Metamorph User Guide

Markus Michael Geipel

February 21, 2012

Abstract

Metamorph is a Java library including an XML based data transformation language, designed to ease dealing with metadata. In this document the design rationale behind Metamorph is discussed, followed by a user guide including many hands-on code examples. As a conclusion several real-world application scenarios are presented.

1 Introduction

Metadata is a central ingredient of any information storage and retrieval system. Being defined as "data on data", metadata provides descriptive information on the items stocked in the system; be it a library, an archive, a museum or a web search engine.

The variety of possibly stored items is reflected in the plethora of existing metadata formats. Obviously, a museum needs to describe its exposition items differently than a library its available books. And even the same book may be described differently depending on the institution holding it: A library has different description needs than a book seller. The first might use a standard such as Marc 21, the latter might use ONIX. Furthermore, standards also differ depending on the country or the language. Finally new technologies bring up new information needs not reflected in existing standards.

In the face of a babylonic confusion of tongues, interoperability is an important issue, and metadata is transformed from one format to another on a regular basis. Transformations are not only performed to exchange data between institutions or to enable cross-collection searches. Metadata also needs to be transformed for indexing or for presentation in user-interfaces (on a webpage as HTML, for example). Despite being such an ubiquitous task, metadata transformation is still a tedious, mostly manual task, with scarce tool support. Software written to perform transformations is often coded from scratch for each and every individual case; despite great potential for component reuse.

The purpose of this document is to explain how the metadata challenge is addressed by Metamorph, a Java library including a domain specific metadata transformation language expressed in XML. In the next section the classical metadata transformation process will be sketched to elucidate the inherent challenges and to motivate the requirements and design goals of Metamorph. Next, the general architecture will be presented, followed by

a description of the language features. Metamorph was applied in various projects at the German National Library. In section 5 these application scenarios will be discussed. A final section on the future prospects and limitations of Metamorph rounds up this document.

2 Metadata Transformation in Practice

Metadata transformation is a complex and sometimes tedious process as its correctness depends on an overlap of domain knowledge and programming skills. First the typical procedure will be skeched followed by a discussion of the resulting challenges.

2.1 Workflow

The first step is to create a Crosswalk or Conceptual Mapping between the two formats. Such a Conceptual Mapping normally consists of a table in human readable form and it is created and maintained by domain experts. Figure 1 shows two simple mapping rules from Pica to RDF, taken from the DNB Linked Data Service.

PICA 3	PICA +	Ind.	Field content	RDF element	Remarks
020	007P	$0	**GKD-Nummer (GKD number)**	rdaGr2:identifierForTheCorporateBody	(DE-588b)...
022	007T	$0	**LoC-Nummer (LoC number)**	rdaGr2:identifierForTheCorporateBody	(DLC)...

Figure 1: Conceptual mapping for the DNB Linked Data Service

On the left we see the respective fields in Pica, followed by an explanation of the content. Next, the target in RDF is defined, followed by remarks, indicating transformations applied to the data. The first line thus reads: Given a Pica+ recode, take the value of field `007P` subfield `0`, prepend `(DE-588b)` and write it to the RDF element `rdaGr2:identifierForTheCorporateBody`. Figure 2 shows a slightly more complex example. In this case values of different subfields need to be combined to fill the RDF element `gnd:preferredNameForTheCorporateBody`. The last column defines this combination by reference to the subfield names.

Based on the conceptual mapping, a piece of software is developed which implements the mapping. This is done by a programmer based the conceptual mapping document.

2.2 Challenges

The just described workflow poses several challenges which shall now be briefly discussed. In section 3 I will refer to them while introducing the architecture of Metamorph.

150	029A		**Körperschaftsname in Ansetzungsform (Authorized form of name of** the **corporate body)**	gnd:preferredNameForTheCorporateBody	$a <$c> / $b <$x> / $b <$x>...
		$a	Hauptkörperschaft (Main corporate body)		
		$c	Ordnungshilfe zu Hauptkörperschaft (Qualifier for main corporate body)		
		$b	Abteilung(en) (Department(s))		
		$x	Ordnungshilfe zu Abteilung(en) (Qualifier for department(s))		

Figure 2: A more complex mapping for the DNB Linked Data Service

2.2.1 From Concept to Code

The transition forms the conceptual mapping to the actual code implementing it is a critical one. A conceptual mapping leaves space for interpretation. The programmer is the one to fill this gap, although he or she lacks the knowledge of the domain specialist. Information flows one-way from domain specialist to programmer. What the programmer really implements cannot be double-checked by the domain specialist as he or she in turn lacks the fluency in the respective computer language the transformation is realized in.

