Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

ACHIM DÄUBLER
MASTER THESIS

# DESIGN AND IMPLEMENTATION OF AN ADAPTABLE METRIC DASHBOARD

Submitted on 4 September 2017

Supervisor: Prof. Dr. Dirk Riehle, M.B.A., Maximilian Capraro, M.Sc.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

# Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

---

Erlangen, 4 September 2017

# License



---

Erlangen, 4 September 2017

# Abstract

Many software companies use open source development practices inside the company‘s boundaries, which is called inner source. The Collaboration Management Suite (CMSuite), developed by the Open Source Research Group at the Friedrich-Alexander-University Erlangen-Nuernberg, is a software tool for extraction, analysis, and visualization of data and metrics regarding inner source.
Prior to this thesis, CMSuite lacked features to visualize metrics and let stakeholders define their own metrics and visualizations. A programmer had to write code from scratch, where he defines a metric and then visualizes the result. Furthermore is not fully researched, which metrics will be important in the future, so adding new ones without wasting much time is desirable. This thesis discusses a new Java-based REST-service, which makes it possible to easily add and define new metrics, using the data integration tool Pentaho Kettle. The result is then visualized in an AngularJS 2.0 client component for a metric dashboard. Now the user does not have to write any code, but only has to define a metric with the help of Kettle and can see the results of his metric, immediately. Thus, this addition to CMSuite will enable him to save time and test new metrics much more efficiently.

# Contents

# 1 An Adaptable Metric Dashboard

Inner Source (IS) is the use of open source software development practices and the establishment of an open source-like culture within organizations (Capraro & Riehle, 2017) and an increasing number of companies is making use of it. Riehle, Capraro, Kips and Horn (2016) note that the number of practitioners indicates the relevance of inner source research. The reason for the increasing interest is, that inner source practices can have many benefits for a company. Among others, these benefits include increased software reuse as everyone in the company can access other teams source code, better code quality due to more people looking at the code and accelerated development speed. (Stol & Fitzgerald, 2015; Vitharana, King & Chapman, 2010).
In order to understand inner source collaboration, the Open Source Research Group at the Friedrich-Alexander-University Erlangen-Nuernberg has developed the Collaboration Management Suite (CMSuite). With its help, software organizations can continuously extract patch-flow (flow of code contributions). Based on this data, different metrics, which are a measure for the evaluation of inner source, can be defined. By visualizing the results of these metrics, insights can be gained, which help developers and development-managers in decision-making and managing the inner source process.

The contributions of this thesis are the following:

- A domain model for data transformations and metric results
- The extension of CMSuite to enable users to define metrics at runtime
- The addition of client components for a dashboard for metric results
- Support for visualization of three types of results

## 1.1 Previous Work

CMSuite provides means to measure code-level collaboration. That is solely the collaboration in the form of patches that are contributed to repositories. A patch is a small package of code changes with bug fixes, new features, or other additions to an ISS component (Capraro & Riehle, 2017). Patch-flow is the flow of code contributions across intra-organizational boundaries such as project, organizational unit, or profit/cost center boundaries.
The Open Source Research Group Erlangen-Nuernberg developed the so called patch-flow crawler as part of CMSuite, which searches various repositories for their received commits. It adds information about the organizational unit the commit originated from and the receiving IS project. The resulting patches are stored in a database. Because the origin and destination of the patch are known there is a measure for the collaboration in the company even across intra-organizational boundaries such as projects or organizational units. Using the database, one can define arbitrary metrics to gain information about inner source performance that is valuable for the company. Which metrics are useful is not fully clear yet. Capraro and Riehle (2017) suggest that future research is needed to identify metrics to measure the progress of IS adoption and the quality of IS programs, projects, and communities.

## 1.2 Purpose

Until now, a developer had to programmatically define how raw patch-flow data and other data is transformed for every new metric. Then he had to write code in order to visualize his results. As mentioned earlier, it is not known which metrics will be important in the future, or in which specific organizations which ones will be of interest, as they might have different needs. It is thus desirable to have a means to try new metrics without loosing too much time.
The purpose of this work is to extend CMSuite with client and server components for a metric dashboard. The components allow users to add metrics not foreseen at the initial design time. Addition of new metrics requires no changes to server code and no changes to client code. The results of metric calculations are visualized in an Angular 2.0 dashboard component. The metrics are easily uploaded through the web-client in the form of files. The server components are Java-based REST-services, which provide the methods to receive the files and persist them. Furthermore they provide the methods to retrieve the results of metric calculations.

## 1.3 Requirements

### 1.3.1 Stakeholders

The main stakeholders for the newly designed components are:

- A *Developer* of CMSuite. He is a programmer, who develops CMSuite.
- An *Administrator* of CMSuite. He oversees the operations of CMSuite.
- A *Metric Designer*. His job is to design new metrics in order to measure the performance of a company.
- An *Inner Source Stakeholder*. He is somebody who is in any kind involved in the inner source process.

Of course there are more roles related to inner source, like contributor, committer, etc. They are essential to define when the goal is to understand inner source. However, this is not the primary motivation of this thesis. The motivation is to enable IS stakeholders to manage inner source, based on facts and insights obtained by applying metrics. So these roles were combined to one "Inner Source Stakeholder" role, which is sufficient for defining the requirements.

### 1.3.2 Functional Requirements

When defining the functional requirements for the new components, the following user stories were found, which describe those requirements:

1. As a metric designer, I want to add new metrics at runtime, without having to change code on the client- or server-side and I do not need to ask a developer to change code or the administrator to restart the system.
2. As a metric designer/administrator, I want to be able to view a list of uploaded metrics, with information like name, last successful execution, etc.
3. As a metric designer/administrator, I want to be able to kick off a run of all metrics from a single end-point.
4. As a metric designer/administrator, I want to be able to view a list of all runs, which also shows details about the run like date, status, etc.
5. As an inner source stakeholder, I want to display visualizations of metric results for either inner source projects, organizational units or individuals of the company.

6. As a metric designer, I want to choose from a set of available visualizations, so that I can choose which visualization fits the result data of my metric calculation best.
7. As a metric designer, I want to write metrics that produce one of the following results: A result that consists of only one value, a time series result (time, value pairs) or a categorized value result (category, value pairs).
8. As an inner source stakeholder, I want to have support for metrics like these:
   (a) Metrics for the whole company (= the root organizational unit)
      i. Percentage of patch-flow between different types of units
      ii. Amount of IS projects hosted
      iii. Amount of active IS projects hosted
   (b) Metrics for organizational units
      i. Amount of IS projects hosted
      ii. Amount of active IS projects hosted
      iii. External contributions received
      iv. External contributions contributed
   (c) Metrics for IS projects
      i. Development activity (amount of patches)
      ii. Development activity by contributing units (by type of unit)
      iii. Amount of contributing units (by type of unit)
      iv. Amount of contributing users
   (d) Metrics for persons
      i. Number of IS projects contributed to
      ii. Amount/percentage of contributions to foreign units (by type of unit)

### 1.3.3 Non-Functional Requirements

1. The time a metric result takes to be displayed in the web-client is a few seconds at maximum.
2. The metric designer does not have to be a programmer. As a result this means defining new metrics can be done with a tool using a GUI rather than manual coding.
3. The tool should not be programmed from scratch but available software should be used, which has to have a permissive licence, to save time and work for developers.
4. The tool offers the possibility to run the metric definitions from Java in order to be able to integrate it seamlessly into CMSuite.
5. The system has to properly handle faulty metric definitions so that nobody has to restart or manually fix the state of the system.

# 2 Architecture and Design

## 2.1 Basic Design Decisions

Prior to this thesis, CMSuite was able to extract patch-flow data and store it in a patch-table in the database. Also the information about the organization is stored in different tables. However, CMSuite lacked components to define metrics, that use this information and to execute them. Furthermore, components for visualizing the results of executed metric calculations were also missing.
The fundamental question of the design phase is how the metrics should be defined and executed. Several ideas are discussed in this chapter.

The first possible solution is to use *design-time defined Java classes for metrics*, meaning the metrics are defined in Java, ideally by extending an existing abstract metric class.
Of course this conflicts with many requirements, as the user would have to be a programmer. More importantly though, the module handling metrics would have to be recompiled everytime a metric is added.

This could be avoided by using a *plugin-mechanism for metrics*, which allows for adding Java-classes at runtime. This could be done by using Java's internal discovery process, using its ServiceLoader ("Creating Extensible Applications", 2017). Another approach would be to use third party software like a component framework like OSGI[1]. Both let the programmer define a services, which can then be loaded at runtime.
However, using this approach, the metric designer still would have to be a programmer.

Another solution is to use *pure SQL to define metrics* on the underlying tables of CMSuite. They could be uploaded or directly entered in the web-client.
For this the metric designer would not have to be a Java-programmer, but he has to know SQL. However, defining metrics using SQL would be too complicated, especially when dealing with organizational units. These have a hierarchical order

[1] https://www.osgi.org/developer/specifications

and their connection is defined via references in a table containing the links. A complex metric would be nearly impossible to write in SQL.

A further idea is to *define a DSL* (domain specific language), which is tailored for CMSuite and which makes it easy to extract the data of the tables and transform them. A file containing the metric definition, using the DSL could simply be uploaded to the server and the server would interpret it and execute the definition.
The metric designer would not have to be a programmer, but he would have to learn the DSL. The big drawback is that this comes with lots of additional work to create and maintain the self-defined DSL. So this solution does not violate any functional requirements, but from a developer's point of view it brings more work than any of the other solutions.

The next, and chosen, solution is to use an open-source *data integration tool for metric definitions*. These tools typically offer a GUI, which lets the user define how data is transformed. Of course, it has to be possible to execute the metric definition from Java, so the tool can be integrated into CMSuite.
Depending on the data integration tool, this approach is conform with all functional and non-functional requirements. The implementations in the thesis are based on Pentaho Kettle, as it does not collide with any requirements. It will be further explained why and how Kettle is used in the next chapters.

## 2.2 Third-Party Tools

### 2.2.1 Pentaho Data Integration (Kettle)

As mentioned in non-functional requirement 2, a tool is needed which can be used to implement different metrics. For this it has to extract the raw patch-flow data from the database, transform the data and store (loads) it to a persistent storage. In the context of data integration such tools are commonly known as ETL (extract, transform, load) tools (Casters, Bouman & Van Dongen, 2010). There are several popular open source tools, such as Pentaho Data Integration[2] (also known as Kettle) and Talend Open Studio[3]. CMSuite now uses Pentaho Data Integration, which is introduced in the next chapter.

[2]http://community.pentaho.com/projects/data-integration
[3]https://de.talend.com/products/talend-open-studio

#### 2.2.1.1 General Info

Pentaho is a business intelligence (BI) software company[4]. It offers open source products which, among many other features, provide the desired ETL capabilities. Pentaho follows a dual-license strategy, which is a known open core business model (Lampitt, 2008). This means Pentaho offers enterprise and community editions for most of their products. The enterprise edition contains extra features not found in the community edition. Development to the core engines in the platform - Analyzer, Kettle, Mondrian are all included in the community edition (CE). These are frequently enhanced by the company itself. But the community also contributes to the overall project - mostly QA, but also bug-fixes (Moody, 2017). Features built on top of the core are only available in the enterprise edition (EE). The enterprise edition is available under a commercial license. The community edition is a free open source product licensed under permissive licenses. The projects are owned by Pentaho, which managed to create an active community around their projects, while generating revenue from them. This so called single-vendor commercial open source business model brings many benefits for the company and the community, as the company offers a professionally developed product of compelling value to the community that this community is free to use under an open source license (Riehle, 2012). On the other hand, the company is solely in control of their products and can relicense or change them as they consider it appropriate.
For this project only the community edition of Kettle is used. It is licensed under the Apache License, Version 2.0 ("Pentaho Licenses", 2017).

#### 2.2.1.2 Kettle Transformations

Pentaho Data Integration (PDI), also known as Pentaho Kettle, offers the mentioned extraction, transformation, and loading (ETL) capabilities. It consists of a data integration (ETL) engine, and GUI applications that allow the user to define so called transformations. In Kettles terminology a transformation handles the manipulation of rows or data. It consists of one or more steps that perform core ETL work such as reading data from files, filtering out rows, data cleansing, or loading data into a database (Casters et al., 2010).


**Figure 2.1:** example transformation

[4]https://www.pentaho.com

Spoon is a graphical design environment that allows users to design transformations by choosing from a wide range of predefined steps. The user can simply drag and drop the steps onto his workspace. He defines the properties of each step and connects them with so called hops. The finished transformation contains a set of connected steps. The first step typically loads data from a data source, such as a database or from the file-system. The subsequent steps define how data is altered until it flows into the last step, which usually stores the data to a destination, such as a database or the file-system. By connecting the steps one can define arbitrary transformations of the data.
Figure 2.1 shows a very simple example of a transformation that is opened in Spoon. It reads rows from a database table, filters them by a criteria defined in the "Filter Rows"-step and writes the filtered rows into the destination table.
The data that flows from step to step over a hop is a collection of so called rows, containing one or more fields. Rows can be imagined the same as rows of a database. Its fields can have different data formats, like numbers, dates or strings.
Steps are executed in parallel. When a transformation is executed, one or more copies of all its steps are started in separate threads. They keep reading rows of data from their incoming hops and pushing out rows to their outgoing hops until there are no more rows left. As the data is processed in this streaming fashion, the memory consumption it kept minimal. This is an important requirement for business intelligence applications, that usually deal with huge amounts of data (Casters et al., 2010).
From Spoon, transformations can be saved to the file-system in the form of xml-structured project files, with the file-ending "ktr". They can be opened by Spoon and can be edited further and can also be executed from there. More importantly for this project, they can also be interpreted and run by Kettle's data integration engine, which is explained in the next chapter.

