Program Description COMA CE

Database Chair at the Institute of Computer Science,
University of Leipzig

Contents

1 Introduction

1.1 The COMA CE

The COMA CE is the community edition of the COMA 3.0 project and contains all classes and libraries that enable the user to load schemas and auxiliary information, select a predefined workflow and perform schema matching. The result can be saved to a text file and preprocessed by further COMA modules, provided by the COMA Business Edition, or different programs.

In detail, the community edition comprises the following features:

- Parsers that load schemas, instances and mappings.
- A graph structure to represent schemas, match results and mappings.
- A variety of matchers.
- Workflow managers that combine matchers to a match workflow.
- Writers to save mappings in a text file (for the sake of mapping processing, mapping enhancement and any further tasks).
- A GUI to easily load schemas, carry out match processes and view the match result.

The matchers itself are located in an external library and are not part of the open source project as such. However, they can be carried out without any limitations.

1.2 Contributions

This document is dedicated to developers and analysts who intend to get a deeper insight into the COMA project, and especially to those who are going to extend or adjust it. Next to a description of the project and its several sub-projects, as well as its general working principle, this manual explains how COMA can be extended by new features and functionality. Thus, the document at hand should enable any participant to get both a quick overview and a solid understanding about the program, and to perform basic program adjustments and extensions.

The following extensions are profoundly described in this document:

- Adding a new parser (e.g., a new schema parser to support further database schema types).
- Adding a new workflow (e.g., to allow an individual workflow for specific matching tasks).
- Adding a new matcher (e.g., to develop or link a new matcher COMA does not offer yet).

1.3 Document Structure

After this introduction, the second chapter explains the general project structure, mainly focusing on the sub-projects and main packages, its meaning and its dependencies. There is also an overview about the program constants in COMA, which are widely used across the project, and thus have a large importance. The third chapter explains the internal representation of schemas and match results as a graph object (*structure module*). The fourth chapter concentrates on the matching process, the workflow and the grammar by which the workflow is defined (*matching module*). The fifth chapter explains the modules *insert* and *export*, so rather focuses on the parsers (schema,

instance and relational parsers). The sixth chapter briefly introduces the COMA GUI.

While these first chapters rather explain and describe the project itself and the structure of seven modules, the 7th chapter is more practically oriented, and explains how the program can be extended or adjusted by the user, so does not focus on describing aspects anymore. It mainly elucidates how COMA can be launched and a workflow can be executed, and how new parsers, matchers and workflows can be added to the program.

Therefore, the 7th chapter might be the most interesting one for developers resp. for COMA extension tasks. It also draws references to previous sections, so that it is possible to solely read the specific sub-sections of chapter 7 that refer to your aims and tasks, and if necessary to look up the further information in the previous chapters. The second chapter is rather important, though, because it gives an general overview about the project. It is therefore recommended to read this chapter first, before you concentrate on the part referring to your aims.

The 8th chapter contains some specific information about COMA CE components, including the database and database connection.

1.4 Acknowledgment

We tried to make this program description as thorough and precise as possible, but due to constant changes and improvements on the COMA project, as well as a large amount of none-documented source code, we cannot warrant total correctness or completeness. Also, we cannot afford to permanently update this manual, nor to answer all questions and comments about COMA. You have to use COMA on your own risk, and we hereby exclude any liability from using the software.

Special thanks to Sabine Maßmann and Salvatore Raunich, who helped much in the documentation and deployment of COMA CE.

Patrick Arnold in March 2012
University of Leipzig, Germany

2 Program Structure

2.1 Introduction

In this section, the basics of the program structure resp. program architecture are explained. It is recommended to any developer who intends to extend or adjust the program or get a deeper insight into the program architecture.

2.2 Overview

COMA CE is a Maven/Eclipse project that consists of 2 sub-projects: coma-gui and coma-engine. In coma-engine, all relevant classes of COMA CE are located which are used to parse schemas and carry out the match process. In coma-gui, only the GUI-related classes are located.

Each sub-project is an independent Maven project under the patronage of the main project coma-ce. According to the Maven specification, a sub-project has normally the following elements:

- A source folder, containing all java classes (src/main/java).
- A test folder, containing all java test classes (src/main/test).
- A reference to the JRE library.
- A reference to the Maven library.
- A source and target folder (always empty).
- The pom.xml.

The pom.xml is used to configure the entire sub-project. Since changes might have severe impact on the project, changing something in the pom.xml should always be performed carefully.

2.3 Modules

The COMA CE is divided in 8 *modules*. The sub-project coma-engine consists of 7 modules, which are the top-level packages in the package hierarchy. The sub-project coma-gui is a module itself.

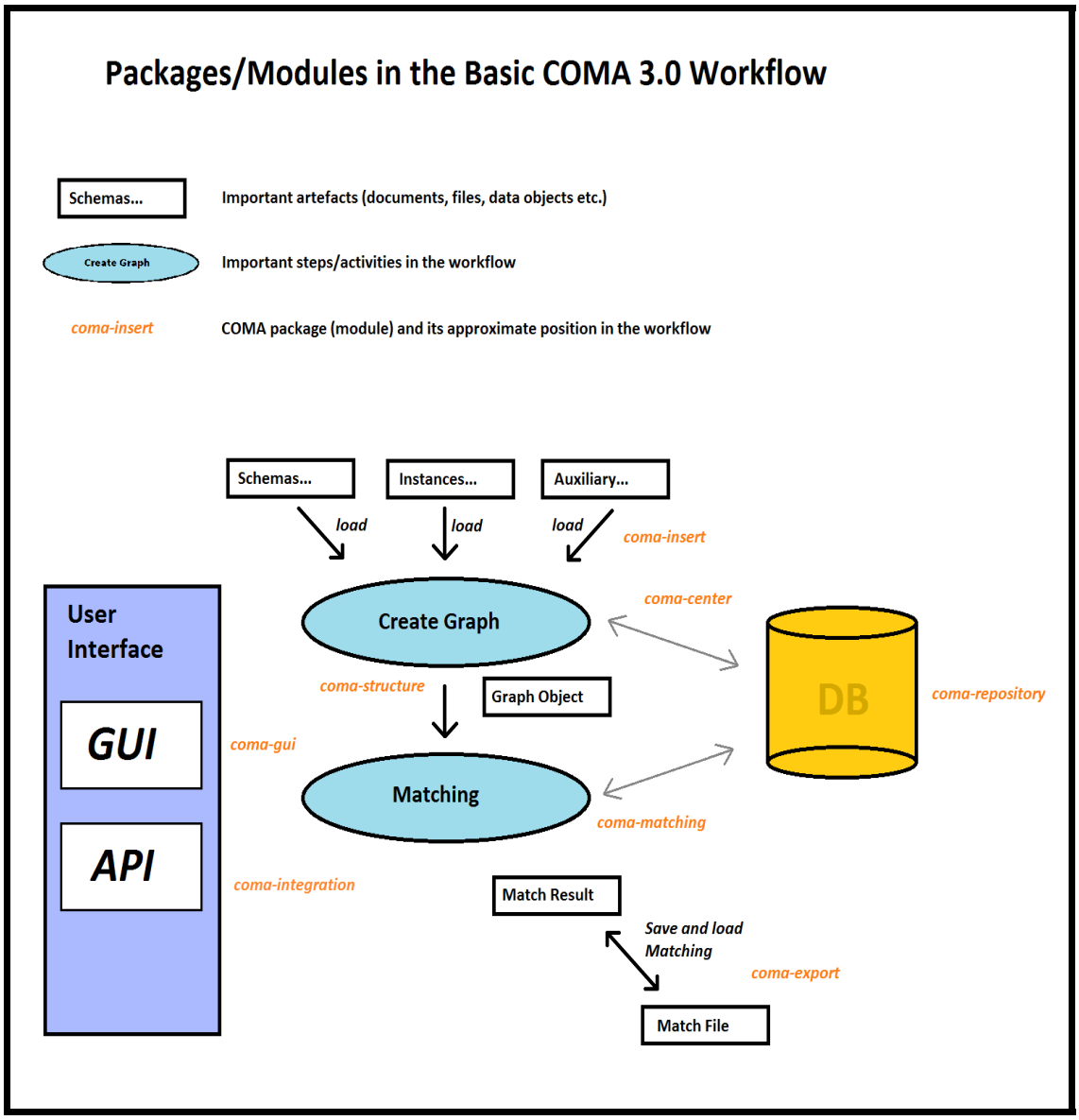
Each module concentrates on a specific part or task in COMA. The entire structuring is rather workflow-oriented (parsing, matching, exporting etc.) than tier-oriented (database, entities, manager, controller, gui). Either way, a strong interdependence between the sub-projects is the result.

The 8 modules are:

- coma-center
- coma-export
- coma-gui
- coma-insert
- coma-integration
- coma-matching
- coma-repository
- coma-structure

The names of the modules origin from former autonomous projects, which were later combined to the two projects coma-engine and coma-gui. However, the former names will be still used in this documentation.

2.4 Module Description



The module coma-center contains basic classes, e.g., classes to automatically configure the workflow and classes to preprocess graphs. It can be seen as a "container" where some general classes are located which does not fit so well into other modules.

The module coma-export is rather little. It only comprises the classes needed to export a mapping or match result, so to offer respective writers. The result is normally stored in a text file, using a COMA-specific format. It can be loaded at a later time again.

The module coma-insert focuses on the parsers needed to load the different schema types. For each file format (e.g., xsd, csv, excel, sql, owl etc.) there exists one parser that loads the schema into an internal data structure. Besides the schema parsers, there are also instance parsers to load instance data, as well as parsers to load relationships between schemas (relationship parsers like match result parser, RDF parser etc.).

To allow easy access to the COMA functions there exists an API class offering several methods to carry out tasks without using the GUI. This single class makes up the module coma-integration.

The center of interest is the module coma-matching, where all workflow strategies are located and the schema matching is performed. There are also different classes to represent the matching workflow and to exploit the workflow grammar. As already explained, the matcher classes itself are not part of the project, but located in an external library.

The module coma-repository refers to the database, in which schemas and match results are stored. The module contains only few classes, which focus on database operations (initialization, adding or fetching data objects etc.).

The module coma-structure is used to internally represent the schemas, instances and mappings. The graph classes and match result classes are the most important classes thereof.

2.5 Constants

Constants are widely used in COMA, also across the several modules. There is no separate class or module that stores all constants yet; instead, the constants are normally declared in the classes that use them most frequently, or to which they belong most likely. Constants are invariably declared as public static final variables at the beginning of a class.

In this section, an overview about the constants is given. The table below itemizes the java classes where important constants are defined and especially what they define. Constants are considered important if they are used by any other class from where they are defined. Blue highlighting marks especially important aspects.

coma-insert	
InsertParser	Defines the available input schema formats, e.g. CSV, XSD etc. and the default source schema directory. Also parses states are defined.
ODBCParser	Defines the DB url for the ODBC parser.
coma-matching	
Combination	Defines the similarity combinations and set combinations, as well as some strategy constants to compute the set combinations (“RESULT_...”).
ComplexMatcher	Defines the complex matchers and some previous strategies like PARENTS, CHILDREN etc. Also defines a list containing all complex matchers.
Constants	Defines the names of the workflow hierarchy, like “workflow”, “strategy”, “matcher” etc. Also defines the separator for workflows.
IDOverview	Defines the ID range for other constant groups, e.g., defines that measure constants have ids between 1000 and 1999, matchers between 2000 and 2999 etc.
Matcher	Defines the matchers and a list containing all matchers.

Resolution	Defines the resolutions (RES1_..., RES2_... and RES3_...), as well as a list of resolutions for each resolution type.
Selection	Defines match directions and different kind of selections, as well as a list of selections.
SimilarityMeasure	Defines the several similarity measures and a list containing all measures. Also defines the default threshold.
Strategy	Defines the strategies and a list containing all strategies.
Workflow	Defines the workflows and a list containing all workflows. Also defines default strategies, complex matchers and matchers.
coma-repository	
MySQL	Defines the several SQL statements to access and use the DB.
Repository	Defines basic DB parameters, e.g., the name of tables, maximal instances per element and status and abbreviation names.
coma-structure	
DirectedGraphImpl	Defines basic graph elements, like ROOT, PARENTS, CHILDREN, LEAF etc.
Element	Defines the several element types, like element, global element etc.
Graph	Defines the preprocessing states of a graph (loaded, resolved, reduced etc.), default states and constants regarding all elements of a specific element type (also see “Element”).
MatchResult	Defines match-specific similarity thresholds, match result operations (like intersect, merge, compose), a list of those operations, compositions (like min, max, average, sum) and further constants used in the match result context.
Source	Defines the several source types allowed in COMA, e.g., XSD, Website, OntoBuilder, CSV etc., as well as a list of these types.
SourceRelationship	Defines the several relationship types like mapping, match result etc., as well as a list of them.

2.6 COMA CE Modules Extension

	#Classes	#LOF
coma-center	7	1426
coma-export	3	257
coma-gui	78	15671
coma-insert	37	6140
coma-integration	1	189
coma-matching	33	9495
coma-repository	4	2265
coma-structure	16	4217
total	**179**	**39660**
Coma-engine only	**101**	**23989**

We used the eclipse add-on Metrics 1.3.6 to measure the size of the several modules. The specifications are as of November 21, 2011, and refer to all files which are located in the source folders of the modules (so without any test classes or additional configuration files). The LOF number refers to the number of actual code lines, i.e., does not include any kind of comments or out-commented code.

3 Data Structure

3.1 Introduction

In this section, the data structure is explained, mainly the graph used to represent schemas and the match result to represent a list of correspondences. Since the data structure is used in the entire program, it is recommended for any developer to get acquainted with it before starting to extend or refine COMA.

3.2 Overview

There are two ways how relevant objects (schemas, instances, match results etc.) are represented in COMA: In the database and in the main memory (in this case, a complex graph structure is used). The database and database classes are united under the term “repository”.

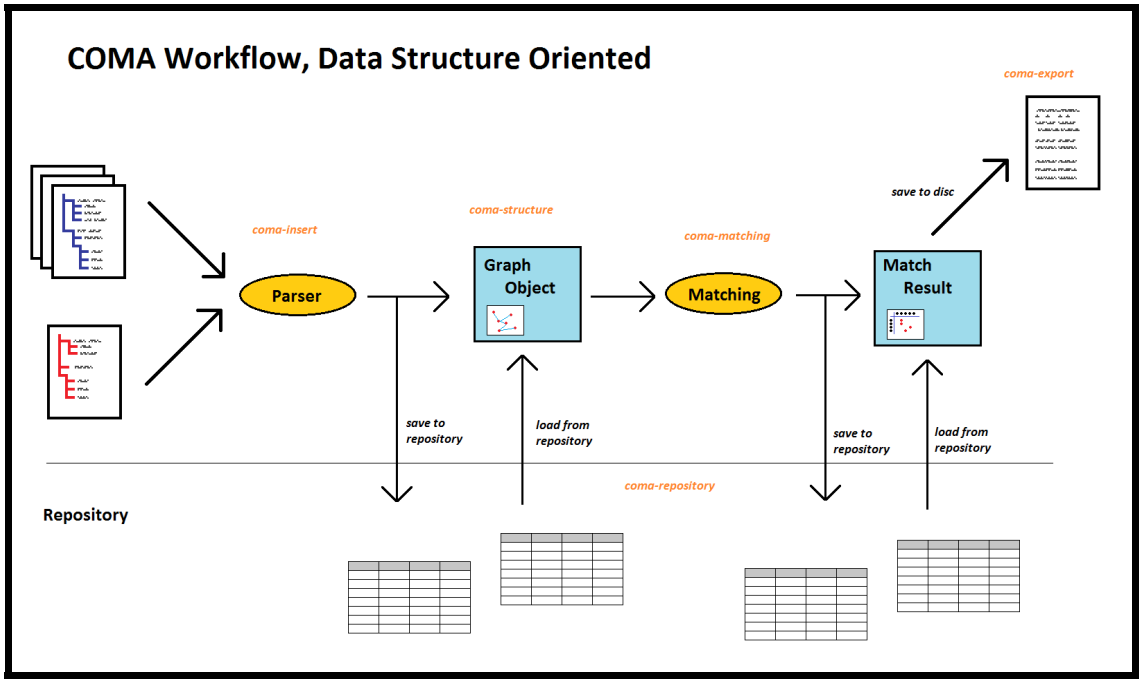
Classes which have to deal with the database are mainly located in the module coma-repository, and to a less degree in coma-center. Classes which have to deal with the internal structure (objects in the main memory), are located in the module coma-structure, which will be the center of interest in this chapter.

Before any schema matching can be performed, the relevant objects have to be loaded into the program. For this, a parser will convert the file (e.g., an xsd schema) into the internal structure. The parsers are located in the module coma-insert, but the result is a data structure, belonging to the module coma-

structure. The data structure (e.g., graph) can be added to the database (repository). This way, it can be reloaded and used at a later time without loading it again (whereas the graph object gets destroyed when the program is quit).

In COMA, schemas are represented as graphs, which can be seen as a set of nodes and as set of edges in between. In the database, they are stored in a slightly different way. There is chiefly a table for elements and a table for element relations. Methods allow to load a graph object into the database, and to create a graph object out of the database, if necessary.

Most program features can be executed with the graph object, so does not necessarily need the database. Thus, the database is rather intended to save match results, schemas and others, and to load it again at a later time.



The picture above describes the usage of graphs and the repository in the default COMA workflow. Starting with a source schema, a target schema and perhaps some additional information, parsers are used to create a graph object

of the map (source schema + target schema) and to save this graph in a relational database (repository). The graph object is used to perform the matching tasks, which results in a match result (a table containing the matches between source and target schema). This match result can be stored in the database as well, and may be additionally saved to disk by the user (coma-export).

