodocus</Data><NamedCell
        ss:Name="Critters"/></Cell>
      <Cell ss:StyleID="s22"><Data ss:Type="Number">22.5</Data>
        <NamedCell ss:Name="Price"/></Cell>
      <Cell><Data ss:Type="Number">127</Data><NamedCell
        ss:Name="Quantity"/></Cell>
      <Cell ss:StyleID="s22" ss:Formula="=RC[-2]*RC[-1]"><Data
        ss:Type="Number">2857.5</Data></Cell>
    </Row>

    <Row>
      <Cell ss:Index="4" ss:StyleID="s23"><Data
        ss:Type="String">Total:</Data></Cell>
      <Cell ss:StyleID="s22" ss:Formula="=SUM(R[-8]C:R[-1]C)">
        <Data ss:Type="Number">15147.5</Data><NamedCell
        ss:Name="Total"/></Cell>
    </Row>
  </Table>
```

Figure B.27.: A collage consisting of the beginning and ending of a SpreadsheetML file example called “spreadsheetML example.xml”. Source is etutorials.org<sup>10</sup>.

```
result = (ArrayList@9882) size = 8
  0 = (LinkedHashMap$Entry@9950) "ID Number" -> "4627"
  1 = (LinkedHashMap$Entry@9951) "Criter" -> "Diplodocus"
  2 = (LinkedHashMap$Entry@9952) "Price" -> "22.5"
  3 = (LinkedHashMap$Entry@9953) "Quantity" -> "127"
  4 = (LinkedHashMap$Entry@9954) "Total" -> "2857.5"
  7 = (LinkedHashMap$Entry@9942) size = 5
    0 = (LinkedHashMap$Entry@9967) "ID Number" -> "8649"
    1 = (LinkedHashMap$Entry@9968) "Criter" -> "Barosaurus"
    2 = (LinkedHashMap$Entry@9969) "Price" -> "17.0"
    3 = (LinkedHashMap$Entry@9970) "Quantity" -> "91"
    4 = (LinkedHashMap$Entry@9971) "Total" -> "1547.0"
```

Figure B.28.: The resulting pseudo-object maps of the “spreadsheetML example.xml” file, with the first and last objects on display.

<sup>10</sup><https://etutorials.org/XML/xml+hacks/Chapter+3.+Transforming+XML+Documents/Hack+42+Create+and+Process+SpreadsheetML/> (Last visited: 11.01.2021)

Appendix B. Use cases and examples

For this example, the XML supertag has to be set to the value “Row”, the name of the pseudo-object tag. Additionally, the “Exceltag” field, the “ignored top supertags” field, and the “ignored bottom supertags” field have to be set so that the SpreadsheetML can be parsed properly. The “Exceltag” is needed as an identifier, so that the parser is able to properly extract the data, as the SpreadsheetML format encapsulates each value in a “Data” tag. The “ignored supertags” are used for filtering meta information pseudo-objects from the parse. Contained within the SpreadsheetML example are two such meta information elements, one in the beginning and one in the end, all of which are not desirable. An example of such meta data can be seen in figure B.27.

SpreadsheetML parser example

Example configuration for a SpreadsheetML parser

spreadsheetML

.xml

Row

Data

2

1

ID Number

Integer

Critter

String

Price

Float

Quantity

Integer

Total

Float

Figure B.29.: A functional configuration of the SimpleXMLParser for the SpreadsheetML file “spreadsheetML example.xml”.

## B.2. Transformation examples

All transformation methods use the generated pseudo-object maps from one of the configured parsers in the previous chapter. Further, each transformation method begins with an example on how its routine might be configured, and concludes with an example on how a loading method is executed. The loading methods can be either “identityTransferAndSave” on its own, or “identityTransfer” in combination with “saveAllObjects”, with the latter combination loading data into a beforehand prepared test domain object. This means that there is no extra chapter dealing specifically with the load, as this is already showcased within the transformation examples. All in all, the basic examples will comprise of around three figures to demonstrate a typical use case, with each building upon the extraction result and then performing transformation and load. Depending on the complexity of the transformation method, more illustrations will be given to fully showcase their features.

### B.2.1. “Update properties” examples

Examples on how to use the “update properties” mechanism can be found [here](#) and [here](#).

Note that whenever “update properties” are used for a table/domain object, all subsequent routines also have to use the same “update properties” for this particular table/domain object. If not done so, the program will inevitably throw an exception when one such misconfigured routine mistakenly appends a duple value to a table, with another later routine using the “update properties” mechanism trying to find a unique matching object, of which there are now multiple instances instead of the expected singular one.

### B.2.2. “Notable objects” examples

Examples on how to use the “notable objects” mechanism can be found [here](#) and [here](#).



B.2.3. appendString example

A pseudo-object map, generated by parsing the “cd.catalog.xml” file, was used for this example. It is shown in figure B.25. The “Sequence number” does not have to be set, as the controller takes the first available number, starting with zero, by default.

Parser:

XMLParser for XML parser example

Object:

testing.XML\_CD\_Catalog\_Test

Submit selected

☐ Update existing

Sequence number

Properties of object to update:

Please select a p:

Please select an <

Figure B.30.: Routine configuration.

Routine:

Routine #0 for 'testing.XML\_CD\_Catalog\_1

Methodname:

appendString

Submit selected

Sequence number

☒ Repetitive procedure

Notable parsing objects:

Please select an

property value

add objectpair

Method parameters:

COMPANY

\_Corporation

add parameterpair

Figure B.31.: Procedure definition of “appendString”.

## B.2. Transformation examples

**Routine:**  
Routine #0 for 'testing.XML\_CD\_Catalog\_Test' in parser 'XML' ▼

**Methodname:**  
identityTransferAndSave ▼

Submit selected

Sequence number

☒ Repetitive procedure

**Notable parsing objects:**  
Please select an entry... ▼ property value

add objectpair

**Method parameters:**

artist	ARTIST
company	COMPANY
country	COUNTRY
price	PRICE
title	TITLE
year	YEAR

Figure B.32.: The corresponding loading procedure.

```
package testing

class XML_CD_Catalog_Test {
    String title
    String artist
    String country
    String company
    float price
    int year