2.2.2 Format Independence

Conceptual mapping between metadata schemata bear a lot of resemblance. The table structure in which the mapping is described is almost always the same. The software side implementing the mappings differs significantly, though. This is due to encoding details of the metadata formats. For instance the code and data structures or classes used load and represent Marc 21 records and pica records in a Java program differ, even though conceptually they are very similar. Both are composed of fields containing subfields. Slight differences in the implementations renders reuse of the code infeasible.

2.2.3 Performance

The input to a transformation from one format to another may well comprise millions of records. Performance is thus an issue. It is also important to note that transformations have to be performed repeatedly as the metadata keeps changing or errors in the mapping are discovered.

3 Architecture

This section introduces the general architecture of Metamorph, pointing out how it addresses the requirements of reusability, transparency and performance.

3.1 Generic Data Structure

At least in the library domain, the organization of the data within a record is fairly similar across formats (Marc, Pica and Mab2, for instance): A record consists of named fields, which in turn contain named subfields. The actual data is stored in these subfields. Many formats also store meta-information such as modification date in the record[1]. Metadata formats in the archive domain in turn tend to exhibit more hierarchical structures. As pointed out in section 2.2.2, a common denominator is needed to enable code reuse. Metamorph thus makes the assumption that the structure depicted in figure 3 may serve as such a common denominator. The structure is fairly general and allows also for hierarchical structures. It turns out that most data structures found in common metadata formats can be mapped to it easily.

In general `Record` forms a self contained and independent unit. It may contain Literals or Entities which themselves may contain further Entities or Literals. A Field in library metadata (e.g. Pica) would map to an Entity, Subfields to Literals. The recursive organization of Entities whould not be called on in this case.

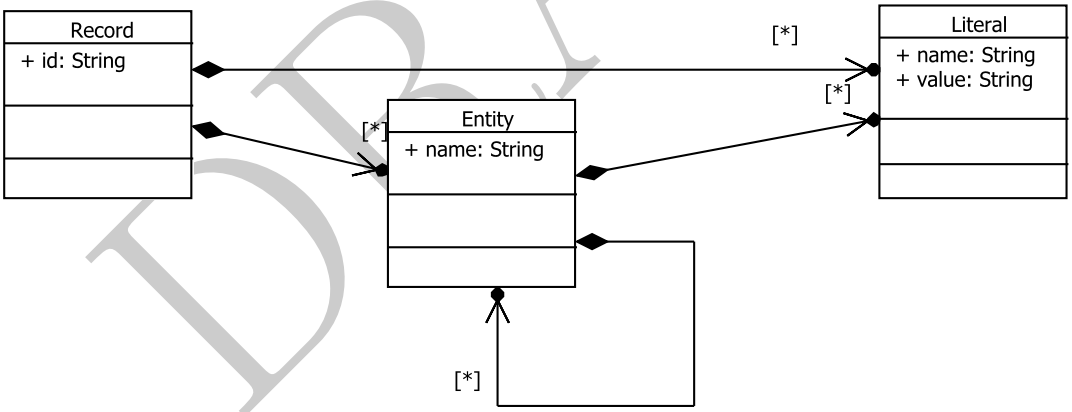

Figure 3: The abstract data model underlying Metamorph

The universality of data structure described in the previous section comes at a cost. To generate it explicitly is costly as it comprises many Java object such as literals, lists or maps, depending on the specific implementation. Handling such generic data structures normally results in convoluted if-infested code. The solution to this dilemma is twofold: Firstly, as described in the next subsection we avoid ever explicitly instantiating the data

[1]Which would actually be meta-meta-data, to be precise.

structure in figure 3. Secondly, we use a domain specific language to handle transformation on the data stream, as sketched in subsection 3.3.