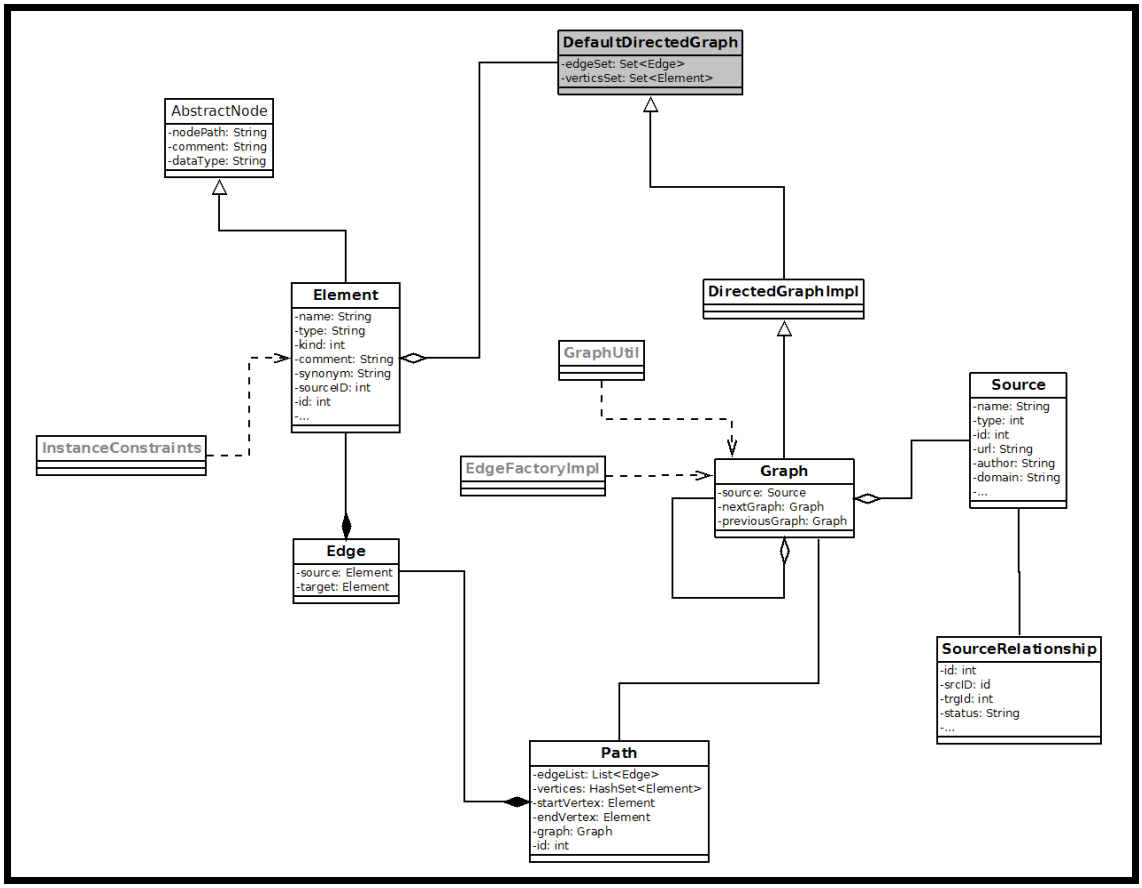
At a later time, the mappings and match results can be loaded from the database again, for instance to allow a mapping refinement, renewed matching etc.

3.3 The Graph

3.3.1 Overview

The graph is used to represent a schema, and for this is widely used by the program, especially by coma-matching. It contains a list of nodes and a list of edges between these nodes. Almost any class in the coma-structure module is used to build and manage the graph structure; the only few classes which are not directly related to a graph are the match result classes and some few subsidiary classes which are not used at the moment (like Statistics.java).

The graph structure is depicted in the chart below. It will be explained in the next sections in more detail. For now, only the class diagram and basics will be elucidated.



In addition to the classic UML specification, there are two special elements in this class diagram. The gray class at the top means that this class is not part of coma-structure module anymore (nor of COMA either), but was included for a better understanding and to complete the chart. The dashed lines express an association which has nothing to do with the structure in general; the classes being connected with it (those having a gray headlines) are auxiliary classes. Originally, it was intended to leave them out at all, but to get a more complete overview, they are now eventually included.

In order to get a clearer view, the class methods are omitted in this chart.

3.3.2 The Graph Classes

The central class in the coma-structure module is Graph.java. It is derived by the superior class DirectedGraphImpl.java, which provides many important graph-specific methods, e.g., methods to get the edge set and the vertex set, as well as to get all kind of paths in the graph (e.g., all leave paths, root paths, inner paths etc.). Unfortunately, the several methods used for the graph object are mixed up between those two classes, so it may sometimes occur that they are either in the first or in the second class.

DirectedGraphImpl.java is derived by DefaultDirectedGraph.java, which is a class of the jGraphT library, and thus is situated outside the project. This class provides a directed graph having edges and vertexes.

A graph as such has a *source* reference, specifying to which *source* (e.g., source schema) it belongs, and a *previousGraph* and *nextGraph* variable. Besides this, it inherits the *edge set* and *vertexes set* of the DefaultDirectedGraph.java class. These two variables are the most important ones, because in graph theory a graph can be exactly represented by the only specification of nodes and edges. The variables *previousGraph* and *nextGraph* are only used to accesses a previous or further graph state (there exist several possible graph states in COMA, e.g., preloaded, reduced, etc.).

GraphUtil.java is an auxiliary class providing further graph functions. It is rather at the edge of graphs and match results. EdgeFactoryImpl.java has the only (yet important) task to create a new edge for a graph path. It is actually of more importance for Edge.java (described below), yet cannot be directly accessed by the latter.

3.3.3 Edges and Elements

An element is the atomic object in a graph and is represented by the class Element.java. It is derived by the abstract class AbstractNode.java, which is only used for further coma sub-projects, some having particular claims on element specifications. An abstract node has a *comment*, a *data type* and a *uniqueNameRepresentation*, the latter specifying the entire path of a node (e.g., DB_STUD.Student.Name). It is not necessary for the coma structure, though.

There are several information attached to an element, e.g., its name, type, comment etc. The class does not offer any methods to get the entire path of the element, but since it is now derived by AbstractNode.java, the method *getUniqueNodeRepresentation()* does just this.

An edge is represented by Edge.java and simply consists of two nodes. The class EdgeFactoryImpl.java, as already mentioned, is used to create a new edge.

Path.java defines a path, which is a list of edges and has an *edge list* as its most important variable. Besides this, there is also a *node set* (implemented as hash set) to allow a faster processing of paths, a *start vertex* and *end vertex* as separate parameters, as well as a *graph* reference.

The path class offers various path-specific functions, for instance to get sub-paths or the last or first element of a path. It also offers methods to add edges to the path, for this using the edge factory.

Since a path is always part of a graph, yet not vice versa, the question might arise why a path needs the graph reference. Yet as mentioned before, EdgeFactoryImpl.java can be only accessed by a graph object and not by the path class alone.

3.3.4 Source and Source Relationship

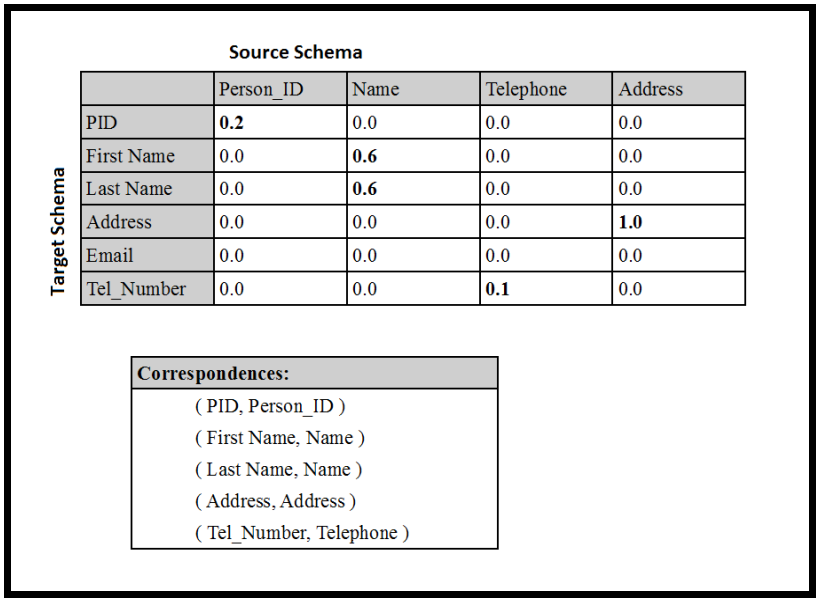
Source.java represents a schema, which can be a source schema or target schema. SourceRelation.java represents a relation between two schemas (source and target schema), thus representing a rough map without correspondences. Note that Source.java and SourceRelation.java are not directly connected, yet only by an *id* specification.

3.3.5 Loadings Schemas and Instances

Schemas are loaded by parser classes, which belong to the coma-insert module and are described in the respective chapter. Instances are loaded by instance parsers, which belong to this module as well. As it could be seen in the previous UML diagrams, instance data is attached to an element, i.e., to get instance data of a specific element, only the element object has to be fetched.

3.4 Match Result

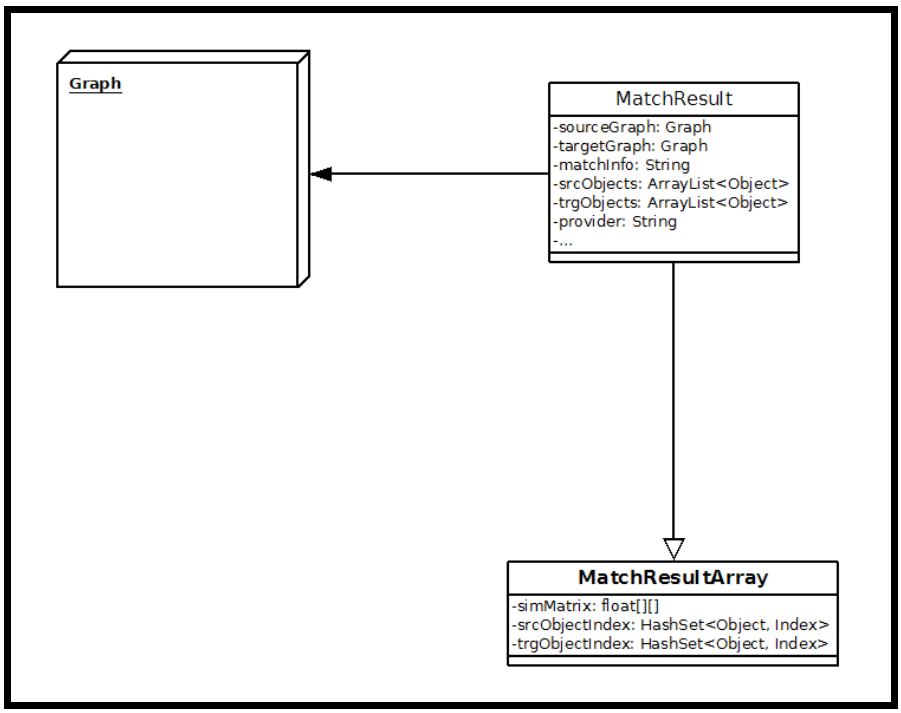
The match result is a matrix where the confidence between any source and target element is stored. As matter of fact, most cells normally contain zeros, meaning that there is no correspondence between those two elements. If a value is



larger than 0 (or, more precisely, larger than a predefined threshold), there is a correspondence between those two elements. Of course, COMA may detect false correspondences, so whether this is an actual correspondence is not certain then. The larger the confidence is (0 .. 1), the more likely the correspondence seems, though. The common threshold for accepting correspondences is 0.4 and has to be specified in the workflow (see the next chapter).

As depicted below, the match result consists of two important classes: MatchResultArray.java and its super class MatchResult.java (abstract). The graph structure, to which these classes are related, was already explained in the previous section.

The class MatchResult.java contains a *source graph* and a *target graph* as its most important attributes. These are inevitable for schema matching, of course. Besides, there is a list of source elements and target elements to allow a faster matching, and some further match-specific variables like *matchInfo*, where matching parameters are stored.



MatchResultArray.java contains *simMatrix* (in COMA called similarity matrix) as its main parameter. It is a float matrix and thus simply represents the match result matrix introduced above. Besides this, there are the variables *srcObjectIndex* and *trgObjectIndex*, which are hash sets to allow an even faster access to source and target elements, and thus to speed up the match process.

Therefore, MatchResult.java can be seen as the base class for the entire matching process, and MatchResultArray.java as the result of it. As it can be seen in the uml diagram, they are connected to the graph structure.

Either class offers several methods to manage and control the match process and match result. MatchResultArray.java concentrates on the match result, so offers functions to set and get the result, as well as some matching features. These include:

- Intersect: Returns the intersection of two different match results (e.g., if several matchers were used for a scenario, and only the correspondences are to be obtained which were detected by all matchers).
- Diff: Returns the matches that occur in a match result A, but do not occur in a match result B.
- Merge/Compose: Merges all correspondences in two match results A and B (analogous to intersect, this can be seen as the union operation).
- Transpose: Swaps source and target elements and their values in a match result.

MatchResult.java rather offers functions for the general matching process. Next to the Getters and Setters there are many abstract methods which are overwritten by MatchResultArray.java, as well as methods concentrating on the graph (e.g., methods to get all source elements that correspond to a target element etc.). Unfortunately, some of the methods implemented in MatchResultArray.java are also implemented in MatchResult.java, so there is no strict separation between those two classes.

Since MatchResult.java contains a source graph and a target graph, it can be seen as the top-level data element in COMA. The constructor of MatchResult.java is called by many different classes in COMA CE, for instance by

the ExecWorkflow.java (the main class to execute a COMA workflow) or DataAccess.java (to load and write a match result from/into the DB).

3.5 Managing Graphs and Match Results

Of course, there must be classes in COMA that contain the graph objects so that a matcher can operate on them. In the current version of COMA CE, the class WorkFlow.java contains a source graph and target graph, this way representing a schema. Subsequently, the method *execute()* in ExecWorkFlow.java is called, which gets a workflow instance as input and returns a complete match result. The class that called the execution method then obtains this match result which can be used for further tasks.

The classes WorkFlow.java and ExecWorkFlow.java are located in the coma-matching module, and will be explained in more detail in the following chapter. Besides, the usage of this classes is also explained in the last chapter (“Getting Started”), especially if a new workflow is to be created or an existing one is to be adjusted.

4 Matching

4.1 Overview

Matching is the central aspect of the COMA CE project, and for this reason is a separate module (coma-matching). In this chapter, the basics of the matching process are to be explained. Hence, the following chapter is only of importance for developer who are interested in the matching process, e.g., who intend to extend or adjust it.

4.2 COMA Workflow Grammar

4.2.1 Introduction

COMA stands for combined matchers, and unlike classic schema matchers, it excels due to a large set of different matching strategies which can be exploited to carry out very sophisticated schema matching. If more than two matchers are used in a scenario (which is very likely), an order has to be determined in which they are carried out. However, some strategies can also be run in parallel, and their results be combined in different ways. Besides, there is a whole hierarchy of strategies and sub-strategies, sometimes making it pretty difficult to understand the interior of the matching process of COMA.

Altogether these many strategies and dependencies can be only managed by defining a workflow. The workflow can be seen as a unique artifact that tells

COMA how to carry out the entire match process. It might appear similar to a typical script or batch file and is quite structured.

A workflow could look as follows (strongly simplified):

> *Run Strategy 1 (run Strategy 1.1, then run Strategy 1.4 and combine the results using the merge combination), then run Strategy 3 (Run strategy 3.1 and 3.2, both in parallel, then create the intersection of their match result), then determine the maximum of the match results of Strategy 1 and 3.*

To express such (and even much more) complex workflows, a complete grammar was developed and integrated in COMA, which is able to express any possible workflow COMA may execute.

4.2.2 The Basics of the Grammar

The workflow is described by a grammar that determines exactly what a workflow must look like, and what kind of workflows are allowed. What a grammar is and in which way it is used cannot be explained at this point, nor can any detailed aspect be elucidated. The aims of this documentation are rather to give an overview how the workflow is composed so that it could be changed or extended if necessary. However, even with little grammar experience it should be feasible to understand the basics of the workflow without much difficulties.

The entire grammar is described in the single file ComaWorkFlow.g, which is situated in the coma-engine project, in the directory resources. It will be the centerpiece of this section.

A coma workflow is represented as a text string. This string tells COMA exactly how the match process has to be carried out. Due to the large amount of matchers in COMA, its various ways of combination, and the different combination strategies, such a string can become very large. A dummy example was already given above, which was merely added to explain the idea; however, a real workflow string rather looks like the following (this is still one of the simpler workflows):

```
((selfnode;(nametoken;trigram;set_average),
(statistics;featvect;set_average);weighted(0.7,0.3);set_average))
```

The grammar is used to check whether a given workflow fulfills the grammar's specification or not. This way, the grammar describes what kind of workflows are allowed. COMA will refuse any workflow specification that is not accepted by the grammar. This way, illegal workflows cannot be created and resulting errors are not possible.

4.2.3 The Structure of the WorkFlowComa.g

For a better understanding and adjustment, the grammar file is divided in four parts. Part II, which is the main part, is divided into sub-parts again. This structuring may help much in understanding the workflow grammar.

In Part I only some basic tokens are defined. For instance, it is defined what a brace is etc. (note that a brace in the workflow string must be already represented by a grammar token, because the brace character itself is used to define the grammar). Apart from this, Part I has no further meaning.