    static constraints = {
    }
}
```

Figure B.33.: The domain class that is used for the load.

53	0	Bob Dylan	Columbia_Corporation	USA	10.9	Empire Burlesque	1985
54	0	Bonnie Tyler	CBS Records_Corporation	UK	9.9	Hide your heart	1988
55	0	Dolly Parton	RCA_Corporation	USA	9.9	Greatest Hits	1982
56	0	Gary Moore	Virgin records_Corporation	UK	10.2	Still got the blues	1990
57	0	Eros Ramazzotti	BMG_Corporation	EU	9.9	Eros	1997
58	0	Bee Gees	Polydor_Corporation	UK	10.9	One night only	1998
59	0	Dr.Hook	CBS_Corporation	UK	8.1	Sylvias Mother	1973
60	0	Rod Stewart	Pickwick_Corporation	UK	8.5	Maggie May	1990
61	0	Andrea Bocelli	Polydor_Corporation	EU	10.8	Romanza	1996
62	0	Percy Sledge	Atlantic_Corporation	USA	8.7	When a man loves a woman	1987
63	0	Savage Rose	Mega_Corporation	EU	10.9	Black angel	1995
64	0	Many	Grammy_Corporation	USA	10.2	1999 Grammy Nominees	1999
65	0	Kenny Rogers	Mucik Master_Corporation	UK	8.7	For the good times	1995
66	0	Will Smith	Columbia_Corporation	USA	9.9	Big Willie style	1997

Figure B.34.: Resulting table of “appendString” and load in the database.

### B.2.4. prependString

This example uses the same routine, loading method, domain class, and file as “appendString” does. It is executed after “appendString”, but before “identityTransferAndSave”. The sequence number of “identityTransferAndSave” was increased for this reason, by manually editing the database.

Additionally, the “notable parsing objects” mechanism is illustrated here. It was used to configure the procedure. so it would only apply the transformation method onto data whose “companies” match the value “Polydor\_Corporation”. Keep in mind that “appendString” is executed prior, meaning “.Corporation” has to be appended to the original value “Polydor”, to make sure they match at runtime.

The image shows a web-based form for defining a procedure. The form is titled 'prependString' and is part of a routine named 'Routine #0 for 'testing.XML\_CD\_Catalog\_Test' i'. The form includes several sections: 'Methodname:' with a dropdown menu set to 'prependString'; 'Sequence number' with an empty text input; 'Repetitive procedure' with a checked checkbox; 'Notable parsing objects:' with a dropdown menu set to 'COMPANY' and a text input set to 'Polydor\_Corporation'; 'Method parameters:' with two rows of inputs. The first row has 'Newcomer:' and 'ARTIST' (dropdown). The second row has 'Innovating' and 'COMPANY' (dropdown). There are two orange buttons: 'Submit selected' and 'add objectpair'.

<b>Routine:</b>	
Routine #0 for 'testing.XML_CD_Catalog_Test' i	
<b>Methodname:</b>	
prependString	
Submit selected	
Sequence number	
<input checked="" type="checkbox"/> Repetitive procedure	
<b>Notable parsing objects:</b>	
COMPANY	Polydor_Corporation
add objectpair	
<b>Method parameters:</b>	
Newcomer:	ARTIST
Innovating	COMPANY

Figure B.35.: Procedure definition of “prependString”.

## B.2. Transformation examples

34	0	Dr.Hook	CBS_Corporation	UK	8.1	Sylvias Mother	1973
35	0	Rod Stewart	Pickwick_Corporation	UK	8.5	Maggie May	1990
36	0	Newcomer: Andrea Bocelli	Innovating Polydor_Corporation	EU	10.8	Romanza	1996
37	0	Percy Sledge	Atlantic_Corporation	USA	8.7	When a man loves a woman	1987
38	0	Savage Rose	Mega_Corporation	EU	10.9	Black angel	1995
39	0	Many	Grammy_Corporation	USA	10.2	1999 Grammy Nominees	1999
40	0	Kenny Rogers	Muck Master_Corporation	UK	8.7	For the good times	1995
41	0	Will Smith	Columbia_Corporation	USA	9.9	Big Willie style	1997
42	0	Newcomer: Van Morrison	Innovating Polydor_Corporation	UK	8.2	Tupelo Honey	1971
43	0	Jorn Hoel	WEA_Corporation	Norway	7.9	Soulsville	1996
44	0	Cat Stevens	Island_Corporation	UK	8.9	The very best of	1990
45	0	Sam Brown	A and M_Corporation	UK	8.9	Stop	1988
46	0	TPau	Siren_Corporation	UK	7.9	Bridge of Spies	1987
47	0	Tina Turner	Capitol_Corporation	UK	8.9	Private Dancer	1983
48	0	Kim Larsen	Medley_Corporation	EU	7.8	Midt om natten	1983
49	0	Luciano Pavarotti	DECCA_Corporation	UK	9.9	Pavarotti Gala Concert	1991
50	0	Otis Redding	Stax Records_Corporation	USA	7.9	The dock of the bay	1968

Figure B.36.: Resulting table of conditional “prependString” and load in the database.

Note how only rows with “Polydor\_Corporation” have the prepend applied.

### B.2.5. trimField

Same deal again, this example uses the same routine, loading method, file, and domain class as its predecessors. This method too is executed after all others, but before “identityTransferAndSave”. Most parsers apply a trim on all the extracted values by default. Luckily, the SimpleXMLParser is not such a parser, allowing for a modification to the “cd\_catalog.xml” file, so that it contains values with an excessive amount of spaces.

```
<CD>
  <TITLE>    Hide your heart    </TITLE>
  <ARTIST>Bonnie Tyler          </ARTIST>
  <COUNTRY>UK</COUNTRY>
  <COMPANY>CBS Records</COMPANY>
  <PRICE>9.90</PRICE>
  <YEAR>1988</YEAR>
</CD>
<CD>
  <TITLE>Greatest Hits</TITLE>
  <ARTIST>                Dolly Parton</ARTIST>
  <COUNTRY>                USA          </COUNTRY>
  <COMPANY>RCA</COMPANY>
  <PRICE>9.90</PRICE>
  <YEAR>1982</YEAR>
</CD>
```

Figure B.37.: The altered file that is used for the showcase.

## Appendix B. Use cases and examples

**Routine:**  
Routine #0 for 'testing.XML\_CD\_Catalog\_Test' i ▼

**Methodname:**  
trimField ▼

Submit selected

Sequence number

☒ Repetitive procedure

**Notable parsing objects:**  
Please select an ei ▼ property value

add objectpair

**Method parameters:**  
ARTIST ▼ COUNTRY ▼  
TITLE ▼ Please select a dat ▼

Figure B.38.: Procedure definition of “trimField”.

id	version	artist	company	country	price	title	year
1	0	Bob Dylan	Columbia_Corporation	USA	10.9	Empire Burlesque	1985
2	0	Bonnie Tyler	CBS Records_Corporation	UK	9.9	Hide your heart	1988
3	0	Dolly Parton	RCA_Corporation	USA	9.9	Greatest Hits	1982
4	0	Gary Moore	Virgin records_Corporation	UK	10.2	Still got the blues	1990
5	0	Eros Ramazzotti	BMG_Corporation	EU	9.9	Eros	1997
6	0	Newcomer: Bee Gees	Innovating Polydor_Corporation	UK	10.9	One night only	1998
7	0	Dr.Hook	CBS_Corporation	UK	8.1	Sylvias Mother	1973
8	0	Rod Stewart	Pickwick_Corporation	UK	8.5	Maggie May	1990

Figure B.39.: Resulting table additional “trimField” and load in the database.

### B.2.6. splitStringField

This method creates new fields into which the split values are saved in. For this reason a new domain class, routine, and loading procedure were created, with the new domain class extending the previously used one by a few variables. Each string is split by a white space character “\_”. The pseudo-object maps generated here are also from the “cd\_catalog” file shown in figure B.25.

Additionally, an illustration is given on how to configure the “notable objects” mechanism. Here, data is skipped whenever their “country” map entry is of the value “EU”.

Figure B.40.: A collage of the loading procedure’s configuration.

Note that the configuration shown in figure B.40 is placed prior to the other figures, but occurs chronologically after the “splitStringField” procedure.

Appendix B. Use cases and examples

**Parser:**  

XMLParser for XML parser example ▾

**Object:**  

testing.XML\_CD\_Catalog\_Test\_V2 ▾

Submit selected

☐ Update existing

Sequence number

**Properties of object to update:**  

Please select a p ▾ Please select an ▾

Figure B.41.: The routine configuration.

```
package testing

class XML_CD_Catalog_Test_V2 {
    String title
    String artist
    String country
    String company
    float price
    int year

    String splitTitle_1
    String splitTitle_2
    String splitTitle_3
    String splitTitle_4
    String splitTitle_5
    String splitTitle_6

