ITEA 3 Office

High Tech Campus 69-3
5656 AG Eindhoven
The Netherlands

Tel : +31 88 003 6136
Fax : +31 88 003 6130
Email : info@itea2.org
Web : www.itea2.org

ITEA 3 is a EUREKA strategic ICT cluster programme

INFORMATION TECHNOLOGY FOR EUROPEAN ADVANCEMENT

D3.2: Final release of the measuring, analysis and visualization tools that conform the MEASURE platform

MEASURE

……………………………………………………………

Executive summary

The main purpose of WP3 is the implementation of tools to measure, analyse, and visualize the metrics and methods defined in WP2 to extract and show information about the software engineering processes. The main challenge of this WP lies on the ability of working with heterogeneous software engineering processes. All the developed tools will be integrated in a unique platform called MEASURE platform that constitutes the main outcome of the project.

D3.2 deliverable is the final release of this MEASURE platform that is delivered for internal evaluation. Then, the feedback obtained from WPs 4 and 5 will be used to refine this platform in order to provide a solid and complete MEASURE framework that will be released by M35.

This deliverable is an update of the “D3.1: First release of the measuring, analysis and visualization tools for internal evaluation” delivered earlier in the project. It provides an overview of architecture evolutions, integrations and new features and functionalities implemented during the final phases of the project.

D3.2 is a software deliverable. The source code of the MEASURE platform is available on Github:

Measure Project GitHub: https://github.com/ITEA3-Measure

It is composed of 4 main components:

- The platform itself: https://github.com/ITEA3-Measure/MeasurePlatform
- The agent allowing the execution of metrics in the client side of the Platform: https://github.com/ITEA3-Measure/MeasureAgent
- The API allowing defining new metrics: https://github.com/ITEA3-Measure/SMMMeasureApi
- The repository to collect the implementation of different measurements: https://github.com/ITEA3-Measure/Measures

This document is an accompanying file that defines the MEASURE platform architecture and provides details about its implementation and usage. The measuring and analysis tools interacting with is platform are also described in this document.

D3.2 software is delivered in this first release as a pre-deployed platform accessible for end users via a public web interface following this link: *http://194.2.241.244/measure/*. The requirements to install this platform locally will be presented in the second release of this platform. Moreover, for both pre-deployed and locally deployed, the MEASURE platform will allow communication with remote computers through the agent component, to collect different direct measurements.

Table of Contents

1. Introduction

1.1. Role of this deliverable

The deliverable D3.2 is the final release of the measuring, analysis and visualization tools in the MEASURE project. These tools are integrated into a unique platform called the MEASURE platform. D3.2 is a software with an accompanying document describing the MEASURE platform architecture, implementation and usage.

1.2. Structure of this document

The D3.2 document is organized in three big sections.

- The section 2 describes the MEASURE platform architecture and components. Details about its implementation and development roadmap are provided,
- The section 3 describes the measuring tools allowing to gather measurements in different software development lifecycle. These tools are connected to the platform using a public API.
- The Section 4 describes the analysis tools allowing to analyse collected measurement. These tools are also connected to the platform using a public API

1.3. Relationship with others MEASURE deliverables

The deliverable D3.2 is an update of the deliverable D3.1. It is related to the different MEASURE deliverables that follow:

- D3.3 Documentation and user guidelines of the MEASURE framework
- D2.2 Formal specification of MEASURE metrics: The metrics specified in D2.2 will be supported by the MEASURE platform. More extensions to include more metrics are planned.
- D5.4 Final evaluation results from the executed case studies: D3.2 will be the input of the deliverable D5.4. The feedback from this evaluation will allow to improve the MEASURE platform.

1.4. Contributors

All the tool providers contributed to this document. Namely:

- SOFT: Design and implementation of the MEASURE platform as well as Hawk measuring tool and Quality Guard analysis tools.
- MTI: Design of the MEASURE platform and extension of MMT and the implementation of MINT tool
- ICAM: Design and implementation of EMIT and M-Elki tools
- IMT: Design and implementation of MINT and the Metric Suggester tools
- Bitdefender and UniBuc: Design and implementation of RIVER tool
- Ericsson: Test Case Generator

1.5. List of updates in documents

- Integration of Analysis Platform Components in global architecture (section 2.1.3)
- New section related to Measurement storage approach (section 2.2.2)
- New section related to the Analysis Platform integration mechanism (section 2.2.4)
- New approach to manage Data Model returned by Measures (2.4.3)
- Introduction of the notion of visualisation packaged with measure (section 2.4.4)
- Introduction of the notion of Monitoring Application which complement Measures (section 2.5)
- Presentation of the Hawk tool (section 3.2)
- Update of the presentation of the MMT tool (Section 3.3)
- Update of the presentation of EMIT tool (Section 3.4)
- Deletion of the FSMHSGen test generation tool of IMT (previously section 3.5) since this tool is no more maintained
- New section to describe 5 analysis tools (section 4)

2. The MEASURE Solution

2.1. Overview the Solution

2.1.1. Role of the Measure Platform

The measure platform is a tool dedicated to measure, analyse, and visualise the metrics to extract and show information of the software engineering processes.

- Implement the tools for automatically measure software engineering processes during the whole software lifecycle by executing measures defined in SMM standard and extracted from a catalogue of formal and platform-independent measurements.
- Provide methodologies and tools which allow measure tools provider to develop a catalogue of formal and platform-independent measure.
- Implement storage solution dedicated to measurements resulting of measure execution in big data context.
- Implement visualization tools to expose the extracted results in an easy-readable fashion, so allowing a quick understanding of the situation and the possible actions that can be taken to improve the diverse stages of the software lifecycle.
- Implement an extension mechanism dedicated to the integration of external analysis tools will provide long terms analysis and predictive evaluations on collected measures.
- Implement of an Extended API allowing to facilitate the integration on Measure Platform with external tools and services.

The platform activity is organised around its ability to collect measurement by executing measures defined by the SMM standard. SMM measures are auto-executable component, implemented externally, which can be interrogated by the platform to collect measurements.

2.1.2. Main process of the platform

The Measure platform provides services to host, configure and collect measures, storing measurement, present and analyse them. These measures are first defined in SMM standard using the Modelio modelling tool and its extension dedicated to SMM modelling. They are packaged under an executable format as Measure Definition (For more details related to measure execution format, please refer to the section 2.4.1 of this document).

Next, measures are registered and stored on Measure platform using the dedicated REST service or the Web user interface. In order to initiate the collect of measurement, the next step consists on defining instance of measure based on measure definitions. A measure represents a generic data collection algorithm that has to be instantiated and configured to be applied on a specific context. For example, a measure which collects data related to an SVN repository must be configured by the URL of this repository.

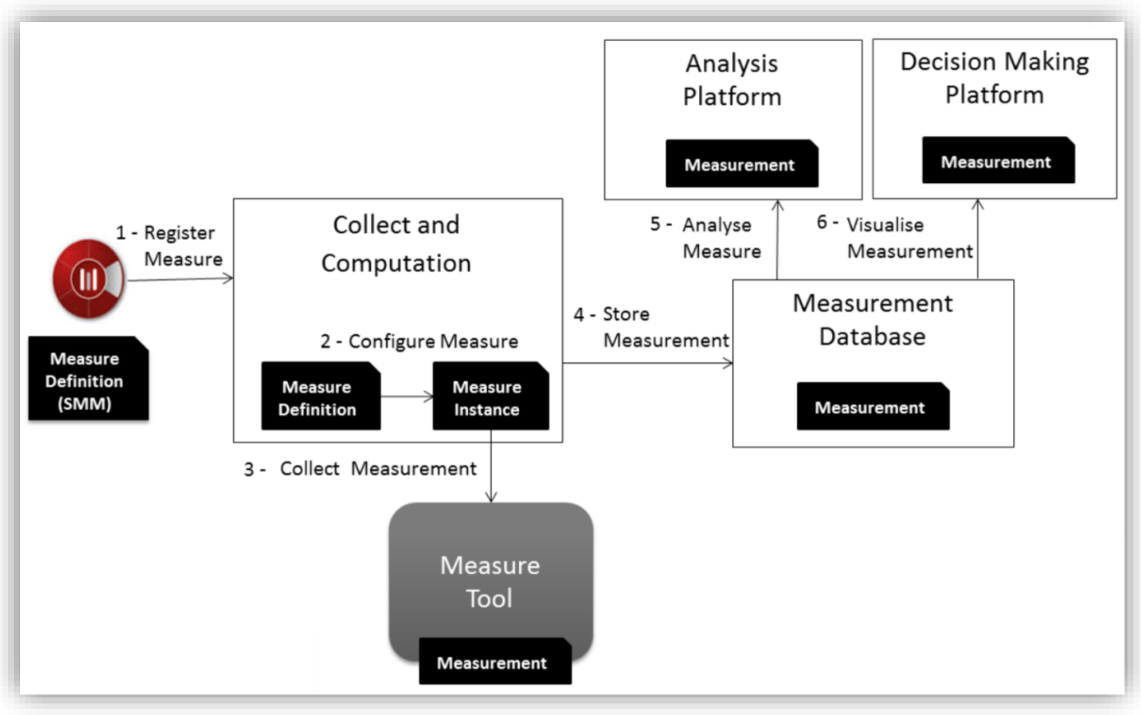


Figure 1. Main process of Measure platform

Next the Measure Platform can start collecting measurement (data resulting of the execution of an instantiated measure). Direct measures collect data in physical world while the Derived Measures are calculated using previously collected measurement as input. Collected measurements are stored on a NoSQL designed to be able to process a very large amount of data. To collect measurements, the direct measures can delegate the collect work to existing Measure tool.

Finally, stored measurements are presented directly to the end user following a business structured way by the Decision-making platform, a web application which allows organising measures based on projects / software development phases and display its under various forms of charts.

The measurements can also be processed by analysis tools to present consolidated results. The analysis platform is not the object of this delivery, please refer to WP4 for more details on this topic.

2.1.3. Platform Architecture

The Measure Platform architecture is organised around three main deployment units: The Measure Platform which manages the collection and computation of measurement and the presentation of this measurement, the analysis platform which regroups measurement analysis tools and finally the Measure Front which regroups measure tools and Platform Agents.

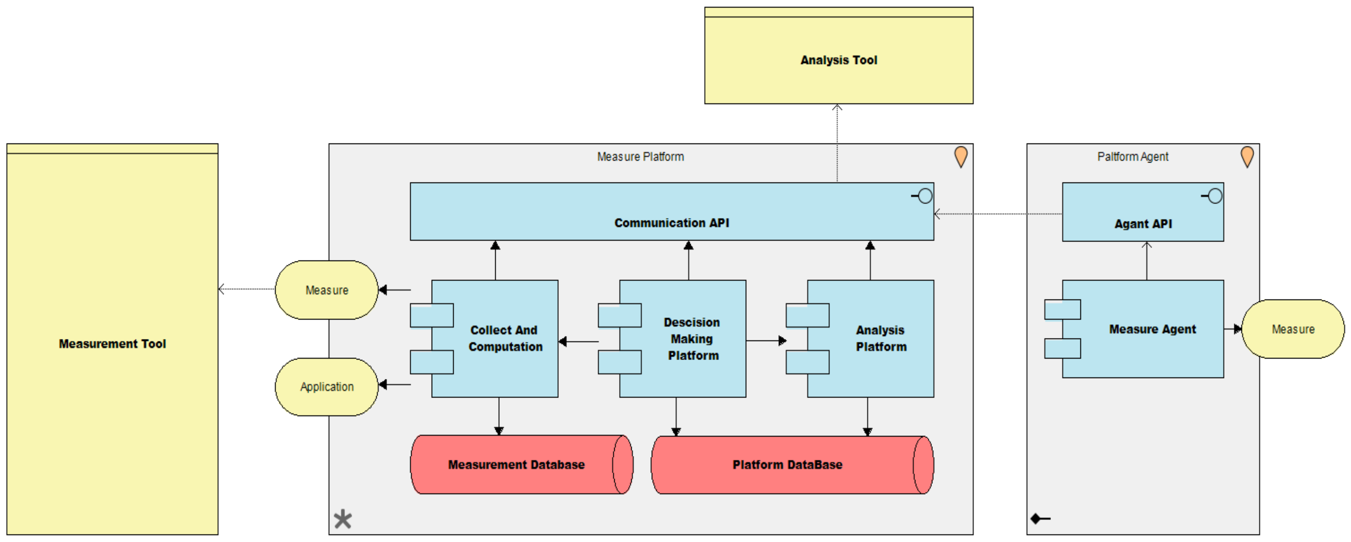


Figure 2. Architecture overview of Measure Platform

Measure Platform: Central component of this deliverable, the measure platform provides services related to data collection, analysis and display. It is composed of six sub-components:

- The Communication API is a remote API which allows the Analysis Platform and Agent to communicate with the Measure Platform. It provides various services like an access to measurements data, the configuration of measures or the ability to manage the scheduling of measure execution.
- The Collect and Compute Component is in charge of the storage of Measure Definition, the instantiation of Measure, the scheduling of measure execution and organizes the measurements collection and processing.
- Decision Making Component represents the web application which assures the presentation of measures and collected measure to the end user.
- The Platform Database is a standard SQL database which manages data in relation with the collect, organisation and presentation of measures.
- The Measurement Database is a separate NoSQL database which ensures the measurement persistence in an efficient and scalable way.
- The Analysis Platform ensures the integration of external analysis tools into the platform. As the Measure Platform focuses its activity around collecting and calculating measurement in real time, the Analysis Platform will provide long terms analysis and predictive evaluations by working on the history of measures stored in measurement database.

Platform Agent: Measure tools on client side which collects data. The executable measure provides a way to collect data in the physical world. A measure can be executed on the Platform side and collects physically this data through an existing measure tool. A Measure can also be directly executed on the client side (a computer close to measured element) by the intermediary of a Platform Agent.

Measurement Tools: A Measurement Tool is an external tool which collects or calculates measurements from a specific source. In the context of the project, 5 Measurements tools have been developed (see Section 3) and lots of exiting tools in the market (like Sonar Cube for code quality metrics) can be also considered as Measurement Tools.

Analysis Tools: An Analysis Tool is a set of external services which works on the historical measures values in order to provide advanced and valuable analysis functions to the platform. In order to support a large set of analyses services and do not limit to it a specific technology, the Analysis Tools are external processes. The analysis tool is integrated to the platform using a specific API. This integration includes embedded visualisation provided by the analysis service into the platform.

Measures: A Measure is a small and autonomous java program based on the SMM specification which allows to collect measurements. A Measure can be Direct (Collect of measurement in physical world), a Proxy (Ensure communication between a Measurement Tool and the Platform) or Derived Measure (Measure calculated by the aggregation of existing Measures).

Applications: An Application is a set of Measures aggregated together in order to address a functional requirement. The application is associated with a visual dashboard which directly integrated into the Decision Making platform when the Application is deployed on a project.

2.2. Measure Platform

2.2.1. Collect and Compute Component

This central component of the Measure Platform has the goal to organize direct measure's collection and derived measure calculation by providing a way to register, instantiate and execute measures defined in SMM. The mains services offered by this component are:

- Providing a registration services for measure's definition in SMM format.
- Providing a configuration service which allows to define Measure Instances.
- Providing a measurement collecting system based on the execution registered measures.
- Providing a calculation service for derived measures.
- Providing a storage service for collected and calculated measurements.

Technical architecture

The implementation of this component is based on a java enterprise application framework: The Spring Boot framework which allows creating stand-alone, production-grade Spring based applications easy to deploy. It uses a standard SQL data base to store platform related data and communicate with other component of the platform using a REST API.

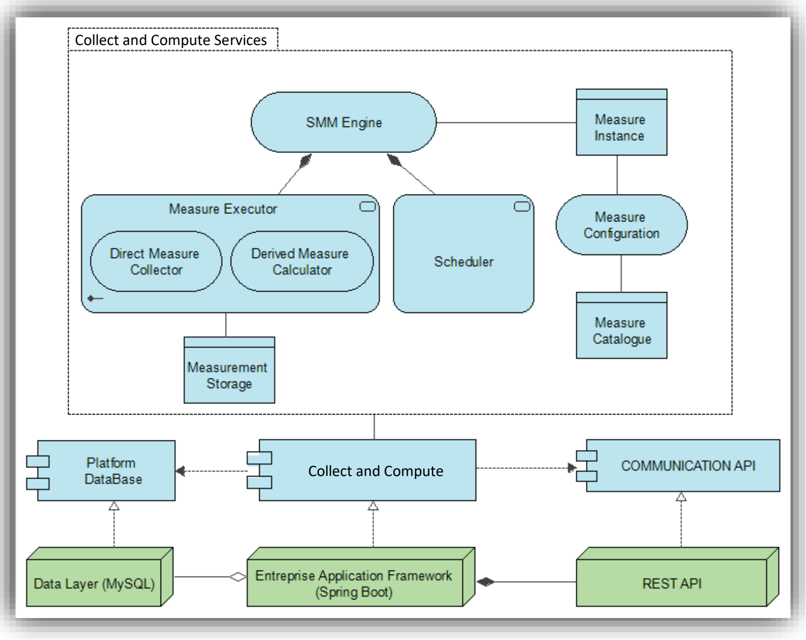


Figure 3. Architecture overview of collect and compute component

Measure catalogue

According to WP2, Measures are defined in SMM (Structured Metrics Meta-model) format, an OMG Standard. The Collect and Compute Component provide services to register Measure defined in specific executable format derived from the SMM standard using the Communication API. Registered measures are stored in the Measure Catalogue. Once registered, the measure catalogue allows the platform to access the measure meta-data that contain information required to identify the measure (unique id), the type of the measure (Direct Measures, Collective Measures, Binary Measures, etc.), how to access the measure implementation, which will be the format of data returned by the measure (Unit) and a list of configuration points provided by the measure (Scope).

Measure configuration and Measure instance

A Measure is composed of a set of meta-data associated with an implementation and represents a generic data collection algorithm that has to be instantiated and configured to be applied on a specific context. Consequently, in order to be executed by the platform, it's required to define an instance of a registered measure which will fill the configuration values describe in measure meta-data.

As example, measure instance will contain:

- An identifier for the measure instance.
- Information related to collect cycle of measurement (when and in what interval a measurement is collected).
- Information required identifying the measured system (configuration values).
- Information related to required measure input and output for Derived Measure.
- Information related the data structure return by the measure

Once instantiated, the data related to measure instances are stored in Platform database.

