Charles University in Prague
Faculty of Mathematics and Physics

ODCleanStore

Linked Data management tool

Programmer's Guide

Release 1.0.0
November 29, 2012

Authors: Jan Michelfeit
Dušan Rychnovský
Jakub Daniel
Petr Jerman
Tomáš Soukup

Supervisor: RNDr. Tomáš Knap

Contents

1. Introduction

ODCleanStore is a server application for management of Linked Data written in Java. It stores data in RDF, processes them and provides integrated views on the data.

This document serves as the main documentation for developers. It describes basic architecture, implementation, development process, used technologies and other important information relevant for people who want to participate in the development of ODCleanStore.

1.1 What is ODCleanStore

ODCleanStore accepts arbitrary RDF data and metadata through a SOAP webservice (*Input Webservice*). The data is processed by *transformers* in one of a set of customizable *pipelines* and stored to a persistent store (OpenLink Virtuoso database instance). The stored data can be accessed again either directly through a SPARQL endpoint or through *Output Webservice*. Linked Data consumers can send queries and custom query policies to Output Webservice and receive (aggregated/integrated) RDF data relevant for their query, together with information about provenance and data quality.

1.2 Related documents

More detailed information about ODCleanStore from the perspective of a user or an administrator can be found in related documents "User Manual" and "Administrator's & Installation Manual". User Manual also contains definition of user roles, glossary of terms etc.

Other working documents related to development are located at the project's page at SourceForge[1]. The Wiki tool at SourceForge is used for working documents, discussion of new features, description of testing scenarios etc. Not all pages are up-to-date, however, and this document is authoritative in case of conflicts.

[1] https://sourceforge.net/p/odcleanstore/wiki/For%20developers/

2. ODCleanStore overview

An overview of how ODCleanStore works is depicted on Figure 2.1.

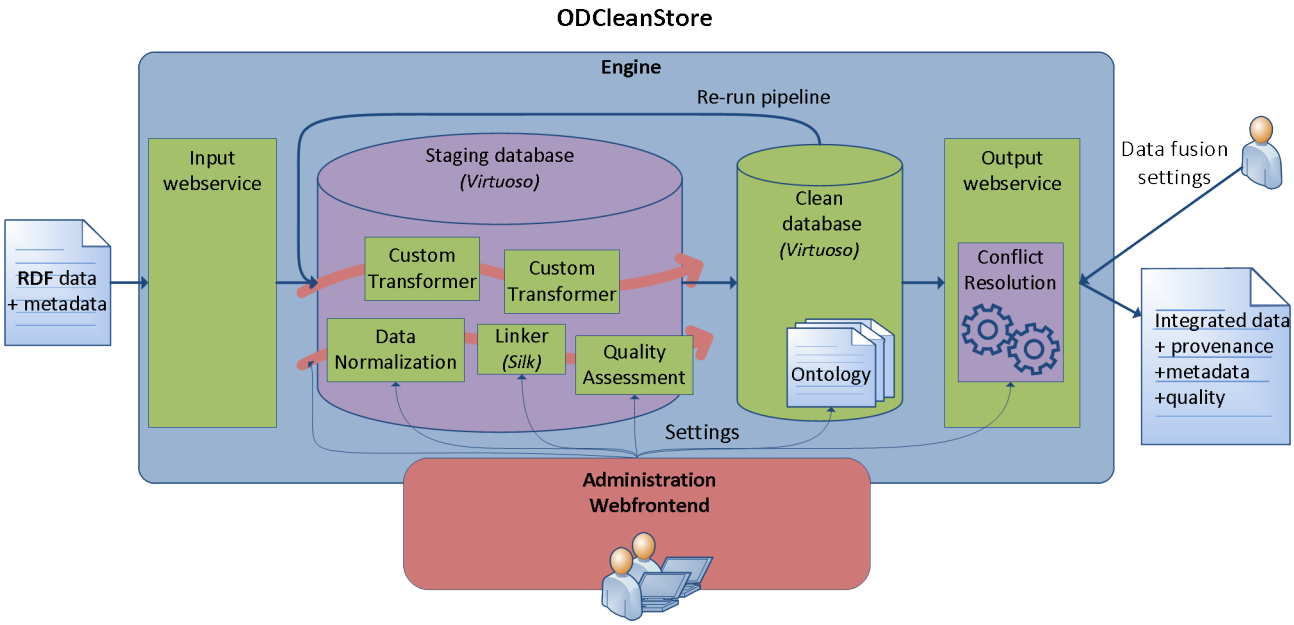

Figure 2.1: Overview of ODCleanStore architecture

The diagram lists all main functional units in ODCleanStore:

- **Engine.** Engine runs the whole server part. It realizes all data processing and starts Input and Output Webservices.
  - **Input Webservice**. SOAP webservice that accepts new data and queues them for processing in the dirty database.
  - **Pipeline processing**. Processes queued data by running a series of transformers in a pipeline on it and moves the data to the clean database.
  - **Ouptut Webservice**. REST webservice for querying over data in the clean database.
- **Query Execution & Conflict Resolution.** Query Execution retrieves all data and metadata relevant for a query asked via Output Webservice. Conflict Resolution then resolves conflicts in the retrieved data, including resolution of `owl:sameAs` links.
- Predefined transformers. Transformers used for data processing that are included by default in ODCleanStore.
  - **Quality Assessment**. Estimates quality of data based on user-defined or generated rules.
  - **Data Normalization**. Transformations of data based on user-defined or generated rules.
  - **Linker**. Generates links (e.g. `owl:sameAs`) between resources in the processed data and contents of the clean database.
  - Other transformers – Other transformers such as Quality Aggregator, Blank Node Remover etc.

- **Administration Frontend.** Web application written in Java from which ODCleanStore can be managed. In Administration Frontend, the user can define pipelines for data processing, rules for transformers, manage ontologies, Output Webservice settings etc.

Each of these parts will be described later in this document. In the source code, the components are divided into several maven projects described in Section 4.3.

2.1 Important concepts

ODCleanStore is about data. More specifically, it works with data represented in RDF[1]. ODCleanStore implements three tasks regarding data:

1. Data processing
2. Storing data
3. Querying over stored data

2.1.1 Data Processing

Data processing is realized by *transformers* that are applied to data being processed by Engine in a *pipeline*. A transformer can be any class implementing the `Transformer` interface but typically it only manipulates (change, add, delete) processed data in database. Several transformers ship with ODCleanStore, such as Quality Assessment, Linker or Data Normalization.

It is important to distinct between a *transformer* and *transformer instance*. By *transformer*, we mean the Java class which implements the `Transformer` interface and is registered in ODCleanStore administration (managed by users in role Administrator). *Transformer instance* is assignment of such transformer to a pipeline. For example, the Quality Assessment transformer is registered in ODCleanStore by default. The user can create two pipelines and assign the Quality Assessment transformer to each of them, thus creating two transformer instances.

Some transformers can be configured in Administration Frontend by *rules*. In general, these rules are grouped to *rule groups*. Rule groups can be then assigned to transformer instances.

See also Section 2.2 Data Lifecycle.

2.1.2 Storing Data

Data are stored using Open Link Virtuoso RDF database. Two instances of this database are used for every deployment of ODCleanStore:

- *Dirty (staging) database* – contains data that are currently being processed. Contents of this instance is not directly visible for data consumers (users in role USR).
- *Clean database* – contains already processed data that are accessible through the Output Webservice to data consumers (users in role USR).

[1] http://www.w3.org/TR/2004/REC-rdf-primer-20040210/

2.1.3 Querying over Stored Data

Querying over stored data is realized by Output Webservice which supports several types of queries (see User Manual). For retrieval of results and resolving conflicts, Output Webservice uses components Query Execution (Section 7) and Conflict Resolution (Section 8).

See also Glossary in Appendix B for explanation of important concepts.

2.2 Data Lifecycle

The lifecycle of data inside ODCleanStore is as follows:

1. RDF data (and additional metadata) are accepted by Input Webservice and stored as a named graph to the dirty database. Data can be uploaded by any third-party application registered in ODCleanStore.
2. Engine successively processes named graphs in the dirty database by applying a pipeline of transformers to it; the applied pipeline is selected according to the input metadata.
3. Each transformer in the pipeline may modify the named graph or attach new related named graphs (such as a named graph with mappings to other resources or results of quality assessment).
4. When the pipeline finishes, the augmented RDF data are populated to the clean database together with any auxiliary data and metadata created during the pipeline execution.
5. Data consumers can use Output Webservice to query data in the clean database. Output Webservice provides several basic types of queries – URI query, keyword and named graph query; in addition, metadata about a given named graph can be requested. The response to a query consists of relevant RDF triples together with their provenance information and quality estimate. The query can be further customized by user-defined conflict resolution policies.
   Data in the clean database can be also queried using the SPARQL query language. While SPARQL queries are more expressive, there is no direct support for provenance tracking and quality estimation.
6. When transformer rules change, the administrator may choose to re-run a pipeline on data already stored in the clean database. Copy of this data is created in the dirty database where it is processed by the pipeline. After that, the processed version of data replaces the original in the clean database.

3. Implementation

3.1 Architecture

3.1.1 Architecture Evolution

The architecture that is depicted on Figure 2.1 was chosen for several reasons.

First, the division into components is natural with regard to the original specification of the software project. All important components are included and we have also kept other concepts, such as two dataspaces for clean and dirty data, or the flexibility of data-processing pipeline.

Second, the selected architecture allowed a clear division of work and enabled a relative independence of each component.

Third, it is a result of a long process of analysis. From the initial vision and requirements specification, which suggested an abstract concept of tranformers run in arbitrary order on data, we extracted several most important cases of transformers that were tightly bound to the system and could run only in a fixed order, only to generalize them back to transformers with a very simple interface and flexible pipelines.

3.1.2 Usage Assumptions

The architecture, and in particular the way editing of pipelines and rules in Administration Frontend is solved, is based on several assumptions about how ODCleanStore will be used. In this section we try to give an overview of these assumptions.

- Data sent to Input Webservice are expected to be reasonably small. ODCleanStore is designed to technically handle even very large data, however, for example results of Quality Assessment may not be relevant for a large graph because a small mistake in several values would decrease the Quality Assessment score for the whole large dataset. Motivation for this assumption is based on planned integration with project Strigil (see related work in Chapter 13).
- Pipeline creators understand the data for which their pipelines are created but not necessarily understand data for pipelines created by other users. For this reason, editing is limited to an author of a pipeline so that a pipeline creator's pipelines cannot be broken by other users without his knowledge.
- The team of people working with a deployed ODCleanStore may be decentralized and open, for example students using a shared instance for their own projects. For that reason the system should be robust, permissions should be reasonably limited, a single user shouldn't be able to block the whole system and there should be a possibility to correct wrong configuration for administrators.
- We expect that pipeline creators will usually be data producers using the Input Webservice at the same time. These users need some technical knowledge – at least they should understand RDF and SPARQL. On the other hand, data consumers need not be technically skilled or understand RDF, even though it is benefitial. Data for these users may be presented in the HTML output of Output Webservice or through a

third-party interface.

3.1.3 Architectural Features

3.1.3.1 Components

ODCleanStore consists of several components that are (mostly) loosely-coupled only through a simple interface specified in advance. This made it easy divide tasks among developers and enabled them to work independently.

3.1.3.2 Internal Interfaces

In order to minimize interfaces between parts of ODCleanStore, to minimize system requirements and to make the system more robust, we decided to prefer communication through data shared in database.

There is no direct interface between Engine and Administration Frontend, but the Administration Frontend saves all configuration to a relational database from which the Engine can retrieve it. This enbles updates of settings in transactions, prevents synchronization issues and enables the two parts to run completely independendently (even on separate machines).

Transformers run in a pipeline are isolated and don't know about each other. Instead of passing data to be transformed through an interface in memory, only the names of named graphs where data are stored are passed to each transformer. This enables to write transformers that are oblivious to ODCleanStore as much as possible and only need to work with data by manipulating the database. Also, it gives transformers the full power of the SPARQL/SPARUL language. Although we should note that in practice, the transformer implementation may be tied with the use of Virtuoso as the underlying database, this is not such a downside because Virtuoso is one of the most popular RDF databases.

3.1.3.3 Extensibility

The main point of extensibility are custom transformers (see Section 9.3). Because one only needs to implement a simple interface and is not bound to any specific technology (except of the limitations of the underlying Virtuoso database), transformers provide a very powerful way how to extend data processing capabilities.

3.1.3.4 Interoperability

The external interfaces are implemented using standard technologies (SOAP for Input Webservice, REST for Output Webservice) and standard formats (RDF/XML, Turtle/Notation3). This should minimize the effort for integration with third-party applications communicating with ODCleanStore. We also provide a Java library for accessing the Input Webservice to futher minimize the effort.

3.1.3.5 Used Technologies

The choice of Java for implementation and Virtuoso as the underlying database ensure platform independence.

Since Java is a very wide-spread language, ODCleanStore can be extended with minimum effort for learning new technologies (e.g. when adding new transformers).

3.1.3.6 Scalability

Although currently the Engine is processing data sequentially, it is designed for parallel processing in the future. It could even be extended to work in a distributed manner on several machines.

Most of the work the Input Webservice has is with data processing in pipelines. Since pipelines can run independently, each Engine instance could even use a dedicated database instance for dirty data.

On the other hand, the Output Webservices uses the clean database in a read-only manner and thus could be also deployed on several database instances if database replication is put in place.

3.2 Other Requirements

The assignment of the project impose several additional requirements. This section lists how they were satisfied.

It should be easy to incorporate other components, such as a component computing popularity of the data sources. This requirement is satisfied with the introduction of custom transformers.

The application will involve graphical user interface enabling management of all kinds of policies etc. Administration Frontend enables management of all relevant settings. In addition, several user roles are supported and user accounts can be managed from Administration Frontend.

The application will run at least on Windows 7, Windows Server 2008, Linux. ODCleanStore requires Java Runtime Environment and Virtuoso installation, both supported on all of the listed platforms. ODCleanStore has been tested on Windows 7, Windows Server 2008 and Debian distribution of Linux.

Application will be freely available under Apache Software License. ODCleanStore is published under the required license (see Administrator's & Installation Manual) and source codes available at a public repository at SourceForge.net

3.3 Used Technologies

3.3.1 Implementation Language

The chosen language for implementation is Java. The reason is that there are many libraries and tools required for implemenation accessible in Java, it enables platform independence (one of the requirements on ODCleanStore) and also it is very wide-spread so that developers don't need to learn a new syntax.

3.3.2 Database

Openlink Virtuoso[1] is used as the underlying data store. It is the most popular RDF storage with a solid support. Both a commercial version and Open Source edition[2] exists.

RDF data store provided by Virtuoso supports reasoning, most notably `owl:sameAs` link resolution, which proved essential for Query Execution/Conflict Resolution components. Virtuoso also provides a relational database which relieves us from the need of another database for that purpose.

The downside is somewhat buggy behavior (especially SPARQL query parsing) and lack of working support for transactions with RDF data.

3.3.3 Administration Frontend

Apache Wicket[3] is used for implemenation of the Administration Frontend. It is a component system for writing web applications in Java. Advantages are proper mark-up and logic separation, POJO data model and rapid development of this particular type of web application. Wicket was also chosen by our sister project Strigil[4] so a more tight integration of the two tools may be possible in the future.

Spring[5] is used for simplified access to the relational database and transaction management. Use of Hibernate[6] for the DAO layer was rejected because of integration problems with Wicket.

3.3.4 Libraries

Jena

Apache Jena[7] is a library for manipulation with RDF data. It supports represenation of RDF data in memory, parsing, loading and updating of RDF models. In ODCleanStore, it is used mainly for representation of RDF triples (quads) and serialization of RDF, because other features proved problematic when used with large data.

We chose Jena over its alternative Sesame[8] because it supports working with named graphs through NG4J and we had previous experience with it.

NG4J

NG4J[9] extends Jena with named graphs API.

[1] http://virtuoso.openlinksw.com/
[2] http://virtuoso.openlinksw.com/dataspace/dav/wiki/Main/
[3] http://wicket.apache.org/
[4] http://strigil.sourceforge.net/
[5] http://www.springsource.org/
[6] http://www.hibernate.org/
[7] http://jena.sourceforge.net/
[8] http://www.openrdf.org/
[9] http://wifo5-03.informatik.uni-mannheim.de/bizer/ng4j/

Restlet

Restlet[10] is an open source lightweight RESTful web framework for Java. Output Webservice is built on Restlet.

SLF4J

SLF4J[11] is a flexible and efficient library used for logging.

3.3.5 Development Tools

Maven

Apache Maven[12] is used as the build tool. It was chosen over Apache Ant because Maven saves work with its many available plugins (such as Maven Jetty Plugin) and offers simple management of dependencies.

Git

Git is used as the version control system. We used it because it enables simple manipulation with branches and merging but most importantly it is less dependent on a central server whose potential unavailability was identified as a potential risk.

[10] http://www.restlet.org
[11] http://www.slf4j.org/
[12] http://maven.apache.org/

4. Setting up Development Environment

4.1 Quick Start

4.1.1 Tools

In order to to prepare environment for building ODCleanStore, first make sure to have installed all necessary tools:

- Java Development Kit[1] version 6 or newer
- Git version control system[2]
- Apache Maven[3]

Also, make sure you have all the binaries used in the following steps on your classpath.

4.1.2 Obtaining sources

ODCleanStore sources are hosted at SourceForge.[4] Use git to check out the sources:

```
git clone git://git.code.sf.net/p/odcleanstore/code odcleanstore-code
```

The above command will create a local clone of the repository in `odcleanstore-code` directory.

4.1.3 Building binaries

Move to directory `odcleanstore` within your local clone of the repository which contains the root maven project (`pom.xml`). Then build the project using maven:

```
cd odcleanstore-code/odcleanstore
mvn clean package install
```

After that, Engine binaries can be found in directory `odcleanstore/engine/target` and the WAR file of Administration Frontend at `odcleanstore/webfrontend/target`. Now you can deploy the application as described in Administrator's & Installation Manual.