    static constraints = {
        splitTitle_1 nullable: true
        splitTitle_2 nullable: true
        splitTitle_3 nullable: true
        splitTitle_4 nullable: true
        splitTitle_5 nullable: true
        splitTitle_6 nullable: true
    }
}
```

Figure B.42.: The extended domain class.

**Routine:**  

Routine #1 for 'testing.XML\_CD\_Catalog\_Test\_V2' ▾

**Methodname:**  

splitStringField ▾

Submit selected

Sequence number

☒ Repetitive procedure

**Notable parsing objects:**  

COUNTRY ▾ EU

add objectpair

**Method parameters:**  

TITLE ▾

title\_split\_1 0

title\_split\_2 1

title\_split\_3 2

title\_split\_4 3

title\_split\_5 4

title\_split\_6 5

Figure B.43.: The procedure configuration for “splitStringField”.

All transformation methods shown in the previous three sections are performed beforehand in a separate transformation routine. This results in the

## B.2. Transformation examples

following table with the values being unlike corresponding values in the original file, possibly causing confusion at first sight.

artist	company	country	price	split_title_1	split_title_2	split_title_3	split_title_4	split_title_5	split_title_6	title	year
Bob Dylan	Columbia_Corporation	USA	10.9	Empire	Burlesque	NULL	NULL	NULL	NULL	Empire Burlesque	1985
Bonnie Tyler	CBS_Records_Corpo...	UK	9.9	Hide	your	heart	NULL	NULL	NULL	Hide your heart	1988
Dolly Parton	RCA_Corporation	USA	9.9	Greatest	Hits	NULL	NULL	NULL	NULL	Greatest Hits	1982
Gary Moore	Virgin_records_Corp...	UK	10.2	Still	got	the	blues	NULL	NULL	Still got the blues	1990
Eros Ramazzotti	BMG_Corporation	EU	9.9	NULL	NULL	NULL	NULL	NULL	NULL	Eros	1997
Newcomer: Bee G...	Innovating Polydor_...	UK	10.9	One	night	only	NULL	NULL	NULL	One night only	1998
Dr.Hook	CBS_Corporation	UK	8.1	Sylvias	Mother	NULL	NULL	NULL	NULL	Sylvias Mother	1973
Rod Stewart	Pickwick_Corporation	UK	8.5	Maggie	May	NULL	NULL	NULL	NULL	Maggie May	1990
Newcomer: Andr...	Innovating Polydor_...	EU	10.8	NULL	NULL	NULL	NULL	NULL	NULL	Romanza	1996
Percy Sledge	Atlantic_Corporation	USA	8.7	When	a	man	loves	a	woman	When a man loves a woman	1987
Savage Rose	Mega_Corporation	EU	10.9	NULL	NULL	NULL	NULL	NULL	NULL	Black angel	1995
Many	Grammy_Corporation	USA	10.2	1999	Grammy	Nominees	NULL	NULL	NULL	1999 Grammy Nominees	1999
Kenny Rogers	Muck Master_Corpo...	UK	8.7	For	the	good	times	NULL	NULL	For the good times	1995
Will Smith	Columbia_Corporation	USA	9.9	Big	Willie	style	NULL	NULL	NULL	Big Willie style	1997
Newcomer: Van ...	Innovating Polydor_...	UK	8.2	Tupelo	Honey	NULL	NULL	NULL	NULL	Tupelo Honey	1971

Figure B.44.: Resulting table of the “splitStringField” transformation and final load.

Every unused map entry is saved as null in the database. This includes data that was skipped via the “notable objects” process, which sets all entries to null by default.

### B.2.7. concatenateFields

This use case builds entirely on the map entries that were added by the previous use case “splitStringField”.

Here, a new routine is created, with a configuration set to update the database via the “update properties” mechanism built into the DIM, simultaneously illustrating said mechanism. As per usual, this routine concludes with a loading procedure that overrides the domain object’s title with the new transformed title.



Appendix B. Use cases and examples

Parser:

XMLParser for XML parser example

Object:

testing.XML\_CD\_Catalog\_Test\_V2

Submit selected

☒ Update existing

Sequence number

Properties of object to update:

TITLE

title

Figure B.45.: The routine configured with update properties

Routine:

Routine #2 for 'testing.XML\_CD\_Catalog\_'

Methodname:

identityTransferAndSave

Submit selected

Sequence number

☒ Repetitive procedure

Notable parsing objects:

Please select an

property value

add objectpair

Method parameters:

title

title\_v2

Figure B.46.: Configuration of another load-ing procedure.

Routine:

Routine #2 for 'testing.XML\_CD\_Catalog\_'

Methodname:

concatenateFields

Submit selected

Sequence number

☒ Repetitive procedure

Notable parsing objects:

Please select an

property value

add objectpair

Method parameters:

title\_v2

|

add parameterpair

title\_split\_1

title\_split\_2

title\_split\_3

title\_split\_4

title\_split\_6

title\_split\_5

Figure B.47.: The procedure configuration for “concatenateFields”.

The loading procedure is the last executed method and is carried out after the transformation procedure “concatenateFields”.

## B.2. Transformation examples

artist	company	country	price	split_title_1	split_title_2	split_title_3	split_title_4	split_title_5	split_title_6	title	year
Bob Dylan	Columbia_Corporation	USA	10.9	Empire	Burlesque	NULL	NULL	NULL	NULL	Empire Burlesque	1985
Bonnie Tyler	CBS_Records_Corpo...	UK	9.9	Hide	your	heart	NULL	NULL	NULL	Hide your heart	1988
Dolly Parton	RCA_Corporation	USA	9.9	Greatest	Hits	NULL	NULL	NULL	NULL	Greatest Hits	1982
Gary Moore	Virgin_records_Corp...	UK	10.2	Still	got	the	blues	NULL	NULL	Still got the blues	1990
Eros Ramazzotti	BMG_Corporation	EU	9.9	NULL	NULL	NULL	NULL	NULL	NULL		1997
Newcomer: Bee G...	Innovating Polydor_...	UK	10.9	One	night	only	NULL	NULL	NULL	One night only	1998
Dr.Hook	CBS_Corporation	UK	8.1	Sylvias	Mother	NULL	NULL	NULL	NULL	Sylvias Mother	1973
Rod Stewart	Pickwick_Corporation	UK	8.5	Maggie	May	NULL	NULL	NULL	NULL	Maggie May	1990
Newcomer: Andr...	Innovating Polydor_...	EU	10.8	NULL	NULL	NULL	NULL	NULL	NULL		1996
Percy Sledge	Atlantic_Corporation	USA	8.7	When	a	man	loves	a	woman	When a man loves woman a	1987

Figure B.48.: Result of the “concatenateFields” procedure and a routine that updates already loaded object.

Note how the property “title” is replaced with the value of the new “title\_v2” property, created by “concatenateFields”. All domain objects shown in the following table were already present in the database from a previously executed routine, then retrieved by the update mechanism, and their title altered by the routine #2 and its procedures. This modification is marked by the incremented “version” column, which is not shown in figure B.48, but would be zero for otherwise fresh objects.

### B.2.8. calculateSum

This example builds upon the pseudo-object maps that are generated by the whitespace delimited file, described in the [TokenSplit-Parser](#) section.

```
package testing

class TokenSplitTestClass {

    // "id" would be problematic
    int identification
    String date
    String description
    float num_1
    float num_2
    float num_3
    float num_4
    float num_5
    float num_6
    String comment
    float mean
    float sum
    float result