3.2 Data as a Stream of Events

Do we really need to explicitly construct the data structure for each record? No, it turns out that it can be easily serialized to a stream of events. Listing 1 shows the interface, a receiver of such a stream needs to implement. Adding an interface such as the one in

```
public interface StreamReceiver {
        void startRecord(String identifier);
        void endRecord();
        void startEntity(String name);
        void endEntity();
        void literal(String name, String value);
}
```

Listing 1: The interface used to serialize the data structure in figure 3.

listing 2 we can easily build processing chains using independent modules.

```
public interface StreamSender {
        <R extends StreamReceiver> R setReceiver(R streamReceiver);
}
```

Listing 2: The interface of an event stream sender. By virtue of the template R we can efficiently define processing chains via method chaining. See listing 5 for examples.

One module implementing `StreamSender` for example might read a bibliographic record encoded in Pica and translating it to events. The events are received by another module which implements `StreamReceiver` and `StreamSender`, thus forming a pipe or filter element in the chain. It may react on the events and transform the stream and its contents. Finally an arbitrary `StreamReceiver` may form the endpoint of chain by reassembling the events to objects or persisting them to a database, logging them, indexing them. The interfaces shield the intricacies of one element in the chain from the others. This means that parts can be easily exchanged. The modularity gained by this schema enables reuse of software components and admits for a more flexible architecture.

3.3 Generic Transformations

3.3.1 Addressing and Dispaching Data

3.3.2 Data Receivers

3.3.3 Data Processors

3.3.4 Data Collectors

4 Using Metamorph

Figure 4 shows the basic setup for data processing with Metamorph. The processing pipeline starts with the input data which is read by a reader specific to the input format. The reader emits messages according to the `StreamReceiver` interface (listing 1). Due to this interface we are free to plug into the processing pipeline a variety of building blocks. In the majority of use cases the data needs to be transformed in one way or the other, a task that falls to the Metamorph object, depicted in the center of figure 4. The actual transformation performed by Metamorph is encoded in the Metamorph definition file. Finally, a writer condenses the event stream into the target data format.

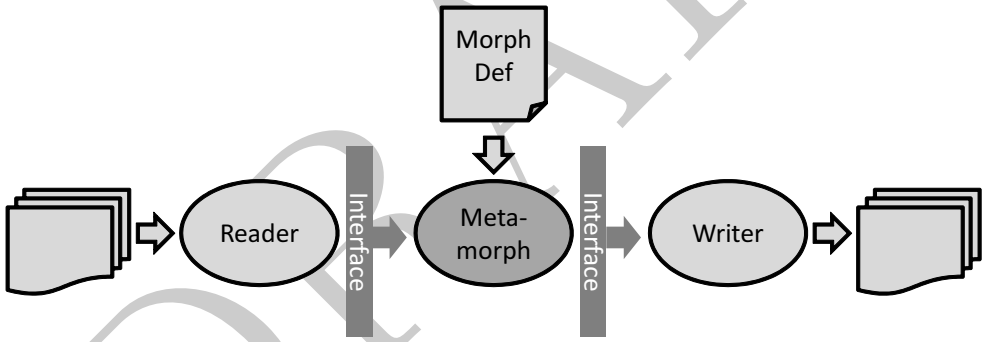

Figure 4: A typical processing pipeline including a Metamorph object for data transformation.

This section explains the how such a setup is implemented in practice. Subsection 4.1 is dedicated to the Java code needed to wire the pipeline parts together, subsection 4.2 focuses on the Metamorph definition language.

4.1 The Java Side

The follwing subsection explain how to create a Metamorph object, how to build a processing pipeline and how to handle exceptions.

4.1.1 Creation

A new Metamorph is created by calling `build()` on MetamorphBuilder. See listing 3. Please note that the Metamorph object is not thread-safe.

```
final Metamorph metamorph = MetamorphBuilder.build("definition.xml");
```

Listing 3: Creating a Metamorph object based on a Metamorph description.

4.1.2 Wiring

The Metamorph object acts as a pipe element in the data stream. See also figure 4 and the explanation in section **??**. This means that we must wire it to a data source (or reader) and a data sink (or writer). Listing 4 shows how. First all elements of the processing chain are created. The wiring is done by calling `setReceiver()`. The call returns its argument, preserving the respective type. Thus the calls can be chained to build up a pipeline as shown in the listing. Finally the processing is stared by calling the respective method on the data source/reader. The method name depends on the reader. In the Metamorph project `read()` is used by convention.

```
// create neccessary objects
final PicaReader reader = new PicaReader();
final Metamorph metamorph = MetamorphBuilder.build("defnition.xml");
final ListMapWriter writer = new ListMapWriter();

//wire them
reader.setReceiver(metamorph).setReceiver(writer);

//start processing
reader.read(input);
```

Listing 4: Putting together a processing pipeline according to the pattern in section **??**.