#### 2.2.1.3 Running Kettle Transformations from Java

As mentioned in non-functional requirement 4, for our purposes, there has to be the possibility to run the transformation from Java. Fortunately Kettle offers the possibility to interpret and run the ktr files, using its Java API. The API is split up into different jars.
For this project, the dependency to kettle-core and kettle-engine are added. Kettle-core contains the core classes for Kettle and kettle-engine contains the needed runtime classes. As the transformations use a database connection, the JDBC driver dependency, for the database used (PostgreSQL), are added as well.
In Java one can do everything that can be done in Spoon, eg. define a new transformation, or alter an existing one. This is useful, as in chapter 3 it will be important to to overwrite an existing database connection of a transformation, so it can be tested locally with another database connection.

Also you can run the transformation with parameters, which is important as it needs some context about our environment. (the transformation-id and the id of the transformation-run).

#### 2.2.1.4 User Defined Steps

Pentaho Kettle offers the possibility to write steps yourself. This is interesting because it allows for defining steps tailored for CMSuite, which can be used to make the metric designer's work easier.
In order to write own steps, one has to implement four interfaces, so that it integrates into Kettle ("Create Step Plugins", 2017). In short, the first one mainly defines the steps settings. The second one defines the setting dialog, using SWT[5]. The third interface defines how the input rows are transformed to output rows. The fourth one defines the field variables used during processing.

#### 2.2.1.5 Why Kettle is used

Talend Open Studio for Data Integration offers the same functionality as Pentaho Kettle. It is licensed under the Apache License v2. It also provides a GUI in which one can simply put together steps, like in Spoon. It also is possible to define steps yourself and use them. Furthermore Talend also allows to run jobs (transformations are called jobs in Talend) from Java. The problem with Talend is that it operates as a code generator (Barton, 2013), which produces Java code, whereas Kettle saves its transformations as xml-styled ktr files, which are interpreted by its engine.

Via Open Studio's GUI the user can export a job as a standalone jar file. It contains all dependencies needed for running the job. In order to integrate the job into an existing Java application, it would be necessary to execute the jar file from Java, eg. by adding the jar to the build path and call the jars main class in the applications code. So the whole application would have to be recompiled everytime a jar is added. Either way the problem is, that the application is isolated and thus it is not possible to alter properties of the job from the Java code. As mentioned before, it makes sense to force the transformation or job to use the database connection defined in CMSuite, which can only be done if this property of the transformation or job can be overwritten in the code. Also if metric designers would use different versions of Talend, the jars would use different implementations. With Pentaho there is always only the version of the engine used, which is defined in the project properties. This engine then can

[5]https://www.eclipse.org/swt

interpret the xml-styled ktr files, regardless with which version of Kettle they were created.

All in all Talend is more suited to be used by itself and run its jobs from the GUI or export a job as jar, which can be run from the command line. When it comes to integrating Talend job into an existing application, which is what we want, it does not seem suitable for that.

### 2.2.2 Chart.js

For visualizing results in the web-client, a JavaScript library is needed that is open source and provides chart types suitable for displaying the different types of results.
Chart.js[6] is a simple but flexible JavaScript library that provides all necessary features. It supports all basic chart types, such as line-, bar- and pie-charts, and more. It also is highly configurable, so things like color, labels, animations, legends can be set in the JavaScript code. Furthermore it has no dependencies to other libraries.
All in all it is easy to use and not overloaded with unnecessary features. Chart.js is open source and is available under the MIT license, so it can be used in this project.

## 2.3 Domain Model

The domain model in figure 2.2 shows all classes that were derived from the requirements descriptions and user-stories. The OrgElement, Person and InnerSourceProject classes however existed before this thesis and represent an organizational unit, a person, that is part of an organization, and an inner source project. The newly introduced domain classes are the following:

- *EconomicAgent*: An interface for all classes that represent an entity for which a transformation can run. These so called agents are OrgElements (organizational units), InnerSourceProjects and Persons.
- *Transformation*: A metric is a measure for the evaluation of the quality of inner source development practices. A transformation, as it is called in Pentaho Kettle, is a description of how data is read, transformed and stored, as mentioned in chapter 2.2.1. It is saved as Kettle Transformation File (.ktr). A metric is calculated by executing a transformation, that defines the steps to create the results for this metric. The Transformation

[6] http://www.chartjs.org

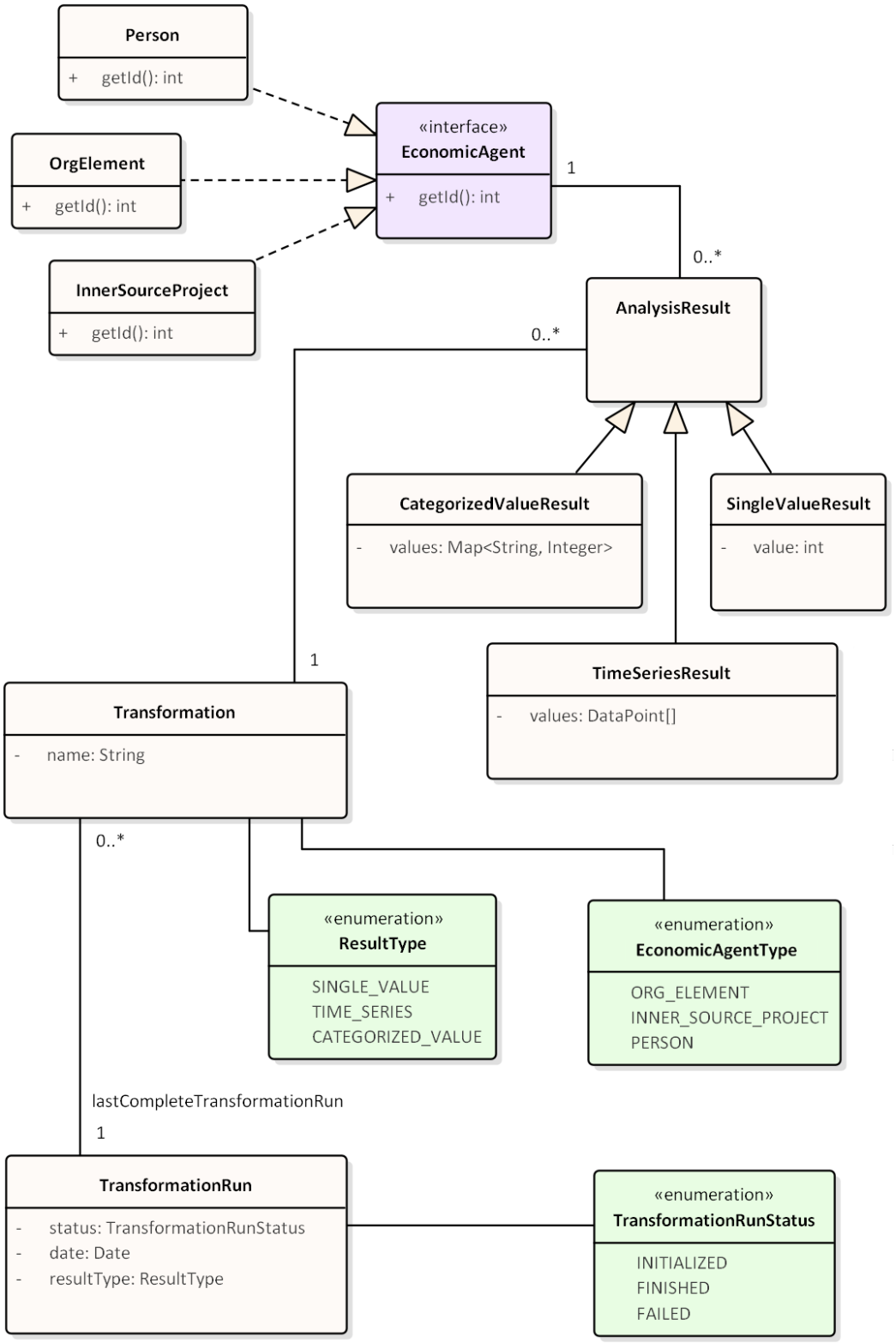


**Figure 2.2:** domain model

class contains the meta-data about the transformation: The name under which it will appear in the client, the type of result it produces and the type of agent it applies to. Also a reference to the latest successful run, in which it was executed, is stored.

- *TransformationRun*: Contains all information about an execution of transformations. It contains the date, when the run was started, as well as information about its current status. That is if the transformations are currently running, if the run finished successfully or if the run failed.
- *AnalysisResult*: Represents the base class for a result of a transformation. It contains references to the EconomicAgent it belongs to and to the Transformation that produced the result.
- *SingleValueResult*: Contains a result with one value. It is used to represent results for metrics, that produce only one value for each EconomicAgent. For example: Metric for persons: "Number of patches ever contributed".
- *CategorizedValueResult*: Contains a result with multiple key-value pairs. This is used to represent results for metrics, that produce multiple key-value pairs for each EconomicAgent. For example: Metric for inner source projects: "Number of patches ever contributed to the project per person".
- *TimeSeriesResult*: Contains a result with multiple key-value pairs, with the key being a date. This is used to represent results for metrics, that produce multiple date-value pairs for each EconomicAgent. For example: Metric for organizational units: "Number of patches originating from the unit per year".

## 2.4 Persistence

Before visualizing, the raw patch-flow data has to be transformed according to a user defined metric. It was decided that the results of such a transformation have to be persisted, as calculating it every time a visualization is requested would lead to very long response times, which conflicts with non functional requirement 1. Of course, Transformation- and TransformationRun-objects also have to be persisted. The different types of AnalysisResult are not directly persisted, but derived from the persisted ResultRows.

### 2.4.1 Database Schema

The transformation-, transformationrun- and resultrow-table were added to the existing database schema:

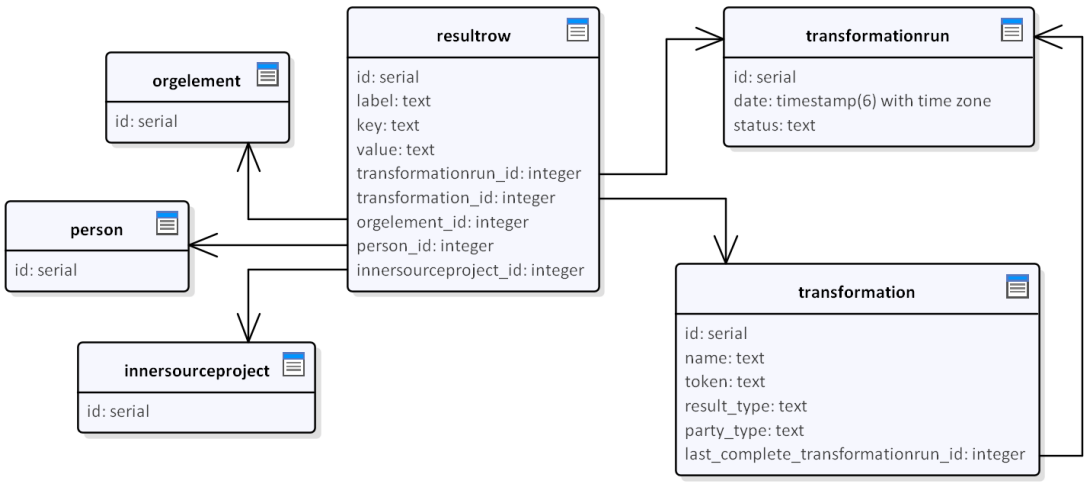


**Figure 2.3:** Database Schema

The transformation- and transformationrun-table directly correspond to their domain objects. The types of result do not appear here, as they are derived from the entries in the resultrow table, which is explained in the next chapter.

### 2.4.2 Persisting Results

When designing the result types, the question arose, how the results generated by a transformation should be represented in the database. In the domain model, the different types of results were introduced. Every type of result needs to know which transformation produced it, as well as in which run it was produced. This is because when the web-client requests a result for a transformation, it has to be clear to which one the result belongs. Also we have to know the run in order to not show results of old runs. Lastly we need information about which agent the result belongs to (so to which OrgElement/Person/InnerSourceProject).
For the specific results, different values have to be stored. In the domain-model (2.2) it can be seen, which fields the different result-types contain. For SingleValueResults this is only one value. For CategorizedValueResults, this is a Map of key-value-pairs (category is the key). Finally for TimeSeriesResults, this is an array of pairs of dates and values. The storing of the values is defined inside the transformation, so it is the responsibility of the metric designer to define which values are stored into which column. For SingleValueResults one can simply store the value in a row for each agent. But for CategorizedValueResults it would be complicated to define transformations, that store multiple key-value pairs into one row. The same goes for TimeSeriesResults. As mentioned in chapter 2.2.1, transformations work with streams of rows, so a better layout is to store each key and value into a separate row into the database. So each row created by

the transformation lands in a row in the database. Because of this the following layout for results was chosen for the table:

| tm_run_id | tm_id | person_id | project_id | orgelement_id | label | key | value |
|---|---|---|---|---|---|---|---|

Every transformation has to map each produced row to a row in this table. This works for each type of result.
Transformations, that create SingleValueResults, store a single value for each agent into the value column.
When creating CategorizedValueResults, one or multiple rows with a key and a value are stored into the key and value columns, for each agent.
For a TimeSeriesResult this works the same way, but the keys have to be dates.