Part II defines the main parser rules and is the most important part in the entire grammar. This part defines chiefly what kind of workflows are allowed (resp. what rules a valid workflow must follow). The workflow hierarchy, explained in the next section, is defined in this very part as well.

Part III defines some additional parser rules, which are less important. The aspects described there are base elements of a workflow, which are needed by

the strategies and matchers, but which are not part of the actual workflow hierarchy.

Part IV defines the lexer rules, so the base elements in a workflow. While the parser rules only deal with variables, which have to be resolved (e.g., a workflow consists of at least one strategy, and a strategy consists of at least one complex matcher etc.), the lexer rules define an atomic item, which cannot be resolved any further (e.g., a result combination can be 'intersect', 'diff' or 'merge'). The lexer will not process these values any further, that is, it does not look for any rule for intersection, difference or merge, but it will check whether the values intersect, diff and merge are defined at all, and if not, immediately deny the workflow.

4.2.4 The Hierarchy in the Grammar

As it can be seen in the grammar, the workflow is the top-level hierarchy (do not consider the coma rule any further, it is simply used to test workflows). The hierarchy is defined in Part II, and will be the center of interest in this subsection.

This is the workflow rule:

```
CHAR_BRACE_LEFT
        (
                ( strategy  CHAR_SEMICOLON strategy (CHAR_SEMICOLON selection)?)
                |
                ( strategy (CHAR_COMMA strategy)+ (CHAR_SEMICOLON RESULT_COMBINATION)? )
                |
                (strategy)
                |
                (reuse)
        )
CHAR_BRACE_RIGHT;
```

The workflow rule explains that a workflow must start with a left brace and end with a right brace. It may contain one of the following elements (note that the symbol | stands for a logical or):

- A strategy, followed by a semicolon and another strategy, and optionally a further semicolon followed by a selection.
- A strategy, followed by at least one further semicolon and strategy (but maybe even more), and optionally a further semicolon followed by a result combination.
- A single strategy.
- A reuse specification.

Therefore, a possible workflow would be the following:

```
(strategy;strategy)
```

However, *strategy* is not an atomic token. It is a further rule, which has to be resolved by the grammar parser, so this workflow would be invalid of course, the parser would now run the strategy rule. This looks as follows:

```
CHAR_BRACE_LEFT
        ( RESOLUTION_1
        CHAR_SEMICOLON
        (
                (
                (complexMatcher ((CHAR_COMMA complexMatcher)+ CHAR_SEMICOLON
                        similarityCombination) )
                |
                (complexMatcher )
                )
        (CHAR_SEMICOLON selection)?
        )
        )
CHAR_BRACE_RIGHT;
```

As it can be seen, a strategy consists mainly of complex matchers (or just one complex matcher). A complex matcher is defined in the next rule and so forth. Thus, there exists a real hierarchy, at whose very bottom the tokens appear, e.g., the name of a matcher like 'trigram' (which is actually not a matcher in the grammar, but part of a matcher – it is called similarity measure).

The workflow hierarchy is the following:

- Workflow
- Strategy
- Complex Matcher
- Matcher

Normally, a matcher consists of one or more than one similarity measures. If more similarity measures are used, a similarity combination has to be specified to tell COMA how it should calculate the confidence out of the different results offered by the different matchers. Both similarity measure and similarity combination are atomic values, which are defined in part IV. They must be conform to one of the specified values, otherwise the parser declares the workflow as invalid.

Please note that the terms “matching strategy”, “matching algorithm” and “similarity measure” are quite the same, so a similarity measure is a matching strategy like trigram, n-token etc.

4.2.5 Resolutions

Next to the hierarchy elements, the resolution plays a very important part. As it can be seen in Part II, each strategy, complex matcher and matcher requires a resolution specification. A resolution is already an atomic element, and each of the three main hierarchy elements (strategy, complex matcher and matcher) uses a specific type of resolution: strategies require a resolution of type I, complex matchers a resolution of type II and matchers a resolution of type III.

To put it simply, a resolution specifies on which part of the schema graph the strategy or (complex) matcher concentrates and operates. In other words, the resolution defines the input the respective strategy or (complex) matcher gets. In many cases one would expect that a matcher gets a complete list of element names, which is quite true for string-based matchers (name matchers) and several other matchers. However, there are many further strategies in COMA which concentrate on different aspects. Some regard the node paths instead of the node names, or the parent nodes of a node. Some regard all inner paths or all leave paths, other check the siblings or children of nodes. There are many different resolutions after all, which are entirely defined in Part IV.

Thus, it can be seen in PART IV that Resolution 1 comprises rather general concepts like paths, leaf paths, inner nodes, nodes etc. Resolution 2 is rather similar and contains concepts like parents, siblings, children, selfnode, leaves. They are still rather general, but resolution 2 rather concentrates on traversing a graph, while resolution 1 rather concentrates on specific, yet immutable parts of a graph. Finally, Resolution 3 consists of less general and more node-specific concepts like node path, node name, node comments, node constraints, node synonyms etc. This is just reasonable, because a single matcher concentrates rather on specific nodes and node attributes, and does not explore the entire schema graph. On the contrary, a strategy containing many (complex) matchers does not focus on single nodes, but rather on all paths, inner nodes etc.

It can be seen that the 3 resolution types seem a little overlapping. This is quite true, and quite okay, because it was an important goal to make sure that each hierarchy has its own specified resolution, although some might use the same concepts of it.

It is absolutely necessary to distinguish between these resolutions if the workflow has to be changed. A matcher must not use a Resolution 1 or 2, just as a strategy must not use a Resolution 2 or 3.

4.2.6 Using the Grammar

The grammar file itself is of no use if a workflow is to be checked by COMA. For this, it has to be transformed into Java code so that COMA can check via a single method call whether a workflow at hand is valid or not.

The grammar was built with ANTLR 3.4, a tool to build and test grammars. It also contains a code generator which automatically creates two Java files: ComaWorkFlowParser.java and ComaWorkFlowLexer.java. These classes make it possible to fully use the defined grammar by a specific programming language like Java. They are both in the coma-matching module, in the sub-folder *validation*.

Adjusting the grammar and recompiling it will be explained in more details in chapter 6 (workflow adjustment).

4.3 Workflow

4.3.1 Overview

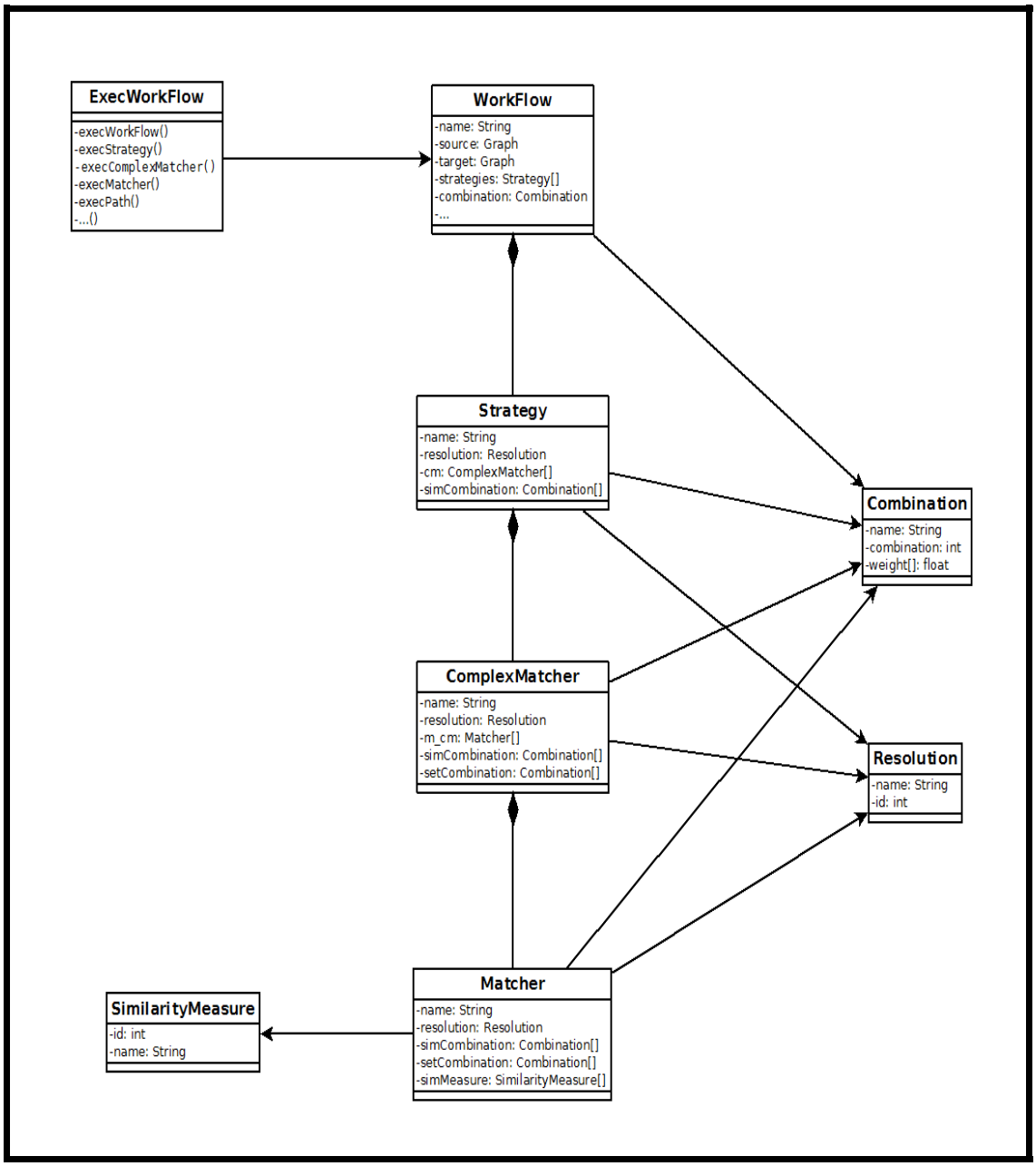
The workflow as described in the section before, is chiefly represented by two classes, which belong to the coma-matching module as well: WorkFlow.java and ExecWorkFlow.java (workflow execution). The first one is used to represent a workflow in the program – it is the same workflow described by the grammar, just in java code. ExecWorkFlow.java is used to execute a workflow, that is, to run the respective strategies and matchers and create the match result.

In this section, the basics of the workflow will be explained; the exact steps that are to be carried out to execute a workflow and get the match result will be explained in the 6th chapter.

4.3.2 WorkFlow.java

An instance of WorkFlow.java has a source and target graph and a list of strategies as its main attributes. Besides, there is a list of combinations which explains how the results produced by the strategies are combined in the final process.

As already pointed out, a strategy may consist of several complex matchers, which again consists of matchers. For each element there is a separate class, which can be seen in the UML class diagram below. At this point, it should become visible that this class structure is very similar to the grammar.



As it can be seen, each element in the workflow uses a resolution and at least one specification how the results of a matcher are to be combined. ComplexMatcher.java and Matcher.java have two kind of combinations, which at first glance might appear a little confusing. The *simCombination* (similarity combination) is only of importance if the matcher resp. complex matcher consists of more than one similarity measure resp. matcher. In this case,

simCombination explains how the result of the several similarity measures (resp. matchers) have to be combined. Possible specifications would be min or max, for instance, where min only considers the correspondences which occurred in all results (intersection) and max considers the correspondences which occurred in at least one result (union).

The *setCombination* explains the combination between different sets of elements. For instance, there could be several children of a node, which are included in the matching algorithm, and thus might cause different results, which have to be combined again.

4.3.3 ExecWorkFlow.java

The class ExecWorkFlow.java is used to execute an entire workflow and return the match result. For this, the class contains the central method execute*()*, which gets, as matter of fact, a workflow instance as input. Its result is a MatchResult.java instance.

Since executing a workflow is a rather complex process, the class obtains several sub-methods, which are called by *executeWorkFlow()* to obtain a partial result. Some examples are *executeStrategy()*, *executeComplexMatcher()*, *executeMatcher()*, *executePath()* etc. Of course, these sub-methods may also call each other; for instance, *executeStrategy()* gets the results from *executeComplexMatcher()* etc. Each of these methods returns a MatchResult.java instance, which at the end, are composed to the final match result.

The exact workflow is described in the next sub-section.

4.3.4 The Workflow Execution in Detail

Executing a workflow starts in *execute()* as described above. This method first gets the selected match result from the workflow. Of course, this should normally be null, because nothing has been matched so far. However, if merging or enhancing match results is intended, there is already a match result available.

execute()

```
MatchResult selected = workflow.getSelected();
if (selected!=null) {
        […]  // Calculate new match result and combine it with the already existing one
}
return executeWorkflow(workflow);   // Execute the workflow
```

If a match result is available, the if-block is executed, which performs a new match result and combines it with the already existing one. Otherwise *executeWorkflow()* is called directly.

The method executeWorkflow() is used to generate exactly one match result. It therefore gets a workflow and returns a match result. The method mainly checks several workflow cases, that is, it checks whether the workflow consists of only one strategy or more strategies. After this, always one of the following methods is called: *executeStrategy()* or *executePath()*. If the entire map is to be matched, *executeStrategy()* is directly called; if only parts of the map are matched (fragment matching), executePath() is called. The latter will call *executeStrategy()* later on.

executeWorkflow()

```
MatchResult selMR = createMRToUseSelected(begins[0], srcGraph, trgGraph,srcSelected, trgSelected);
if (selMR==null) {  // Entire map to matched (default case)
        result = executePath(srcGraph, trgGraph, begins[0], null, null, null);
} else {  // Fragment matching (special case)
        result = executeStrategy(srcGraph, trgGraph, begins[0], selMR);
 }
```

We do not regard *executePath()* any further, but skip right to executeStrategy(). This method mainly extracts the complex matchers the strategy contains, and then calls *executeComplexMatcher()* for each complex matcher. Each call returns a match result, so that *executeStrategy()* has to deal with a list of match results. They are subsequently combined to one match result, which is then returned to *executeWorkflow()*.

executeStrategy() - SIMPLIFIED!

```
ComplexMatcher[] match_strat = strategy.getComplexMatcher();
MatchResult[] results = new MatchResult[match_strat.length];

for (int i = 0; i < match_strat.length; i++) {
        ComplexMatcher cm = match_strat[i];
        String idString = cm.toString(false);
        results[i] = executeComplexMatcher( srcGraph, trgGraph, cm, srcObjects, trgObjects);
        resultCS.put(idString, results[i]);
}
```

The method executeComplexMatcher() is rather similar to *executeStrategy()*, it chiefly extracts the matcher of a complex matcher, checks whether it is really a matcher and not a complex matcher again, and then calls *executeMatchMemory()*. This is the main method that executes a single matcher, e.g., EditDistance. Of course, this method returns a match result.

executeComplexMatcher() - SIMPLIFIED!

```
Object[] c_cm = comatcher.getMatcherAndComplexMatcher();
MatchResult[] results = new MatchResult[c_cm.length];
for (int i = 0; i < c_cm.length; i++) {
        Object current = c_cm[i];
        if (current instanceof ComplexMatcher){
                results[i] = executeComplexMatcher(srcGraph, trgGraph, (ComplexMatcher)current,
                                srcSingleObjects, trgSingleObjects);
                } else if (current instanceof Matcher){
                        MatchResult resultsTmp2 = executeMatcherMemory(srcGraph, trgGraph,
                                (Matcher) current, srcSingleObjects, trgSingleObjects);
                        results[i] = resultsTmp2;
                }
        }
}
```

The method executeMatchMemory() does some preparations, for instance preparations for the synonyms, and then extracts the similarity measures of the method. For each similarity measure the method *executeMemory()* is called to get the match result. If more than one similarity measure is used, the match results are combined afterward.

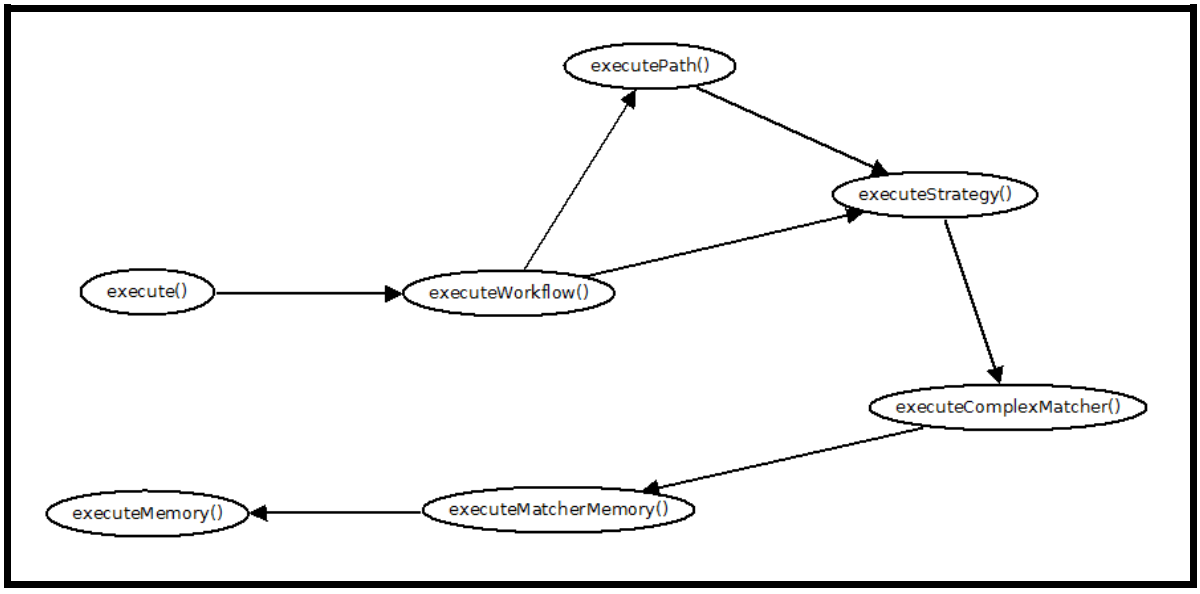
executeMatchMemory() - SIMPLIFIED!

```
SimilarityMeasure[] simmeas = matcher.getSimMeasures();
MatchResult[] results = new MatchResult[simmeas.length];

for (int i = 0; i < simmeas.length; i++) {
        results[i] = executeMemory(srcGraph, trgGraph, simmeas[i], srcSingleSet, trgSingleSet, oipA, oipB);
}
```

The method executeMemory() is the bottom of the workflow and will be described in the next section, now focusing on the actual matchers and not on the workflow any longer.