    static constraints = {
        comment nullable: true
    }
}
```

Figure B.49.: Domain class used for load.

Appendix B. Use cases and examples

Parser:

Token-split-parser for Whitespace delimiter

Object:

testing.TokenSplitTestClass

Submit selected

☐ Update existing

Sequence number

Properties of object to update:

Please select a p

Please select an

Figure B.50.: The routine configuration.

Routine:

Routine #0 for 'testing.TokenSplitTestClass

Methodname:

calculateSum

Submit selected

0

☒ Repetitive procedure

Notable parsing objects:

Please select an

property value

add objectpair

Method parameters:

sum

parametervalue

Number 1

Number 2

Number 3

Number 4

Number 5

Number 6

Figure B.51.: Configuration of the “calculateSum” procedure.

Routine:

Routine #0 for 'testing.TokenSplitTestCl

Methodname:

identityTransferAndSave

Submit selected

10

☒ Repetitive procedure

Notable parsing objects:

Please select a

property value

add objectpair

Method parameters:

identification

ID

description

Description

date

Date

num\_1

Number 1

num\_2

Number 2

num\_3

Number 3

num\_4

Number 4

num\_5

Number 5

num\_6

Number 6

sum

sum

Figure B.52.: The loading configuration executed after “calculateSum”.

## B.2. Transformation examples

comment	date	description	identification	mean	num_1	num_2	num_3	num_4	num_5	num_6	result	sum
NULL	20181101	type1	1111	0	943.23	403.91	1149.2	495.38	1001.49	259.75	0	4252.96
NULL	20181101	type2	1111	0	502.08	440.03	694.12	272.13	439.41	42.96	0	2390.73
NULL	20181101	type1	1234	0	870.34	947.29	1633.8	1222.15	0	0	0	4673.58
NULL	20181101	type2	1234	0	592.12	464.46	907.61	580.85	0	0	0	2545.04
NULL	20181101	type1	2222	0	1331.13	777.82	633.91	0	0	0	0	2742.86
NULL	20181101	type2	2222	0	405.68	454.11	455.96	0	0	0	0	1315.75
NULL	20181101	type1	4588	0	748.82	1312	0	0	0	0	0	2060.82
NULL	20181101	type2	4588	0	874.31	827.15	0	0	0	0	0	1701.46
NULL	20181101	type1	5006	0	1771.53	852.21	2830.33	922.68	2700.3	1111.33	0	10188.4
NULL	20181101	type2	5006	0	479.05	351.01	1099.49	581.89	739.62	623.25	0	3874.31
NULL	20181101	type1	6504	0	0	1029.81	0	1831.85	0	0	0	2861.66
NULL	20181101	type2	6504	0	0	293.36	0	651.05	0	0	0	944.41
NULL	20181101	type3	1432	0	10.51	0	0	0	0	0	0	10.51
NULL	20181101	type3	1433	0	83.34	0	0	0	0	0	0	83.34
NULL	20181101	type3	1444	0	34.75	0	0	0	0	0	0	34.75

Figure B.53.: Resulting table of the “calculateSum” procedure and final load.

### B.2.9. calculateMean

Another illustration that expands on the domain class, routine, and loading procedure from the [previous “calculateMean” example](#), with the difference that the load was adapted to now also include the mean. Additionally, the transformation method’s multiplication functionality is used, by setting the corresponding parameter to ten, multiplying the result of the calculation by the same value.

Appendix B. Use cases and examples

Routine:

Routine #0 for 'testing.TokenSplitTestClass' ir

Methodname:

calculateMean

Submit selected

1|

☒ Repetitive procedure

Notable parsing objects:

Please select an (

property value

add objectpair

Method parameters:

mean

10

Number 1

Number 2

Number 3

Number 4

Number 5

Number 6

Figure B.54.: Configuration of the “calculateMean” procedure.

Routine:

Routine #0 for 'testing.TokenSplitTestClas:

Methodname:

identityTransferAndSave

Submit selected

Sequence number

☒ Repetitive procedure

Notable parsing objects:

Please select an

property value

add objectpair

Method parameters:

identification

ID

description

Description

date

Date

num\_1

Number 1

num\_2

Number 2

num\_3

Number 3

num\_4

Number 4

num\_5

Number 5

num\_6

Number 6

sum

sum

mean

mean

Figure B.55.: Adapted loading configuration that is executed after “calculateMean”.

## B.2. Transformation examples

Transformation and loading procedures can not be edited via the UI as of now, meaning figure B.55 was provided as an example on how the new load would look like if it were configured afterwards.

comment	date	description	identification	mean	num_1	num_2	num_3	num_4	num_5	num_6	result	sum
NULL	20181101	type1	1111	7088.27	943.23	403.91	1149.2	495.38	1001.49	259.75	0	4252.96
NULL	20181101	type2	1111	3984.55	502.08	440.03	694.12	272.13	439.41	42.96	0	2390.73
NULL	20181101	type1	1234	11684	870.34	947.29	1633.8	1222.15	0	0	0	4673.58
NULL	20181101	type2	1234	6362.6	592.12	464.46	907.61	580.85	0	0	0	2545.04
NULL	20181101	type1	2222	9142.87	1331.13	777.82	633.91	0	0	0	0	2742.86
NULL	20181101	type2	2222	4385.83	405.68	454.11	455.96	0	0	0	0	1315.75
NULL	20181101	type1	4588	10304.1	748.82	1312	0	0	0	0	0	2060.82
NULL	20181101	type2	4588	8507.3	874.31	827.15	0	0	0	0	0	1701.46
NULL	20181101	type1	5006	16980.6	1771.53	852.21	2830.33	922.68	2700.3	1111.33	0	10188.4
NULL	20181101	type2	5006	6457.18	479.05	351.01	1099.49	581.89	739.62	623.25	0	3874.31
NULL	20181101	type1	6504	7154.15	0	1029.81	0	1831.85	0	0	0	2861.66
NULL	20181101	type2	6504	2361.02	0	293.36	0	651.05	0	0	0	944.41
NULL	20181101	type3	1432	105.1	10.51	0	0	0	0	0	0	10.51
NULL	20181101	type3	1433	833.4	83.34	0	0	0	0	0	0	83.34
NULL	20181101	type3	1444	347.5	34.75	0	0	0	0	0	0	34.75
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Figure B.56.: Resulting table of the “calculateMean” procedure and final load.

Note that the fourth and fifth row from the bottom both contain zero on column “num\_1” and “num\_3”. These zero values were parsed from the file and are used for the mean calculation, whereas zeroes on “num\_5” and “num\_6” are default values not used for the calculation.

### B.2.10. arithmeticOperation

For this showcase, the same routine as “calculateSum” and “calculateMean” is used. It also follows those two in their execution, but precedes the load, with the load again being adapted to now include the result into the load. This example benefits from the pseudo-object maps generated from the whitespace delimited file shown in the [TokenSplitParser example](#).

“ArithmeticOperation” allows the user to either chose map entries, or to put in custom numbers. The showcase uses a map entry for the left value and a custom number for the right value.

Appendix B. Use cases and examples

This method does not work with optional values as the previous methods do, i.e. keys that return null values for map entries will throw an exception if not parsable.

Routine:

Routine #0 for 'testing.TokenSplitTestClass' in parse

Methodname:

arithmeticOperation

Submit selected

2

☒ Repetitive procedure

Notable parsing objects:

Please select an er

property value

add objectpair

Method parameters:

result

+

Number 1

1000

Figure B.57.: Configuration of the “arithmeticOperation” procedure.

Routine:

Routine #0 for 'testing.TokenSplitTestClass' in parse

Methodname:

identityTransferAndSave

Submit selected

Sequence number

☒ Repetitive procedure

Notable parsing objects:

Please select an entr

property value

add objectpair

Method parameters:

identification

ID

description

Description

date

Date

num\_1

Number 1

num\_2

Number 2

num\_3

Number 3

num\_4

Number 4

num\_5

Number 5

num\_6

Number 6

sum

sum

mean

mean

result

result

Figure B.58.: Adapted loading configuration that is executed after “arithmeticOperation”.

## B.2. Transformation examples

Figure B.58 provides an example on how the new load would look like if it were configured afterwards, as there is currently no way to edit previous configurations via the UI.

comment	date	description	identification	mean	num_1	num_2	num_3	num_4	num_5	num_6	result	sum
NULL	20181101	type1	1111	7088.27	943.23	403.91	1149.2	495.38	1001.49	259.75	1943.23	4252.96
NULL	20181101	type2	1111	3984.55	502.08	440.03	694.12	272.13	439.41	42.96	1502.08	2390.73
NULL	20181101	type1	1234	11684	870.34	947.29	1633.8	1222.15	0	0	1870.34	4673.58
NULL	20181101	type2	1234	6362.6	592.12	464.46	907.61	580.85	0	0	1592.12	2545.04
NULL	20181101	type1	2222	9142.87	1331.13	777.82	633.91	0	0	0	2331.13	2742.86
NULL	20181101	type2	2222	4385.83	405.68	454.11	455.96	0	0	0	1405.68	1315.75
NULL	20181101	type1	4588	10304.1	748.82	1312	0	0	0	0	1748.82	2060.82