Collecting measures using the SMM engine

The SMM standard defines two main types of measures: the direct measures which are measures collected in physical world and calculated measures (Collective Measures and Binary Measures) which are measure derived from direct measures.

- Direct measure Collecting: The SMM engine will invoke the implementation of the direct measure after having communicated to it the configuration parameters defined by the measure instantiation.

- Derived Measure Calculation: Using pre-existent measurements stored in measurement database, provided by direct measures, the SMM Engine will provide services to calculate derived measures by providing this measurement as input to derived measures implementation.

The SMM Engine provides services to execute measures. The execution of a measure can be unique and trigged manually or can be scheduled and repeated over time. In this second case the SMM Engine use a specific scheduling component to organise the executions of measures.

Once collected or calculated, the SMM engine will store the resulting measurements in measurement database.

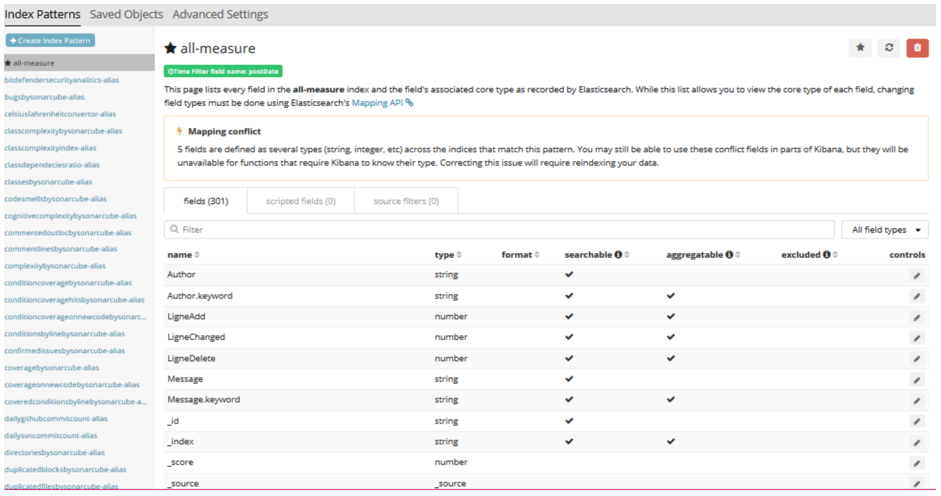
2.2.2. Measurement Storage

Collected and calculated measurements have to be stored in a database for future exploitation by the Decision-making Platform and the Analysis Platform. In the context of our project, we have to deal with a very important quantity of data. Our first estimates tell us that the Measure Platform must be capable of serving several millions of data every day.

Furthermore, due to the nature of the Measures, we need a lot of flexibility related to the nature of the data to store in this database. In fact, we are not in a position to make hypotheses about the nature of the measurements returned by the measurement tools. A measurement can be a simple value or a complex and structured data. In this context, we have used Elasticsearch[1]: a distributed JSON-based search and analytics engine designed for horizontal scalability, maximum reliability, and easy management.

Elasticsearch is a search engine based on the Lucene library. It provides a distributed, multitenant-capable full-text search engine with an HTTP web interface and schema-free JSON documents. The database is developed in Java and is released as open source under the terms of the Apache License.

Elasticsearch used "index", a logical namespace, to store information. An index is like a table in a relational database. It has a mapping which contains a type containing the fields in the index. To store measurements collected by the Measure Platform, we have applied indexing strategy in which each Measure type deployed in Measure Platform catalogue is associated with one Elasticsearch index. Each measurement resulting of the execution of a measures instances is next stored using this index. Discrimination of data between instances of measures is then performed using the standard field "_type" associated with each index.


Index Patterns
Saved Objects
Advanced Settings
Create Index Pattern
all-measure
bitdefendersecurityanalitics-alias
bugsbysonarcube-alias
celsiusfahrenheitconvertor-alias
classcomplexitybysonarcube-alias
classcomplexityindex-alias
classdependeciesratio-alias
classesbysonarcube-alias
codesmellsbysonarcube-alias
cognitivecomplexitybysonarcube-alias
commentedoutlocbysonarcube-alias
commentlinesbysonarcube-alias
complexitybysonarcube-alias
conditioncoveragebysonarcube-alias
conditioncoveragehitsbysonarcube-alias
conditioncoverageonnewcodebysonarc...
conditionsbylinebysonarcube-alias
confirmedissuesbysonarcube-alias
coveragebysonarcube-alias
coverageonnewcodebysonarcube-alias
coveredconditionsbylinebysonarcube-a...
dailygithubcommitcount-alias
dailysvncommitcount-alias
directoriesbysonarcube-alias
duplicatedblocksbysonarcube-alias
duplicatedfilesbysonarcube-alias
fields (301)
scripted fields (0)
source filters (0)
Filter
All field types

all-measure

Time Filter field name: postDate

This page lists every field in the **all-measure** index and the field's associated core type as recorded by Elasticsearch. While this list allows you to view the core type of each field, changing field types must be done using Elasticsearch's Mapping API

Mapping conflict

5 fields are defined as several types (string, integer, etc) across the indices that match this pattern. You may still be able to use these conflict fields in parts of Kibana, but they will be unavailable for functions that require Kibana to know their type. Correcting this issue will require reindexing your data.

name	type	format	searchable	aggregatable	excluded	controls
Author	string		✓			
Author.keyword	string		✓	✓		
LigneAdd	number		✓	✓		
LigneChanged	number		✓	✓		
LigneDelete	number		✓	✓		
Message	string		✓			
Message.keyword	string		✓	✓		
_id	string		✓			
_index	string		✓	✓		
_score	number					
_source	_source					

Figure 4 : Indexation of measurement in Elasticsearch

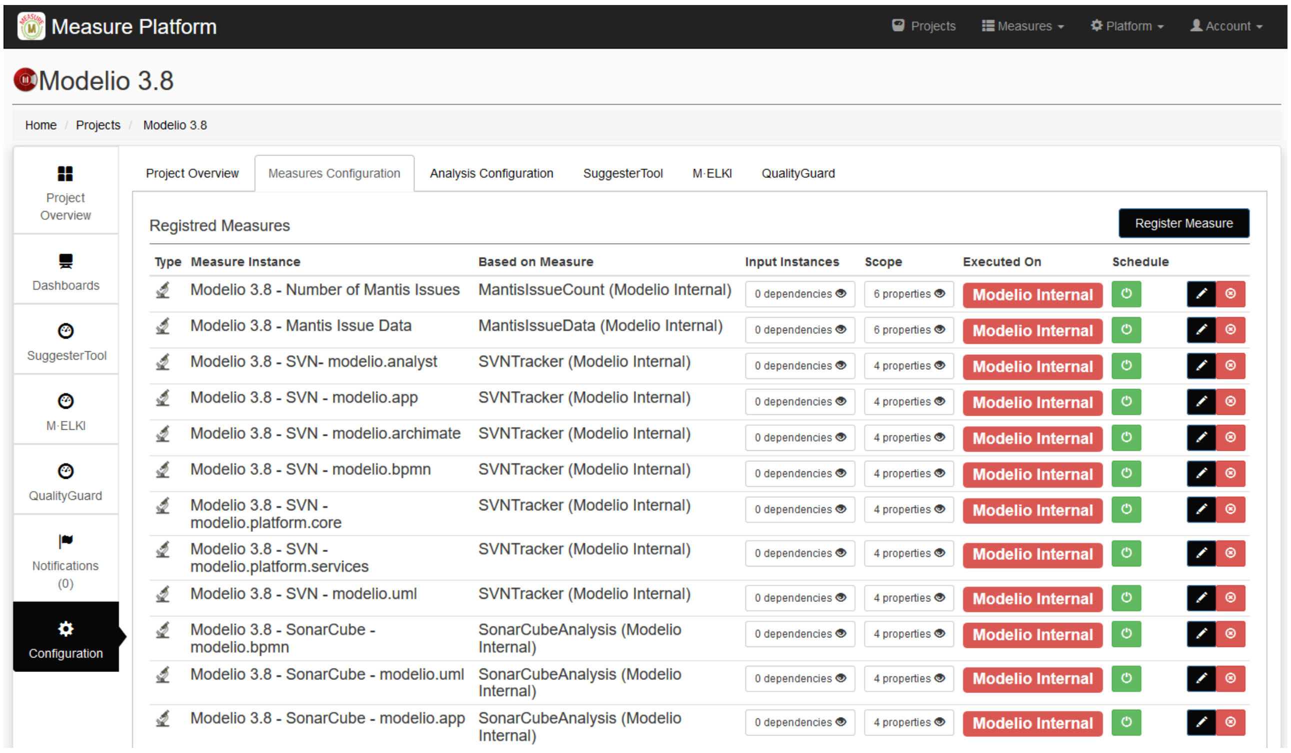
2.2.3. Decision Making Component

The primary goal of the decision-making component is to provide a working environment driven by Measures that can help managers during their decision-making process. The Decision-Making Platform will provide a way to display graphical representation of measured information as graphics and reports by defining and organizing several dashboards, each dashboard presenting a selected pool of measures as graphics, tables or specific indicators. The main services offer by this component are:

- Provide a web application which allows the project manager to organise and configure Measure computation.
- Provide a web application which will be able to present measures executions results and measurements as customisable charts.
- Allow organising these charts on project, phases and dashboard in order to address modern iterative development practices.

[1] *https://www.elastic.co/*

- Integrate consolidated results provided by the analysis tools.



Registred Measures

Type	Measure Instance	Based on Measure	Input Instances	Scope	Executed On	Schedule
	Modelio 3.8 - Number of Mantis Issues	MantisIssueCount (Modelio Internal)	0 dependencies	6 properties	Modelio Internal	
	Modelio 3.8 - Mantis Issue Data	MantisIssueData (Modelio Internal)	0 dependencies	6 properties	Modelio Internal	
	Modelio 3.8 - SVN- modelio.analyst	SVNTracker (Modelio Internal)	0 dependencies	4 properties	Modelio Internal	
	Modelio 3.8 - SVN - modelio.app	SVNTracker (Modelio Internal)	0 dependencies	4 properties	Modelio Internal	
	Modelio 3.8 - SVN - modelio.archimate	SVNTracker (Modelio Internal)	0 dependencies	4 properties	Modelio Internal	
	Modelio 3.8 - SVN - modelio.bpmn	SVNTracker (Modelio Internal)	0 dependencies	4 properties	Modelio Internal	
	Modelio 3.8 - SVN - modelio.platform.core	SVNTracker (Modelio Internal)	0 dependencies	4 properties	Modelio Internal	
	Modelio 3.8 - SVN - modelio.platform.services	SVNTracker (Modelio Internal)	0 dependencies	4 properties	Modelio Internal	
	Modelio 3.8 - SVN - modelio.uml	SVNTracker (Modelio Internal)	0 dependencies	4 properties	Modelio Internal	
	Modelio 3.8 - SonarCube - modelio.bpmn	SonarCubeAnalysis (Modelio Internal)	0 dependencies	4 properties	Modelio Internal	
	Modelio 3.8 - SonarCube - modelio.uml	SonarCubeAnalysis (Modelio Internal)	0 dependencies	4 properties	Modelio Internal	
	Modelio 3.8 - SonarCube - modelio.app	SonarCubeAnalysis (Modelio Internal)	0 dependencies	4 properties	Modelio Internal	

Figure 5. Measures configuration

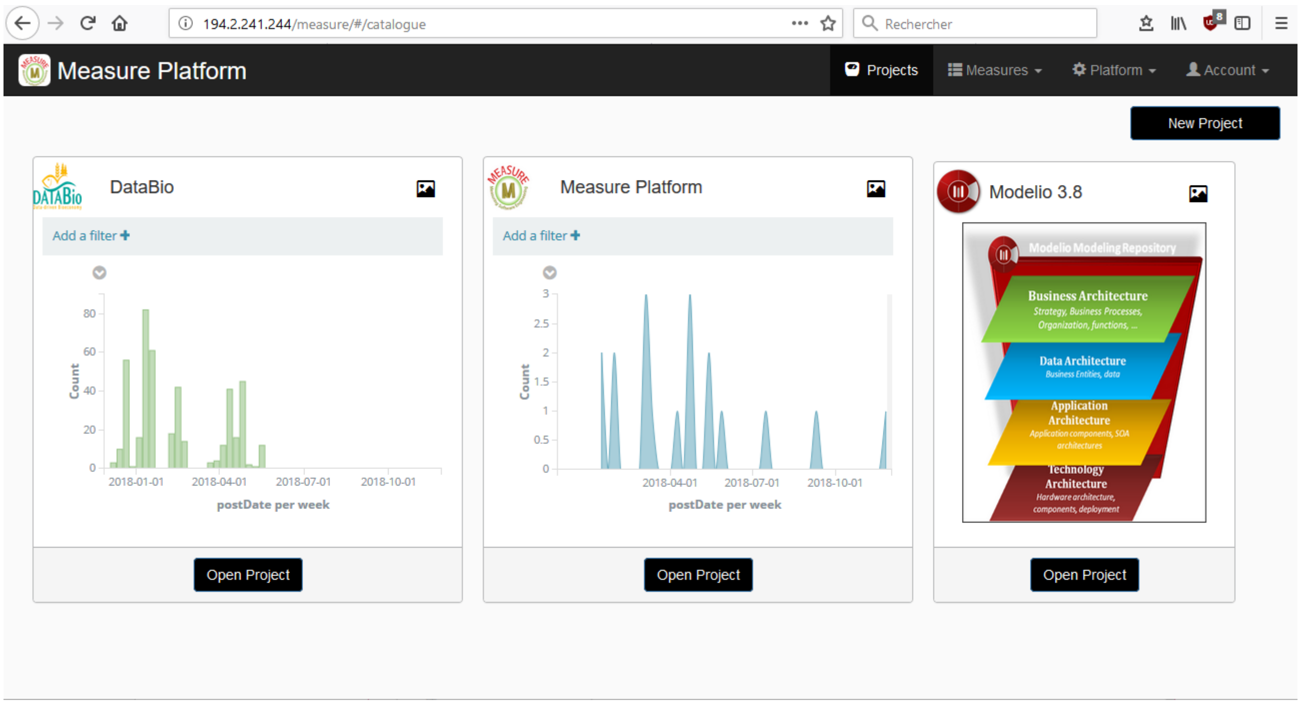


Figure 6. List of Projects

Technical architecture

The implementation of the decision-making component is based on a technological stack composed of Angular JS and Bootstrap as front end, Spring Boot and an SQL data base as back end. The representation of measures itself is provided by Kibana, a graphical framework which allows the representation of extracted data from an Elasticsearch database as complex charts.

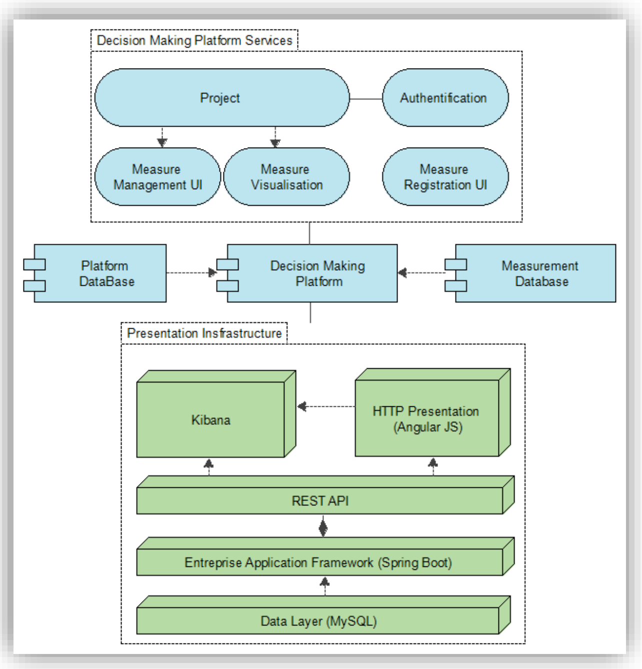


Figure 7. Architecture of the decision-making Platform

Visualise Measure with Kibana

The main goal of this web application is to provide dashboards which will aggregate some measures representations. This individual dashboard will be integrated and organized in a global web application in order to present this dashboard as several viewpoints of a unique decision making platform. We plan to use an existing open source dashboard solution to define individual visualization dashboards: Kibana.

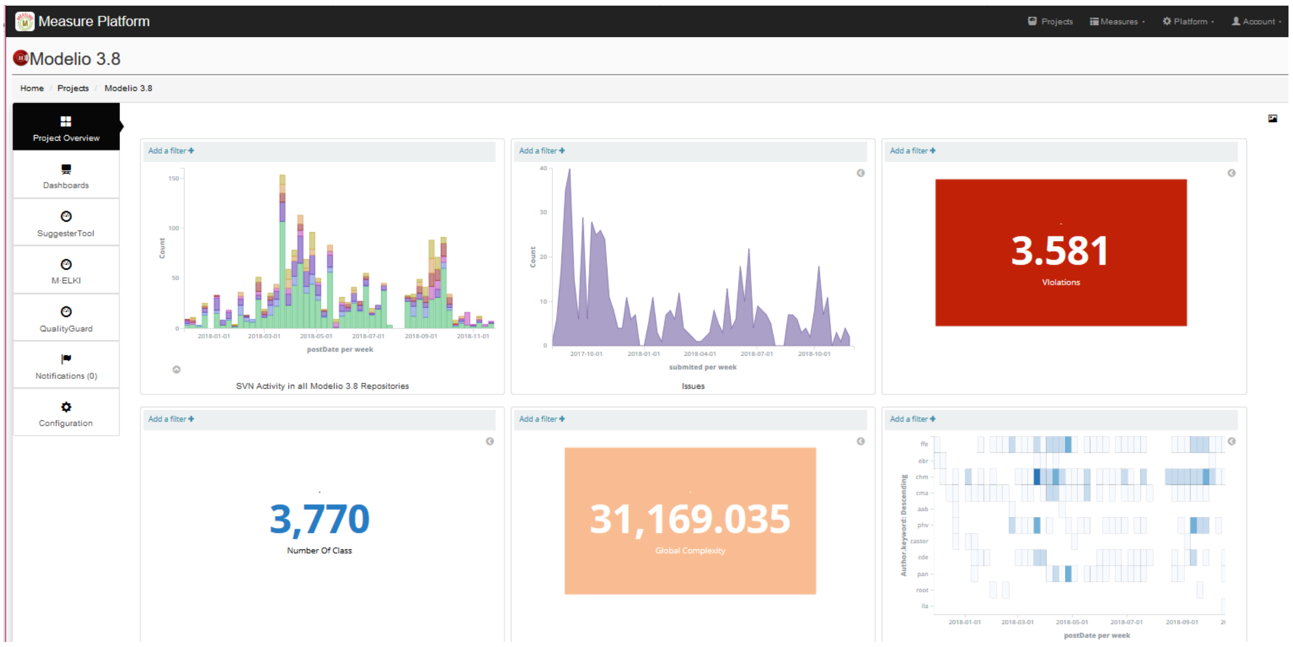


Figure 8. Examples of charts products by Kibana

Kibana can be used to search, view, and interact with data stored in Elasticsearch database. With Kibana, we can perform data analysis and visualize data in a variety of charts, tables, and maps. Dashboards defined using Kibana can be exported as Iframe and integrated in standard web application.