The following picture condenses the execution process again. It shows what methods are called until *executeMemory()* calculates the match result of a single matcher. Remember that each method returns a match result!



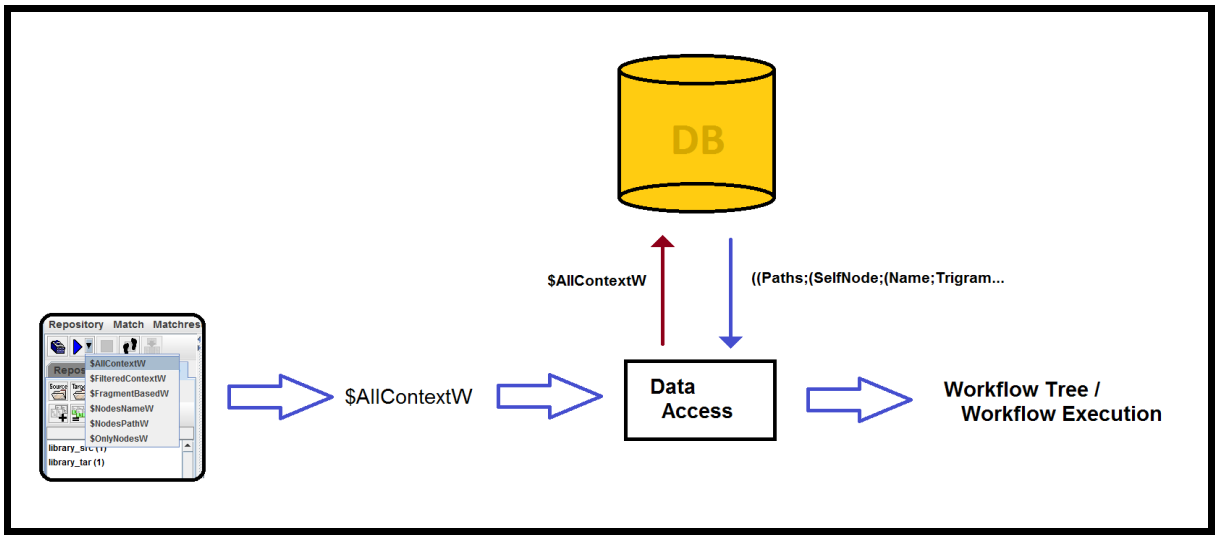
4.3.5 Predefined Workflow Management

There are already some predefined workflows in COMA++ (like $AllContextW etc.). They reside in the database and are obtained when the user runs COMA using such a predefined workflow (in COMA CE, only predefined workflows can be executed).

In *ExecuteMatchingThread.executeMatching()* it can be seen how predefined workflows are handled. When the user selects one of the predefined workflows, the workflow id (a string, e.g., "$AllContextW") is set in the variable *workflow*. Now, in *executeMatching()*, the data accessor converts this id into the actual workflow string, which is conform to the COMA workflow grammar. After this is done, this grammar string is transformed into a tree object, and the tree objects again is translated into a workflow object. This object can now be executed. Next to the code snippet, the picture below visualizes this process once more.

executeMatching()

```
System.out.println("Execute " + workflow);
String value = accessor.getWorkflowVariable(workflow);
System.out.println(value);
String workflowValue = accessor.getWorkflowVariableWithoutVariables(workflow);    // Conversion String - WF
System.out.println("build tree from: " + workflowValue);
Tree tree = TreeToWorkflow.getTree(workflowValue);        // The workflow as tree
Workflow w = TreeToWorkflow.buildWorkflow(tree);          // The workflow object (ready to be executed)
```



4.4 Matcher

4.4.1 Introduction

The matchers are the centerpiece of every workflow, and actually the centerpiece of the entire COMA project. They are not part of the COMA CE source code and are located in a separate library, which is connected to the program.

Thus, to add a new matcher, it is recommended to link your own library to COMA, or to create a separate class containing the matcher (in a separate folder of coma-matching).

4.4.2 Overview

A matcher in COMA is accessed via the similarity measure, which is formally defined in the class SimilarityMeasure.java. SimilarityMeasure.java has a constructor which gets an id as its only parameter (the id specifies which matcher has to be executed). Then, the prime method *getMatcher()* can be called, which returns the matcher according to the id. This matcher is an instance of IAttributeObjectMatcher.java, which is not part of the COMA CE project, but is located in the matcher library.

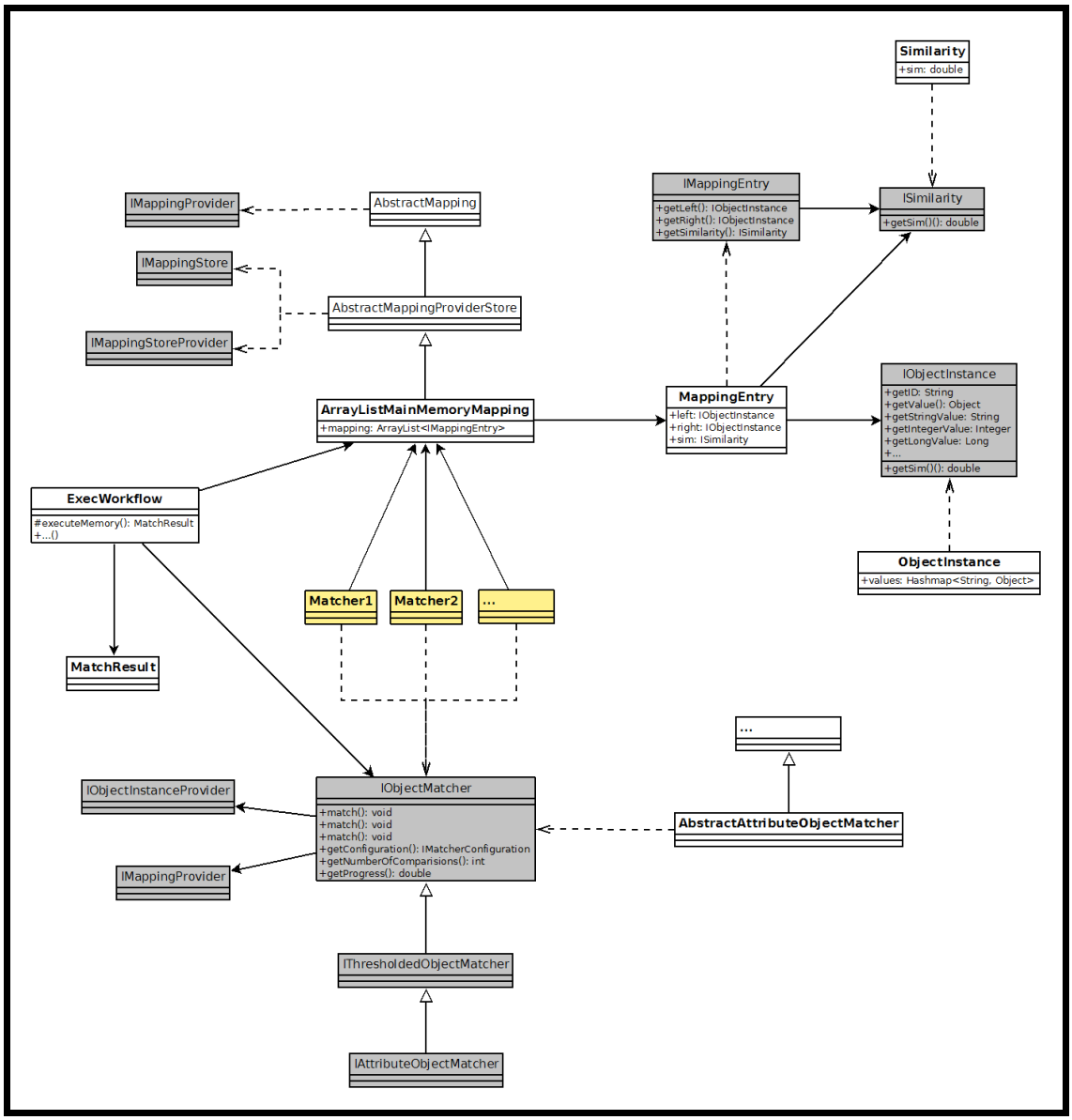
Therefore, a matcher must always implement the interface IAttributeObjectMatcher.java (which is a specification of IObjectMatcher.java) and override the required methods. These include 3 different match methods, having different method specifications. Each of these methods gets an ImappingStore.java instance and either an IMappingProvider.java, or one or two

IObjectInstanceProvider.java instances. The matcher has to be implemented in at least one of these methods.

The class ExecWorkFlow.java contains the method *executeMemory()*, which is the main method to execute a matcher in the main memory. The method gets the matcher object from the SimilarityMeasure.java instance, and then executes the overriden method *match()* of whatever matcher has been returned. This method also gets a variable *mapping*, which is an instance of ArrayListMainMemoryMapping.java (located in the library MatchLibrary-ProviderStores). This instance can be seen as an array list containing the several match result values, which are instances of IMappingEntry.java. The list is initialized by the respective matcher and is later used to initialize the match result object. Thus, your matcher can be executed from *executeMemory()*, but the structure described above has to be satisfied for this.

The following UML diagram gives an overview about the matcher structure. Although pretty large, the diagram is strongly simplified and many classes are omitted. The gray classes are interfaces, the yellow classes are the matchers (like TrigramMatcher etc.), and the dashed lines express “implements”-relations.

The only classes which belongs to COMA CE, are ExecWorkflow.java and MatchResult.java; all other classes are part of the several MatchLibrary libraries. In the class diagram, the classes (resp. interfaces) ArrayListMainMemory.java, IObjectMatcher.java, MappingEntry.java and Similarity.java are the most important ones to understand the match process. Also, the UML diagram might be quite useful if it is intended to add one's own matcher to COMA, which is described in the last chapter.


Similarity
+sim: double
IMappingProvider
AbstractMapping
IMappingEntry
+getLeft(): IObjectInstance
+getRight(): IObjectInstance
+getSimilarity(): ISimilarity
ISimilarity
+getSim(): double
IMappingStore
AbstractMappingProviderStore
IMappingStoreProvider
ArrayListMainMemoryMapping
+mapping: ArrayList<IMappingEntry>
MappingEntry
+left: IObjectInstance
+right: IObjectInstance
+sim: ISimilarity
IObjectInstance
+getID: String
+getValue(): Object
+getStringValue: String
+getIntegerValue: Integer
+getLongValue: Long
+...
+getSim(): double
ExecWorkflow
#executeMemory(): MatchResult
+...()
ObjectInstance
+values: Hashmap<String, Object>
Matcher1
Matcher2
...
MatchResult
...
IObjectInstanceProvider
IObjectMatcher
+match(): void
+match(): void
+match(): void
+getConfiguration(): IMatcherConfiguration
+getNumberOfComparisions(): int
+getProgress(): double
AbstractAttributeObjectMatcher
IMappingProvider
IThresholdedObjectMatcher
IAttributeObjectMatcher

4.4.3 Workflow

The method *executeMemory()* gets the source and target graph, the source and target objects of the graphs, the similarity measure which specifies the matcher to be executed, and two object instance providers:

```
MatchResult executeMemory( Graph srcGraph, Graph trgGraph, SimilarityMeasure simmeas,
ArrayList<Object> srcObjects, ArrayList<Object> trgObjects, IObjectInstanceProvider oipA,
        IObjectInstanceProvider oipB) {

        if (printSteps){
                System.out.println("Execute similarity measure (in memory) " + simmeas.getName());
        }

        if (srcObjects.isEmpty() || trgObjects.isEmpty()) return null;
```

The object instance providers are defined in the method *executeMatcherMemory()*, which calls *executeMemory()*:

```
Resolution resolution3 = matcher.getResolution();

// depending on resolution 2 a single or list of paths or nodes for each resolution 1

ArrayList<ArrayList<Object>> srcCombinedObjects = resolution3.getResolution3(srcGraph, srcObjects);
ArrayList<ArrayList<Object>> trgCombinedObjects = resolution3.getResolution3(trgGraph, trgObjects);

ArrayList<Object> srcSingleObjects = Resolution.getSingleObjects(srcCombinedObjects);
ArrayList<Object> trgSingleObjects = Resolution.getSingleObjects(trgCombinedObjects);

ArrayList<Object>  srcSingleSet = new ArrayList(new HashSet<Object>(srcSingleObjects));
ArrayList<Object>  trgSingleSet =  new ArrayList(new HashSet<Object>(trgSingleObjects));

// put source objects into the source instance provider
IObjectInstanceProvider  oipA = createAndFillInstanceProvider(srcSingleSet);

// put target objects into the target instance provider
IObjectInstanceProvider  oipB = createAndFillInstanceProvider(trgSingleSet);
```

Now, back in *executeMemory()*, a mapping variable is created, which is an instance of ArrayListMainMemoryMapping.java. See the UML diagram above – it contains a list of mapping entries. A mapping entry has two objects (*left* and *right*) and a similarity, which is basically a double variable, yet got a separate class. The objects *left* and *right* are not necessarily nodes of the schemas as one might expect at first glance – they actually can be anything: Leaf paths, parent nodes, siblings, node types etc., depending on the resolution used by the matcher (which must be resolution 3, though, because matchers always use a resolution 3).

```
ArrayListMainMemoryMapping mapping = null;

boolean forward = true;

if (forward){
        mapping = new ArrayListMainMemoryMapping(oipA, oipB);
} else {
        mapping = new ArrayListMainMemoryMapping(oipB, oipA);
}
```

Of course, the mapping is still empty, because no matcher has been executed so far. However, it will be given to the respective matcher then. The variable "forward" simply specifies in which direction the mapping has to be carried out (so source to target or vice versa). An important step is the following:

```
IObjectMatcher matcher = SimilarityMeasure.getMatcher(simmeas.getId());
```

This single line gets an instance of the matcher which is needed to perform the match strategy that is to be executed (and which is defined by the similarity measure id).

Taking a closer look at *getMatcher()* in SimilarityMeasure.java, it becomes clear how the connection between similarity measure id and matcher is: The id is used to return the specific matcher:

```
public static IObjectMatcher getMatcher(int id){

        IObjectMatcher matcher = null;

        switch (id) {
                case SIM_VECT_FEATURES:
                        matcher =  new VectorMatcher("attr", "attr", 0f);
                        break;
                case SIM_DATATYPE:
                        matcher =  new DatatypeMatcher("attr", "attr", 0f);
                        break;
                case SIM_STR_TRIGRAM:
                        matcher =  new ComaTrigram2("attr", "attr", DEFAULT_THRESHOLD);
                        break;

                ...
```

Assuming that the matcher is not null, the following is performed (back in *executeMemory()* again):

```
if (forward){
        matcher.match( oipA, oipB, mapping);
else {
        matcher.match( oipB, oipA, mapping);
}
```

In this step, the matcher is now executed. As already described, each Matcher overrides the three match methods by the IAttributeObjectMatcher.java interface, and must at least implement one of them. In this case, the match method is called which gets two object instance providers and the mapping (which every match method gets as parameter).

The matcher now initializes the mapping list, which at last contains the result. The following step is performed to convert the result in the match result structure of COMA.

```
float[][] simMatrix = new float[srcObjects.size()][trgObjects.size()];

for( final IMappingEntry me : mapping) {

		if (me==null) continue;
		IObjectInstance source = me.getLeft();

		int src = Integer.parseInt(source.getId());
		IObjectInstance target = me.getRight();

		int trg = Integer.parseInt(target.getId());
		ISimilarity similarity = me.getSimilarity();

		float sim = (float) similarity.getSim();

		if (sim>0) {
			if (forward) {
				simMatrix[src][trg]=sim;
			} else {
				simMatrix[trg][src]=sim;
			}
		}
}


mapping = null;
MatchResult result = new MatchResultArray(srcObjects, trgObjects, simMatrix);
matcher = null;
result.setSourceGraph(srcGraph);
result.setTargetGraph(trgGraph);

return result;
```

As it can be seen, the for-loop does the main part. It grasps all elements of the mapping list (which are instances of MappingEntry.java), gets the source and target id as well as the similarity value (the “confidence”) and adds it into a newly created float matrix. As already described, a match result is basically nothing else but a float matrix.

4.5 Confidence and Thresholds

4.5.1 Overview

Thresholds are important criteria in schema matching. The class MatchResultArray.java offers the method *getSimilarity()*, which gets two objects (source and target element) and returns the confidence between those two objects. As already mentioned, it is a float value between 0 and 1.

There are some basic thresholds defined in MatchResultArray.java, mainly the following three final constants: SIM_UNDEF, SIM_MIN and SIM_MAX. The first one is used to specify the similarity if no value is available. It is normally 0 and may occur if the similarity between objects is to be checked, which cannot be compared or which have not been compared so far. SIM_MIN defines the maximum value to declare a correspondence as false (this is 0 by default) and SIM_MAX the minimum value to declare it as certain (this is 1 by default).