## 2.5 Architecture

CMSuite consists of several services, that are segregated by concerns. In this thesis two new services are added to CMSuite. The following two goals were kept in mind during the design phase. First, loose coupling of the different services and second, high cohesion within a service.
Loose coupling means that a change to one service should not require a change to another service. As the services are standalone one service can even be changed or updated without affecting the others. The services know nothing from each other. Of course there has to be some way of communication, as they still have to work together to perform certain tasks. This communication is done via the network using HTTP/REST.
High cohesion means that related functionality can be found in one place. This is because if something has to be changed it should only be necesaary to change in one place and it should not be necessary to search in other places, as making changes in different places increases the risk of bugs.
Keeping this in mind, the first service only deals with transformations. That is storing them, deleting them or running them.
The second service, deals only with the results. So it generates results from the result-rows and offers methods to retrieve them.

## 2.6 REST

CMSuite implements a RESTful API to do its server-client communication. The term REST was coined by Fielding (2000). It is a programming paradigm for distributed systems, especially web-services. The fundamental concept of REST-APIs is the so called resource. Generally speaking, a resource is something that

can be addressed with a URI (Unique Resource Identifier). A resource has to be uniquely identifiable via its URI. It also has to be stateless. So if previously entered information is needed, it has to be sent again. Furthermore it can can be accessed via a set of common methods. These are for example the standard HTTP methods like GET, POST, PUT, etc. If a REST endpoint triggers a method, that needs parameters, these parameters are specified in the URI itself. One of the first steps in developing RESTful web-services is designing the resource model, which identifies and classifies all the resources the client uses to interact with the server (Allamaraju, 2010). The identified resources, based on the domain model, for this project are Transformation, TransformationRun and AnalysisResult.
The next step is to define the URIs under which the resources are accessible. The URIs start with the address and port, the server listens to, followed by the the URI's portions defined in the code.

The following URIs were defined for the Transformation resource:

| **URI** | **HTTP method** |
|---|---|
| /transformations?agenttype={agentType} | GET |
| /transformations/{tmId} | GET |
| /transformations/{tmId} | DELETE |
| /transformations/ | POST |

The GET method with the query-parameter agentType, takes the EconomicAgentType and retrieves all transformations for it, or all of them, if no query-parameters is specified. The DELETE and POST methods are used to delete a specific Transformation and to create a new one.

These URIs were defined for the resource for TransformationRuns:

| **URI** | **HTTP method** |
|---|---|
| /transformationruns/ | GET |
| /transformationruns/{tmId} | GET |
| /transformationruns/ | POST |

The GET methods are for returning all or a specific TransformationRun. The POST method is for kicking off a run.

The URIs for AnalysisResults look like this:

| URI |
| --- |
| /transformations/{tmId}/results/innersourceprojects/{agentId}/singlevalue |
| /transformations/{tmId}/results/innersourceprojects/{agentId}/categorizedvalue |
| /transformations/{tmId}/results/innersourceprojects/{agentId}/timeseries |
| /transformations/{tmId}/results/innersourceprojects/singlevalue |
| /transformations/{tmId}/results/orgelements/{agentId}/singlevalue |
| /transformations/{tmId}/results/orgelements/{agentId}/categorizedvalue |
| /transformations/{tmId}/results/orgelements/{agentId}/timeseries |
| /transformations/{tmId}/results/orgelements/singlevalue |
| /transformations/{tmId}/results/persons/{agentId}/singlevalue |
| /transformations/{tmId}/results/persons/{agentId}/categorizedvalue |
| /transformations/{tmId}/results/persons/{agentId}/timeseries |
| /transformations/{tmId}/results/persons/singlevalue |

These endpoints are used to retrieve the different types of result. As they are all just for retrieving, they all use the HTTP GET method. The id of the requested transformation, as well as the type of agent and the id of the agent are encoded in the URI. The URI is designed in a way, that the last part is the same of every type of agent. This allows for using inheritance with JAX-RS annotated resource classes in order to generate the URIs, which will be covered in the implementation chapter (chapter 3).

## 2.7 Server Components

In the CMSuite project, each service resides in its own module. Each module contains a service layer and a REST-service layer.
The purpose of the service layer is to abstract away the the underlying technologies. So for example a module that needs to save a transformation can simply call the services save() method but does not need to know where and how the service actually saves the object. Furthermore this way the implementation can easily be changed without affecting other modules, but only affecting the service module. From the perspective of interfacing layers, the service layer defines the applications set of available operations (Fowler, 2008).
The REST-service layer is the layer that exposes the API, that is used by the webclient. The REST-service layer contains the resources that apply to the components domain. The resources make use of the underlying service layer. Furthermore it holds a main application class in order to run the REST-service.
In the component diagram in figure 2.4, the two new modules, and their components can be seen. They will be explained in the following chapters.

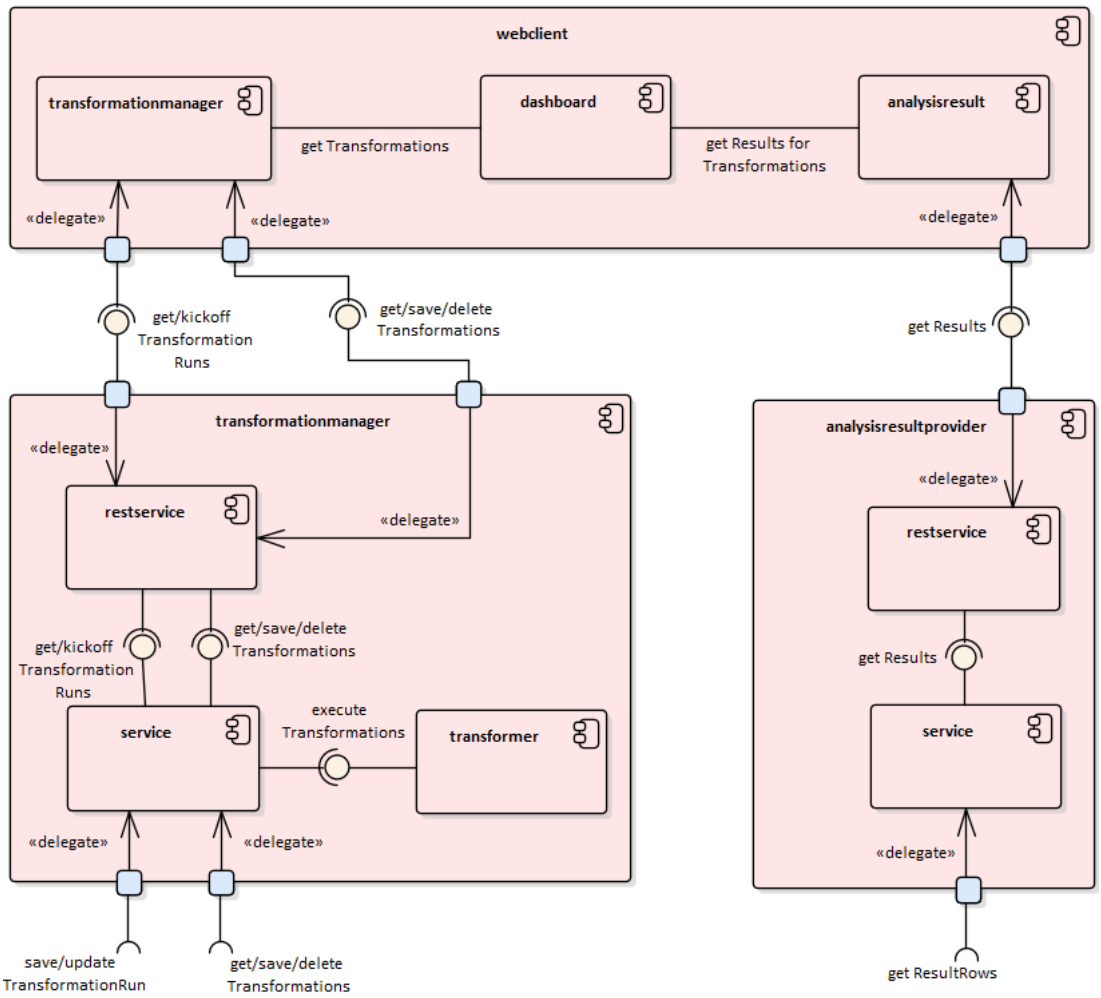


**Figure 2.4:** component diagram

The figure shows, that the services require an API to access the underlying database. This logic is contained in a separate module, that represents the persistence layer. It contains the Data Access Objects (DAO) for the domain objects, which will be further explained in chapter 3.2.

### 2.7.1 Transformation Manager

The *transformationmanager* component is responsible for storing and retrieving Transformations, which can be seen in figure 2.4. Furthermore it manages the execution of transformations, which includes keeping track of the current state of running, finished or failed executions.

#### 2.7.1.1 Overview

The transformationmanager contains a service and a restservice module. Additionally it contains the so called transformer component. The following class diagram shows the classes contained in the module:

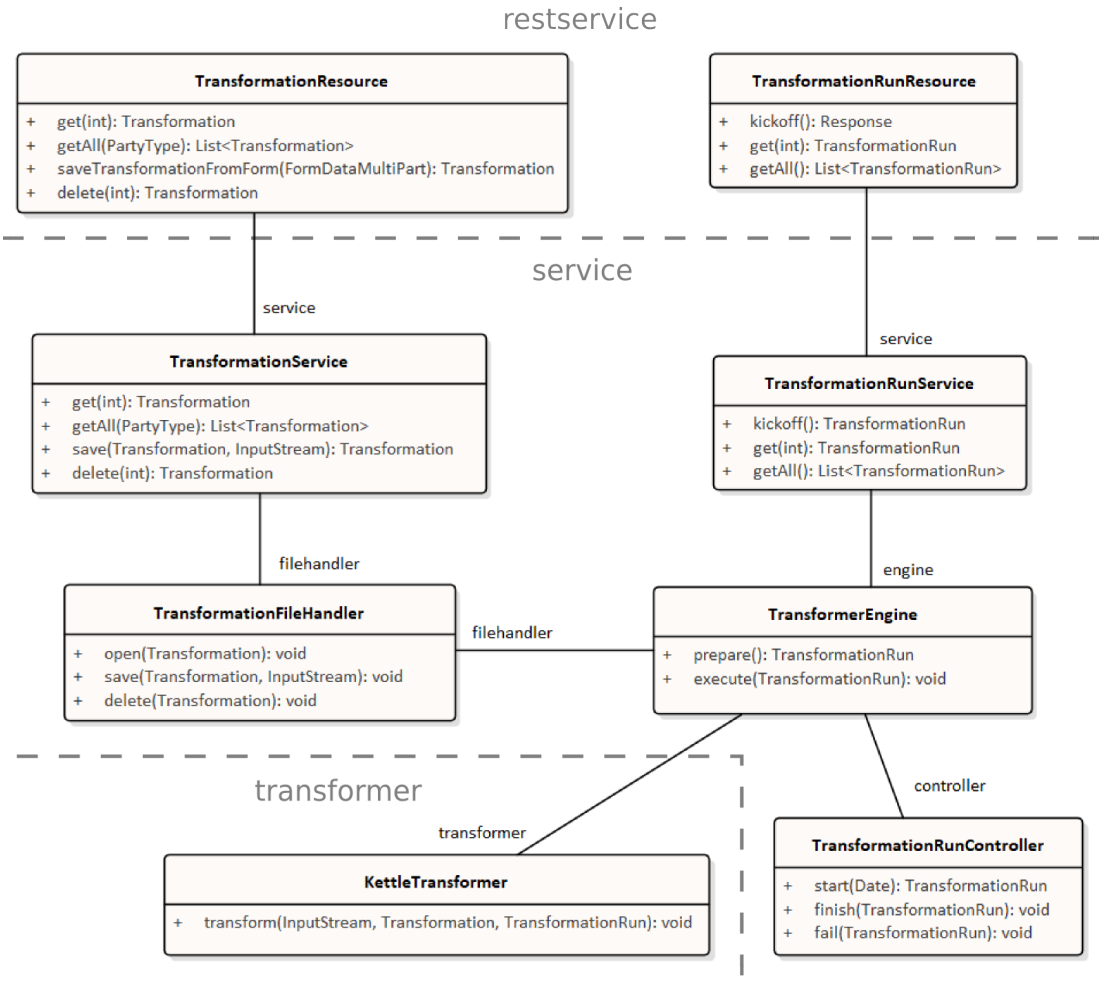


**Figure 2.5:** class diagram of the transformationmanager module

The service module represents the service layer. It provides methods to save, load and delete transformations. Furthermore it contains a method to kickoff all transformations, which creates an entry in the transformationrun table. Also it provides methods to return TransformationRun objects from the table. These contain information about the status of the runs.

The restservice module defines a RESTful API to save and load Transformations and to kickoff their execution. Calling the API methods delegates to the underlying service, which is responsible for the actual execution of the tasks.

