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

HARISH VIJAYAMOHAN
MASTER THESIS

A METRIC DASHBOARD FOR INNER SOURCE

Submitted on 29 April 2019

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, 29 April 2019

License

Erlangen, 29 April 2019

Abstract

Inner Source is a software development approach that uses open source practices within an organization. The Collaboration Management Suite (CMSuite) developed by the Open Source Research Group at the Friedrich-Alexander University Erlangen-Nürnberg supports the analysis and visualization of data about the inner source. However, it lacked the feature to show visualizations for all organizational units of different organizational dimensions. Also, a metric designer could not get descendants of the selected organizational unit within a single transformation. Consequently, One had to create duplicate steps to deal with the hierarchical order of the organizational units.
In this thesis, the dashboard infrastructure is iteratively extended in the way needed for the new metric implementations. It focuses on basic descriptive statistics and patch-flow metrics. Additionally, it elaborates a Java-based REST-service, which allows the user to execute metrics using the hierarchical data injected via the Pentaho data integration tool. The result is then visualized in Angular 6.0 client component for the metric dashboard. Now the metric calculations are performed for selected organizational units including its descendants. Thus, this addition to CMSuite will enable a metric designer to define new metrics without having to think about handling parent-child relations.

Contents

1 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). In inner source, companies open up source code internally so that all employees can see, reuse and contribute changes to it independently of their team. Managers and other individuals in software industry show an increasing interest in measuring inner source collaboration and showing metrics about it. To better understand IS, researchers and practitioners need to measure IS collaboration(Capraro, Dorner & Riehle, 2018).
In order to analyze inner source collaboration, the Open Source Research Group at the Friedrich-Alexander-University Erlangen-Nuernberg is developing the Collaboration Management Suite (CMSuite). Using this software, an organization can retrieve patches and its related information from source code repositories like Git. With this extracted data, it is possible to define different metrics, which are a measure for inner source evaluation. By transforming the results of these metrics into different visualizations, developers and managers can manage the inner source process efficiently.

The contributions of this thesis are:

- The extension of CMSuite dashboard to display visualizations for all dimensions.
- The storage of transformation files in the database instead of file system.
- A feature to load default kettle transformation files on a fresh install of CMSuite.
- The implementation of different inner source metrics, visualizations.
- A feature to update the transformation metadata.
- The categorization of visualizations for the whole organization and other org. elements.
- A method to collect hierarchical information of the organization and then inject it inside transformations at runtime.

1.1 Previous Work

CMSuite allows a user to extract data about code contributions from the repositories using the patch-flow crawler. These extracted patches were the only means of information on which the measurements were carried out. Later, CMSuite is extended to have a metric dashboard that enables the user to visualize results. This was made possible by using a data integration tool Pentaho Kettle. The dashboard is designed in a way such that it can display the results stored in the database. These results are stored by running kettle transformation files in CMSuite. Pentaho Kettle data integration tool provides the option to run the transformations inside java application. However, there were no kettle transformations implemented. There were only some examples which show how the data integration tool executes the transformation and how the results are shown as visualizations.

1.2 Purpose

Before this thesis, the metric designers were able to create transformations only for leaf nodes(deepest org. element). Also, it was not known which metrics were important for creating different kinds of visualizations. Therefore, it was thus desirable to implement all possible metrics (Daeubler, 2017). In contrast, this thesis focuses on the implementation of the metrics that help to measure inner source instead of implementing all possible ones. The purpose of this work is to extend the dashboard component of CMSuite to allow the user to add metrics that would work for all org. elements agnostic of granularity. In order to inject all the hierarchical data of the organization, CMSuite is extended to use row listeners that are features provided by Pentaho data integration tool. The results of metric calculations are visualized in Angular 6.0 dashboard component. Most of the metrics to measure patch-flow across organization levels are implemented based on ideas described in (Capraro et al., 2018). The metrics are uploaded through web-client and stored in the database as well. Furthermore, it also allows the user to modify some of the metadata of these metric files. In addition to that, the web client is also extended to display the visualizations in a larger size. As a result, it helps users to perform their analysis effectively.

1.3 Requirements

1.3.1 Stakeholders

The stakeholders of dashboard module are:

- A *Developer*, whose role is to develop new features of CMSuite.

- An *Administrator*, who monitors the operations of CMSuite.
- A *Metric Designer*, who defines new metric that facilitates the evaluation of inner source process.
- An *Inner Source Stakeholder*, who is interested in knowing more about ISPs within an organization.

In addition to the roles mentioned above, there are also other roles like contributor, committer, etc. However, defining metrics based on a particular role is not the motivation of this thesis. Because there are no implementations based on role till now, it is thus desirable to create all possible metrics. Therefore, all the above roles are considered to be one role called "Inner Source Stakeholder" throughout this thesis.

1.3.2 Functional Requirements

Following are the functional requirements defined based on workshops and discussions.

1. As a metric designer, I want to store the KTR files in the database instead of the file system, so that the files will be in sync with the tables in the database.
2. As a metric designer, I want to be able to update the transformation, so that I do not have to delete and then create a new one if any modifications are required.
3. As a metric designer, I want to download the KTR file so that I can easily bring out my changes to KTR file.
4. As an administrator, I would like to see all transformation available in database after a fresh install, so that I do not have to load them one by one manually every time.
5. As an inner source stakeholder, I want to display the visualization that presents the number of code contributions over time, so that I can observe if the contributions are increasing or decreasing over time.
6. As an inner source stakeholder, I want to see the number of code contributions made per month by namespace agnostic of Org. element granularity, so that I can find out which namespace had most of the contributions over time.
7. As a metric designer, I would like to have an option to choose if visualizations should be displayed only for root Org. element or other Org. elements, so that I do not have to use separate steps inside the ktr file for segregating it.

8. As an inner source stakeholder, I would like to view all the visualizations in a larger size, so that the analysis can be carried out effectively. In this larger tile the legend of the chart should be displayed.
9. As an inner source stakeholder, I want to see the number of code contributions received by an inner source project per month, so that I can see the development activity and estimate the ISP's survivability.
10. As an inner source stakeholder, I want to see the number of persons involved over time so that I can estimate if the ISP is still alive.
11. As an inner source stakeholder, I want to see the number of persons making their first code contribution agnostic of type and granularity, so that I can find how many new persons are involved over time.
12. As an inner source stakeholder, I want to visualize the number of persons that had their last contributions per month agnostic of type and granularity, so that I can find how many persons have lost involvement in the inner source.
13. As an administrator, I want to see the data completeness of the number of code contributions received over time, so that I can have an overview of how good and reliable the data is.
14. As an inner source stakeholder, I want to see the data completeness of the number of code contributions contributed over time, so that I can investigate where I need to manually tweak the data.
15. As an inner source stakeholder, I want to see the patch-flow by Org. levels over time, so that I can measure how many code contributions were made across organizational boundaries.
16. As an inner source stakeholder, I want to see the number of code contributions received over time by Org. levels, so that I can find all the active ISP's that received patches across organizational units.
17. As an inner source stakeholder, I want to see the number of authors and their code contributions made.
18. As an inner source stakeholder, I want to see the relative patch flow over time, so that I can find the patch-flow relative to the total amount of code contributions to the IS projects per month.
19. As an inner source stakeholder, I want to see the code contributed by teams of selected org. element over time.
20. As an inner source stakeholder, I want to export the visualizations to PDF.

21. As a administrator, I want to have a better design for dashboard components, so that the two different modules of dashboard can be merged into one.
22. As an inner source stakeholder, I want to be able to personalize the dashboard, so that I can pick specific org. element or inner source project to be displayed.

1.3.3 Non-Functional Requirements

1. The response time of the dashboard module should stay within one to five seconds.
2. The tool handles faulty results properly so that nobody has to restart the broken system.
3. The steps implemented in the kettle transformation (ktr) files should not be very complicated.
4. The transformations should execute for all types of datasets irrespective of size of the dataset.
5. The visualizations should not be incomprehensible. That is, chart clutter should be avoided.
6. The legends displayed in the chart should not show the id of org. element or ISP. Only the name of the agent should be displayed.

2 Architecture and Design

2.1 Design Decisions

Prior to this thesis, the transformation manager module was implemented such that it uses Pentaho data integration tool for metric definitions. This tool is integrated into CMSuite, which lets the user execute the transformations from Java. However, there were no metrics defined to display proper visualizations agnostic of granularity. The spoon is a graphical user interface that allows users to quickly design metrics with the help of predefined steps. But it lacked the ability to perform loops over steps in a transformation, that were needed to be repeated. CMSuite needed this feature so that it makes it easier for a metric designer to deal with the hierarchical order of organizational units.

The fundamental question of design phase is how to get these hierarchical information of oganizational units without affecting the execution speed of the kettle transformations and without making the KTR files too complicated. Several solutions are discussed in this chapter.

The first possible solution is to repeat the combination of steps inside the single kettle transformation file to reach the depth of organizational tree. But to get complete hierarchical order, one must know the number of levels present in the selected organization. For example, if the depth of the organizational tree is 14 levels, then the metric designer would have to append the specific combination of steps for 14 times. As a result, it makes the KTR file very complicated and difficult to read. Moreover, it affects the execution speed of the transformation.

The next possible solution is to use SQL to extract the hierarchical information of the selected organizational unit. They could be entered in the *Table input* step provided by GUI spoon. However, it results in poor performance when the query is executed for every output row from the previous step in spoon. Moreover, the metric designer would have to be a programmer to create a recursive SQL to extract the hierarchical order of organizations. Thus, it is not desirable to use SQL to resolve this problem.

In contrast to previous solution, the *User Defined Java Class* step could be used

to improve the performance. This step allows the metric designer to write his own step plugin in the form of Java code that is entered in the dialog of the step itself. This code is compiled at run-time and executed at optimal performance (Casters, Bouman & Van Dongen, 2010). It uses third party library called *Janino* to compile the java code at runtime. Again here the problem is that metric designer would have to be a programmer. Furthermore, this step supports only the older version of java which does not have generics and other important features that are available in latest versions("User Defined Java Class", 2019).

The another possbile solution is to execute kettle jobs instead of transformations. It enables metric designer to create loops over transformations in spoon. However, it requires major changes to be done to the already existing architecture of components related to dasboard. Moreover, if a metric designer needs to perform a large number of iterations he would notice that the solution is slow. This performance problem is caused due to the loading of the metadata for the job that performs the actual work. All this extra work would slow down the job and run the risk of running out of heap space (Casters et al., 2010).

The easiest possible solution for this problem is to extend the transformation manager to use row listener interface of Pentaho data integration tool. Row listener could be used for reading the data from the step at runtime. For the data read from step, the required data could be gathered and then injected back into the same step using *StepInterface* . For example, the descendant org. elements could be injected into the step for the org. element read by the row listener. A developer of CMSuite could implement different row listeners that would be used for injecting data into step. A metric designer does not have to be a programmer but he must know what attribute names should be used for getting this injected data. Thus, CMSuite could be extended to use different types of row listeners for injecting data into the step at runtime. It will be further explained how it is integrated inside the transformation manager.

2.2 Third-Party Tools

Pentaho data integration(PDI Kettle), Ngx-charts, Rhino (JavaScript engine), JavaMail and The Apache Commons Mathematics Library are the third party tools used in this thesis. The PDI Kettle was already included in CMSuite prior to this thesis. All others dependencies are newly added for this project. Below are the license information about newly added third party dependencies.

- Rhino(JavaScript engine) - Mozilla Public License (MPL) MPL 1.1
- JavaMail - Common Development And Distribution License (CDDL-1.1) and GNU General Public License, version 2 GPL-2.0

- The Apache Commons Mathematics Library - Apache License, Version 2.0

2.2.1 Pentaho Data Integration (Kettle)

Kettle contains a rich set of data integration functionality that is exposed to a set of data integration tools (Casters et al., 2010). However, Kettle can also be used as a library in own software and solutions (Casters et al., 2010). Kettle Java API enables us to execute a kettle transformations to extract, transform and load the data. As mentioned earlier, CMSuite now uses Pentaho Data Integration tool to extract the raw patch-flow data from the database, transform the data and stores it to persistent storage.

Kettle Transformations

A *Kettle tranformation* handles the manipulation of rows or data from database tables. 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. The steps in a transformation are connected by transformation *hops*. The hops define a one-way channel that allows data to flow between the steps.

Injecting Data into Kettle Transformations

As mentioned earlier, the hierarchical data of organizational units has to be injected for some metric calculations. In order to facilitate this injecting process, current rows from the steps are needed to be read at runtime. This can be achieved by using *RowListener*. Data can be read from a step in a streaming fashion using row listener interface. Once the current rows are read, the corresponding hierarchical data can be injected inside that particular step at runtime.

2.2.2 Ngx-charts

Before this thesis, the dashboard module was using chart.js to visualize the results. The problems in using chart.js with Angular is that they violate the single point of DOM contact policy. Because the chart.js framework is not Angular2+ code, they would be touching the DOM independently. This can cause problems when both Angular and the chart.js framework is manipulating the DOM simultaneously. To overcome this problem, it is replaced with ngx-charts. Ngx-charts is a unique charting framework for angular because it is using Angular to render and animate the SVG elements with all of its binding and speed goodness, and uses d3 for the excellent math functions, scales, axis and shape generators, etc. Ngx-charts is open source and is available under the MIT license, so it can be used in this project.

2.2.3 Rhino(JavaScript engine) and JavaMail

Rhino is a JavaScript engine written in Java and managed as open source software by the Mozilla Foundation. Rhino converts scripts from JavaScript into classes. JavaMail is a Java API used to receive and send an email via IMAP, SMTP, and POP3. In this thesis, the *ModifiedJavaScript* scripting step from spoon's menu is used to perform some metric calculations using javascript. However, Pentaho data integration uses *Rhino* and *JavaMail* to support this scripting step. Thus, it works well when the KTR file is executed in spoon. But when it is executed through CMSuite, it breaks the execution by throwing classes not found exception. Although JavaMail is not required for our thesis, it is added because *ModifiedJavaScript* step of PDI requires it ("ModifiedJavaScript step requires JavaMail", 2019). In order to execute the KTR files successfully in CMSuite, these dependencies are added to CMSuite as same as how kettle PDI uses it in its pom.xml file ("PDI Kettle's Third-party dependencies", 2019).

2.2.4 The Apache Commons Mathematics Library

Commons Math is a library of mathematics and statistics components that helps in solving the most common problems not available in the Java programming language. In this thesis, the *GroupBy* statistic step is used from spoon's menu to perform some statistical calculations. As mentioned before, Pentaho data integration uses third-party dependencies to support this statistical step ("Groupby step requires Apache math", 2019). Thus, it works well when the KTR file is run in spoon. But when it is executed through CMSuite, it breaks the execution by throwing classes not found exception. In order to execute the KTR files successfully in CMSuite, this dependency is added to CMSuite as same as how kettle PDI uses it in its pom.xml file ("PDI Kettle's Third-party dependencies", 2019).

2.3 Domain Model

Prior to this thesis, only three result types were there in the domain model of CMSuite dashboard. The *SingleValueResult*, *CategorizedValueResult* and *TimeSeriesResult* existed before, which was used for visualizing simple results. In order to visualize complicated results from metric calculations, new types of results are included in the already existing domain model as shown in figure 2.1. The newly introduced domain classes are the following:

- *CategorizedTimeSeriesResult*: Contains a result with nested multiple key-value pairs, with the parent key being a category. This is used to represent results for metrics, that produce multiple date-value pairs for different categories of that particular EconomicAgent. For example: Metric for organizational units: "Number of Code Contributions made to various namespace

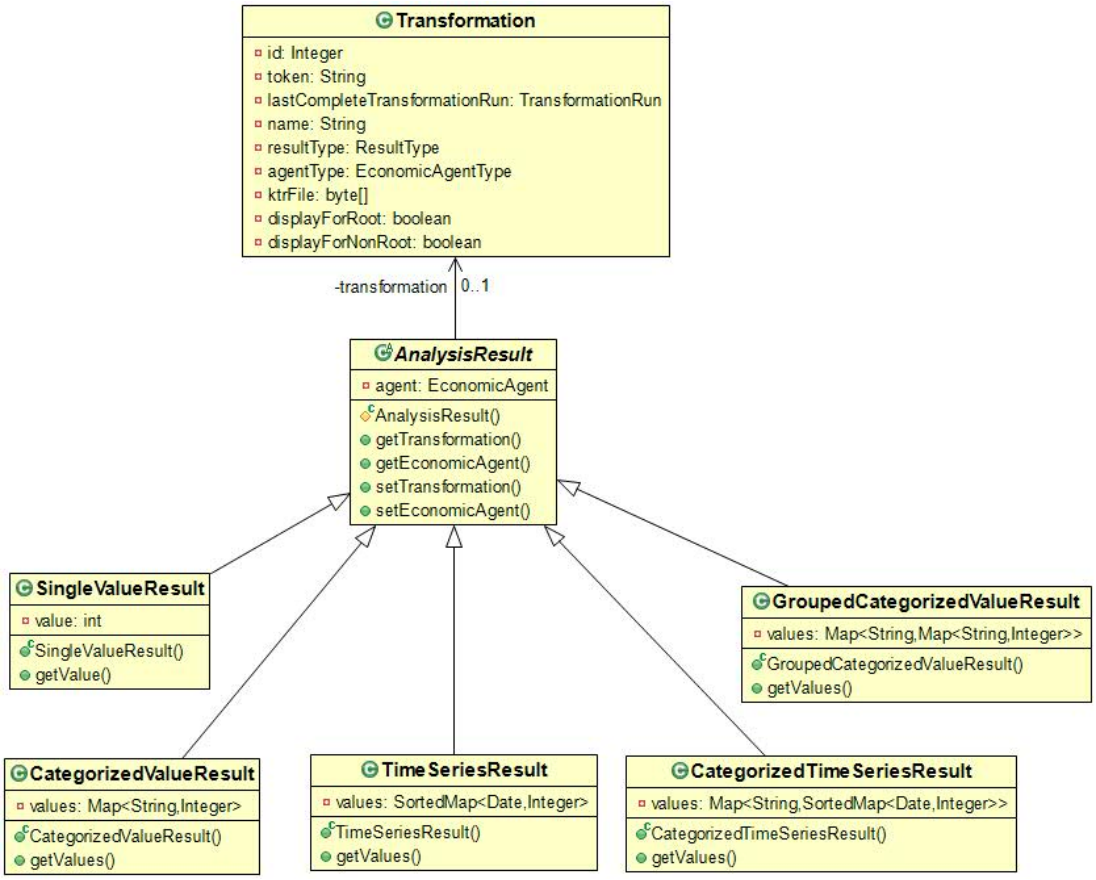


Figure 2.1: Class Diagram of Results

per month".