It has to be noted that various classes do not stick to these constants, especially SIM_MIN, but use the not-final value 0, like *if(sim > 0)* instead of *if(sim > SIM_MIN)*. This is a little inconsistent and troublesome, yet searching for methods that call *getSimilarity()* in the program might help to find the places where these statements occur.

Matcher-specific thresholds are defined in the matcher algorithms itself or left out completely (there might be some reasons against matcher-specific thresholds, because the final confidence is calculated out of the entire result the workflow execution provides, and not out of single matchers results).

4.5.2 Thresholds for Correspondences

The thresholds for correspondences are used to select correspondences with regard to their confidence. An important threshold is the minimum confidence of a correspondence, which is 0.4 by default, meaning that correspondences below 0.4 will not be displayed in COMA (resp. will not be regarded as possible correspondences at all).

This threshold is called selection threshold in the grammar, yet there are some further thresholds that can be applied. According to the grammar, a selection can only be declared for a strategy, that is, selections are carried out on the upper level of the workflow hierarchy (not in the matchers itself). That means that single matchers can provide correspondences below 0.4, but if the correspondence is still below 0.4 when the match process is over, they will be abandoned.

If other thresholds are desired, the predefined workflows have to be changed. You may also take a look at the method *Selection.select()*, where the selection takes part.

5 Insert and Export

5.1 Overview

The first step in schema matching is to load database schemas, normally a source and a target schema, and perhaps also some instance data to support several matching strategies. The module coma-insert is solely designed to load schemas and instances of different types into the program. As already pointed out, each schema will be transformed into a unique graph object, which allows to compare schemas of different types. In addition to this process, such a schema (resp. graph) can also be inserted into a database to allow a fast access at a later time.

The classes that load schemas and transform them into graph objects are called parser. There are three kind of parsers: Parsers for schemas, for instances and for relationships. The schema parsers have the largest importance in COMA.

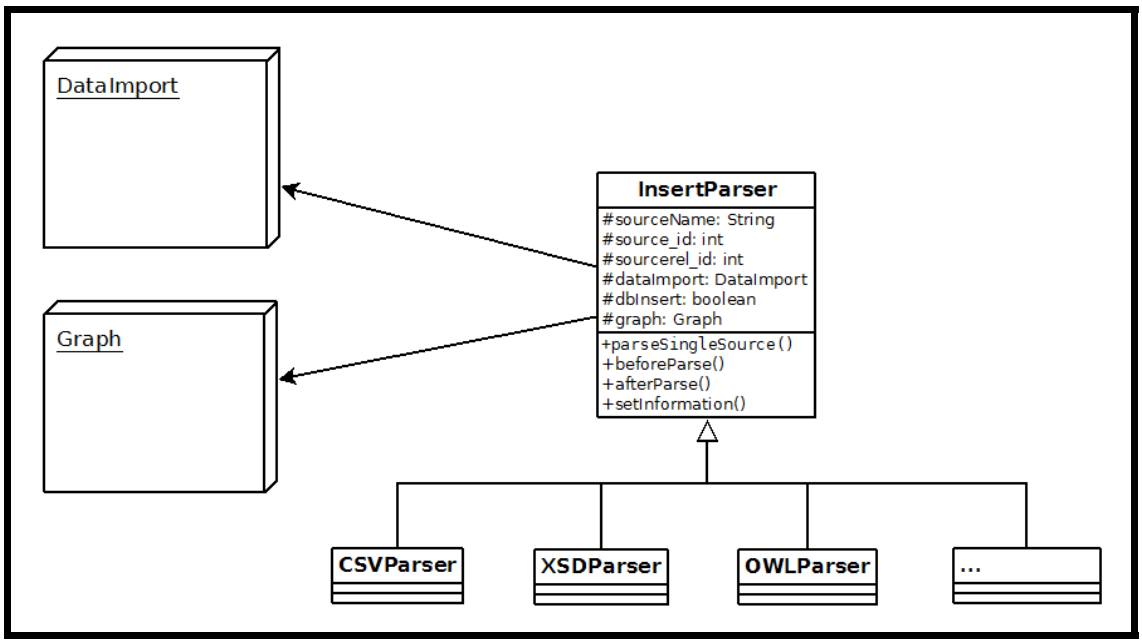
There are also a few classes to export data, so to convert some results from COMA into a file again. These export classes are located in the coma-export module, but actually have only a minor importance right now.

This chapter concentrates on the parsers, especially on schema parsers. The basics of adding a new parser to COMA CE are explained in the last chapter.

5.2 Schema and Ontology Parser

5.2.1 Basics

For each schema type there exists a unique parser class, e.g., CSVParser.java or OWLParser.java, which are located in the metadata package of the coma-insert module. Each parser is a specification of the central class InsertParser.java. InsertParser.java contains a type (e.g., XSD, OWL, etc.), a name, a source id and source relation id, and a graph object, which is the result when the parse process is accomplished. The variable *dbInsert* specifies whether the graph is to be inserted into the database or not. For this, the variable *dataImport* is used to insert a graph object into the database. If *dbInsert* is true, the graph is just inserted into the DB, and no graph object is built (this way, many schemas can be loaded into the DB without any difficulties). If *dbInsert* is false, the graph object is loaded, and the schema is not inserted into the DB.



The schema types are defined as global final variable in the class Source.java, which is located in the project coma-structure.

5.2.2 Structure of a Parser

As mentioned before, each parser must extend InsertParser.java and should be located in the metadata package. Also, each parser overrides the abstract method *parseSingleSource()*, which gets the file name, schema name, author and some further attributes as input.

To parse a schema, a parser instance is created by calling the parser's constructor. This process shall be explained by the example of the class CSVParser.java. The constructor calls the super constructor, so the one of InsertParser.java, which either loads the graph from the database (no parsing as such) or from a file (actual parsing), depending on the *insertDB* specification.

After this, the mandatory method *parseSingleSource()* of CSVParser.java is called to start the parse process. This normally contains four method calls: *beforeParse()*, *parse()*, *setInformation()* and *afterParse()*.

The methods *beforeParse()* and *afterParse()* are overwritten by CSVParser; *beforeParse()* is used to create a default schema name if no such name is specified, to create a new log file for the parse process and to initialize the graph instance of InsertParser.java. The method *afterParse()* takes care of several formalities after the graph was loaded, e.g., to check whether the graph contains any cycles or simply to preprocess the graph. The method *parse()*, which is in the respective parser class, is the central method to perform the parsing. This method is not overriden by CSVParser.java.

Besides, a parser calls the method *setInformation()* when *parse()* was executed. This will set additional graph information like author, domain, comment etc. to the graph object. At the end, the source_id will be returned.

5.3 Additional Parser

Next to the schema and ontology parsers there are the packages instance, where a couple of instance parser are located, and relationships, where relationship parsers (like the match result parser to load a match result) are located.

Like in schema parsing, for each instance type exists exactly one parser class, e.g., XMLParser.java etc. The structure of these classes is less complex than in schema parsing (for instance, there is no superior class). The instance parser classes normally contain a central method *parseInstance()*, which normally gets a graph object and also returns a graph object. Remember that instance data is directly attached to the elements of the schema graph, so an instance parser can only be called if a respective schema parser has already been executed, and a respective graph object is already at hand. Subsequently, the instance parser adds the instance values to the graph elements – for this, instance file and schema file must be consistent, of course.

5.4 Export

The project coma-export contains classes that export data objects. They do the opposite of the parsers – they transform a data object into a file again. There are two export classes right now: MatchResultExport.java, which writes a match result into a text file, and RDFExport.java, to write a RDF result into a text file.

The third class of coma-export is ExportUtil.java, which is chiefly used to perform the actual file writing.

6 COMA GUI

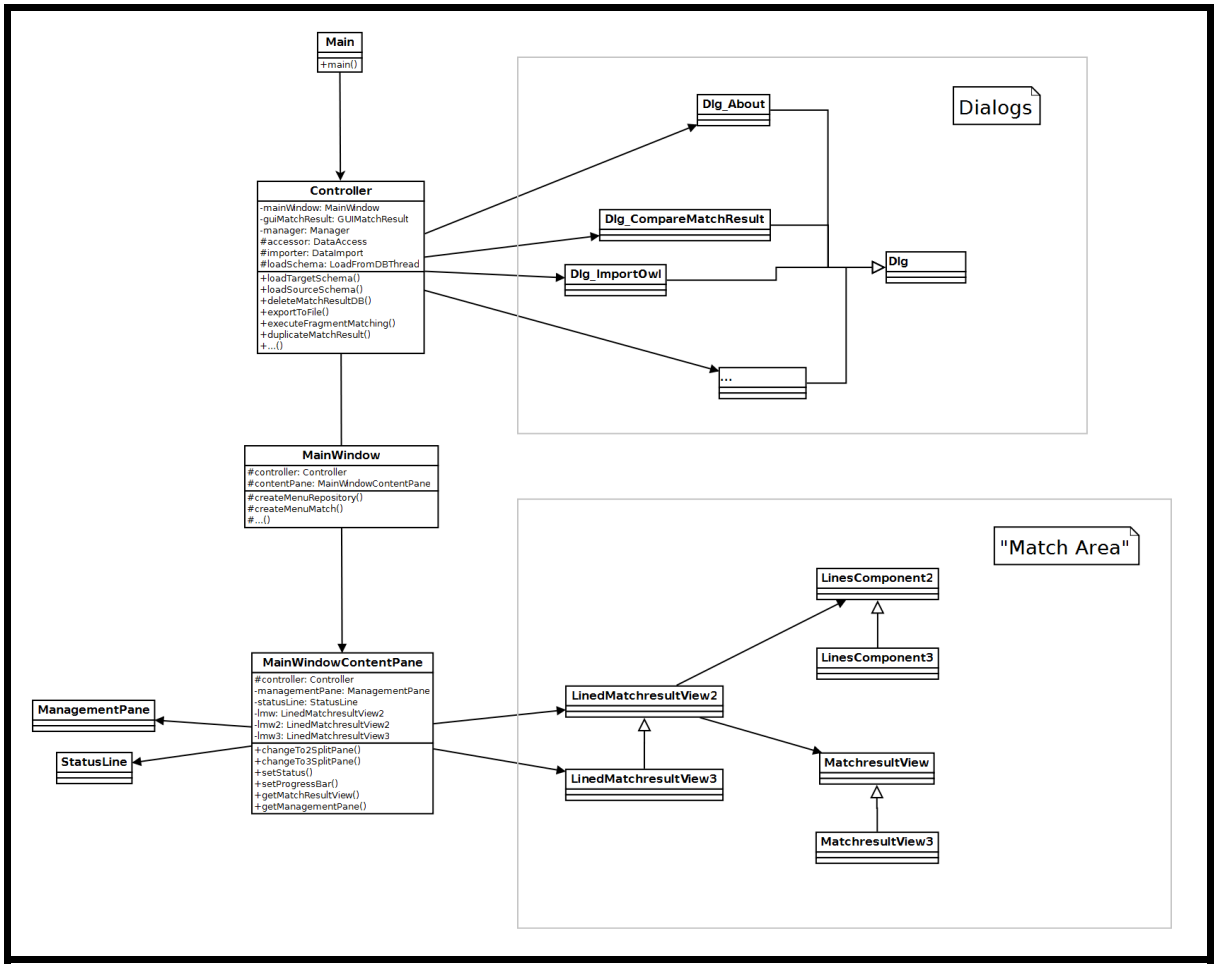
6.1 Basics

6.1.1 Introduction

All classes referring to the GUI are situated in the separate project coma-gui. As it can be seen in the following UML diagram, there are the three major classes Controller, MainWindow and Manager, thus representing the classic MVC architecture, a main class to start the program, a main view content pane which is closely connected to the main window, as well as several classes for the dialogs and a bundle of class for the “match area”. This area is an important part of the main view content pane; it comprises the two or three columns where the source and target schema (and additional information) are displayed, and the correspondence lines in between, as well as the context menu (pop-up menu) belonging to this area.

In the main directory of this project are the base classes needed for the GUI. Additionally, there are three sub-directories:

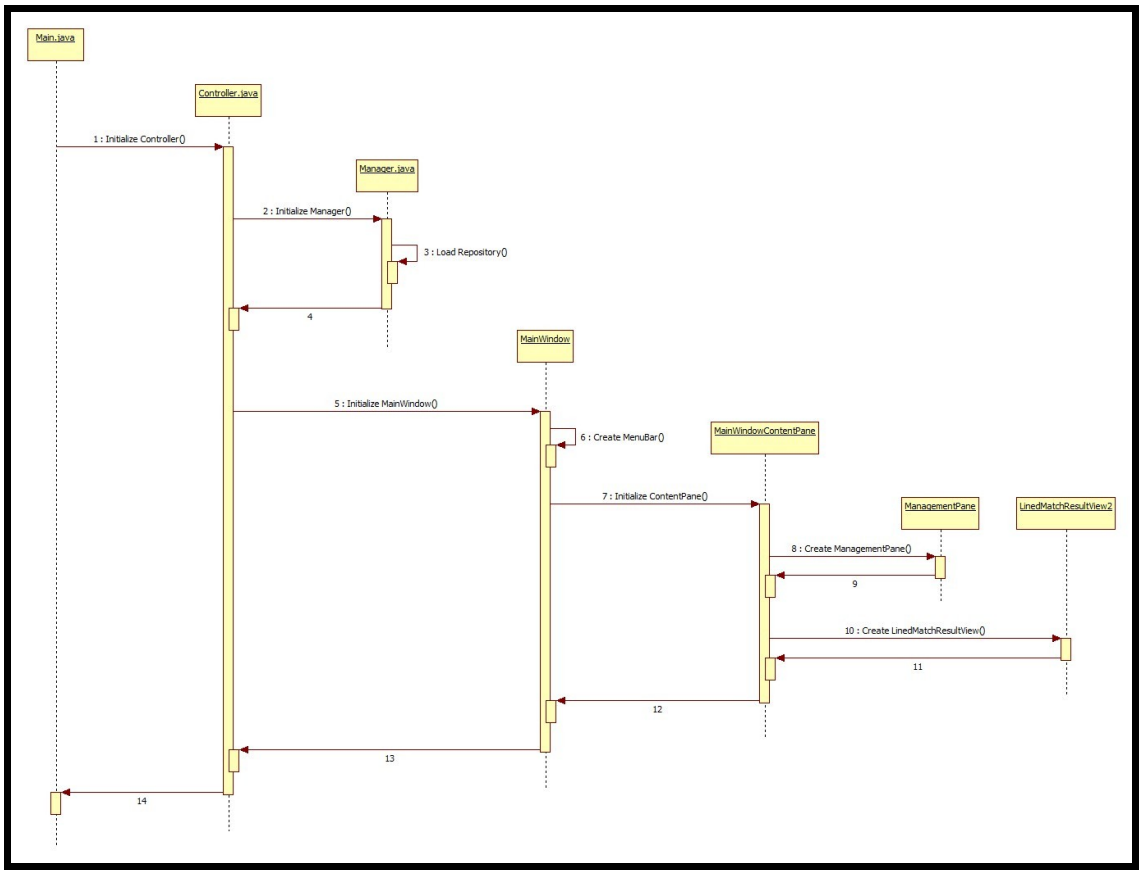
- Package dlg: Contains all dialog classes (normally, each dialog consists of exactly one class).
- Package extjtree: Contains classes that create and manage the schema trees (these classes are extensions of JTree).
- Package view: Contains all “view” classes of the match area (!), e.g., classes to create the two or three columns of the match result, the lines of the correspondences etc.



6.1.2 Starting the Program

COMA is started by the main class (Main.java) in the coma-gui package. This class contains the main method as its only method, and only two important steps are performed: An instance of Controller.java is generated and its method *start()* is run.

The picture below shows the basic workflow (simplified) as a sequence diagram. It will be explained in the following sections.



6.1.3 The Controller

The Controller is a large class which chiefly contains an instance of the view (an instance of MainWindow.java) and of the manager, thus following the MVC standard. Besides this, there exist instances of all classes needed by the controller, e.g., match result, data importer, data accessor etc., as well as a huge amount of methods to perform any task that can be triggered by the user, such as loading and deleting schemas, starting the match process, cleaning the repository, displaying schema information etc. These methods are triggered by action listeners, which are added to the GUI components in the MainWindow instance.

Following the workflow, the constructor of Controller.java creates a new instance of the manager, which loads the repository and creates instances of the data importer and data accessor. Subsequently, the main window is created. The following method *controller.start()* is only used to make the view visible.