The service delegates the execution of a transformation to the so called transformer module. This module contains all dependencies to needed Pentaho artifacts, namely "kettle-core" and "kettle-engine". This separation into a new module allows to replace the use of pentaho by another data integration tool easily, should it ever be necessary.

#### 2.7.1.2 Connection between CMSuite and Pentaho Kettle

As mentioned in chapter 2.2.1.3, Kettle transformations can be run from Java. This is what happens in the transformer module. The TransformerEngine calls the KettleTransformers transform() method and passes it the transformations ktr file as InputStream, as well as the corresponding Transformation object and the TransformationRun object of the current run (see figure 2.5). Using the Kettle API, the transformation can be run and parameters can be passed to it.

In the following image this connection between CMSuite and Pentaho Kettle is visualized:

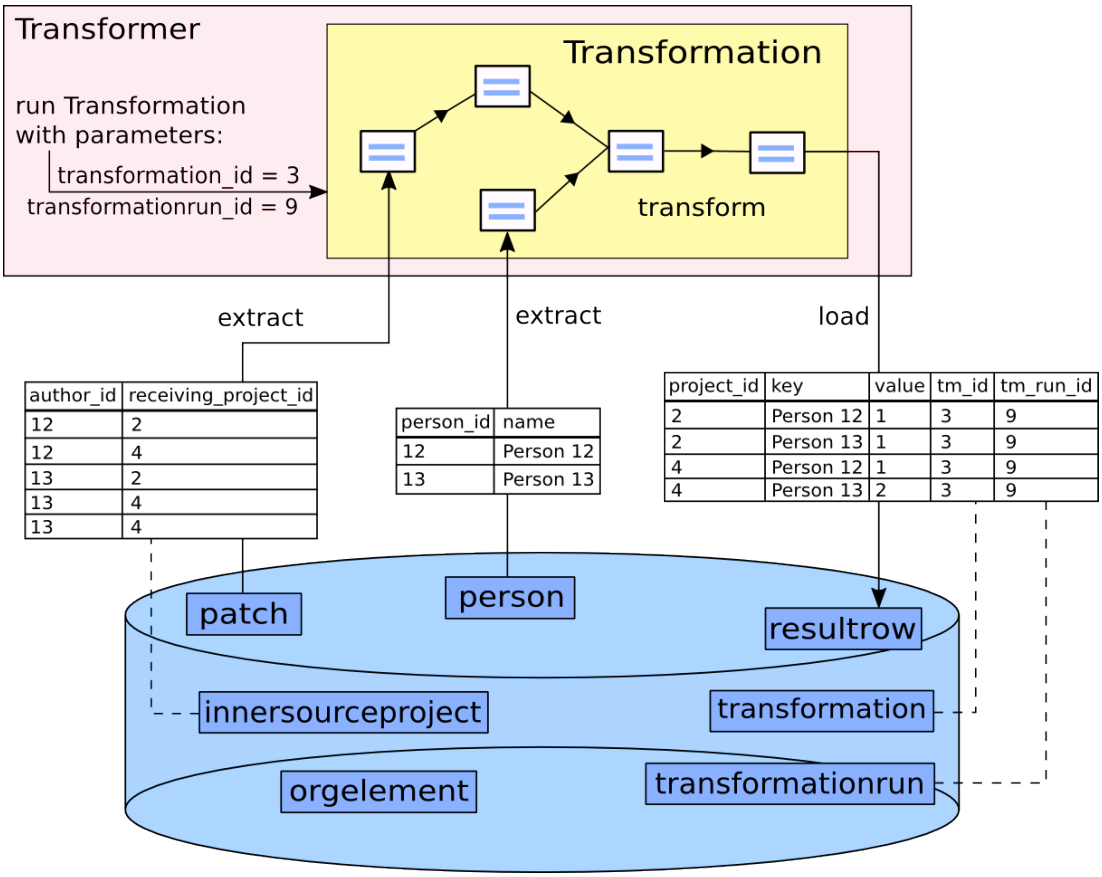


**Figure 2.6:** Running Transformations

The figure shows a sketch of how the transformer module uses Kettle. In the box above, a sketch of how a transformation looks like in Spoon can be seen. It is placed inside the box, labelled "Transformer", to visualize, that the transformation runs from the transformer module. Below, entries of relevant tables for an example transformation are shown. At the bottom the table names can be seen. The lines with arrows, show the flow of data rows through the transformation.

The figure shows an example transformation, which does the following for every inner source project. It reads the number of patches that were contributed to the project by each person. It can be seen that rows are extracted from the patches and person tables, with only the relevant fields. From the patch table, this is the id of the author, who committed the patch along with the id of the project, that received the patch. From the person table the id with the corresponding name of the person are read. Then the rows are transformed, according to the metric definition, defined in the ktr file. In the last step the generated rows are stored into the resultrow table. The input data was altered, so that each result row now contains the project id, to which the result applies. Also it contains a key, which is the author, who committed patches to the project. Corresponding to the author-key, there is the value, which is the number of patches he contributed to the project. In this example transformation, the patches were counted and for the project with the id 2, one patch was counted for both persons. For the other project, one patch was counted for the author with the name "Person 12" and two were counted for "Person 13". Additionally the transformations id and the id of the current run are stored. This is important, so the result can later be related to the transformation and the run in which it was produced so it is clear, which results are the latest ones. The figure shows that the transformation is called with these two parameters. Currently these ids have to be manually mapped to the result rows in the transformation, so it is the task of the metric designer to ensure this. The result rows in the example represent a CategorizedValueResult, with the categories "Person 12" and "Person 13" and their values, which will be visualized as pie-chart in the web-client. This visualization can be seen in the example chapter (4.2), which presents this example transformation and its result-visualization in more detail. Currently, when a new run is finished, all results of older runs are of no interest anymore, as the new run of all transformations created new results from scratch, without taking older results into account. In the future there could be a mechanism to do this to save time and storage.

### 2.7.2 Analysis-Result Provider

The *analysisresultprovider* module implements the logic for retrieving results from the server. It contains a *service* and a *restservice* module. The classes contained in the module are shown in the class diagram in figure 2.7.

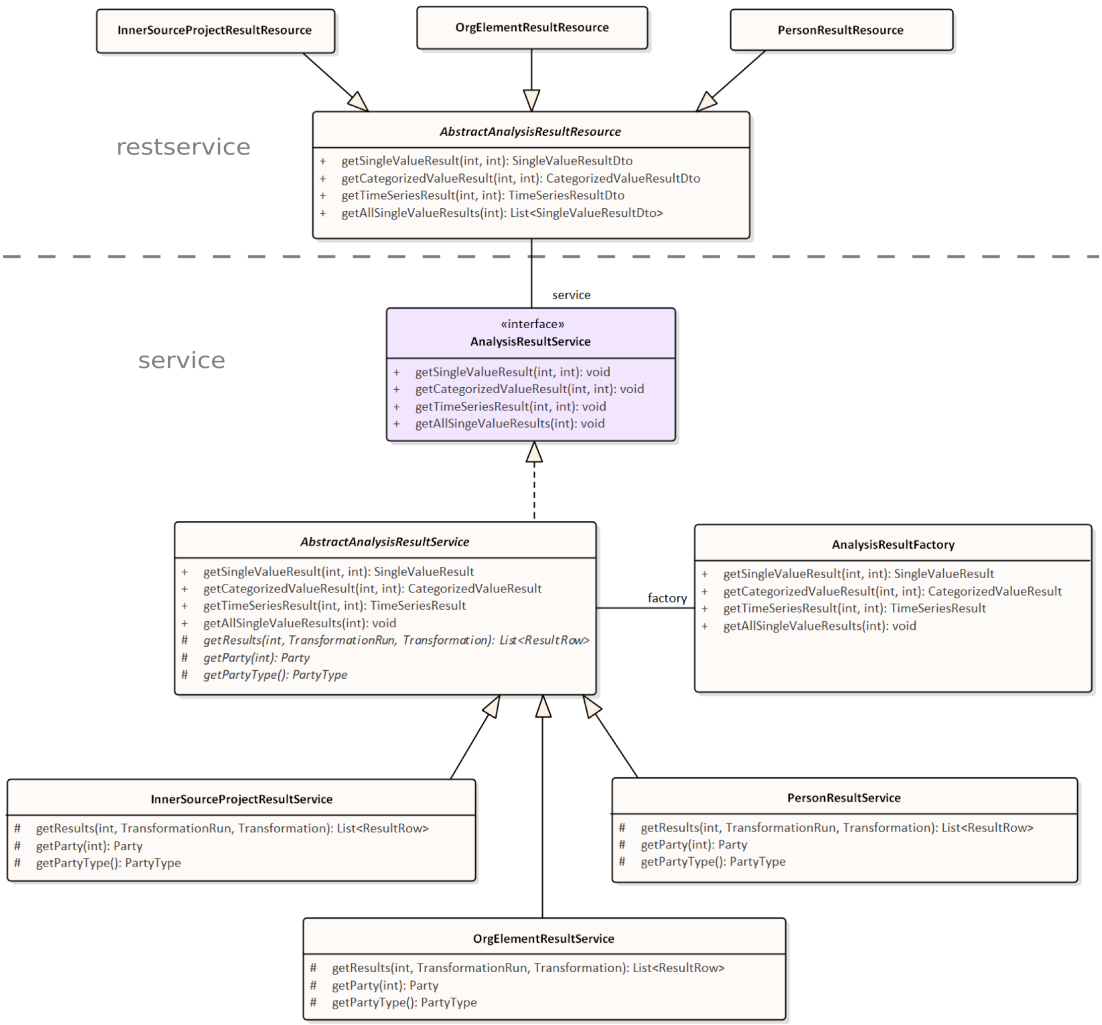


**Figure 2.7:** class diagram of the analysisresultprovider module

The service module provides methods to retrieve the different types of results. It is responsible for converting the data from the resultrow-table into the appropriate result objects. These are either SingleValueResults, CategorizedValueResults or TimeSeriesResults. The getter methods for these results require two parameters, which are the id of the transformation that produced this result and the id of the agent for which the result applies. The TransformationRun is not specified, as the methods will return only the results of the latest run that was successful, as older results are not interesting. The concrete service classes implement the getResults() method, which takes the id of the agent as parameter. So for example the PersonResultService uses the id to search for the id in the person_id column of the resultrow table.

The restservice module defines the corresponding REST API to retrieve results

from the server. Every transformation contains information about which type of result they produce, which can be seen in figure 2.2. Consequently there are methods for retrieving each type: SingleValueResults, CategorizedValueResults, and TimeSeriesResults. Calling the API methods again delegates to the underlying service, which is responsible for the actual execution of the tasks.
This time however, there is an abstract resource class, extended by concrete resource classes for every type of agent. The concrete resource classes only define the base URI for the REST-endpoints. The URI contains the id of the requested transformation, as well as the name of the requested type of agent. For persons this is *"/transformations/{tmId}/results/persons"*. The abstract resource class provides the methods and defines the remaining part of the URI for each method. For example the getSingleValueResult() method adds the URI *"{agentId}/singlevalue"* to the base URI. So if the server requests the SingleValueResult for the person with the id "agentId", that was produced by the transformation with the id "tmId", the server has to issue a GET request to the following URI:
*"/transformations/{tmId}/results/persons/{agentId}/singlevalue"*.
It contains all parameters the service needs to retrieve the result.

## 2.8 Webclient

CMSuites web-client uses Node.js[7] with its package manager npm. It is based on the front-end web-framework Angular 2, which supports the use of Typescript, a typed superset of JavaScript that complies into plain JavaScript ("Typescript Homepage", 2017). Typescript allows for class-based object-oriented programming and adds a type system and support for generics.

The relevant parts for the dashboards functionality are contained in three Angular 2 modules (see figure 2.4). These are the so called *transformationmanager*-, *dashboard*-, and the *analysisresult*-modules.

The transformationmanager- and analysisresult-module each contain a service. Through dependency injection, Angular provides the components of the modules with their corresponding services.
The services methods correspond to the ones of the services on the server.

[7] https://nodejs.org

### 2.8.1 Transformation-Manager-Module

The transformationmanager-module is responsible for uploading transformations, as well as kicking them off. The module contains components for the following:

- *TransformationListComponent*: Shows a list of all currently uploaded transformations
- *TransformationCreationComponent*: Used for uploading a new transformation. A form element is used, which lets the user enter a name for the transformation and the ktr file exported from Kettle
- *TransformationRunComponent*: Shows a list of all runs of transformations, including their status and date

The components use the services, to manage the transformations and transformation-runs.

### 2.8.2 Analysis-Result-Module

The analysisresult-module contains these components:

- *SingleValueResultComponent*
- *CategorizedValueResultComponent*
- *TimeSeriesResultComponent*

The components contain the logic to retrieve the results from the service and the logic to visualize the results using Chart.js. This visualization is then shown inside their corresponding tile of the dashboard.

### 2.8.3 Dashboard-Module

The dashboard-module's responsibility is to present the user with the results for a given agent. On the left side of the page in the web-client there is a tree menu, which contains all organizational units, which can be expanded to show the full hierarchy. Along with the organizational units that are part of a parent unit, all persons and inner-source-projects, that belong to the unit, are shown.
Clicking on one of these parties, creates the dashboard page, which is shown next to the tree-menu. It contains visualizations of all results for all transformations that have been run.