NULL	20181101	type2	4588	8507.3	874.31	827.15	0	0	0	0	1874.31	1701.46
NULL	20181101	type1	5006	16980.6	1771.53	852.21	2830.33	922.68	2700.3	1111.33	2771.53	10188.4
NULL	20181101	type2	5006	6457.18	479.05	351.01	1099.49	581.89	739.62	623.25	1479.05	3874.31
NULL	20181101	type1	6504	7154.15	0	1029.81	0	1831.85	0	0	1000	2861.66
NULL	20181101	type2	6504	2361.02	0	293.36	0	651.05	0	0	1000	944.41
NULL	20181101	type3	1432	105.1	10.51	0	0	0	0	0	1010.51	10.51

Figure B.59.: Resulting table of the "+" "arithmeticOperation" procedure and final load.

## Bonus results for all available arithmetic operations

Only the originally configured procedure's operator was altered for the bonuses, the rest was left the same.

comment	date	description	identification	mean	num_1	num_2	num_3	num_4	num_5	num_6	result	sum
NULL	20181101	type1	1111	7088.27	943.23	403.91	1149.2	495.38	1001.49	259.75	-56.77	4252.96
NULL	20181101	type2	1111	3984.55	502.08	440.03	694.12	272.13	439.41	42.96	-497.92	2390.73
NULL	20181101	type1	1234	11684	870.34	947.29	1633.8	1222.15	0	0	-129.66	4673.58
NULL	20181101	type2	1234	6362.6	592.12	464.46	907.61	580.85	0	0	-407.88	2545.04
NULL	20181101	type1	2222	9142.87	1331.13	777.82	633.91	0	0	0	331.13	2742.86

Figure B.60.: Snippet of the resulting table using the "-" "arithmeticOperation" procedure and final load.



## Appendix B. Use cases and examples

comment	date	description	identification	mean	num_1	num_2	num_3	num_4	num_5	num_6	result	sum
NULL	20181101	type1	1111	7088.27	943.23	403.91	1149.2	495.38	1001.49	259.75	0.94323	4252.96
NULL	20181101	type2	1111	3984.55	502.08	440.03	694.12	272.13	439.41	42.96	0.50208	2390.73
NULL	20181101	type1	1234	11684	870.34	947.29	1633.8	1222.15	0	0	0.87034	4673.58
NULL	20181101	type2	1234	6362.6	592.12	464.46	907.61	580.85	0	0	0.59212	2545.04
NULL	20181101	type1	2222	9142.87	1331.13	777.82	633.91	0	0	0	1.33113	2742.86

Figure B.61.: Snippet of the resulting table using the “/” “arithmeticOperation” procedure and final load.

comment	date	description	identification	mean	num_1	num_2	num_3	num_4	num_5	num_6	result	sum
NULL	20181101	type1	1111	7088.27	943.23	403.91	1149.2	495.38	1001.49	259.75	943230	4252.96
NULL	20181101	type2	1111	3984.55	502.08	440.03	694.12	272.13	439.41	42.96	502080	2390.73
NULL	20181101	type1	1234	11684	870.34	947.29	1633.8	1222.15	0	0	870340	4673.58
NULL	20181101	type2	1234	6362.6	592.12	464.46	907.61	580.85	0	0	592120	2545.04
NULL	20181101	type1	2222	9142.87	1331.13	777.82	633.91	0	0	0	1331130	2742.86

Figure B.62.: Snippet of the resulting table using the “\*” “arithmeticOperation” procedure and final load.

comment	date	description	identification	mean	num_1	num_2	num_3	num_4	num_5	num_6	result	sum
NULL	20181101	type1	1111	7088.27	943.23	403.91	1149.2	495.38	1001.49	259.75	943.23	4252.96
NULL	20181101	type2	1111	3984.55	502.08	440.03	694.12	272.13	439.41	42.96	502.08	2390.73
NULL	20181101	type1	1234	11684	870.34	947.29	1633.8	1222.15	0	0	870.34	4673.58
NULL	20181101	type2	1234	6362.6	592.12	464.46	907.61	580.85	0	0	592.12	2545.04
NULL	20181101	type1	2222	9142.87	1331.13	777.82	633.91	0	0	0	331.13	2742.86

Figure B.63.: Snippet of the resulting table using the “%” “arithmeticOperation” procedure and final load.

## B.2. Transformation examples

### B.2.11. unaryArithmeticOperation

**Routine:**  
Routine #0 for 'testing.TokenSplitTestClass' ▼

**Methodname:**  
unaryArithmeticOperation ▼

Submit selected

Sequence number

☒ Repetitive procedure

**Notable parsing objects:**  
Please select an ▼ property value

add objectpair

**Method parameters:**

result ▼	++ ▼
mean ▼	-- ▼
sum ▼	- ▼

This example too uses the same routine as the “calculateSum”, “calculateMean”, and “arithmeticOperation” use cases. It follows those three in their execution, but precedes the load concluding the routine. To showcase this method, the fields created by the previously mentioned operations were each respectively altered by the “unaryArithmeticOperation”.

Figure B.64.: Configuration of the “unaryArithmeticOperation”

comment	date	description	identification	mean	num_1	num_2	num_3	num_4	num_5	num_6	result	sum
NULL	20181101	type1	1111	7087.27	943.23	403.91	1149.2	495.38	1001.49	259.75	1944.23	-4252.96
NULL	20181101	type2	1111	3983.55	502.08	440.03	694.12	272.13	439.41	42.96	1503.08	-2390.73
NULL	20181101	type1	1234	11683	870.34	947.29	1633.8	1222.15	0	0	1871.34	-4673.58
NULL	20181101	type2	1234	6361.6	592.12	464.46	907.61	580.85	0	0	1593.12	-2545.04
NULL	20181101	type1	2222	9141.87	1331.13	777.82	633.91	0	0	0	2332.13	-2742.86
NULL	20181101	type2	2222	4384.83	405.68	454.11	455.96	0	0	0	1406.68	-1315.75
NULL	20181101	type1	4588	10303.1	748.82	1312	0	0	0	0	1749.82	-2060.82
NULL	20181101	type2	4588	8506.3	874.31	827.15	0	0	0	0	1875.31	-1701.46

Figure B.65.: Resulting table of the “unaryArithmeticOperation” procedure and final load.

## Appendix B. Use cases and examples

comment	date	description	identification	mean	num_1	num_2	num_3	num_4	num_5	num_6	result	sum
NULL	20181101	type1	1111	7088.27	943.23	403.91	1149.2	495.38	1001.49	259.75	1943.23	4252.96
NULL	20181101	type2	1111	3984.55	502.08	440.03	694.12	272.13	439.41	42.96	1502.08	2390.73
NULL	20181101	type1	1234	11684	870.34	947.29	1633.8	1222.15	0	0	1870.34	4673.58
NULL	20181101	type2	1234	6362.6	592.12	464.46	907.61	580.85	0	0	1592.12	2545.04

Figure B.66.: Snippet of original table for comparison.

### B.2.12. regexReplace

Two use cases will be shown to give some insight into the versatility of “regexReplace”.

#### Fixing formatted floats

Here, a potential way is displayed on how to deal with problematic float values introduced by the [ColumnWidthParser example](#). These float values of the columns “credit” and “debit” had to be parsed as strings, because Java’s “parseFloat” method is not able to parse such values with commas, or periods, depending on the localization.

This illustration builds upon the aforementioned section’s file and its generated pseudo-object maps.

## B.2. Transformation examples

**Routine:**  
Routine #0 for 'testing.FixedWidthTest' in parser 'E': ▼

**Methodname:**  
regexReplace ▼

**Submit selected**

Sequence number

☒ Repetitive procedure

**Notable parsing objects:**  
Please select an ent ▼ property value

**add objectpair**

**Method parameters:**  
Credit\_V2 ⓘ Credit ▼

, parametervalue

Figure B.67.: First procedure configuration for “regexReplace”.

**Routine:**  
Routine #0 for 'testing.FixedWidthTest' in parser 'E': ▼

**Methodname:**  
regexReplace ▼

**Submit selected**

Sequence number

☒ Repetitive procedure

**Notable parsing objects:**  
Please select an ent ▼ property value

**add objectpair**

**Method parameters:**  
Debit\_V2 Debit ▼

, parametervalue

Figure B.69.: Second procedure configuration for “regexReplace”.

**Routine:**  
Routine #0 for 'testing.FixedWidthTest' in parser 'Exam': ▼

**Methodname:**  
regexReplace ▼

**Submit selected**

Sequence number

☒ Repetitive procedure

**Notable parsing objects:**  
Please select an entry ▼ property value

**add objectpair**

**Method parameters:**  
Credit\_V2 Credit\_V2 ▼

^\$ 0

Figure B.68.: Third procedure configuration for “regexReplace”.

**Routine:**  
Routine #0 for 'testing.FixedWidthTest' in parser 'Exam': ▼

**Methodname:**  
regexReplace ▼

**Submit selected**

Sequence number

☒ Repetitive procedure

**Notable parsing objects:**  
Please select an entry ▼ property value

**add objectpair**

**Method parameters:**  
Debit\_V2 Debit\_V2 ▼

^\$ 0

Figure B.70.: Fourth procedure configuration for “regexReplace”.

Appendix B. Use cases and examples

The first two procedures use a simple replace operation to remove all the commas from the floats, whereas the last two procedures use regex and replace empty strings with zero. The last mentioned strings are generated due to “Credit” and “Debit” being optional in the parser configuration. Empty strings can not be parsed to float, and any attempt of doing so will result in exceptions being thrown, which is why replacing these empty string is necessary.