[1] www.oracle.com/technetwork/java/javase/downloads/
[2] http://git-scm.com/
[3] http://maven.apache.org/
[4] http://sourceforge.net/p/odcleanstore/code/

4.2 Repository structure

4.2.1 Branches

There are several branches in the git repository. The latest development version of is on branch `master`. Then there are release branches for each release named `release-0.1.x`, `release-0.2.x` etc. which contain stable versions for releases packages. Each commit that was used to prepeare a release package is labeled with a tag `release-`<*version*>. Finally, there are feature branches prefixed with `feature-`.

Common development takes place on branch `master`. New feature branches are created for features that may or may not be accepted or that need time to be finished before they are applied to `master`. When whey are finished, they are merged to `master` and the branch may be removed in time. Release branches stem from `master` for every new major release. Fixes and modifications for minor releases take place on release branches and may be merged from/to `master`.

4.2.2 Directory structure

This is an outline of directory structure in the git repository:

`data/`

- `initial_db_import/` – database import files
  - `clean_db/` – SQL files to be imported to the clean database
  - `dirty_db/` – SQL files to be imported to the dirty database
- `odcs_configuration/` – the default ODCleanStore configuration file
- `virtuoso_configuration/` – configuration files for Virtuoso database instances

`doc/` – documentation sources (in LaTeX)

`odcleanstore/` – Java sources

- `backend/` – sources of `odcs-backend` artifact (transformers, Query Execution, Conflict Resolution)
- `comlib/` – sources of `odcs-comlib` artifact (code related to sending data to Input Webservice)
- `conf/` – configuration files for development in Eclipse
- `core/` – sources of `odcs-core` artifact (common code shared by other artifacts)
- `engine/` – sources of `odcs-engine` artifact (Engine component)
- `installer/` – sources of `odcs-installer` artifact (ODCleanStore installer)
- `inputclient/` – sources of `odcs-inputclient` artifact (Java client for Input Webservice)
- `simplescraper/` – simple Input Webservice import tool
- `simpletransformer/` – example of a custom transformer
- `webfrontend/` – sources of Administration Frontend
- `pom.xml` – the root maven POM file

4.3 Maven Build

The project is divided into several artifacts. The root POM (`pom.xml`) in `odcleanstore` directory defines the parent project while each component of ODCleanStore is in a separate artifact in subdirectories of `odcleanstore`. These artifacts are:

- `odcs-core` – common code shared by other artifacts; also defines interfaces for custom transformers
- `odcs-backend` – predefined transformers, Query Execution and Conflict Resolution
- `odcs-engine` – Engine component (running Input and Output Webservice and the data processing queue)
- `odcs-comlib` – components shared by `odcs-inputclient` and Input Webservice
- `odcs-webfrontend` – Administration Frontend web application
- `odcs-installer` – ODCleanStore installer
- `odcs-inputclient` – client library for Input Webservice (provides a Java API for accessing the webservice)
- `odcs-simplescraper` – a simple command-line tool for importing data through Input Webservices
- `odcs-simpletransformer` – example of a custom transformer

It is recommended to always build using the root POM. Building from subprojects may require issuing `mvn install` on the root POM first. The entire project can be build with the following command:

```
cd odcleanstore-code/odcleanstore
mvn clean package
```

There are two profiles in addition to the default one in the root POM.

- `javadoc` profile – enables generation of javadoc (which is disabled in the default profile).
- `systest` profile – enables unit tests in `systest/` subdirectories, which test functionality related to the database. In order to run these tests, a new Virtuoso instance with settings as in `/data/virtuoso_configuration/virtuoso.ini-test` must be set up first.

Maven build for the selected profile can be executed with command line option `-P`:

```
mvn clean package -P javadoc
mvn clean package -P systest
```

5. Shared Code

Code shared by multiple components in ODCleanStore is extracted to Maven artifact `odcs-core`. It contains:

- Classes for accessing configuration from the global configuration class.
  Java package: `cz.cuni.mff.odcleanstore.configuration`
- Helper classes for accessing the Virtuoso database.
  Java package: `cz.cuni.mff.odcleanstore.connection`
- Helper classes for imports of data to Virtuoso database from files.
  Java package: `cz.cuni.mff.odcleanstore.data`
- Classes related to transformer interface.
  Java package: `cz.cuni.mff.odcleanstore.transformer`
- Definitions of vocabularies used in ODCleanStore.
  Java package: `cz.cuni.mff.odcleanstore.vocabulary`
- And other utility and helper classes, such as unique URI generators, filesystem utilities etc.

5.1 Configuration

The global configuration can be accessed using static methods of `ConfigLoader`. First, the configuration must be loaded using `loadConfig()` methods. Both Engine and Administration Frontend ensure that this is done as soon as they start so that other components may access the configuration already when they are loaded. See Administrator's & Installation Manual for description of the configuration file.

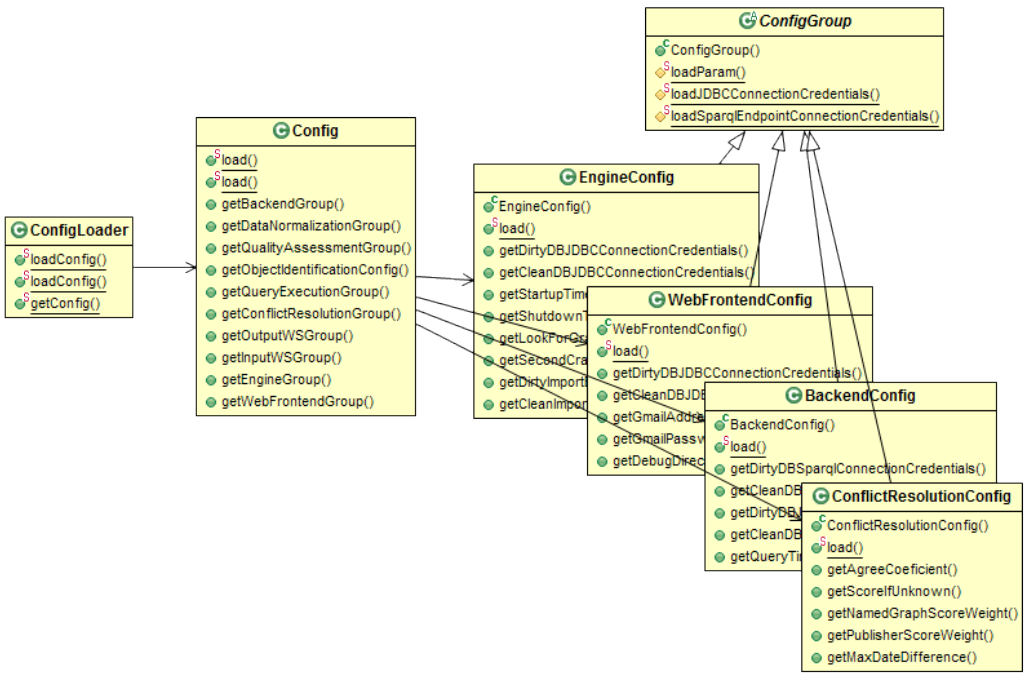

Figure 5.1: Diagram of (selected) configuration classes

Configuration for each component is in classes called `XXXConfig` inheriting from `ConfigGroup`. Each instance of `ConfigGroup` loads configuration relevant only for that component so as to minimize dependencies. Configurations for all components are grouped in `Config` class accessible from `ConfigLoader`.

5.2 Database Access

Classes `JDBCConnectionCredentials` and `SparqlEndpointConnectionCredentials` are containers for information necessary for connecting to the database.

Administration Frontend uses its own database access layer using Spring templates. The rest of ODCleanStore should use classes `VirtuosoConnectionWrapper` and `WrappedResultSet`. These two classes provide methods for both querying and updating the relational database and working with SPARQL. They also provide methods for conversion of Virtuoso SQL types to Java types, working with transactions etc. Note that SPARQL update operations should be executed with log level set to AUTOCOMMIT (default).

Use of `VirtuosoConnectionWrapper` is also recommended for implementation of custom transformers. An example of how it can be used is in Listing 5.1.

```
@Override
public void transformGraph(TransformedGraph inputGraph, TransformationContext context)
    throws TransformerException {
  VirtuosoConnectionWrapper connection = null;
  WrappedResultSet resultSet = null;
  try {
    connection = VirtuosoConnectionWrapper.createConnection(
        context.getDirtyDatabaseCredentials());

    String query1 = "SPARQL SELECT ?s WHERE {?s ?p ?o}";
    resultSet = connection.executeSelect(query1)
    while (resultSet.next()) {
      String s = resultSet.getString("s");
    }

    String query2 = "SPARQL INSERT INTO <a> { <b> <c> <d> }";
    connection.execute(query2);
  } catch (DatabaseException e) {
    throw new TransformerException(e);
  } catch (SQLException e) {
    throw new TransformerException(e);
  } finally {
    if (resultSet != null) { resultSet.closeQuietly(); }
    if (connection != null) { connection.closeQuietly(); }
  }
}
```

Listing 5.1: Example of programmatic access to Virtuoso database from a transformer

5.3 Data Import

When importing data to Virtuoso, one should use the `GraphLoader` class or methods in `VirtuosoConnectionWrapper`. These make Virtuoso import the data itself. Other methods, such as using the Jena library may fail when used with too large data. Note that when importing from a file, it must be in directory listed in `DirsAllowed` directive of Virtuoso. See also `engine.clean_import_export_dir` configuration option.

6. Engine

6.1 Purpose

Engine runs the whole server part. This is the component that actually processes the RDF data. It also starts Input and Output Webservices and thus provides the outer interface for accepting data and querying over data.

6.2 Implementation

Class `Engine` contains the `main()` method which represents the entry point of the application. It checks the environment, loads and validates configuration and starts the services that Engine consists of. Thereafter, Engine periodically updates its state in the database (for monitoring from Administration Frontend) and waits for shutdown (using the standard JVM shutdown hook).

There are timeouts for service initialization and shut down. When the timeout is exceeded, the service is finished forcibly and Engine stops.

6.2.1 Services

Engine consists of three independent services: `InputWSService`, `OutputWSService`, and `PipelineService` for Input Webservice, Output Webservice and pipeline processing, respectively. Services inherit from a common base class `Service`. Services are initialized and started as soon as Engine is started and can be asynchronously shut down when Engine is being shut down. Services have their own state and inform `Engine` about its changes.

Engine is implemented as a multithreaded server. Multiple requests for both Input and Output Webservice can be handled at the same time. The data processing part processes named graphs sequentially, one at a time. Engine is implemented so that extending to parallel data procesing should be simple, however.

6.2.2 Transactional Processing

Data processing in ODCleanStore is done by running a pipeline of transformers on the processed set of named graphs. Running a pipeline is a long-lasting operation on potentionally large data. Databases, including Virtuoso, usually process such operations in a non-transactional way. Engine, however, ensures a transactional character of running a pipeline.

As long as used transformers adhere to the contract specified in Section 9.2, pipelines have all ACID properties. In addition, an increased attention was given to robustness of Engine to errors. The foundation of this robustness is in implementation that ensures that Engine can be shut down at any time without permanent loss of data integrity. These features were accomplished in the following way:

1. RDF data processing is divided into a sequence of operations that move from one consistent state to another consistent state.

2. All changes of state data at the beginning and end of each operation are executed strictly in a context of single transaction in the relational database.
3. In case of an error during an operation, the operation can be always either rolled back or finished.
4. Data processing is executed in the isolated dirty database instance so that inconsistent states of data being processed are invisible to the outside.
5. Since moving data from the dirty database instance to the clean database instance cannot be executed in a transaction, the named graphs inserted to the clean database are made invisible to Query Execution by adding a special prefix to their names. Query Execution ignores all graphs with this prefix. After the import is complete, the named graphs are renamed and made visible for queries. If the new data in the clean database replace older data after re-running a pipeline on them, the old data would be made invisible at the same time by adding the special prefix to them and deleted afterwards.

Several other mechanisms for transactional processing provided by Virtuoso were examined during development. They proved very limiting and often even caused the database instance to crash, however. Transactions for RDF data in Virtuoso were rejected for this reason and transactional properties are ensured by the mechanism presented above.

All services are also resilient to a loss of database connection. Engine doesn't need to restart when the connection is regained again. Input Webservice and pipeline processing require connection to both database instances, Output Webservice needs only the clean database instance.

6.2.3 Database Access

Engine uses class `DbOdcsContext` extending `DbContext` which wraps `VirtuosoConnectionWrapper` and provides transactional processing of relational data. Class `SQL` contains all SQL queries used by `DbOdcsContext`.

Note that relational database (in the clean database instance) is also the only means of communication with Administration Frontend. Settings for Engine managed in Administration Frontend are written to relational database from where Engine loads it and vice versa – Engine updates its status and state of graphs processing in the relational database from where Administration Frontend can load it.

6.3 Input Webservice

6.3.1 Purpose and Features

Input Webservice is a multithreaded SOAP webservice that accepts new data and queues them for data processing by a pipeline. It is implemented with a streaming SOAP message processing so that memory usage is minimized and even large data can be accepted. Input Webservice also supports secure communication over HTTPS.

The counterpart of Input Webservice on the client side is `odcs-inputclient` library which is provided to data producers for convenience. Shared parts of Input Webservice and `odcs-inputclient` are in library `odcs-comlib`. `odcs-comlib` is implemented with

minimum memory requirements compared to the standard jax-ws implementation. Received data are evaluated as soon as they are received and errors are reported to the client immediately (unlike jax-ws). Streaming processing of the SOAP message is implemented using `javax.xml.parsers.SAXParser` supplemented by SOAP 1.1 envelope schema validation in `odcs-comlib`.

WSDL and XSD schema of the SOAP service are parts of Input Webservice as embedded resources and dynamically served according to the HTTP request.

When Input Webservice is started, it starts a HTTP server listening on a given port and then tries to run recovery. If there are any unfinished requests from the previous run, then their records in the database and data files are deleted. HTTP 503 – Service unavailable response is given until recovery is finished.

Shutdown of the service stops all pending requests and the HTTP server.

6.3.2 Implementation

The implementation is in classes `SoapMethodInsertExecutor` extending `SoapMethodExecutor` in a way similar to the standard `DefaultHandler` for `SAXParser`. `SoapMethodExecutor` is part of `odcs-comlib` and filters out parts of the message that belong to the SOAP protocol. The main execution part of message processing is in the `InsertExecutor` class which subsequently takes input parameters and saves received data to the filesystem as files with suffix `-d.ttl` or `-d.rdf` for the payload data, suffix `-m.ttl` or `-m.rdf` for metadata and `-pvm.ttl` or `-pvm.rdf` for provenance metadata, depending on the serialization format (Turtle or RDF/XML). In case of success, it signalizes that a new named graph was accepted to pipeline service which will eventually run a transformer pipeline on the new data.

If updating the state of processing in the relational database fails, Input Webservice retries with in an interval defined in global configuration. If the process of receiving data fails in other cases, `InsertExecutor` throws an `InsertExecutorException` exception which can assemble a `InsertException` SOAP message for the client. These messages are sent in `InputWSHttpServer` together with other errors in the SOAP protocol. Created files are deleted on error and so are records of the import in the relational database.

6.4 Pipeline Service

6.4.1 Purpose

Pipeline Service is responsible for processing of data which are marked to be processed or deleted. Data processing is realized by transformers that are applied to a set of related named graphs in a pipeline. Pipelines work exclusively on data in the dirty database instance and pipeline service is responsible to move the data to the clean database when a pipeline finishes successfully.

Data to be processed are either new named graphs stored through Input Webservice, or data that were marked for deletion or for processing in Administration Frontend. The last case occurs when a user chooses to re-run a pipeline on data already stored in the clean database.

Copy of this data is created in the dirty database where it is processed by the pipeline. After that, the processed version of data replaces the original in the clean database.

6.4.2 Graph States

Every named graph inserted through Input Webservice has a record in relational table `DB.ODCLEANSTORE.EN_INPUT_GRAPHS` with an associated state, among other things. The states can be:

`IMPORTING`
: The named graph is being imported through Input Webservice.

`QUEUED_FOR_DELETE`
: The named graph is queued for deletion. When a graph is deleted, all related graphs in both database instances are deleted. Related temporary files are deleted as well.

`QUEUED_URGENT`
: Reserved for future use.

`QUEUED`
: Named graph is queued to be processed by its respective pipeline.

`PROCESSING`
: Named graph is being processed in a pipeline. That means the graph is loaded from temporary Input Webservice files or the clean database, and pipeline transformers are applied to it. All settings for the pipeline including the plan of transformers to execute and their assigned rule groups are loaded only when the graph transitions to this state so that their consistency during the pipeline processing is ensured.

`PROCESSED`
: Transformers were successfully applied to the named graph and the data are being moved to the clean database into named graphs whose name starts with a special prefix hiding them from Query Execution (see point 5 in Section 6.2.2). Let us call this prefix the *temporary prefix*.

`PROPAGATED`
: The processed named graphs were all moved to the clean database. If there was an old version of the data, the old named graphs are prefixed with the temporary prefix.

`OLDGRAPHSPREFIXED`
: The temporary prefix is removed from the processed graphs and the possible old versions of named graphs, now prefixed with the temporary prefix, are removed.

`NEWGRAPHSPREPARED`
: Related graphs in the dirty database are deleted.

`FINISHED`
: The processing of the named graphs was successfully finished and the data are stored in the clean database.

`DELETING`
: The named graphs are being deleted in the clean and/or dirty database instance. Possible temporary files stored by Input Webservice will be also deleted.

`DELETED`
: The named graph was successfully deleted.

`WRONG`
: An error occurred during the named graph processing.

`DIRTY`
: If an error occurs while the named graph is in state `PROCESSING` or `PROCESSED`, the graph is first marked as `DIRTY` and Engine cleans up unfinished work and only after that the graph is moved to state `WRONG`.

6.4.3 Implementation

Data processing in Engine is driven by settings in the relational database which are managed from Administration Frontend.

Engine logs its activity to a log file called odcs.engine.log in the working directory of Engine. In addition, there is a log file for each transformer instance where the respective transformer logs its activity. These transformer instance logs are in the working directory of the respective transformer.

When shutdown is called on the pipeline processing service then if there is a running transformer, its `shutdown()` method is called. Engine waits until the current operation is finished or the given timeout is exceeded and the finishes the pipeline processing service.