2.2.4. Analysis Platform Component

The main objective of the analysis platform implements analytics algorithms, to correlate the different phases of software development and to perform the tracking of metrics and their value. The platform also connects and assures interoperability among the tools and defines actions for improvement and possible countermeasures.

In order to support a large set of analyses services and do not limit to a specific technology, the Analysis Tools are external processes. Although external, we wanted a deep integration between the platform and the analysis tools. We solved this issue in the following way:

- The Measure platform provides a REST API which allows an analysis tool to register it on the platform, to receive notifications from the platform and access to information related to the project defined and the measures collected by the platform.
- On its side, the analysis tool provides some web pages which are embedded into the platform web application.

2.2.5. Integration Mechanism of Analysis Tool into

In order to ensure the integration of various kinds of analysis tools into the measure platform, the Analysis component provides an integration mechanism external tool to register it into the measure platform, to access to measurement data, to receive notifications from the platform and finally to provide an analysis of the project.

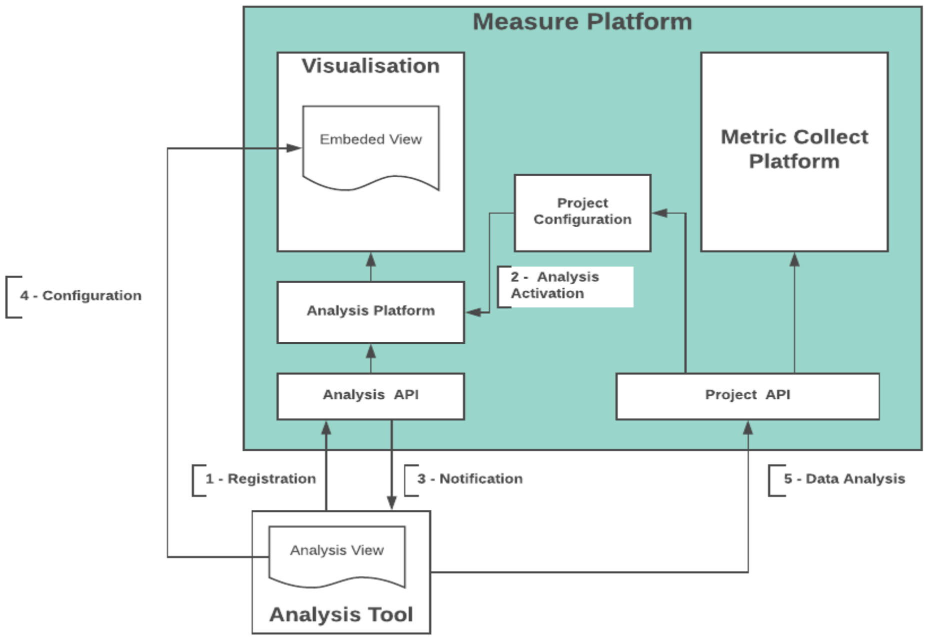


Figure 9 : Registration process of an external analysis tool into the platform

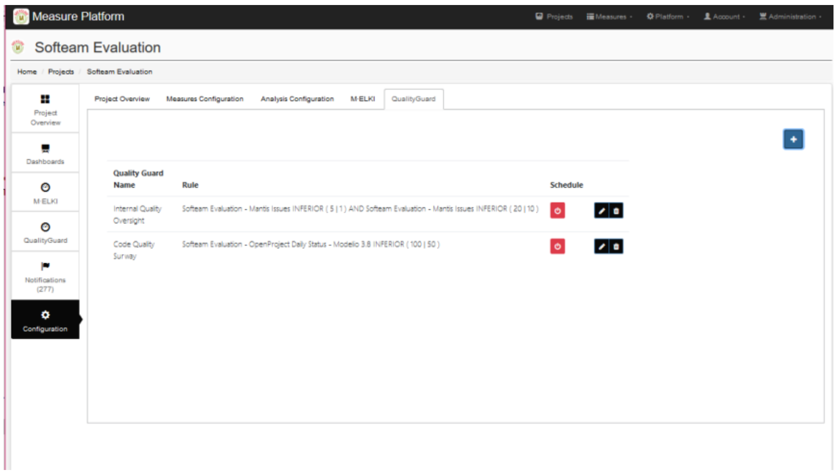
- **Registration**: At start-up of the Analysis Tool, it must register itself to the platform using the Registration service. This would allow the project to activate the analysis tools.
- **Wait for Notifications**: The Analysis Tool must listen to notifications from the platform in order to know when a project requests the usage of the analysis tool. The notification (Alert) system is based on pooling system. The Analysis tool pools the platform periodically using the alert service to received notifications.
- **Configure Analysis**: When a project activates an analysis tool, the analysis tool must configure it for the project and provide URLs for the project-specific configuration page, the project main view and optionally the dashboard cards.
- **Analyse the Project**: When configured, the analysis tool can start its analysis work for the specific project. In order to perform this work, the analysis tool can explore the project configuration using the various services provided by the Measure platform. It can also configure new Alerts to receive notifications when the project configuration has changed.

2.2.5.1 User Interface integration using Embedded View

In order to integrate deeply the analysis tool to the Measure Platform, the analysis tools have to provide some web pages which will be embedded to the platform web application. Each of these views is defined on the platform side by a specific URL. For project specific views, this URL is different for each project. You will see below the list of view which can be provided by the analysis tool and embedded into the Measure Platform.

- **Global Configuration Page (optional):** If the analysis tool requires a way to provide some configuration interface which will be shared by all project, it can provide a global configuration web page.

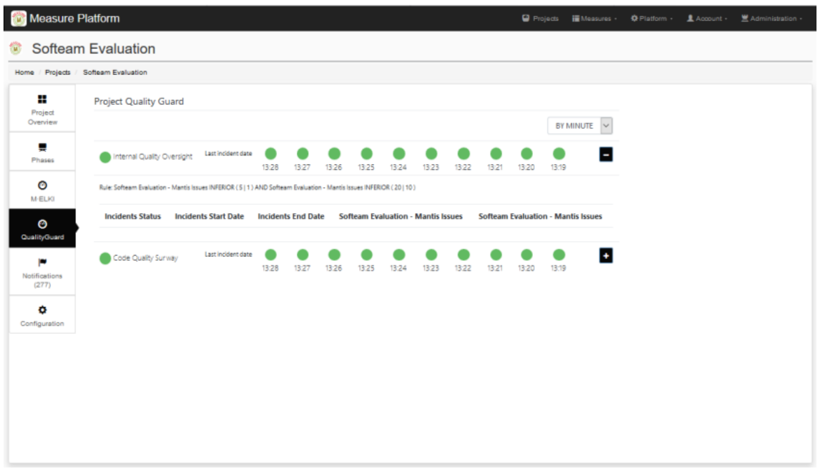
- **Project Specific Analysis Configuration page:** Configuration pages that are specific for each project. This page is embedded into the project configuration page and allows to configure the analysis service provided by the external analysis tool.



Quality Guard Name	Rule	Schedule
Internal Quality Oversight	Softeam Evaluation - Mantis Issues INFERIOR (5 | 1) AND Softeam Evaluation - Mantis Issues INFERIOR (20 | 10)	
Code Quality Surway	Softeam Evaluation - OpenProject Daily Status - Modelio 3.8 INFERIOR (100 | 50)	

Figure 10 : Analysis tool configuration page of Quality Guard Analysis tool

- **Analysis Tool Main View:** Main view of the analysis tool that is specific for each project. In this view, the analysis service.



Internal Quality Oversight — Last incident date — 13:28 13:27 13:26 13:25 13:24 13:23 13:22 13:21 13:20 13:19

Rule: Softeam Evaluation - Mantis Issues INFERIOR (5 | 1) AND Softeam Evaluation - Mantis Issues INFERIOR (20 | 10)

Incidents Status	Incidents Start Date	Incidents End Date	Softeam Evaluation - Mantis Issues	Softeam Evaluation - Mantis Issues

Code Quality Surway — Last incident date — 13:28 13:27 13:26 13:25 13:24 13:23 13:22 13:21 13:20 13:19

Figure 11 : Main view of the Quality Guard Analysis Tool

- **Dashboard Card:** Optional small view that can be integrated to projects dashboards in order to provide some key information to project managers related to the service provided by the analysis tool.

2.2.5.2 Platform Querying Services

The platform provides several other services which can be used by the analysis tools to retrieve platform and project configurations data, information related to measures and measurements and more.

The list of available services can be consulted via Swagger directly on the deployed Measure platform. To access this specification, one must be connected as Administrator to the platform. The complete API specification is available on Administration > API menu

Some examples of available HTTP services:

- GET /api/measure/findall : List all measures

- GET /api/measure/{id} : information related to a specific measure
- GET /api/measure-properties/{id} : List of scope properties associated with one measure
- GET /api/projects : List all projects
- GET /api/projects/{id} : Information related to a specific project
- GET /api/phases/byproject/{id} : Get phases of a specific project
- GET /api/phases/{id} : Information of a specific phase
- GET /api/measure-instances : List of all measure instances
- GET /api/measure-instances/{id} : Information of a specific measure instance
- GET /api/project-measure-instances/{id} : List of measure instances of a specified project
- GET /api/measure-instance/scheduling/execute/{id} : Execute a specific measure
- GET /api/measure-instance/scheduling/start/{id} : Activate scheduling of a specific measure
- GET /api/measure-instance/scheduling/stop/{id} : Deactivate scheduling of a specific measure

2.3. Platform Agent

As presented before, SMM Measures are auto-executable components used by the platform to collect measurements. In the majority of the cases, measures are executed on platform side and manage a connection with external data sources to collect required data. But it's not always possible to execute measure on platform for various reasons like network configuration, security policy, scalability or because of the mechanisms used by the measure to collect data. To address this issue, we have developed an autonomous agent which provide the fooling services:

- Allow to execute direct measure on client side, closer of measured element instead of execute it on platform side.
- Ensure the scalability of the Measure platform by providing a way to execute time-consuming measures on remote computers while maintaining a centralized control of measure's execution.
- Solve common issues related to Network Security rules in Industrial context by allowing to deploy the Measure Platform different networks than where the monitored data sources is stored.

The Platform Agent in an autonomous application connected to the Measure Platform via the Communication API. It allows to deploy and execute measure on client side du the configuration of measure execution stay administrated remotely by the platform.

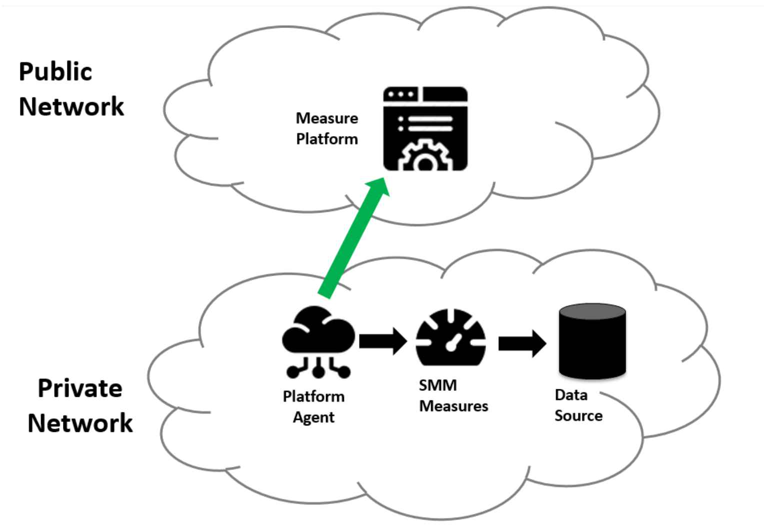


Figure 12: Platform agent in Private network

Technical architecture

The Agent is a sub component of the Collect and Compute Component. Based on a Spring Boot application, it integrates a light version of the SMM Engine whit the ability to execute Direct Measures. The execution itself is managed by the Measure Platform using the REST API provided by the agent.

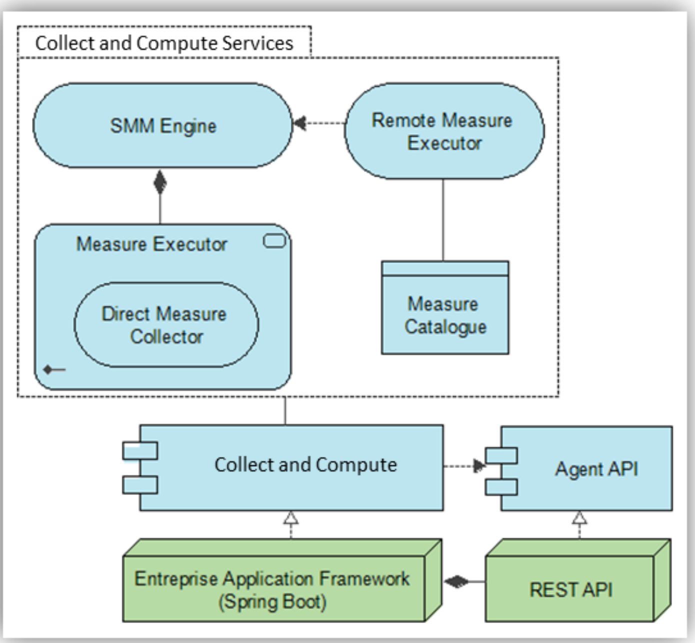


Figure 13. Architecture overview of Agent Component

Execute measures on agent

- When an agent is deployed, the agent registers the set of measures it can manage on the platform.
- Measure Instances are configured on Platform side as would have been the case for measurements executed on the platform.
- When the platform required the execution of the measure, it delegates the execution to the remote agent.

- The result of the execution is returned to the platform who manages the storage of this measure into measurement database.

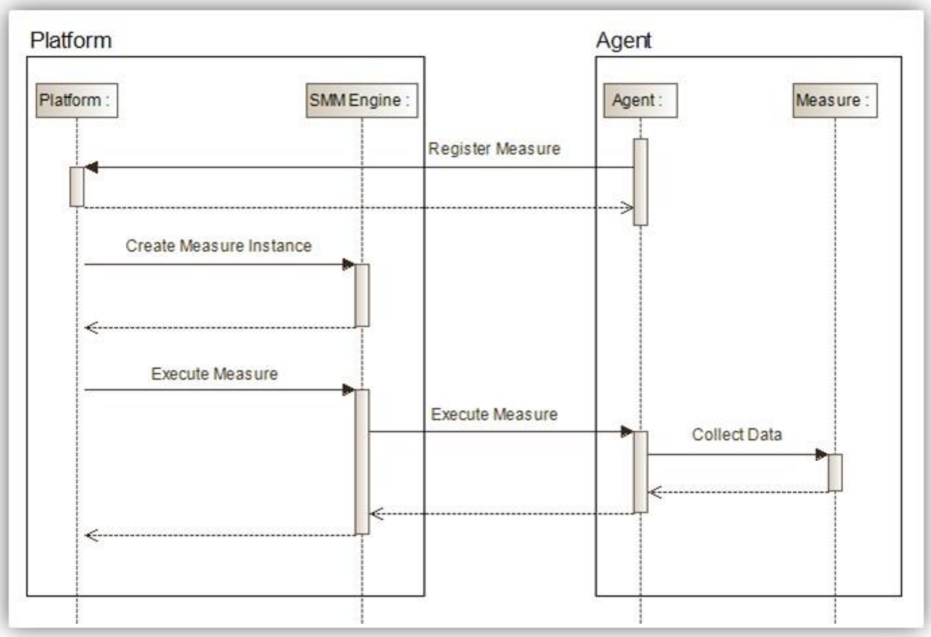


Figure 14. Measure execution on agent

2.4. Measure

A Measure is a small and autonomous java program based on the SMM specification which allow to collect measurements. A Measure can be Direct (Collect of measurement in physical world), a Proxy (Ensure communication between a Measurement Tool and the Platform) or Derived Measure (Measure calculated by the aggregation of existing Measures).

2.4.1. Measure Packaging

The SMM specification provides a standardized way to exchange measure definition. Unfortunately, they promote XMI (Xml Model Interchange) to realize this operation. As indicated by his name, this format has been designed to exchange models between modelling tools.

In context of Measure Project, we realized that this format is not adapted to register Measure definition in the Platform. We plan to use a format more appropriate to our requirement: Each measure definition would be composed of two elements: an XML file which store meta-data of the Measure and another file which will contain the Measure implementation.

In SMM, direct and calculated measures definitions are associated with an Operation which represents the implementation of the measure. These operations can be expressed in natural language or may contain executable code. In order to be able to collect direct measures and to execute calculated measure, we have to choose a common executable language. For now, we plan to use Java.