Listing 5 shows a few more sophisticated wiring patterns, such as adding an additional element, junctions or splitters.

4.1.3 Error Handling

If an exception occurs during the processing of a stream of records, it is back propagated to the first element in the chain. This normally means that processing is terminated which may not be the preferred action. Imagine processing a million records. One normally prefers to log any error but continue the processing. For this reason an error handler may be registered with the Metamorph object. It catches all exceptions occurring in the Metamorph object and below. Listing 6 shows the respective code snippet.

```
//add logging
reader.setReceiver(new LogPipe()).setReceiver(metamorph).setReceiver(writer);

//adding a tee junction
reader.setReceiver(new Tee()).setReceivers(writer1, writer2);

//splitting based on a metamorph description
final Splitter splitter = new Splitter("morph/typeSplitter.xml");
reader.setReceiver(splitter).setReceiver("Tn", writer1);
splitter.setReceiver("Tp", writer2);
```

Listing 5: Advanced wiring.

```
metamorph.setErrorHandler(new MetamorphErrorHandler() {
        @Override
        public void error(final Exception e) {
                // TODO fill in your error handling code
        }
});
```

Listing 6: Registering an error handler.

4.2 Metamorph Definition Language

The transformation a specific Metamoph instance performs are defined in a Metamorph definition in XML. See also figure 4. The structure of the XML is constrained by a schema (`metamorph.xsd`). Listing 7 shows the high level organization of a Metamorph definition.

```
<?xml version="1.0" encoding="UTF-8"?>
<metamorph xmlns="http://www.culturegraph.org/metamorph"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.culturegraph.org/metamorph metamorph.xsd"
        entityMarker="." version="2">
   <meta><!-- Metadata --></meta>
   <functions><!-- Function definitions --></functions>
   <rules><!-- Transformation rules --></rules>
   <maps><!-- Data maps --></maps>
</metamorph>
```

Listing 7: Structure of a Metamorph definition file.

The root element `metamorph` has two attributes: One indicates the Metamorph version the document it intended to work with, the second indicates the character used to separate entity names (see also section 3.3.1). Within the `metamorph` tag there are four sections.

The first and optional one holds metadata for the definition file. The second section – also optional – holds definition of custom functions. See section 4.2.4. The `rules` block defines the actual transformation rules. All of the following subsection refer to them. Finally the optional `maps` block allows to define maps/dictionaries for lookup functionality. See section 4.2.3 for details.

4.2.1 Receiving Pieces of Data

The `data` tag is used to receive literals. Use the `source` attribute to address the literal you want to catch. Listing 8 would receive the value of any literal with name *literalname*, enclosed in an entity named *entityname*. The value is then sent to the downstream `StreamReceiver` under the name *newName*. It is thus the most basic form of mapping data.

```
<data source="entityname.literalname" name="newName" />
```

Listing 8: Receiving values from literals

4.2.2 Processing Pieces of Data

After picking up a literal, its content can be processed sending it to the downstream `StreamReceiver`. Processing steps are added inside the `data` tag. Listing 9 shows an example in which the date of death of an author in the PND is extracted from the Pica records and renamed to the corresponding RDF property (for the complete mapping description see the DNB linked data service documentation).

```
<data name="rdaGr2:dateOfDeath" source="032Aa.a">
  <replace pattern="␣" with="" />
  <regexp match="-((\d*?))$" format="${1}" />
</data>
```

Listing 9: Processing data within the `data` tag

In the PND birth and death of an author are stored both in one subfield (literal in Metamorph speak) in the form 'birth - death' . So the need for processing arises. First we eliminate all whitespaces by using a `replace` operation. Next we apply regular expression matching `regexp` and extract the firs match group (${1}) corresponding to the year of death.

Please note that functions may return zero to n values. If no value is returned, the processing is stopped and nothing will be sent downstream. If for instance a `regexp` does not match, processing stops and there will be no 'rdaGr2:dateOfDeath' in the output stream.