Running a pipeline is transactionally safe, as explained in Section 6.2.2. If an error is caused by Engine (e.g. when it is shut down forcibly), the pending operations will be finished eventually. When an error occurs and a graph is in `PROPAGATED`, `OLDGRAPHSPREFIXED` or `NEWGRAPHSPREPARED`, the pipeline service continues in a forward way on recovery; other temporary states are rolled back. If an error is caused by a transformer, then the named graph is moved to state `WRONG` and it is up to the user to decide what to do next with the graph. An overview of failed graphs is in Administration Frontend. The reason why the transformer failed can be fixed and the pipeline re-run on the graph, or the graph may be queud for deletion.

If an error is caused by unavailability of a Virtuoso database, Engine detects it and pending processing is retried when Engine is started again.

Classes that cover implementation of pipeline service are `PipelineService`, which handles processing of named graphs in different states, `PipelineGraphManipulator` for manipulation with named graphs (e.g. moving between database instances) and `PipelineGraphStatus` for working with state data (e.g. loaded transformer instances for the processing pipelines). `PipelineGraphTransformerExecutor` called from `PipelineService` is responsible for executing transformers on named graphs and `TransformationContext` and `TransformationGraph` are implementations of context objects passed to transformers.

6.5 Output Webservice

6.5.1 Purpose

Output Webservice is a RESTful webservices for queries over data in the clean database. More details about types of queries and request format are described in User Manual.

6.5.2 Implementation

Output Webservice is built on top of the Restlet library. `OutputWSService` started by Engine registers a Restlet application implemented by class `Root` which sets up URI routes and handlers for each type of query.

Each type of query is handled by a class inheriting from `QueryExecutorResourceBase` which in turn implements Restlet `ServerResource`. This base class loads necessary configuration and handles requests (methods annotated with Restlet `@Get` and `@Post` annotations) – parses request parameters (as described in User Manual), delegates the execution to the abstract `execute()` method implemented in child classes and handles returning of a proper response in case of an error.

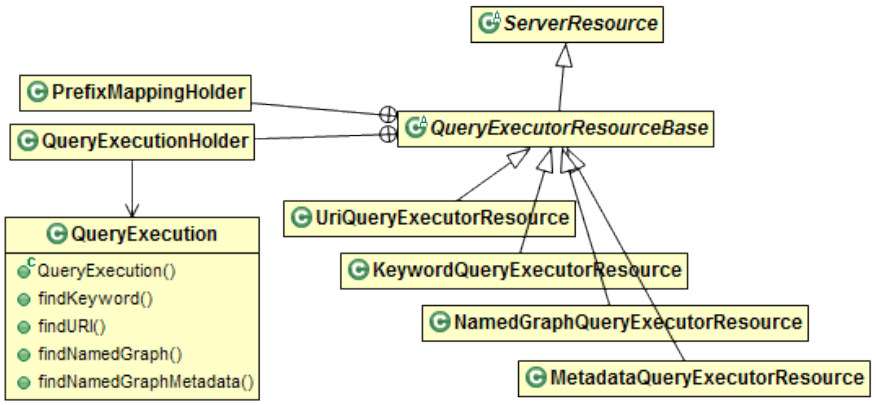

Figure 6.1: Diagram of selected Output Webservice classes

Classes implementing the actual execution of the query are `UriQueryExecutorResource`, `KeywordQueryExecutorResource`, `NamedGraphQueryExecutorResource` and `MetadataQueryExecutorResource`. They redefine the abstract `execute()` method where query-specific parameters are parsed and the execution is delegated to the Query Execution component. Instance of the `QueryExecution` class is shared between requests in order to utilize caching implemented in Query Execution. Finally, the result of the query is formatted and sent to the user.

6.5.3 Output Formatters

Query results returned from Query Execution are formatted using the format requested by the user. Formatting is done by classes implementing `QueryResultFormatter`.

The default formatter is `HTMLFormatter` which outputs results in HTML. `RDFXMLFormatter` and `TriGFormatter` inherit from `RDFFormatter` and output results in RDF/XML and TriG, respectively. `DebugFormatter` formats result for output to console and is not accessible for users.

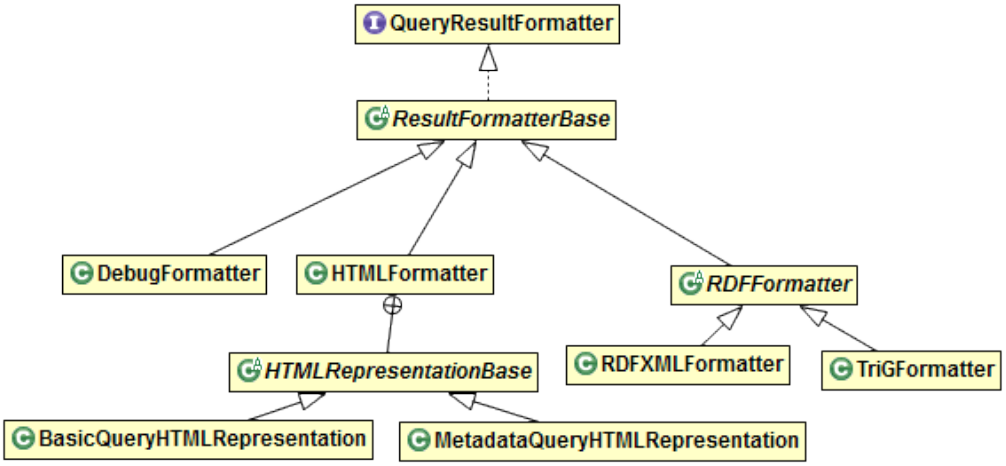

Figure 6.2: Diagram of output formatters hierarchy

6.5.4 Extending

In order to add a new type of query to Output Webservice, the following steps should be taken:

1. Implement a new `ServerResource` executing the query, preferably inheriting from `QueryExecutorResourceBase`. Typically, the actual query will be delegated to the Query Execution component – see Section 7.4.
2. Register the new `ServerResource` in method `Root#createInboundRoot()`.

7. Query Execution

7.1 Purpose

The purpose of Query Execution is to retrieve result for a query (asked through Input Webservice), resolve conflicts using the Conflict Resolution and return result.

Triples that Query Execution retrieves are:

1. Triples relevant for the query (e.g. containing the given URI).
2. Triples with metadata about named graphs containing triples from (1).
3. Triples containing human-readable labels for URI resources occuring in triples from (1).

A special case is the metadata query which retrieves only named graph metadata.

Because the result of conflict resolution depends on the data it is given, Query Execution and Conflict Resolution are not independent but rather Query Execution extracts exactly the data that Conflict Resolution needs and calls it directly.

7.2 Interface

The public interface of the Query Execution component is represented by class `QueryExecution`.

This class exposes methods for executing all kinds of supported queries and returns result as an instance of `MetadataQueryResult` (wraps collection of provenance metadata triples and other metadata) or `BasicQueryResult` (wraps collection of `CRQuad`s returned from Conflict Resolution plus metadata). The query can be further parametrized by passing `QueryConstraintSpec` and `AggregationSpec` affecting the retrieved data and the conflict resolution process, respectively.

`QueryExecution` is thread-safe and its instance should be kept between requests in order to effectively utilize caching.

7.3 Implementation

The actual implementation is in classes inheriting from `QueryExecutorBase`, each implementing one type of query: `URIQueryExecutor`, `KeywordQueryExecutor`, `NamedGraphQueryExecutor` and `MetadataQueryExecutor`. These classes are called internally from the `QueryExecution` class. Implementing classes are in Java package `cz.cuni.mff.odcleanstore.queryexecution`.

For each query, the following steps are executed: input is validated, result quads retrieved from the database, metadata and labels are retrieved from the database and conflict resolution is applied to the result.

To improve performance, values that are used for each query but rarely changed are cached. Cached values are: default aggregation settings, prefix mappings and label properties.

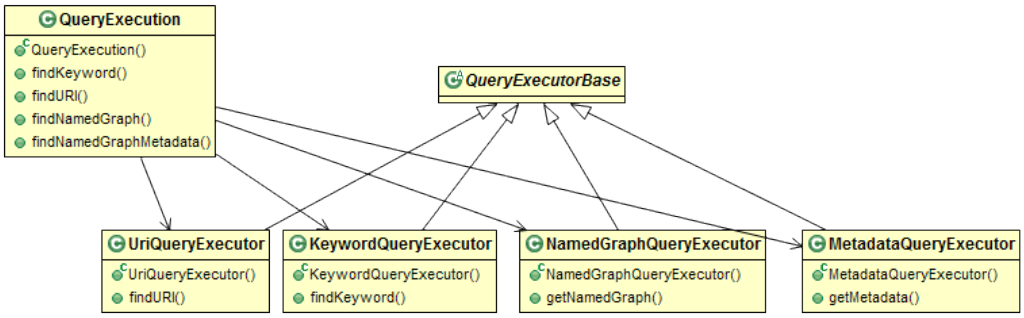

Figure 7.1: Diagram of main Query Execution classes

7.4 Extending

In order to implement a new type of query, the following steps should be taken:

1. Create a class implementating the new query, preferably inheriting from `QueryExecutorBase`.
2. Extend `QueryExecution` class with method for executing the query.
3. Extend Input Webservice to provide access to the new query for users.

7.5 Database

Query Execution retrieves RDF data from the clean database instance. Because Virtuoso doesn't fully support transactions with RDF data, the clean database may contain incomplete data partially inserted by Engine. In order to filter such data from the result, Query Execution ignores all named graphs whose URI starts with an agreed prefix[1] given by Engine to such named graphs.

In addition, Query Execution loads settings from the following tables in relational database (see Appendix C):

- `DB.ODCLEANSTORE.QE_LABEL_PROPERTIES`
- `DB.ODCLEANSTORE.CR_PROPERTIES`
- `DB.ODCLEANSTORE.CR_SETTINGS`
- and tables referenced from tables above

[1] `http://opendata.cz/infrastructure/odcleanstore/internal/hiddenGraph/`

8. Conflict Resolution

8.1 Purpose

Data stored in ODCleanStore may come from multiple sources and conflicting statements may emerge. For example, data about a city stored in ODCleanStore may state multiple different values for its population. The purpose of Conflict Resolution is to resolve such conflicts according to default or user-defined policies (e.g. show the latest, average or all values for the population). In addition, it computes an estimate of aggregate quality of each RDF triple in the resulting data and provides provenance tracking, i.e. provides URIs and metadata of named graphs from which the resulting value was selected or calculated from. Finally, Conflict Resolution can filter out old versions of data for which a newer version was inserted.

8.2 Interface

Interface of the Conflict Resolution component is represented by Java interface `ConflictResolver`. It contains a single operation:

```
Collection<CRQuad> resolveConflicts(Collection<Quad> quads)
    throws ConflictResolutionException;
```

Conflict Resolution accepts a collection of quads (RDF triple + named graph) and returns a collection of `CRQuad`s (quad + aggregate quality + source named graphs).

An instance of `ConflictResolver` can be obtained from factory class `ConflictResolverFactory`. Constructor of this class takes query-independent settings and its `createResolver()` methods takes query-dependent settings (named graph metadata, sameAs links, aggregation settings, preferred URIs) and returns a new instance of `ConflictResolver` for these settings. The signatures of the constructor and the `createResolver()` method are in Listing 8.1.

```
public ConflictResolverFactory(
    String resultGraphPrefix,
    ConflictResolutionConfig globalConfig,
    AggregationSpec defaultAggregationSpec)

public ConflictResolver createResolver(
     AggregationSpec aggregationSpec,
     NamedGraphMetadataMap metadata,
     Iterator<Triple> sameAsLinks,
     Set<String> preferredURIs)
```

Listing 8.1: `ConflictResolverFactory` interface

8.3 Implementation

The actual implementation of the `ConflictResolver` interface is in class `ConflictResolverImpl`. Its constructor requires two parameters – one of type `ConflictResolverSpec` contains all set-

tings for conflict resolution and one of type `ConflictResolutionConfig` is a container for global settings (configurable in the ODCleanStore configuration file). `ConflictResolverSpec` contains default and per-property aggregation methods to be used, metadata of relevant named graphs including Quality Assessment scores, additional `owl:sameAs` links to consider and other technical settings.

Implementation of the `resolveConflicts()` methods does the following:

1. `owl:sameAs` links are used to find resources representing the same entity. The implementation is in class `URIMappingImpl` which uses the DFU (Disjoint Find and Union) data structure with path compression to find weakly connected components of the `owl:sameAs` links graph.
2. URI resources in input quads are translated so that a single URI is used for every resource representing the same entity using mapping created in the previous step.
3. The resulting quads are sorted and grouped to clusters of (potentially) *conflicting quads*, i.e. those sharing the same subject and predicate. Implemented in `ResolveQuadCollection`.
4. Conflict Resolution iterates over groups of conflicting quads and applies the actual conflict resolution procedure.
   (a) The next group of conflicting quads is retrieved. All such quads have the same value in place of the subject and predicate.
   (b) If there are Identical triples which come from named graphs where one named graph is an update of the other named graphs, the old versions are removed (see Section 10.4.2 for definition of an update).
   (c) An aggregation method is chosen based on the predicate of quads in the current group and conflict resolution settings.
   (d) The aggregation method is applied to the current group of conflicting quads. The output is a collection of `CRQuad`s and it is added to the result.
5. The resulting `CRQuad`s are returned.

We call steps 1-3 *implicit conflict resolution* and it doesn't depend on the given aggregation settings. It prepares input data for step 4 so that result quads and aggregate quality can be computed independently on chosen resource URIs. Its time complexity is $\mathcal{O}(S \log S + N \log N)$ where S is the number of `owl:sameAs` links and N number of input quads. Step 4 is applied to sets of conflicting quads having the same subject and predicate so the context of aggregation is given mainly by quad objects.

8.3.1 Aggregation Methods

Conflict Resolution accepts an argument of type `ConflictResolverSpec` which specifies which agregation method should be used for which predicate, among other things. This is set either by the user as a parameter of Output Webservice or a default setting in ODCleanStore is used. The selected aggregation method determines how conflicts are resolved and aggregate quality of the result is computed in step 4d of the conflict resolution algorithm.

An aggregation method is represented by Java interface `AggregationMethod` with the following method:

```
Collection<CRQuad> aggregate(
    Collection<Quad> conflictingQuads,
    NamedGraphMetadataMap metadata);
```

Objects implementing `AggregationMethod` are created by `AggregationMethodFactory`. Classes implementing an aggregation method inherit from `AggregationMethodBase` and their hierarchy is depicted on Figure 8.1.

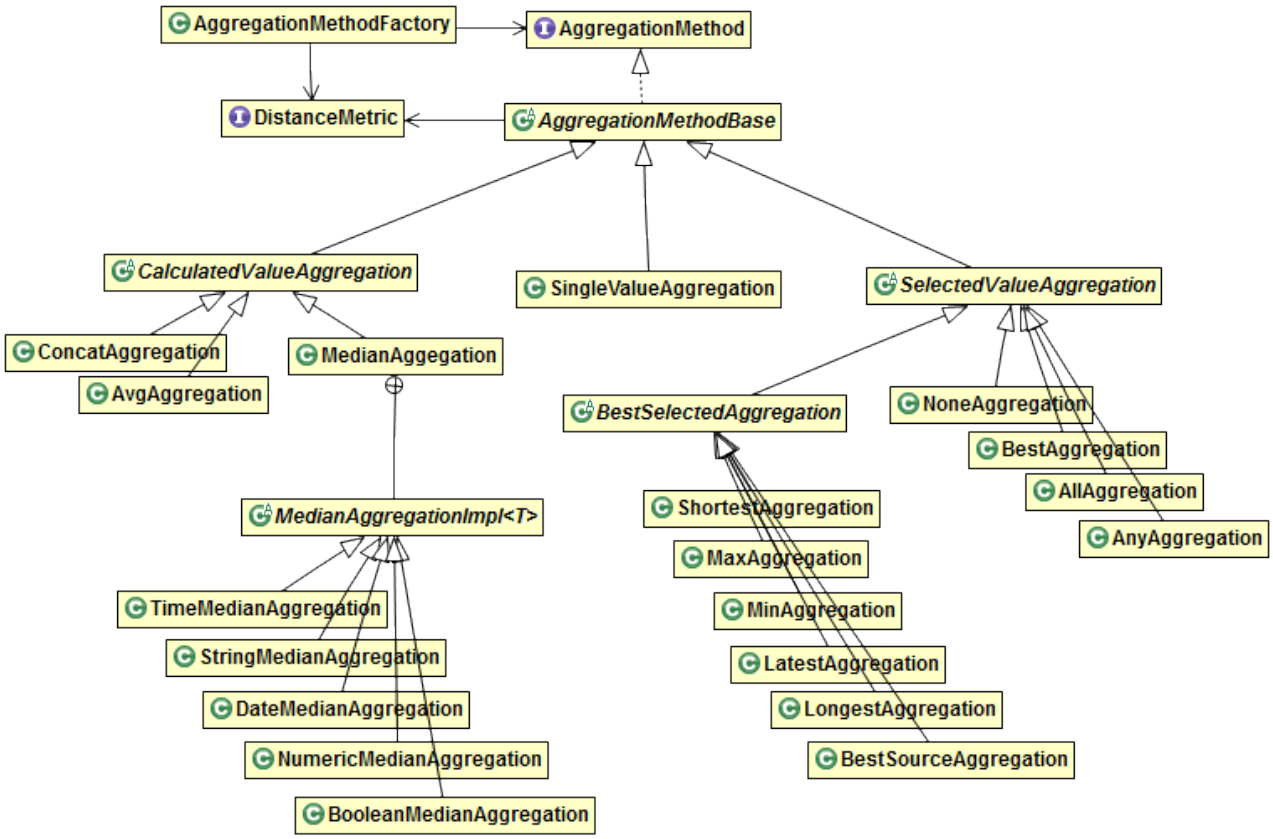

Figure 8.1: Implementation classes of aggregation methods

There are two basic types of aggregations. First type called *selected aggregations* selects one or more quads from input quads while the result of the second type called *calculated aggregation* returns values computed from all the input quads (e.g. average). The common functionality of these two types is in abstract classes `CalculatedValueAggregation` and `SelectedValueAggregation`.

A special type of selected aggregation is `BestSelectedAggregation` which selects a single best quad based on a metric given as an instance of `AggregationComparator` (Figure 8.2).

As an optimization, if there is only a single quad in a group of conflicting quads, a special optimized `SingleValueAggregation` aggregation can be used because all aggregation methods should return the same result in this case.