- *GroupedCategorizedValueResult*: Contains a result with nested multiple key-value pairs, with parent key being a category. This is used to represent results for metrics, that produce multiple string-value pairs for different categories of that particular EconomicAgent. For example: Metric for organizational units: "Number of code contributions received per ISP across Org. levels".

The modified domain classes are the following:

- *Transformation*: A metric is calculated by executing a transformation, that defines the steps to create the results for this metric. The Transformation class contains the meta-data about the transformation. Furthermore, it is extended now to carry the Kettle Transformation File (.ktr) in the format of a byte array. Also, it is extended to have the information about whether the transformation must be displayed for root Org. element or other Org. elements.

2.4 Persistence

2.4.1 Persisting Transformations

Before running the transformations in transformation manager, kettle transformation(KTR) files have to be uploaded manually. Prior to this thesis, CMSuite saves the uploaded files to a temporary directory. Consequently, the user had to upload the transformation files on every fresh install. In order to avoid this manual adding process, CMSuite has to persist the KTR files in a database. To achieve this, the KTR files are converted to byte array format and then stored along with transformation object.

2.4.2 Database schema

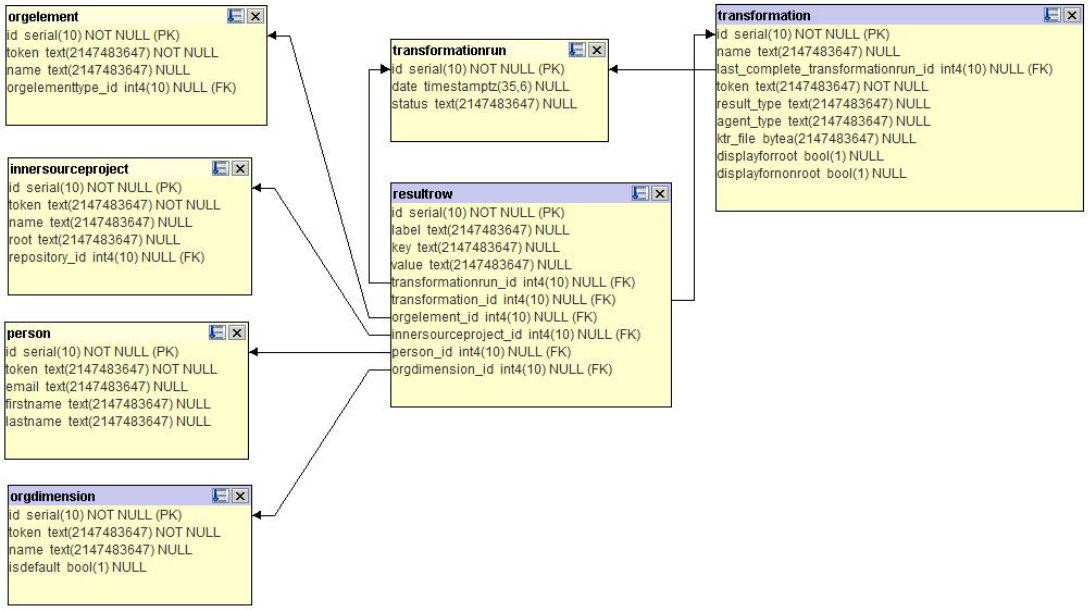


Figure 2.2: Database Schema

Prior to this thesis, the *resultrow* table was created with foreign key constraints to maintain referential integrity with other tables like *orgelement*, *innersourceproject*, and *person*. However, the results were stored in the resultrow table without considering *organizational dimensions*. In order to store the results with respect to different dimensions, another constraint is added to relate *resultrow* table and orgdimension table. As shown in figure 2.2, a column *orgdimension_id* is added to *resultrow* table with a foreign key constraint to have referential integrity with *orgdimension* table. In addition to that, *transformation* table is altered to have three new columns such as *ktr_file*, *displayforroot* and *displayfornonroot*.

ktr_file is used for storing the kettle transformation file as binary data. Other two columns are of boolean data type and it is used to mark if the transformation should be displayed for root Org. element or other Org. elements.

Storing Binary Data

PostgreSQL provides two separate ways for binary data to be stored. Binary data can be stored in a table using the *bytea* data type or by using the Large Object feature that stores the binary data in a special format in a separate table and refers to that table by storing a value of type *oid* in the table. The Large Object method for storing binary data is better suited only to store very large values of more than 1GB ("PostgreSQL Storing Binary Data", 2018). In contrast, a column of type *bytea* data type is well suited for storing small amounts of binary data. Since kettle transformation files are not going to be very large in size, the *bytea* data type is chosen to store it in the transformation table.

2.4.3 Persisting Results

Similar to other three result types, the transformations that create *Categorized-TimeSeriesResult* stores the category(parent-key) to *label* column, date(child-key) to *key* column and value to the *value* column of *resultrow* table. Similarly, the transformations that create the *GroupedCategorizedValueResult* also stores category(parent-key) to *label* column, string(child-key) to *key* column and value to the *value* column of *resultrow* table. The table below shows the layout of result row table in which different types of results are stored. Additionally, the dimension is also included to this resultrow table in this thesis. The dimension column stores the *orgdimension_id* of the corresponding result. With its help, analysis result provider can retrieve the results based on dimension id that is passed as a query parameter as shown in table 2.1.

tm_run_id	tm_id	person_id	org_id	isp_id	dim_id	label	key	value

2.5 Architecture

CMSuite has two services that deal with transformations and results. 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 (Daeubler, 2017).

2.6 REST Endpoints

CMSuite implements a RESTful API to do its server-client communication. The resources that were modified for this project are *Transformation* and *AnalysisResult*. Within CMSuite the organizational hierarchy is logically segregated into different perspectives, called organizational dimensions. Navigation of the organizational hierarchy was implemented but was not used in the dashboard module, i.e. organizational dimensions were ignored. While the ETL approach is valid for computing metrics, it does not work well with data that is logically segregated (by the org. dimensions)(Hansen, 2018). In order to overcome this problem, the organizational dimension is also added as one of the query parameters in the existing URIs to retrieve results based on a specific dimension as shown in table 2.1.

Table 2.1: URIs of AnalysisResult resource.

URI
{agentId}/singlevalue/?dim={dimId}
{agentId}/categorizedvalue/?dim={dimId}
{agentId}/timeseries/?dim={dimId}
{agentId}/categorizedtimeseries/?dim={dimId}

These endpoints are modified to retrieve the different types of result for the given dimension. As they are all just for retrieving, they all use the HTTP GET method. Furthermore, a PUT method is added to transformation resource as shown in table 2.2. It is used to modify the transformation object.

Table 2.2: URIs of AnalysisResult resource.

URI	**HTTP Method**
/transformations/	PUT

2.7 Server Components

2.7.1 Transformation Manager

The *transformationmanager* component retrieves and stores the transformation. In addition to this implementation, now it also retrieves and stores the kettle transformation file (.ktr) in the byte array format. Furthermore, it manages the execution of transformations, which includes keeping track of the current state of running, finished or failed executions(Daeubler, 2017). However, it lacked the

ability to inject the hierarchical order of organizational units into the transformation during runtime. In order to fulfill this requirement, the transformation manager is extended to have *rowlistener* package which deals with reading data from current rows during runtime. As a result, the CMSuite injects the hierarchical order information into the transformation step for the corresponding incoming row. This hierarchical order details of an organization unit are collected using Depth First Search(DFS) algorithm, which will be discussed in section 3.2. This DFS method for traversing the tree is implemented in a separate class called DescendantLoader, as it loads all the descendants of the given Org. element. The figure 2.3 shows the class diagram of rowlistener package in transformation manager.

Row Listener

RowListener is an interface provided by Pentaho data integration(PDI) tool, which helps in reading the data from transformation while they are being executed (Casters et al., 2010). This Interface has three abstract methods called rowReadEvent, rowWrittenEvent, and errorRowWrittenEvent. Among these three methods rowReadEvent is considered to be a useful method for our project. It reads the rows from the current transformation at runtime, which we use as a parameter to gather hierarchical information of that current row. As shown in figure 2.3, the *AbstractRowListener* implements the *RowListener* interface provided by PDI. Following are the two rowlisteners that are extended from *AbstractRowListener* and customized based on their purpose.

- *DescendantRowListener*: To collect all the descendants of orgelement read from the current row that is being executed in the transformation step.
- *SegregationRowListener*: To segregate the external and internal Org. elements based on the comparand columns of the current row that is being executed in the transformation step. As shown in figure 2.3, this rowlistener is further extended to *IspSegregationRowListener* and *PersonSegregationRowListener* to perform segregation(Internal/External) process for ISPs and Persons respectively.

Furthermore, the *KettleTransformer* is modified to create an instance of factory class *RowListenerFactory*, which helps in creating the different instances of rowlistener based on the specific step names like *cmsuite_inject_descendants*, *cmsuite_inject_isp_segregator* and *cmsuite_inject_person_segregator* that occur in the current transformation. Finally, the rowlistener object created by *RowListenerFactory* is attached to those specific step in the transformation at runtime ("Executing a PDI Transformation", 2019).

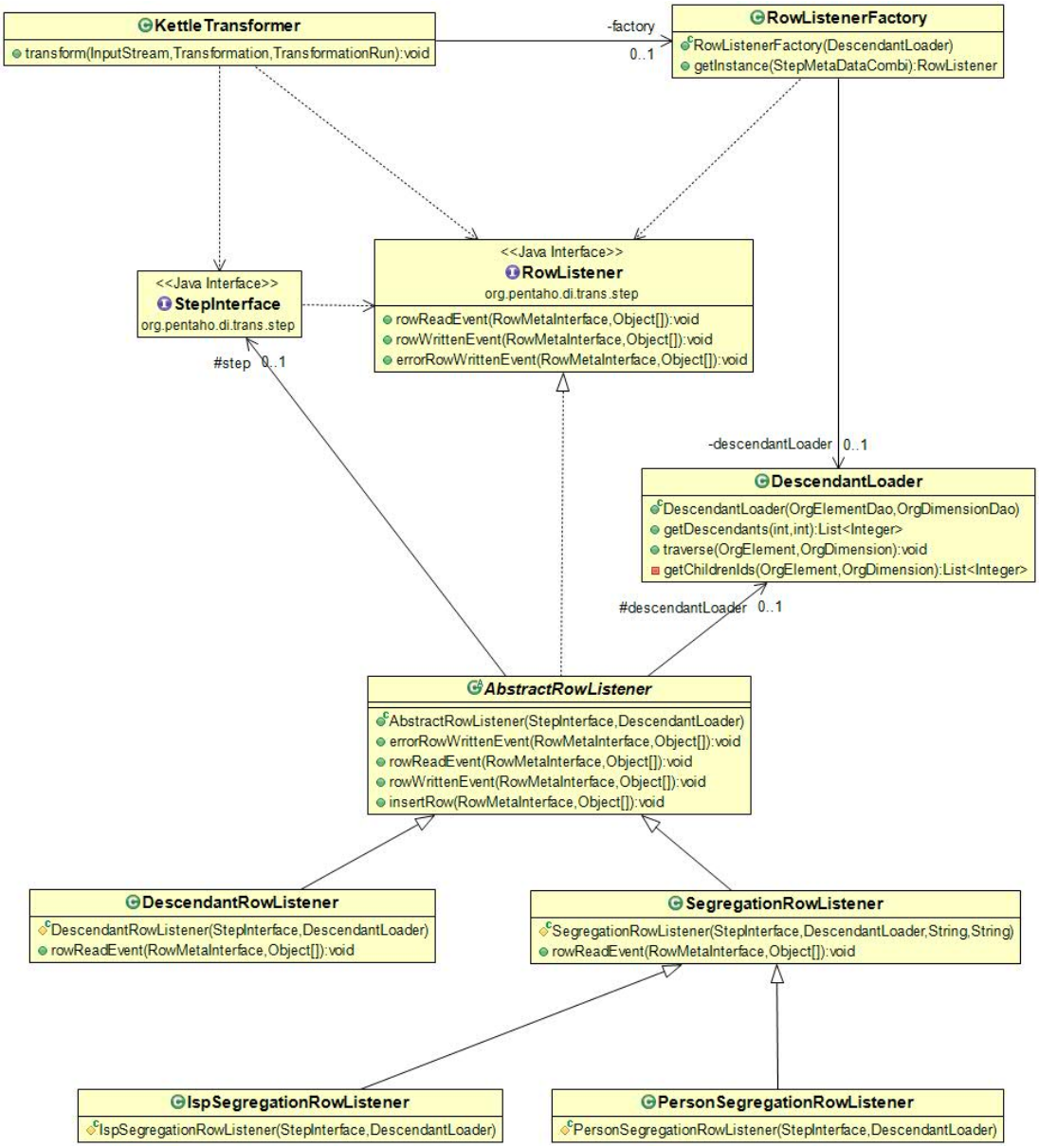


Figure 2.3: Class Diagram of Transformation Manager

Descendant Loader

A *DescendantLoader* class is created to collect all the descendants of a selected Org. element during the execution of the transformation. As shown in class diagram 2.3, an instance of *DescendantLoader* is created by *RowListenerFactory* and then it is passed to *AbstractRowListener* through the constructor. Since *DescendantRowListener* and *SegregationRowListener* are extended from *AbstractRowListener*, this instance of *DescendantLoader* can be used by them. As all the row listeners require the same heirarchical information of organizational units,

only one instance of descendant loader is created per Transformation run. That is, same instance is used by all the row listeners until the end of one transformation run.

Default Ktr File Controller

As mentioned before, the KTR files are stored as byte array along with the transformation object in the database. Before this thesis, KTR files were handled using FileHandler to retrieve it from a temporary directory. Now that all the files are located in the default folder, a question arose how to retrieve the data from the default folder to show it on the transformation page. A simple solution to resolve this problem is to read all the KTR files from a default folder and then store it in the database. To achieve this, the already existing *FileHandler* is replaced with *DefaultKtrFileController*. As shown in figure 2.4, the TransformationResource has an instance of TransformationService which will try to call getAll() method to get list of transformations. In order to load the known transformations(KTR files), an instance of DefaultKtrFileController is used in TransformationService.

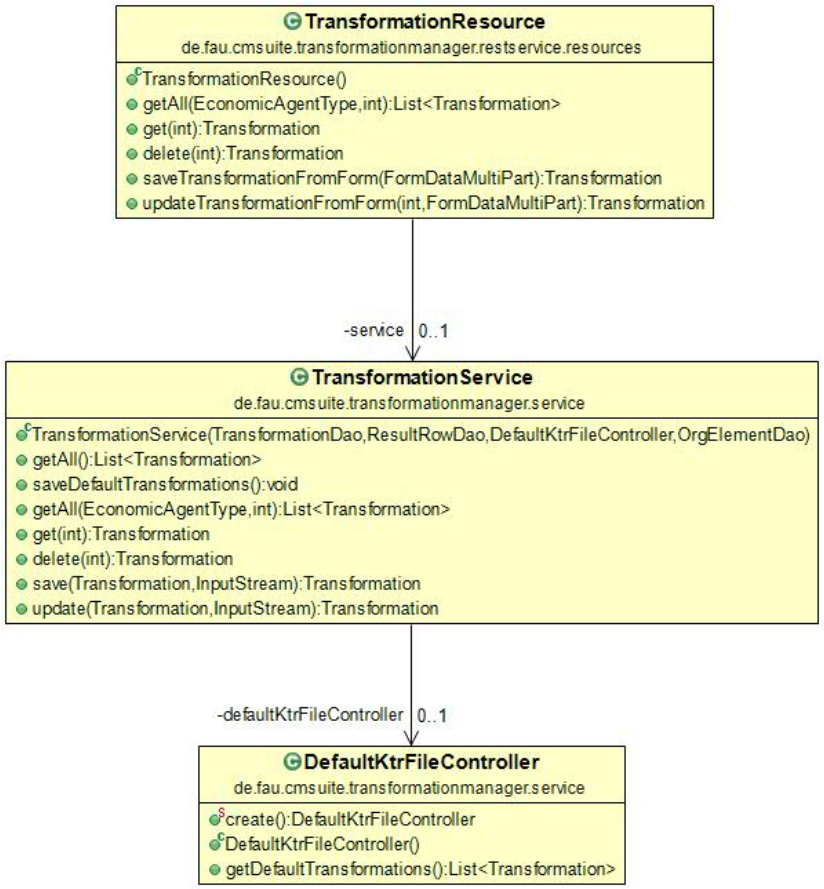


Figure 2.4: Class diagram of transformation manager

2.8 Webclient

CMSuite's web-client is built using a front end web framework called Angular 6.0. The modules that contains the functionality of dashboard are *transformationmanager*, *dashboard* and *analysisresult*. Changes made to these modules are discussed in the following sections.

2.8.1 Transformation-Manager-Module

Transformationmanager-module allows the user to create new transformations, run, and delete them as well. However, it lacked the feature to update the transformations that were stored in the database. In order to allow user to modify the transformation, *TransformationCreationComponent* has been replaced with *TransformationDetailsComponent*, which has both add and edit functionality.

2.8.2 Analysis-Result-Module

Analysisresult-module is responsible for retrieving the results that were stored in database after transformation run. Prior to this thesis results were already classified into *SingleValueResultComponent*, *CategorizedValueResultComponent* and *TimeSeriesResultComponent*. Additionaly, now *CategorizedTimeSeriesResultComponent* and *GroupedCategorizedValueResultComponent* are added to this module, to display timeseries and grouped value results in categorized manner respectively. Further details will be discussed in chapter 3.4.