The dashboard module contains components for showing said dashboard for either persons, inner source projects or organizational units:

- *ProjectDashboardComponent*: Shows a dashboard for an inner source project
- *OrgElementDashboardComponent*: Shows a dashboard for an organizational unit
- *PersonDashboardComponent*: Shows a dashboard for a person
- *TileGridComponent*: Used by all the above to display the results in a grid of tiles
- *SingleValueResultTileComponent*
- *CategorizedValueResultTileComponent*
- *TimeSeriesResultTileComponent*

The tile-components are used in the tile-grid of the dashboard and define the layout of the dashboard-tile for their type of result.

# 3 Implementation

In the following chapters, the technologies and design patterns used for the project are explained, that are common to all modules.

## 3.1 REST-Services

The REST-services use Jersey[1], which is a REST framework that provides a JAX-RS Reference Implementation and more. It is part of the glassfish project. JAX-RS (Java API for RESTful Web Services) is an API that provides support in creating web services according to the REST architectural pattern. JAX-RS uses annotations to simplify the definition of REST endpoints. This means that no code has to be written manually to define the url under which a method will be reached. It is simply defined in the annotation, as well as to which HTTP method it should respond (eg. POST). Furthermore it can automatically handle exceptions thrown in the code. In the code all uncatched exceptions are eventually catched by Jersey and and Jersey sends a response to the client indicating that there was an error and continues to listen for incoming requests. In the code exceptions are thrown eg. when assertions fail.

When data is transferred from server to client, so called Data Transfer Objects (DTO) (Fowler, 2008) are used, if it is not reasonable to send the domain object itself. This is the case, if the domain object contains a lot of references to other domain object. The DTO bundles only the data, that the client needs in a simple POJO. This allows for easier serialization and reduces network load by leaving out fields of the domain objects, that are not needed in the client.

[1] https://jersey.github.io

## 3.2 Services

In order to persist the domain objects, the Object/Relational Mapping (ORM) framework Hibernate[2] is used. The domain classes use JPA (Java Persistence API) annotations. Hibernate contains an implementation of the JPA specification along with its own additional API, so it knows how to map the object to the database.
To persist the domain objects so called Data Access Objects (DAOs) (O'Neil, 2008) are used. DAOs encapsulate all access to the data source (Oracle, 2017), which is a database in this case. This design pattern hides all implementation details about how to access the data from the database from the service layer. If the database changes, eg. tables are altered, only the DAO has to be adjusted and no changes to the service layer has to be made.

## 3.3 Pentaho Steps

As mentioned in 2.2.1.4, one can define Pentaho steps. This was not done in this theses, but will surely be of interest in the future. As a temporary solution to aid the metric designer in writing transformation, some template transformations were added to CMSuite, which should be used when writing a new one.

First, there is the ktr file called Template.ktr, which serves as starting point for writing new transformations. It is not a working transformation by itself, but contains the settings, needed for compatibility with CMSuite, as well as useful predefined steps.
The file defines the CMSUITE_TRANSFORMATIONRUN_ID and CMSUITE_TRANSFORMATION_ID parameters. This has to be done, as executing the transformation with parameters, that are not defined inside the transformation, results in an error. Furthermore, it defines a connection named "Default", which is necessary, as mentioned before.

[2] www.hibernate.org

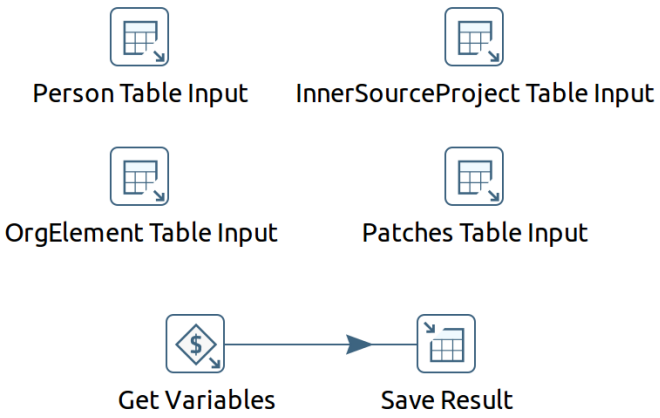


**Figure 3.1:** template steps

Also several input steps are defined, which read data from the tables of CMSuite and they are configured to use the "Default" connection. There is each one step to read rows from the patches-, orgelement-, innersourceproject- and person-tables. These can be used and altered if needed when creating new transformations. Moreover, the template contains the last two steps, that every transformation needs to have in order to save its results. The "Get Variables"-step appends the two parameters, the ids of the current run and transformation, to every row it receives. This does never need to be changed. The "Save-Result"-step then maps the rows to rows in the resultrow-table. In other words it defines the mapping of the fields of the transformations rows to columns in the resultrow-tables rows. The metric designer of course has to change the mapping for his current transformation, but the mapping of the ids is already predefined in the step.
So when creating a new transformation, this template should be opened in Spoon and altered rather than creating a whole new transformation, which would require to define the steps and settings manually.

Organizational units are connected via the orglink table, which defines a parent and a child unit per row. This results in an hierarchy of organizational units. Via the innersourceprojectlink and personlink tables, inner source projects and persons are attached to the units.
When writing transformations for OrgElements, is is not easy to relate persons and projects to units, that are not directly linked to them but are further away in the link-chain. A temporary solution is to use the step defined in the Template_GetAllOrgElementLinks.ktr file. The step creates result rows with two columns: parent_orgelement_id and child_orgelement_id. It creates a row for every parent-child link, also taking into account children, grand-children, grand-grand-children, etc. So in the end, for every node in the orgElement tree, the result contains a row for all its descendands in the tree.

So this can be used as input step for transformations rather than using a row for each OrgElement. Using this, if we look at a parent, we have all paths down the hierarchy as a row for this parent. Now if a child is linked to a person, or inner source project, the information to which parent, grand-parent, grand-grand-parent, etc. it belongs is also available to the transformation.

## 3.4 Transformation Manager

The transformationmanager component is responsible for loading and saving Transformations, as well as kicking off transformation runs, as mentioned in the design chapter 2.8.1.

### 3.4.1 REST Service

The restservice component defines the REST-API, which defines the set of functions which can be used to communicate with the service. There are two resource classes, one for Transformations and one for TransformationRuns. The TransformationResource provides the following methods:

```
@GET
@Path("/")
@Produces({ MediaType.APPLICATION_JSON })
public List<Transformation> getAll(@QueryParam("agenttype")
    EconomicAgentType _agentType) { ... }

@GET
@Path("/{id}")
@Produces({ MediaType.APPLICATION_JSON })
public Transformation get(@PathParam("id") int _id) { ... }

@DELETE
@Path("/{id}")
@Produces({ MediaType.APPLICATION_JSON })
public Transformation delete(@PathParam("id") int _id) { ... }

@POST
@Path("/")
@Consumes({ MediaType.MULTIPART_FORM_DATA })
@Produces({ MediaType.APPLICATION_JSON })
public Transformation saveTransformationFromForm(FormDataMultiPart
    _form) { ... }
```

Here the JAX-RS annotations can be seen. Above each method, there is an annotation, that defines the HTTP method it should respond to. Then there is

the @Path annotation, which defines the URI, which the client has to request in order for the method to be executed. The @Produces annotation defines, that all results will be serialized to JSON format. Finally, @Consumes defines the data format, that the method expects to receive from requests sending data, like POST requests. The @PathParam annotation is used to map parameters, that are part of the URI to parameters in the Java code.

There are two methods that respond to a HTTP GET request. The getAll(EconomicAgentType _agentType) method returns a list of all transformations for the given type of agent. If _agentType is null, all transformations are returned. The getAll(int _id) methods returns the transformation with the id=_id.
The delete() method responds to a HTTP DELETE request and deletes the transformation with the id=_id
The saveTransformationFromForm() method saves a transformation. It receives a FormDataMultiPart object, which represents the form that was filled by the user in the web-client. It contains all information to create a new transformation entry in the database. Also it contains the ktr file as InputStream. As this is an non-idempotend operation, which creates a new resource, it should use the HTTP POST method (Allamaraju, 2010).

All methods return the result in JSON format, which can be easily converted into the Typescript equivalent of the results. So for example a Transformation object is sent in JSON format to the webclient. In the web-client it is simply converted into a Transformation object. Of course the Transformation class in Typescript is the same as the Transformation class in JAVA in order to make this possible.

The TransformationRunResource provides the following methods:

```
@GET
@Path("/")
@Produces({ MediaType.APPLICATION_JSON })
public List<TransformationRun> getAll() { ... }

@GET
@Path("/{id}")
@Produces({ MediaType.APPLICATION_JSON })
public TransformationRun get(@PathParam("id") int _id) { ... }

@POST
@Path("/")
public Response kickOff() { ... }
```

The getAll() method returns all, the get() method returns only one TransformationRun object with the id=_id.
The kickoff() method is special, as it kicks off a run of all transformations and thus starts a long-running task. As suggested in Allamaraju, 2010 POST is used

to create the resource and response with a status code 202 (Accepted), indicating the TransformationRun has been created. It is then used to track the status. The web-client has to manually ask for the TransformationRun and read its status. If a subsequent POST follows, it will be detected in the service and simply be discarded and a message with 409 (Conflict) status is sent.

### 3.4.2 Service

The service component represents the service layer, which is responsible for executing the tasks.

For Transformations, it uses the corresponding DAOs to retrieve, create or delete them from the database using Hibernate. When a new Transformation should be created, the services save() method checks if the information provided through the form in the REST-servcie is valid. If it is, a new entry is created in the database, and the InputStream coming from the REST-service is written into a file. The filename is simply the id of the transformation, that was automatically created by hibernate when storing it into the database. This avoids duplicate filenames. The delete() method deletes the database entry and the file again.

Kicking off a run creates a new TransformationRun object, which is persisted in the database, using the DAO for TransformationRuns. Its status is set to INITIALIZED to indicate the run has started. The status is used so that no other run can be started, while there is already an initialized (= running) one. After this, the run is started in a new thread and the object is returned to the web-client, so it knows the execution was actually started. The run executes all known transformations, which is done by the transformer component. If a transformation fails, the status is set to FAILED. This way when loading all runs, there is the information that the last run failed. Upon successful completion, the status is set to FINISHED. If there are any further kickoff requests, the service will throw an exception, which will be catched in the REST-layer, which will response with an HTTP error message (409 Conflict), as mentioned above. Only after the execution is finished (the TransformationRun's state is either FINISHED or FAILED), a new request will be accepted.

### 3.4.3 Transformer

Execution is done by the transformer component. It contains the KettleTransformer class, which is the bridge between CMSuite and the Kettle engine.

```
public void transform(InputStream _ktrAsStream,
    Transformation _transformation, TransformationRun
    _transformationRun) {
  try {
    KettleEnvironment.init();
    executeTransformation(_ktrAsStream, _transformation,
    _transformationRun);
  } catch (KettleException e) {
    throw new TransformerException(e);
  }
}

private void executeTransformation(InputStream _ktrAsStream,
    Transformation _transformation, TransformationRun
    _transformationRun)
    throws KettleException {

  TransMeta transMeta = new TransMeta(_ktrAsStream, null, false,
    null, null);

  DatabaseMeta dbMeta = ...

  // replaces the connection defined in the ktr that has the same
    name
  transMeta.addOrReplaceDatabase(dbMeta);

  transMeta.setParameterValue("CMSUITE_TRANSFORMATION_ID", "" +
    _transformation.getId());
  transMeta.setParameterValue("CMSUITE_TRANSFORMATIONRUN_ID", "" +
    _transformationRun.getId());

  Trans trans = new Trans(transMeta);
  trans.execute(null);
  trans.waitUntilFinished();

  if (trans.getErrors() > 0) {
    throw new TransformerException("More than zero errors");
  }

}
```

The executeTransformation() method is called for every transformation. It creates a TransMeta object from the InputSteam, that comes from the opened transformation file. This object will then be used by the Kettle engine to run the transformation. After that, a DatabaseMeta object is created. Here all database

parameters, such as the database-type, host, database-name, port, username and password are set. Also a name is set to identify this connection. In the code this connection is always named "Default". The addOrReplaceDatabase() method then replaces the connection named "Default" that is already defined in the transformation file, with the one that is defined in the code. So when running the transformation from CMSuite, the connection that is defined in CMSuite's configuration file will be used.
As previously shown in chapter 2.7.1.2, in order for CMSuite to know, which results of the transformation belong to which transformation as well as to which run, the transformation needs some parameters when started. This happens in the setParameterValue method. Here the id of the TransformationRun, which is running at the moment, is set as parameter. Also the current Transformation's id is set. The parameters are used by the transformation to store its results along with the parameters into the database.
After setting up the meta-data, the transformation is executed. If an error occurs during execution, an exception is thrown. It will be catched in the service, which sets the current TransformationRuns status to FAILED, and cleans up any results, that were already produced by the failed run. On success, the status is set to FINISHED.

For the connection between CMSuite and Kettle to work properly the metric designer currently has to take care of the following two points. Firstly, when creating transformations in Spoon, he has to use a database connection with the name "Default". If a database connection with another name is used for the transformation, CMSuite would create the "Default"-connection additionally, but it would not be used within the transformation. So the wrong connection may be used and changes to CMSuites database-connection would not be updated for the transformation.
Secondly the metric designer also has to make sure the parameters, which are the ids, are properly assigned to every result row, the transformation produces. Otherwise there is no way to know to which transformation or run the row belongs.