If the selected aggregation method cannot be applied to a value (e.g. average of a string literal), the behavior depends on the given aggregaiton error strategy - the value may be either discarded, or included in the result without aggregation applied.

More details about each aggregation method and their time complexity can be found in javadoc of the respective classes.

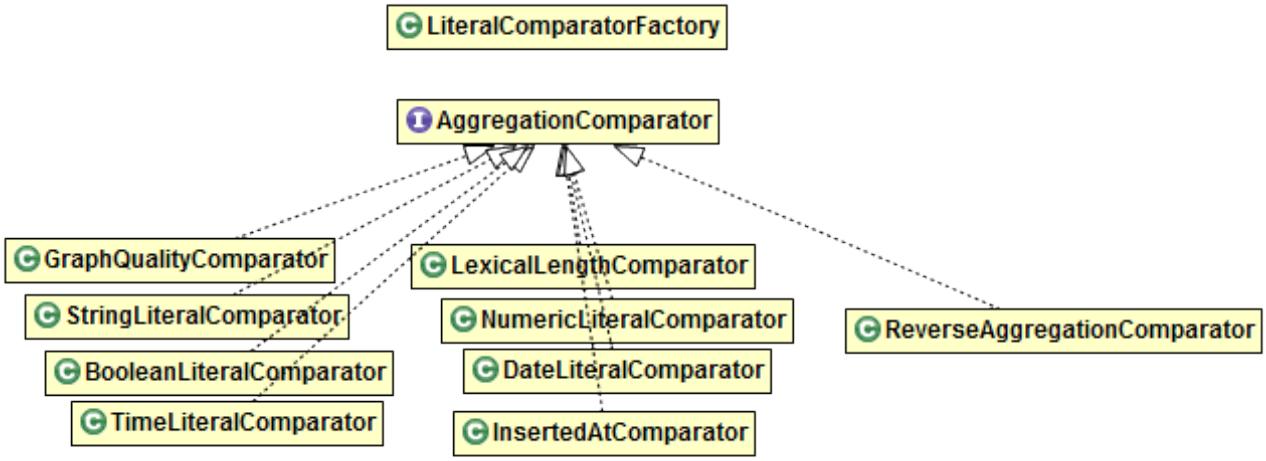

Figure 8.2: Comparators for `BestSelectedAggregation`

8.3.2 Quality and Provenance Calculation

The aggregation methods described in the previous section solve conflicts by calculating or selecting values in place of the object of a quad (e.g. choose the quad with the highest Quality Assessment score, the latest insertion date, maximum value in place of the object, or calculate the average of all values in place of the object). In addition, they add provenance and quality information.

The provenance information consists of a list of named graphs, let us denote them *source named graphs*, from which the result was selected or calculated from. For calculated aggregations, these are named graphs of all aggregated quads, for selected aggregations these are named graphs containing the quad(s) selected to the result.

The aggregate quality estimate is done for each result quad and is based on several factors based on real-world scenarios: quality scores of the source named graphs, number of graphs that agree on a value and the difference between a value and other (conflicting) values.

8.3.2.1 Notation

An aggregation method works on quads having the same subject and predicate and they may only differ in the object and the named graph. Let n be the number of aggregated quads (in a single group of conflicting quads). Let o_i be the value in place of object and g_i the named graph of i-th quad.

Let $s_{ng}(g)$ denote the Quality Assessment score of a named graph g and $s_p(g)$ the average score of publishers of the named graphs. Both these values are given in named graph metadata as input of conflict resolution (they are computed in advance by Quality Assessment and Quality Aggregator transformers, respectively). $s(g)$ denotes the total score of a named graph based on $s_{ng}(g)$ and $s_p(g)$ (see Section 8.3.2.3).

Let $agree(v)$ denote the set of named graphs that agree on the value v, i.e.

$$agree(v) = \{g_i \mid v_i = v\}.$$

Finally let $q(v)$ denote the aggregate quality of a result quad that has value v as its object.

8.3.2.2 Constraints on Quality Calculation

The algorithm calculating aggregate quality was designed so that several constraints hold:

- If $n = 1$, then $q(v_1) = s(g_1)$.
- If there is a named graph g asserting a non-conflicting value v, the quality (based just on the value v) should be at least $s(g)$.
- $q(v)$ is increasing with Quality Assessment scores of source named graphs of quads having v as object.
- $q(v)$ is decreasing with increasing difference from other object values in input quads, taking their Quality Assessment scores into consideration (higher Quality Assessment score means higher weight of the value).
- If multiple source named graphs agree on the same value v, then $q(v)$ is increased.
- If k sources with Quality Assessment scores equal to 1 (highest score) claim a value completely different from value v, then quality of v should be decreased approximately k times. If the sources have lower Quality Assessment scores, the decrease should be lower.

8.3.2.3 Quality Calculation

Aggregate quality $q(v)$ is calculated in the following steps:

1. A total Quality Assessment score $s(g)$ for each source named graph is determined. It is calculated as
$$s(g) = \alpha s_{ng}(g) + (1 - \alpha) s_p(g),$$
where $\alpha \in [0, 1]$ is a configurable parameter.
2. Quality based on source named graphs is calculated. For each quad object value v, we compute quality based on source named graphs, let us denote it $q_1(v)$.
$$q_1(v) = \begin{cases} \text{avg}\{s(g) \mid g \in \{g_1, \ldots, g_n\}\} & \text{calculated aggregations} \\ \max\{s(g) \mid g \in agree(v)\} & \text{selected aggregations} \end{cases}$$
3. Next, the differences between conflicting values are taken into consideration. We use a metric $d : U \times U \to [0, 1]$ satisfying $d(v, v) = 0$ for this purpose.
We use $d(x, y) = |(x - y)/\text{avg}(x, y)|$ in case of numeric literals, normalized Levenshtein distance in case of string literals, difference divided by a configurable maximum value in case of dates and $d(x, y) = 1$, where $x \neq y$, for URI resources, blank nodes and nodes of incompatible types.
Whether decreasing the aggregate quality based on conflicting values is the right solution depends on context, however. Predicate `rdf:type` often has multiple valid values which are not in fact conflicting. Therefore, whether to decrease score based on conflicts is set by the *multivalue* setting for the current predicate.

If multivalue is false and there are conflicting values different from v, the quality of v is reduced increasingly with the value of metric d and the score of the source of the conflicting value:

$$q_2(v) = \begin{cases} q_1(v) \cdot \left(1 - \frac{\sum_{i=1}^{n} s(g_i) d(v, v_i)}{\sum_{i=1}^{n} s(g_i)}\right) & \text{multivalue is false for the current property} \\ q_1(v) & \text{multivalue is true for the current property} \end{cases}$$

4. Aggregate quality should be increased if multiple values agree on the same value.

$$q_3(v) = q_2(v) + (1 - q_2(v)) \cdot \min\left(\frac{-q_1(v) + \sum_{g \in agree(v)} s(g)}{C}, 1\right),$$

where $C \in \mathbb{N}$ is a configurable constant.

5. Each aggregation method may adjust the general algorithm. In case of the CONCAT aggregation, computing q_2 and q_3 doesn't make sense and $q(v) = q_1(v)$ is returned. AVG and MEDIAN aggregations return $q(v) = q_2(v)$, and finally $q(v) = q_3(v)$ is returned for other aggregations.

8.3.2.4 Time Complexity

The time complexity of the aggregate quality computation for a fixed value v is $\mathcal{O}(n \cdot D)$, where D is the complexity of the distance metric evaluation. This gives us the overall complexity of $\mathcal{O}(n^2 \cdot D)$ for ALL and BEST aggregations, $\mathcal{O}(n \log n + n \cdot D$ for MEDIAN and $\mathcal{O}(n \cdot D)$ for others.

8.4 Time Complexity

Let N be the total number of input quads of the conflict resolution process, S number of `owl:sameAs` links, G number of named graphs for which we have metadata given. Let $CQ = \{cq_1, cq_2, \ldots, cq_K\}$ be the set of groups of conflicting quads and $n_i = |cq_i|$ be the size of i-th group of conflicting quads. D is the complexity of distance metric evaluation.

The complexity of implicit conflict resolution algorithm is $\mathcal{O}(N \log N + S \log S)$. Grouping the quads into clusters of conflicting quads requires sorting in $\mathcal{O}(N \log N)$, assuming comparison of two RDF nodes is in constant time. Filtering of old versions is implemented in $\mathcal{O}(n_i \log n_i \log G)$ (`NamedGraphMetadataMap` is implemented using a `TreeMap`, could be improved to $\mathcal{O}(n_i \log n_i)$ with a `HashMap`). Finally an aggregation method is applied with complexity given in Section 8.3.2.4.

To sum up, the total time complexity is:

- In case of ALL and BEST aggregation:

$$\mathcal{O}\left(N \log N + S \log S + \sum_{i=1}^{K} \left(n_i \log n_i \log G + n_i^2 D\right)\right)$$

- For aggregations other than ALL, BEST and MEDIAN:

$$\mathcal{O}\left(N \log N + S \log S + \sum_{i=1}^{K} (n_i \log n_i \log G + n_i D)\right)$$

In the worst case when $K = 1$ and $G = N$ ($G \leq N$ because at most N named graphs can be among input quads and Conflict Resolution gets in fact metadata only for these named graphs), this gives us

- In case of ALL and BEST aggregation:

$$\mathcal{O}\left(N^2 D + S \log S\right)$$

- For aggregations other than ALL, BEST and MEDIAN:

$$\mathcal{O}\left(N \log^2 N + ND + S \log S\right)$$

Distance metric is evaluated in linear time for strings (modified Levenshtein distance) and in constant time for other cases.

8.5 Extending

In order to add a new aggregation method, the following steps should be taken:

1. Implement the aggregation in a class implementing the `AggregationMethod` interface.
2. Create a constant representing this aggregation method in enum `EnumAggregationType`.
3. Extend `AggregationMethodFactory` to create an instance of the new aggregation.

If a new distance metric for a specific type of literal is to be added, this should be done in the `DistanceMetricImpl` class.

9. Transformers – Introduction

In this section, by a transformer we mean a Java class implementing the `Transformer` interface shown in Listing 9.1 (and other related classes used for implementation).

```
package cz.cuni.mff.odcleanstore.transformer;
public interface Transformer {
  void transformGraph(TransformedGraph inputGraph, TransformationContext context)
      throws TransformerException;

  void shutdown() throws TransformerException;
}
```

Listing 9.1: Transformer interface

The purpose of a transformer is to somehow process data. The data are not passed in memory, but rather stored in the (dirty) database instance and only the URI of the named graph to be processed and connection credentials for accessing the database are given to the transformer. This should minimize the need of complicated interfaces for data passing, make it easier to work with large data, let the transformer choose its own method of accessing the database and give it the full power of SPARQL/SPARUL (as implemented in Virtuoso).

The actual data processing should be implemented in the `transformGraph()` method. All required information is passed in its arguments, one with information about the processed graph and one about the environment – see Listings 9.2 and 9.3.

The `shutdown` method is called when Engine shuts down and can be used e.g. to release acquired resources.

```
package cz.cuni.mff.odcleanstore.transformer;
import java.util.Collection;

public interface TransformedGraph {
  String getGraphName();
  String getGraphId();
  String getMetadataGraphName();
  String getProvenanceMetadataGraphName();
  Collection<String> getAttachedGraphNames();
  void addAttachedGraph(String attachedGraphName) throws TransformedGraphException;
  void deleteGraph() throws TransformedGraphException;
  boolean isDeleted();
}
```

Listing 9.2: TransformedGraph interface

```
package cz.cuni.mff.odcleanstore.transformer;
import java.io.File;
import cz.cuni.mff.odcleanstore.connection.JDBCConnectionCredentials;

public interface TransformationContext {
  JDBCConnectionCredentials getDirtyDatabaseCredentials();
```

```
    JDBCConnectionCredentials getCleanDatabaseCredentials();
    String getTransformerConfiguration();
    File getTransformerDirectory();
    EnumTransformationType getTransformationType(); /* NEW or EXISTING */
}
```

Listing 9.3: TransformationContext interface

9.1 Transformer Instance Configuration

Each instance of a transformer in a pipeline may have its own configuration (for explanation of the difference between *transformer* and *transformer instance*, see Section 2.1.1). From the point of view of a transformer, it is a plain string which can be obtained by calling the `getTransformerConfiguration()` method.

This configuration string can be edited in Administration Frontend. The transformer may use the value in any way it needs, e.g. it may contain XML configuration, the recommended practice is to use the Java `Properties` file format, however. This format is used by transformers included by default in ODCleanStore unless stated otherwise.

Instances of important transformers (Quality Assessment, Data Normalization, Linker) can be also configured by assigning rule groups to them in Administration Frontend.

9.2 Contract between Engine and Transformers

Although Virtuoso doesn't fully support transactions over RDF data, data processing in ODCleanStore is implemented so as to keep data consistent. In order to make it work, however, a contract between Engine and transformers must be satisfied.

The Engine ensures that:

- When a transformer is applied to a transformed graph, no other transformer (in the same nor different pipeline) is applied to it. In other words, the transformed graph is not changed externally while a transformer is working on it.
- If the transformer throws an exception, all changes made in the pipeline on the graph in the dirty database are safely reverted (and the state of the graph is changed to WRONG). The graph may be processed again later. If the transformer was run on a graph already in the clean database, the version in the clean database in intact.
- Transformers may use the directory given by the `getTransformerDirectory()` method for they own purposes, e.g. storing temporary files, log files etc. It is a subdirectory named as the ID of the executed transformer instance inside the "working directory" of the transformer (configurable in Administration Frontend).
  Data specific for one transformer instance may be stored in this directory. Data shared by all instances of the same transformer may be stored in the parent directory of that returned by `getTransformerDirectory()`. Engine ensures that this parent directory will be the same for all instances of the same transformer (unless working directory is changed in Administration Frontend, of course).

In return, transformers should satisfy:

- Transformers may add/update/delete data in the `payload` graph, metadata graph, `provenance` graph or attached graphs. It may also add data to new graphs, but the transformer must
  1. register the graph by calling `addAttachedGraph()` *before* it writes any data to it,
  2. make sure that the name of the new graph is unique in the database (transformer may use the `getGraphId()` method to create names unique for each named graph).

  Transformers shouldn't modify contents of the dirty database in any other way.
- Transformers may access the clean database, but should use it only for reading. Because other transformer in the pipeline may fail, the changes executed by the current transformer in the dirty database may be discarded but changes in the clean database would be kept which may cause inconsistencies. The same applies should the transformer execute any other persistent actions.
- Transformer should only use the directory given by `getTransformerDirectory()` or its parent directory for accessing the filesystem.

9.3 Custom Transformers

The administrator may extend data-processing capabilities of ODCleanStore by adding new custom transformers. How to do so is described in Administrator's & Installation Manual.

From the technical point of view, a transformer implementation must implement the `Transformer` interface. This interface and other necessary classes are included in maven artifact `odcs-core`, so that only this artifact need to be referenced.

Note that custom transformers should satisfy conditions listed in Section 9.2.

10. Transformers Included in ODCleanStore

10.1 Quality Assessor & Quality Aggregator

10.1.1 Purpose

Quality Assessor: Data processed by ODCleanStore come in a raw form that may be inconsistent with the expected format. The purpose of the Quality Assessor is to provide a way to identify patterns in data that are responsible for those inconsistencies. This is achieved by user-provided or generated rules that specify RDF data patterns and a degree of inconsistency by use of SPARQL Select Where Clause and a coefficient that reduces the overall quality of the processed data chunk (named graph) and a single floating point coefficient.

Quality Aggregator: For Conflict Resolution purposes it is important to be able to associate a data source with quality of data it provides. Quality Assessor computes quality of individual data chunks, these have to be aggregated into a quality indicator for their sources.

10.1.2 Interface

While the `QualityAggregator` is a trivial extension of `Transformer` the `QualityAssessor` provides additional functionality – debugging:

```
package cz.cuni.mff.odcleanstore.qualityassessment;

public interface QualityAssessor extends Transformer {
  public static interface GraphScoreWithTrace extends Serializable {
    public String getGraphName();
    public void setGraphName(String graphName);
    public Double getScore();
    public List<QualityAssessmentRule> getTrace();
  }

  public List<GraphScoreWithTrace> debugRules(HashMap<String, String> graphs,
    TransformationContext context,
    TableVersion tableVersion) throws TransformerException;
}
```

Listing 10.1: Quality Assessor interface

The method `debugRules` takes into account that any input graph may need to be loaded to the database under a different name so that no collisions occur during debugging. Therefore input graphs correspond to pairs $\langle \text{originalName}, \text{actualName} \rangle$ and are passed in `HashMap`. The graph is expected to already exist in the database when this method is invoked. The `context` parameter provides environment similar to the one during ordinary transformation of a graph. `tableVersion` specifies whether committed or uncommitted versions of rule groups are to be

used. This approach allows the author of the debugged rule groups can test her latest revisions of rules while she is not forced to save the changes and overwrite their previous form. The output is a list of structures containing quality and all rules that affected it for each graph on the input.

Generating Rules from Ontology

Another publicly accessible part of the interface to the quality assessment is the generation of rules from ontologies.

```
package cz.cuni.mff.odcleanstore.qualityassessment.rules;

public class QualityAssessmentRulesModel {
  public Collection<QualityAssessmentRule> compileOntologyToRules(String ontologyGraphURI)
    throws QualityAssessmentException
}
```

Listing 10.2: Quality Assessment Rule Generation

The method `compileOntologyToRules()` expects a named graph `ontologyGraphURI` to be in the database. The statements of the graph are considered a definition of an ontology and are processed resource by resource and the following types of resources are handled:

- Functional property implemented as `FunctionalProperty`
- Inverse functional property implemented as `InverseFunctionalProperty`
- Enumerated property implemented as `ConceptScheme`

10.1.3 Implementation

Quality Assessment Rules

```
public QualityAssessmentRule (Integer id, Integer groupId, String filter, Double coefficient,
  String label, String description)
```

Listing 10.3: Quality Assessment rule constructor

The data filters are described with `SPARQL Where Clause`[1] followed by optional `Group by Clause`[2] and `Having Clause`[3]. The coefficient of quality is a number $c \in [0, 1]$.