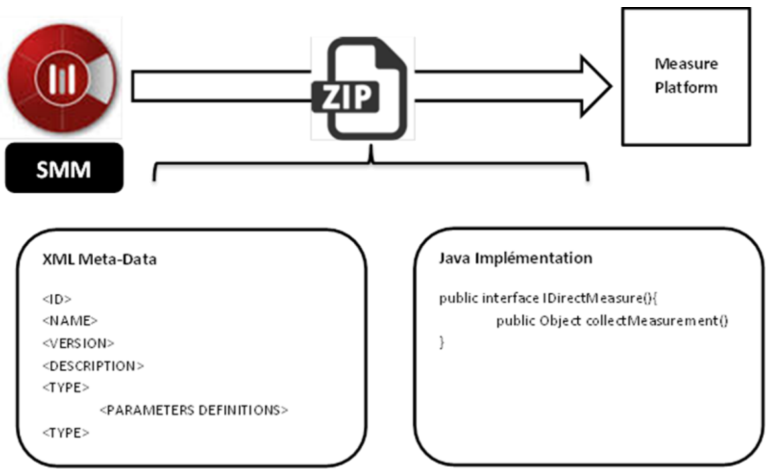


Figure 15. Measure Packaging

2.4.2. Measure Implementation

A SMM Measure is a small and independent software component which allows retrieving or calculates Measurement. The implementation of a measure is based on a library developed in parallel of the Measure Platform: the “SMMMeasureApi”.

A direct Measure is used to collect data in physical world. This kind of measure can be executed on Platform Side on Client Side. To define a Direct Measure, the IDirectMeasure has to be implemented. This interface will be called by the Measure Platform to retrieve the measurements.

```
Public interface IDirectMeasure {
  public List<IMeasurement> getMeasurement() throws Exception;
  public Map<String,String> getProperties();
}
```

- **getMeasurement()**: Calculate and Return a list of Measurement.
- **getProperties()**: Provide a way for the Measure Platform to communicate properties to DirectMeasure implementation.

A Derived measure is used to define a combined measure which will calculate new measurements using one or more measurements stored on measure platform. To define a Direct Measure, the IDerivedMeasure has to be implemented.

```
Public interface IDerivedMeasure {
 public List<IMeasurement> calculateMeasurement() throws Exception;
 public void addMeasureInput(String reference, String role, IMeasurement value);
 public Map<String,String> getProperties();
}
```

- **calculateMeasurement()**: Calculate and Return a list of Measurement based on provided measurement inputs.
- **addMeasureInput()**: Provide a way for the Measure Platform to communicate input measurement into the Derived Measure implementation.
- **getProperties()**: Provide a way for the Measure Platform to communicate properties to the Derived Measure implementation.

2.4.3. Measure Data Model

In the second version of the platform, a structure of the measurement returned by a Direct or by a Derived measure can be complex, it means that the collected measurement can be composed of several fields of various types. The Format of data returned by the measure is specified in the measure-metadata file.

```
<unit name="MantisIssue">
        <fields fieldName="name" fieldType="u_text"/>
        <fields fieldName="project" fieldType="u_text"/>
        <fields fieldName="priority" fieldType="u_text"/>
        <fields fieldName="severity" fieldType="u_text"/>
        <fields fieldName="reproductibility" fieldType="u_text"/>
        <fields fieldName="status" fieldType="u_text"/>
        <fields fieldName="os" fieldType="u_text"/>
        <fields fieldName="platform" fieldType="u_text"/>
        <fields fieldName="version" fieldType="u_text"/>
        <fields fieldName="submited" fieldType="u_date"/>
        <fields fieldName="lastupdate" fieldType="u_date"/>
</unit>
```

The list of data types supported by the platform is presented in the table below:

Kind	Data Type
String	u_text, u_ keyword
Numeric	u_long, u_integer, u_short, u_byte, u_double, u_float, u_half_float, u_scaled_float
Date	u_date
Boolean	u_boolean
Binary	u_binary
Range	u_integer_range, u_float_range, u_long_range, u_double_range, u_date_range
Geo	u_geo_point, u_geo_shape

2.4.4. Measure Visualisation

In the second version of the platform, we decide to integrate the possibility to define default visualization schema (how the measure its display as graphic in project dashboard) in measure packaging.

The integration of this visualization configuration has as a goal to facilitate the usage of the platform by automate the creation of visualization in the project dashboard when a new measure is integrated into a project and so facilitate the adoption of the platform by new users.

Configuration of this visualization required the definition of a set of new properties in measure-metadata file which will be used by the platform to recreate a visualization.

- view_type : Type of Graphic (Area, Line, Simple Value, Bar, Complex)
- visualised_property : Name of the field of the measure data model which will be displayed
- time_periode : Periode of time displayed in the graph
- date_index : Name of the field of the measure data model used as an access of the graph

Based on these parameters, a measure visualisation is automatically created in Project Dashboard when the measure is created.

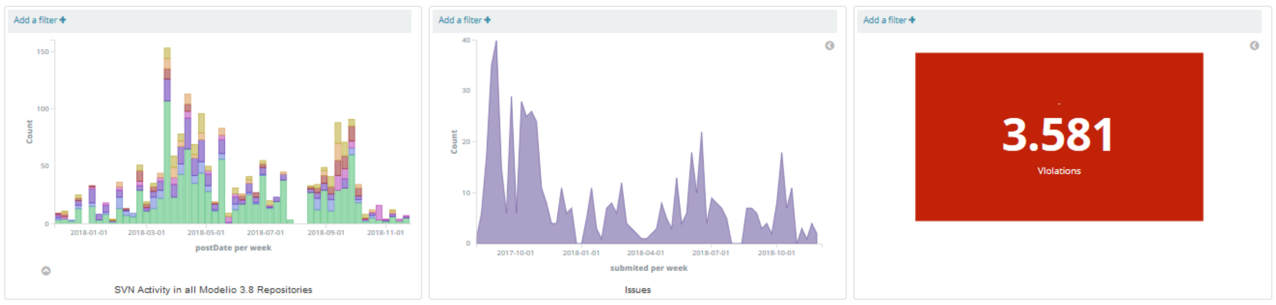


Figure 16 : Measure visualisations

2.4.5 Examples of implemented measures

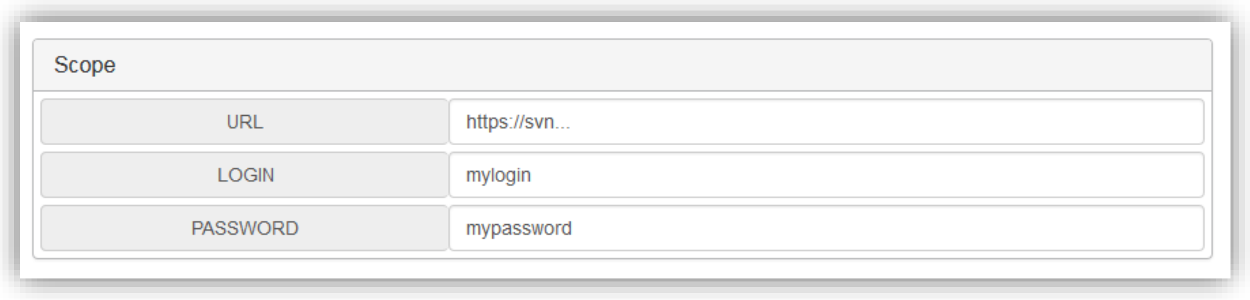
2.4.5.1 Cognitive complexity

As detailed in the MEASURE deliverable D2.2, this measure computes the cognitive weight of a Java Architecture. The cognitive weight represents the complexity of a code architecture in terms of maintainability and code understanding.

Scope

The scope (named as *OO architecture* in the deliverable D2.2) of this metric is represented by three elements: the URL of a SVN repository containing the measured Java project, the login and the password necessary to connect to this repository.

Those elements are added by the user as shown in the figure below:



Scope

URL	https://svn...
LOGIN	mylogin
PASSWORD	mypassword

Figure 17. Scope of the cognitive complexity measure

Implementation

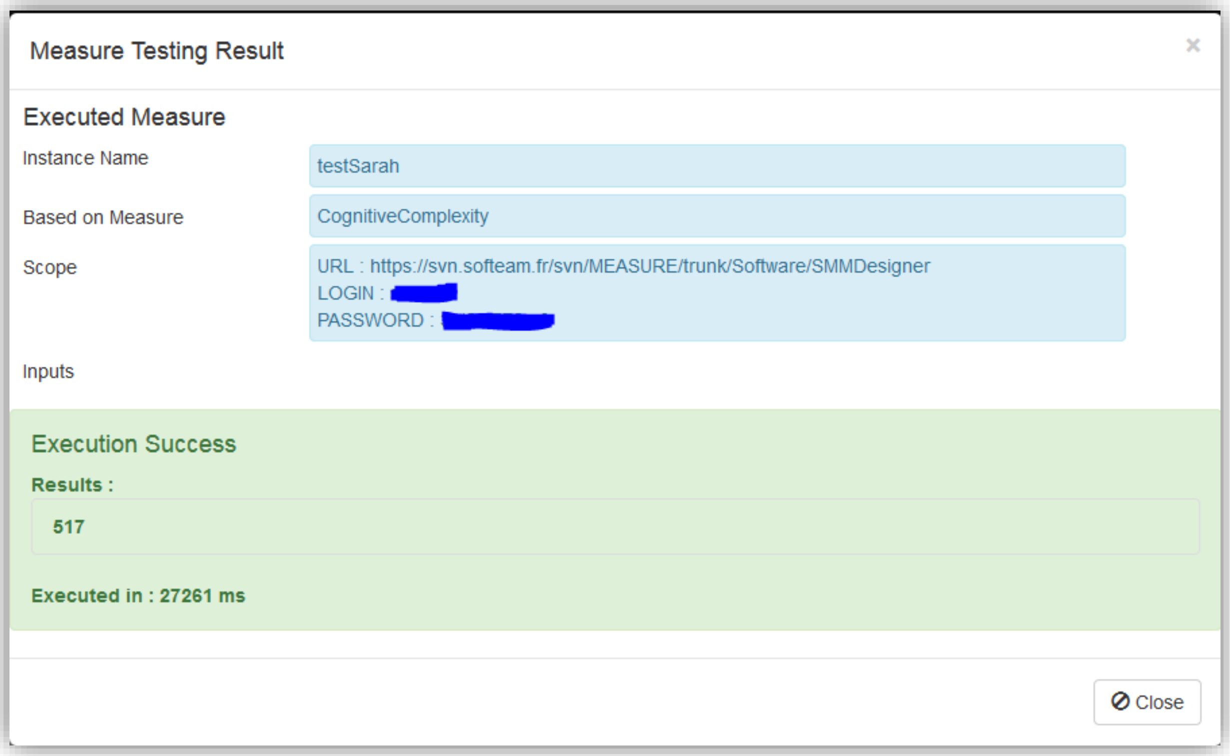
The implementation of that measure computes the cognitive weight of a Java project by adding the cognitive weight of each method of each class. The GitHub of the measure is available at: *https://github.com/ITEA3-Measure/Measures/tree/master/CognitiveComplexityMeasure*

In order to compute the cognitive weight of a method, we focus on the nested level of conditional statements/blocks (*if-then-else*) and the *for* and *foreach* loops. As shown in the pseudocode below, each level is associated with a weight (a number) and thus all these weights are summed. Therefore, we add the weights of all methods to have the cognitive weight of a class.

```
CognitiveWeight(statements block, integer weight, integer level):
    For each statement S of block
        If S is conditional statement or for statement of foreach statement:
            level2=level+1
            weight2=weight+level
            S1 = block1 of S
            CognitiveWeight(S1,weight2,level2)
        End If
    End For
End
```

Execution result

The result of the cognitive measure is the number representing the cognitive weight of the project, in other terms, it returns the sum of the weight of each method of each class of the project provided as the scope parameter. This is illustrated in the figure below, a screenshot of the current MEASURE platform:



Measure Testing Result

Executed Measure

Instance Name: testSarah

Based on Measure: CognitiveComplexity

Scope: URL : https://svn.softeam.fr/svn/MEASURE/trunk/Software/SMMDesigner
LOGIN : [redacted]
PASSWORD : [redacted]

Inputs

Execution Success

Results :

517

Executed in : 27261 ms

Figure 18. Cognitive measure result provided through the current MEASURE platform

2.4.5.2 CPU usage

This measure computes the CPU usage by the execution of the JVM and by the whole system which execute the JVM.

Scope

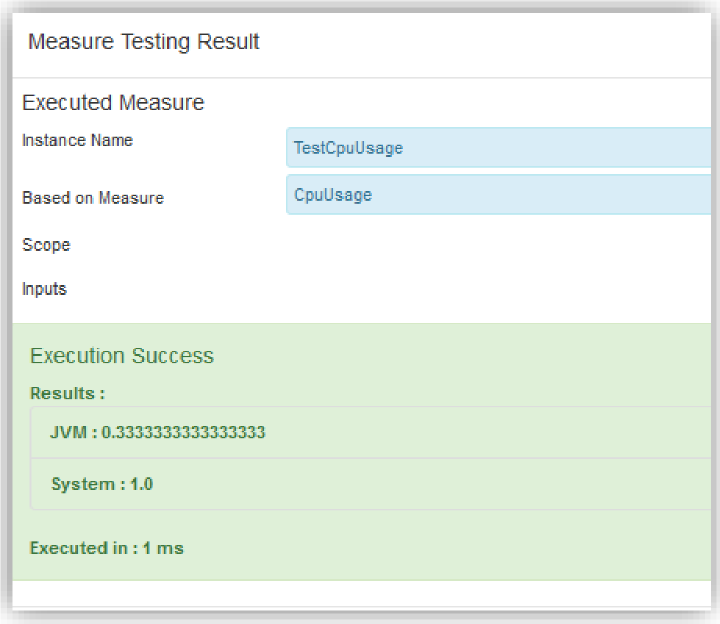
The scope is the JVM but to launch the measure, there is no scope properties to insert.

Implementation

The implementation of that measure uses the library OperatingSystemMxBean to return the CPU usages of the JVM execution and of the entire system executing the JVM. It returns two Double corresponding to the both types of CPU usage computed. Those results are either between 0.0 and 1.0 or a negative value if the CPU usage is not available. The GitHub of the measure is available at: *https://github.com/ITEA3-Measure/Measures/tree/master/CpuUsageMeasure*

Execution result

The result of the CpuUsage measure is represented by two numbers associated to the type of CPU usage computed (the JVM and the System). This is illustrated in the figure below, a screenshot of the current MEASURE platform:



Measure Testing Result

Executed Measure

Instance Name	TestCpuUsage
Based on Measure	CpuUsage
Scope	
Inputs	

Execution Success

Results :

JVM : 0.3333333333333333

System : 1.0

Executed in : 1 ms

Figure 13 – CpuUsage measure result provided through the current MEASURE platform

2.4.5.3 Memory usage

This measure computes the memory usage of the heap, memory reserved for the objects allocation by the JVM and the memory usage used by the JVM for its execution (the heap memory excluded).

Scope

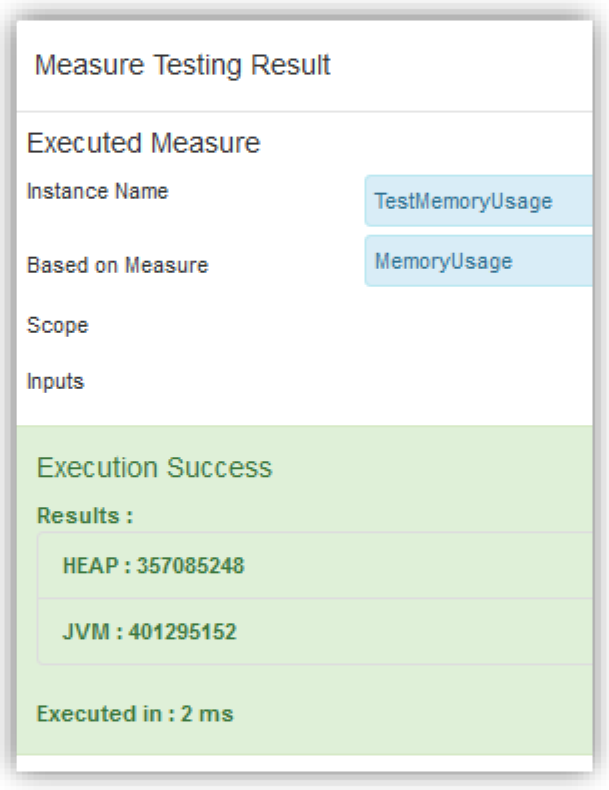
The scope is the JVM but to launch the measure there is no scope properties to insert.

Implementation

The implementation of that measure uses the library MemoryMxBean to return the memories usages for the JVM execution. It returns two Long corresponding to the amount of the different memories used in bytes. The GitHub of the measure is available at: *https://github.com/ITEA3-Measure/Measures/tree/master/MemoryUsageMeasure*.

Execution result

The result of the MemoryUsage measure is represented by two numbers associated of the type of memory usage computed (the Heap and the JVM). This is illustrated in the figure below, a screenshot of the current MEASURE platform:


Measure Testing Result

Executed Measure

Instance Name	TestMemoryUsage
Based on Measure	MemoryUsage
Scope	
Inputs	

Execution Success

Results :

HEAP : 357085248

JVM : 401295152

Executed in : 2 ms

Figure 19. Memory Usage measure result provided through the current MEASURE platform

2.4.5.4 Message exchange rate

This measure computes the number of messages exchanged over a certain communication channel in a given period of time. The motivation of this metric comes from the distributed testing setting. In the security domain, a popular dynamic testing approach is fuzz testing, where many inputs are (randomly) generated and the application under test is run with these inputs, observing its behavior. Given the large number of inputs to test during fuzz testing, modern architectures run several instances of the application under test in parallel. The distribution of input data to the parallel machines uses a message-oriented middleware, such as RabbitMQ[2], which implements various message queuing protocols.

The measure will compute the message exchange rate, i.e., the number of messages distributed over a certain channel in a time interval determined by two measurements:

MessageExchangeRate := [no of exchanged messages] / [time interval length]

Scope

The scope of this metric is represented by four elements: (1) the URL of the host where the message broker resides, (2) the login and (3) the password necessary to connect to the host, and also (4) the monitored communication channel.

Implementation

The implementation is tailored to the RabbitMQ, which is the most popular open-source message-oriented middleware. In order to compute the metric, we instantiate a RabbitMQ monitor instance which counts the number of messages distributed by a certain so-called “exchange” (which is the RabbitMQ broker between

[2] *https://www.rabbitmq.com*

the message producers and consumers via several queues). The time interval length is the difference between the current time of the measurement and the last saved time of the measurement.