```
package testing

class FixedWidthTest {

    int entry
    int period
    String post_date
    long GL_Account
    String description
    String source
    String cflow
    String reference
    String post
    double debit
    double credit
    String allocated

    static constraints = {
        post_date nullable: true
        source nullable: true
        cflow nullable: true
        reference nullable: true
        post nullable: true
        allocated nullable: true
    }
}
```

Figure B.71.: The domain class that is used for the load.

Routine:

Routine #0 for 'testing.FixedWidthTest' in parser 'Exz

Methodname:

identityTransferAndSave

Submit selected

10

☒ Repetitive procedure

Notable parsing objects:

Please select an entr

property value

add objectpair

Method parameters:

GL\_Account

GL Account

allocated

Allocated

cflow

Cflow

credit

Credit\_V2

debit

Debit\_V2

description

Description

entry

Entry

period

Period

post

Post

post\_date

Post Date

reference

Reference

source

Source

Figure B.72.: Configuration of the loading procedure.

## B.2. Transformation examples

As a result of the previous transformation procedures, the credit and debit columns in the following table are now both proper floating point values.

gl_account	allocated	cfow	credit	debit	description	entry	period	post	post_date	reference	source
3937681126	Yes	Yes	0	75308.61	PYZNYBL ELQT	285643	3	Yes	02/20/2013	PYPL	S1
2069410641	Yes	Yes	0	59884.43	XVQ OIA XLVE	278434	3	Yes	02/15/2013	NFKY	S1
5958787235	Yes	Yes	0	43.45	VJX VYRCHRJ RXB T...	281241	3	Yes	02/18/2013	TOCG	S1
2584327630	No	Yes	0	1.89	TKQ JXP ISXV CEIF	284329	3	Yes	02/19/2013	PWMA	S1
5475117985	Yes	Yes	66233.44	0	QFR RPN KSECD QJRA	281604	3	Yes	02/18/2013	ZTHE	S1
2226484431	Yes	Yes	0	97818.54	MSJ NBZ NIXS	278436	3	Yes	02/15/2013	BMCN	S1
5661794541	Yes	Yes	45379.28	0	LJB XE PQ USKWBJT...	281239	3	Yes	02/18/2013	EBVO	S1
3394861553	Yes	Yes	0	92299.81	GVQ AEP ZKFLY GOAO	281603	3	Yes	02/18/2013	IIBY	S1
5660526806	Yes	Yes	0	12298.17	FDN QLT CYTM GSVJ	284331	3	Yes	02/19/2013	SRJF	S1
5649580335	Yes	Yes	72855.65	0	BMS FFB	261098	3	No	02/08/2013		S1
4732911590	Yes	Yes	618.88	0	DHV UYQ	261096	3	Yes	02/08/2013		S1
2068136558	Yes	No	63683.68	0	QLGZIUJ PY WTU SZL	294832	3	Yes	02/25/2013	PJLT	S1
0	NULL		12804234.52	21739353.22	ENDING BALANCE P...	0	0				
0	NULL		176367802.34	33417988.84	BALANCE	0	0				

Figure B.73.: Resulting table of all “regexReplace” procedures and final load.

### Regex example

Use was made of a previous table generated in the [ColumnWidthParser example](#) section. Here, the american date format is converted to a european format via the “regexReplace” transformation method. The procedure created for this example is once again placed into the same routine in which its predecessor “[regexFloat](#)” resides, and is executed after such but before the final load.

## Appendix B. Use cases and examples

Routine:

Routine #0 for 'testing.FixedWidthTest' in parser 'Examj'

Methodname:

regexReplace

Submit selected

Sequence number

☒ Repetitive procedure

Notable parsing objects:

Please select an entry

property value

add objectpair

Method parameters:

Post Date

Post Date

^(\d\*)(\d\*)(\d\*)\$

\$2/\$1/\$3

Figure B.74.: The procedure configuration for “regexReplace”.

The original map entry value is replaced with the new rearranged date, meaning no changes were needed for the loading procedure.

gl_account	allocated	cflow	credit	debit	description	entry	period	post	post_date	reference	source
3937681126	Yes	Yes	0	75308.61	PYZNYBL ELQT	285643	3	Yes	20/02/2013	PYPL	S1
2069410641	Yes	Yes	0	59884.43	XVQ OIA XLVE	278434	3	Yes	15/02/2013	NFKY	S1
5958787235	Yes	Yes	0	43.45	VJX VYRCHRJ RXB T...	281241	3	Yes	18/02/2013	TOCG	S1
2584327630	No	Yes	0	1.89	TKQ JXP ISXV CEIF	284329	3	Yes	19/02/2013	PWMA	S1
5475117985	Yes	Yes	66233.44	0	QFR RPN KSECD QJRA	281604	3	Yes	18/02/2013	ZTHE	S1
2226484431	Yes	Yes	0	97818.54	MSJ NBZ NIXS	278436	3	Yes	15/02/2013	BMCN	S1
5661794541	Yes	Yes	45379.28	0	LJB XE PQ USKWBJT...	281239	3	Yes	18/02/2013	EBVO	S1
3394861553	Yes	Yes	0	92299.81	GVQ AEP ZKFLY GOAO	281603	3	Yes	18/02/2013	IIBY	S1
5660526806	Yes	Yes	0	12298.17	FDN QLT CYTM GSVJ	284331	3	Yes	19/02/2013	SRJF	S1
5649580335	Yes	Yes	72855.65	0	BMS FFB	261098	3	No	08/02/2013		S1
4732911590	Yes	Yes	618.88	0	DHV UYQ	261096	3	Yes	08/02/2013		S1
2068136558	Yes	No	63683.68	0	QLGZIUJ PY WTU SZL	294832	3	Yes	25/02/2013	PJLT	S1
0	NULL		12804234.52	21739353.22	ENDING BALANCE P...	0	0				
0	NULL		176367802.34	33417988.84	BALANCE	0	0				

Figure B.75.: Resulting table of yet another “regexReplace” procedure and final load.

Note how the “post\_date” column now has dates in the european format.

### B.2.13. setValueFromOptionalValues

This showcase makes use of the generated pseudo-object maps from the whitespace delimited file shown in the [TokenSplitParser example](#). The tables from the previously depicted “[calculateSum](#)”, “[calculateMean](#)” and “[arithmeticOperation](#)” sections all contain a “comment” column that was empty. Said column was prepared beforehand for an optional string value that can occur in certain lines, and on any of position within such a line. The **TokenSplitParser** configuration requires the definition of multiple optional entries, so that the “comment” value is caught, if it occurs within any of these possible positions. This transformation method exists solely to sift through these multiple entries, find the one entry with a value, and store it into an object’s property. For this purpose, a new routine is created and configured to retrieve objects from the database, according to the specified “update properties”, in which the valid “comment” value is then ultimately saved.

The image shows a web-based configuration form. At the top, under the heading "Parser:", there is a dropdown menu currently showing "Token-split-parser for Whitespace delimited file parser". Below this, under the heading "Object:", is another dropdown menu showing "testing.TokenSplitTestClass". An orange button labeled "Submit selected" is positioned below the "Object:" dropdown. Underneath the button is a checkbox labeled "Update existing" which is checked. Below the checkbox is a text input field labeled "Sequence number". At the bottom of the form, under the heading "Properties of object to update:", there are two rows of dropdown menus. The first row has "ID" on the left and "identification" on the right. The second row has "Description" on the left and "description" on the right. All dropdown menus have a small downward arrow on their right side.

Figure B.76.: Routine configured with “update properties”.



Appendix B. Use cases and examples

Routine:

Routine #1 for 'testing.TokenSplitTestClass' in parser 'V'

Methodname:

setValueFromOptionalValues

Submit selected

Sequence number

☒ Repetitive procedure

Notable parsing objects:

Please select an entry

property value

add objectpair

Method parameters:

COMMENT\_V2

parametervalue

Comment 1

Comment 2

Comment 3

Comment 4

Comment 5

Comment 6

Figure B.77.: The configuration of the “setValueFromOptionalValues” procedure.

Routine:

Routine #1 for 'testing.TokenSplitTestClass' in parser 'V'

Methodname:

identityTransferAndSave

Submit selected

Sequence number

☒ Repetitive procedure

Notable parsing objects:

Please select an entry

property value

add objectpair

Method parameters:

comment

COMMENT\_V2

Figure B.78.: Adapted loading configuration that is executed after “setValueFromOptionalValues”.

As seen in figure B.79, some rows are now updated and contain the optional “comment”, that can appear in one out of six optional values. The column contains null whenever a datum only has null “comment” entries.

version	comment	date	description	identification	mean	num_1	num_2	num_3	num_4	num_5	num_6	result	sum
0	NULL	20181101	type1	2222	9141.87	1331.13	777.82	633.91	0	0	0	2332.13	-2742.86
0	NULL	20181101	type2	2222	4384.83	405.68	454.11	455.96	0	0	0	1406.68	-1315.75
1	WOW--	20181101	type1	4588	10303.1	748.82	1312	0	0	0	0	1749.82	-2060.82
1	WOW--	20181101	type2	4588	8506.3	874.31	827.15	0	0	0	0	1875.31	-1701.46
0	NULL	20181101	type1	5006	16979.6	1771.53	852.21	2830.33	922.68	2700.3	1111.33	2772.53	-10188.4
0	NULL	20181101	type2	5006	6456.18	479.05	351.01	1099.49	581.89	739.62	623.25	1480.05	-3874.31
1	WOW--	20181101	type1	6504	7153.15	0	1029.81	0	1831.85	0	0	1001	-2861.66
1	WOW--	20181101	type2	6504	2360.02	0	293.36	0	651.05	0	0	1001	-944.41
0	NULL	20181101	type3	1432	104.1	10.51	0	0	0	0	0	1011.51	-10.51

Figure B.79.: Resulting table of the “setValueFromOptionalValues” procedure and final load.

### B.2.14. setTimestamp

For the purpose of this transformation method and the following methods' illustrations, a new domain class was introduced that contains a timestamp property of the class "Timestamp". Additionally, a new routine for the "cd\_catalog" parser is created, in which the timestamp procedure will be inserted.

It is recommended to apply this method on all created objects, parsed by the DIM. The reason here being that the timestamp value can be used to filter data, if it was parsed improperly, or a transformation method went awry, both resulting in garbage being loaded into the database.

Parser:  
XMLParser for XML parser example ▼

Object:  
testing.RelationTestClass ▼

Submit selected

☐ Update existing

Sequence number

Properties of object to update:  
Please select a parser ▼ Please select an objec ▼

Figure B.80.: New routine configuration for the domain class "RelationTestClass".

Routine:  
Routine #3 for 'testing.RelationTestClass' in parser 'XML' ▼

Methodname:  
setTimestamp ▼

Submit selected

Sequence number

☒ Repetitive procedure

Notable parsing objects:  
Please select an entry ▼ property value

add objectpair

Method parameters:  
timestamp ▼ parametervalue

Figure B.81.: Configuration for the "setTimestamp" procedure.

Appendix B. Use cases and examples