2.8.3 Dashboard-Module

Dashboard-module is responsible for displaying the visualizations for the corresponding agent that are selected from the tree menu. Prior to this thesis, this module had *SingleValueResultTileComponent*, *CategorizedValueResultTileComponent* and *TimeSeriesResultTileComponent*. In addition to these components, now *CategorizedTimeSeriesResultTileComponent* and *GroupedCategorizedValueResultTileComponent* is also added to display the CategorizedTimeSeriesResults and GroupedCategorizedValueResults in it. Furthermore, it was also required to display the results in a larger size. In order to fulfill this requirement, following components were added.

- *TimeSeriesResultModalComponent*
- *CategorizedTimeSeriesResultModalComponent*

These modal components are implemented using package called Angular Material. It provides material design components for Angular applications("Angular Material Dialog page", 2019).

3 Implementation

In this chapter, the implementation of new modules and how they are integrated inside CMSuite are discussed in detail. It also dicusses what are the changes made to already existing classes so that new ones could be integrated. Additionally, the most commonly used kettle steps inside all the transformations are explained.

3.1 Kettle Transformations

Kettle Transformations are implemented using the GUI called spoon which allows the user to connect steps with hops, as mentioned earlier. Types of transformation implemented and the common steps used in it are discussed in the following sections.

3.1.1 Types of Transformation

The transformations are categorized based on the type of result that they are going to store in the database. Every transformation has to map its output to a resultrow table in database. Therefore, different types of results are required to identify the stored rows so that it can be shown in the corresponding visualization in webclient. For example, to visualize the stacked bar chart, the results should be categorized and then grouped again. To show the multi line chart, the time series result should be categorized. Thus, the result types *GroupedCategorized-ValueResult* and *CategorizedTimeSeriesResult* are introduced in this thesis. It will be further explained how these different types of results generated by metrics are stored in database in chapter 4. Transformations implemented in this thesis are as follows.

1. SingleValueResult
   (a) Code Contributions made(Excluding Internal)
   (b) Code Contributions made(Including Internal)
2. TimeSeriesResult

(a) Code Contributions made per month(Excluding Internal)

(b) Code Contributions made per month(Including Internal)

(c) Persons Involved per month

(d) First Time Contributors per month

(e) Last Time Contributors per month

(f) Code Contributions Received per month

3. CategorizedTimeSeriesResult

   (a) Code Contributions made per month by namespace(Excluding Internal)

   (b) Code Contributions made per month by namespace(Including Internal)

   (c) Data Completeness of Received Code Contributions

   (d) Data Completeness of Contributed Code Contributions

   (e) Patch-Flow by Org. Level

4. CategorizedValueResult

   (a) Authors and their patch contributions

5. GroupedCategorizedValueResult

   (a) Code Contributions Received per ISP across levels

3.1.2 Common Kettle Steps

The kettle steps are used for creating metric calculations in the *Spoon*. Following are a few combinations of steps in the kettle transformation (KTR) files which are most commonly used.

Getting Descendant Orgelements

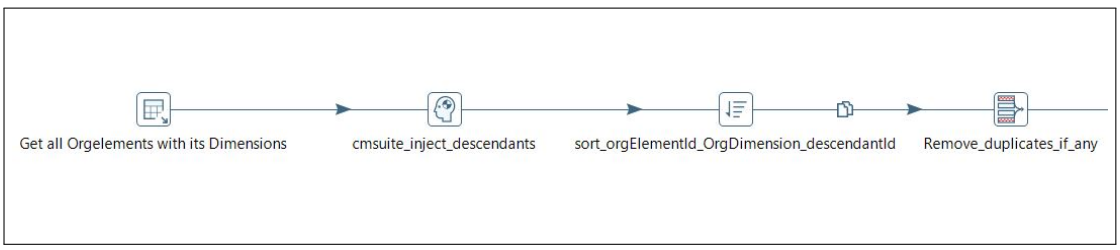


Figure 3.1: Steps to get descendant orgelements

As stated in (Capraro et al., 2018), organizational units are modeled using the composite design pattern and an organizational unit can be composed of child organizational units. In order to deal with this organizational hierarchy in kettle transformation files, the combination of steps as shown in figure 3.1 is used. At first, all the org. elements with its dimensions are retrieved using a simple SQL query in a *Table input* step. After that, all the descendants of the org. elements are retrieved. There are many possible ways to get descendants. One among them is by using a complex recursive query in the *Table input* step. However, it affected the performance during the execution. In order to get the descendants without recursive process, the DFS(Depth First Search) method is used. As mentioned earlier, this method is implemented in the transformation manager module which will be discussed in section 3.2. As a result, the descendants are injected into the transformation without using a recursive query. As shown in figure 3.1, *cmsuite_inject_descendants* is the dummy step in which the descendants are injected. After this, it is sorted and then duplicates are removed if any appears.

Retrieving Persons of Orgelement

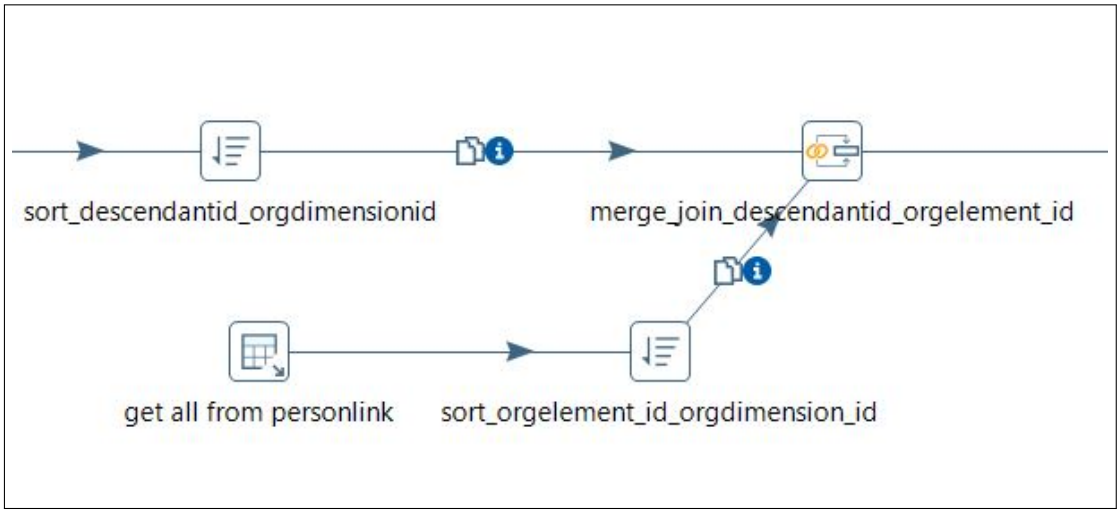


Figure 3.2: Steps to retrieve persons of orgelement

As mentioned in (Capraro et al., 2018), for each code contribution, it contains the person (Person class) authoring a code contribution and they are associated with an organizational unit (OrgUnit class). With the help of steps shown in figure 3.2, all the persons of given org. elements are retrieved. At first, the input elements are sorted based on *orgelementid* and *orgdimensionid*, at the same time all rows from *personlink* table are also retrieved and sorted based on same keys. The *Merge join* step requires all the elements to be sorted based on the joining

key. Now that incoming rows from both directions are sorted, it is eligible for entering *Merge join* step. In this step, *left outer join* is chosen to perform the merge operation. As a result, all the persons of the org. element are placed in the adjacent columns, which makes it easier for using it in further steps.

Retrieving Code Contributions of Persons

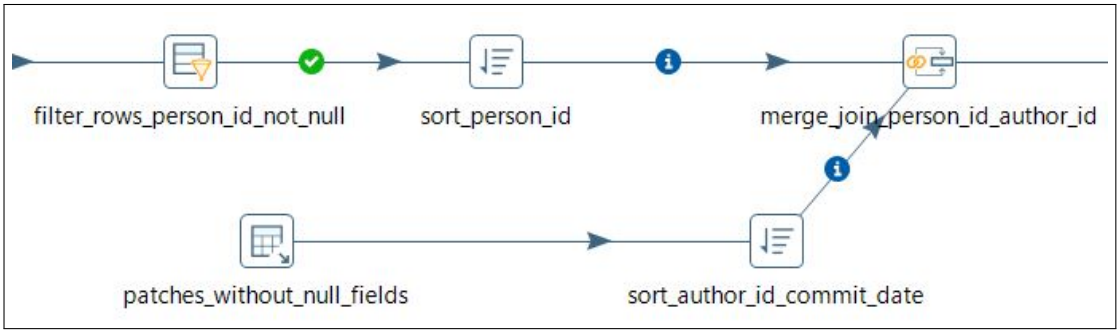


Figure 3.3: Steps to retrieve code contributions of persons

To find the patches contributed by a person, the combination of steps as shown in figure 3.3 is used. A patch is a code contribution from an individual to an inner source project(Capraro et al., 2018). At first, the null fields are filtered out to make sure that all fields in *person_id* column has a personid for further steps. At the meantime, all patches are also retrieved from the *patch* table in the database. Following this, data from both steps are sorted based on *personid* and then output is passed to *Merge join* step. As a result of *left outer join*, all the patches that were contributed by persons are retrieved.

Retrieving ISPs of Orgelement

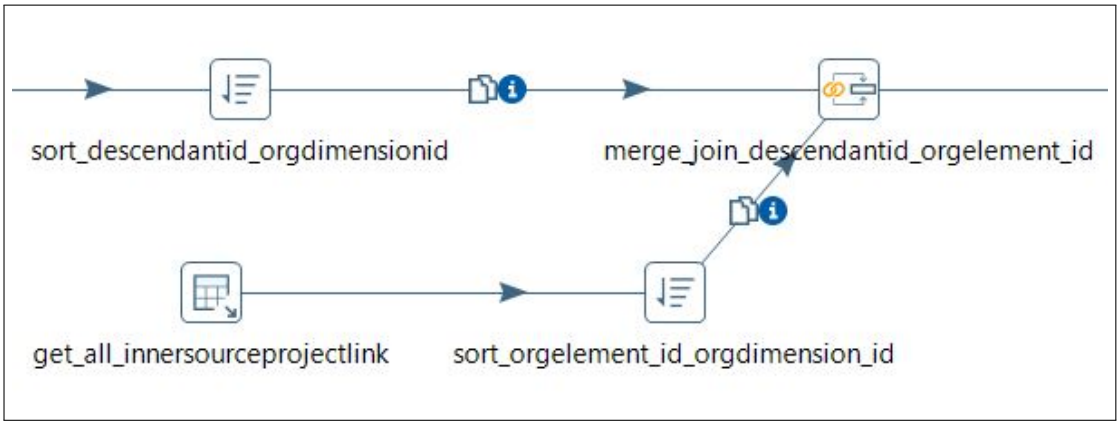


Figure 3.4: Steps to retrieve ISPs of orgelement

According to the object oriented model presented in (Capraro et al., 2018), for each code contribution, it contains the IS projects (InnerSourceProject class) receiving it and is associated with an organizational unit (OrgUnit class). Thus, for a few metric calculations, it is required to find all the inner source projects of one or more org. elements. In order to retrieve all the inner source projects of org. elements, the combination of steps as shown in figure 3.4 is used. At first the input elements are sorted based on *orgelementid* and *orgdimensionid*, at the same time all rows from *innersourceprojectlink* table are also retrieved and sorted based on *orgelementid* and *orgdimensionid*. Now that we have both steps sorted based on the same key, it is eligible for entering *Merge Join* step. In this step, *left outer join* is the option which is chosen to perform the merge operation. As a result, all the innersourceprojects of the orgelement are placed in the adjacent columns, which makes it easier for us to use it for further steps.

Retrieving Code Contributions of ISPs

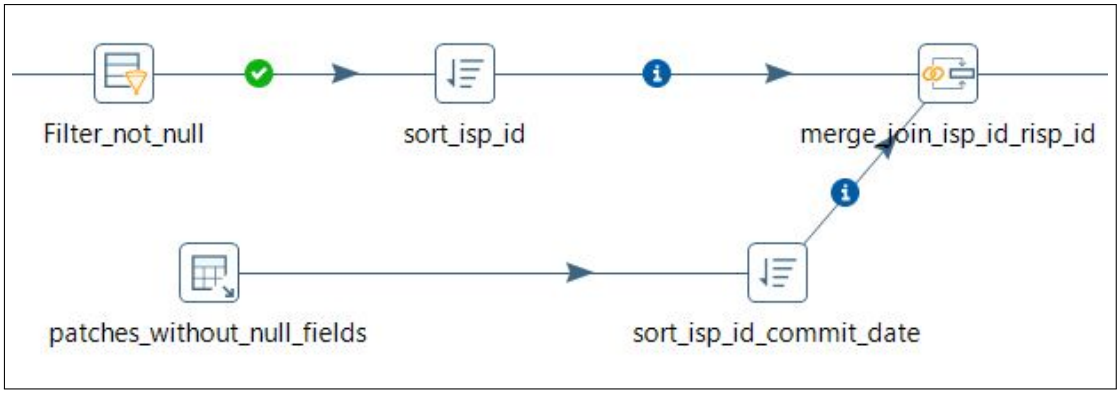


Figure 3.5: Steps to retrieve code contributions of ISPs

To find the patches received by the *innersourceproject*, the combination of steps as shown in figure 3.5 is used. At first, the null fields are filtered out to make sure that all fields in *innersourceproject* column has a *innersourceprojectid* for further steps. At the meantime, all patches are also retrieved from the patch table in the database. Following this, data from both steps are sorted based on *innersourceprojectid* and then passed it for *Merge join* step. As a result of *left outer join*, all the patches that were received by *innersourceproject* are retrieved.

Segregating Internal/External Agents

In OS, a patch is a code contribution from an individual external to an OS project. A developer is considered external to a project if not a member of the organizational unit owning the IS project (Capraro et al., 2018). Thus, for some of the metric calculations, it is required to segregate the external agents for showing

only external activities. In order to get this external or internal information, *cmsuite_inject_isp_segregator* and *cmsuite_inject_person_segregator* step is used in a combination as showed in figure 3.6 and figure 3.7 respectively. These steps injects the boolean value to the column *is_external_isp* and *is_external_person*. The process of injecting this boolean value is explained later in this chapter.

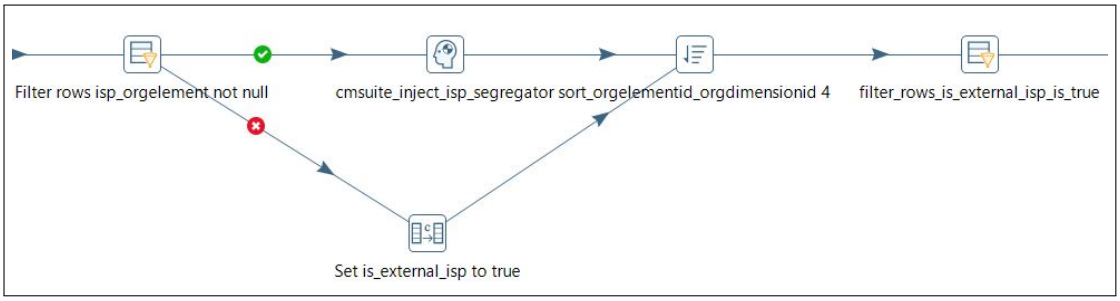


Figure 3.6: Steps to segregate external innersourceprojects

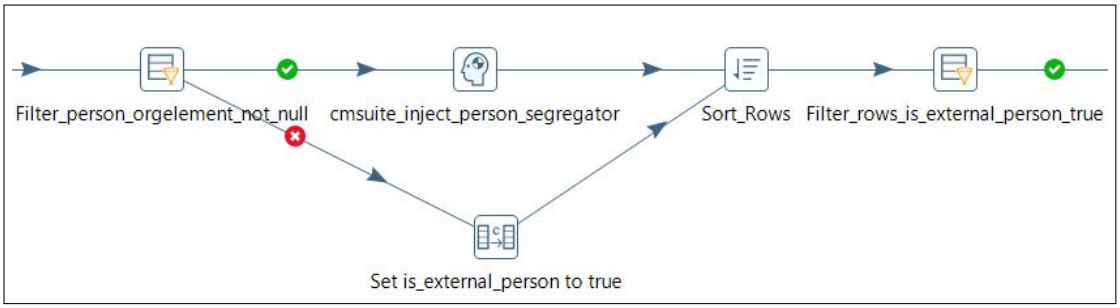


Figure 3.7: Steps to segregate external persons

Appending Missed Agent

For some of the metric calculations, the agents who did not have any activity to show would have got filtered out in initial steps. Consequently, the webclient breaks when the user tries to see some visualizations for that particular agent. In order to resolve this problem, this combination of steps as shown in figure 3.8 is used. The output values from *Group by* are the main output values of every metric calculation. However, values for some agents would be missing as mentioned earlier. In order to resolve this issue, distinct org. elements are retrieved from the database and then *Merge join* operation is done with output elements of *Group by* step. As a result, new rows will be added for each missing agent. Following this, some values which are the main result of metrics are set to zero. Finally, it is appended to the main results and then sent to further steps. Now that we have all the agents in the result, it prevents webclient from collapsing when the user tries to see visualizations for the missing agent.