6.1.4 The Manager

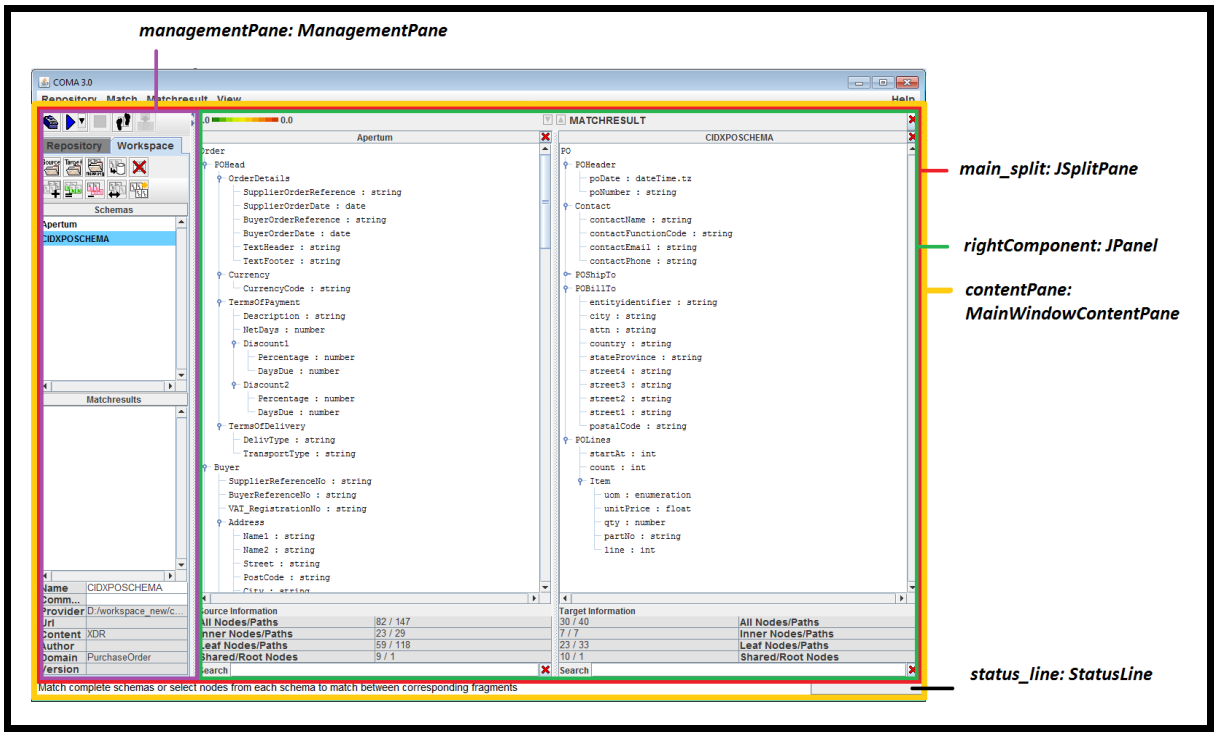
The manager (class Manager.java) is an additional class that supports the controller. It concentrates rather on loading and saving schemas and match results. Note that Manager.java is located in the module coma-center.

6.1.5 The Main Window

The main window constructor is called by the constructor of the Controller; the controller instance itself is given to the main window as a reference. The main window constructor takes care on creating the entire GUI, i.e., the JFrame. For this, it uses several sub-methods. The largest sub-methods are *createMenuRepository()*, *createMenuMatch()* etc., which are used to create the menus in the menu bar. Yet most importantly, the constructor creates an instance of MainWindowContentPane.java. This content pane presents everything in the frame except the menu bar.

The main window has instances to other classes again, which focus on a specific parts of the content pane, and which are initialized in the constructor of MainWindowContentPane.java as well. The entire left-hand area (“sidebar”) is an instance of ManagementPane.java. Parts of the matching area are an instance of LinedMatchresultView2.java. They are embedded in the central panel rightComponent, a global variable in MainWindowContentPane.java. Also

the status line is a separate class (StatusLine.java). The sketch below shows the structuring of the panes in the GUI.



6.2 Dialogs

Dialogs are separate classes. They are located in the dlg package of coma-gui. For the sake of uniformity, the dialog classes start with “Dlg_”.

Each dialog class extends Dlg.java, which contains a few dialog specific methods each dialog may need. A dialog constructor normally gets the JFrame instance and possibly the controller instance or further parameters that are necessary.

Sometimes, most of the code creating the content pane of a dialog is located in the constructor, and sometimes it is rather distributed across sub-methods or simply excluded to an init()-method. There is no standardization about this.

The certainly simplest dialog is the About Dialog (Dlg_About.java), which only displays the About message. On the example of this dialog, it can be seen that it is solely called from the method *showAbout()* in the Controller. Of course, this method is called when the respective action event is triggered, that is, when the user clicked the About menu item. The menu item “About” is defined in the class MainWindow.java. As it can be seen in the excerpt below, there is also the action listener declared.

```
item = getMenuItem(GUIConstants.ABOUT);
item.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent _event) {
            controller.showAbout();
      }
});
help.add(item);
```

To add a new dialog, one simply has to add a new menu item to one of the menus, declare an action listener where a method in the Controller is called to open the respective dialog, and write a new dialog class.

6.3 The Match Area

The match area consists mainly of 5 classes: Line.java, LinedMatchresult2.java, Matchresult.java, LinesComponent2.java and Match.java. The classes ending with “3” refer to the scenario where the match area contains 3 columns for specific tasks; they are extensions of the classes that end with “2” and are not regarded any further.

The class LinedMatchresultView2.java is the central pane of the match area. It has instances of MatchresultView2.java and LinesComponent2.java. The class LinesComponent2.java contains the source schema and target schema graph (the trees) and the match result, as well as a list containing all lines (“linesList”); this list can be seen as the accumulation of all lines. A line is the “atomic” element in the match area and simply represents a correspondence (“line”) between source and target side. The most important characters of a line are its color, stroke and the similarity value of the correspondence; the colors are defined as public final static variables at top of the class.

The class MatchResultView2.java provides the two columns where the source and target schema are displayed. Thus, the entire match area both needs a match result view and a line component. They are put over each other, as the code snippet shows:

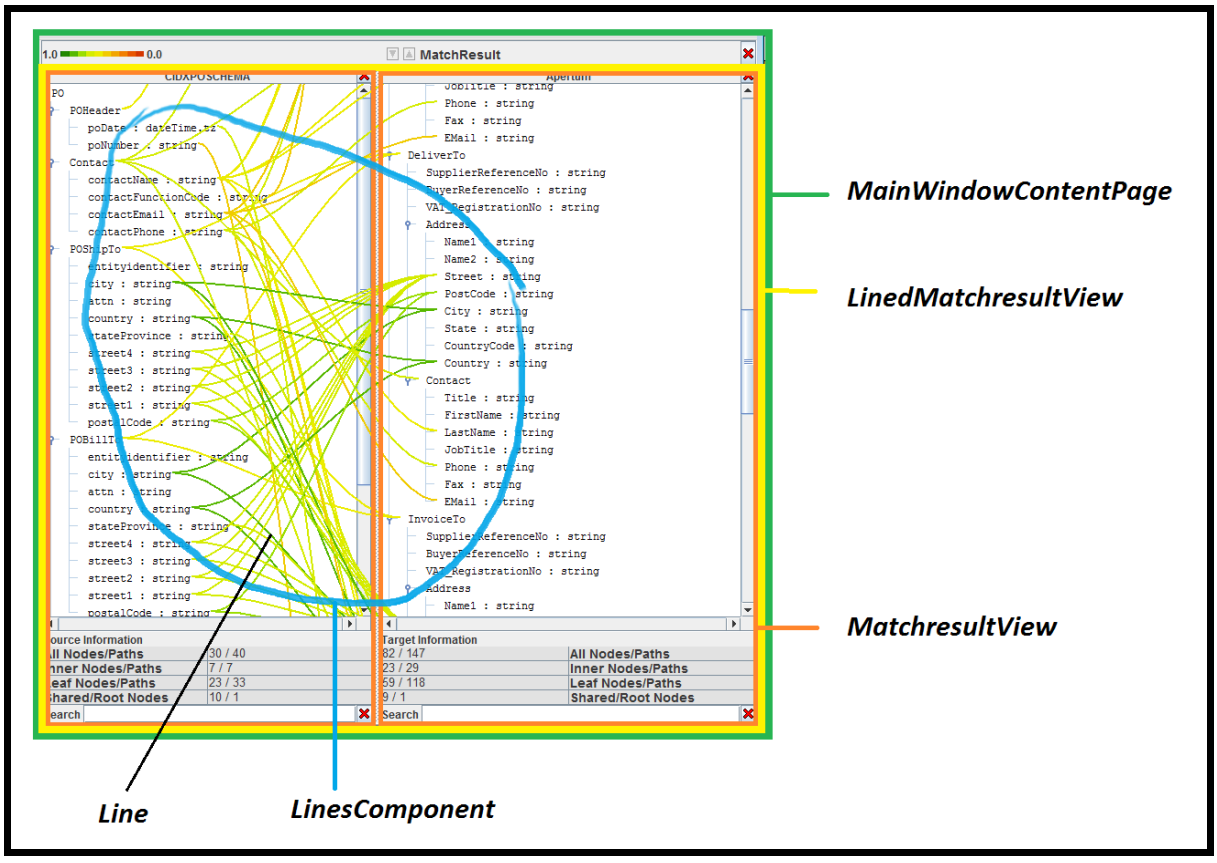
```
void init2(Controller _controller) {
    mw = new MatchresultView2(this, _controller);
    add(mw);  // Add the matchresultview to the pane (the "columns")
    setLayer(mw, JLayeredPane.DEFAULT_LAYER.intValue());
    //LinePane
    linesComponent = new LinesComponent2(_controller);
    add(linesComponent);  // Add the lines component to the pane (the "lines")
    // position of the linesComponent in front of the trees
    setLayer(linesComponent, JLayeredPane.DEFAULT_LAYER
            .intValue() + 1);

    simComponent = new SimComponent2(_controller);
    add(simComponent);
    setLayer(simComponent, JLayeredPane.DEFAULT_LAYER
            .intValue() + 2);

    mw.setDividerLocation(DIV_LOCATION_MATCHRESULT);
}
```

Also, in MatchresultView2.java the context menu (pop-up menu) of the match area is created. The method *initPopUp()* is chiefly responsible for this. The menu items are added to the pop-up menu in the same way as menu items are added to the program menus of COMA.

The following picture condenses the meaning of the several classes again:



7 Extension and Adjustment

7.1 Overview

While the previous chapters rather concentrated on the program structure and the structure and meaning of classes and modules, this chapter will focus more on the practical part. It will be explained, what steps have to be performed to start COMA and to get a match result, and how some crucial parts of COMA can be extended or adjusted.

This chapter therefore explains...

- What code has to be run for the COMA basis scenario (loading two schemas, running the matcher and obtaining a match result).
- How a new schema parser is to be added.
- How new matchers and new workflows can be created.

7.2 Getting Started

7.2.1 Introduction

In the following, a demo class is presented that explains in a very simple way how two schemas are matched. For this, we have the following assumptions:

- We have two xsd schemas Library_Source.xsd and Library_Target.xsd.
- We want to perform a complete match process, thus...
    - we have to load the two schemas into the program,
    - we have to run the matchers,
    - we have to output the result (we simply use the console output for this task).

We first present the method which does all these assumptions, and after this explain the steps that are to be performed. The method, which we simply called “match”, is mainly divided in 4 sections that together form two parts: Section 1 .. 3 belong to the first part where we set up the database and load the schemas, and section 4 is the second part, where the workflow is executed and the match result is presented.

The method gets just the two file paths of the xsd schemas. It will be now presented in a whole, before the several sections are explained in detail.

```
public static void match( String sourceSchemaPath, String targetSchemaPath) {

        /* (1) We connect to the database. */

        System.setProperty( "comaUrl", "jdbc:mysql://localhost/coma-project?autoReconnect=true");
        System.setProperty( "comaUser", "");
        System.setProperty( "comaPwd", "");


        /* (2) We create a new manager and controller. */

        Manager manager = new Manager();
        DataAccess accessor = manager.getAccessor();
        Controller c = new Controller( manager, accessor);
        c.createNewDatabase( false);

        manager = c.getManager();
        accessor = manager.getAccessor();


        /* (3) We tell COMA the paths and explain what we want to match. */

        String[][] schemaImport = { { sourceSchemaPath, "library_src"},
                                    { targetSchemaPath, "library_tar"} };

        String[][] schemaPairs = { {"library_src", "library_tar"} };
```

```
		/* (4) We execute the workflow, running the ALLCONTEXT strategy. */

		Workflow w = new Workflow( Workflow.ALLCONTEXT);
		ExecWorkflow exec = new ExecWorkflow();


		/* (4.1) We load the two xsd schema using the XSD Parser. */

		boolean dbInsert = true;
		XSDParser xsdParser = new XSDParser( dbInsert);

		for (int i = 0; i < schemaImport.length; i++) {
			String schemaLocation = schemaImport[i][0];
			String schemaName = schemaImport[i][1];
			xsdParser.parseSingleSource(schemaLocation, schemaName);

		}


		/* (4.2) Parsing accomplished - we update the information. */

		c.updateAll( true);  // update information
		manager = c.getManager();
		accessor = manager.getAccessor();

		for( int i = 0; i < schemaPairs.length; i++) {

			String sourceName = schemaPairs[i][0];
			String targetName = schemaPairs[i][1];
			int sourceId = accessor.getSourceIdsWithName( sourceName).get(0);
			int targetId = accessor.getSourceIdsWithName( targetName).get(0);
			Graph sourceGraph = manager.loadGraph( sourceId);
			Graph targetGraph = manager.loadGraph( targetId);

			w.setSource( sourceGraph);
			w.setTarget( targetGraph);
			MatchResult result = exec.execute( w);   // Match the two schemas

			System.out.println( result.toString());   // Print the result

		}


	}
```

7.2.2 Database Setup

In the first and second section, a connection to the database is created. As matter of fact, we have to set url, user and password (section 1).

In the second section we create a manager, data accessor and controller instance. Basically, the controller controls the manager instance, which again controls the accessor. While the manager is used to load and manage graphs (amongst other tasks), the accessor is used for database access. The controller itself is part of the GUI.

The next step is to create a new database. The boolean value false just means that no confirm dialog is shown, i.e., that the old database is directly deleted.

```
/* (1) We connect to the database. */

System.setProperty( "comaUrl", "jdbc:mysql://localhost/coma-project?autoReconnect=true");
System.setProperty( "comaUser", "");
System.setProperty( "comaPwd", "");


/* (2) We create a new manager and controller. */

Manager manager = new Manager();
DataAccess accessor = manager.getAccessor();
Controller c = new Controller( manager, accessor);
c.createNewDatabase( false);

manager = c.getManager();
accessor = manager.getAccessor();
```

7.2.3 Path Setting

In the third step we set the paths and tell COMA what we want to match. First, we set the directory. This is where COMA searches for the schemas. For this, *directory + sourceSchemaPath* and *directory + targetSchemaPath* must be valid paths, otherwise a “file not found error” would be the result.

The variable *schemaImport* stores the schemas which are to be imported. In our case, we import two schemas (the source and target schema). A schema import specification consists of two values: The schema file path and the schema name. Therefore, the first value specifies the path so that COMA can load the file, while the second value is just used to give the schema a name (which can be displayed in the GUI, for instance). For simplicity, we used constants here, but normally one would rather use the schema names or default names if they do not exist.

The variable *schemaPairs* is a list of schema pairs, which are to be matched. In our case, this is just one pair, the library schemas.

```
/* (3) We tell COMA the paths and explain what we want to match. */

String[][] schemaImport = { { sourceSchemaPath, "library_src"},
                            { targetSchemaPath, "library_tar"} };

String[][] schemaPairs = { {"library_src", "library_tar"} };
```

7.2.4 Workflow Execution

First of all, we need two important instances: One of the Workflow class, *w*, and one of the ExecWorkflow class, *exec*. Workflow.java is used to represent workflows. It contains a list of strategies (remember that a workflow consists of one strategy or more than one strategies), a source graph, a target graph and some further important variables necessary for workflows. ExecWorkFlow.java is just used to execute an existing workflow, and for this reason gets a Workflow instance (see below).

We initialize the workflow with the predefined ALLCONTEXT workflow, the standard workflow of COMA.

```
/* (4) We execute the workflow, running the ALLCONTEXT strategy. */

Workflow w = new Workflow( Workflow.ALLCONTEXT);
ExecWorkflow exec = new ExecWorkflow();
```

After this is done, the schemas are to be loaded. For this, we need a parser, more specifically: An XSD parser. We create a new instance of this parser and tell it that the loaded schema is to be inserted into the database. Then we iterate the *schemaImport* field and load all schemas that it contains. If there are different kind of file formats in the list, we would need some further if-constructions that first check the file extension and then call the respective schema parser (remember that we could match, for instance, an XSD schema with a database schema etc.).

```
/* (4.1) We load the two xsd schema using the XSD Parser. */

boolean dbInsert = true;
XSDParser xsdParser = new XSDParser( dbInsert);

for (int i = 0; i < schemaImport.length; i++) {
        String schemaLocation = schemaImport[i][0];
        String schemaName = schemaImport[i][1];
        xsdParser.parseSingleSource(schemaLocation, schemaName);
}
```

In the last step, we match the schemas and print the result. We first have to update the database (because we have loaded new schemas) and then iterate all schema pairs. Since we only have one schema pair, the for-method is iterated only once.

The process performed in the for-loop is pretty simple now. We determine the source id and target id of the source and target schema, and having the id, are able to load the source and target graph, which are the most important elements in the match process. This accomplished, we set the source graph and target graph of the workflow, which is complete now. We run the ExecWorkFlow.java instance now by calling the *execute* method, which gets the workflow instance and returns a match result. Now having this match result, we can print it, or do whatever else we like (e.g., enhancement and postprocessing, data transformation, …). Of course, these features are not part oft COMA CE anymore.

```
/* (4.2) Parsing accomplished - we update the information. */

c.updateAll( true);  // update information
manager = c.getManager();
accessor = manager.getAccessor();

for( int i = 0; i < schemaPairs.length; i++) {

        String sourceName = schemaPairs[i][0];
        String targetName = schemaPairs[i][1];
        int sourceId = accessor.getSourceIdsWithName( sourceName).get(0);
        int targetId = accessor.getSourceIdsWithName( targetName).get(0);
        Graph sourceGraph = manager.loadGraph( sourceId);
        Graph targetGraph = manager.loadGraph( targetId);

        w.setSource( sourceGraph);
        w.setTarget( targetGraph);
        MatchResult result = exec.execute( w);   // Match the two schemas

        System.out.println( result.toString());   // Print the result

}
```

7.3 Creating a New Schema Parser

COMA CE also comes with a large selection of schema parsers, allowing schema matching between most different kind of databases. However, there are many specific formats which cannot be handled by COMA so far, thus adding a new schema parser can be an important task. The basics of schema parsers were already explained in the 5th chapter. In this section it is described how a new schema parser can be developed and added to the program.