Builtin functions include: `compose`, `constant`, `count`, `regexp`, `replace`, `substring`, `lookup`, `whitelist`, `blacklist`, `isbn`, `equals`, `htmlanchor`, `trim`, `split`, `normalize-utf8` and `occurence`. For the time being, see the test-cases and `schema/metamorph.xsd` for arguments and usage examples.

4.2.3 Looking up Pieces of Data

A certain group of functions takes a map/dictionary as argument: `lookup`, `whitelist`, `blacklist` etc. In this section the usage of such maps will be explained. We start with a simple example of data lookup.

Local Lookup Take for instance an operation in which you want to replace values according to a lookup table: Value 'A' maps to 'print', 'B' maps to 'audiovisual' an so forth. This is accomplished by the `lookup` function. The lookup table is defined inside the `lookup` tag. Listing 10 depicts this situation.

```
<data source="002@.0" name="dcterms:format">
  <substring start="0" end="1" />
  <lookup>
    <entry name="A" value="print" />
    <entry name="B" value="audiovisual" />
    <entry name="O" value="online" />
  </lookup>
</data>
```

Listing 10: Performing a simple lookup operation

Maps The same lookup tables may used in different places in a Metamorph definition. To enable reuse, a map/dictionary can be defined separately from the respective lookup function. In the following listing the `lookup` function refers to the table using the name *material*. Later in the code the actual map is defined using the `map` tag. See listing 11.

External Data Sources The situation might arise that the data used in lookup operations cannot be hardcoded in xml; or at least hard-coding it would be inconvenient. Imagine we want to resolve author ids to author names: Putting all the id-name mappings into the Metamorph definition file is certainly not desirable. To address this issue, any data source implementing the `Map` interface can be connected to the `Metamorph` object as shown in listing 12. The data is referenced in the Metamorph definition file by '*name of map*'.

```
[...]
<lookup in="material">
[...]
<map name="material">
  <entry name="A" value="print" />
  <entry name="B" value="audiovisual" />
  <entry name="O" value="online" />
</map>
```

Listing 11: Defining a standalone map, which can be independently addressed by different lookup functions.

```
//create a Map. Any object implementing Map<String, String> will do
final Map<String, String> map = new HashMap<String, String>();
map.put("one key", "first String");
map.put("another key", "another String");

//tell metamorph to use it during lookup operations
metamorph.putMap("name of map", map);
```

Listing 12: Registering a map with the metamorph object.

4.2.4 Integration of custom Java code

If the predefined functions for data processing do not satisfy your needs, you may define new functions as shown in listing 13. In the definition statement a Java class is bound to a name which is subsequently used to refer to it. There are a few important points to note: The class must implement the `Function` interface. For each attribute provided when referencing the function, a respective `set` method is called right after instantiation. This mechanism correctly handles the types `String`, `boolean` and `int`. A function is instantiated once per use in Metamorph definition and may thus maintain a state. See the JavaDoc for more details, examples and advanced issues such as dealing with state.

```
<functions>
   <def name="my_super_function" class="org.myorg.myfunctions.MySuperFunction" />
</functions>
```

Listing 13: User defined functions.

4.2.5 Recursion

Pieces of data processed with Metamorph are by default sent to the `StreamReceiver` registered with Metamorph. There is however the posibillity to send a piece of data into a

feedback loop. In this case the data reenters Metamorph just as it came from the upstream `StreamSender`. This recursion is accomplished by prepending an '@' to the name of the data as shown in listing 14.

```
<data source="002@.0" name="@format">
   <!-- processing -->
</data>

<!-- catch the data -->
<data source="@format" name="dcterms:format">
```

Listing 14: Prepending '@' to the literal name to enable recursive processing.

This pattern comes in handy when a piece of data is needed at several other places after preprocessing. It relieves you from copying and pasting the same preprocessing steps. It also improves efficiency as Metamorph will perform the preprocessing only one. Be careful though not to build infinite loops by forgetting to rename the data (removing the '@') in the final processing step.

4.2.6 Collecting Pieces of Data

In the case that an output depends on the values from more then one literal, we need to collect literals. Collectors are defined under the `rules` tag, just as `data` tags. Put `data` tags inside the respective collectors to indicate which literals are to be collected. The following paragraphs briefly introduce the different collectors available. Note that all types of collectors except `entity` can be nested, and that post-processing steps can be added by using the `postprocess` tag.