The measure is implemented using the Agent Component of the MEASURE platform to collect the data on a remote host. The current implementation can be found at:
https://github.com/ITEA3-Measure/Measures/tree/master/MessageExchangeRateMeasure

Execution result

The result of the MessageExchangeRate measure is represented by the number representing the message exchange rate over the period of time determined by the scheduler of the measure instance.

2.5. Monitoring Application

Although we tried to facilitate as much as possible the process of deploying and visualizing measures with the platform, the process can still be complex depending on the complexity of the measure it-self. To integrate a simple Measure, the current process required to:

1. Deploy the Measure in Measure Platform catalogue
2. Create a measure instance (instantiation of the measure in a project)
3. Configure the measure instance (credential, URL of monitored resources)
4. Start the data collection process
5. Create a Visualization in a dashboard

Although useful in the case of isolated measures, this process becomes problematic when we try to create a complete dashboard of measure addressing the same data source (a dashboard related to a GitHub repository as example composed of several measures about commits, issues, projects users,…).

To address this issue, we have imagined a new way to package and deploy a set of measures dedicated to the functional scope: The Monitoring Application.

The Monitoring Application is composed of a set of metrics which addresses the same data source. These metrics are packaged together and come with a configuration file allowing to specify how these metrics will be display.

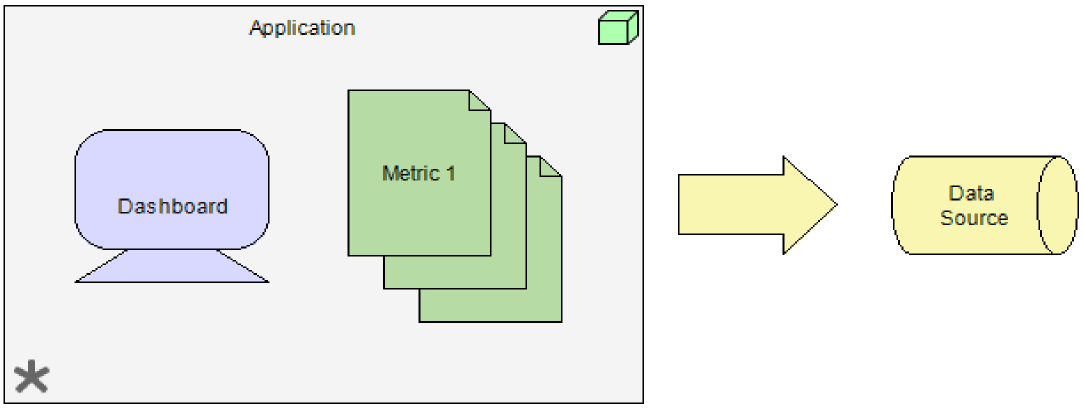


Figure 20 : Monitoring Application

The Application is pre-deployed on the platform on a specific Application catalogue like standard Metrics. When a customer decides to integrate this Monitoring Application in an existing platform project, all Measures associated with this application are automatically deployed in the project and an automatic dashboard is created.

To illustrate the interest for the customer to use an Application instead of measure, our will take the example of someone wishing to monitor an SVN repository. To achieve this goal, the customer has to:

Select the SVN Application to add it in his project

Configure his GitHub credentials

As a result, he will obtain automatically a complete up to date dashboard addressing several aspects of the GitHub repository like history of commit, list of committers, list of issues associated with the project...

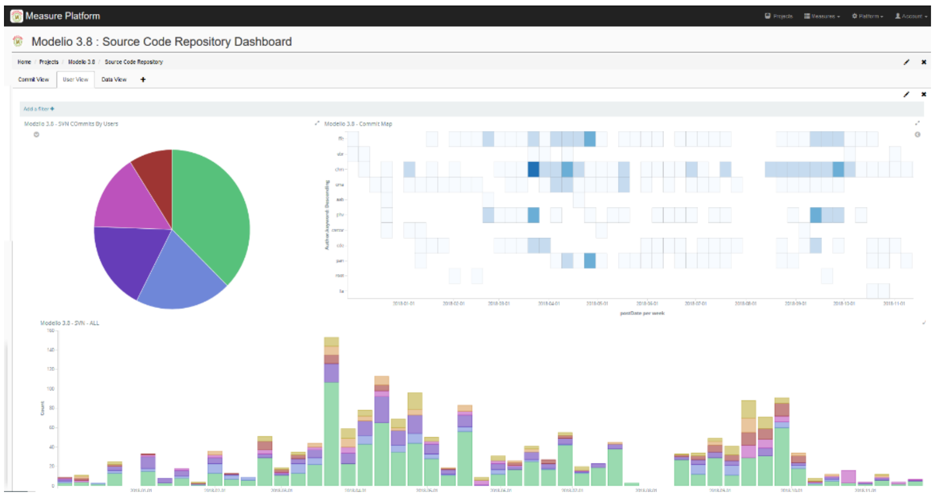

Measure Platform
Modelio 3.8 : Source Code Repository Dashboard
Home / Projects / Modelio 3.8 / Source Code Repository
Commit View
User View
Data View
Modelio 3.8 - SVN Commits By Users
Modelio 3.8 - Commit Map
Modelio 3.8 - SVN - ALL
Count

Figure 21: GitHub Measurement Application dashboard

3. The measuring tools

3.1. Introduction

Several metrics has been specified in SMM in D2.2 deliverable: Formal specification of MEASURE metrics. Some of them are direct metrics to be gathered by measuring tools. In the following section, we present a set of tools provided by different tool providers in MEASURE project to support these direct measurements. This set of tools will be extended during the project to cover more metrics.

3.2. Hawk: Model Measurement Tool of Softeam

3.2.1. Overview

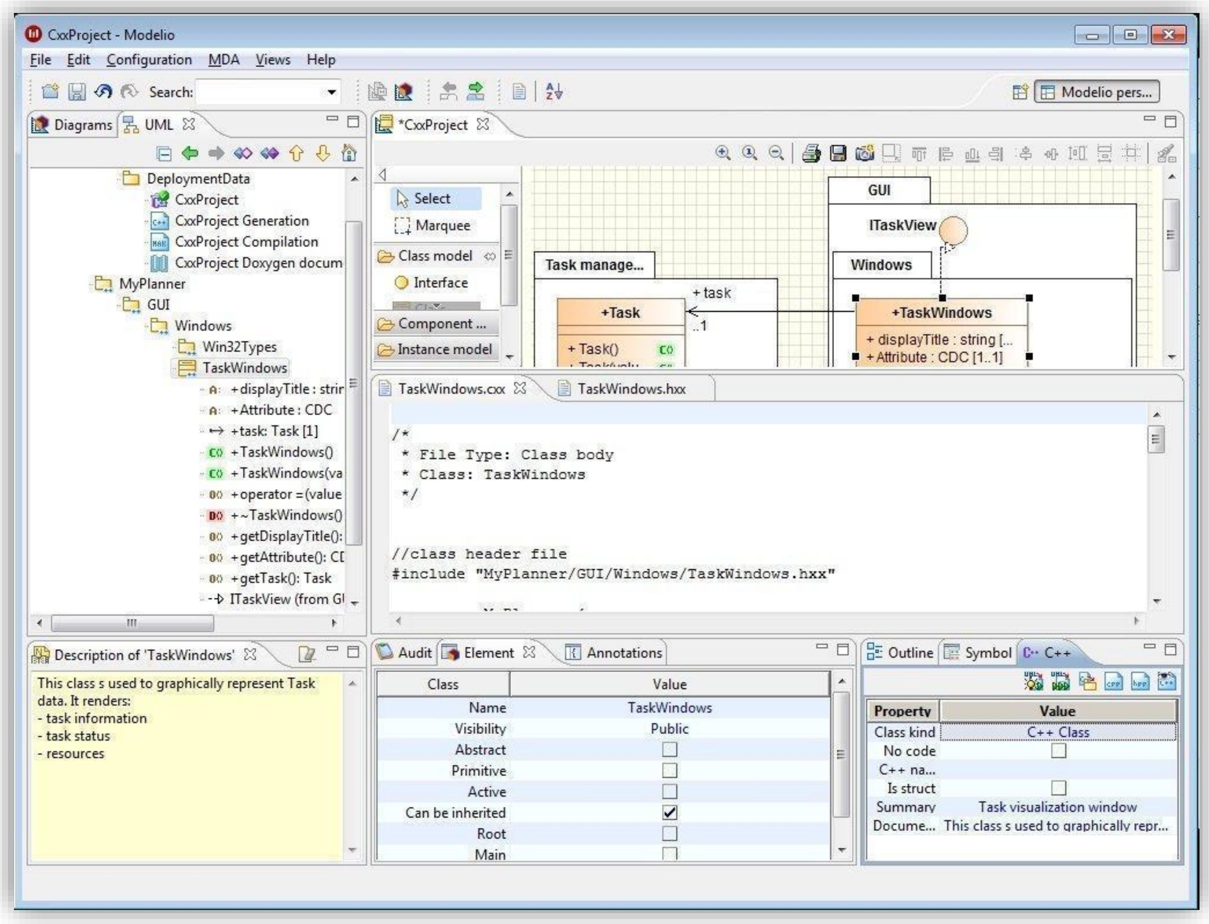
As Model Driven Engineering (MDE) is increasingly applied to larger and more complex systems, the current generation of modelling and model management technologies have been pushed to their limits in terms of capacity and efficiency, and as such, additional research is needed to enable MDE to remain relevant within industrial practice and to continue delivering its well-recognized benefits of increased productivity, quality, and maintainability [8].

Model Driven Engineering (MDE) is a software development methodology that uses high-level models as central artefacts in a software development activity. MDE derived methodologies have been shown to increase productivity by a factor of 10 and to increase non-functional qualities of the produced artefacts, such as, maintainability, consistency and traceability. The software development world currently experiences however an unprecedented growth in the complexity of the developed systems. This complexity manifests itself either in terms of the variety of domains software needs to bridge, but also in the diversity in backgrounds of modelers, and in the amount of people involved on a given project.

The current generation of modelling and model management technologies has been pushed to its limits in terms of capacity and efficiency. It turns out that modelling driven engineering, as most of current development practices, is inherently dependent on complex tools that serve as integrated development environments to be effective. However, existing modelling systems were built to support the requirements of modelling projects from decades ago. Any non-trivial MDE project involves querying and manipulation of a substantial number of models. Model queries are primary means to extract views and to formally capture and validate well-formedness constraints, design rules and guidelines on the fly. Therefore, scalability of model queries is a key element in any scalable MDE solution. Our experience with industrial case studies is that current transformation technologies do not scale, which discourages some potential adopters from using MDE.

The measures and analyses tools applied to Models are particularly subject to these scalability constraints as measurements must be performed periodically and the execution of a measure usually requiring browse an important part of Models. In Context of Measure Project and to address these issues, we have chosen to design and deploy a specific measurement tool dedicated to models and based on an indexing tool supporting complex queries.

Even though these tools have been studied to support several modeling tools, we will use this tool as part of this project to perform measurements on Modelio modeler. Modelio [12] is a comprehensive MDE workbench tool supporting the UML2.x standard. Modelio provides a central IDE which allows various languages (represented as UML profiles) to be combined in the same model. Modelio proposes various extension modules, enabling the customization of this MDE environment for different purposes and stakeholders.



```
/*
 * File Type: Class body
 * Class: TaskWindows
 */


//class header file
#include "MyPlanner/GUI/Windows/TaskWindows.hxx"
```

This class s used to graphically represent Task data. It renders:
- task information
- task status
- resources

Class	Value
Name	TaskWindows
Visibility	Public
Abstract	☐
Primitive	☐
Active	☐
Can be inherited	✓
Root	☐
Main	☐

Property	Value
Class kind	C++ Class
No code	☐
C++ na...	
Is struct	☐
Summary	Task visualization window
Docume...	This class s used to graphically repr...

Figure 22. Modelio MDA Tool

3.2.2. Architecture

The Model Measurement Tool will be built integrating an advanced version of a tool developed by the University of York and University of Aston in the context of the MONDO project: The Hawk indexing tool [7] [8].

The following describes Hawk as it is presented in MONDO Deliverable D5.5 – Model Indexing Framework - Final Version [9] [10].

Hawk is a heterogeneous model indexing framework offering an orthogonal approach for achieving scalable model querying on large collections of file-based models. Hawk has a component-based architecture, making it possible to be extended in various ways, by providing additional components to:

- **Model parser components:** these components provide parsers for specific model persistence formats, such as Ecore models persisted in XMI or Modelio models persisted in EXML.
- **Model indexer components:** these components, specific for each back-end used (such as Neo4j[3] or OrientDB[4] NoSQL databases), receive a model resource and a file revision number/identifier and insert the former into the database.

[3] http://neo4j.com/

[4] http://orientdb.com/orientdb/

- **VCS components:** specific for each version control system, these components take a VCS URL and a revision number/id (representing the current top-level revision in the relevant index) and compute a set of changed files with respect to that revision. Currently supported version control systems are Subversion repositories, local folders, Git repositories and Eclipse workspaces.

- **Query API:** provides a bridge between Hawk and modelling and model management tools that need to query its indexes. The query API has been used to integrate model management frameworks such as Epsilon [7] or the query languages developed in the MONDO project [9].

Hawk operates by watching over a collection of version control systems and maintaining an integrated view of the latest version of all their models in a NoSQL graph database. Using this graph database, it is possible to answer global queries without needing to reload all the files. It is also possible to use indexed and derived attributes to speed up query execution [7].

The model indexes built by Hawk are property graphs, with their structure shown in the Figure below:

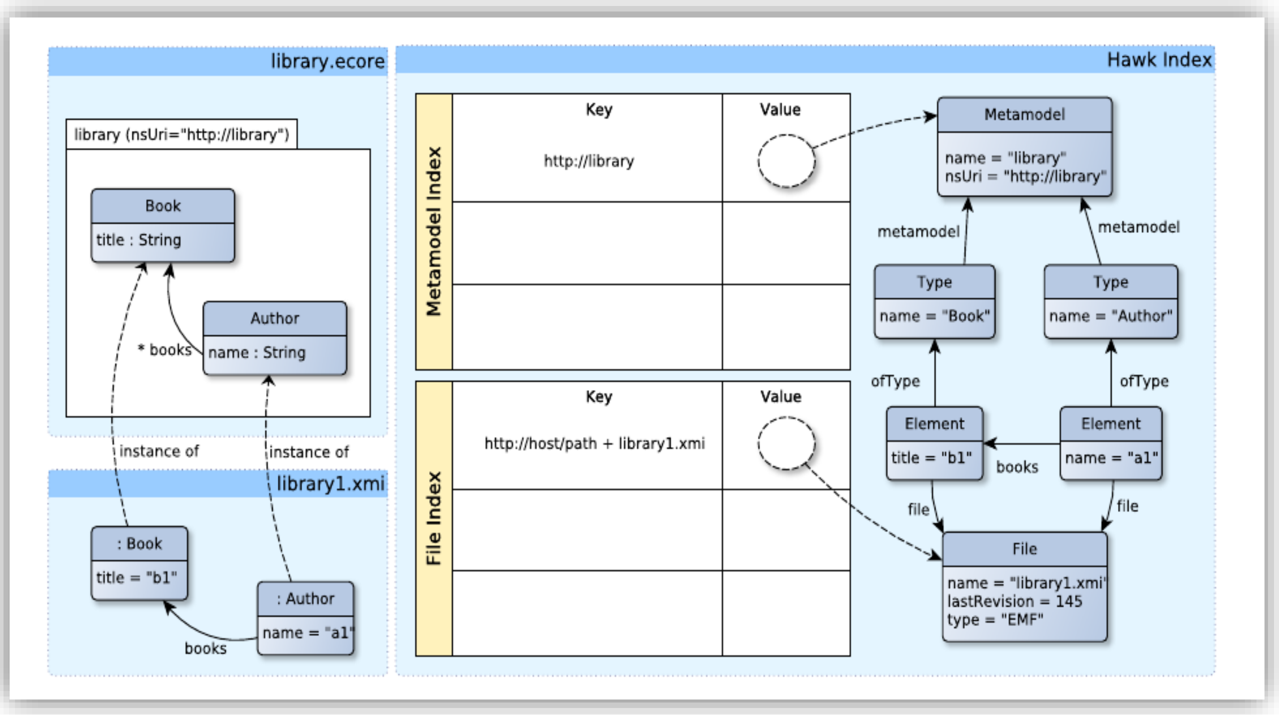


Figure 23. Hawk indexing tool Architecture Overview

Hawk includes a server that exposes its functionality through a set of Thrift APIs. This server product is a headless Eclipse application that can be run from the command line. The general structure is as shown here:

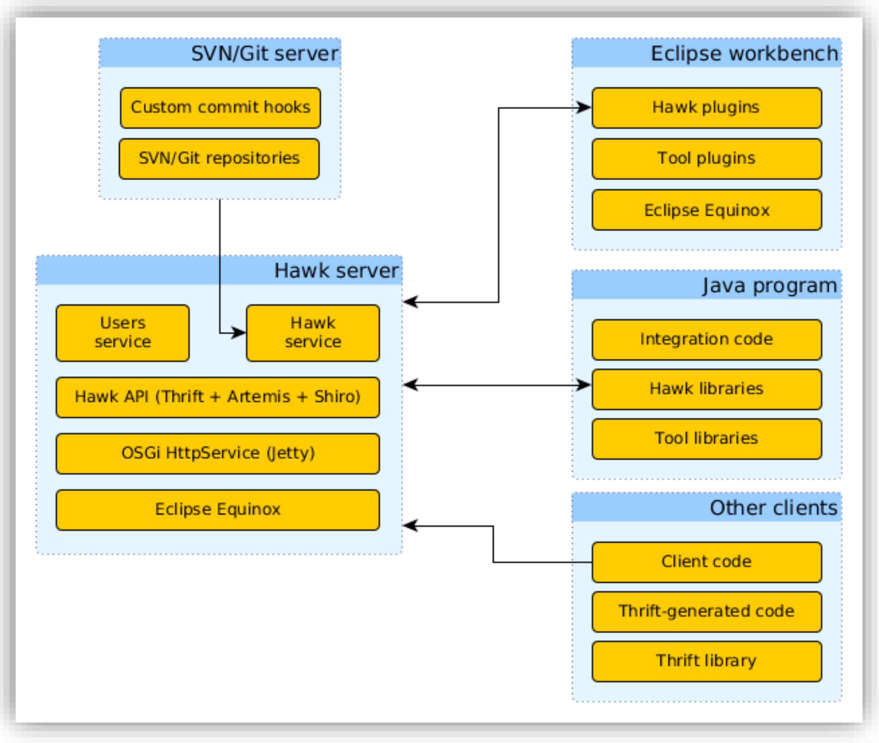


Figure 24. Hawk Server Communications

The server component is implemented as an Eclipse application, based on the Eclipse Equinox OSGi runtime. Using Eclipse Equinox for the server allows for integrating the Eclipse-based tools with very few changes in their code, while reducing the chances of mutual interference. The OSGi class loading mechanisms ensure that each plugin only "sees" the classes that it declares as dependencies, avoiding common clashes such as requiring different versions of the same Java library or overriding a configuration file with an unexpected copy from another library.