```
package testing

import java.sql.Timestamp

class RelationTestClass {

    XML_CD_Catalog_Test relation
    Timestamp timestamp

    static hasMany = [relations: XML_CD_Catalog_Test]

    static constraints = {
        relation nullable: true
        timestamp nullable: true
    }
}
```

Figure B.82.: Domain class that is introduced with “setTimestamp”.

Routine:

Routine #3 for 'testing.RelationTestClass' in parser 'XML' ▾

Methodname:

saveAllObjects ▾

Submit selected

10

☒ Repetitive procedure

Notable parsing objects:

Please select an entry ▾ property value

Figure B.83.: Loading configuration for “setTimestamp”.

Note how a different loading procedure called “saveAllObjects” was used, instead of the usual “identityTransferAndSave”. The “setTimestamp” method is applied on each of the domain objects’ properties, instead of the pseudo-object map. This means a simple save of the modified domain object instance is enough and no explicit transfer is needed.

relation_id	timestamp
NULL	2019-01-19 17:22:38
NULL	2019-01-19 17:22:38
NULL	2019-01-19 17:22:38
NULL	2019-01-19 17:22:38
NULL	2019-01-19 17:22:38
NULL	2019-01-19 17:22:38
NULL	2019-01-19 17:22:38
NULL	2019-01-19 17:22:38
NULL	2019-01-19 17:22:38
NULL	2019-01-19 17:22:38
NULL	2019-01-19 17:22:38
NULL	2019-01-19 17:22:38

Figure B.84.: Resulting table of the “setTimestamp” procedure and final load.

### B.2.15. createRelation

This procedure builds upon the domain class introduced in the “[set-Timestamp](#)” section. Further, the procedure is also put into the same routine, and makes use of the same loading procedure. The individual “XML\_CD\_Catalog\_Test” object instances are retrieved by their title, which are assumed to be unique. Each retrieved object instance is then saved in the “RelationTestClass” object’s “relation” property.

Figure B.85.: Configuration of the “createRelation” procedure.

timestamp	artist	company	country	price	title	year
2019-01-19 19:04:14	Bob Dylan	Columbia_Corporation	USA	10.9	Empire Burlesque	1985
2019-01-19 19:04:14	Bonnie Tyler	CBS Records_Corporation	UK	9.9	Hide your heart	1988
2019-01-19 19:04:14	Dolly Parton	RCA_Corporation	USA	9.9	Greatest Hits	1982
2019-01-19 19:04:14	Gary Moore	Virgin records_Corporation	UK	10.2	Still got the blues	1990
2019-01-19 19:04:14	Eros Ramazzotti	BMG_Corporation	EU	9.9	Eros	1997
2019-01-19 19:04:14	Newcomer: Bee Gees	Innovating Polydor_Corporation	UK	10.9	One night only	1998
2019-01-19 19:04:14	Dr.Hook	CBS_Corporation	UK	8.1	Sylvias Mother	1973
2019-01-19 19:04:14	Rod Stewart	Pickwick_Corporation	UK	8.5	Maggie May	1990
2019-01-19 19:04:14	Newcomer: Andrea Bocelli	Innovating Polydor_Corporation	EU	10.8	Romanza	1996

Figure B.86.: Inner join of the resulting “relation\_test\_class” table and the one to one relations to the “XML\_CD\_Catalog\_Test” table it references.

### B.2.16. createOneToManyRelation

Routine: Routine #3 for 'testing.RelationTestClass' in parser 'XML' ▼

Methodname: createOneToManyRelation ▼

Submit selected

Sequence number

☒ Repetitive procedure

Notable parsing objects: Please select an entry ▼ property value

add objectpair

Method parameters: relations ▼ testing.XML\_CD\_Catal ▼

company ▼ COMPANY ▼

Figure B.87.: Configuration of the “createRelation” procedure.

Almost identical to the previously displayed “createRelation” procedure is the “createOneToManyRelation” transformation method. The difference here being that it allows to create a relation from one object to many other objects, instead of just one-to-one. This procedure too builds on the domain class introduced in the “setTimestamp” section, and is also put into the same routine before the load. The example here is rather nonsensical, basically the same companies are grouped together via their “RelationTestClass” object. All “CDs” of a certain “company” will belong to the same “RelationTestClass”, meaning there is a one-to-many relation from “RelationTestClass” to “XML\_CD\_Catalog\_Test”.

relation_test_class_id	artist	company	country	price	title	year
1	Will Smith	Columbia_Corporation	USA	9.9	Big Willie style	1997
1	Bob Dylan	Columbia_Corporation	USA	10.9	Empire Burlesque	1985
2	Bonnie Tyler	CBS Records_Corporation	UK	9.9	Hide your heart	1988
3	Dolly Parton	RCA_Corporation	USA	9.9	Greatest Hits	1982
4	Gary Moore	Virgin records_Corporation	UK	10.2	Still got the blues	1990
5	Eros Ramazzotti	BMG_Corporation	EU	9.9	Eros	1997
6	Newcomer: Van Morrison	Innovating Polydor_Corporation	UK	8.2	Tupelo Honey	1971
6	Newcomer: Andrea Bocelli	Innovating Polydor_Corporation	EU	10.8	Romanza	1996
6	Newcomer: Bee Gees	Innovating Polydor_Corporation	UK	10.9	One night only	1998
7	Dr.Hook	CBS_Corporation	UK	8.1	Sylvias Mother	1973
8	Rod Stewart	Pickwick_Corporation	UK	8.5	Maggie May	1990

Figure B.88.: Inner join of the resulting “relation\_test.class” table and the one to many relations to the “XML\_CD\_Catalog\_Test” table.

### B.2.17. cacheInfoForCrossProcedure

There are no illustrations for the “cacheInfoForCrossProcedure” per se, as this transformation procedure is used to cache a peculiar datum and create a new temporary procedure with said datum as parameter. Therefore, this section is split in two subsections, with each illustrating one of the two possible submethods, or as they are generally called in this document, cross methods.

Both use cases build upon the “cd\_catalog” file, its generated pseudo-object maps, and the “RelationTestClass” domain class/database table. Further, all previous routines and procedures described in this chapter, which are applied for said pseudo-objects, were still in effect before both cross methods and had to be taken into account here. Keep in mind that “cacheInfoForCrossProcedure” can only cache a single datum, which is achieved via the “notable objects” mechanism.

#### B.2.17.1. crossSetValue

In this illustration, a realization came to be that Europe is in dire need of more “Joe Cocker”. To alleviate this problem, a decree was given that whenever a datum contains “EU” as its country, “Joe Cocker” will be set in the “artist” property of the now extended “RelationTestClass” instances. This selective behaviour is achieved with the “notable objects” mechanism of the cross method itself. The name “Joe Cocker” was used to find the datum that has to be cached, which is also the value set in the new “artist” properties.

The screenshot shows a web-based configuration form for a cross-procedure. The form is organized into several sections:

- Routine:** A dropdown menu showing "Routine #3 for 'testing.RelationTestClass' in parser 'XML'".
- Methodname:** A dropdown menu showing "cacheInfoForCrossProcedure".
- Crossmethod:** A dropdown menu showing "crossSetValue".
- Submit selected:** An orange button.
- Sequence number:** A text input field.
- Repetitive procedure:** A checkbox that is currently unchecked.
- Notable parsing objects:** A section with a dropdown menu showing "ARTIST" and a text input field containing "Joe Cocker".
- add objectpair:** An orange button.
- Method parameters:** A section with a checkbox for "Repetitive cross-procedure" that is unchecked.
- Notable parsing objects for cross-procedure:** A section with a dropdown menu showing "COUNTRY" and a text input field containing "EU".
- add objectpair for cross-procedure:** An orange button.
- artist:** A dropdown menu showing "ARTIST".

Figure B.89.: Configuration of the “crossSetValue” procedure.

## Appendix B. Use cases and examples