Assessment & Aggregation

The implicit Quality Assessor Implementation is bound to concrete rule groups at instantiation. As any other transformer the Quality Assessor and Aggregator are provided with an input named graph, a metadata graph and a list of groups of rules to be applied to the input. The `transformGraph` method then does the following:

[1] http://www.w3.org/TR/rdf-sparql-query/#rWhereClause
[2] http://www.w3.org/TR/sparql11-query/#rGroupClause
[3] http://www.w3.org/TR/sparql11-query/#rHavingClause

Quality Assessor implemented by class `QualityAssessorImpl`

1. Assume the graph quality has the maximal value
2. Load committed versions of appropriate rules from database
3. For each rule determine whether the pattern is present in the input graph by means of `SPARQL SELECT COUNT(*) WHERE ...` and decrease the graph quality accordingly and log successful application of the rule
4. Store overall quality and all logged applications of rules to the metadata graph

The reason why `SELECT` is used in step 3 instead of `ASK` is that underlaying Virtuoso supported `GroupClause` and `HavingClause` only in `SELECT` despite it being added to `SPARQL 1.1` to a production rule expanding a shared nonterminal.

Debugging invokes the same implementation but does not store the resulting score and log of rule applications into the database. This information is returned in a structured form of `GraphScoreWithTrace` object (declared in listing 10.1) instead.

Quality Aggregator implemented by class `QualityAggregatorImpl`

The `transformGraph` method collects quality of all the graphs stored in the clean database and the currently processed graph and calculates the average value which is then assigned to the source. This operation is idempotent and thus robust to manual transformation of the database content because it is always corrected after update of a graph published by the given publisher.

Rule Generation implemented in `QualityAssessmentRulesModel.compileOntologyToRules()`

After the ontology is stored in the database in a separate named graph, its graph name can be passed to the `compileOntologyToRules()` method. All resources specified in the ontology are processed based on their `rdf:type` property.

Type of property p	constraint checked by generated rule
`owl:FunctionalProperty`	$[x, y_1], [x, y_2] \in p \rightarrow y_1 = y_2$
`owl:InverseFunctionalProperty`	$[x_1, y], [x_2, y] \in p \rightarrow x_1 = x_2$
`skos:ConceptScheme`	$[x, y] \in p \rightarrow y \in p.hasTopConcept$

10.2 Data Normalization

10.2.1 Purpose

The input data may come in different formats and flavours and it may show easier to normalize it before further transformations instead of adapting the rest of the process for all the forms the data may come in.

10.2.2 Interface

```
package cz.cuni.mff.odcleanstore.datanormalization;
```

```
public interface DataNormalizer extends Transformer {
  public interface TripleModification extends Serializable {
    public String getSubject();
    public String getPredicate();
    public String getObject();
  }

  public interface RuleModification extends Serializable {
    public void addInsertion(String s, String p, String o);
    public void addDeletion(String s, String p, String o);
    public Collection<TripleModification> getInsertions();
    public Collection<TripleModification> getDeletions();
  }

  public interface GraphModification extends Serializable {
    public void addInsertion (DataNormalizationRule rule, String s, String p, String o);
    public void addDeletion(DataNormalizationRule rule, String s, String p, String o);
    public Iterator<DataNormalizationRule> getRuleIterator();
    public RuleModification getModificationsByRule(DataNormalizationRule rule);
    public String getGraphName();
    public void setGraphName(String graphName);
  }

  List<GraphModification> debugRules (HashMap<String, String> graphs,
    TransformationContext context,
    TableVersion tableVersion) throws TransformerException;
}
```

Listing 10.4: Data Normalizer interface

The method `debugRules` takes into account that any input graph may need to be loaded to the database under a different name so that no collisions occur during debugging. Therefore input graphs correspond to pairs $\langle \text{originalName}, \text{actualName} \rangle$ and are passed in `HashMap`. The `context` parameter provides environment similar to the one during ordinary transformation of a graph. `tableVersion` specifies whether committed or uncommitted versions of rule groups are to be used. The output is list of `GraphModification` structures describing how each of the input graphs changed.

```
package cz.cuni.mff.odcleanstore.datanormalization.rules;

public class DataNormalizationRulesModel {
  public Collection<DataNormalizationRule> compileOntologyToRules(String ontologyGraphURI)
    throws DataNormalizationException;
}
```

Listing 10.5: Data Normalization Rule Generation

`compileOntologyToRules()` expects a named graph of a name `ontologyGraphURI` to be present in the database. Its contents are interpreted as definition of an ontology and examined as such.

10.2.3 Implementation

Data Normalization Rules

There exist three types of data normalization rules:

- `INSERT`,
- `DELETE`,
- `MODIFY`.

They all closely follow semantics of `SPARQL` update queries[4]. Due to incomplete support for `SPARQL 1.1`, specificly missing `BIND(`*expression* `AS` *var*`)`, it was necessary to allow use of subqueries for data manipulation and transformation by introduction of `$$graph$$` macro used intead of `iri` in `GraphRef` which is replaced with actual graph name during the transformation.

Data Normalizer

1. Load committed versions of appropriate rules
2. Invoke `SPARQL INSERT/DELETE/MODIFY` on the input graph for all components of all selected rules.

This process is further extended with modification detection for debugging purposes. The underlaying software does not provide an easy way to determine triples affected by application of a rule, thus it is necessery to compare original state of the graph with the outcome of the operation (`INSERT`/`DELETE`/`MODIFY`) after each step. This considerably slows down the whole process but provides precise information about what happend during the graph transformation. The modification can be represented in a sense of standard `diff` between two plain text resources. The method `debugRules` of `DataNormalizer` returns a `GraphModification` structure containing differences grouped by rules that introduced them. This extension does not affect the ordinary transformation procedure.

Rule generation

The method `DataNormalizationRulesModel.compileOntologyToRules()` searches for resources with `rdfs:range` statements in the ontology graph in database and creates adequate rules. The following rules can be generated:

- `DataNormalizationBooleanRule`
  converts `0, 1, y, n, yes, no, true, false` (case insensitively) to `0, 1`
- `DataNormalizationDateRule`
  attempts to interpret the converted value as date string (2012, 2012-01 ...)
- `DataNormalizationStringRule`
  converts to string so any subsequent calls to substring or regexp functions do not fail
- `DataNormalizationNonNegativeIntegerRule`
  converts to number ≥ 0 dropping fractional part if any ($1.9 \mapsto 1$, $-1 \mapsto 0$, "yes" $\mapsto 0$)
- `DataNormalizationNonPositiveIntegerRule`
  converts to number ≤ 0 dropping fractional part if any ($1 \mapsto 0$, $-1.9 \mapsto -1$, "yes" $\mapsto 0$)

[4] http://www.w3.org/Submission/SPARQL-Update/#rUpdate

- `DataNormalizationIntegerRule`
  drops fractional part of a number interpretation ($1.9 \mapsto 1$, $-1.9 \mapsto -1$, "yes" $\mapsto 0$)
- `DataNormalizationNumberRule`
  converts to number ("yes" $\mapsto 0$)

Number conversion uses Virtuoso built-in functions (`__min_notnull`, `__max_notnull`, `sign`, `abs`, `floor` ...) and thus heavily relies on their presence and implementation.

These rules cover the basic and often used `XSD` datatypes with a reasonable conversion for general purposes. More specific transformations could be defined but are not suited for implicit rule generation as they may not effectively cover usual cases or could even produce bad results in different circumstances. For example, date conversions would often need additional information about the source of unnormalized date substrings to recongnize the format. The pre-transformation value is often ambiguous which could lead to incorrect interpretation (confusion of month for day of the month).

10.3 Linker

10.3.1 Purpose

The main purpose of this transformer is to interlink URIs which represent the same real-world entity by generating `owl:sameAs` links. It can be also used for creating other types of links between differently related URIs. Silk framework[5] is used as the linking engine.

10.3.2 Interface

While the `Linker` is a trivial extension of `Transformer` the `Linker` provides additional functionality – debugging:

```
package cz.cuni.mff.odcleanstore.linker;

public interface Linker extends Transformer {
  public List<DebugResult> debugRules(String input, TransformationContext context,
    TableVersion tableVersion) throws TransformerException;
  public List<DebugResult> debugRules(File inputFile, TransformationContext context,
    TableVersion tableVersion, SerializationLanguage language) throws TransformerException;
}
```

Listing 10.6: Linker interface

The `debugRules` method has two variants. First one gets the input RDF data in a string and tries to guess the format of the data by itself. Supported formats are RDF/XML and N3 (including its subsets - N-Triples and Turtle). The second one gets the data in a file, its format is specified by the `language` parameter. It does not open the file, just passes it to Silk. The `context` parameter provides environment similar to the one during ordinary transformation of a graph. `tableVersion` specifies whether committed or uncommitted versions of rule groups

[5] http://wifo5-03.informatik.uni-mannheim.de/bizer/silk/

are to be used. The list of IDs of the linkage rules groups to be debugged is passed to the `Linker` in its constructor. Output is a list of structures, one for each rule, containing rule label and a list of links generated by it. Link is represented by a `LinkedPair` class, which contains both interlinked URIs, corresponding labels (if found) and confidence of the link (real number).

Another publicly accessible methods can be found in the `ConfigBuilder` class. It contains static methods for working with XML (namely Silk-LSL[6]) configuration file for Silk. The following two methods are used for importing/exporting linkage rules from/to Silk-LSL:

```
package cz.cuni.mff.odcleanstore.linker.impl;

public class ConfigBuilder {
  public static SilkRule parseRule(InputStream input)
    throws javax.xml.transform.TransformerException, ParserConfigurationException,
    SAXException, IOException { ... }
  public static String createRuleXML(SilkRule rule, List<RDFprefix> prefixes)
    throws ParserConfigurationException, DOMException, InvalidLinkageRuleException,
    SAXException, IOException, javax.xml.transform.TransformerException { ... }
}
```

Listing 10.7: ConfigBuilder interface

10.3.3 Implementation

The actual implementation is in three classes. `LinkerImpl` implements the `Linker` interface, `ConfigBuilder` creates XML configuration file in Silk-LSL and `LinkerDao` accesses the database.

10.3.3.1 LinkerImpl

Usage of the linker starts by calling the `LinkerImpl` constructor, which takes a list of linkage rule groups IDs and a boolean flag `isFirstInPipeline` as its arguments. Only when the flag is set to true, linker deletes existing links before transforming existing graph (allready present in clean database). This prevents multiple linkers in one pipeline from deleting their own links. Next the `transformGraph` method is called, which does following steps:

1. Load linkage rules from the database.
2. Load all registered RDF prefixes from the database.
3. For each loaded rule:
   (a) Create configuration file in Silk-LSL.
   (b) Call Silk with this configuration (`Silk.executeFile` method).

At first the implementation was different. One configuration file was created for the whole set of rules and passed to Silk only once. Then we discovered, Silk was processing the rules one-by-one anyway, therefore we changed the implementation to improve logging possibilities without really affecting the performance.

[6] http://www.assembla.com/wiki/show/silk/Link_Specification_Language

By default, links are created between transformed graph and graphs in the clean database. If you want to interlink transformed graph with itself, you can do it by setting `object_identification.link_within_graph` to true in the global configuration or adding `linkWithinGraph=true` to the particular transformer configuration (see Section 9.1). When activated, linker creates two linkage rules in Silk-LSL for each rule loaded from database. One rule is for linking transformed graph with clean database, another for linking with itself.

It can be useful to use attached graphs (RDF data generated by preceeding transformers in the pipeline) in linkage rules, e.g. links generated by another linker. This feature can be activated by setting `object_identification.link_attached_graphs` to true in the global configuration or adding `linkAttachedGraphs=true` to the particular transformer configuration. When activated, linker creates a graph group[7] from tranformed graph and its attached graphs and passes it to Silk for linking.

When transforming existing graph in the clean database, it is first copied to the dirty database, transformed, then it replaces the original graph. This allows links between old and new version of the same graph to emerge. It is not possible to exclude a graph from linking in Silk-LSL. To avoid this inconsistency, graph group containing all graphs in the clean database excluding transformed graph and its attached graph is created and passed to Silk for linking.

10.3.3.2 ConfigBuilder

This class is responsible for creating a XML configuration file in Silk-LSL. Standard Java API for XML is used in this class (mostly DOM). An example of created configuration file follows. Description of individual elements can be found in Silk-LSL specification[8].

```
<Silk>
  <Prefixes>
    <Prefix id="adms" namespace="http://www.w3.org/ns/adms#"/>
    <Prefix id="dcterms" namespace="http://purl.org/dc/terms/"/>
    <Prefix id="gr" namespace="http://purl.org/goodrelations/v1#"/>
    <Prefix id="rdf" namespace="http://www.w3.org/1999/02/22-rdf-syntax-ns#"/>
    <Prefix id="rdfs" namespace="http://www.w3.org/2000/01/rdf-schema#"/>
    <Prefix id="skos" namespace="http://www.w3.org/2004/02/skos/core#"/>
  </Prefixes>
  <DataSources>
    <DataSource id="sourceA" type="sparqlEndpoint">
      <Param name="endpointURI" value="http://example.com"/>
    </DataSource>
    <DataSource id="sourceB" type="sparqlEndpoint">
      <Param name="endpointURI" value="http://example.com"/>
    </DataSource>
  </DataSources>
  <Interlinks>
    <Interlink id="ic-based">
      <LinkType>owl:sameAs</LinkType>
      <SourceDataset dataSource="sourceA" var="a">
        <RestrictTo>?a rdf:type gr:BusinessEntity</RestrictTo>
```

[7] http://docs.openlinksw.com/virtuoso/rdfgraphsecurity.html
[8] http://www.assembla.com/wiki/show/silk/Link_Specification_Language

```
    </SourceDataset>
    <TargetDataset dataSource="sourceB" var="b">
      <RestrictTo>?b rdf:type gr:BusinessEntity</RestrictTo>
    </TargetDataset>
    <LinkageRule>
      <Aggregate type="min">
        <Compare metric="equality" required="true">
          <Input path="?a/adms:identifier/skos:notation"/>
          <Input path="?b/adms:identifier/skos:notation"/>
        </Compare>
        <Compare metric="equality" required="true">
          <Input path="?a/adms:identifier/dcterms:creator"/>
          <Input path="?b/adms:identifier/dcterms:creator"/>
        </Compare>
      </Aggregate>
    </LinkageRule>
  <Filter threshold="0.9"/>
  <Outputs>
    <Output maxConfidence="0.8" type="file" >
      <Param name="file" value="test_be_sameAs_verify_links.ttl"/>
      <Param name="format" value="ntriples"/>
    </Output>
  </Outputs>
 </Interlink>
</Interlinks>
</Silk>
```

Listing 10.8: Configuration file in Silk-LSL

10.3.3.3 LinkerDao

Linker accesses database using this class. It utilizes the `VirtuosoConnectionWrapper` to work with the database. SPARQL queries are used to work with RDF data, SQL queries for relation data, namely linkage rules, stored in following tables (see Appendix C):

- `DB.ODCLEANSTORE.OI_RULES`
- `DB.ODCLEANSTORE.OI_RULES_UNCOMMITED`
- `DB.ODCLEANSTORE.OI_OUTPUTS`
- `DB.ODCLEANSTORE.OI_OUTPUT_TYPES`
- `DB.ODCLEANSTORE.OI_FILE_FORMATS`

10.4 Other Transformers

10.4.1 Blank Node Remover

Blank Node Remover is a simple transformer for replacing of blank nodes in the `payload` graph with unique URI resources. It is implemened in class `ODCSBNodeToResourceTransformer`.

The generated URIs have format *<prefix><random UUID>-<Virtuoso nodeID>*. The transformer guarantees that occurrences of the same blank node within the transformed graph

will be assigned the same URI, however, occurrences of the blank node in other graphs will be assigned a different URI when they are processed by the transformer.

Value of `input_ws.named_graphs_prefix` configuration option concatenated with "genResource/" is used as the default value of the <*prefix*> part. It can be overriden by `uriPrefix` option in transformer instance configuration.

10.4.1.1 Configuration

Possible configuration options for an instance of this transformer:

`uriPrefix`
: Sets the prefix of URIs generated in place of blank nodes.

10.4.2 Latest Update Marker

Latest Update Marker is an internal transformer for marking the latest version of a named graph with `odcs:isLatestUpdate` property. The marker may be used when accessing the clean database directly through the SPARQL endpoint. Latest Update Marker is implemented in class `ODCSLatestUpdateMarkerTransformer`.

A named graph A is considered an update of named graph B if:

1. Named graphs A and B have the same update tag, or both have an unspecified (`null`) update tag.
2. Named graphs A and B were inserted by the same (SCR) user.
3. Named graphs A and B have the same set of sources in metadata.

The transformed graph will be labeled as the latest version by adding the triple <*payload-graph*>-`odcs:isLatestUpdate`-"1". If it updates another graph in the clean database, the other graph will be unmarked as being the latest version.

This transformer is automatically added by Engine to the end of every pipeline. This is neccessary because the transformer may modify the clean database and therefore should ensure that the pipeline won't fail afterwards.

10.4.3 Property Filter

Property Filter is an internal transformer for filtering of properties used internally by ODCleanStore from input data. It is implemented in class `ODCSPropertyFilterTransformer`.

Property Filter simply removes all triples that have any of the filtered URIs in place of the predicate (see Input Processing in User Manual) from the `payload` and `provenance` named graphs.

This transformer is automatically added by Engine as the first transformer of every pipeline.

11. Administration Frontend

11.1 Codebase structure

The code of Administration Frontend is divided into multiple packages which all share a common name prefix - `cz.cuni.mff.odcleanstore.webfrontend`. In this section you'll find a brief description of each of them and of classes they contain. For more in-dept information please refer to the javadoc documentation.

11.1.1 behaviours

This package is intended to contain custom Wicket behaviours. Currently there's only a single class, the `ConfirmationBoxRenderer`, which serves to add javascript-based confirmation dialog boxes to delete buttons.

11.1.2 bo

The `bo` package contains all business objects used in the web frontend application, divided into subpackages. Every business entity represents a single relational table and is formed by a single Java class. The classes are simple Java beans where bean properties match columns of the represented table and can aditionaly contain other methods to simplify using them.

The hierarchy of the most important BO classes is depicted on on Figure 11.1.