## 3.5 Analysis-Result-Provider

### 3.5.1 REST Service

The REST service component defines the REST-API, which defines the set of functions which can be used to communicate with the service. For the analysisresultprovider, there is one abstract resource class, which is extended by three subclasses.

The concrete resource classes define the base REST URI for results for either OrgElements, InnerSourceProjects, or Persons (so all classes, that implement the EconomicAgent interface).

```
@Path("/transformations/{tmId}/results/orgelements/")
public class OrgElementResultResource extends
    AbstractAnalysisResultResource { ... }

@Path("/transformations/{tmId}/results/innersourceprojects")
public class InnerSourceProjectResultResource extends
    AbstractAnalysisResultResource { ... }

@Path("/transformations/{tmId}/results/persons/")
public class PersonResultResource extends
    AbstractAnalysisResultResource { ... }
```

The abstract resource class *AbstractAnalysisResultResource* defines the methods, that each resource provides:

```
public abstract class AbstractAnalysisResultResource {

  ...

  @GET
  @Path("{agentId}/singlevalue")
  @Produces({ MediaType.APPLICATION_JSON })
  public SingleValueResultDto getSingleValueResult(
    @PathParam("tmId") int _tmId,
    @PathParam("agentId") int _agentId) { ... }
  }

  @GET
  @Path("{agentId}/categorizedvalue")
  @Produces({ MediaType.APPLICATION_JSON })
  public CategorizedValueResultDto getCategorizedValueResult(
    @PathParam("tmId") int _tmId,
    @PathParam("agentId") int _agentId) { ... }


```

```
22    @GET
      @Path("{agentId}/timeseries")
24    @Produces({ MediaType.APPLICATION_JSON })
      public TimeSeriesResultDto getTimeSeriesResult(
26      @PathParam("tmId") int _tmId,
        @PathParam("agentId") int _agentId) { ... }
28    }

30    @GET
      @Path("/singlevalue")
32    @Produces({ MediaType.APPLICATION_JSON })
      public List<SingleValueResultDto> getAllSingleValueResults(
34      @PathParam("tmId") int _tmId) { ... }

36  }
```

The JAX-RS Path annotations above the concrete classes, are interpreted as the first part of the URI. The annotations above the methods, defined in the abstract class, contain the second part. It is appended to the first part, which results in the complete URIs described in chapter 2.6.

The resources provide the endpoints to retrieve the different types of results. So there is one endpoint for SingleValueResults, CategorizedValueResults and TimeSeriesResults. All of them take the ids of the requested transformation and agent as parameter, which is read from the URI. Additionally there is the getAllSingleValueResults() method which retrieves the SingleValueResults of all parties for the given transformation. The results are converted to Data Transfer Objects (DTO) before retrieving, as they do not need all information about the agent.

### 3.5.2 Service

The service layer also contains one abstract base class, *AbstractAnalysisResultService*, which is extended by three subclasses for each type of agent (see figure 2.7). This is needed, so the concrete classes know which type of agent they need to request, when retrieving results.
The AbstractAnalysisResultService provides the corresponding methods for each of the REST endpoints. It checks if the requested result-type matches the result-type of the transformation and if the requested type of agent matches the transformation's type of agent. If not, an IllegalStateException is thrown. If everything is OK, the service extracts the result rows of the requested agent and transformation for the latest run, that was successful. The rows are then converted to the requested result format.

The converted formats extend *AnalysisResult*. The transformation and the agent

that were requested are stored in it (see figure 2.2).
For results with only a single value, thus only a single result-row, the row's value column is stored as value field in a new *SingleValueResult* object.
*CategorizedValueResults* are created by storing key and value of each row of the resultrow- table as an entry in its Map. The key and value of a row are simply stored as key and value in the map.
*TimeSeriesResults* are created by storing each key and value of a row of the resultrow- table as an entry in its array. The key has to be a date in the format yyyy-MM-dd.

## 3.6 Web-client

The *CategorizedValueResultComponent* is responsible for retrieving *CategorizedValueResults* from the service and for visualizing them. For this Chart.js is used. The following Typescript code snippet shows the conversion of the retrieved result values to a format, that is accepted by Chart.js:

```
result: CategorizedValueResult = null;

chartLabels: string[] = [];
chartData: number[] = [];
chartType: string = 'pie';

private convertData(): void {
  let data = this.result.values;
  let chartLabels: string[] = [];
  let chartData: number[] = [];
  Object.keys(data).forEach(function(key) {
    chartLabels.push(key);
    chartData.push(data[key]);
  });
  this.chartLabels = chartLabels;
  this.chartData = chartData;
}
```

The chart accepts the keys and values as two separate arrays. One for the keys and one for the values. The convertData() method first reads the values from the CategorizedValueResult, which was retrieved when the module was initialized. That is when the user clicked on an agent in the tree-menu. Afterwards, for every entry in the values map, the key and value are stored in the separate arrays. In the end the arrays are assigned to to the classes chartLabel and chartData fields and the pie chart is initialized. The chart type can simply be changed by choosing another chart type than "pie". A doughnut chart would also work for this type of input data.

The next snippet shows how it works for *TimeSeriesResults* in the *TimeSeriesResultComponent*:

```
result: TimeSeriesResult = null;

chartData: {x: Date, y: number}[] = [];

chartType: string = 'scatter';

private convertData(): void {
  let data = this.result.values;
  let chartData: {x: Date, y: number}[] = [];
  for (let i = 0; i < this.result.values.length; i++) {
      chartData.push({
        x: data[i].date,
        y: data[i].value
      });
  }
  this.chartData = chartData;
}
```

Here, a scatter chart is used, which accepts only an array of data-points (with x- and y-values). Scatter-charts can make use of dates in contrast to line-charts in Chart.js. The convertData() method reads the date and the corresponding values from the retrieved TimeSeriesResult. The date is stored as x-value and the value is stored as y-value. Using the scatter-chart, the data-points are not necessarily evenly distributed along the x-axis, as it would be the case for line-charts. If the date-intervals of sequential data-points vary, this variations can be seen on the x-axis. Furthermore the date is automatically converted to a format, that fits best for the x-axis. For example, only the years are shown as x-axis labels even if there are 1000 data-points for each day, as labelling each day would clutter the x-axis.

# 4 Examples

## 4.1 Single-Value-Result Example

For every organizational unit, this transformation does the following: It reads the number of projects, that are hosted by it. Not only projects directly hosted are counted. For units, that are higher up in the hierarchy, all projects that are hosted by themselves or their children are counted. In the end, the result contains one single value, which is the number of hosted inner source projects.

How this transformation looks like in Spoon can be seen in figure 4.2.

First, the step from the template, that generates a row for each parent-child relation is used (see 3.3). Also, a row for each OrgElement is added with a parent- and child-reference pointing to itself.

So the rows now have this format:

| parent_orgelement_id | child_orgelement_id |
|---|---|

Then the output is sorted by child_orgelement_id. Sorting is required by Kettle or otherwise the "Merge-Join"-step may not produce the correct result. In this step, the links between OrgElements and InnerSourceProjects are read from the projectlink table and joined with the child_orgelement_ids.

So now the rows contain these fields:

| parent_orgelement_id | child_orgelement_id | project_id (linked to the child) |
|---|---|---|

The join operation is configured to be an inner-join. Consequently, rows referencing children, that do not have a project attached to them, will be dismissed. Then it is sorted by the parent_orgelement_id. Again this is needed for the next step to work, which counts the rows per parent. So it produces rows containing the parent_orgelement_id and the number of projects for this OrgElement. Lastly the parameters are added to each row and each row is mapped to a row in the resultrow-table, using the two final steps from the template transformation. The mapping of the fields to the table columns is the following:

| | |
|---|---|
| TRANSFORMATIONRUN_ID | → tm_run_id |
| TRANSFORMATION_ID | → tm_id |
| parent_orgelement_id | → orgelement_id |
| nr_of_projects_hosted | → value |

As its a transformation for organizational units, the unit it mapped to the orgelement_id column of the resultrow-table, so it is clear to which OrgElement the result belongs. The number of projects, that are hosted by it, is stored in the value column.

The complex part about this transformation is dealing with the hierarchy of organizational units. If they are just read from the table and merged with their assigned projects, the result would only contain the number of projects that a unit directly hosts. OrgElements, that are higher up in the hierarchy may have no projects assigned but only child-orgelements. They would have no result then. This is solved with the step from the template.

In the web-client, the result is visualized as a single value:

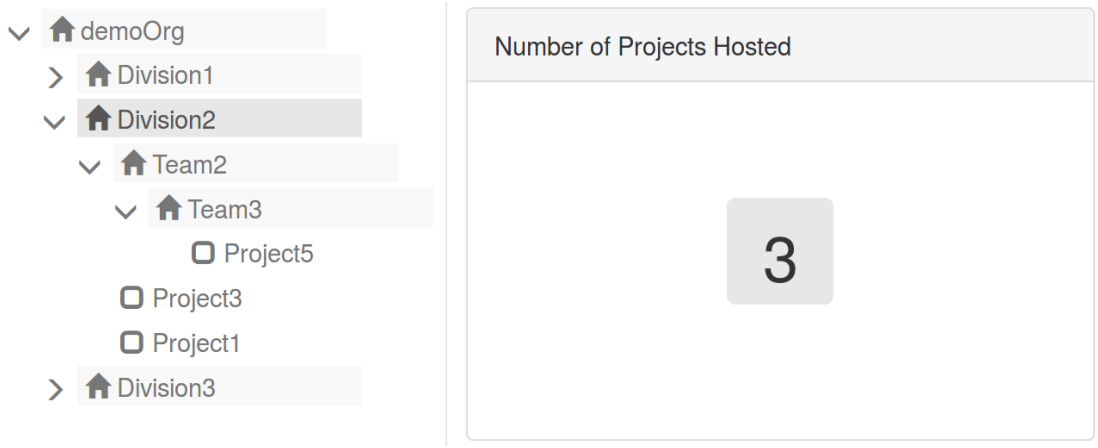


**Figure 4.1:** visualization of the result for "Division2" in the web-client

The result for the organizational unit with the name "Division2" is shown in the figure. The division has a total of three hosted projects, which can also be seen in the tree-menu on the left side. Although not all projects are directly connected to the unit, all projects are counted.

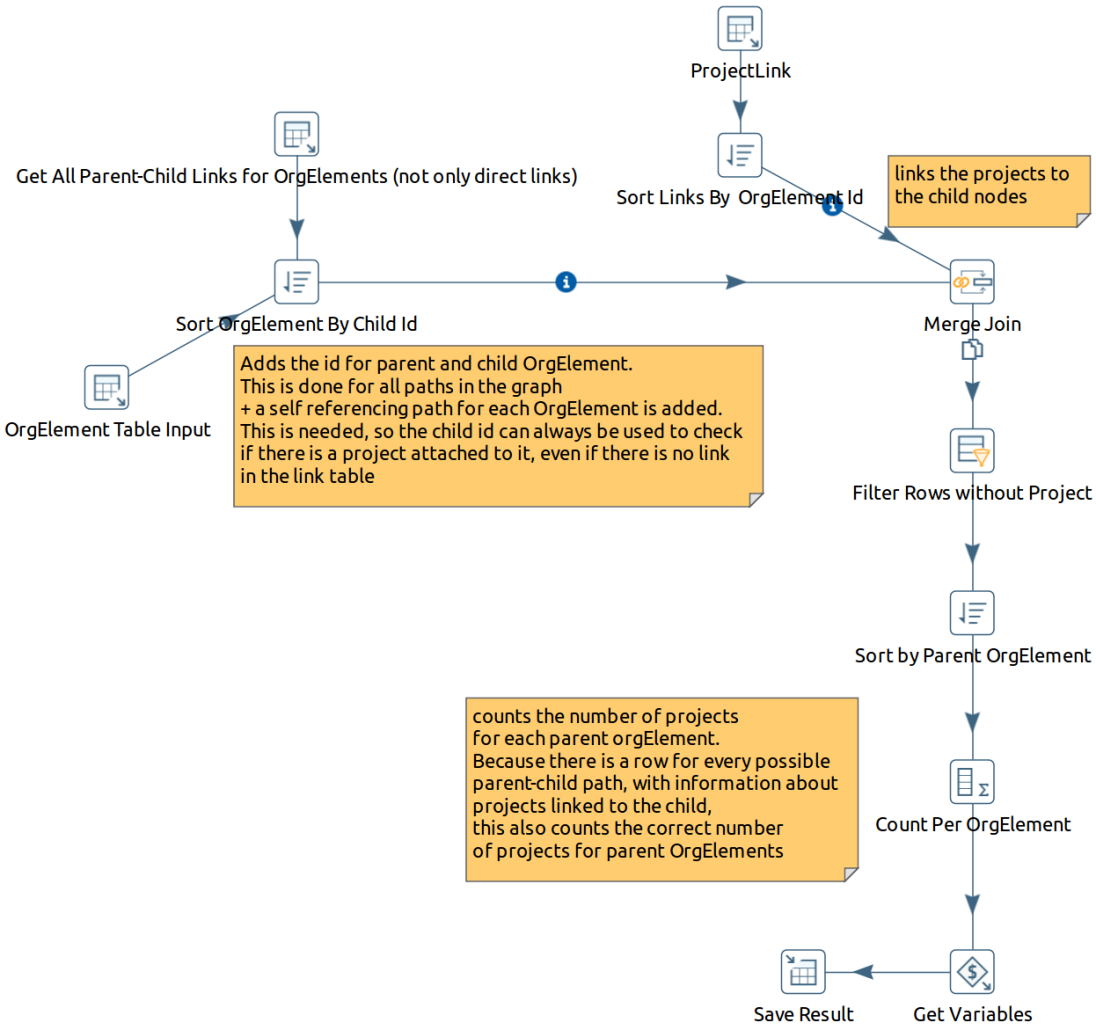


**Figure 4.2:** transformation: number of inner source projecst hosted

## 4.2 Categorized-Value-Result Example

In the following, a transformation for inner source projects is presented, which calculates the number of patches, that were contributed to the projects, by each person. The categorized-value-results contain multiple categories with a value for each of them.
In this example, the categories are the names of the persons that contributed patches. The value is the number of patches, a person contributed.