Combine is used to build one output literal from a combination of input literals. The example in listing 15 for instance collects the sur- and forename which are stored in separate literals to combine them according to the pattern 'surname, forename'. There are several important points to note: By default `combine` waits until all at least one value from each `data` tag is received. If the collection is not complete on record end, no output is generated. After each output, the state of `combine` is reset. If one `data` tag receives literals repeatedly before the collection is complete only the last value will be retained.

```
<combine name="gnd:variantNameForThePerson" value="${surname}, ${forename}">
  <data source="028A.a" name="surname" />
  <data source="028A.d" name="forename"/>
</combine>
```

Listing 15: Combining data from two different data sources.

The standard behavior of `combine` can be controlled with several arguments: `flushOn="entityname"` generates output on the end of each entity with name *entityname*. Variables in the output pattern which are not yet bound to a value, are replaced with the empty string. Use `flushOn="record"` to set the record end as output trigger. `reset="false"` disables the reset after output. `sameEntity="true"` will reset the `combine` after each entity end and thus enforce combinations stemming from the same entities only. Note that the implementation only executes a reset if actually needed. Using `sameEntity="true"` has thus no negative impact on performance.

Concatenate collects all received values and concatenates them on record end. `flushOn="entityname"` can be used to concatenate at the end of entity *entityname*.

Choose collects all received values and emits the most preferred one on record end. Preference is assigned according to the order the data sources appear within the `choose` tag. Eligible arguments are `sameEntity` and `flushOn`.

Group is syntactic sugar. Use it to set name, value or both once for an entire group of `data` or collect (`combine`, `choose`, etc.) tags.

Entity collects literals to rearrange them as an entity. Use the argument `name` to assign a name to the entity. Further arguments are `sameEntity`, `flushOn` and `reset`.

For the time being, see the test-cases and `schema/metamorph.xsd` for more information.

4.3 Splitting Metamorph Definitions for Reuse

In a complex project setting there may be several Metamorph definitions in use, and it is likely that they share common parts. Imagine for instance a transformations from Marc 21 record holding data on books to RDF, and Marc 21 records hodling data on authors to RDF. Both make use of a table assinging country names to ISO country codes. Such a table should only exist once. To accomodate for such reuse, Metamorph offes an include mechanism based on XInclude. Listing 16 shows an example in which a `map` is included.

Use the `include` tag from the `http://www.w3.org/2001/XInclude` namespace to insert an external xml file into your definition. The included file must be valid xml itself, containig syntactically valid tags from the Metamorph namespace.

4.4 Testing Framework for Metamorph Definitions

Testing Metamorph definition files nearly always follows a simple pattern. Given the respective definition and an input, a specific output is expected. This can easily be expressed in XML as illustrated in listing 17.

```
<!-- main metamorph definition -->
[...]
<maps>
   <include href="src/test/resources/mymap.xml" parse="xml"
         xmlns="http://www.w3.org/2001/XInclude" />
</maps>
[...]

<!-- mymap.xml -->
<?xml version="1.1" encoding="UTF-8"?>
<map name="island_map" xmlns="http://www.culturegraph.org/metamorph">
         <entry name="Aloha" value="Hawaii" />
</map>
```

Listing 16: Including further XML files into a metamorph definition.

How to integrate such a test definition written in XML into JUnit? JUnit feeds on Java classes. Thus we need to provide such a class as binding point. Listing 18 shows how.

The `RunWith` annotation instructs JUnit to let `org.culturegraph.metamorph.test.TestSuite` handle the testing. Use the `TestDefinitions` annotation to tell `TestSuite` were to look for tests. If no such annotation is found `TestSuite` looks for an XML files with the same name as the binding class. The XML files are expected to be located in the same folder as the binding class. The rationale is that both belong together and separating them would be confusing. Colocating the xml files with the class files is causing trouble with same build environments though. In the case of Maven there is an straight forward remedy: see listing 19

An example test output in Eclipse is shown in figure 5. The root element is the binding class. Its children are the XML files, with the actual tests as leafs.