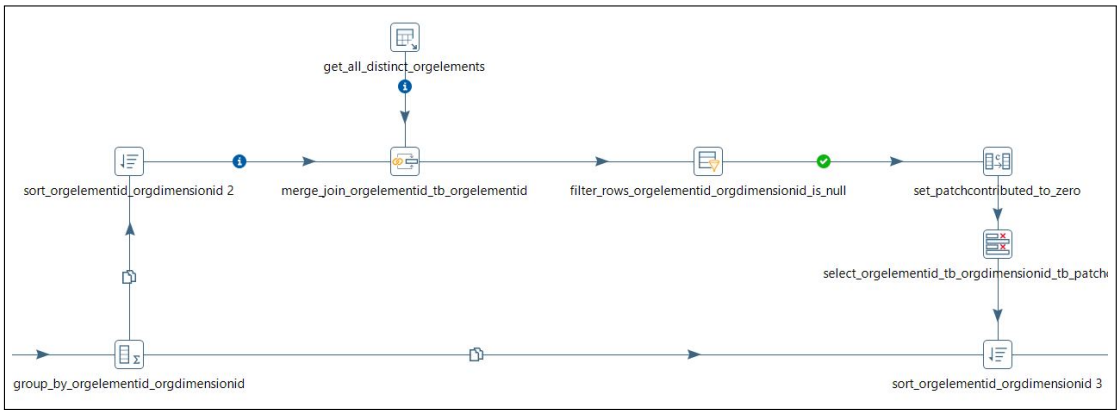
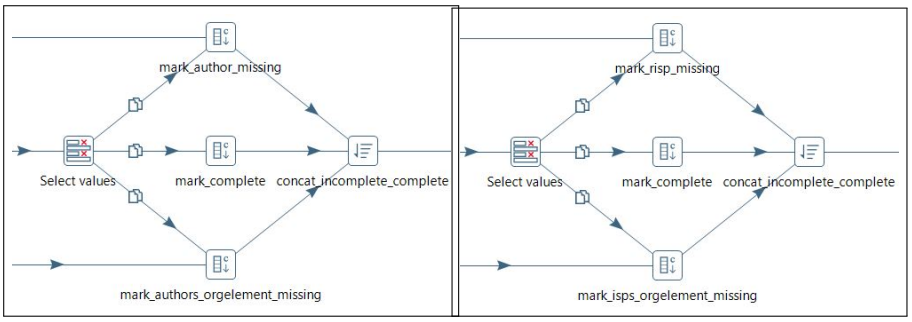


Figure 3.8: Steps to append missing agent

Appending Complete and Incomplete data



(a) Missing Author **(b)** Missing ISP

Figure 3.9: Steps to append complete and incomplete data

In order to analyse the data quality of the code contribution, it is required to visualize incomplete and complete data separately. The code contribution data is considered to be incomplete if the following cases occur.

- Author missing
- Author's org. element missing
- ISP missing
- ISP's org. element missing

As shown in figure 3.9, combination of steps shown on left(a) is used to append missing author and missing authors orgelement to complete data. And the combination of steps shown on right(b) is used to append missing ISP and missing ISP's authors to complete data. As a result, the output rows from either of these combination can be grouped into categories and stored in database.

3.2 Transformation Manager

As mentioned earlier, information about hierarchical oder of organizational units were required to generate results by taking children, grandchildren and so on into the account. At first, the kettle transformation (ktr) files were implemented to collect descendants by repeatedly using the fixed number of complicated steps. For reducing all those complicated steps into a single step in ktr files, an existing interface from Pentaho data integration called *RowListener* is used. It enables CMSuite to listen to the current rows in a step at the time of execution. With its help, the hierarchical information of the current organization unit can be collected using *DescendantLoader* and then injected in that step. In the following sections, implementation of *RowListeners*, *DescendantLoader* and how it is connected to *KettleTransformer* are discussed in detail. Furthermore, the methods to update the already existing transformations are also added to this module.

3.2.1 RowListener

In order to reduce the complexity of ktr files, RowListeners are introduced in CMSuite. They are customized according to the metric calculation which is required to be performed. The two types of custom RowListeners that were created in this project are *DescendantRowListener* and *SegregatorRowListener*. They are explained in detail in following subsections.

DescendantRowListener

DescendantRowListener is responsible for reading the input org. elements for which it has to collect all the descendants and then write it to output step. This collecting descendant process is done using *DescendantLoader* which will be explained in the next section. It injects all the descendant org. elements for the incoming parent org. element.

As stated in ("Work with Rows", 2019), a row in PDI is represented by a Java object array, Object[]. Each field value is stored at an index in the row. The row array itself does not carry the meta data. The *RowMetaInterface* contains the row meta data of the current row. This interface provides methods such as *getInteger()* which can be used to reach values directly in a row. As shown in lines 5 and 6 of listing 3.1, the *getInteger()* method requires Object[](row data), String(value name) and Long(default value) as arguments. With its help, the value names *OrgElementId* and *OrgDimensionId* are identified from the row. The only rule is that the name of a value needs to be unique in a row. When a value with the same name is added twice to a row, the second occurrence will be automatically renamed(Casters et al., 2010). Therefore, it is metric designers responsibility to provide the input in right format i.e names as shown in line

5 and 6 of listing 3.1. The lines 11 to 14 are responsible for giving the list of descendants of a parent org. element read from the current row. Finally, the descendants are injected into the step as shown in lines 15 to 21.

```
1  @Override
2  public void rowReadEvent(RowMetaInterface _rowMeta, Object[] _row)
       throws KettleStepException {
3
4  try {
5  currentOrgElementId = _rowMeta.getInteger(_row, "OrgElementId", null);
6  currentOrgDimensionId = _rowMeta.getInteger(_row, "OrgDimensionId",
       null);
7  } catch (KettleException kettleException) {
8  throw new TransformerException(kettleException);
9  }
10
11 // Get DescendantIds of currentOrgElementId
12 List<Integer> currentDescendantIds = descendantLoader
13 .getDescendants(currentOrgElementId.intValue(), currentOrgDimensionId.
       intValue());
14
15 // Insert row for each descendantId found under currentOrgElementId
16 for (Integer descendantId: currentDescendantIds) {
17    insertRow(_rowMeta.clone(), new Object[]
18    {currentOrgElementId, currentOrgDimensionId,
19     descendantId.longValue()});
20 }
21
22 }
```

Listing 3.1: DescendantRowListener.java

SegregatorRowListener

SegregatorRowListener is responsible for finding out if the incoming org. element is external or internal org. element by comparing it with another comparand column in the same row. Therefore, as mentioned earlier, it is the responsibility of a metric designer to make sure the right format (column names as shown in lines 5-6 of listing 3.2) is passed to this step. It is also mandatory that input rows should have the existing column name called *is_external_isp*. It is used for carrying the boolean value, which is helpful for segregating external or internal org. element. So the process is as follows. For the given input org. element and *isp_orgelement* it checks if this *isp_orgelement* is descendant of org. element. If it is not found, then it assigns false to the field *is_external_isp* as shown in line 19. Obtaining a value by index is always the fastest way of getting values from rows(Casters

et al., 2010). Thus, the method *indexOfValue()* is used to look up the index of a field value *is_external_isp* in a row. The lines 13 to 16 is responsible for getting the descendants only if a new *isp_orgelement* is found. That is, it checks if *isp_orgelement* from the previous row is same as the one from the current row. Thus, the performance is increased by allowing the descendantLoader to collect descendants only when a new *isp_orgelement* is encountered.

```
1  @Override
2  public void rowReadEvent(RowMetaInterface _rowMeta, Object[] _row)
       throws KettleStepException {
3
4     try {
5        currentOrgElementId = _rowMeta.getInteger(_row,
6         "OrgElementId", null);
7        currentOrgDimensionId = _rowMeta.getInteger(_row,
8         "OrgDimensionId", null);
9        currentIspOrgElementId = _rowMeta.getInteger(_row,
10        "isp_orgelement", null);
11
12    } catch (KettleException kettleException) {
13       throw new TransformerException(kettleException);
14    }
15
16    if ((currentOrgElementId != previousOrgElementId) || (
       currentOrgDimensionId != previousOrgDimensionId)) {
17       currentDescendantIds = descendantLoader
18       .getDescendants(currentOrgElementId.intValue(),
       currentOrgDimensionId.intValue());
19    }
20    if ((currentOrgElementId.equals(currentIspOrgElementId))
21    || currentDescendantIds.contains(currentIspOrgElementId.intValue()))
        {
22       _row[_rowMeta.indexOfValue("is_external_isp")] = false;
23    }
24    previousOrgElementId = currentOrgElementId;
25    previousOrgDimensionId = currentOrgDimensionId;
26
27 }
```

Listing 3.2: SegregatorRowListener.java

RowListenerFactory

RowListenerFactory is responsible for creating the instance of rowlistener based on the step that was found in the transformation. As discussed in previous

chapter, specific names are given to dummy steps like *cmsuite_inject_descendants*, *cmsuite_inject_isp_segregator* and *cmsuite_inject_person_segregator*. With the help of these names, this factory class decides which instance of RowListener has to be created as shown in listing 3.3.

```
public RowListener getInstance(StepMetaDataCombi _stepMetaDataCombi) {
   if (_stepMetaDataCombi.stepname
   .equals("cmsuite_inject_descendants")) {
      return new DescendantRowListener(_stepMetaDataCombi.step,
       descendantLoader);
   }
   if (_stepMetaDataCombi.stepname
   .equals("cmsuite_inject_isp_segregator")) {
      return new IspSegregationRowListener(_stepMetaDataCombi.step,
       descendantLoader);
   }
   if (_stepMetaDataCombi.stepname
   .equals("cmsuite_inject_person_segregator")) {
      return new PersonSegregationRowListener(_stepMetaDataCombi.step,
       descendantLoader);
   }
   return null;
}
```

Listing 3.3: RowListenerFactory.java

3.2.2 DescendantLoader

DescendantLoader is responsible for collecting all the descendants of the given root node. It is implemented based on DFS(Depth First Search) method. To describe it elaborately, it starts collecting the descendants from the depth of the tree, so that finally when it reaches the root node, it will be having all the descendant nodes collected in it. One of the possible solution to get descendants from the tree is a recursive method. However, it consumes more time as it gets called every time for each input row. The recursive method has higher space requirements than the iterative method as all functions will remain in the stack until reaching the base case. It also has more time requirements because of function calls and returns overhead. In order to avoid this high overhead, the Depth-First Search method is used. It collects all the nodes and its descendants when it traverses once through the tree from leaf nodes(Depth) to the root node. As a result, it avoids the repeatedly using recursive method to collect descendants for each node. A recurring theme in data structures and algorithm design is the ability to trade space for time(Weiss Mark, 2014). Thus, DescendantLoader is created such that it uses some space for collecting descendants in a map. So it

has the advantage that tree traversing can be made faster.

Tree Structure

The two convenient ways of representing tree structure are *Adjacency List* and *Adjacency Matrix*. As the organizational units in CMSuite are modeled using the composite design pattern(Capraro et al., 2018) and the parent-child relationships between organizational units are stored as adjacency list in the database, already existing *OrgElement* objects are used as *adjacency list* for performing depth-first search tree traversal method. The figure 3.10 shows an example organization tree and its adjacency list representation of the parent-child relationship between organizational units. As described in (Jan Pahl & Damrath, 2012), a directed graph G=(V,E) is defined as set V of vertices and a set E of edges. It is suitable for describing relationships between the vertices. The relationships between the vertices are called edges.

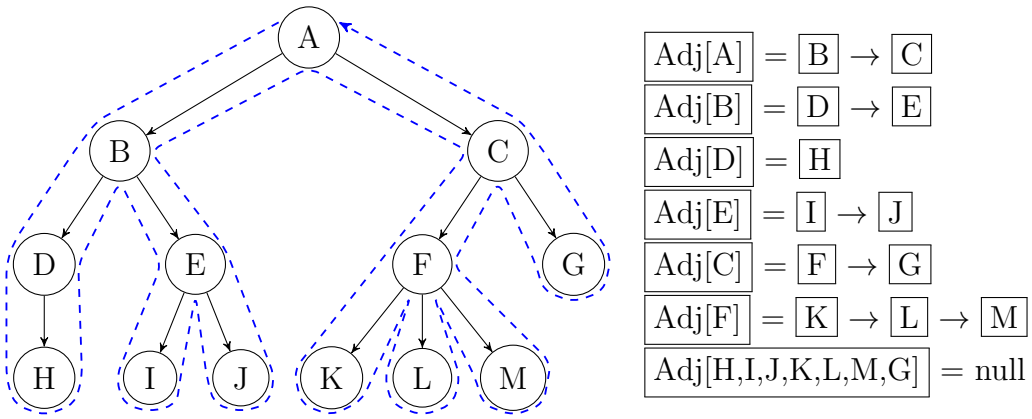


Figure 3.10: Depth-First Search Tree Traversal

Tree Traversal

As described in (Khuller & Raghavachari, 2014), DFS is a fundamental tree traversal technique. The algorithm first initializes all vertices of the graph as being unvisited. Processing of the graph starts from an arbitrary vertex, known as root vertex. Each vertex is processed when it is first discovered (also referred to as visiting a vertex). It is first marked as visited, and its adjacency list is then scanned for unvisited vertices. The process is repeated until all the descendants are visited, as shown in algorithm 1. A vertex *n* is processed as soon as it is encountered, and therefore at start of depth first search, *visited[n]* is false. Since *visited[u]* is set to true as soon as DFS starts execution, each vertex is visited only once. DFS processes each edge of the graph exactly twice, once from each of its incident vertices. Since the algorithm spends constant time processing each

edge of tree, it runs in *O(—V—+—E—)* time (Khuller & Raghavachari, 2014).

Algorithm 1: Depth-First Search

Data: N := all nodes in a tree. *Adj* := Adjacency List.
Result: List of all descendant nodes of input node n

```
1  Start with any n ∈ N;
2  for n ∈ N do
3  |  visited[n] = false;
4  end
5  S = EmptyStack;
6  S.push(n);
7  while Stack is not empty do
8  |  Take top node u from S;
9  |  visited[u] = true;
10 |  Mark visited all children of u as true;
11 |  for w in Adj[u] do
12 |  |  if not visited[w] then
13 |  |  |  S.push(w);
14 |  |  |  visited[w] = true;
15 |  |  |  Mark visited all children of u as false;
16 |  |  |  break;
17 |  |  end
18 |  end
19 |  if visited all children of u then
20 |  |  remove top element u using stack.pop();
21 |  |  collect u in descendant list;
22 |  end
23 end
```

As shown in listing 3.4, the method traverse() accepts the root node element and its dimension as input arguments. It traverses once from depth of the tree to the root. At the end of this tree traversal process, it will be having all the nodes with its descendants in memory. Thus, this collected information is used by rowlisteners to inject required details, as mentioned earlier.

The process is as follows:

1. Begin processing at root orgElement (line 1).
2. Initialize *visited* and *stack*(lines 3-6).
3. Select the org. element that is at top of stack (elements in stack are inserted and removed according to the last-in first-out (LIFO) principle) (lines 10-11).

4. Mark this orgElement as visited(line 13).
5. Push one child of this org. element to stack and mark it as visited.(Only one child is considered here in order to move till depth of the tree)(lines 22-23)
6. If all the children of this org. element are visited or a leaf node is found, pop the orgElement from stack. Also, store this org. element with its children in a map, so that it can be added as descendants when parents are getting processed.(lines 29-39)
7. Repeat the above steps from 3 to 6 until the stack gets empty(line 8).(Repeating the process until it visits all the children of a node is called backtracking)

```
1  public void traverse(OrgElement _rootNode, OrgDimension _dim) {
2
3     Boolean[] visited = new Boolean[adjacentList.size()];
4     Arrays.fill(visited, false);
5     Stack<OrgElement> stack = new Stack<OrgElement>();
6     stack.push(_rootNode);
7
8     while (!stack.isEmpty()) {
9
10       OrgElement parentNode = stack.pop();
11       stack.push(parentNode);
12       int parentIndex = adjacentList.indexOf(parentNode);
13       visited[parentIndex] = true;
14
15       OrgElement currentOrgElement = orgElementDao
16       .getRichWithChildren(parentNode.getId());
17       Set<OrgElement> currentChildren = currentOrgElement
18       .getChildren(_dim , null);
19       Boolean visitedChildren = true;
20
21       for (OrgElement childNode : currentChildren) {
22          int childIndex = adjacentList.indexOf(childNode);
23          if (!visited[childIndex]) {
24             stack.push(childNode);
25             visited[childIndex] = true;
26             visitedChildren = false;
27             break;
28          }
29       }
30
31       if (visitedChildren) {
```

```
32          stack.pop();
33
34          tree.computeIfAbsent(_dim.getId(), value ->
35             new HashMap<Integer, List<Integer>>())
36          .put(currentOrgElement.getId(),
37             new ArrayList<>(getChildrenIds(currentOrgElement, _dim)));
38
39          for (OrgElement child: currentChildren) {
40             List<Integer> descendants = tree.get(_dim.getId())
41             .get(child.getId());
42             tree.get(_dim.getId()).get(currentOrgElement.getId())
43             .addAll(descendants);
44          }
45       }
46    }
47 }
```

Listing 3.4: DescendantLoader.java

3.2.3 KettleTransformer

KettleTransformer is used for executing the transformations that are stored in the database. For this project, it is modified to attach a *RowListener* object to a step copy for obtaining the data from that step. When a *RowListener* is attached, CMSuite gets notified when a row is read or written, or when a row is written to an error handling step. This notification happens in sync with the execution of the transformation. This allows CMSuite to handle the records in a streaming fashion, as same as how it is handled in the spoon.

The changes are implemented in KettleTransformer class. It creates a TransMeta object from the InputStream, that comes from the opened transformation file. This object will then be used by the Kettle engine to run the transformation(Daeubler, 2017). Prior to this thesis, transformations were executed using *trans.execute()* method. Now it is replaced with *trans.prepareExecution()* method to allow addition of rowlisteners before actual execution is started using *trans.startThreads()* method. The *trans.prepareExecution()* method prepares the transformation for execution. This includes setting the arguments and parameters as well as preparing and tracking the steps and hops in the transformation("Trans pentaho javadoc", 2019).

In addition to this previous implementation, now rowlisteners are created using *RowListenerFactory* and then attached to the particular step by iterating over all the steps in that transformation. RowListenerFactory creates a suitable instance based on the steps that are defined with particular names as mentioned before.