In Spoon the transformation that achieves this looks like this:

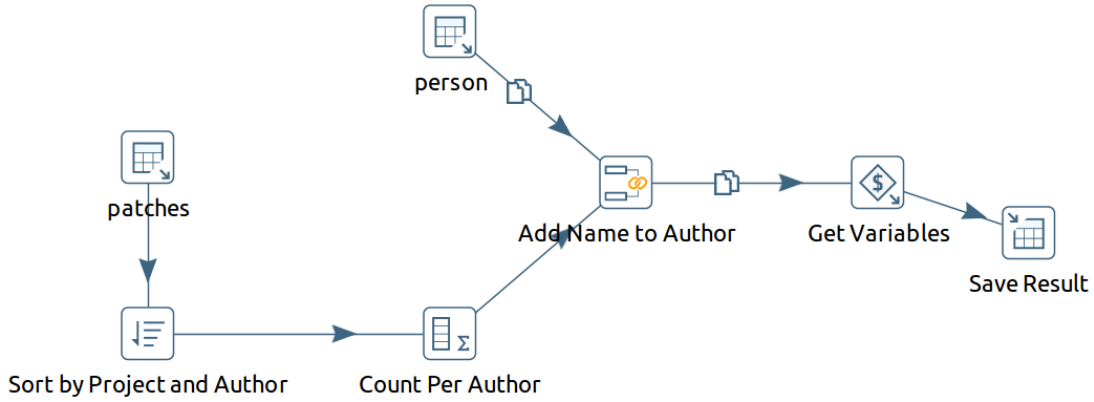


**Figure 4.3:** transformation: number of patches contributed by each person

In the "patches"-step, all rows of the patches table are extracted, containing the id of the receiving project and the id of the author. Then the output is sorted. First by the project's ids and then, for each project-id, it is sorted by the ids of the authors. This is again a requirement of Kettle for the next step to work properly. The next step aggregates the result by project and author and counts the rows. So it generates rows with a new field, which is the number of rows of the input, with the same combination of project and author id. This field is the number of patches.
So now the rows contain the following fields:

| receiving_innersourceproject_id | author_id | nr_of_patches_contributed |
|---|---|---|

The result is almost complete, but the names of the authors are missing. So in the next step, for every row, the name of the person that belongs to the author_id, is read from the person table.
Again, in the "Get Variables" step, the parameters are added to each row. In the last step each row is mapped to a row in the resultrow table. The mapping of the fields to columns is the following:

| | |
|---|---|
| TRANSFORMATIONRUN_ID | → tm_run_id |
| TRANSFORMATION_ID | → tm_id |
| receiving_innersourceproject_id | → innersourceproject_id |
| person_full_name | → key |
| nr_of_patches_contributed | → value |

The parameters are mapped as usual. The project, that received the patches is mapped to the innersourceproject_id column of the table, so it is clear to which project the result belongs. The name of the author who contributed is mapped to the key column. This column is used as the category for the visualization. Lastly, the number of the author's patches is mapped as value.

In the web-client, the the result is visualized as pie-chart:

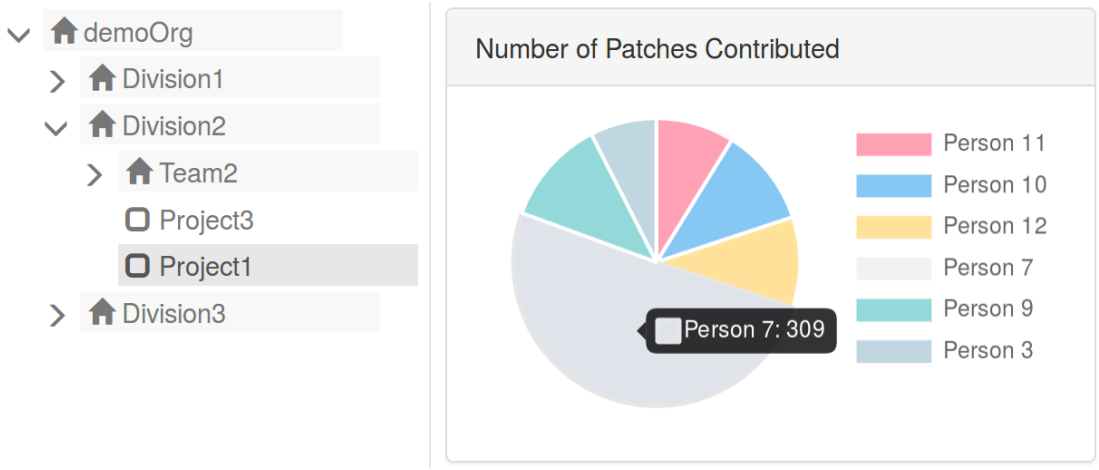


**Figure 4.4:** visualization of the result for "Project1" in the web-client

The figure shows the result for the inner source project with the name "Project1". The pie-chart provides a quick overview about the ratio of contributed patches among the authors, that contributed to the project. Hovering over a slice, shows the number of patches of the person. In the example, the person with the name "Person 7" has contributed 309 patches in total.

## 4.3 Time-Series-Result Example

This example transformation calculates results for persons. It reads the number of patches that were contributed by each person per year. So one result contains multiple date-keys, with a value for each of them.
In this example, the value for each year is the number of patches that a person contributed during this year.

In Spoon this transformation looks like this:

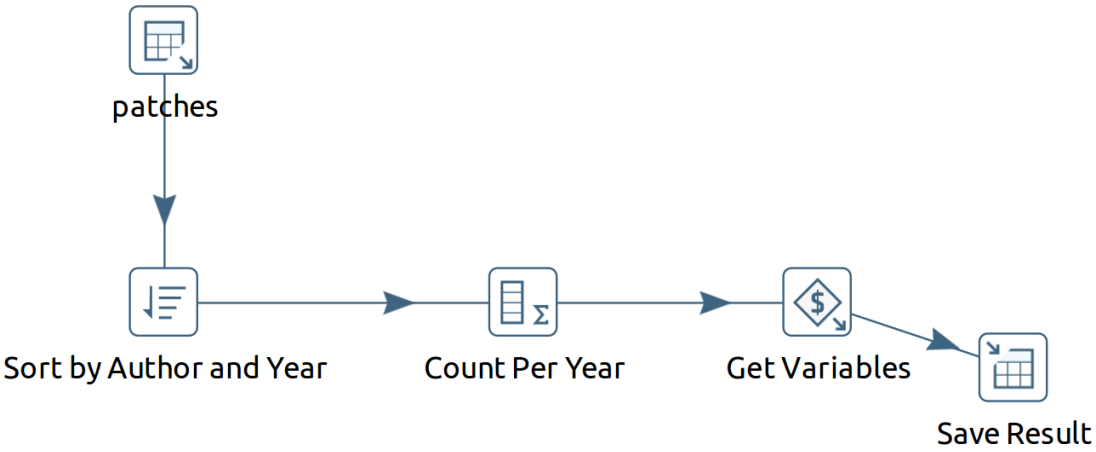


**Figure 4.5:** transformation: number of patches contributed per year

In the first step, all rows of the patches table are extracted. Their fields are the id of the author and the year portion of the date, when the patch was committed. Then the output is sorted. First by the author's id and then for each author-id it is sorted by the year in order for the next step to work. The next step aggregates the result by author and year and counts the rows containing the same year and author. This creates a new field containing the number of patches of the author for each year.
So now the rows now have this format:

| author_id | commit_year | nr_of_patches_contributed |
|---|---|---|

Lastly, the parameters are added to each row and each row is mapped to a row in the resultrow table. The mapping of the fields to the table columns is the following:

| | |
|---|---|
| TRANSFORMATIONRUN_ID | → tm_run_id |
| TRANSFORMATION_ID | → tm_id |
| author_id | → person_id |
| commit_year | → key |
| nr_of_patches_contributed | → value |

As its a transformation for persons, the author who committed the patches is mapped to the person_id column of the table, so it is clear to which person the result belongs. The year in which the author contributed the patches is mapped to the key column and the number of the patches is stored in the value column.

In the web-client, the the result is visualized as line-chart:

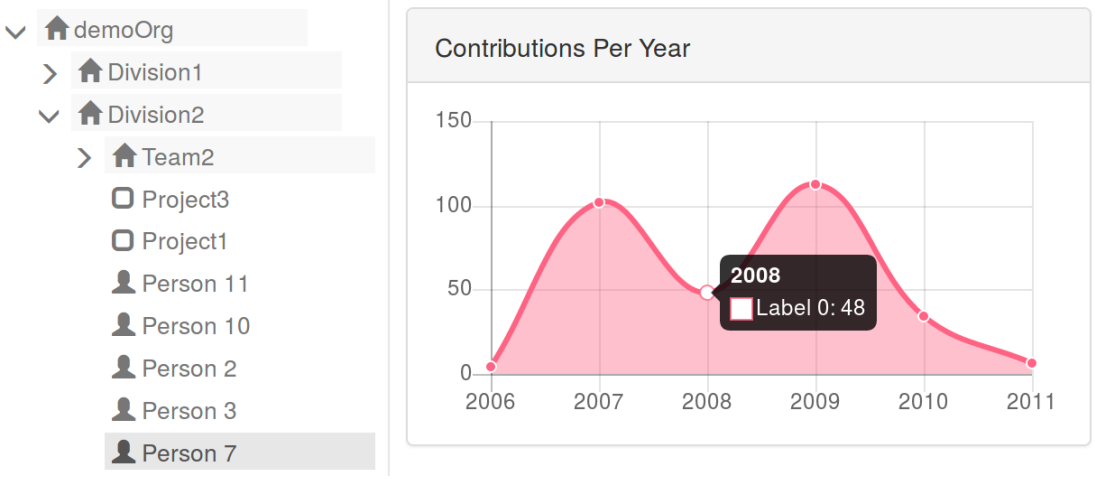


**Figure 4.6:** visualization of the result for "Person 7" in the web-client

In the figure, the result for the person with the name "Person 7" can be seen. The line-chart shows how the amount of contributions of this person changed over the years. Hovering over a data-point, shows the number of patches of the person for the corresponding year. In the example, the person with the name "Person 7", has contributed 48 patches in the year 2008.

# 5 Evaluation

## 5.1 Functional Requirements

**Metrics**

The most important requirement (1) was to make adding of new metrics possible without having to change code. This was achieved by using Pentaho Kettle to define metrics in Kettle's ktr-files, which are then uploaded to the server and executed there. So no change of any code, neither on client- nor server-side is needed to test new metrics.

**Showing List of Metrics**

In the web-client, clicking on the "Transformation"-tab, retrieves all uploaded transformations from the server and shows them in a list, with their names. This fulfils requirement 2.

**Executing Runs**

All transformations can be executed with a single click from the "Transformation Runs"-tab, this sends a single request to the server, on which the execution is then started. So requirement 3 is fully implemented.

**Showing List of Runs**

A list of all runs can be shown, including information about the status of the run and the date when the run was started, in the same tab in the web-client. So requirement 4 is also implemented.

**Showing Visualization of Results**

Requirement 5, being able to show a visualization of results for every available type of agent, is also implemented.

Using Kettle, arbitrary transformations can be defined and results can be stored for every agent of any type, though it is more difficult to write transformations

for organizational units. With the temporary fix from chapter 3.3, however, it is not too hard to do so.

### Choose Visualization

In the current state of the project, it is not possible to choose how exactly the result should be visualized. That is, which colors should be used or which chart-type (that is applicable for the type of result). Currently the settings for the chart are hardcoded in the web-client and every type of result is bound to the same visualization. So requirement 6 is is not implemented.

### Result Types

The next requirement (7) that three types of results should be supported, is met. The three required types are supported in the form of the SingleValueResult, CategorizedValueResult and TimeSeriesResult.

### Supported Metrics

The last requirement demanded that all metrics described there should be supported.

The "Amount of IS projects hosted" metric can be applied to the whole company as well as to each organizational unit. The calculation of this metric is shown in the example chapter (4.1) An it creates results for every organizational unit, so also the top unit, which is the whole company. The "Amount of active IS projects hosted" metric is thus also possible, as the transformation just needs to be extended to decide, which projects are active (eg. by reading the date of the last commit to the project). The visualization is the same.
The other examples calculate metrics, not contained in the requirement. The remaining required metrics are:

### Metrics for the whole company (= the root organizational unit)

1. Percentage of patch-flow between different types of units

   The corresponding transformation can read the number of patches that are committed across different types of units. The hierarchy of the units has to be taken into account. The type of unit is the key, the percentage of patches is the value, so it will be visualized as pie-chart.