All services provide a JSON endpoint, since it is compatible across all languages supported by Thrift and works well with web-based clients. However, since Hawk is performance sensitive (as we might need to encode a large number of model elements in the results of a query), it also provides endpoints with the other Thrift protocols. Binary is the most portable after JSON, and Tuple is the most efficient but is only usable from Java clients. Having all four protocols allows Hawk clients to pick the most efficient protocol that is available for their language.

3.2.3. Interfaces

Hawk contains an up-to-date global index of all relevant models in its monitored VCSs, for it to be of practical value, Hawk needs to be able to provide correct and efficient responses to queries made on its model indexes.

The Epsilon platform is an extensible family of languages for common model management tasks and includes tailored languages for tasks such as model-to-text transformation (EGL), model-to-model transformation (ETL), model re-factoring (EWL), comparison (ECL), validation (EVL), migration (Flock), merging (EML) and pattern matching (EPL). All task-specific languages in Epsilon build on top of a core expression language – the Epsilon Object Language (EOL) – to eliminate duplication and enhance consistency.

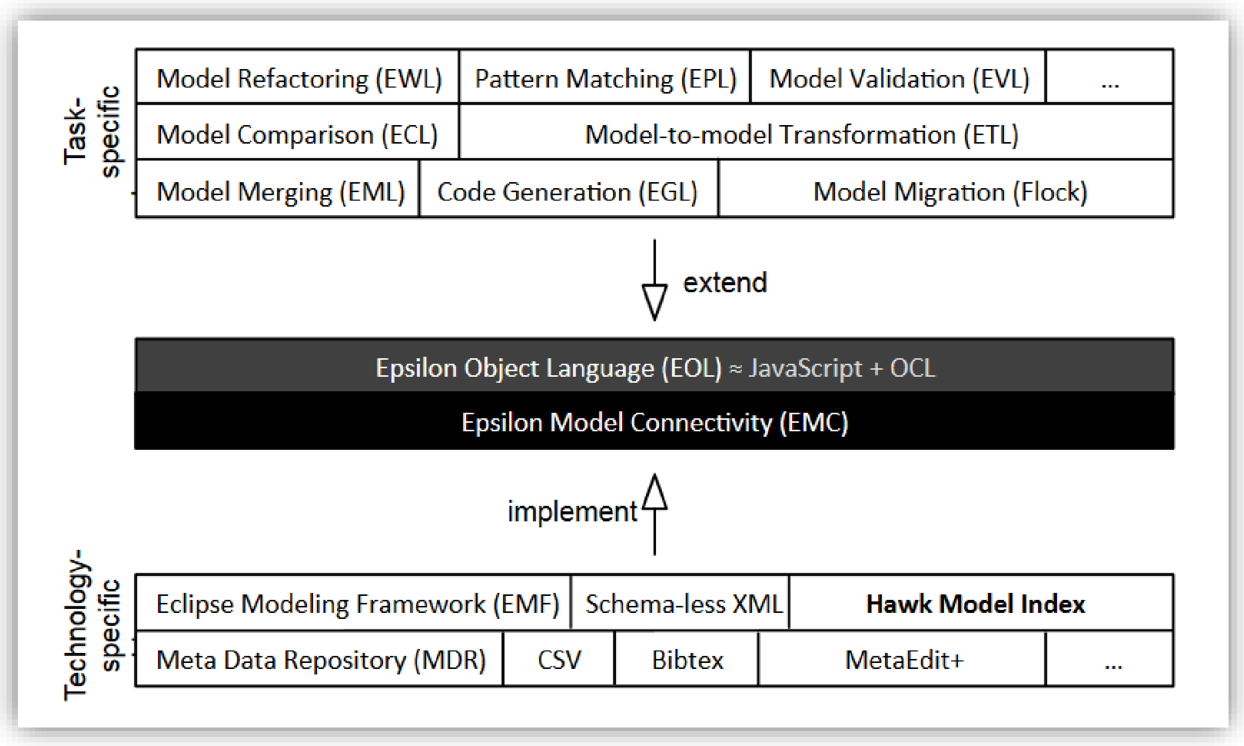


Figure 25. The Epsilone Query Language

As seen in the figure Above, EOL – and as such all languages that build on top of it – is not bound to a particular metamodeling architecture or model persistence technology. Instead, an intermediate layer – the Epsilon Model Connectivity layer – was introduced to allow for seamless integration of any modelling back-end.

This layer of Epsilon uses a driver-based approach where integration with a particular modelling technology is achieved by implementing a driver that conforms to a Java interface IModel provided by EMC. In order to use Epsilon's EOL to query model indexes stored in Hawk, an implementation of the IModel interface is required.

3.2.4. Integration with the MEASURE platform

The Hawk server [11] provides a set of services allowing us to index large collections of file-based models provided by Modelio [12], a complex query language based on the Epsilon Query Language and a remote API which allow to configure, index and query the analyzed models. The remote query API and the advanced capability of the Epsilon Query Language will allow SMM Measures implementers to develop a set of measures by querying the indexed models.

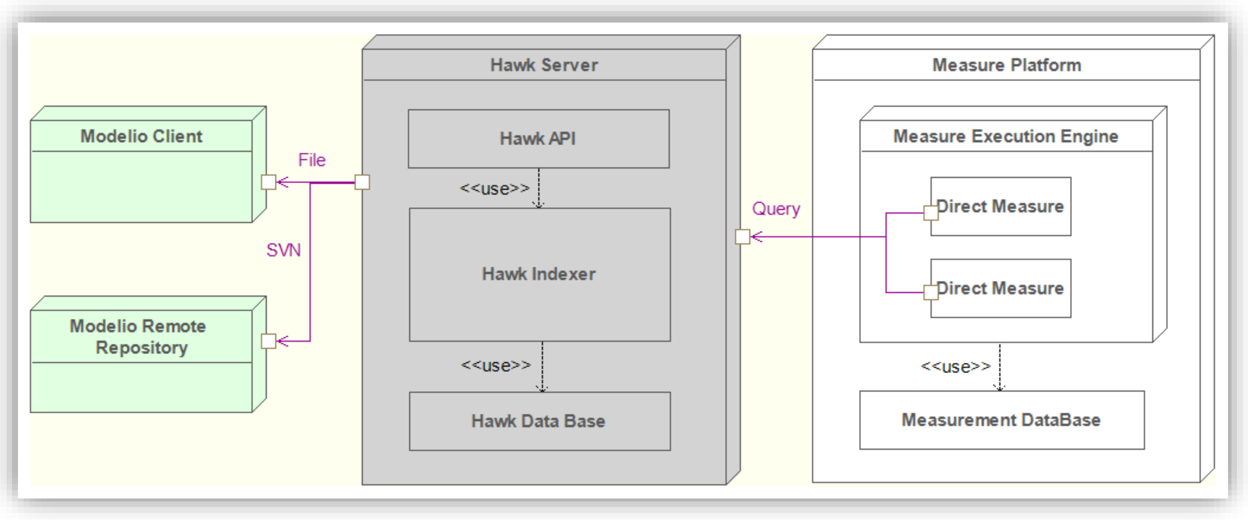


Figure 26. Model Measurement Tool - Measure Platform integration

3.2.5. Development status

As presented, the Hawk server is an existing tool developed in the context of the MONDO European project [9]. To archive specific requirements in relation with Measure project, Softeam worked in collaboration with the University of Aston to integrate it with Modelio [12] and to extend capabilities of the tool and to develop the services needed to use Hawk as a MEASURE external Measurement tool. Today, Hawk is fully integrated to the MEASURE platform.

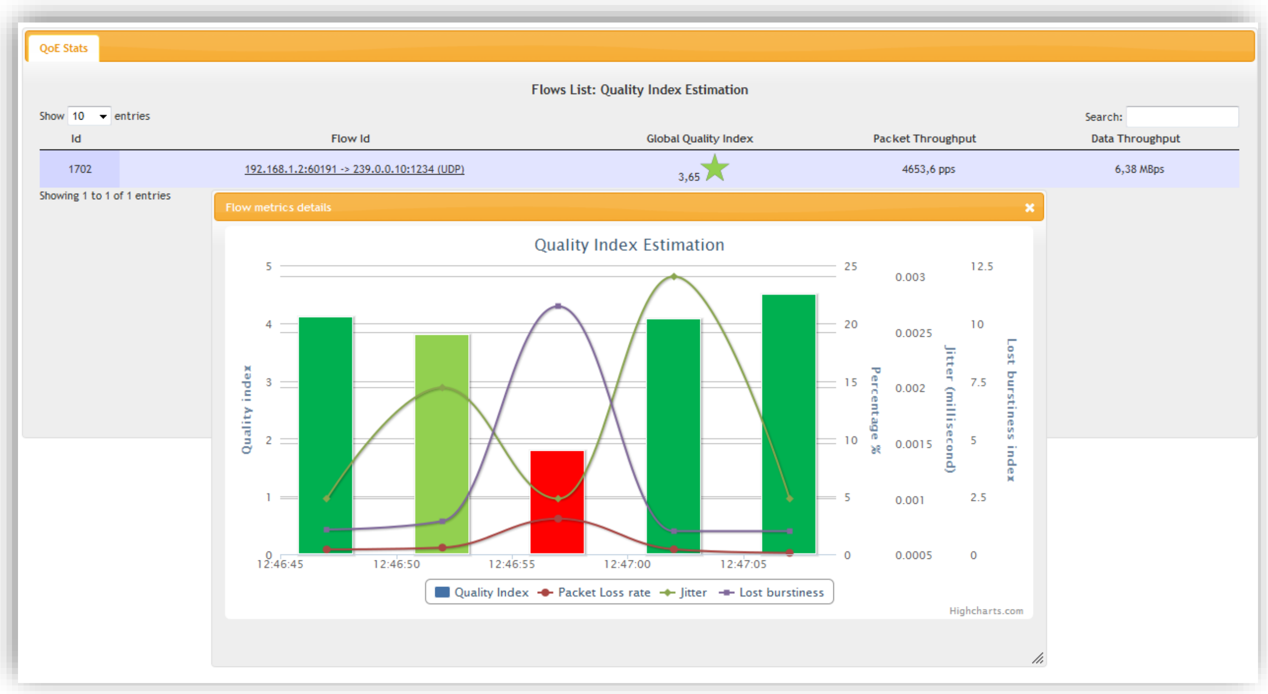
3.3. MMT: Montimage Monitoring Tool

3.3.1. Overview

MMT stands for Montimage Monitoring Tool. It is a network monitoring solution based on Deep Packet Inspection (DPI) technology. MMT combines

- data capture, filtering and storage,
- events extraction and statistics collection, and,
- traffic analysis and reporting providing, network, application and user level visibility.

Through its real-time and historical views, MMT facilitates network performance monitoring and operation troubleshooting. With its advanced rules engine, MMT can correlate network and application events in order to detect performance, operational, and security incidents. An easy-to use customizable graphical user interface makes MMT suitable for different user needs.


QoE Stats

Flows List: Quality Index Estimation

Show 10 entries   Search:

Id	Flow Id	Global Quality Index	Packet Throughput	Data Throughput
1702	192.168.1.2:60191 -> 239.0.0.10:1234 (UDP)	3,65	4653,6 pps	6,38 MBps

Showing 1 to 1 of 1 entries

Flow metrics details

Figure 27. Example of report provided by MMT for video quality analysis

In the context of MEASURE, MMT tool can support several metrics that can be collected during the software operation phase (see table in the conclusion section). Extension of this tool is planned to cover more metrics during the project.

3.3.2. Architecture

MMT is composed of two complementary, yet independent, modules.

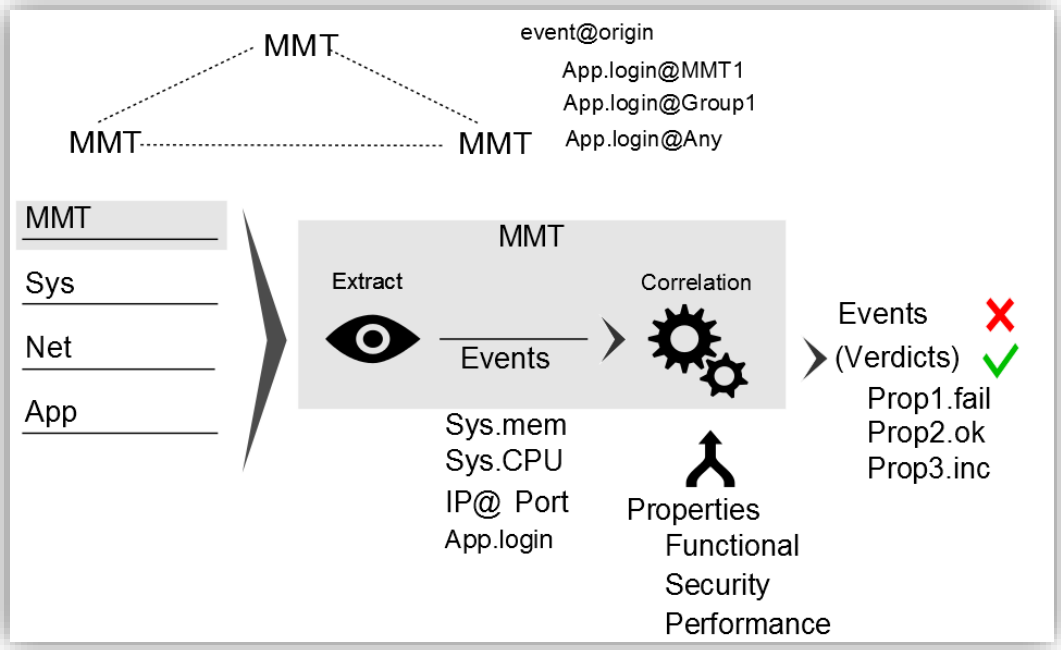


Figure 28. MMT global architecture

- **MMT-Extract** is the core packet processing module, it is a C library that analyses network traffic using Deep Packet and Flow Inspection (DPI/DFI) techniques in order to extract hundreds of network and application based events, measure network and per-application QoS/QoE parameters and KPIs. MMT-Extract is powered with plugin architecture for the addition of new protocols and public API for integration in third party probes.

- **MMT-Correlation** is an advanced rule engine that analyses and correlates network and application events to detect performance, operational and security incidents. It is powered with self-learning capabilities to derive the baseline network and application parameters for dynamic threshold based analysis.

Since the MMT extraction module has plugin architecture, it allows the extension of the MMT tool to parse data coming from system logs or application logs. In this perspective MMT can manage more metrics that can come from different software lifecycle phases.

3.3.3. Interfaces

The current input interfaces of MMT include:

- The network interfaces to capture packets at strategic point of the network to intercept the software under evaluation communication.
- Any pre-captured pcap[5] file containing network traffic of the analysed communicating software.
- System logs and application logs
- A remote MMT probe sharing remote data.

The current output of MMT includes:

- Web based reports that display different collected metrics in the format of timeline, tables and graphs.
- A mongodb database storing all the historical information collected by different probes.

3.3.4. Integration with the MEASURE platform

The MongoDB database content of MMT can be shared with the MEASURE platform. In this case, a dedicated program will be implemented to access this database and send the information to the centralized MEASURE platform.

3.3.5. Development status

MMT tool has been implemented by Montimage and already covers a set of network-based metrics that can be relevant to MEASURE (e.g., bandwidth usage per application or per IP, packet loss rate, TCP retransmission rate, etc.).

MMT has been fully integrated today to the MEASURE platform.

3.3.6. Public link

http://www.montimage.com/products.html

3.4. EMIT: Energy Monitoring Tool

[5] *Pcap is binary format of network captured packets*

3.4.1. Overview

EMIT has been designed and implemented in order to support Green Computing measures and measurements in the MEASURE project. It is composed of a tool set that enables software energy consumption monitoring using web services.

3.4.2. Architecture

The experimental design is depicted below:

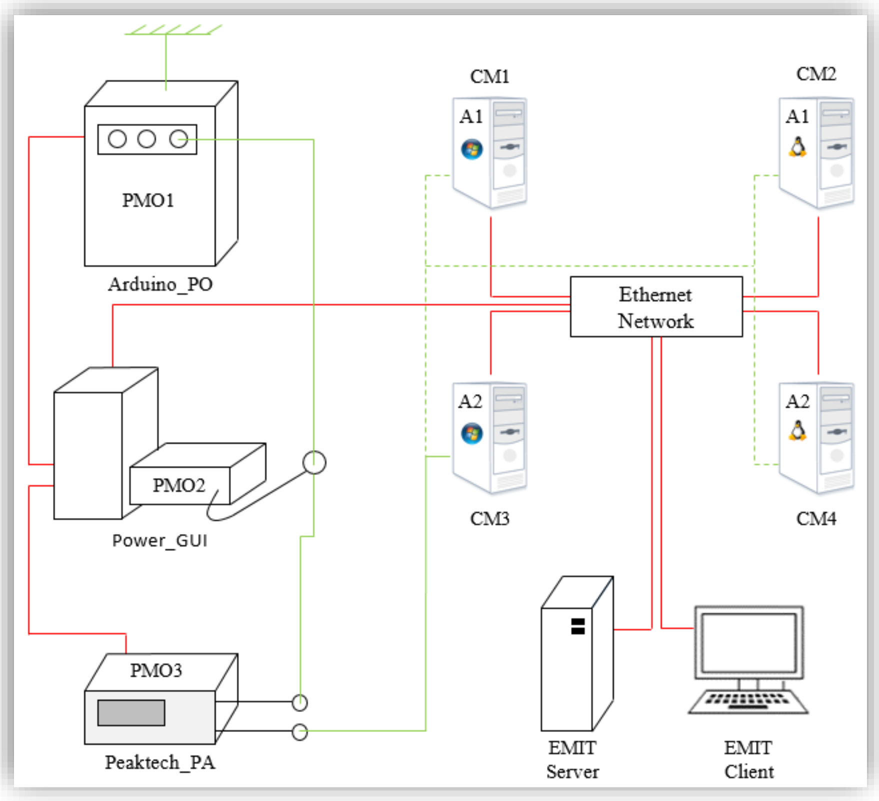


Figure 29. EMIT Experimental design

The elements denoted as CM1/2/3/4 are computers with different operating systems (MS Windows vs Ubuntu Linux) and architecture (32bits vs 64bits). The Arduino_PO, Power_GUI, Peaktech_Pa are different devices that provide all energy consumption measurements. EMIT is composed of several web services that allows users to manage this experimental design by the means of web services.