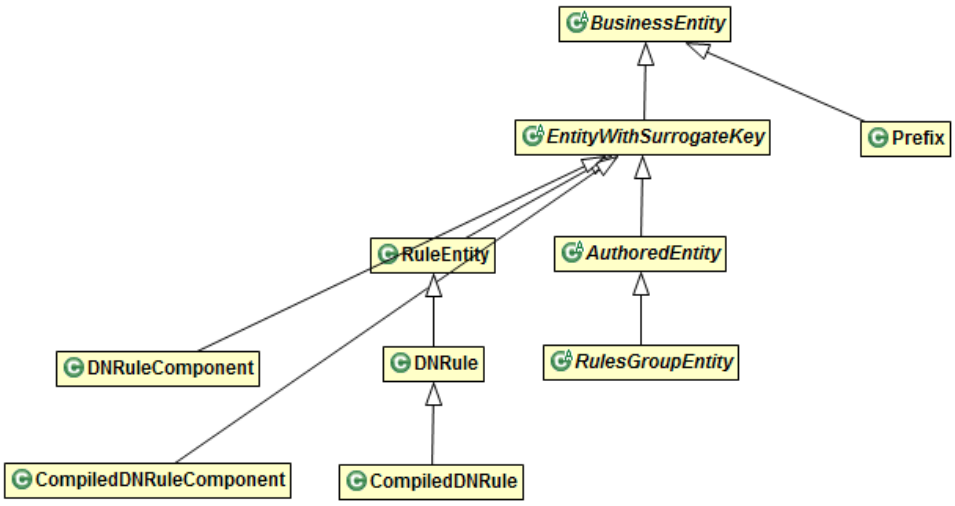

Figure 11.1: Selected BO classes used in Administration Frontend

Description of the most important BO classes follows:

`BusinessEntity`
: The base class for all BO classes. It provides no functionality and serves just as a common abstract parent.

`EntityWithSurrogateKey`
: The base class for BO classes for all entities with surrogate primary keys. It only contains the `id` property.

`AuthoredEntity`
: This interface serves as an abstract parent of all entities that track their authors.

`Prefix`
: This BO represents an URI prefix. It maps to an internal Virtuoso relational table and it's structure is therefore fixed.

`DNRule and DNRuleComponent`
: These entities represent DN rules and rule components and are useful when working with rules in the raw form.

`CompiledDNRule and CompiledDNRuleComponent`
: These entities represent DN rules and rule components and are easier to use when working with rules compiled from template instances.

11.1.3 dao

This package (and it's subpackages) contains classes which form the DAO layer. See the related section for more information on that.

11.1.4 core

The `core` package contains `ODCSWebFrontendApplication` and `ODCSWebFrontentSession` classes extending the standard Wicket classes representing a web application and session, respectively, and the `ODCSWebFrontendApplication.properties` file. The properties file contains custom validation messages and declarations of other string literals. In addition, the package contains the following items:

`URLRouter`
: Handles URL routing - e.g. conversion from raw Wicket-like URL addresses to human readable ones (and vice versa).

`DaoLookupFactory`
: Encapsulates DB connections and serves DAO objects to page components (see Section 11.2).

11.1.5 core.models

This package contains custom implementations of the `IDataProvider` and `SortableDataProvider` interfaces, which adhere to our DAO layer implementation.

11.1.6 core.components

In the `core.components` package you'll find several custom components, such as labels, buttons, etc. The following list describes some of them. For a more comperhensive description, please see the javadoc documentation.

`UnobtrusivePagingNavigator`
: An implementation of the `PagingNavigator` component which is not visible if the list contains only a single page.

`TruncatedLabel`
: A label which only displays a few first characters of the given string.

`TimestampLabel`
: A label to display a properly formated timestamp.

`RedirectButton`
: A button to redirect to a different page, adjusted to suit the structure of the Administration Frontend application.

`LogInPanel`
: A panel which contains the log-in form.

`LimitedEditingForm`
: A form that can only be confirmed by authorized users.

`HelpWindow`
: A generic modal window to display help information.

11.1.7 pages

In this package there are Wicket components for individual pages. The structure of subpackages mirrors the Administration Frontend menu. These are mostly standard Wicket components (but for a few conventions we have applied - read more in Section 11.4.1). In this section, a brief description of some of the non-standard components will be given.

`FrontendPage`
: The parent component for almost all (see below) page components. Provides page layout and content common to all pages, as well as factory methods to create some basic child components.

`LogoutPage`
: This component does not extend `FrontendPage`, for it does not represent a proper page. It's purpose is to log out the current user and then redirect to home-page.

`UserPanel`
: This component forms the user-information panel - it shows the username and list of roles of the currently logged in user and provides means to log-in and log-out. It is included as a child component in the `FrontendPage`.

For more information on how to create a new page, see Section 11.4.1.

11.1.8 util

This package is intended for Administration Frontend specific utility classes, such as classes to ease working with arrays (`ArrayUtils`), sending email messages (`Mail`, `NewAccountMail`, `NewPasswordMail`) and generating and hashing passwords (`PasswordHandling`) - to name some of them.

Some methods in other parts of the Administration Frontend codebase accept code snippets (e.g. closures) as arguments. Because the current version of Java (1.6) does not directly support closures, we have created two classes - `CodeSnippet` and `EmptyCodeSnippet` - to be used instead. They too belong in the `util` package.

11.1.9 validators

This package contains custom Wicket form validators, such as `IRIValidator` - ensures that the given value is a valid IRI, `EnumValidator` - ensures that the given value is an element of the represented enumeration or `OldPasswordValidator` - ensures that the given new password matches the original one.

All our custom validators extend the `CustomValidator` class, which handles error message propagation so that validators can focus just on the validation process.

11.2 Database Access Layer

Administration Frontend has a layer for accessing the database based on Spring and its JDBC templates. We chose not to use Hibernate due to integration problems with Wicket and use custom implemenation of business and DAO objects.

Entities retrieved from database are represented by POJOs (Plain Old Java Objects). The code that actually retrieves them is in a DAO class, by convention having suffix `Dao` and inheriting from the base class `Dao`. DAO objects internally call methods of the Spring's `JdbcTemplate` and passes to it a class extending `CustomRowMapper` which implements creation of the POJO business object(s) from query results.

The DAO objects can be obtained from an instance of `DaoLookupFactory` (available e.g. as a protected member of `FrontendPage`). A DAO object can be obtained by calling a `getDao()` method which returns an existing DAO object or creates a new one if necessary. Signatures of `getDao()` methods are:

```
public <T extends Dao> T getDao(Class<T> daoClass)
public <T extends Dao> T getDao(Class<T> daoClass, boolean commitable)
```

We utilize generics in Java to obtain a specific type of a DAO class. In addition, there may be two versions of a DAO class – one for a read-only view of commited version of an entity and one for the working version visible only to the author (see Section 11.3.2). One can use the second version of the `getDao()` method and request either the read-only or commitable version.

Commitable and read-only DAOs are implemented using a custom `@CommitableDao` annotation. The read-only version should be annotated with `@CommitableDao` having the

commitable DAO class as its argument. The commitable version must inherit from the read-only DAO.

Hierarchy of DAO objects is depicted on Figure 11.2.

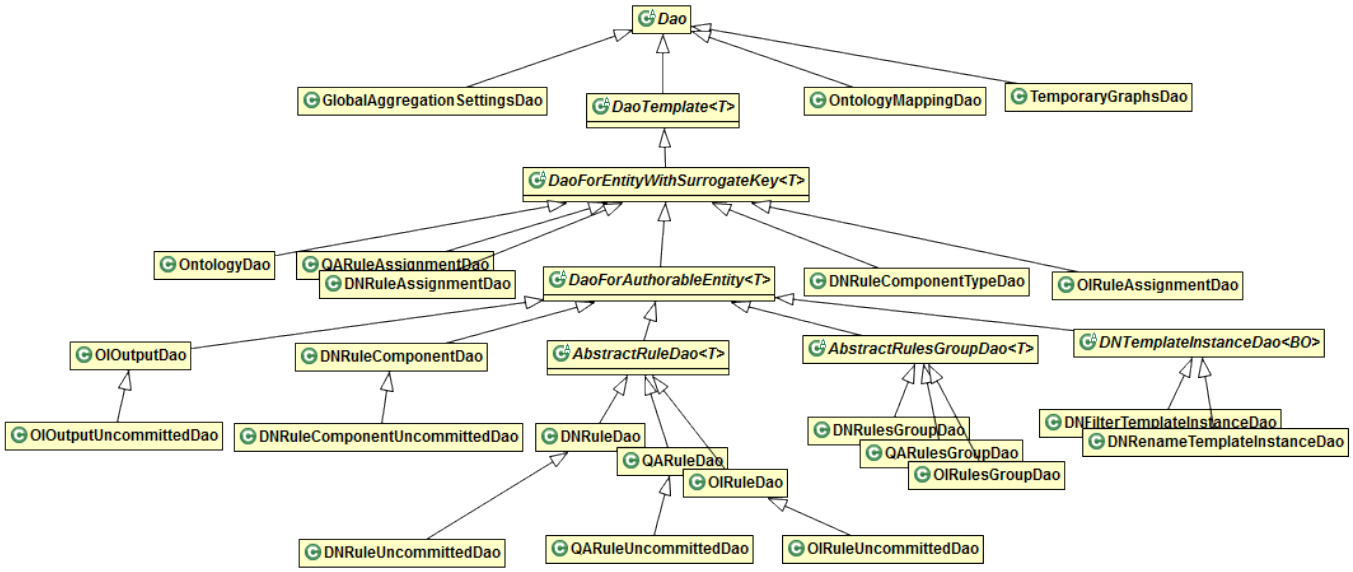

Figure 11.2: Selected DAO classes used in Administration Frontend

11.2.1 Important DAO Classes

Description of the most important DAO classes follows:

`Dao` This is the base class of all DAO classes. It keeps an instance of `JdbcTemplate` and provides access to it either directly or through utility methods `jdbcQuery()`, `jdbcQueryForInt()`, `jdbcQueryForList()`, etc. It can also execute code in a Spring transaction with `executeInTransaction()` and handles proper recognition of some exceptions thrown by Virtuoso JDBC driver.

`DaoTemplate`
This method provides convenience methods for loading of one or more entities from the database. Its `getTableName()`, `getRowMapper()`, `getSelectAndFromClause()`, `postLoadAllBy()` and `postLoadBy()` methods can be used to customize the loading. Other methods are declared as final.

`DaoForEntityWithSurrogateKey`
This DAO is used for working with entities with a primary key. The corresponding business objects must inherit from `EntityWithSurrogateKey`. It extends `DaoTemplate` with additional methods for loading, deleting and saving an entity by its primary key.

`DaoForAuthorableEntity`
This class is intended for entities that can be edited only by their author. It adds an abstract method `getAuthorId()`.

`AbstractRuleDao`
: Base class for (QA, DN, Linker) transformer rules. It provides methods for commiting of changes and disables any delete and update operations.

`XXXRuleDao`
: Concrete classes inheriting from AbstractRuleDao. It can be used for read-only access to rules (their commited version, respectively). `save()` and `update()` methods throw an exception. It is annotated with `@CommittableDao(XXXRuleUncommittedDao.class)` so that the commitable/editable version can be obtained.

  An instance of this DAO may be obtained by calling e.g.
  `daoLookupFactory.getDao(XXXRuleDao, false)`.

`XXXRuleUncommittedDao`
: These classes inherit from `XXXRuleDao` and provide the editable and commitable view on transformer rules. Changes may be commited in transaction by calling `commitChanges()`.

  An instance of this DAO may be obtained by calling e.g.
  `daoLookupFactory.getDao(XXXRuleDao, true)`.

`OntologyDao`
: This class extends `DaoForEntityWithSurrogateKey` and is used for working with ontologies. When storing an ontology, firstly its definition is stored to a RDF graph, which's name is derive from ontology label. After that the ontology is stored to the relational database. This order is chosen to avoid using transaction across RDF and relation data, which does not work. Ontology definition is stored to the relational database as well to keep its formatting and possible comments. Finally quality assessment and data normalization rules are generated from stored ontology (see Sections 10.1 and 10.2).

11.3 Authorization

There are two main scopes of authorization in ODCleanStore– authorization based on roles and authorization based on the authorship of an entity.

11.3.1 Roles

Authorization based on roles recognizes 5 roles: Administrator (ADM), Pipeline Creator (PIC), Ontology Creator (ONC), Data Producer (SCR) and Data Consumer (USR). Their detailed description is given in User Manual. Roles can be assigned to users in Administration Frontend and a user can have any number of roles.

We use means provided by Wicket to apply authorization by role. Pages and components can be marked with `@AuthorizeInstantiation` annotation with enumeration of roles that are required to access the page or component (at least one of the roles from the given list is required). The roles assigned to the currently logged-in user are kept in the session object `ODCSWebFrontendSession` which extends Wicket `AuthenticatedWebSession` for this purpose.

11.3.2 Authorship

Authorization based on authorship is necessary for entities that can be only edited by their author. Rule groups and rules, for example, can be only edited by the user who created them or by user having the role Administrator. To facilitate checking of whether the current user is authorized for entity editing, class `LimitedEditingPage` extending `FrontendPage` was introduced.

`LimitedEditingPage` requires two additinal arguments in its constructor: edited entity ID and a DAO class for the authorable entity (`DaoForAuthorableEntity`), which can retrieve author based on entity ID. It then checks whether the current user is authorized using a helper class `AuthorizationHelper` and makes this information accessible with protected methods.

Every page that needs information about whether the user is authorized for editation of can call protected methods `checkUnauthorizedInstantiation()` to prevent the user from displaying the page or `isEditable()` to detect whether the user is authorized for editation.

Because transformer rules and related settings must be commited before the changes are visible to users not authorized for editation (who have a read-only access to rules), there are two versions of rules in the database – one version is visible for the author and Administrators (these tables have suffix `_UNCOMMITED`) and one version visible for Engine and other users. The proper table version for the current user can be obtained by method `getVisibleTableVersion()`.

11.4 Extending

11.4.1 How to Add a New Page

First of all make sure that you are familiar with the Wicket framework, for the whole web application is based on it.

A standard Wicket `WebPage` component is used to create new pages. That means that you can use all the standard stuff that Wicket provides, such as adding standard child components (forms, tables, links, ...). Aditionaly, you should adhere to some specific conventions, which have been established to ease and fasten the development process and to make the web well structured and consistent. This section provides you with details on these conventions.

The POJO part of the page component should extend the `FrontendPage` class. The `FrontendPage` component provides page layout and content for parts of the page which are common throughout the whole application (such as logo and menu bar). That's why, when creating a new page, you only need to take care of the custom content. There are two components handled by the `FrontendPage` which need per-page parameters - the page title and the bread crumbs. The newly created page should supply these values via a call to the constructor of the parent's class. The `FrontendPage` also contains several helper factory methods which can be used to construct simple child components. Last but not least, the `FrontendPage` provides two shortcut methods - `getApp()` to obtain the application object and `getODCSSession()` to get the session object. All of these are protected scoped and can be used arbitrarily in new page components.

The POJO must provide a constructor. The constructor should be either parameter-less or

accept a single parameter – the id of entity to be described on the page. Inside the constructor you generaly need to call the parent's constructor (and suply the page title and crumbs values), obtain all needed DAO objects via the `daoLookupFactory` class attribute (see Section 11.2) and add all child components, ideally through calls to private methods, one for each component.

You will also want to update the HTML file of the `FrontendPage` component in order to add a link to the new page to the menu bar. Simply add a new standard Wicket link to the HTML list.

All parts of the web frontend application adhere to the following structure rules. Every section describes registered entities of a single type (such as transformers, pipelines, ontologies, user accounts). For every entity type (say XXX) there are four page components - `XXXListPage` (shows a list of all registered entities), `NewXXXPage` (provides a form to register a new entity), `XXXDetailPage` (shows an existing entity in details and provides a form to edit it; if there are subentities bound to the entity then shows a list of all of them).

As a rule, every page should contain a single help pop-up window for every entity the user can work with via that page. To create a help pop-up window for a new entity type (say XXX again), you only need to create an XXXHelpPanel component (you can copy an existing help panel component and update the contents in the related HTML page). Then use the `addHelpWindow` method of the `FrontendPage` class to add the pop-up window to the page.

11.4.2 How to Add a New Data Normalization Template

To add a new Data Normalization Template (named say XXX, such as Filter, Rename, Replace), follow these steps.

Create a new relational table in the Virtuoso database. The table should be named `DB.ODCLEANSTORE.DN_XXX_TEMPLATE_INSTANCES` and contain `id` as a surrogate primary key, `groupId` as a foreign key which points to the group of rules to which the template instance belongs, `rawRuleId` as a foreign key which points to the raw version of the represented rule and columns for attributes of the template.

Add a new Business Object to the `cz.cuni.mff.odcleanstore.webfrontend.bo.dn` package named `DNXXXTemplateInstance`. The class should adhere to standard BO conventions (see the relevant section) and relate to the table created in the previous step.

Add a compiler class to the same package called `DNXXXTemplateInstanceCompiler`. It must extend the `DNTemplateInstanceCompiler<DNXXXTemplateInstance>` class and override the `compile` method. This methods accepts an `DNXXXTemplateInstance` instance as a parameter and compiles it into a standard `CompiledDNRule` instance (e.g. into a raw rule form), which it then returns.

Add a new section to the `DNGroupDetailPage` component and implement the `NewDNXXXTemplateInstancePage` and `EditDNXXXTemplateInstancePage` components. You will also need to add the `DNXXXTemplateInstanceHelpPanel` and use it as a pop-up help window. Fore a more detailed explanation of this step, see Section 11.4.1.

Note that when implementing a new DN template type it is a good idea to copy and rename the classes which belong to an existing one and just overwrite the template specific parts.

12. Future Work

ODCleanStore could be extended in many ways. Suggestions for future work and improvements are maintained at a dedicated page[1] at project website. This chapter lists the most important ideas for future work. Items in bold should be implemented in future and were omitted for time reasons.