To create a new parser, called “SampleParser” in this section, it is advised to do the following:

- Add a new schema type specification “SAMPLE” at the head of the class *Source.java* in the *coma-structure* module. The name should match the schema type name.
- Create a new class SampleParser.java in the metadata package of the module coma-insert. It must extend InsertParser.java and thus override *parseSingleSource()*.
- Add a constructor to the class that calls the super constructor, which requires a *dbInsert* specification and a schema type specification (SAMPLE).
- In the overriden method *parseSingleSource()*, call the methods *beforeParse()*, *parse()*, *setInformation()* and *afterParse()*.
- The return value should be *source_id*.
- After this, create a new *parse()* method which is used to perform the actual parsing.
- Write the parser. As matter of fact, you may add further (auxiliary) methods to the class if your parser should be more complex.
- After the parser is written, you may create a new instance *mySampleParser* of the file at any place in COMA CE and load a schema by calling *mySampleParser.parseSingleSource(...)*.

The picture below should condense the main ideas again.

At this point, it would be too extensive to explain how a parser has to be written in detail. You may have a look at the parsers already implemented, e.g., the CSV parser (which is a rather simple one), to get acquainted with this topic. It is also recommended to have a look at the structure chapter, where it is explained how the graph data structure is expressed in COMA.

```java
package de.wdilab.coma.insert;

import de.wdilab.coma.structure.Source;


/**
 * The parser class to parse a Sample Schema.
 */
public class SampleParser extends InsertParser {


    /**
     * Constructor - Creates a new instance of the parser.
     * @param dbInsert Specifies whether the graph is imported from the DB or from a file.
     * @param type The parser type.
     */
    public SampleParser( boolean dbInsert, int type) {
        super( dbInsert, Source.TYPE_SAMPLE);
    }


    @Override
    public int parseSingleSource(String filename, String schemaName,
            String author, String domain, String version, String comment) {

        beforeParse();
        parse();
        setInformation( author, domain, version, comment);
        afterParse();

        return source_id;
    }


    /**
     * Parser to parse a "Sample Schema".
     */
    private void parse() {

        // MY PARSER

    }


}
```

The picture shows what SampleParser.java should look like. In the method parse() the respective parser code has to be added.

7.4 Workflow Adjustment

7.4.1 Introduction

The basics of COMA workflows were already explained in chapter 4.3. In this section it will be described, how the predefined workflows are expressed in COMA and how the user may create a new one. It is recommended not to change the existing workflows – they are pretty powerful and useful already. Instead, it is much more convenient to create a new workflow, if the existing ones turn out to be not sufficient enough.

7.4.2 Different Ways of Workflow Adjustment

There are basically two ways how workflows can be expressed in COMA. The first possibility is to create a hard-coded predefined workflow; there are already several predefined workflows in COMA. These workflows are directly implemented in the system, i.e., cannot be changed at runtime, and require recompilation if anything is to be changed or a new workflow is to be created. Thus, this solution is appropriate if a new workflow is supposed to be used for a long (permanent) time and is supposed to be immutable.

The second possibility is to write a workflow at run time. One can be very flexible then, but this solution has a few drawbacks right now: You have to fully

understand the workflow grammar, and this kind of workflow representation is not stored persistently (because COMA offers no feature to load and save grammar strings yet). On the contrary, this solution is generally easier, because you do not have to change the java code. If you build add-ons to load workflows from a file and to create and edit them in COMA, you will be even able to adjust it whenever you wish, and recompilation would not be necessary.

7.4.3 Predefined Workflows in COMA

It was already explained in the 4th chapter that a workflow in COMA CE is represented by Workflow.java, which is interrelated with the classes Strategy.java, ComplexMatcher.java, Matcher.java and further classes (also see the UML diagram in chapter 4.3 for a better understanding).

At the top of these 4 classes the name of a workflow, strategy, complex matcher and matcher are declared as final integer variables. So a workflow has a name and consists of several strategies, which have names again (described in the class Strategy.java) etc.

The main constructor in Workflow.java gets only an integer value as parameter, which specifies which workflow has to be run. Then, a simple switch-case-structure executes the respective workflow. This way it is very easy to execute a specific workflow from every part of COMA.

```
public Workflow (int workflow){

    switch (workflow) {
    case ALLCONTEXT :
        strategies = new Strategy[1];
        strategies[0] = new Strategy(Strategy.COMA_OPT);
        setName("AllContextW");
        break;
    case FILTEREDCONTEXT :
        strategies = new Strategy[1];
        strategies[0] = new Strategy(Strategy.NODE_SELECTION);
        secondStrategy = new Strategy(Strategy.UPPATH_SELECTION);
        setName("FilteredContextW");
        break;
    case FRAGMENTBASED :
        strategies = new Strategy[1];
        strategies[0] = new Strategy(Strategy.FRAG_SELECTION);
        secondStrategy = new Strategy(Strategy.DOWNPATH_SELECTION);
        // TODO: automatically transfer result back to path from root
        setName("FragmentBasedW");
        break;
    case OPTIMISTIC:
        strategies = new Strategy[3];
        strategies[0] = new Strategy(Strategy.COMA);
        strategies[1] = new Strategy(Strategy.COMA_OPT);
        strategies[2] = new Strategy(Strategy.SIMPLE_NAMEPATH);
        combination = new Combination(Combination.RESULT_MERGE);
        setName("OptimisticW");
        break;
    case NODES:
        ...
```

As it can be seen, each case statement defines a specific workflow. At a closer look on the case statements, it can be seen that in each workflow a list of strategies is defined, and if this list is larger than 1, a combination specification is done. The method *setName()*, which is generally called in each case, just sets the name of the workflow instance.

Now the strategy class looks almost like the workflow class and contains a list of predefined strategies.

```
...
case COMA :
    this.resolution = new Resolution(Resolution.RES1_PATHS);
    cm = new ComplexMatcher[5];
    cm[0] = new ComplexMatcher(ComplexMatcher.NAMETYPE); // NAMETOKENSYN
    cm[1] = new ComplexMatcher(ComplexMatcher.PATH);
    cm[2] = new ComplexMatcher(ComplexMatcher.LEAVES);
    cm[3] = new ComplexMatcher(ComplexMatcher.PARENTS);
    cm[4] = new ComplexMatcher(ComplexMatcher.SIBLINGS);
    simCombination = new Combination(Combination.COM_AVERAGE);
    selection = new Selection(Selection.DIR_BOTH, Selection.SEL_MULTIPLE, 0, (float)0.008, (float)0.5);
    setName("ComaCS");
    break;
case SIMPLE_NAMEPATH :
    this.resolution = new Resolution(Resolution.RES1_PATHS);
    cm = new ComplexMatcher[2];
    cm[0] = new ComplexMatcher(ComplexMatcher.NAMETYPE); // NAMETOKENSYN NAMESTAT
    cm[1] = new ComplexMatcher(ComplexMatcher.PATH);
    simCombination = new Combination(Combination.COM_AVERAGE);
    selection = new Selection(Selection.DIR_BOTH, Selection.SEL_MULTIPLE, 0, (float)0.01, (float)0.2);
    setName("SimpleNamePath");
    break;
case NODE_SELECTION:
    this.resolution = new Resolution(Resolution.RES1_NODES);
    cm = new ComplexMatcher[1];
    cm[0] = new ComplexMatcher(ComplexMatcher.NAMETYPE); // NAMETOKENSYN
    selection = new Selection(Selection.DIR_BOTH, Selection.SEL_THRESHOLD, 0, 0, (float)0.3);
    setName("NodeSelectionCS");
    break;
case UPPATH_SELECTION:
    this.resolution = new Resolution(Resolution.RES1_UPPATHS);
    cm = new ComplexMatcher[1];
    cm[0] = new ComplexMatcher(ComplexMatcher.PATH);
    selection = new Selection(Selection.DIR_BOTH, Selection.SEL_THRESHOLD, 0, 0, (float)0.5);
    setName("UpPathSelectionCS");
    break;
case FRAG_SELECTION:
...
```

As it can be seen, a strategy consists of a resolution specification, a list of complex matchers and a selection specification. If the list of complex matchers is larger than 1, a similarity combination specification has to be done as well.

The classes ComplexMatcher.java and Matcher.java have the same structure as Workflow.java and Strategy.java, and shall not be described any further.

7.4.4 Adding a New Predefined Workflow

Adding a new predefined workflow always requires to add a new workflow case in the workflow constructor. Next to this, a new constant has to declared, and a name to be specified. This step is inevitable to create a new workflow, while the following steps are rather optional, depending on how much new strategies, complex matchers or matchers your new workflow should consist of.

If you only want to create a workflow using already existing strategies, this is quite easy. You have to create a new strategy list and specify the strategies your workflow should consist of. Also, a combination specification is necessary if more than one strategy is specified. At this point you are already done then.

However, if the strategies already available are not sufficient, you also have to create a new strategy in the strategy class – the structure of a strategy was already described above. You must take care that the grammar of a strategy is fulfilled; a strategy normally consists of a set of complex matchers, a selection specification, and a resolution, which has to be of type 1 under all circumstances. If your strategy consists of more than one complex matcher, a combination specification is necessary again.

This way getting down to the matchers you are able to built any possible workflow, i.e., you can also create new complex matchers and new matchers. Of course, you may easily mix newly developed matchers and existing ones in your workflow. You must strictly apply to the grammar though, because otherwise your workflow could be refused or cause severe errors. See the grammar description in the 4th chapter for more information.

Note that adding new matchers to COMA requires to adjust the grammar as described below. Adding new workflows, strategies or complex matchers does not require grammar changes, though.

7.4.5 Adding a New Resolution

Resolutions are defined in the Resolution class. In many cases, these predefined resolutions should be sufficient to create an individual workflow. If a new resolution should still become necessary, you have to add a new final integer resolution variable and built a new resolution. This is pretty similar to creating a new workflow (and less complex).

The constructor of Resolution.java gets the id of the resolution, which is defined as a final integer variable at the top of the class.

Now having a resolution instance, Resolution.java offers three important methods: *getResolution1()*, *getResolution2()* and *getResolution3()*. Each method gets a graph object as input (the schema graph) and possibly additional parameters for special cases (so there are several types of *getResolution2()* and *getResolution3()*). They always return a list of the type object, which can be a list of node paths, leaves, children, element names, element types etc., depending on the resolution at hand. Normally, these methods consist of a switch-case-structure again, so each case statement concentrates on exactly one resolution.

To add a new resolution, it is therefore necessary to add a new final variable specifying its name and to add the respective statements in the method *resolutionToString()* and *stringToResolution()* methods, because a resolution has both an *id* and a *name*. In the respective *getResolutionX()* method (X is the type of your resolution), add a new case statement which returns the result set that the resolution obtains. Such case statements are often pretty simple, as the following excerpt from *getResolution1()* shows:

```
// Resolution 1
// getting from a graph a list of paths or nodes
public ArrayList<Object> getResolution1(Graph input){
    if (id==IDOverview.UNDEF){
        return null;
    }
    // TODO: what about user selected matching???
     ArrayList<Object> constituents = null;
    switch (id) {
    case RES1_PATHS:
        constituents = new ArrayList<Object>(input.getAllPaths());
        break;
    case RES1_INNERPATHS:
        constituents = new ArrayList<Object>(input.getInnerPaths());
        break;
    case RES1_LEAFPATHS:
        constituents = new ArrayList<Object>(input.getLeafPaths());
        break;
    case RES1_SHAREDPATHS:
        ArrayList<Path> shared = input.getSharedPaths();
        if (shared!=null){
            constituents = new ArrayList<Object>(shared);
        }
        break;
    case RES1_NODES:
        constituents = new ArrayList<Object>(input.getAllNodes());
        break;
    case RES1_INNERNODES:
        constituents = new ArrayList<Object>(input.getInners());
        break;
    case RES1_LEAFNODES:
        constituents = new ArrayList<Object>(input.getLeaves());
        break;
    case RES1_ROOTS:
        constituents = new ArrayList<Object>(input.getRoots());
     ...
```

For instance, to get all leaf paths of a graph, the resolution LEAFPATHS has to be chosen. In the case statement it can be seen, that the array list *constituents* is filled with the result of *getLeafPaths(),* which the graph instance returns. You have to be careful with the result set, though, because the array list has no further type specification.

Please be aware that the graph object may not offer the necessary getter-methods for very individual resolutions. In this case, you have to create such a getter method in the GraphImpl.java or DirectedGraphImpl.java class first. To do this, you certainly need to understand the graph structure (described in chapter 3).

Adding a new resolution also requires to adjust the workflow grammar.

7.4.6 Adding a New Combination

As introduced before, there are two different types of combinations, the set combination and the similarity combination. COMA stores any combination in the class Combination.java in the coma-matching module. The set and similarity combinations are declared as public static final variables at top of the class, where similarity combinations start with COM_ and set combinations start with SET_. Since a combination has an id and a name, there are methods to get the name of a combination by its id and vice versa. Besides these two attributes, a combination may has a variable *weighted* that specifies in which way match results are to be combined (for instance 30:70 or 50:50).

There are several combine methods that carry out the combination process. They normally get a list of match results A_1, A_2, … A_n and return a match result A', which is the match result after the combination process was performed.

Next to the general method *combine()* there exists also the subordinate methods *combineArrays()* to combine several match results, *combineCube()*, to combine a similarity cube and *combineNotSameObjects()* to combine a list of different kind of match results. These methods are called by the major method *combineArrays()* only and only concentrate on **similarity combinations**.

Further on, the method *setCombination()* concentrates on **set combinations** and has several subordinate *computeSetSimilarity()* methods which are called by it. They only distinguish in their method parameters specification.

To add a new combination, a new combination variable has to be declared and the respective code lines added to *combinationToString()* and *stringToCombination()*. Then, in the respective combine method, the new combination has to be implemented. Combinations are normally distinguished by switch-case-expressions. The actual implementation of combinations is located in the several methods *combine()*, *setCombination()* and *computeSetSimilarity().* Some of these methods rather deal with a similarity

matrix (float[][]) than with the entire match result object. It is advised to examine these methods closer to understand how the different set and similarity combinations are implemented. Also, remember the difference between similarity combination and set combination (described more specifically in chapter 4).

Adding a new combination also requires to adjust the workflow grammar.

7.4.7 Building Workflows at Runtime

COMA is generally able to execute any workflow defined as string, as long as it is conform with the grammar. So instead of implementing new workflows, it is also possible to give COMA such a grammar string; it will check whether it is valid, and if so, carry it out. With this, you can run any possible workflow.

To execute a workflow defined in the workflow grammar (so to execute a grammar string), there exists the method *buildWorkflow()* in the class TreeToWorkflow.java, which is located in the *validation* package of the project coma-matching. This method simply gets a string, which must be conform with the grammar. It then transforms this string into a workflow object, which is conform with COMA. The workflow can now be run like any other workflow.

There are no opportunities to load and write workflows so far, so if this should be desired, a Writer and Loader has to be developed so that a workflow can be saved and be run at a later time again. You can also add a predefined workflow in the database where it will be stored permanently. Either way, you have to understand the COMA grammar to do this.

7.4.8 Grammar Adjustments

If a new similarity measure, combination or resolution is to be added, the workflow grammar has to be changed. For this, it is necessary that you understand the grammar (described in section 4) and have installed ANTLR 3.4, which enables you to edit and test grammars and to convert them into java code.

Ass an example, the following steps are to be performed if a new similarity measure is to be added:

- Add the name of the new matcher to the SIMMEASURE set (Part IV) in the grammar file (ComaWorkFlow.g, located in the resources folder of the sub-project coma-engine).
- Compile the grammar again, using ANTLR 3.4. It will create the respective classes ComaWorkFlowParser.java and ComaWorkFlowLexer.java. Take care that you **compile the grammar in Debug Mode** (so that the Parser class extends DebugParser), otherwise you will encounter errors in the COMA source code!
- Copy these files in the coma-matching module, in the package validation of the main source code folder (remove the old files before, or replace them by the new files).
- Adjust the respective java classes (Strategy, ComplexMatcher, Matcher etc.) as already described above and in chapter 4.

Adding new resolutions and combinations is rather similar; in this case, the new resolution or combination has to be added to the grammar, yet the procedure is the same as already described.