These web services provide management functionalities over the data model below.

The Measurand entity corresponds to processes or parametrized programs that can be observed by different instruments. Its property called process defines the command line that can be launched.

The Observee entity corresponds to the computer on which measurands (i.e. programs or processes) are installed and can be monitored. Observees are identified by their URI that stands for the path of web services able to launch a process on the computer observers are installed on.

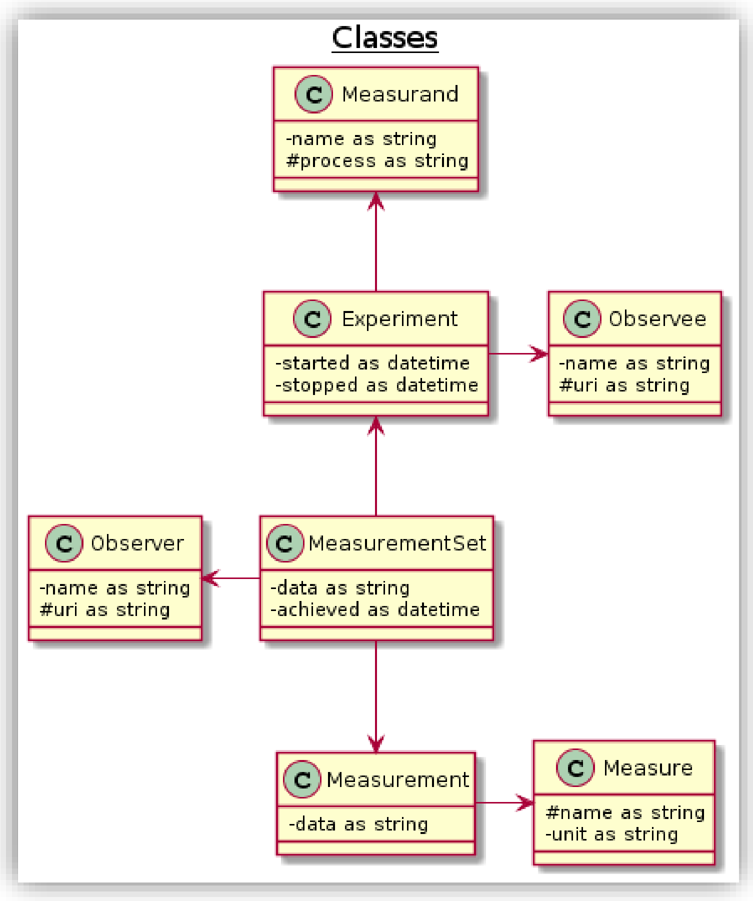


Figure 30. EMIT Class diagram

The Observer entity corresponds to instruments that provide observations or measurements. For instance, Peaktech PA, Arduino PO and Power GUI consist in 3 instruments or observers for energy consumption. As with observees, observers are identified by their URI that stands for the path of web services able to start, stop such instrument activities.

The Experiment entity corresponds to measurement acquisition of a single measurand or process launch on one observe or computer.

The MeasurementSet entity corresponds to data retrieved from one observer or instrument for a given experiment. Such data consists of one or several time series (measurements) for the different measures such instruments can monitor.

The Measurement entity consists of a single time series that is drawn out from its related MeasurementSet.

The Measure entity corresponds to a type of measurements defined by a name and its unit of measure.

EMIT behaves following the activity diagram below.

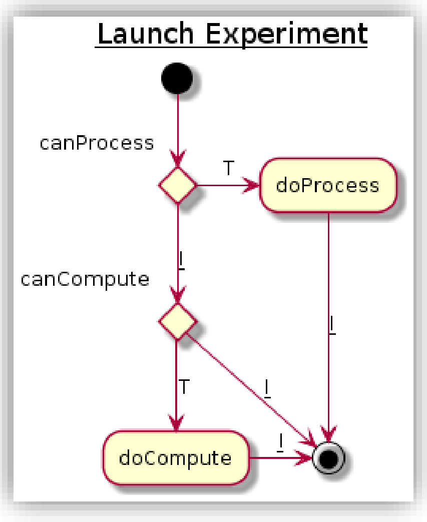


Figure 31. Activity diagram

Every minute, EMIT retrieves from the database experiments that have to be launched i.e. that have neither started nor stopped properties. If such experiments exist, EMIT processes a single measurement acquisition. Otherwise EMIT retrieves from the database measurement sets that have to be split i.e. that have no value for their properties achieved. If such measurement sets exist, EMIT extracts the different time series from their measurement set data file and splits it into several JSON data files that contain a single time series only. Otherwise, EMIT process ends until it is launched again after a 1-minute period.

EMIT's main process is described by the sequence diagram below.

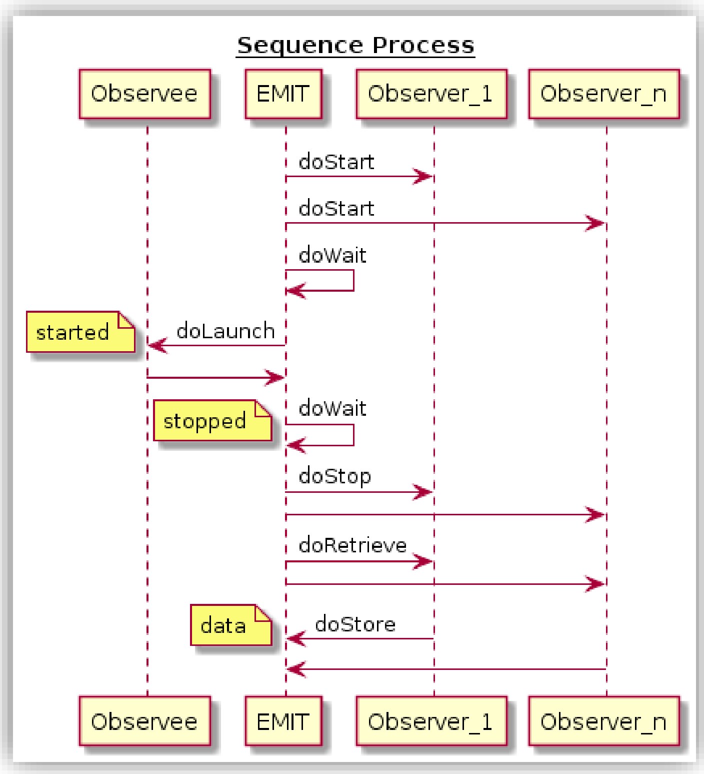


Figure 32. Sequence diagram

EMIT first starts observers. The latter should then record internally their measurements. EMIT gets the current timestamp that corresponds to the experiment property started and then waits for a 5-second period. EMIT launches the measurand process and waits until it is completed. Once done, it waits for 5-seconds period again and gets the current timestamp that corresponds to the experiment property stopped. It retrieves measurement data from every observer, stores them into files on the server filesystem and stores these filenames into the property data of their corresponding measurement sets.

3.4.3. Interfaces

EMIT provides web service-based interfaces. The latter are represented by the use case diagram below.

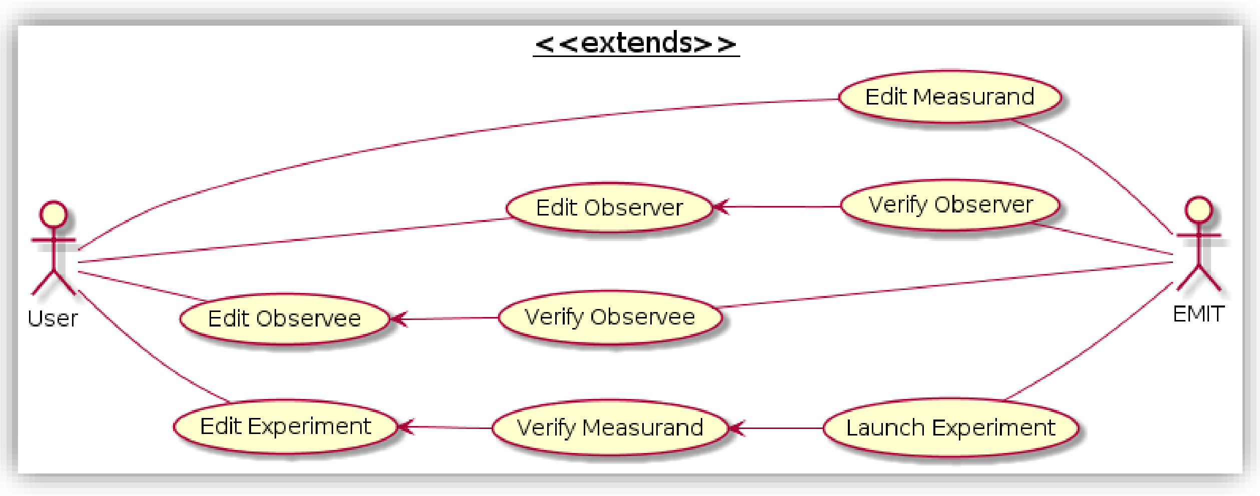


Figure 33. Use Case diagram

EMIT's users can edit several entities drawn out from the data model. It means that users can create, update, delete and retrieve occurrences of such entities. EMIT allows users to register physical devices (observers and observees) by specifying their URI and their name. EMIT also allows users to specify programs to monitor (measurands) by specifying their process and their name. EMIT allows users to configure their experiment in selecting 1 observee and several observers.

EMIT checks that observers and observees exist and comply to the EMIT programming interface. EMIT also checks that experiments are correctly defined i.e. that measurands exist on the target observee and their programs are executable. Moreover, EMIT makes possible to launch experiments as seen previously.

The web services of EMIT measurement platform are listed below.

Measurand web services

- /measurand/create POST (process, name) → id
- /measurand/update POST (id, process, name)
- /measurand/delete POST (id)
- /measurand/retrieve POST () → (Measurand) list

Observer web services

- /observer/create POST (uri, name) → id
- /observer/update POST (id, uri, name)
- /observer/delete POST (id)
- /observer/retrieve POST () → (Observer) list

Observee web services

- /observee/create POST (uri, name) → id
- /observee/update POST (id, uri, name)
- /observee/delete POST (id)
- /observee/retrieve POST () → (Observee) list

Experiment web services

- /experiment/create POST (uri) → id
- /experiment/update POST (id, uri)
- /experiment/delete POST (id)
- /experiment/enable POST (uri)
- /experiment/disable POST (uri)

3.4.4. Integration with the MEASURE platform

Data models created for EMIT have been translated in Structured Metrics Meta-model (SMM) in order to ensure interoperability between all modules of the MEASURE platform.

3.4.5. Development status and plan

The EMIT measurement platform has been designed, implemented and integrated to the MEASURE platform.

3.4.6. Public link

The EMIT project is hosted on GitHub following this link: https://github.com/JeromeRocheteau/emit.

3.5. Test Case Generator

3.5.1. Description:

Test Case Generator of Turkish consortium is a web application that aims to increase the quality of software development and improve efficiency in test process.

The tool has been developed by Java; Ericsson Turkey, Turkcell, Tmob and Turkgen in Turkish Consortium have been working to complete development activities of the tool. This tool will be integrated to Measure Platform to analyze test results of the product, coverage metrics, test succeed and other test and quality assurance kPI's.

3.5.2. **Business added value**

It provides two use-cases:

- Automatic test case generation from requirement document

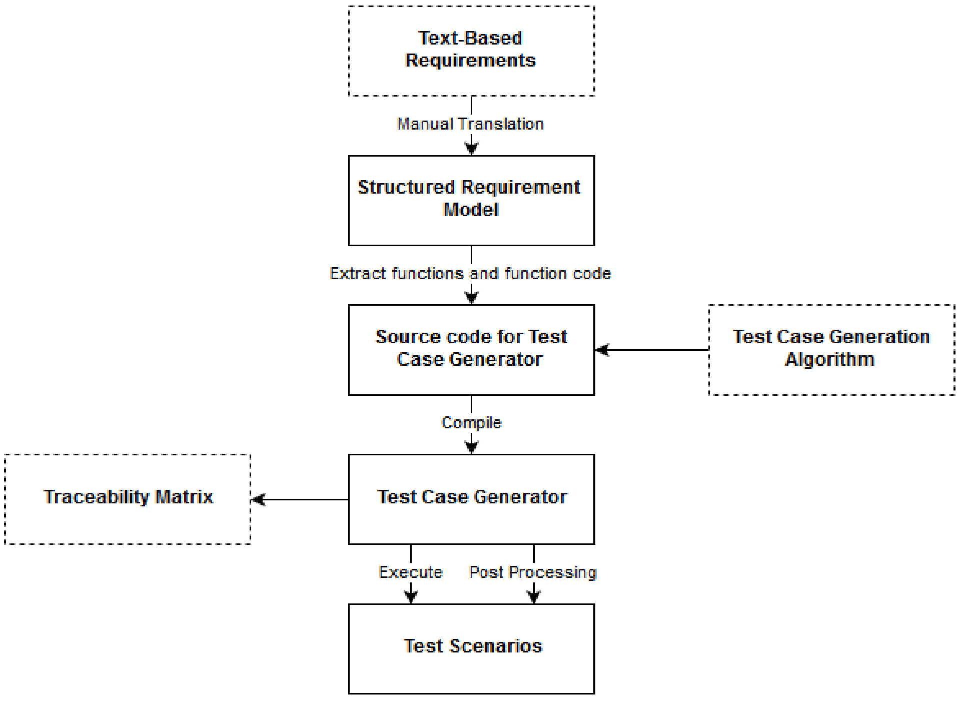

Text-Based Requirements
Manual Translation
Structured Requirement Model
Extract functions and function code
Source code for Test Case Generator
Test Case Generation Algorithm
Compile
Traceability Matrix
Test Case Generator
Execute
Post Processing
Test Scenarios

Figure 34. Turkey Use Case diagram - 1

- Automation framework that executed derived test cases

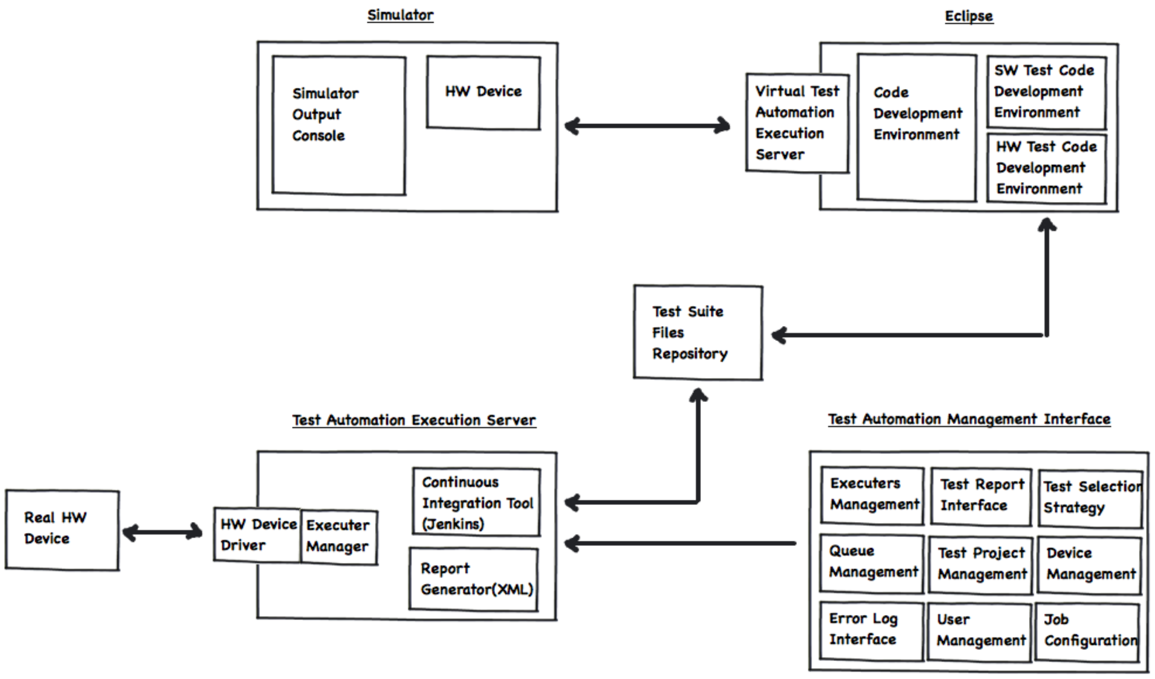


Figure 35. Turkey Use Case diagram - 2

Automatic test case generation will enable customers to derive efficient number of test cases from their requirements, helps them to understand their test coverage and track easily traceability matrix of business requirements.

Automation framework will enable to see status of the product in test results. We will have a clear sight of how many test scenarios are passed or failed.

3.5.3. Integration with the MEASURE platform

This tool will be integrated to Measure Platform to analyze test results of the product, coverage metrics, test succeed and other test and quality assurance kPI's.

3.5.4. Development status and plan

The tool has been designed and implemented. The integration with the MEASURE platform will start in 2019.

4. Integrated Analysis Tools

4.1. Quality Guard

4.1.1. Quality Guard Tool

The Measure Platform allows us to collect measures on various aspects of a development process. In order to monitor these measurements, the Quality Guard Tool allows project managers and quality experts to define quality constraints to compare, in real-time measures, collected by the platform to predefined measure thresholds.