12.1 Data Processing

- Improved logging. As of now, There is one log file for Engine and a dedicated log file for each transformer instance. In addition, some information about Engine is updated in the relational database. However, structured logs in the database for transformers (e.g. with information about transformer instance, pipeline, its author etc.) may improve usage for both pipeline creators and transformer creators.
- Quarantine. Data that are considered suspicious (e.g. having too low Quality Assessment score) would be moved to an isolated dataspace where it would wait to be checked by an administrator and then accepted, corrected or deleted.
- Possibility to review data and modify it from a user interface. An advanced extension would be possibility to modify multiple named graphs in a batch operation. The provenance information for affected graphs should contain information about editing.
- Transformer that would detect identical update of a named graph already stored in the database and would only update metadata of the stored graph instead of storing both copies.
- Store the original unprocessed version of data so that the original version can be used again when a pipeline damages the data.
- Automatically download missing ontologies used in processed data.
- **Introduce concept of "post-transformers"** that would be run on data *after* they are processed by a pipeline and moved to the clean database. They could be used e.g. for updating data in the clean database depending on stored graphs, such as what Quality Aggregator does.

12.2 Quality Assessment

- Track more quality dimensions (e.g. completnes, timeliness).
- Machine learning. Rules could be derived from a given set of named graphs and their scores.

12.3 Data Normalization

- More rule templates, improved administration in case there would be too many templates.

[1] https://sourceforge.net/p/odcleanstore/wiki/Future%20extensions/

12.4 Output Webservice & Conflict Resolution

- Support queries for any SPARQL (CONSTRUCT) (not only the current URI, keyword and named graph query). This is a more complex extension as Conflict Resolution would need to be able to load metadata and `owl:sameAs` links by itself and moreover efficiently.
- Paging of results.
- Sorting of results of a keyword query by relevance for the given keyword.
- Add query which would return all resources of the given `rdf:type`.
- Generalize the interface for passing metadata to Conflict Resolution. As of now, there is an exactly defined set of metadata that are accepted by Conflict Resolution which limits extending.
- More customizable aggregate quality computation. For example, the user could have the possibility to specify how much she trusts each publisher and the aggregate quality would take that into consideration.
- More aggregation methods, such as TOP-K – K best values. This would require the possibility to parametrize aggregation methods.

12.5 Administration Frontend

- **The possibility to create a custom (deep) copy of an arbitrary group of rules.**
- **The possibility to create a custom copy of a rule.**
- **The possibility to create a custom copy of a pipeline.**
- **Labels and descriptions for transformer instances.**
- Filtering of entities displayed in Administration Frontend by values for each column. Possibility to show only entities created by the current user.
- Notification about changes in a rule group. If the author of a rule group modifies it, a notification would be sent to all users whose pipelines use the rule group and they would be provided with the option to accept of refuse the changes (would require versioning of rule groups or cloning of rule groups.).
- Transformer instance templates. It would be possible to assign a pre-prepared transformer instance to a pipeline which could contain transformer configuration, assigned rule groups etc.
- Possibility to upload a .jar archive containing a transformer directly through Administration Frontend.
- Possibility to run affected pipelines when a rule group is deleted.
- Show list of affected pipelines for a rule group or transformer.
- Check syntactical validity of rules when they are entered. Could be implemented by running the rule on an empty testing graph and checking for an error.

12.6 Miscellaneous

- **Installer will be able to install Engine as a system service on Windows or a daemon on Linux, respectively.**

- Provide a tool (command line or with GUI) for import of large graphs. ODCleanStore is designed mainly for processing of smaller graphs (e.g. results of Quality Assessment might not be relevant for too large graphs). A large graph could be divided into multiple small graphs and sent to Input Webservice in parts.

12.7 Known Issues

- Non-ASCII characters may get broken when entered into certain fields (descriptions, ontology definitions) in Administration Frontend. This is caused by incompatibility of Spring's JdbcTemplate and Virtuoso.

13. Related Work

ODCleanStore provides means for cleaning, linking, and scoring incoming RDF data, storing it, and provides aggregated and integrated views on the data to Linked Data consumers. In addition, we support trustworthiness of the data with aggregate quality and provenance tracking.

ODCleanStore focuses on the data processing and queries over stored data. Nevertheless, the extraction process that feeds data to ODCleanStore is also important – our sister project Strigil implements a web scraper and document extractor that produces RDF data and integrates with ODCleanStore for storing the produced data.

13.1 Data Extraction

Strigil

Strigil[1] implements a web scraper and document extractor that produces RDF data and integrates with ODCleanStore as the producer of data.

Linked Data Manager

Linked Data Manager[2] (LDM) is a Java based Linked (Open) Data Management Suite to schedule and monitor required Extract - Transform - Load jobs for web-based Linked Open Data portals as well as for sustainable Data Management and Data Integration usage.

LDM data processing pipeline is similar to the data processing pipeline in ODCleanStore. LDM is a counterpart of ODCleanStore in that it provides facilities for managing the extraction process but doesn't provide any permanent storage or direct access to the data. Thus an LDM Loader could be used to send data to ODCleanStore and access it from here.

13.2 Data Processing

Linked Data Integration Framework

Linked Data Integration Framework[3] (LDIF) is an open-source Linked Data Integration Framework that can be used by Linked Data applications to translate Web data and normalize URI while keeping track of data provenance. The framework consists of a Scheduler, Data Import and an Integration component with a set of pluggable modules.

LDIF components encompass the whole process from data import and processing to integration and quality assessment. We use some of LDIF componets internally in ODCleanStore (Silk). The main difference is that LDIF is a framework other applications can built on, while ODCleanStore is a ready-to-use solution that can be easily deployed and

[1] http://sourceforge.net/p/strigil/home/Home/
[2] http://www.semantic-web.at/linked-data-manager
[3] http://www4.wiwiss.fu-berlin.de/bizer/ldif/

managed via a web interface. Differences in quality assessment and data aggregation with Sieve, a part of the LDIF framework, are described below. LDIF also supports provenance tracking.[4]

13.3 Data aggregation and quality

Sieve

Sieve[5] adds quality assessment and data fusion capabilities to the LDIF architecture. It uses metadata about named graphs in order to assess data quality, agnostic to provenance vocabulary and quality models. Sieve uses customizable scoring functions to output data quality descriptors. Based on these quality descriptors (and/or optionally other descriptors), Sieve can use configurable FusionFunctions to clean the data according to task-specific requirements.

Sieve offers functionality similar to our Conflict Resolution component; however the purpose of Sieve in LDIF is different - it aggregates data while being stored to the clean database (unlike Conflict Resolution used at query time). This may be suitable when the desired data are known in advance but is not sufficient for open Web environments, where every consumer has different requirements on the aggregated data. Furthermore, ODCleanStore provides quality for each result statement where Sieve computes quality only for whole named graph.

Integration systems in relational databases

The problem of integration of heterogeneous data (solved in ODCleanStore for RDF data) is solved by several systems for relational databases, e.g. Aurora[6] or Fusionplex[7].

Aurora is an integration system of heterogenous data residing in relational and object-oriented databases, i.e. deals with non-RDF data; its query model enriches the SQL SELECT by enabling to define attribute conflict resolution functions (e.g. age[ANY] means that any attribute value for the attribute age is used in the query) and record conflict resolution function, which deal with key attributes of the records. ODCS offers more built-in aggregation methods, on the other hand, Aurora allows user defined attribute aggregation functions; in ODCS, record conflicts are either discovered by linkers, or there is no record conflict.

[4] See Figure 2 of LOD2 Deliverable 4.3.2
[5] http://sieve.wbsg.de/
[6] http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.50.7261
[7] http://dl.acm.org/citation.cfm?id=1221048

14. Conclusion

The goal of the project was to build a server application that would automate processing – especially cleaning, linking and scoring – of data in RDF format and provide aggregated and integrated views on the processed data to Linked Data consumers. We have successfully implemented all these features in ODCleanStore and hope that the project will continue to be used and developed.

ODCleanStore brings several contributions. There are other tools for RDF data processing (see related work in Chapter 13) but they are mostly frameworks that other developers can build on. ODCleanStore was developed as a whole solution that can be readily deployed and used. It ships with an web administration interface so that all neccessary configuration can be done in a web browser. It should be noted, however, that administrators of ODCleanStore should have some technical knowledge – at least understand RDF and SPARQL query language.

Another novel aspect is provenance tracking and data fusion/conflict resolution. This feature should support addoption of Linked Data by justifying why users should (or shouldn't) believe the data they are presented and where they can verify it and thus increase trust in Linked Data. Unlike other solutions, ODCleanStore solves conflicts at query time which is suitable for the open Web environment where every consumer may have different requirements.

We hope that ODCleanStore will provide basis for further work with Linked Data and other developers will extend it and customize it for their own purposes. We introduced the concept of pluggable transformers so that whatever functionality is needed can be added.

Several papers, listed in Appendix E, about ODCleanStore were accepted at scientific conferences. The reception was positive and we used the feedback to improve our project. In addition, ODCleanStore was evaluated on real data, e.g. data extracted from information system for public contracts[1] run by the Ministry of Regional Development in the Czech Republic. We also used the feedback to improve our project and adapt it to a real-world use case.

We hope that ODCleanStore will prove to be a useful tool for RDF data processing and will contribute to adoption of Linked Data. It was designed as a general tool for Linked Data management with basic data processing capabilities. We hope that the development will continue, new uses for ODCleanStore will emerge and the project will be extended and customized for the particular use cases.

[1] http://www.isvzus.cz/

A. Team & Work Progress

The Team

The project was solved in team of five people with the following responsibilities:

Jan Michelfeit
Conflict Resolution, Query Execution, project management, documentation.

Petr Jerman
Engine, Input and Output Webservices.

Dušan Rychnovský
Administration Frontend, global configuration.

Jakub Daniel
Quality Assessment, Data Normalization, Administration Frontend, documentation.

Tomáš Soukup
Linker, ontology management, debugging.

Our supervisor was RNDr. Tomáš Knap.

Project phases

The project was developed according to the classical waterfall model.

We started with the initial analysis and specification of main functionalities during the first two months, approximately. We devoted a lot of time to estabilishing goals of the project and finding real use-cases because the assignment of the project was not very specific in this point. The most important use case was integration with project Strigil for processing of data about public contracts scraped from web pages and XLS documents. Other use cases included student projects at the Faculty of Mathematics and Physics.

After the initial period, we divided responsibilities over the main functional units and started researching available tools and technologies and building a prototype. The output of this phase was our choice of technologies (most motably Virtuoso database and Silk), a basic working prototype and an outline of software architecture. The prototype provided us with a proof-of-concept and helped us to realistically estimate the scope of features that could be implemented in the given timeframe.

After that, we started working on the actual implementation, which was the longest phase lasting approximately 4-5 months. We started by defining interfaces between components, building the components and finally integrating them. In order to avoid pitfalls of the waterfall model, we maintained a working prototype during development and whenever a major functionality was finished, it was integrated to the prototype and evaluated. This helped us to get feedback, re-evaluate use cases and continuously test our results.

The last two months were devoted to parallel testing, documentation writing and debugging according to workload of team members. We regularly produced releases which were deployed to a dedicated server and tested on real data.

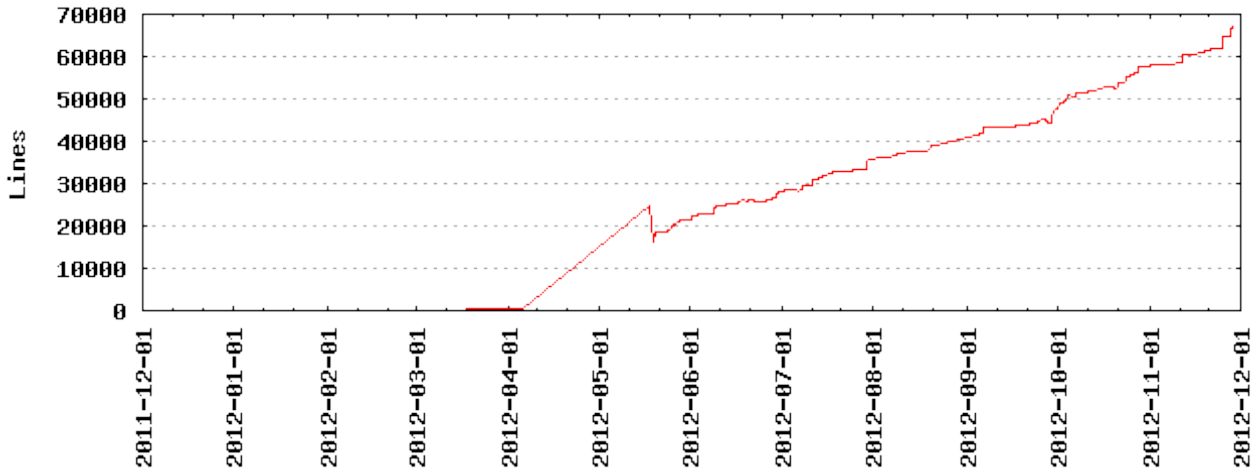

Figure A.1: Graph of number of lines of code during development

FigureA.1 presents the number of lines of code in our repository in time of the development. The graph clearly shows rapid development of a prototype in May 2012 and the even distribution of our work afterwards which is the result of good planning and unrelenting effort.

Organization

Since the very beginning of the project, we held regular weekly meetings and used several tools to facilitate management of the project. We estabilished an internal mailing list, a project page at SourceForge with a wiki tool for working documents, a public and private repository, an issue tracker and a testing server.

The exact process was flexibly adjusted to the current project phase and circumstances. During the analysis phase, we held weekly personal meetings and used our wiki to present research results, track ideas and discuss suggested solutions. During the development phase, we held regular meetings either in person or over the internet once or twice a week. A member of the team would be typically assigned a task, write specification which would be discussed by the other team members, implement the task, and the result would go through the process of code review, integration and eventually release. In the final phases, we prefered immediate contact in our mailing list to regular meetings and employed the issue tracker.

An immensely important things were regular releases during the development of the project. Since we had several oportunities to present our efforts at various meetings at conferences, we were pushed to finish integrated pieces of work, integrate them to the working prototype, debug them and publish. In addition, it provided us a valuable feedback.

B. Glossary

RDF-related

RDF
: Resource Description Framework, a language for representing information about resources in the World Wide Web[1]

RDF triple
: Statement about a resource expressed in the form of subject-predicate-object expression

URI
: Uniform Resource Identifier, identifies RDF resources

Named graph
: A set of related RDF triples (RDF graph) named with a URI[2]

RDF quad
: An RDF triple plus named graph URI (subject, predicate, object, named graph)

Ontology
: Representation of the meaning of terms in a vocabulary and of their interrelationships

OWL
: The Web Ontology Language[3]

SPARQL
: RDF query language[4]

RDF/XML
: An XML-based serialization format for RDF graphs[5]

TTL
: Turtle – Terse RDF Triple Language[6]; a human-friendly alternative to RDF/XML

Data & Data Quality

Dirty (staging) database
: Database where incoming data are stored until they are processed by a processing pipeline (e.g. clean, linked to other data, etc.)

[1] http://www.w3.org/RDF/
[2] http://www.w3.org/2004/03/trix/
[3] http://www.w3.org/TR/owl-features/
[4] http://www.w3.org/TR/rdf-sparql-query/
[5] http://www.w3.org/TR/rdf-syntax-grammar/
[6] http://www.w3.org/TeamSubmission/turtle/

Clean database
: Database where incoming data are stored after they are successfully processed by the respective processing pipeline; this database can be accessed using the Output Webservice

`Payload` graph
: Named graph where the actual inserted data, given in the `payload` parameter of Input Webservice, are stored

`Provenance` graph
: Named graph where additional provenance metadata, given in the `provenance` field of Input Web Service, are stored

Metadata graph
: Named graph where other metadata about a `payload` graph (such as source, timestamp, license, etc.) are stored

Attached graph
: Named graph attached to a `payload` graph by a transformer

Named graph score
: Quality of a single (`payload`) named graph estimated by the Quality Assesment component and stored in the database, expressed as a number from interval [0,1]

Publisher score
: Average score of named graphs from a publisher

Aggregate quality
: Quality of a triple in the results calculated by the Conflict Resolution component during query time, expressed as a number from interval [0,1]

Data Processing

Pipeline
: A configurable sequence of transformers that is used to process a named graph. The pipeline to process data sent to Input Webservice can be selected explicitly, or the default pipeline is used.

Transformer
: A Java class which implements the *Transformer* interface that and is registered in ODCleanStore Administration Frontend by an administrator.

Transformer instance (or transformer assignment)
: Assignment of a *transformer* to a *pipeline*. A single transformer can be assigned to multiple pipelines (or even to a single pipeline multiple times), thus creating multiple transformer instances.

Rule
: Some transformers included in ODCleanStore can be configured in Administration Frontend by rules. Rules are grouped together to *rule groups*.

Rule group
: A group of transformer *rules*. Rule groups can be assigned to transformer instances.