As shown in line 6 of listing 3.5, the method *trans.getSteps()* returns all the steps as a list of *StepMetaDataCombi* instance. This instance returns a *StepInterface* by calling *StepMetaDataCombi.step*. This interface handles the actual execution of the functionality described in the metadata (Casters et al., 2010). Finally, an instance of row listener is attached to step using method *addRowListener()* of stepInterface ("StepInterface pentaho javadoc", 2019). This idea of attaching row listener for reading data from step is obtained from ("Executing a PDI Transformation", 2019).

```
1  Trans trans = new Trans(transMeta);
2  // Preparing the execution of the transformation (instead of simply
       executing)
3  trans.prepareExecution(null);
4
5  // Iterate over steps to add rowListener
6  for (StepMetaDataCombi stepMetaDataCombi : trans.getSteps()) {
7     RowListener rowListener = factory.getInstance(stepMetaDataCombi);
8     if (rowListener != null) {
9        stepMetaDataCombi.step.addRowListener(rowListener);
10    }
11 }
12
13 // Starting the threads once preparation is done
14 trans.startThreads();
15
16 trans.waitUntilFinished();
```

Listing 3.5: KettleTransformer.java

3.3 Analysis-Result-Provider

As mentioned earlier, *AnalysisResultProvider* is responsible for retrieving the results based on type of result that we chose while saving transformation. Prior to this thesis, this module had only three result types. Now it is extended to have *CategorizedTimeSeries* and *GroupedCategorizedValue* result type also. While the types of required result were decided, it was unclear what are the datatypes that should be chosen for attribute of these results. Easiest possible solution is to use the nested map types. Since the key and value are just primitive types, memory issues will not be caused by the nested map. Therefore, based on the values that are needed to represent these results, the corresponding map types are chosen.

Categorized Time Series Result

- *Attribute type* : `Map<String, SortedMap<Date, Integer>>`
- *Definition* : `Map<Category, SortedMap<Date, Value>>`
- *Example* : values: {Category1:{2018-11: 34, 2018-12: 25},
  Category2:{2018-09: 26, 2018-10: 34}
  }

Grouped Categorized Value Result

- *Attribute type* : `Map<String, Map<String, Integer>>`
- *Definition* : `Map<Category, Map<Name, Value>>`
- *Example* : values: {ISP1:{Level-1: 34, Level-2: 25},
  ISP2:{Level-1: 26, Level-2: 34}
  }

Apart from this inclusion, methods related to time series lacked functionality to fill the missing time period for which values has to be zero. For this reason, changes has been done so that it fills the missing keys(date) with values as zero. Listing 3.6 shows the code snippet which fills the missing date in Categorized-TimeSeriesResult.

The code snippet works like this.

- Gets all the childValues in a SortedMap to find start date and end date (line 1-3).
- Gets all months between start and end date (line 5-6).
- Fills the missing months for all categories. For each category(key) in a map, it sets the value filled with missing date value pairs (line 7-9).
- The method *doFillMissingDates* accepts the list of dates between start and end date and a map of date value pair. It returns the map with missing dates included in it (line 11-22).
- For each datesInRange it checks if date is present in a map's key. If not present then it puts that date(key) with zero(value) in map (line 14-18).

```
1
2 SortedMap<Date, Integer> allChildValues = new TreeMap<Date, Integer>();
3 values.values().forEach(value -> allChildValues.putAll(value));
4
5 List<Date> datesInRange = DateUtil.getMonthsBetween(allChildValues.
    firstKey(), allChildValues.lastKey());
6
```

```
values.entrySet().forEach(
   category -> category.setValue(doFillMissingDates(datesInRange,
    category.getValue()))
);

public SortedMap<Date, Integer> doFillMissingDates(List<Date>
    _datesInRange, SortedMap<Date, Integer> _values) {

   _datesInRange.forEach(
   date -> { if (!_values.containsKey(date)) {
   _values.put(date, 0);
   }
});

return  _values;

}
```

Listing 3.6: AnalysisResultFactory.java

3.4 Web-client

3.4.1 Analysis-Result-Module

The components in *analysisresult-module* of webclient are responsible for converting the retrieved result into the format that can be used for displaying visualizations. For these visualizations, ngx-charts are used. As discussed in previous sections, the same data type is used in the models of newly introduced result types in webclient. However, the chosen data type does not match the types required by the ngx-charts. Therefore, the retrieved key value pair data should be converted into another format that can be used as input for charts.

Listing below shows the example data format that is required by line chart of ngx-charts. It requires field name(string) and series[ ] which is an array of value(number) and name(string). It is clear that, the datatype of result retrieved from CMSuite's server does not match the format shown below. Thus, the retrieved data is required to be converted into format required by ngx-charts.

```
{
   "name": "Category1",
   "series": [
      {"value": 34, "name": "2018-11"},
      {"value": 25, "name": "2018-12"}
   ]
```

```
},
{
   "name": "Category2",
   "series": [
      {"value": 26, "name": "2018-09"},
      {"value": 34, "name": "2018-10"}
   ]
}
```

Listing 3.7: Format required by ngx-charts

As shown in listing 3.7 , the *convertData()* method uses the retrieved key value pairs to collect them in chartData in different format. At first, the fields are initialized as shown in lines 2-4. Afterwards, for each category found in retreived result, the values are stored in chartData as *name* and *value* as shown in line 6-14. Initially, the name(category) with series(empty array) is pushed inside chart data. Following this, for each child map name(Date) and value(Integer) is pushed inside the chart data. The captured index in the forEach function of parentdata is used here to identify for which parentData the child data should be pushed. Finally, the format as shown above is created inside the chartData. Thus, the result data from server are converted to another format.

For some of the selected organizational unit, it is possible that they have no code contributions received or contributed. Such cases are already handled inside the kettle transformations as shown in section 3.1.2. However, webclient has to be in sync with these manually appended result in ktr files. For this reason, the conditions to check if chart data is valid or not is performed. It checks if result has only one category and only one key and value is zero. If all of them are true, then result is set as invalid i.e hasValidResult is set as false. With the help of this variable, the template of this component is informed to show the alert message "Nothing to see" in the tiles instead of blank chart. Furthermore, predefined colors are set for this chart.

```
private convertData(): void {
   let parentData = this.result.values;
   let chartData: any[] = [];
   let hasValidResult: boolean = true;

   Object.keys(parentData).forEach(function(parentKey, index): void {
   chartData.push({name: parentKey, series: []});
   let data = parentData[parentKey];
   Object.keys(data).forEach(function(key): void {
      chartData[index].series.push(
         {'name': new Date(key),
         'value': data[key]});
```

```
13          }
14       )
15    });
16    this.chartData = chartData;
17    if ((this.chartData.length === 1) // Has only one category
18    && (this.chartData[0].series.length === 1) // Has only one key(month
      )
19    && (this.chartData[0].series[0].value === 0)) { // Has value zero
20       hasValidResult = false;
21    }
22    this.colorPaletteService.getColorPalette().forEach((col) => {
23       this.colorScheme.domain.push(col.hexValue)
24    });
25    this.hasValidResult = hasValidResult;
26 }
```

Listing 3.8: categorized-time-series-result.component.ts

Similar to this component, other components like *TimeSeriesResultComponent*, *GroupedCategorizedValueResultComponent* and *CategorizedValueResultComponent* are also modified to use ngx-charts instead of using previously implemented chart.js. It converts the retrieved result data into data that can be fed to ngx-charts.

3.4.2 Transformation-Manager-Module

The transformation manager module is responsible for uploading and executing the transformations. Prior to this thesis, the components like *TransformationListComponent*, *TransformationRunComponent* and TransformationCreationComponent existed. The requirement 2 states that the transformations should be able to be updated. Thus, the already existing transformationCreationComponent is modified to have functions to update the transformations.

As the name suggests, *transformationCreationComponent* is used for creating the new transformations by entering the name, resulttype, agenttype and uploading file in the existing form. While the update functionality has to be added somewhere in transformation manager, the question arose if the creation component can be modified to use it for updating. As all the required variables and template were already existing, this idea of modifying the creation component to include update functions was chosen.

As shown in listing below the *TransformationCreationComponent* is renamed into *TransformationDetailComponent*. The basic functions are split to be used by both edit and create functions as shown in line 32-40. The form fields used are same for both functionality. Thus, the logic related to form handling are used as

a common method. Please note that listing 3.9 omits certain lines of code that did not undergo any changes, as they are not relevant for this thesis. When the life cycle hook method *ngOnInit()* is triggered, it calls the *getTransformation()* which is used for placing the values in the *Form* to make it available for editing. This is achieved by using the inbuilt function *FormGroup.patchValue()*. However, the KTR file retrieved in the format of *bytearray* is not sufficient for making it downloadable. To resolve this, the bytearray are converted to Uint8Array format so that they can be used to construct a Blob object. As ktr file is basically a xml file, the type *application/xml* is chosen. Finally, the File object is created using this Blob. Now it can be downloaded using the method getDownloadUrl() as shown in line 43-47.

```
1  export class TransformationDetailComponent implements OnInit {
2
3  @Input('create-mode')
4  isInCreateMode: boolean = false;
5
6  uploadForm: FormGroup;
7  submitted: boolean;
8  isLoaded: boolean = false;
9  transformationPromise: Promise<Transformation>;
10 oldFile: File;
11
12
13 ngOnInit(): void {
14    if (!this.isInCreateMode) {
15    this.getTransformation(this.transId);
16    }
17 }
18
19 getTransformation(id: number): void {
20    this.transformationPromise = this.transformationService.getById(id);
21    this.transformationPromise.then(transformation => {
22    this.uploadForm.patchValue(transformation);
23    let bytes = new Uint8Array(transformation.ktrFile);
24    let blob = new Blob([bytes], {type: 'application/xml'});
25    this.oldFile = new File([blob], transformation.token + '.ktr');
26    this.isLoaded = true;
27    }
28    );
29
30 }
31
32 onSubmit(): void {
33    this.submitted = true;
```

```
34      let formData = new FormData();
35      formData = this.appendFormData(formData);
36      if (this.isInCreateMode) {
37      this.uploadTransformation(formData);
38      } else {
39      this.updateTransformation(formData);
40      }
41  }
42
43  getDownloadUrl(): any {
44      return this.sanitizer
45      .bypassSecurityTrustResourceUrl(window.URL
46      .createObjectURL(this.oldFile));
47  }
48
49  }
```

Listing 3.9: transformation-detail.component.ts

4 Results

4.1 Metric Visualizations

Metric visualizations are categorized based on type of result that were stored in the database. In this chapter, different kinds of metric visualizations that were implemented will be discussed. Before discussing about the different visualizations, it is important to understand the Patch-Flow Phenomenon. Below are the following concepts defined by (Capraro et al., 2018).

- A code contribution is any code change performed on a software component.
- A patch is a code contribution made by a developer who is external to a project.
- Patch-flow is the flow of patches across organizational boundaries such as project or organizational unit boundaries within a company.

4.1.1 Single Value Result

Following table shows how single value results generated by the metrics are mapped to columns in database.

Table 4.1: Mapping of metric fields to resultrow table

Output Fields from Metric	Columns of ResultRow Table
TRANSFORMATIONRUN_ID	transformationrun_id
TRANSFORMATION_ID	transformation_id
orgelementid	orgelement_id
orgdimensionid	orgdimension_id
patchcontributed	value

Code Contributions made(Excl. Internal)

This metric shows us how many contributions were made by an organizational unit. Of course, this metric is applied for all available organizational units. When

designing the transformation for this metric, a question arose, if the code contributions made to ISPs that are owned by the unit itself should be considered or not. In order to show the difference between internal and external, visualizations are split into two types. Including Internal and Excluding Internal. This metric shows us the code contributions made by organization unit excluding internal contributions i.e patches. As this metric just sums up all the code contributions made, This metric is visualized as a single value result.

Code Contributions made(Incl. Internal)

In contrast to previous metric, this metric considers the internal code contributions also. It adds up all the code contributions made by an organization unit without excluding internal contributions. That is, code contributed to the ISPs that is owned by this organizational unit is also considered.

4.1.2 Time Series Result

Code Contributions made per Month(Excl. Internal)

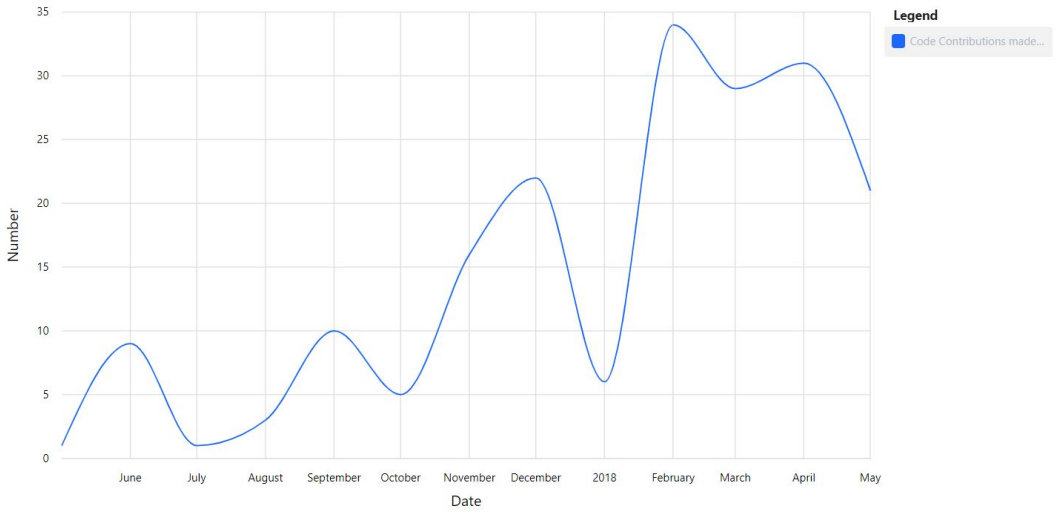


Figure 4.1: Code Contributions made per month(Excl Internal)

This metric shows the number of code contributions made by an organizational unit for each month. As mentioned before, here it is calculated by excluding internal contributions. Since the internal contributions are excluded in calculations, this metric can also be called as patches contributed per month. With the help of this metric, the user can find if the organizational unit's contribution is increasing or decreasing over time. As shown in figure 4.1 , the x-axis presents time (i.e months) and the y-axis presents number of code contributions

or patches. For example, by looking at the figure 4.1, we can say that more than 33 patches were contributed by a selected org. element in Feb-2018.

Table 4.2: Mapping of metric fields to resultrow table

Output Fields from Metric	Columns of ResultRow Table
TRANSFORMATIONRUN_ID	transformationrun_id
TRANSFORMATION_ID	transformation_id
orgelementid	orgelement_id
orgdimensionid	orgdimension_id
patchcontributed	value
commit_date	key

Code Contributions made per Month(Incl. Internal)

This metric is calculated as same as the previous one except that the internal contributions are also taken into consideration. This metric is of interest to users who want to see the pattern of data not only for code contributions that are flowing to external but also to internal ISPs. As shown in figure 4.2, the number of code contributions are more than what was found in figure 4.1. That is, code contribution per month is huge in contrast to the previous metric because internal contributions are also considered. For example, by looking at the figure 4.2, one can find that, the total code contributions made per month by selected org. element has declined over the period.

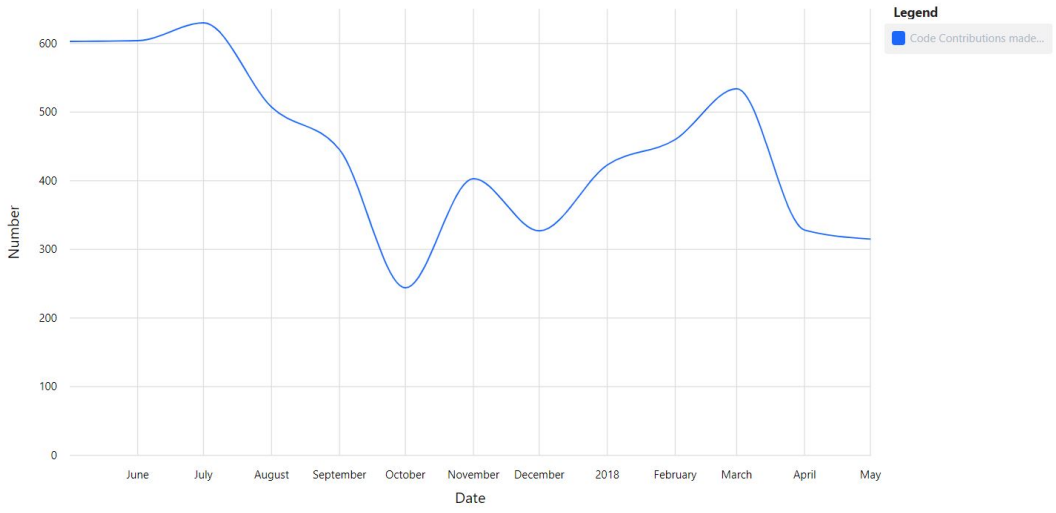


Figure 4.2: Code Contributions made per month(Incl Internal)

Table 4.3: Mapping of metric fields to resultrow table

Output Fields from Metric	Columns of ResultRow Table
TRANSFORMATIONRUN_ID	transformationrun_id
TRANSFORMATION_ID	transformation_id
orgelementid	orgelement_id
orgdimensionid	orgdimension_id
patchcontributed	value
commit_date	key

First Time Contributors per Month

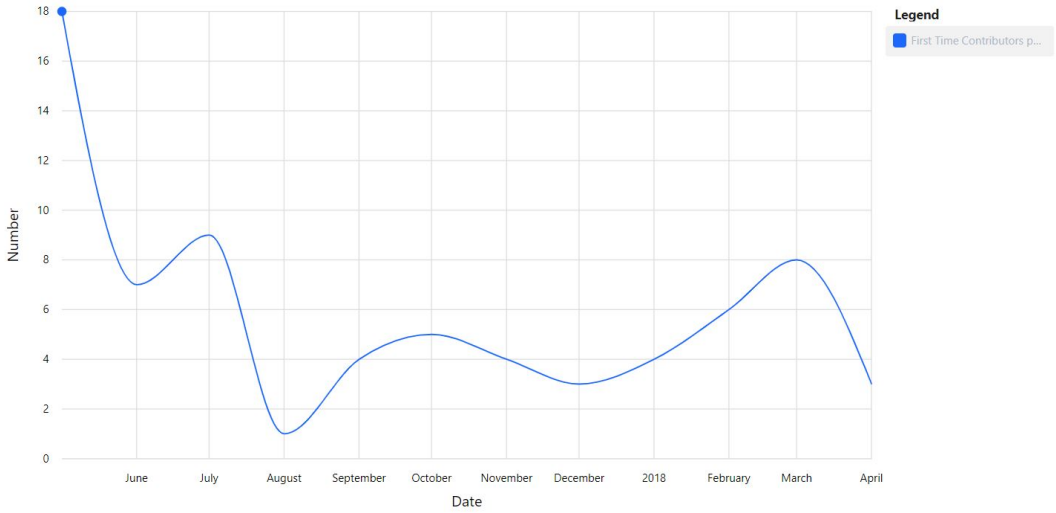


Figure 4.3: First Time Contributors per Month

Table 4.4: Mapping of metric fields to resultrow table

Output Fields from Metric	Columns of ResultRow Table
TRANSFORMATIONRUN_ID	transformationrun_id
TRANSFORMATION_ID	transformation_id
orgelementid	orgelement_id
orgdimensionid	orgdimension_id
contributors	value
commit_date	key

This metric shows us the number of authors who made their first contribution in a month. It is calculated by finding the least commit date from the list of

contributions made by an author. Of course, all persons of the selected organization unit will be taken into account for this calculation. Here there is no need for segregating contributions as internal or external because the main purpose is just to find the first time contributors irrespective of which project does that contribution goes to. As shown in figure 4.3, y-axis presents the number of authors and x-axis presents time(months). By looking at the figure 4.3, one could observe that the number of first-time contributors of a selected org. element has decreased over time.