### Metrics for organizational units

1. External contributions received

   The number of commits has to be counted, that the organizational unit received but were not authored by persons of the same unit. The hierarchy of the units has to be taken into account, which makes the transformation a bit more difficult. The result is a single value.

2. External contributions contributed

   The number of commits has to be counted, that were authored by persons of the organizational but not received by same unit. The hierarchy of the units has to be taken into account. The result is a single value.

**Metrics for IS projects**

1. Development activity (amount of patches)

   Counts the number of patches, contributed to the project. The result is a single value.
   Example 4.2 shows a related transformation, but it also splits the number of patches per person.

2. Development activity by contributing units (by type of unit)

   Reads the number of patches, the project received, and counts them per type of unit. The hierarchy of the units has to be taken into account. So the result contains a key, which is the type of unit, and a value which is the number of patches that all units of the type contributed to the project. The result is thus visualized as pie-chart.

3. Amount of contributing units (by type of unit)

   Counts all unique occurrences of patches (contributed to the project) for each type of unit. The result is visualized as a single value.

4. Amount of contributing users

   Counts all unique occurrences of patches (contributed to the project) for each user. The result is a single value.

**Metrics for persons**

1. Number of IS projects contributed to

   Reads the number of unique occurrences of patches (the person authored) for each project. The result is a single value.

2. Amount/percentage of contributions to foreign units (by type of unit)

   The number of commits has to be counted, that were authored by the person but not committed to the person's organizational unit. The hierarchy of the units has to be taken into account to also check if the patch was committed to an external unit of another type. The result contains categories, which are the type of unit and values, which is the amount of contributions.

All in all, with Kettle, the transformations can be implemented like described above. The main problem is most of the time, that the hierarchical order of organizational units has to be taken into account. But as a temporary solution

to make this easier, the template steps from chapter 3.3 can be used and in the long run, specific Kettle steps, tailored for CMSuite, can be implemented.

## 5.2 Non-Functional Requirements

**Latency**

As stated in non-functional requirement 1, the time it takes to visualize results should be kept minimal. The computation of the results is run on the server as soon as it is kicked off and the results are available as soon as the computation finishes. When the web-client requests results, the results of the latest finished run are immediately retrieved. The latency only comes from sending the data over the network and from Chart.js interpreting the results. The time obviously scales with the number of uploaded transformations (and thus the number of results to display and retrieve). Furthermore, if transformations are defined, that produce a large amount of result rows, it takes more time to visualize them. So in part it is also the responsibility of the metric designer to define reasonable transformations. Not only does it take more time but eg. a pie-chart with 1000 categories will be impossible to read.

**Tool with GUI**

As demanded by non-functional requirement 2, the metric designer does not have to be a programmer. He has to use Kettle with its GUI (Spoon), which does not require programming skill, though some programming experience and database-knowledge may be of help.

**Permissive Licence Tool**

Kettle uses a permissive licence, as required by non-functional requirement 3. So it can be used without restriction.

**Integration into Code**

Kettle provides the possibility to run transformations and jobs from Java, which makes it possible to integrate it into CMSuite, which fulfils non-functional requirement 4. Kettle offers an API, which allows to define anything, that can also be defined via the GUI (Spoon).

**Fault Handling**

If faulty transformations are uploaded, an exception is thrown, originating from the Kettle-engine. It is then catched and execution continues, thus faulty transformations will not cause a crash, which was demanded by requirement 5.

# 6 Future Work

## 6.1 Kettle Steps for CMSuite

It was shown in chapter 2.2.1.4, that own steps for Pentaho Kettle can be defined. This allows for defining new steps, tailored for CMSuite. Of course there are many possible steps that could be implemented.
For example when reading rows from the patch table, they do not contain author or project information other than the id. So often information about these has to be added inside the transformation. There could be an input step, which adds the information to the rows, if needed.
Furthermore, it is complex to deal with the hierarchical order of the organizational units. In chapter 3.3, this problem was described and a temporary solution was presented, using the step from the template. To use this step, it has to be copied into a transformation and is thus just duplicated. The step uses a complex recursive sql-query, which means changes to CMSuite's database layout could possibly break the step and it would have to be fixed in all transformations, that are using it. A cleaner solution is to write a self-defined step with the same functionality (extracting all (recursive) parend-child relations). This way, the step could simply be chosen like any other step from Spoon's menu.
Moreover, there could be a step, which just extracts all persons of an organizational unit, also taking into account all persons that are part of a child of this unit. The same step should also be available for inner source projects of course.
Another useful step could be one, that saves the result without needing to manually add the parameters to each row and map them to the resultrow-table. In other words, a step that replaces the "Get Variables"- and "Save Result"-steps from the template presented in 3.3.

## 6.2 Reusing Previous Results

Currently, if a transformation is executed, it often takes into account all entries of the patch-table. If a new crawl-run is started, which creates new entries in the patch-table, and afterwards a transformation-run is started, it uses all results of the patch-table, regardless of the crawl-run. After a run of transformations, the old results are of no interest anymore. Of course this scales very badly, as more and more entries are added to the patch-table.
The solution is to take old results into account when executing transformations. So the transformation also gets the id of the latest crawl-run as parameter and uses only entries of the patches-table, which are from crawl-runs, that were not used for this transformation yet. Then the result rows also store information about the crawl-run. When the web-client requests a result, the server does not only read the result-rows for the latest run. The row's values are increments instead of absolute values. The increments of all transformation-runs are then added up to get the absolute value, which is the actual result.
This way, the time to run a transformation stays constant, as only new patches are taken into consideration. Ideally, reading only the results from crawl-runs, not used for the transformation yet, should not be the responsibility of the metric designer, which makes this another candidate for a self-defined Kettle-step.

## 6.3 More Types of Results

In chapter 2.4.2, it was described how results are stored in the resultrow-table. The label column is not used by any of the result-types. So why is it there? The column is intended to be able to store another type of result, which could be implemented in the future. This type of result stores not only one set of key-value pairs, but multiple sets of key-value pairs. Each of the sets is assigned with a label and thus the label-column exists. So a transformation, creating this type of result, has to set a label for each set of key-value pairs. An example transformation for organizational units for this type of result would be: "Number of patches contributed by each person of this unit per year" In the web-client, this would then be displayed as a stacked line-graph or as a stacked bar-graph, like in this figure:

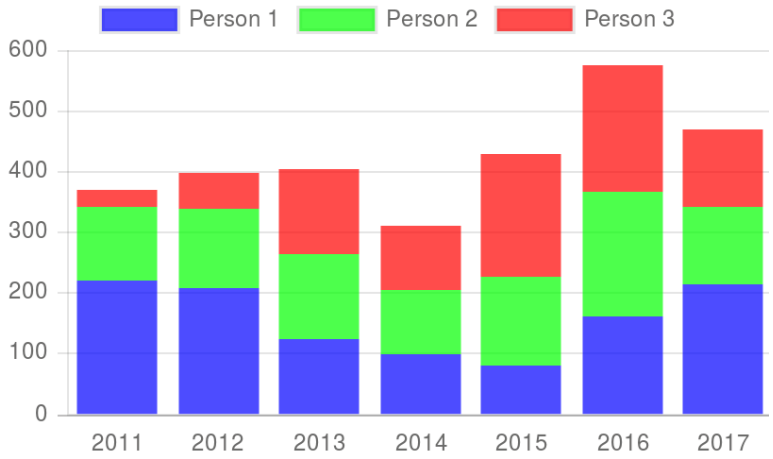


**Figure 6.1:** possible visualization for the transformation "Number of patches contributed by each person of this organizational unit per year"

Here, "Person 1", "Person 2" and "Person 3" are the labels. The years are the keys and the number of patches, of each of the persons, are the values.

## 6.4 Simpler Transformations

Furthermore, with the current implementation, the metric designer has to make sure his transformation stores result rows for every agent. This makes transformations complex, as usually result rows have to be grouped by the id of the agent first and then results are calculated for the groups. For example to get the number of patches each person contributed, the transformation first has to group by the person id and then count the number of patches for each id.
It is easier to define a transformation for only one single agent. Then the metric designer only has to worry about storing the result-rows for this agent instead of all agents. Of course the transformation has to run for all agents then. Pentaho provides a mechanism to do this.
Pentaho offers the functionality to create so called jobs. They can be defined in Spoon just like transformations, but one can connect transformations in a job and pass data from one transformation to another. For this, transformations offer the "Copy Rows to Result"-step, which allows for transferring rows of data (in memory) to the next transformation[1]. The next transformation is configured to run for each input row and to read the fields of the input row, it receives, as parameter. This is done via the transformation settings tab of the selected transformation, inside the job.

[1]http://wiki.pentaho.com/display/EAI/Copy+rows+to+result

So the job can call the first transformation, which reads rows from a table and passes them to the "Copy Rows to Result"-step. Then the job calls the second transformation, which takes the rows fields as input values. The second transformation runs for every row and uses the rows fields as parameters.

So for CMSuite one could define a job like the following:

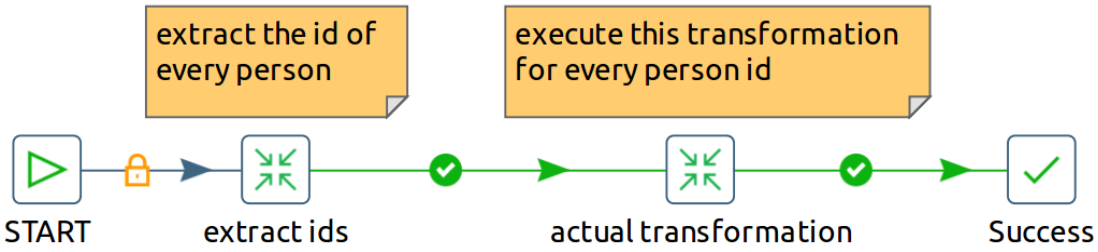


**Figure 6.2:** job with transformations

When the job is started, the first transformation extracts the ids of all persons from the person table. It stores all ids into memory using the "Copy Rows to Result"-step. The second transformation is the actual transformation, that the metric designer designs. It runs for every id and the id is available as parameter, and thus simple to use inside the transformation.
All in all this would make the metric designers life easier, as he would only need to think about one single person when defining the transformation instead of all persons (or any type of agent) and would also not have to manually extract the ids himself in every transformation but can just use the incoming parameter.
The change to CMSuite, that would have to be done, is to run the job from Java, containing the transformation uploaded by the metric designer, instead of the transformation itself. As the job is actually never changing it is possible to always use the same job and replace the transformation in the Java code.

# References


Allamaraju, S. (2010). *Restful web services cookbook: solutions for improving scalability and simplicity.* " O'Reilly Media, Inc.".

Barton, R. (2013). *Talend open studio cookbook.* Packt Publishing Ltd.

Capraro, M. & Riehle, D. (2017). Inner source definition, benefits, and challenges. *ACM Comput. Surv. 49*(4), 67:1–67:36. doi:10.1145/2856821

Casters, M., Bouman, R. & Van Dongen, J. (2010). *Pentaho kettle solutions: building open source etl solutions with pentaho data integration.* John Wiley & Sons.

Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures* (Doctoral dissertation, University of California, Irvine). AAI9980887.

Fowler, M. (2008). *Patterns of enterprise application architecture.* Addison-Wesley.

Lampitt, A. (2008). Open-core licensing (ocl): is this version of the dual license open source business model the new standard.

Moody, G. (2017). How do you make a pentaho? http://www.computerworlduk.com/it-business/how-do-you-make-a-pentaho-3568902. Accessed: 2017-08-29.

O'Neil, E. J. (2008). Object/relational mapping 2008: hibernate and the entity data model (edm). In *Proceedings of the 2008 acm sigmod international conference on management of data* (pp. 1351–1356). SIGMOD '08. Vancouver, Canada: ACM. doi:10.1145/1376616.1376773

Oracle. (2017, August). Core j2ee patterns - data access object. http://www.oracle.com/technetwork/java/dataaccessobject-138824.html. Accessed: 2017-08-10.

Create Step Plugins. (2017). https://help.pentaho.com/Documentation/6.1/0R0/0V0/010/000. Accessed: 2017-08-18.

Creating Extensible Applications. (2017). https://docs.oracle.com/javase/tutorial/ext/basics/spi.html. Accessed: 2017-08-30.

Pentaho Licenses. (2017). http://www.pentaho.com/license. Accessed: 2017-08-13.

Typescript Homepage. (2017, May). http://www.typescriptlang.org. Accessed: 2017-05-16.

Riehle, D. (2012). The single-vendor commercial open course business model. *Information Systems and e-Business Management*, *10*(1), 5–17.

Riehle, D., Capraro, M., Kips, D. & Horn, L. (2016, December). Inner source in platform-based product engineering. *IEEE Transactions on Software Engineering*, *42*(12), 1162–1177. doi:10.1109/TSE.2016.2554553

Stol, K. J. & Fitzgerald, B. (2015, July). Inner source–adopting open source development practices in organizations: a tutorial. *IEEE Software*, *32*(4), 60–67. doi:10.1109/MS.2014.77

Vitharana, P., King, J. & Chapman, H. (2010, October). Impact of internal open source development on reuse: participatory reuse in action. *J. Manage. Inf. Syst. 27*(2), 277–304. doi:10.2753/MIS0742-1222270209