User Roles

ADM
: Administrator

ONC
: Ontology creator

PIC
: Pipeline creator

SCR
: Data producer (scraper)

USR
: Data consumer

C. Relational Database Schema

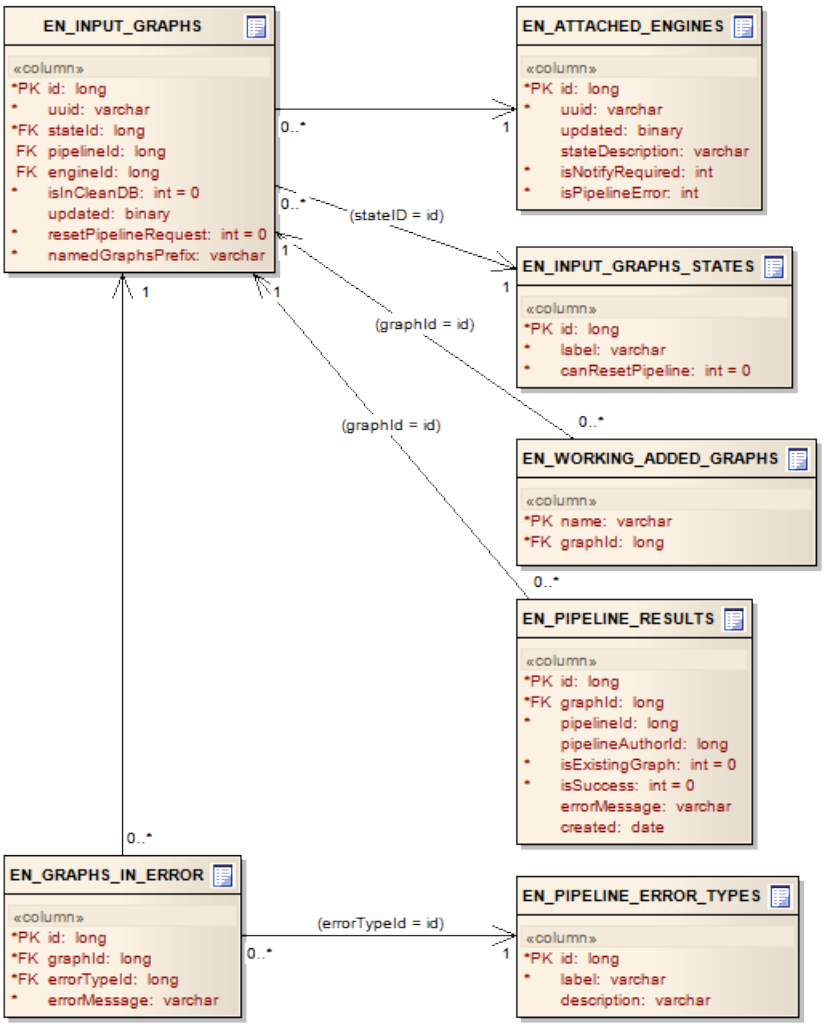

Figure C.1: Diagrams of database tables related to Engine

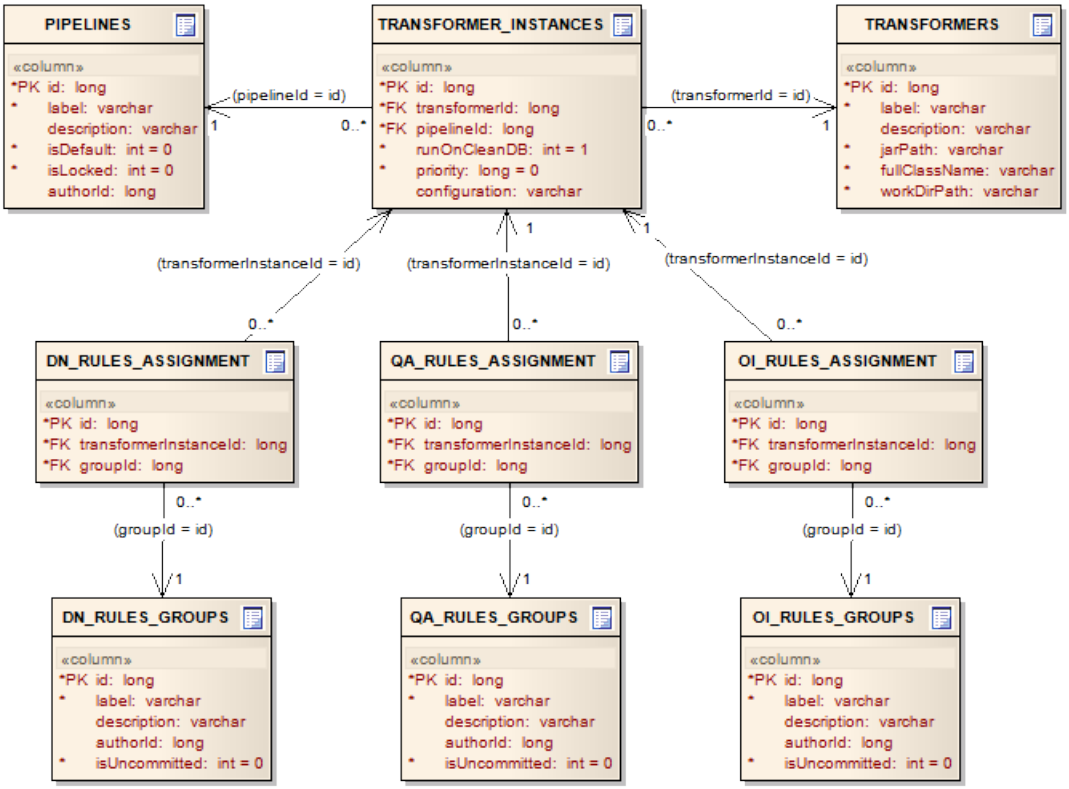

Figure C.2: Diagrams of database tables related to pipelines

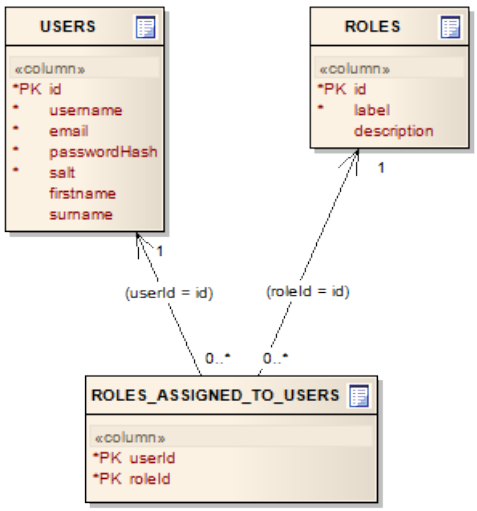

Figure C.3: Diagrams of database tables related to Administration Frontend

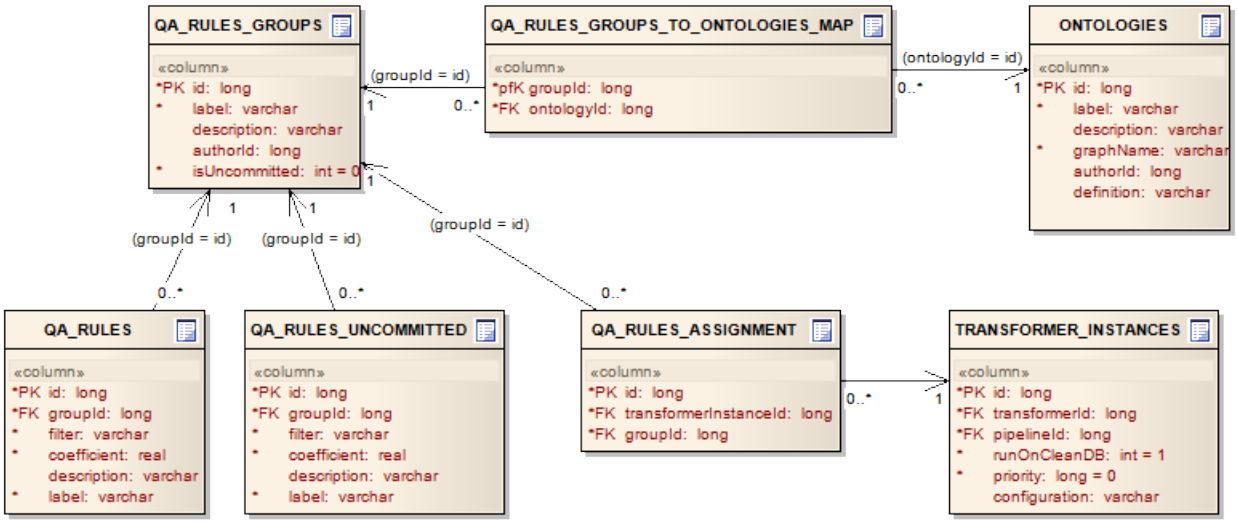

Figure C.4: Diagrams of database tables related to Quality Assessment

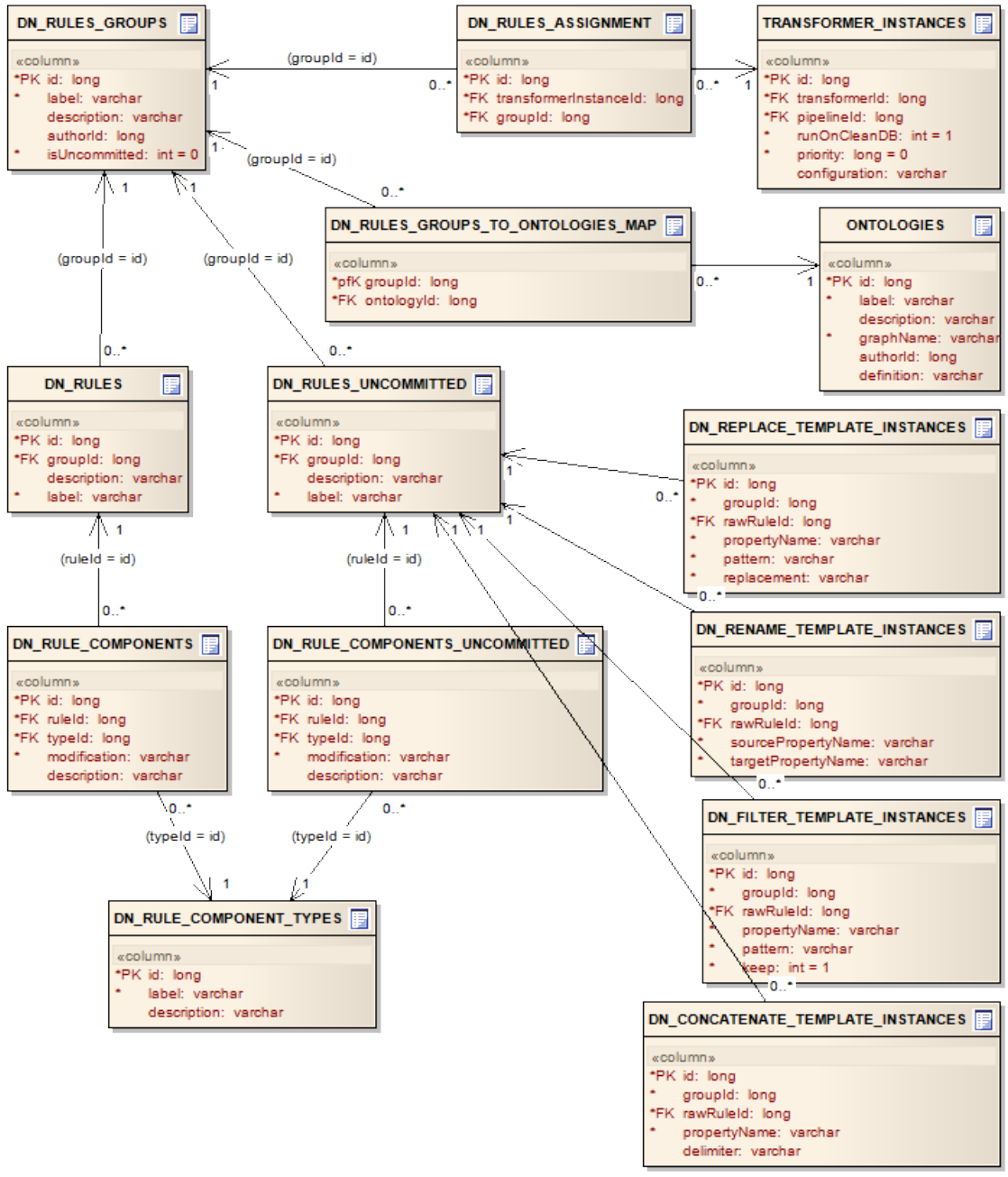

Figure C.5: Diagrams of database tables related to Data Normalization

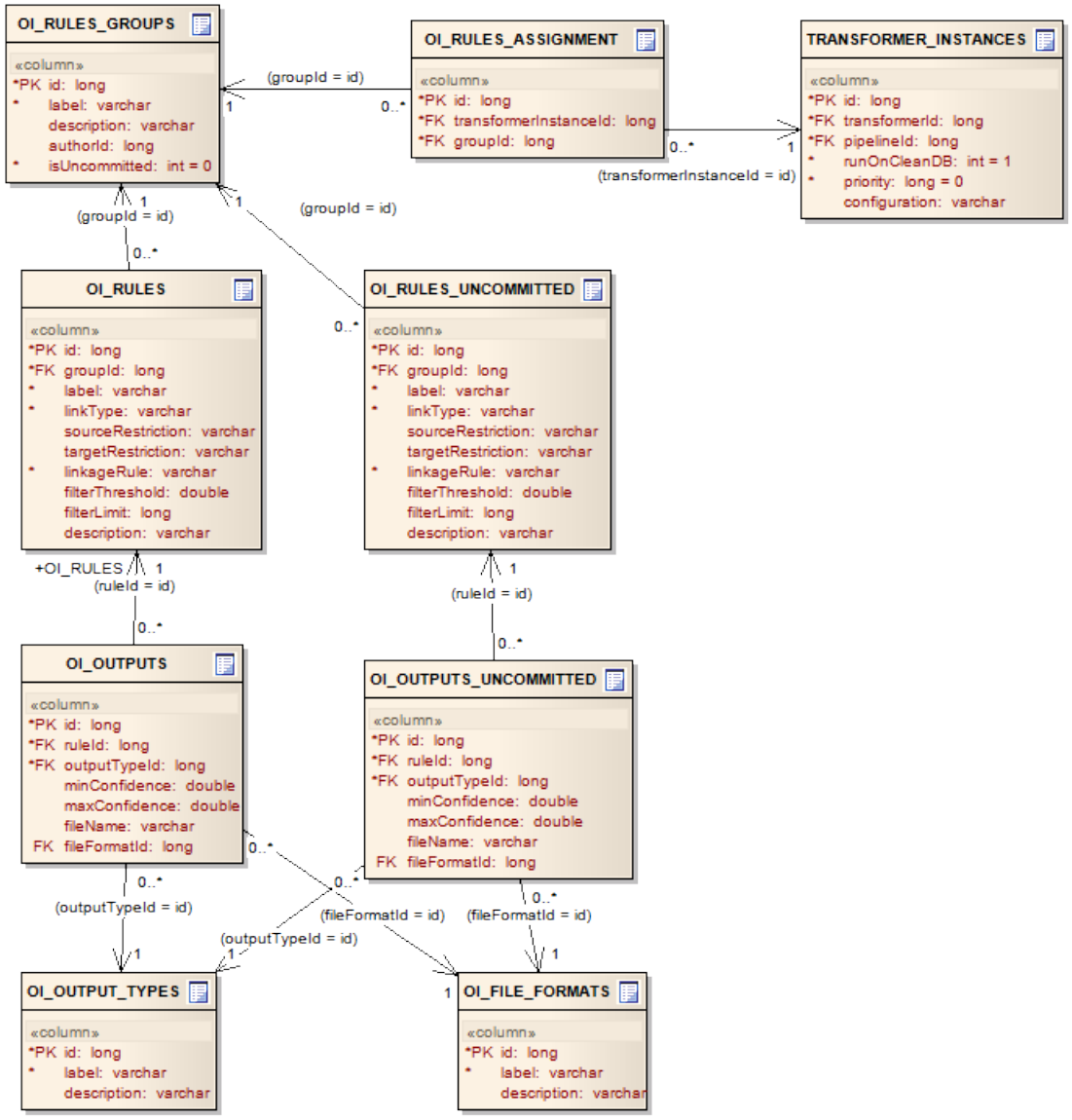

Figure C.6: Diagrams of database tables related to Linker

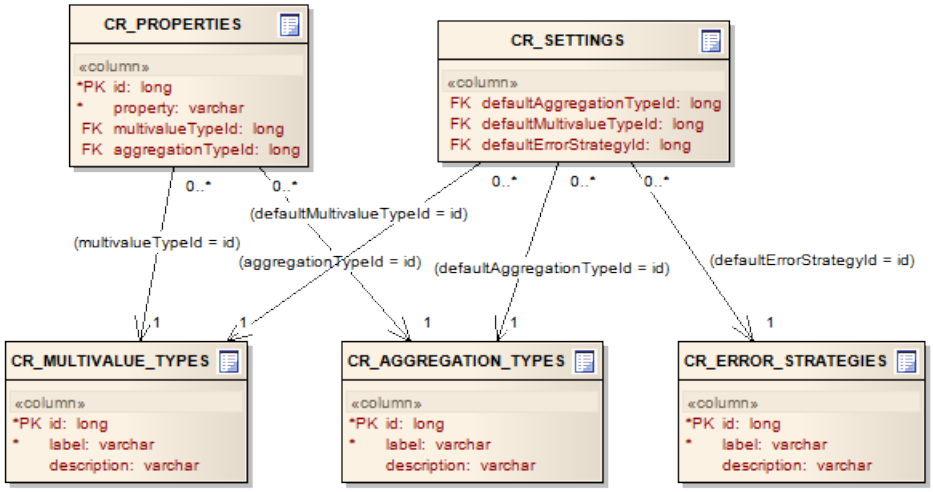

Figure C.7: Diagrams of database tables related to Conflict Resolution

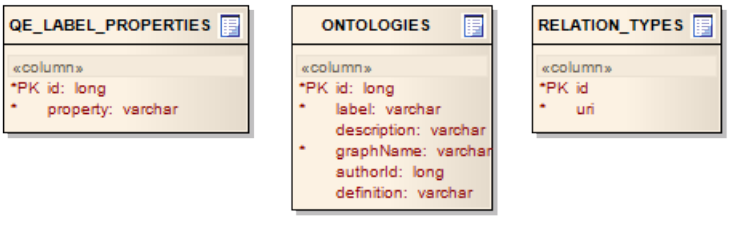

Figure C.8: Diagrams of miscellaneous database tables

D. List of Used XML Namespaces

Prefix	URI
odcs	http://opendata.cz/infrastructure/odcleanstore/
w3p	http://purl.org/provenance#
dc	http://purl.org/dc/terms/
rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs	http://www.w3.org/2000/01/rdf-schema#
owl	http://www.w3.org/2002/07/owl#
xsd	http://www.w3.org/2001/XMLSchema#
dbpedia	http://dbpedia.org/resource/
dbprop	http://dbpedia.org/property/
skos	http://www.w3.org/2004/02/skos/core#

Table D.1: List of used XML namespaces

E. Publications

[1] Tomas Knap, Jan Michelfeit, Jakub Daniel, Petr Jerman, Dusan Rychnovský, Tomás Soukup, and Martin Necaský. ODCleanStore: A Framework for Managing and Providing Integrated Linked Data on the Web. In *WISE*, pages 815–816, 2012.

[2] Tomas Knap, Jan Michelfeit, and Martin Necaský. Linked Open Data Aggregation: Conflict Resolution and Aggregate Quality. In *COMPSAC Workshops*, pages 106–111, 2012.

[3] Jan Michelfeit and Tomas Knap. Linked Data Fusion in ODCleanStore. In *International Semantic Web Conference (Posters & Demos)*, 2012.