It is also possible to change the workflow rules (PART II) so that different kind of workflows become possible. This would change the concepts of COMA considerably, so should generally be avoided.

```
// ================================================================================================
//     PART IV - LEXER RULES
// ================================================================================================

allowedToken :                  RESOLUTION_1 | RESOLUTION_2 | RESOLUTION_3 | SIMMEASURE |
                                SETCOMBINATION | COMPOSITION | SIMCOMBINATION1 | SIMCOMBINATION2 | RESULT_COMBINATION |
                                DIRECTION | SELECTION_THRESHOLD | SELECTION_MAXDELTA | SELECTION_MAXN | SELECTION_MULTIPLE
                                ;

RESOLUTION_1 :          ( 'paths' | 'innerpaths' | 'leafpaths' | 'nodes' | 'innernodes' | 'leafnodes' | 'roots' | 'shared' | 'uppaths' | 'downpaths');
RESOLUTION_2 :          ( 'selfpath' | 'selfnode' | 'parents' | 'siblings' | 'children' | 'leaves' | 'allnodes' | 'successors');
RESOLUTION_3 :          ( 'name' | 'nametoken' | 'path' | 'pathtoken' | 'comment' | 'commenttoken' | 'datatype' | 'statistics' |
                        'synonyms' | 'constraints' | 'instance_constraints' | 'instance_content' |'instance_content_indirect' | 'nameandsynonyms' );


SIMMEASURE:             ('trigram' | 'soundex' | 'editdistance' | 'vector' | 'datatypesimilarity' | 'featvect' | 'tfidf' |
                        'usersyn' | 'cosine' | 'jaccard' | 'jarowinkler' | 'sim_equal' |
                        'trigramcoma' | 'trigramifuice' | 'trigramlowmem' | 'trigramopt' | 'edjoin' |
                        'levenshteinlucene' | 'levenshteinsecondstring' | 'levenshteinlimes' |'cosineppjoin+fullycached' | 'cosineppjoin+' |
                        'cosinesimmetrics' |
                        'jaccardppjoin+fullycached' | 'jaccardppjoin+' | 'jaccardsecondstring' | 'jaccardsimmetrics' | 'jarowinklerlucene' |
                        'tfidflucenefullycached' | 'tfidflucenefullycachedalternative' | 'tfidfsecondstring' );

SETCOMBINATION:         ('set_average' | 'set_dice' | 'set_max' | 'set_min' | 'set_highest');

COMPOSITION:            ('com_average' | 'com_max' | 'com_min');

SIMCOMBINATION1 :       ('max' | 'min' | 'average' | 'nonlinear' | 'openii' | 'harmony' | 'sigmoid' | 'owa' | 'owa_most' | 'weighted2');
SIMCOMBINATION2 :       ('weighted');

RESULT_COMBINATION :    ( 'intersect' | 'diff' | 'merge');


DIRECTION :             ('both' | 'forward' | 'backward' | 'simple');

SELECTION_THRESHOLD :   ('threshold');
SELECTION_MAXDELTA :    ('maxdelta');
SELECTION_MAXN :        ('maxn');
SELECTION_MULTIPLE :    ('multiple');
```

The picture shows an excerpt from the grammar file where the many identifiers for resolutions, combinations, similarity measures etc. are stored. If some new element is added to COMA, it must be added to the grammar, which must be compiled again and updated in the COMA project. Only then COMA keeps consistent with the grammar and is able to deal with new workflows.

7.5 Adding a New Matcher

In the following, we assume that you want to create a new matcher “SampleMatcher”, which will be implemented in the single class SampleMatcher.java. To get a better understanding in the matcher structure, it is recommended to read the section about the matchers in the 4th chapter.

The following checklist describes how a new matcher can be added to the project:

1. If not already done, create a new package *matchers* in the module coma-matching. It will serve for all matchers added to the project.
2. Create a new class SampleMatcher.java. The class must implement IAttributeObjectMatcher.java, and the required methods have to be overriden. SampleMatcher.java should also extends AbstractObjectMatcher.java, which provides the progress functions.
3. Implement the matcher. As described above, the matcher must be implemented in at least one of the match methods. For additional information, see below.
4. Add a new final variable which represents your matcher, and register it in the methods *measureToString()* and *stringToMeasure()*, because a matcher has an id and a name.
5. In the method *getMatcher()* in Similarity.java, add a new case statement where your matcher is called. The case id is the similarity id your matcher possesses.

The following excerpt demonstrates the implementation of the match method of the date distance matcher:

```
@Override
public void match(IObjectInstanceProvider oip1, IObjectInstanceProvider oip2, IMappingStore mrs)
throws MappingStoreException {

	for (IObjectInstance oiLeft : oip1) {
		for (IObjectInstance oiRight : oip2) {

			ISimilarity similarity = calculateSimilarity(oiLeft, oiRight);

			if (similarity.getSim() < threshold)  {
				continue;
			}

			else if (similarity.getSim() <= 1) {
				mrs.add(new MappingEntry(oiLeft, oiRight, new
					Similarity(similarity.getSim())));
			}

			else {
				mrs.add( new MappingEntry( oiLeft, oiRight, new Similarity(0)));
			}

		}
	}
}
```

Basically, there are two for-loops which are used to compare any object in the left-hand schema with any object in the right-hand schema. Inside the second loop, the similarity between these two objects at hand is calculated, and if they are above the threshold, they are added to the mapping store (resp. ArrayListMainMemoryMapping.java instance).

The method *calculateSimilarity()* is a further method in this class, which gets the two objects, calculates its date similarity and returns a similarity object. As matter of fact, you will create your own methods to calculate the similarity.

7.6 Threshold Management

The basics of thresholds in COMA were already explained in chapter 4.5. A common task might be to get a set of correspondences within a specific confidence range, e.g., the correspondences having a confidence between 0.5 and 0.9.

As already explained, the match result list contains the confidences between all element combinations, which is a value between 0 and 1. The method *getSimilarity()* in MatchResultArray.java is the key method to get a set of correspondences within a specific range. Although such a method does not exist, it can be easily written. For this, the similarity between each element combination has to be checked by *getSimilarity()* and after this it has to be checked whether the similarity is within a predefined minimum and maximum threshold.

This way you are able to go on with a list of correspondences that are in a specific confidence range, to color correspondence lines in the GUI etc.

8 Miscellaneous

8.1 COMA API

The class COMA_API.java in the package coma-integration offers several methods to carry out the most important COMA CE functions, especially obtaining a match result.

The class contains a couple of methods to access the COMA functions. The most important function may be *matchModels()* to match two schemas; these can be either URLs or local files. The method returns a match result which subsequently can be processed by the calling method.

8.2 COMA Database

8.2.1 Overview

All database-relevant classes are stored in the module coma-repository. Instead of “database”, the term repository is widely used in COMA, but actually refers to the database after all.

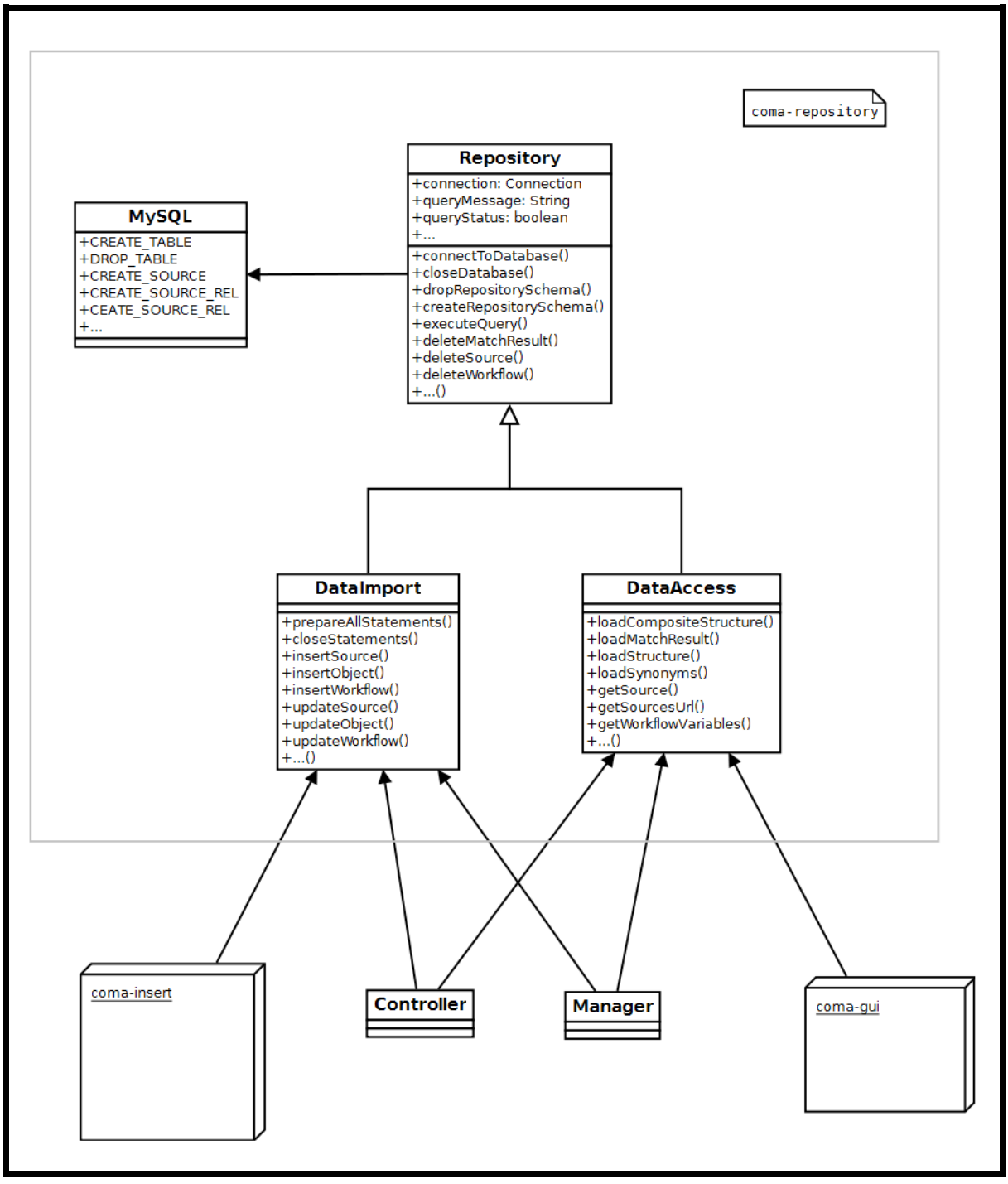
The coma-repository module consists of only 4 classes, as it can be seen in the uml diagram below. Repository.java is the main class to connect to the database. It offers crucial methods to create and close database connections, drop (delete) the database, create a new database, execute queries etc. For deleting the repository, several delete methods are provided.

Still, Repository.java does not offer any methods to perform insert or update operations. For this, DataImport.java is used, which extends Repository.java. It offers the relevant insert() and update()-methods, which are mainly accessed by classes in coma-insert as well as the Manager and Controller.

Next to DataImport.java there is DataAccess.java, which is used to get all kind of information from the database. This class collects many functions to directly get a match result, a source etc. stored in the database, so that another class is not compelled to use any SQL statements. Some methods in DataAccess.java start with “load”, like *loadMatchResult()* while others start with “get”, like *getSource()*. However, they practically do the same, returning some information from the database. For this, the class is especially important for the manager and controller, as well as some GUI classes (mainly some dialogs that need information from the database).

Most of the SQL statements to perform delete, create, update and insert statements are located in MySQL.java. It is only a collection of static final String variables, which are commonly used to create the tables of the database. However, the table names of the database tables are declared in Repository.java.

The URL diagram on the next page gives an overview about the 4 classes and their relation to other important classes and packages in COMA.


coma-repository
Repository
+connection: Connection
+queryMessage: String
+queryStatus: boolean
+...
+connectToDatabase()
+closeDatabase()
+dropRepositorySchema()
+createRepositorySchema()
+executeQuery()
+deleteMatchResult()
+deleteSource()
+deleteWorkflow()
+...()
MySQL
+CREATE_TABLE
+DROP_TABLE
+CREATE_SOURCE
+CREATE_SOURCE_REL
+CEATE_SOURCE_REL
+...
DataImport
+prepareAllStatements()
+closeStatements()
+insertSource()
+insertObject()
+insertWorkflow()
+updateSource()
+updateObject()
+updateWorkflow()
+...()
DataAccess
+loadCompositeStructure()
+loadMatchResult()
+loadStructure()
+loadSynonyms()
+getSource()
+getSourcesUrl()
+getWorkflowVariables()
+...()
coma-insert
Controller
Manager
coma-gui

8.2.2 How to Set up the Database

The database parameters are located in a the file “_coma++.txt”, which is located in the coma-gui project folder (root directory). Next to the database parameters some file paths and urls are defined in this file, but with regard to the database connection only the parameters comaUser, comaPwd and comaURL are of importance.

```
comaUser=
comaPwd=
comaUrl=jdbc:mysql://localhost/coma-project
```

By the way, you can disable a line by adding a number sign (#) at the beginning of the line.

To use the database, you must have a running MySQL database on your computer.

8.2.3 How a New Database is Created

A new database is created by Controller.createNewDatabase(), although the actual task is performed by the data importer. The method *createNewDatabase()* first creates an instance of DataImport.java, which drops the former repository (deletes the DB) and then creates the new repository. The result is an empty DB.

After this is performed, a list parser is instantiated, which loads the abbreviations and synonyms used in COMA. Finally, the default variables are loaded and the database is updated.

The following code snippet shows it in more detail.

```
DataImport importer = new DataImport();

importer.dropRepositorySchema();
importer.createRepositorySchema();

ListParser parser = new ListParser(true);
parser.parseAbbreviation(file_abb);
parser.parseSynonym(file_syn);

importDefaultVariables();
updateAll(true);   // parse and import synonym info

if (mainWindow!=null){
        mainWindow.getNewContentPane().setProgressBar(false);
        setStatus(GUIConstants.DEL_DB_DONE);
}
```

8.2.4 The Database Schema

The COMA Database consists of 6 tables. Table source represents a common source (schema), having a name, a type (e.g., xsd), possibly an url, author, version etc. Table object is closely connected to source, because each schema consists of several objects (“elements” or “nodes”). An object is rather independent, but has a unique source reference. Besides, it has a name, a type (integer, string etc.) and possibly further specifications.

Table instance represents a single instance entry and is therefore connected to table object. An instance is represented by three identifiers: id (a number specifying the instance entry), an element id to specify to which element (object) the instance entry refers and an instance_id, specifying to which instance file this instance belongs. The most important attribute is “value”, where the instance value is stored.

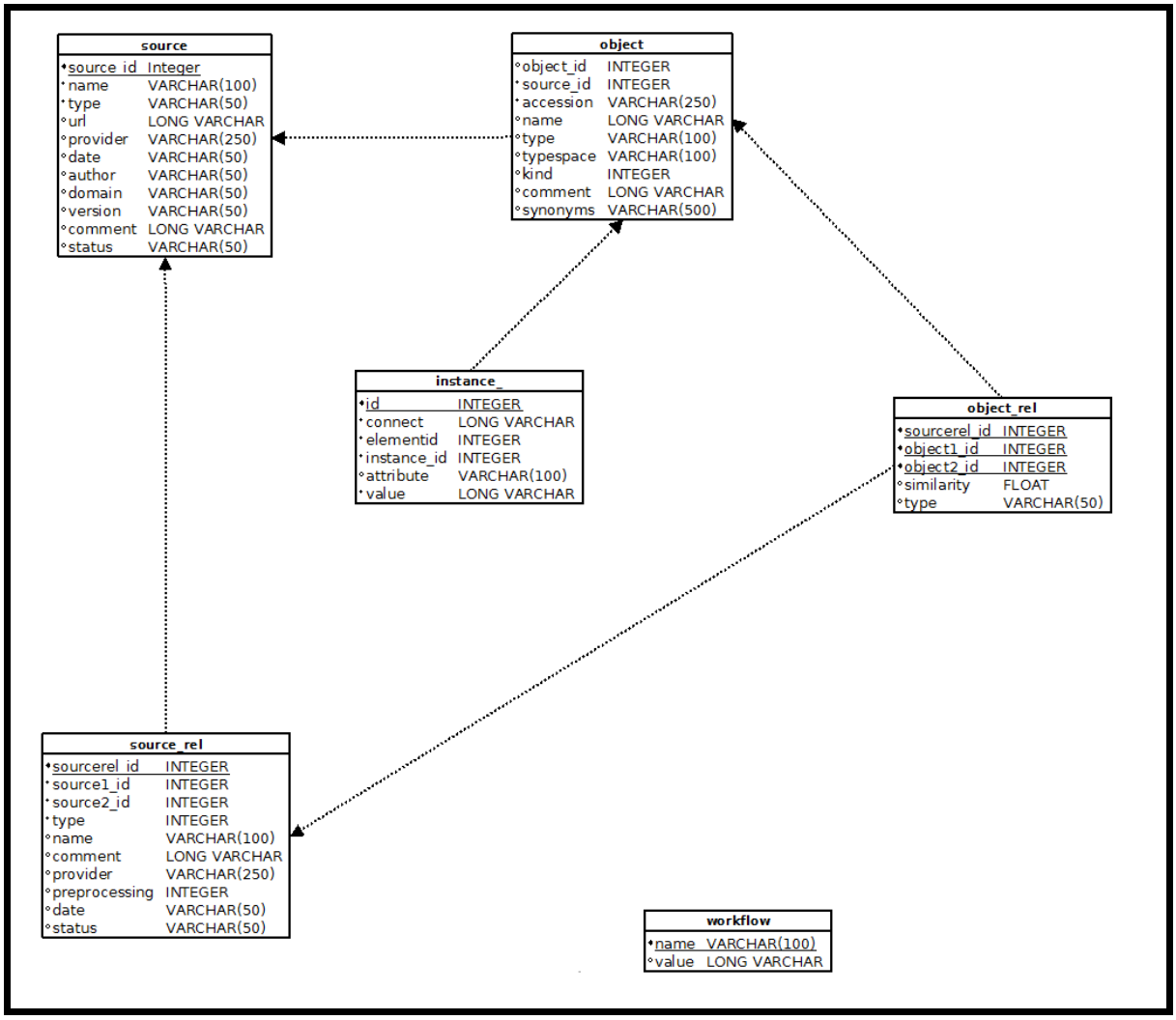


Table source_rel represents a common map, consisting of exactly two sources (a source schema and target schema). Table object_rel represents a common correspondence. As matter of fact, two objects take part in a correspondence, and sourcerel_id specifies to which map the correspondence belongs (remember that there might be several mappings in the repository).

The table workflow is used to store matcher workflows; it is completely independent from the other tables.

Index