Persons Involved per Month

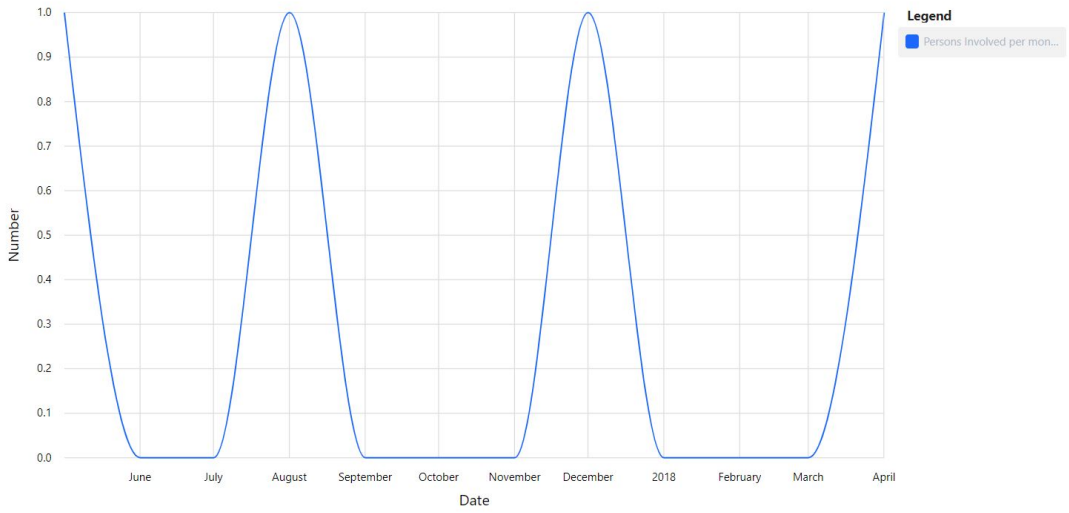


Figure 4.4: Persons involved per Month

Table 4.5: Mapping of metric fields to resultrow table

Output Fields from Metric	Columns of ResultRow Table
TRANSFORMATIONRUN_ID	transformationrun_id
TRANSFORMATION_ID	transformation_id
receiving_innersourceproject_id	innersourceproject_id
orgdimensionid	orgdimension_id
persons	value
commit_date	key

This metric helps the user to find how many persons are involved in a particular inner source project. This metric applies to all the inner source projects i.e when the user picks an inner source project from a tree menu, this visualization will be

shown. As shown in figure 4.4, the x-axis presents time in months and the y-axis presents the number of persons who contributed to this project. This metric is calculated irrespective of internal or external persons. That means, it simply adds up all the persons from whom the code contributions are received.

Code Contributions received per Month

This metric is implemented to show how many code contributions have been received by an inner source project per month. It helps the user to see the development activity of the selected ISP. So that he can estimate the ISP's survivability. Moreover, it helps in identifying if the ISP is still alive. It is calculated by grouping the patches by month and then adding up all the patches that were received by an inner source project for that month. By looking at the figure 4.5, one can find that the selected ISP had been inactive during the last three months of the year 2017.

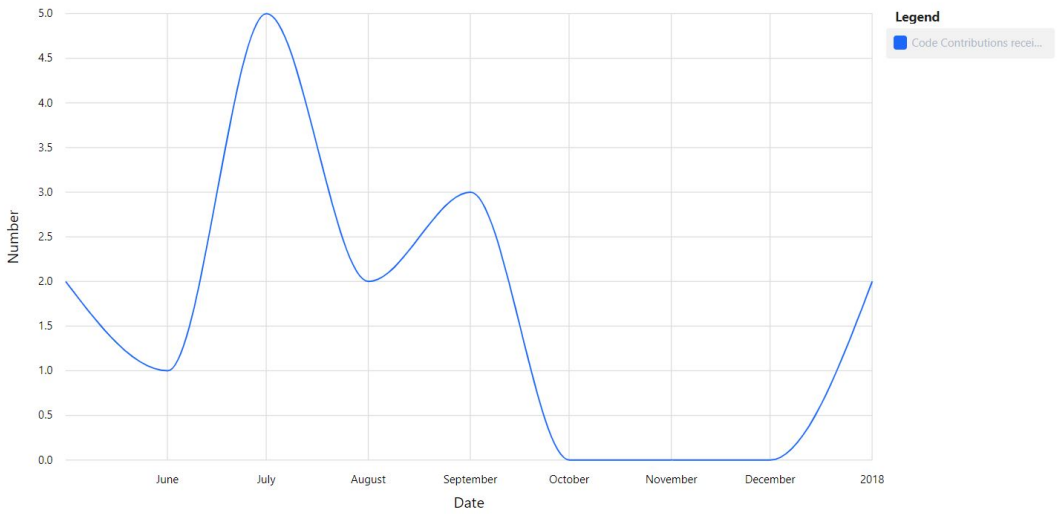


Figure 4.5: Code Contributions received per month

Table 4.6: Mapping of metric fields to resultrow table

Output Fields from Metric	Columns of ResultRow Table
TRANSFORMATIONRUN_ID	transformationrun_id
TRANSFORMATION_ID	transformation_id
receiving_innersourceproject_id	innersourceproject_id
orgdimensionid	orgdimension_id
patchcontri	value
commit_date	key

Last Time Contributors per Month

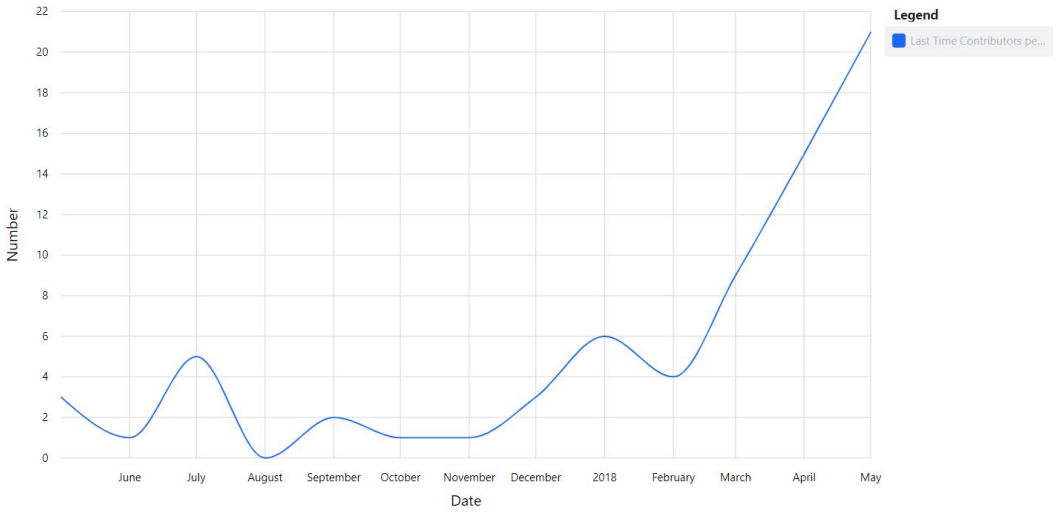


Figure 4.6: Last Time Contributors per Month

This metric is implemented to visualize the number of persons who had made their last contribution in a month. With this visualization, users can find how many persons have stopped contributing in a particular month. As shown in figure 4.6, for the selected org. element the number of persons who had made their last contribution has increased over time. That means they haven't contributed after a certain month. If they contribute later, then this number will be reduced from this month. It is calculated by sorting the patches in descending order based on commit date and then the first one is picked from that row which is basically the last contribution from that particular person. Once this picking process is carried out for all the persons, it is grouped with respect to the month to find the number of persons.

Table 4.7: Mapping of metric fields to resultrow table

Output Fields from Metric	Columns of ResultRow Table
TRANSFORMATIONRUN_ID	transformationrun_id
TRANSFORMATION_ID	transformation_id
orgelementid	orgelement_id
orgdimensionid	orgdimension_id
contributors	value
commit_date	key

4.1.3 Categorized Time Series Result

To get detailed information about code contributions, it is not sufficient to simply count code contributions over time. For this reason, here the time series results are categorized further to identify and compare the pattern in the data. Thus, in the following charts, the lines with different colors will be displayed.

Code Contributions made per Month by Namespace(Excl. Internal)

This metric groups the number of code contributions based on the namespace that received it. Furthermore, the internal contributions are excluded in this metric in order to visualize only the code contributions that were made to external namespaces. Here the namespace indicates the name of one or more ISP's code repository. This metric helps the user to make comparisons between code contributed by a selected organizational unit to different namespaces.

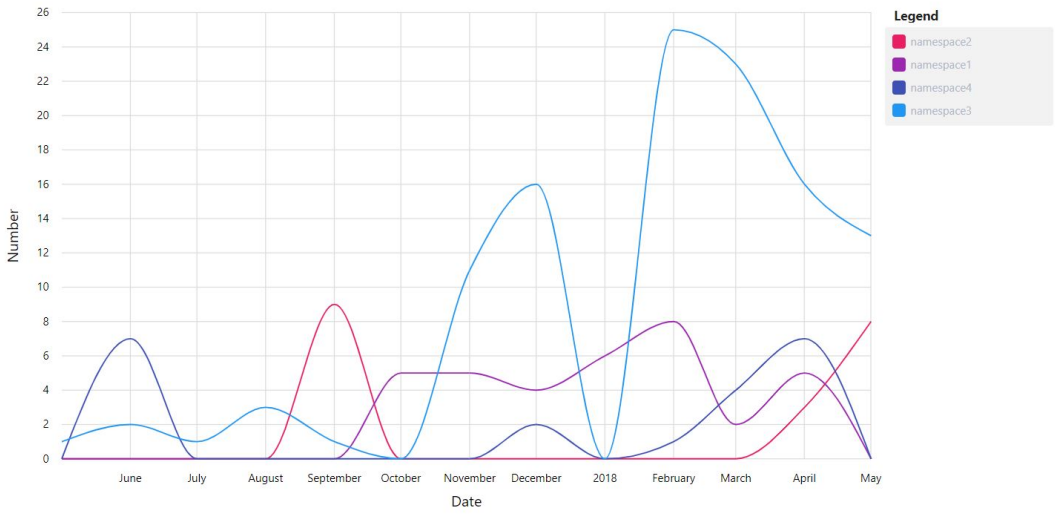


Figure 4.7: Code Contributions made per month by Namespace(Excl. Internal)

Table 4.8: Mapping of metric fields to resultrow table

Output Fields from Metric	Columns of ResultRow Table
TRANSFORMATIONRUN_ID	transformationrun_id
TRANSFORMATION_ID	transformation_id
orgelementid	orgelement_id
orgdimensionid	orgdimension_id
patchcontributed	value
commit_date	key
namespace	label

For calculating this metric, namespace details are taken from the inner source project table.

In the original data, the data has the following format:

- repository location:

  http://github.faucompany.de/api/v3/repos/[namespace]/[name]

- inner source project name: [namespace]/[name]

Since it is easier to extract the namespace from inner source project name, the name column is chosen instead of repository location of inner source project.

Code Contributions made per Month by Namespace(Incl. Internal)

This metric is same as previous one except that internal contributions are also taken into consideration. That is, it adds up all the code contributions made by selected organizational unit to the namespace of internal projects also. As shown in figure 4.8, it shows us the variation between number of code contributions made to different namespaces. This metric is of interest to users who want to see the pattern of data not only for code contributions that are flowing to external but also to internal ISPs. As shown in figure 4.8, the number of code contributions are more than what was found in figure 4.7. That is, code contribution per month is huge in contrast to the previous metric because internal contributions are also considered.

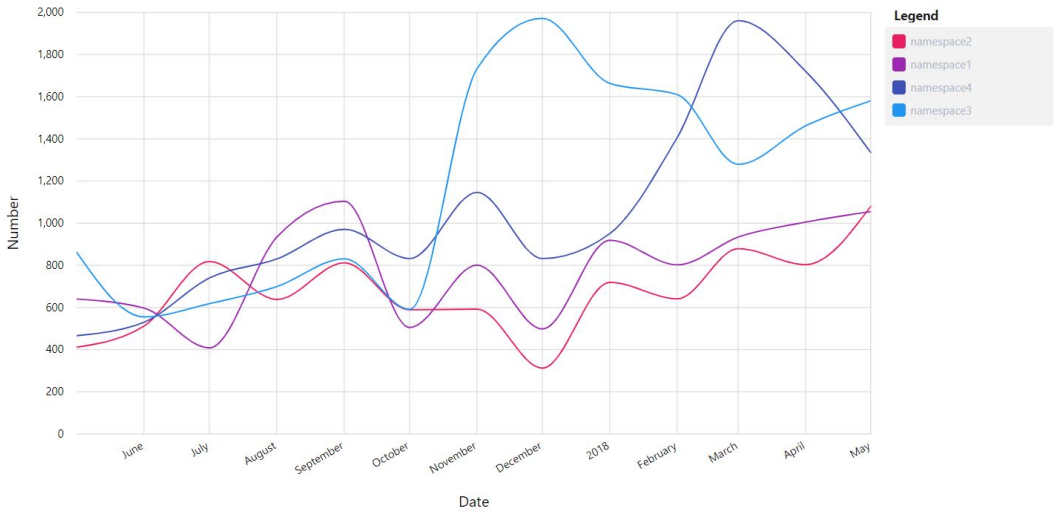


Figure 4.8: Code Contributions made per month by Namespace(Incl. Internal)

Table 4.9: Mapping of metric fields to resultrow table

Output Fields from Metric	Columns of ResultRow Table
TRANSFORMATIONRUN_ID	transformationrun_id
TRANSFORMATION_ID	transformation_id
orgelementid	orgelement_id
orgdimensionid	orgdimension_id
patchcontributed	value
commit_date	key
namespace	label

Patch-Flow by Org. Level

When measuring the patch-flow, it is not sufficient to simply count patches over time. One must address the organizational structure contextual to the patch(Capraro et al., 2018). Therefore, this metric is implemented to calculate the number of patches contributed by Org. elements per month and categorized it based on highest orglevel crossed by the patches. As shown in figure 4.9, x-axis presents the time and y-axis displays the number of patches contributed by a selected orgelement. Each color indicates the highest level crossed by the patches.

The concept of levels is applied to the organizational hierarchy in order to distinguish patch-flow between organizational units of different granularity. In a tree, each node's level is n+1 with n being its parent's level. The level 0 is always the root node. The organization itself has level 0 translated into organizational hierarchies and its top organizational units have level 1. Their children units have level 2 etc. Level 0 is considered the highest level.

(Capraro et al., 2018) defined the following terms:

- The lowest common ancestor (LCA) of two organizational units is the lowest node that has both as descendants.
- Patch-flow crosses level n if and only if `n < nlca` with nlca being the level of the lowest common ancestor of the contributing and the receiving organizational unit.

Code contributions between descendants and ancestors are not considered as patch-flow. Patch-flow crossing level n always crosses level n+1. The highest level crossed of a patch-flow (n=nlca-1) serves as a metric for the distance between the two involved organizational units.