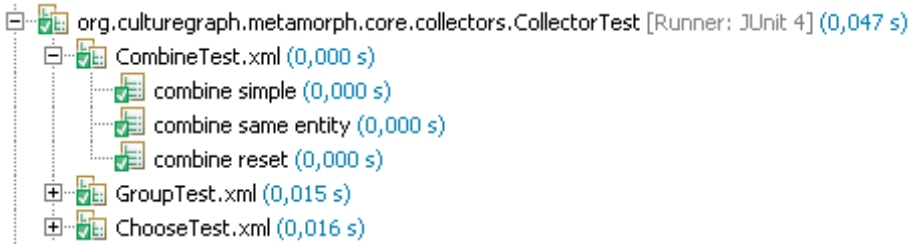

Figure 5: Testcases shown in Eclipse

```
<metamorph-test version="1.0"
        xmlns="http://www.culturegraph.org/metamorph-test"
        xmlns:mm="http://www.culturegraph.org/metamorph"
        xmlns:cgxml="http://www.culturegraph.org/cgxml">
    <test-case name="My Testcase1">
        <input type="text/x-cg+xml">
            <!-- Your test input goes here -->
        </input>
        <transformation type="text/x-metamorph+xml">
            <!-- the metamorph definition you want to test goes here -->
        </transformation>
        <result type="text/x-cg+xml">
            <!-- the expected result goes here -->
        </result>
    </test-case>
</metamorph-test>
```

Listing 17: XML test definition

```
import org.culturegraph.metamorph.test.TestSuite;

@RunWith(TestSuite.class)
@TestDefinitions({"My Testcase1.xml", "My Testcase2.xml"})
public final class MyTest {/*bind to xml test*/}
```

Listing 18: Binding XML test definitions to a test class.

5 Application Examples

5.1 Culturegraph

Culturegraph is a Resolving- and Lookup-Service for bibliographic identifier. A Linked Open Data service that aims to establish shared identifiers for cultural works to ensure these resources can be reliably and persistently referenced.

From a technical view point, the following is happening: metadata from different sources is written to a database and matched to find correspondences. The matches and the original data are indexed and published on a web-portal. To accomplish this, metadata has to be transformed in various stages of the processing pipeline. Figure 6 provides an overview.

```
<testResources>
  <testResource>
    <directory>src/test/java</directory>
      <excludes><exclude>**/*.java</exclude></excludes>
  </testResource>
  <testResource>
    <directory>src/test/resources</directory>
  </testResource>
</testResources>
```

Listing 19: Telling Maven not to ignore resources colocated with java files in the test source directory.

6 Future Prospects

6.1 Open Issues and Limitations

6.2 Possible Extensions

6.2.1 Indexing Data

6.2.2 Writing RDF

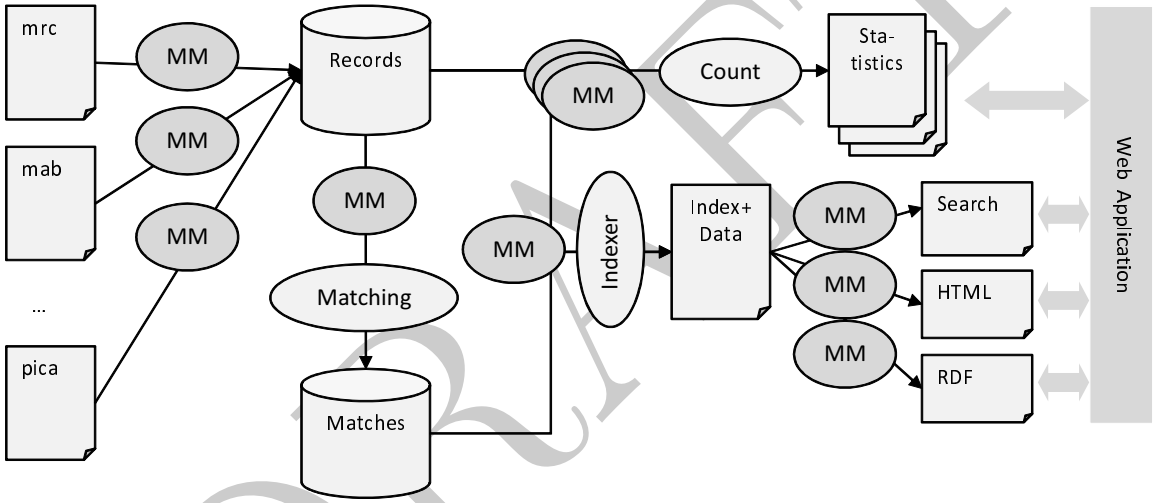

Figure 6: Dataflows in Culturegraph. MM stands for a data transformation using Metamorph