```
package testing

import java.sql.Timestamp

class RelationTestClass {

    XML_CD_Catalog_Test relation
    Timestamp timestamp
    String artist

    static hasMany = [relations: XML_CD_Catalog_Test]

    static constraints = {
        relation nullable: true
        timestamp nullable: true
        artist nullable: true
    }
}
```

Figure B.90.: The extended “relation\_test.class” domain class.

id	version	artist	relation_id	timestamp	artist	company	country
1	0	NULL	26	2019-01-20 18:07:04	Bob Dylan	Columbia_Corporation	USA
2	0	NULL	26	2019-01-20 18:07:04	Bonnie Tyler	CBS Records_Corporation	UK
3	0	NULL	26	2019-01-20 18:07:04	Dolly Parton	RCA_Corporation	USA
4	0	NULL	26	2019-01-20 18:07:04	Gary Moore	Virgin records_Corporation	UK
5	0	Joe Cocker	26	2019-01-20 18:07:04	Eros Ramazzotti	BMG_Corporation	EU
6	0	NULL	26	2019-01-20 18:07:04	Newcomer: Bee Gees	Innovating Polydor_Corporation	UK
7	0	NULL	26	2019-01-20 18:07:04	Dr.Hook	CBS_Corporation	UK
8	0	NULL	26	2019-01-20 18:07:04	Rod Stewart	Pickwick_Corporation	UK
9	0	Joe Cocker	26	2019-01-20 18:07:04	Newcomer: Andrea Bocelli	Innovating Polydor_Corporation	EU
10	0	NULL	26	2019-01-20 18:07:04	Percy Sledge	Atlantic_Corporation	USA
11	0	Joe Cocker	26	2019-01-20 18:07:04	Savage Rose	Mega_Corporation	EU

Figure B.91.: The resulting “relation\_test.class” table and the indirectly related “XML\_CD\_Catalog\_Test” table.

Note that there is no direct association between the tables above. The “XML\_CD\_Catalog\_Test” table is shown instead of the pseudo-object maps, because they are mostly identical. The right table is used as an indication when the value “EU” appeared in the datum, which is when “RelationTest-Class” has gained a “Joe Cocker”.

## B.2. Transformation examples

### B.2.17.2. crossCreateRelation

Here, another silly example is given that recognizes “Joe Cocker” from “EMI\_Corporation” as the greatest artist of his time, and that every “RelationTestClass” object must reference his “CD”, which is an “XML\_CD\_Catalog\_Test” instance. This procedure overrides the [previous procedure’s](#) result, in which each “RelationTestClass” object got a relation referencing an “XML\_CD\_Catalog\_Test” object. Here, the “Joe cocker” “CD” datum is found via the unique value “EMI\_Corporation” in the company column. Such can be done, as his company’s and artist’s name are both unique for this table. Similarly, the object is also retrieved from the database by its cross method via its unique “company” property. Note that “\_Corporation” was appended to “EMI” in a previous “appendString” transformation.

Routine: Routine #3 for 'testing.RelationTestClass' in parser 'XML' ▼

Methodname: cacheInfoForCrossProcedure ▼

Crossmethod: crossCreateRelation ▼

Submit selected

Sequence number

☐ Repetitive procedure

Notable parsing objects: COMPANY ▼ EMI\_Corporation

add objectpair

Method parameters:

☒ Repetitive cross-procedure

Notable parsing objects for cross-procedure: Please select an entry ▼ property value

add objectpair for cross-procedure

relation ▼ testing.XML\_CD\_Catal ▼

company ▼ COMPANY ▼

Figure B.92.: Configuration of the “cross-CreateRelation” procedure.

relation_test_class_id	artist	company	country	price	title	year
1	Joe Cocker	EMI_Corporation	USA	8.2	Unchain my heart	1987
2	Joe Cocker	EMI_Corporation	USA	8.2	Unchain my heart	1987
3	Joe Cocker	EMI_Corporation	USA	8.2	Unchain my heart	1987
4	Joe Cocker	EMI_Corporation	USA	8.2	Unchain my heart	1987
5	Joe Cocker	EMI_Corporation	USA	8.2	Unchain my heart	1987
6	Joe Cocker	EMI_Corporation	USA	8.2	Unchain my heart	1987
7	Joe Cocker	EMI_Corporation	USA	8.2	Unchain my heart	1987

Figure B.93.: Inner join of the resulting “relation\_test\_class” table and the “XML\_CD\_Catalog\_Test” table it references.