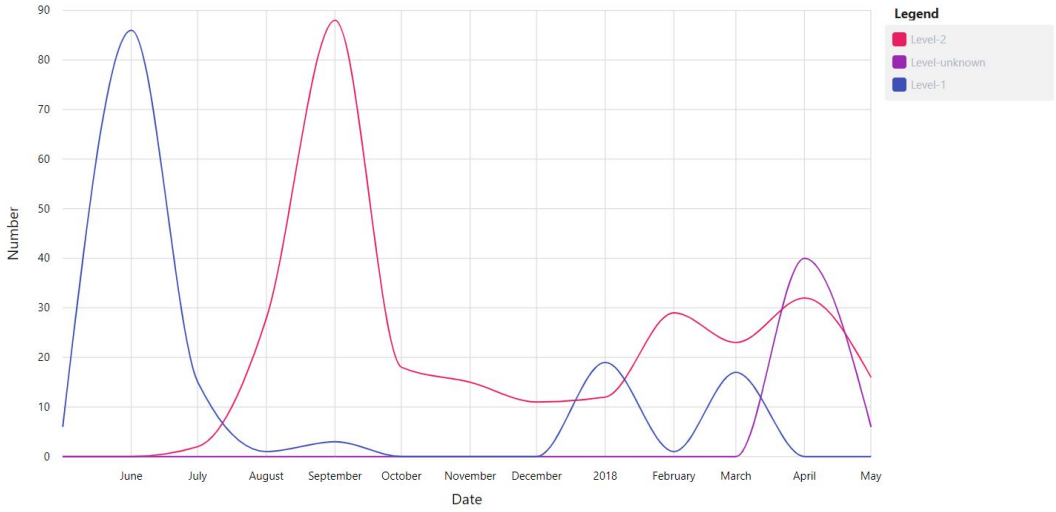


Figure 4.9: Patch-Flow by Org. Level

Table 4.10: Mapping of metric fields to resultrow table

Output Fields from Metric	Columns of ResultRow Table
TRANSFORMATIONRUN_ID	transformationrun_id
TRANSFORMATION_ID	transformation_id
orgelementid	orgelement_id
orgdimensionid	orgdimension_id
patchcontributed	value
commit_date	key
maxlevelcrossed	label

Data Completeness of Received Contributions Over Time

This metric calculates the number of code contributions received by a selected orgelement per month and then the results are categorized based on data quality. That is, it categorizes the contributions based on complete and incomplete data. In this metric , the missing author and missing author's org. element is considered as an incomplete data. This visualization helps user to analyse how good and reliable the data is, so that he can investigate where he can manually tweak the data. It is displayed on dashboard for all the org. elements including root org. element. In figure 4.11, the red line presents missing author's org. element, the purple line indicates complete data and blue line presents missing author. By observing that the red line is close to the purple line, once can fix all the missing author's org. element so that both lines merge together. Similarly, this metric can be used to fix the missing authors data also. If all three lines are merged

together then it means that all the data that we have for the selected org. element is complete.

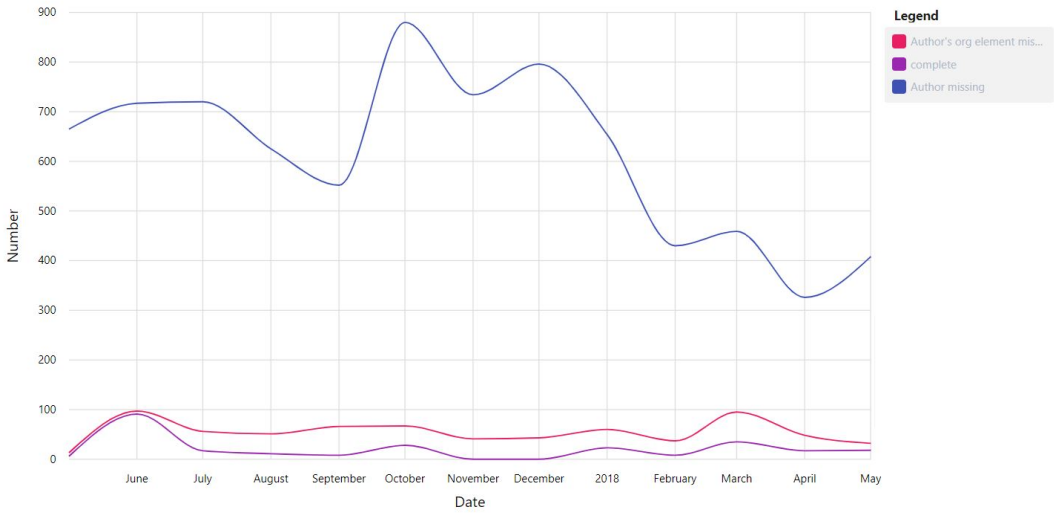


Figure 4.10: Data Completeness of Received Contributions Over Time

Table 4.11: Mapping of metric fields to resultrow table

Output Fields from Metric	Columns of ResultRow Table
TRANSFORMATIONRUN_ID	transformationrun_id
TRANSFORMATION_ID	transformation_id
orgelementid	orgelement_id
orgdimensionid	orgdimension_id
patchreceived	value
commit_date	key
category	label

Data Completeness of Contributed Contributions Over Time

In contrast to previous metric, here the data quality analysis is carried out on code contributions made by selected org. element. In this metric, data is categorized into complete, missing innersource project and missing inner source project's org. element. With this result, user can analyse how good the data is. This metric is applied for all the org. elements including root org. element. In figure 4.11, red indicates missing inner source project, purple shows inner source project's org. element is missing and the blue line presents complete data. As mentioned before, if all the three lines merge together then the data we have for a selected org. element is complete. For example, in this figure, the red line and the blue

line is merged. It means that there is no missing inner source project for the selected org. element. However, on september 2017 we can see that for few code contributions the inner source projects doesnt have org. element associated to it.

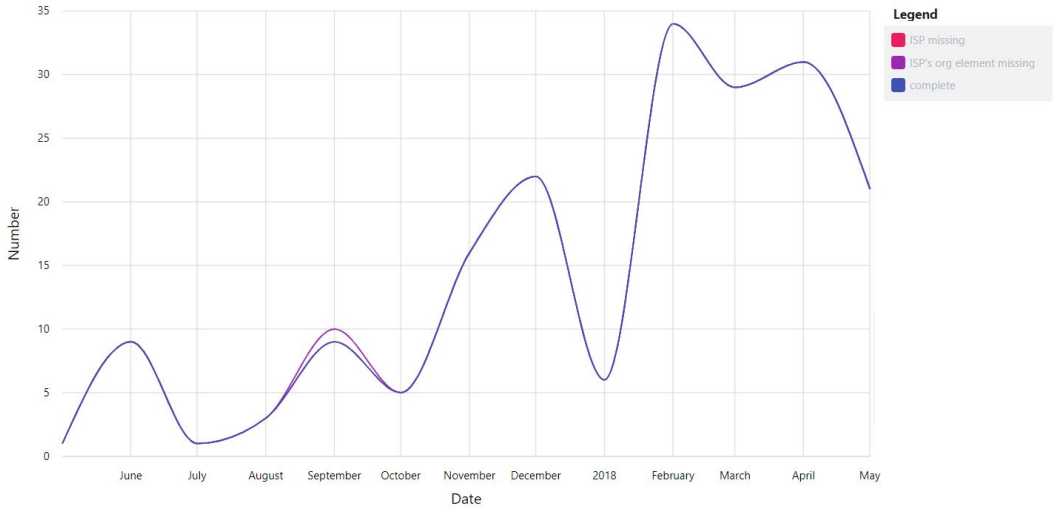


Figure 4.11: Data Completeness of Contributed Contributions Over Time

Table 4.12: Mapping of metric fields to resultrow table

Output Fields from Metric	Columns of ResultRow Table
TRANSFORMATIONRUN_ID	transformationrun_id
TRANSFORMATION_ID	transformation_id
orgelementid	orgelement_id
orgdimensionid	orgdimension_id
patchreceived	value
commit_date	key
category	label

4.1.4 Grouped Categorized Value Result

Code Contributions received per ISP Across Levels

This metric is used to visualize the number of code contributions received by ISP across different levels in an organization. As shown in figure 4.12, x-axis shows number of patches received and y-axis shows the top 20 ISPs of selected orgelement. The different colors represents the highest orglevel crossed by the patches. Only top 20 ISPs are chosen in order to make the chart comprehensible. For example, root orgelement can have more than 100 ISPs which makes the chart

incomprehensible to read. The top 20 ISPs are chosen based on total number of patches received by the ISP. Internal contributions are also considered in this metric. For example, the red bar indicates contributions that do not constitute patch-flow (contributions by the team running the IS project).

The inner source projects receive a varying number of code contributions. For example in figure 4.12, two inner source projects (namespace1/963, namespace3/1688) received very less code contributions. In contrast, other two inner source projects (namespace2/784, namespace1/7) received more than 600 code contributions. Figure 4.12 displays only the top 20 projects we consider active.

Table 4.13: Mapping of metric fields to resultrow table

Output Fields from Metric	Columns of ResultRow Table
TRANSFORMATIONRUN_ID	transformationrun_id
TRANSFORMATION_ID	transformation_id
orgelementid	orgelement_id
orgdimensionid	orgdimension_id
patchcontributed	value
maxlevelcrossed	key
isp_name	label

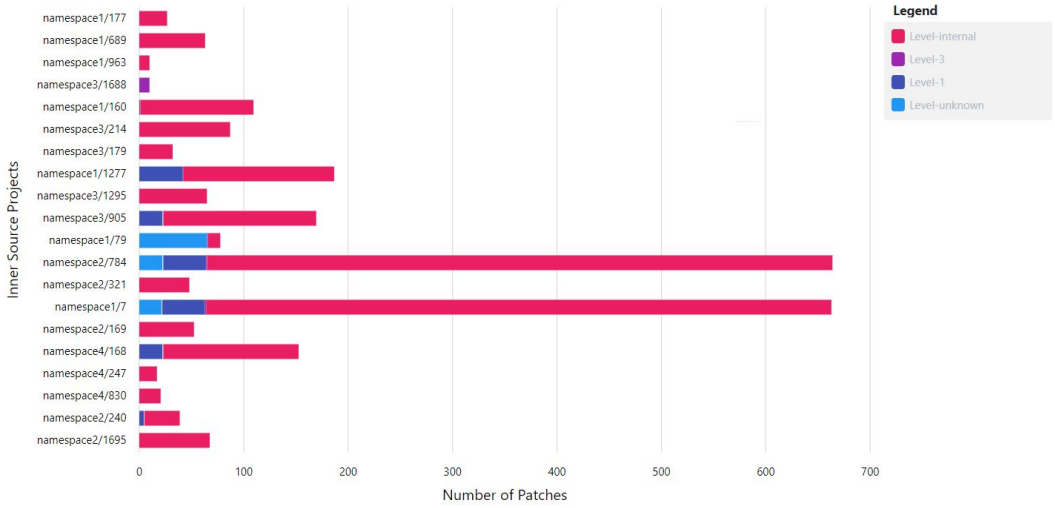


Figure 4.12: Code Contributions received per ISP across levels

As mentioned before, With an increasing amount of received code contributions, such a graph can quickly become incomprehensible when there is no categorization based on organizational levels. Thus, As shown in figure 4.12, received code

contributions are categorized based on hierarchical level of organization. Here the level *unknown* shows that the patch received from an person had no org. element associated to it. Thus, the level from where the patches are received is unknown.

4.1.5 Categorized Value Result

Authors and their Patch Contributions

This metric shows the histogram of authors and their patch contributions. It calculates how many number of authors have contributed a specific range of patches. For example, if 3 authors had contributed more than 35 patches, then those 3 authors will fall under interval 31-62 patches. As shown in figure 4.13, x-axis presents number of patches in intervals and y-axis presents number of authors. As there were no predefined histogram in ngx-charts library, usual bar chart is used to visualize this histogram. Freedman rule is used to determine the bin widths of the histogram.

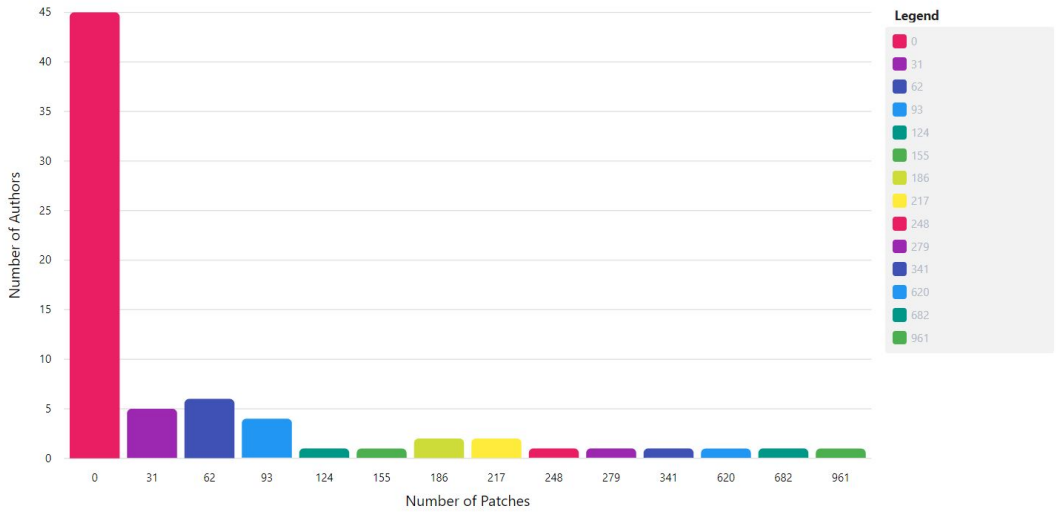


Figure 4.13: Authors and their Patch Contributions

Table 4.14: Mapping of metric fields to resultrow table

Output Fields from Metric	Columns of ResultRow Table
TRANSFORMATIONRUN_ID	transformationrun_id
TRANSFORMATION_ID	transformation_id
orgelementid	orgelement_id
orgdimensionid	orgdimension_id
numofpersons	value
comparand	key

A bin is called as interval and it is a way of sorting data in a histogram. When the data is put into categories, they are put without any thoughts about how that data might tell us something. when it comes to histogram the aim is to analyse how the data is spread out. Therefore categories should be carefully chosen. That is, right amount of bins has to be chosen to get the information we need.

It is difficult to choose bin intervals for large dataset. To find the bin intervals, various types of approach were applied to dffierentiate the results and choose the best one out of it. At the end of this analysis *Freedman-Diaconis's Rule* is chosen for finding bin intervals. Advantage of this approach is that the bin intervals are chosen by finding interquartile range(IQR). IQR is the range where the most of the values lie.

Rule : Choose the bin interval as twice the interquartile range of the data, divided by the cube root of the sample size(Freedman & Diaconis, 1981). Below is the formula to find the bin interval.

$$BinWidth = 2\frac{IQR}{\sqrt[3]{n}}$$
$$IQR = Q3 - Q1$$

Example to find IQR:

Step 1: Sort the number of patches.
1, 2, 5, 6, 7, 9, 12, 15, 18, 19, 27.
Step 2: Find median.
1, 2, 5, 6, 7, 9, 12, 15, 18, 19, 27.
Step 3: Split the numbers into two category (left and right) by finding the median.
(1, 2, 5, 6, 7), 9, (12, 15, 18, 19, 27).
Step 4: Q1 = median of left and Q3 = median of right
(1, 2, 5, 6, 7), 9, (12, 15, 18, 19, 27). Q1 = 5 and Q3 = 18.
Step 5: Subtract Q1 from Q3 to find the IQR.
18 – 5 = 13.

5 Evaluation

This chapter evaluates whether the initially defined requirements are fulfilled. Along with the requirements described in section 1.3, the ways how to verify them in the webclient is also explained.

5.1 Functional Requirements

Store KTR files in database

After uploading the KTR file in the transformation page of web-client, it gets saved in the database when the *save* button is clicked. This fulfills requirement 1.

Update transformation

When *Transformation* tab is clicked in the web-client, the list of transformations are shown with edit button next to it. On clicking the edit button, it leads us to *Update Transformation* page. On this page, the changes can be made to the transformation using the form fields. Finally, the transformation gets updated in database on clicking the update button. So requirement 2 is implemented.

Download KTR files

In *Update Transformation* page, a link appears above the *save* button. The KTR file of the selected transformation will get downloaded when this link is clicked. Thus, the requirement 3 is fulfilled.

Load transformations on fresh install

After fresh install there will be no existing transformation in the database. On clicking transformation tab, it loads all the default transformation from source folder and then saves it in database and finally displays them in the list. Hence, the requirement 4 is satisfied.

Choose metrics for root or other org. elements

In the *Update Transformation* and *Create Transformation* page, two check boxes appear when the *Organizational Element* is chosen as economic agent. First one is to choose if the visualization should appear for whole organization. Second one is to choose if the visualization should be shown for other orgelements. Thus, the requirement 5 is fulfilled.

View charts in larger tile

When any one of the tile is selected from dashboard tab, it maximizes to larger size. This is achieved using angular material. As a result, the chart is viewed in the bigger tile. Thus, the requirement 6 is fulfilled.

Reduce complexity of KTR files

At the beginning of this thesis, the certain steps were repeatedly used to get all the descendants of selected org. element. These combination of complicated steps are now reduced to single step in which the hierarchical information of selected org. element is injected. Thus ths requirement is also fulfilled.

Apply transformation for all the agents of selected type

With the help of *cmsuite_inject_descendants* step , metric calculation can be performed on all the organizational units agnostic of granularity. This is achieved by integrating rowlisteners in java classes. Therefore, this requirement is satisfied.

Metrics

The remaining functional requirements are to implement different types of metric transformations. That is, for each metric calculation a kettle transformation file should be created. the implemented ktr files are located in the default folder of transformation manager. In the webclient, one can execute all these transformations by clicking *Run Transforamtion* button from *Transformation Run* tab. The metrics marked with ☑ are implemented in this thesis. The results of these metrics are already discussed in previous chapter. Hence these requirements are also satisfied.

Single-Value-Result

- ☑ Code contributions made(Excl. Internal)
- ☑ Code contributions made(Incl. Internal)

Time-Series-Result

- ☑ Code contributions made per month(Excl. Internal)
- ☑ Code contributions made per month(Incl. Internal)
- ☑ Persons Involved per month
- ☑ First Time Contributors per month
- ☑ Last Time Contributors per month
- ☑ Code Contributions Received per month

Categorized-Time-Series-Result

- ☑ Code Contributions made per month by namespace(Excluding Internal)
- ☑ Code Contributions made per month by namespace(Including Internal)
- ☑ Data Completeness of Received Code Contributions
- ☑ Patch-Flow by Org. Level
- ☐ Code Contributions made by teams per month
- ☐ Relative patch flow over time

Grouped-Categorized-Value-Result

- ☑ Code Contributions received per ISP across levels

5.2 Non-Functional Requirements

Response Time

First non functional requirement states that response time should stay within one to five seconds. Although many visualizations are shown in dashboard, the response time stays within 5 seconds. Therefore, this requirement is met.

Fault Handling

There are cases in which there are no values to be shown in visualization for particular agent. In such cases, dashboard should notify user that there were no values for this metric instead of showing error message. The fault handling of null values are fulfilled and thus this requirement is met.

Comprehensibility

All the visualizations are implemented without having any clutters in the chart. For example, in the metric "code contributions received per ISP across levels", it is nearly impossible to show all the ISP's of selected organizational unit. Therefore, only top 20 ISP's are displayed. Similarly it is taken care for other metrics also when chart clutter was encountered. As a result, it is comprehensible. Thus this requirement is fulfilled.

Complexity of Kettle Transformations

The kettle transformations are implemented such that it does not have any complicated steps which could affect the performance or readability. As the complexity of the KTR files are reduced, this requirement is fulfilled.

Chart Legends

The ktr files are implemented such that it passes the suitable legend information in the final result. For some metrics, the categories on legend are org. element or ISP. When the name of these agents are not retreived during metric calculation, then the id of these elements will be shown as legends. In order to avoid id as legend, the combination of steps are included which takes care of merging the names to corresponding rows. Thus this requirement is also fulfilled.

6 Future Work

Time Interval

Currently, all the metrics are implemented such that entire time is taken into consideration. That is, at what time interval the particular metric should be displayed for, is not yet decided. For example, there could be options like Recent (this month. or maybe rolling last month) , Something long run (this business year, or maybe rolling last year), All of time. CMSuite could let user to choose any one from this option, so that he can get results only for the selected time interval.

Chart Labels

The chart labels that are displayed on both the axes are same for all the visualizations of same result type. For example, in time series results the label on x-axis is month and y-axis is "Number". However, this number could have different meaning for each metric like number of persons, number of patches etc. Thus, in order to use customized chart labels for each transformation the label information can be stored in transformation table of the database. As a result, when transformation are retrieved for particular tile in dashboard, the label can be assigned for the corresponding chart.

Granularity

Until now, all the metrics were implemented agnostic of granularity. For example, if the transformation was saved with orgElement as agent type, then that transformation computes the result for all the orgelements found in database. Thus, CMSuite could have a feature that enables user to pick the granularity for a transformation.

Personalization

Currently, the dashboard shows all transformations in a arbitrary sequence and size. It is applied for all the agentType that is chosen. There could be a feature

that allows user to pick specific agents(OrgElement/InnerSourceProject/Person) for which the visualizations should be shown. For example, in a transformation that computes result for orgElement, a list of orgElements could be chosen, so that it computes result only for these specific ones.

Export visualizations to PDF

At present, the visualizations can be viewed in a larger tile to perform any analysis. However, the user will have to open the application everytime when he wants to see the same visualization. Therefore, the feature to export these visualizations to PDF would be beneficial for users to carry out their analysis even when they logged out from the application. There are plenty of library which supports exporting the html content to PDF. But the easiest possible solutions is by using CSS @media print rules to customize the printing styles of the web page which is required to be exported. As a result, the browser will adapt to these rules and produce the document when it is saved as pdf.

Complete vs Incomplete Data

In a metric "Data completeness over time", only three categories were shown in the chart. One is to see the code contributions with complete data, other with missing authors and no orgElement associated with author. There could also be other possibilities like missing ISP, no orgElement associated with ISP. For the whole organization, showing different categories for each of this incomplete data could be considered as future work.

Relative Patch Flow over time

This metric could help the user to see the patch-flow relative to the total amount of code contributions to the IS projects per month. This can be achieved by using groupby step from spoon's menu. GroupBy step of spoon allows the metric designer to calculate percentile, ratios etc. This metric can be easily implemented by simply extending the already existing metric "Patch-flow by Org. level".

Code Conributed by teams over time

This metric would help user to analyse how many code contributions have been contributed by the teams of selected org. element. Here teams does not have to be only the org. units that has person attached to it. For a selected Org. Element, stepping down to one level downwards in a organization hierarchy will lead us to all child units. When all the descendants of these child units are agreggated, this could act as a teams for this metric calculation.

7 Conclusion

In this work, the CMSuite's dashboard infrastructure is extended iteratively in the way needed for new metric implementations.

Earlier, it was difficult to create transformations that stores results for every agent. Moreover, there were no metric implemented that helps measuring the inner source collaboaration. Furthermore, collecting descendants of all the organizational units was creating huge overhead when recursive method was used. Consequently, the transformations were runnning for hours.

Decisions were made to use *rowInterface* and *stepInterface* of Pentaho data integration tool to inject required data at runtime. As a result, the execution speed of transformation is increased. Usage of fundamental graph search techniques (Depth First Search), suitable datastructures and other memory management concepts to reduce the computational complexity has facilitated the process of collecting the descendants efficiently without affecting the performance and execution speed.

After applying the above mentioned ideas, now the CMSuite enables the metric designer to easily create the clean and comprehensible kettle transformations without facing trouble of handling parent-child relationships within single transformation. As a result, many new metrics were implemented and executed using the introduced architecture.

Advantages of the results that we get from this work are that the user can visualize the metrics in webclient for all the organizational units. That is, if a metric is defined for particular purpose and if it is applicable for one organizational unit then it gets executed for all the organizational units. This includes other economic agents, that are part of the organizational hierarchy. Thereby, user does not have to create metrics for each organizational units separately.

Appendix A Update Transformation

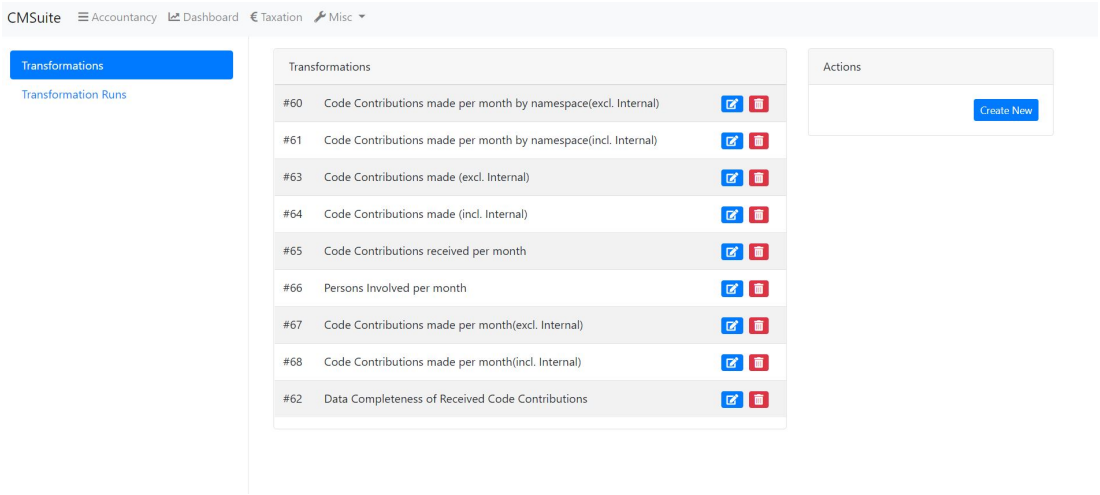


Figure 7.1: List of transformations with edit button next to it.

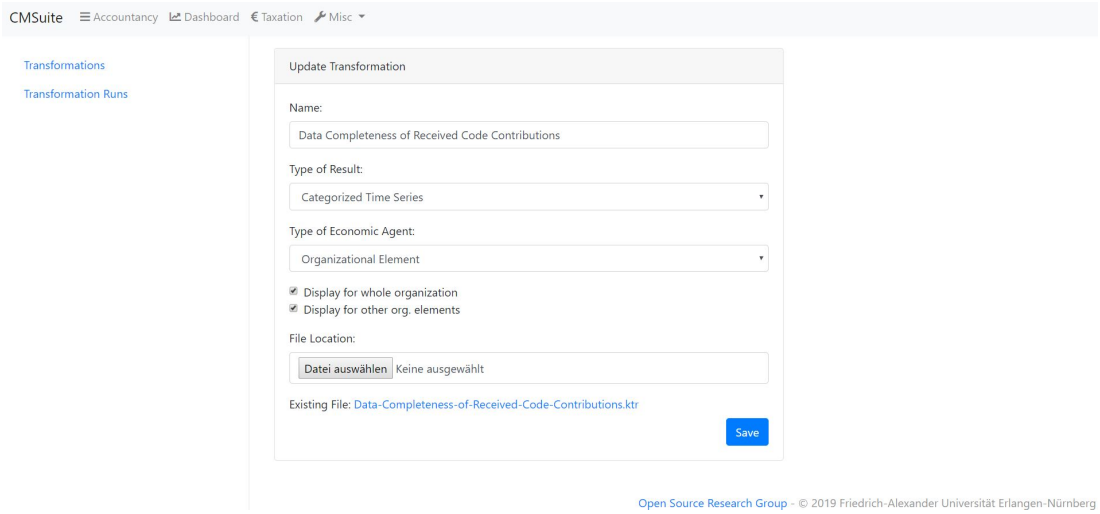


Figure 7.2: Page to update transformation

Appendix B Dashboard

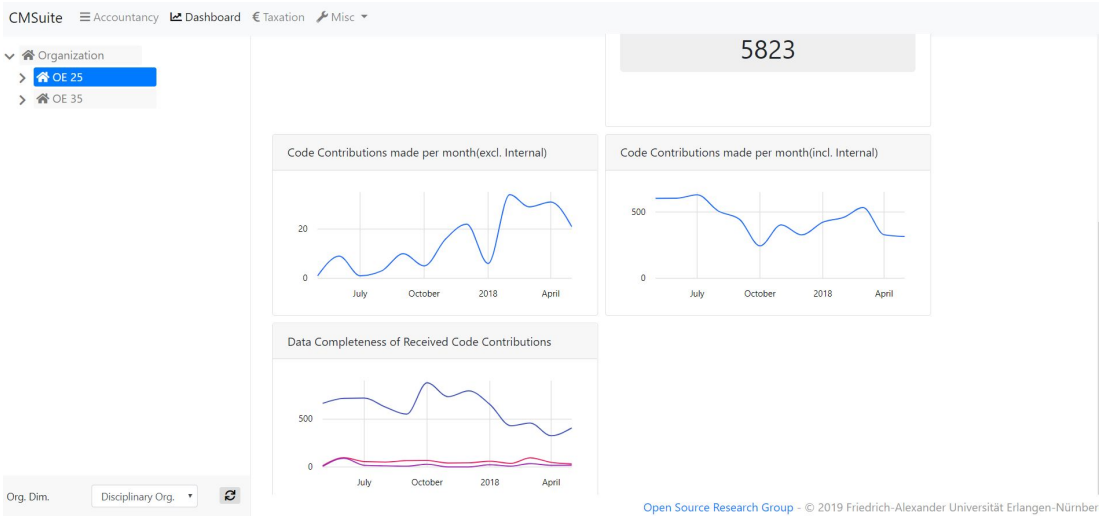


Figure 7.3: Tiles in dashboard

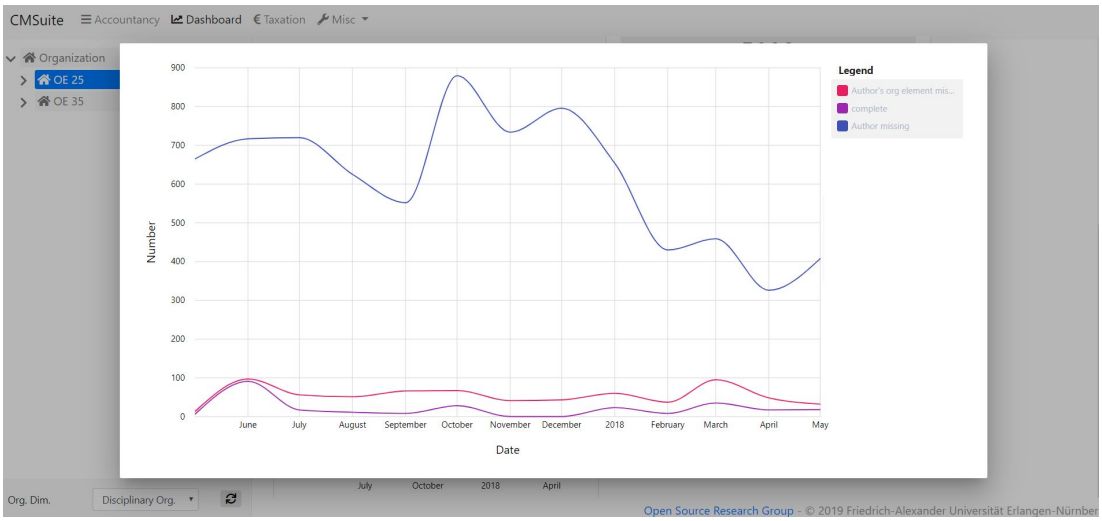


Figure 7.4: Maximized view of single tile

Appendix C Default KTR File Location

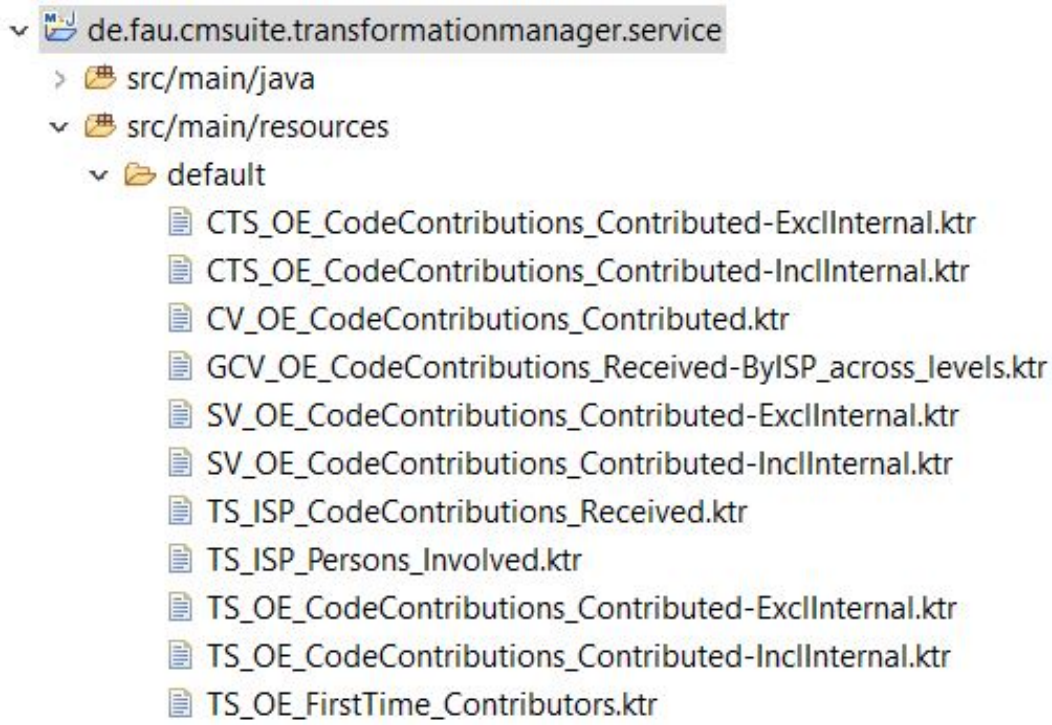


Figure 7.5: Default KTR File Location

References

Angular Material Dialog page. (2019). https://material.angular.io/components/dialog/overview. Accessed on 2019-01-09.

Capraro, M., Dorner, M. & Riehle, D. (2018). The patchflow method for measuring inner source collaboration. *International Conference on Mining Software Repositories*.

Capraro, M. & Riehle, D. (2017). Inner source definition, benefits, and challenges. *ACM Computing Surveys*, *49*(4).

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

Daeubler, A. (2017). Design and implementation of an adaptable metrics dashboard. *Master's thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg*.

Executing a PDI Transformation. (2019). https://wiki.pentaho.com/display/EAI/Executing+a+PDI+transformation. Accessed on 2019-01-12.

Freedman, D. & Diaconis, P. (1981). On the histogram as a density estimator: L 2 theory.

Groupby step requires Apache math. (2019). https://github.com/pentaho/pentaho-kettle/blob/master/engine/src/main/java/org/pentaho/di/trans/steps/groupby/GroupBy.java. Accessed on 2019-02-12.

Hansen, M. (2018). An accounting tool for inner source contributions. *Master's thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg*.

Jan Pahl, P. & Damrath, R. (2012). Mathematical foundations of computational engineering: A handbook. In *Graphs from: Mathematical foundations of computational engineering: A handbook* (pp. 489–580). Springer Science Business Media.

Khuller, S. & Raghavachari, B. (2014). Computing handbook, computer science and software engineering. In *Graph and network algorithms from: Computing handbook, computer science and software engineering* (pp. 5-1–5-5). CRC Press.

ModifiedJavaScript step requires JavaMail. (2019). https://github.com/pentaho/pentaho-kettle/blob/master/engine/src/main/java/org/pentaho/di/

trans/steps/scriptvalues_mod/ScriptValuesAddedFunctions.java. Accessed on 2019-02-12.

PDI Kettle's Third-party dependencies. (2019). https://github.com/pentaho/pentaho-kettle/blob/master/engine/pom.xml. Accessed on 2019-02-12.

PostgreSQL Storing Binary Data. (2018). https://www.postgresql.org/docs/7.4/jdbc-binary-data.html. Accessed on 2018-12-12.

StepInterface pentaho javadoc. (2019). https://javadoc.pentaho.com/kettle530/kettle-engine-5.3.0.0-javadoc/org/pentaho/di/trans/step/StepInterface.html. Accessed on 2019-01-24.

Trans pentaho javadoc. (2019). https://javadoc.pentaho.com/kettle530/kettle-engine-5.3.0.0-javadoc/org/pentaho/di/trans/Trans.html. Accessed on 2019-01-24.

User Defined Java Class. (2019). https://wiki.pentaho.com/display/EAI/User+Defined+Java+Class. Accessed on 2019-01-06.

Weiss Mark, A. (2014). Computing handbook, computer science and software engineering. In *Data structures from: Computing handbook, computer science and software engineering* (pp. 3–17). CRC Press.

Work with Rows. (2019). https://help.pentaho.com/Documentation/6.0/0R0/0V0/010/000/020/010. Accessed on 2019-01-12.