The defined constraints are based on a simple expression language which supports logic operators like “AND”, “OR”, “NOT”, comparison operators like “>”, ”<”, ”<=”, ”<=”, constants and measurements value collected by the platform.

This constraint expression is interpreted by the Constraint Evaluator component whose role is to detect constraints violations. This violation is managed by a Violation Manager component that will apply the strategy defined by the quality gate in this eventuality.

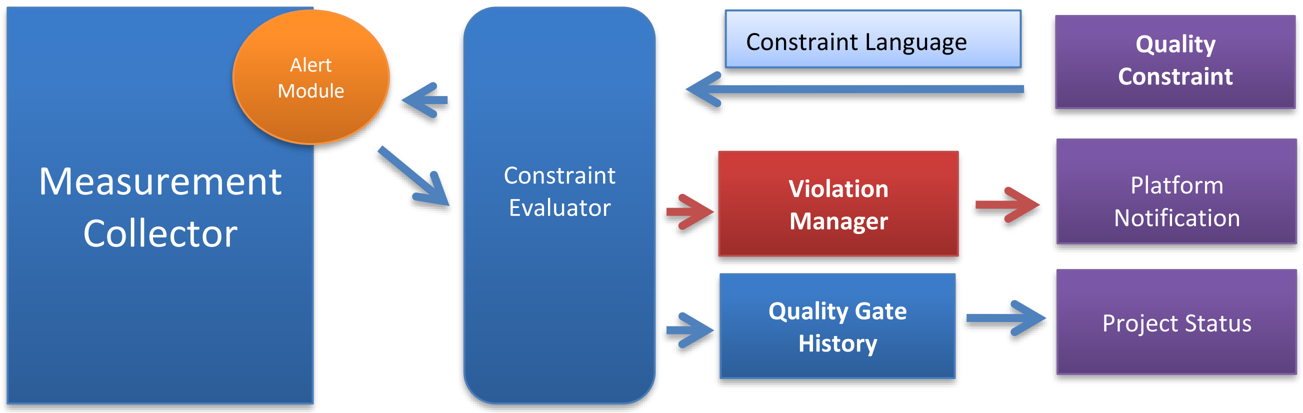


Figure 36 : Constraints violations detection in Measure Platform

The Quality Guard Tool has been released as an open source tool available on GitHub.

- Last tool release can be download at this address: https://github.com/ITEA3-Measure/QualityGuardAnalysis/releases/tag/0.0.4
- A user manual is available online: https://github.com/ITEA3-Measure/QualityGuardAnalysis/wiki
- The source code itself can be downloaded from GitHub: https://github.com/ITEA3-Measure/QualityGuardAnalysis

4.1.2. Business Added Value

The Measure Platform is a data collection and aggregation platform which collects information from the entire product development chain. This data provides a good picture of the state of a software development process but can not reveal their full potential if they are not part of an overall quality approach. A Quality Gate is a process which reviews the quality of all factors involved in production. As part of quality management process, quality controls focused on fulfilling quality requirements.

The Quality Guard approach is a standard way to enforce a quality policy in an organization. The goal of Quality Guard is to answer several questions related to the actual state of a software product:

- What is the actual status of my product for each development phase?

- Can I move to the next development phases of my product?
- Can I deliver a project to production today or not?
- Is there a critical issue that appeared during the past week?

The answer of these key questions can be summarized as Quality Gate, acceptance criteria reviews that can be used throughout any project. It can be seen as a set of predefined quality criteria that a software development project must meet in order to proceed from one stage of its lifecycle to the next.

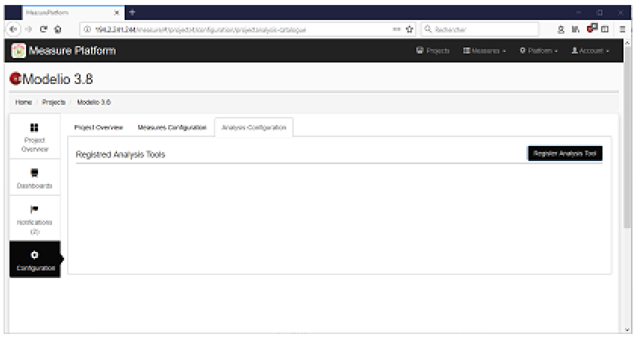
In order to integrate data collected by the platform in a quality process, this data must be constantly compared to threshold values, the quality criteria, identified by quality engineer relying on his expertise and a history of data previously collected. The Quality Guard tool allows then quality engineers to formalize quality criteria as quality rules and, integrated to a notification system which reports all quality violation in a synthetic way, allows the project manager to easily monitor the evolution of the state of his project based on this quality criteria.

4.1.3. Main Analysis Services

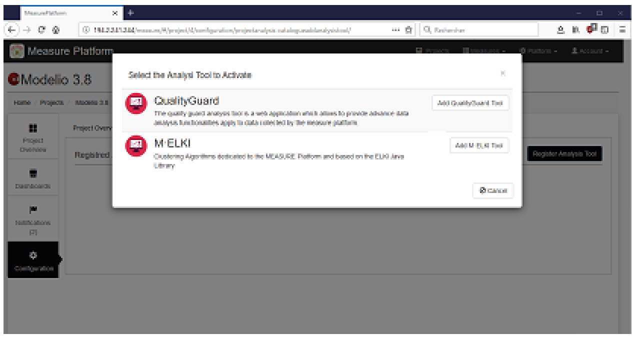
4.1.4. Activate the Quality Guard function in Measure Project.

As for other analysis tools, a Measure project which would like to use Quality Guard services must activate it in Project Configuration view.

- As Project Administrator, go to Configuration view of the project and select the Analysis Tools tab.



- Click on the *Register Analysis Tool* button.
- Select the Quality Guard tool in the list of available analysis tools.

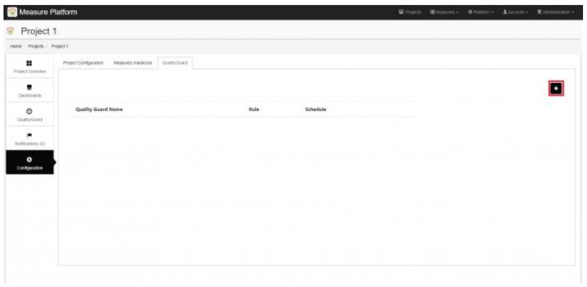

Select the Analysi Tool to Activate

QualityGuard
The quality guard analysis tool is a web application which allows to provide advance data analysis functionalities apply to data collected by the measure platform

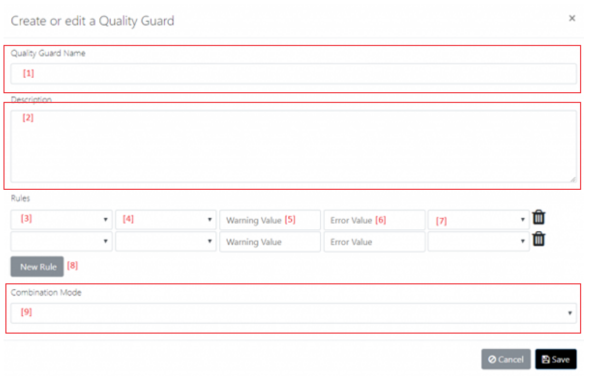
M·ELKI
Clustering Algorithms dedicated to the MEASURE Platform and based on the ELKI Java Library

4.1.4.1 Configure a new Quality Guard Rules

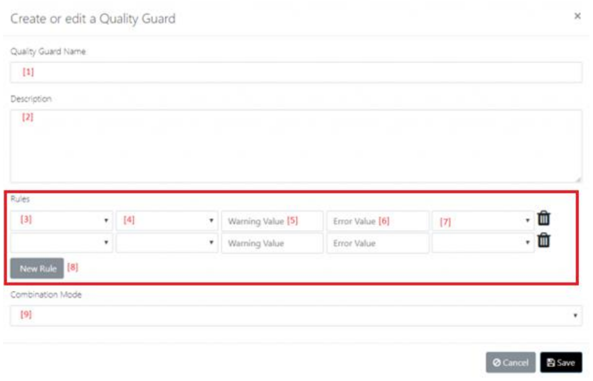
- Create new Quality Guard Rule



- Name and provide a description for the rule. (1,2)
- Chose the aggregation mode: The aggregation model defines how the tool will combine the different conditions. (9)

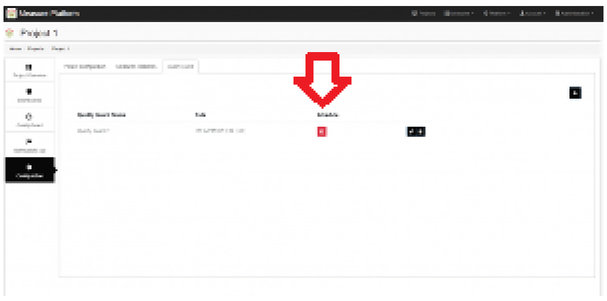


- Define and configure a new Quality Condition
  - Select the monitored measure (3)
  - Select the condition operation (Superior or Inferior) (4)
  - Define the Alert and the Error threshold (5,6)
  - Define the aggregation interval (The measure value is compare to the threshold based on the average value of all collected value during the interval) (7)



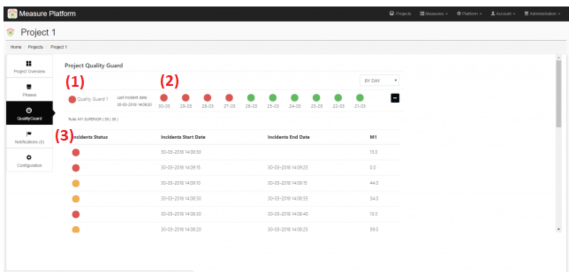
4.1.4.2 Activate and Deactivate a Quality Rule

- Quality Rules can be activated or deactivated independently using the *Scheduling* button



4.1.4.3 Visualise Quality States

- In Quality View, show current state of all quality rules. (1)
- Show the history of quality states. (2)
- Show the list of last quality violations. (3)



4.1.5. Tool Architecture Overview

The Quality Guard Analysis Tool is an independent web application based on the Spring Boot framework. Constraints evaluations, violation management and history analysis are implemented using spring services. A full integration with the Measure Platform is ensured using the dedicated API defined by the platform.

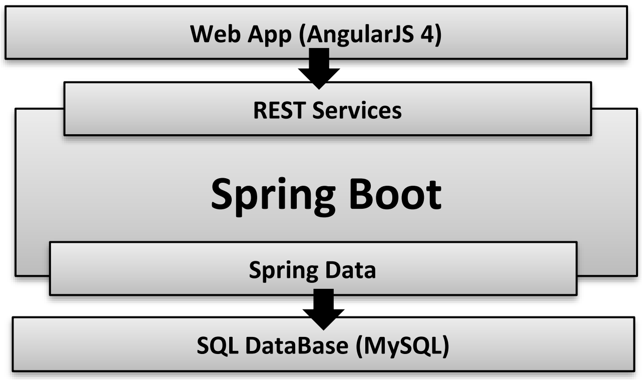


Figure 37 : Quality Guard Tool Technological stack

1.1.1 Technology stack on the client side

- Angular 4 or AngularJS v1.x
- Responsive Web Design with Twitter Bootstrap
- HTML5 Boilerplate

1.1.2 Technology stack on the server side

- Spring Boot for easy application configuration
- Spring Security
- Spring MVC REST + Jackson
- Spring Data JPA + Bean Validation

4.2. MINT

4.2.1. Description

Metrics Intelligence Tool (MINT) is a software solution designed to correlate metrics from different software development life cycle in order to provide valuable recommendations to different actors impacting the software development process. MINT considers the different measurements collected by the MEASURE platform as events occurring at runtime. The correlation is designed as extended finite state machines (EFSMs) allowing to perform Complex Event Processing (CEP) in order to determine the possible actions that can be taken to improve the diverse stages of the software life cycle and thus the global software quality and cost.

This tool is available as web application with NodeJs and uses Montimage's MMT-correlator to implement the extended finite state machines.

The integration to the Measure Platform is made using the provided API to register and configure MINT as an analysis tool. https://github.com/ITEA3-Measure/Mint

4.2.2. Business added value

This tool contributes to improve software quality development identifying and designing correlations between metrics and providing recommendations that help developers to take actions and decisions about the development process. The proposed correlations cover all aspects of the system like functional behavior, security, green computing, and timing. For instance, correlations covering different phases of development and correlations of two metrics from the same development phase at different times.

4.2.3. Background

a) Metrics correlation: The correlation can be defined as a mutual relationship or association between metrics (or the values of its application). Metrics correlation can be the basis for the reuse of metrics; it can help to predict one value from another; it can indicate a causal relation between metrics and can establish relations between different metrics and increase the ability to measure. Examples of correlation are: to correlate two metrics from the same development phase; to correlate the same metric at different times; to correlate a metric (a set of metrics) from phase X regarding metrics of phase Y. As an outcome, recommendations and a selection of metrics will be proposed to the developer to improve the software development. MINT is based on correlation techniques.

b) Complex Events Processing: Complex event processing (CEP) technology addresses exactly the need of matching continuously incoming events against a pattern. Input events from data streams are processed immediately and if an event sequence is matching a pattern, the result is emitted straight away. CEP works very efficiently and in real-time, as there are no overheads for data storing. CEP is used in many areas that include for instance manufacturing processes, ICT security, etc. and is adapted in this work for software quality assessment process.

c) Extended Finite State Machine: In order to formally model the correlation process, the Extended Finite State Machine (EFSM) formalism is used. This formal description allows to represent the correlation between metrics as well as the constraints and computations needed to retrieve a meaningful recommendation related to software quality assessment.

4.2.4. Writing correlation processes

a. *Correlation process inputs and outputs:*

The basic idea behind the MINT approach is to specify a set of correlation rules based on the knowledge of an expert of the software development process. These rules can rely on one or different sets of metrics (seen as inputs) and allow to provide different recommendations (seen as outputs) to different kinds of actors:

- Actors from the DevOps team: Analysts, designers, modelers, architects, developers, testers, operators, security experts, etc.
- Actors from the management plan: product manager, project manager, responsible of human resources, responsible of financial issues etc.

The automatic generation of such rules or their continuous refinement based on some artificial intelligence techniques is an ongoing work.

b. *Example of correlation processes:*

The correlation processes rely on different measurements that are computed and collected by external tools. Some examples of correlations are presented in the Figure 6.

<u>Software Modularity</u>

The assessment of the software modularity relies on two metrics provided by the SonarQube tool that are the class complexity and the maintainability rating. The class complexity measure (also called cognitive complexity) computes the cognitive weight of a Java Architecture. The cognitive weight represents the complexity of a code architecture in terms of maintainability and code understanding. The maintainability rating is the ratio of time (according to the total time to develop the software) needed to update or modify the software.

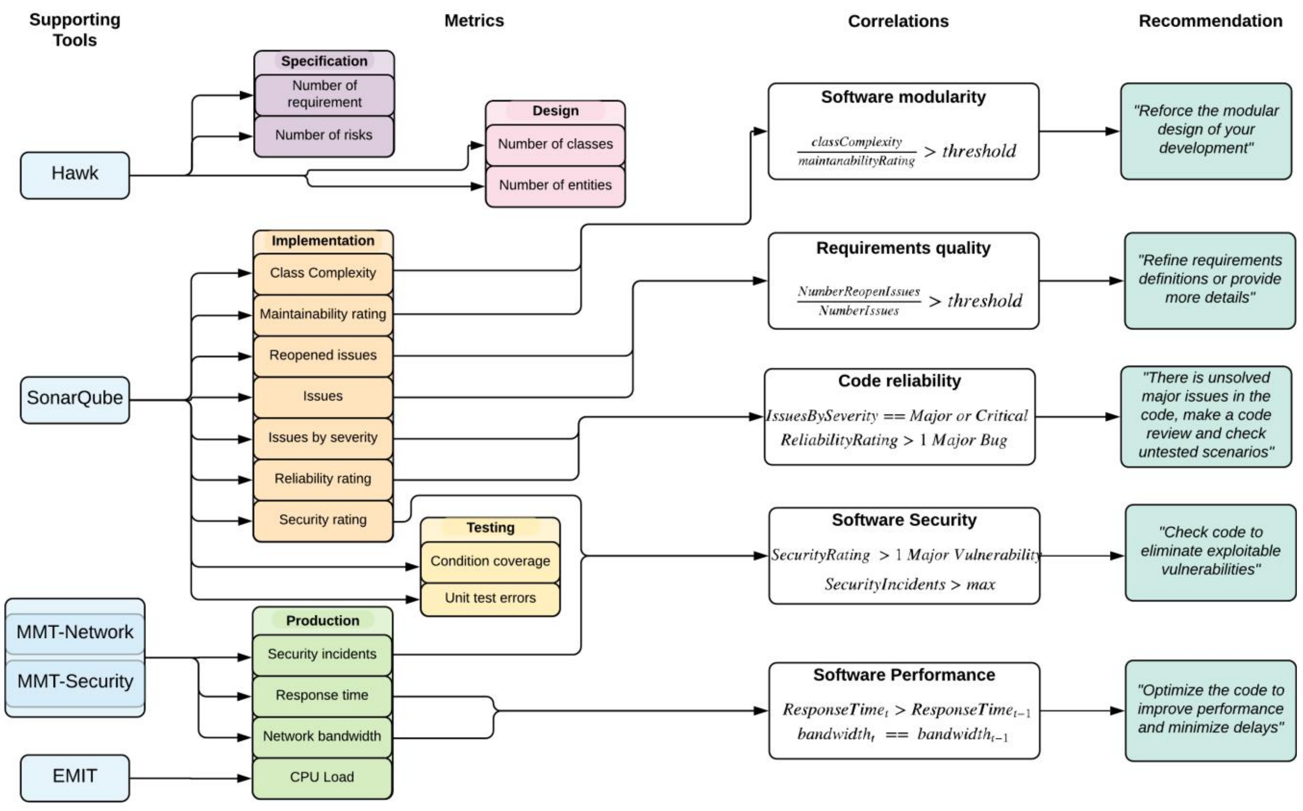

Supporting Tools
Metrics
Correlations
Recommendation
Specification
Number of requirement
Number of risks
Design
Number of classes
Number of entities
Hawk
Implementation
Class Complexity
Maintainability rating
Reopened issues
Issues
Issues by severity
Reliability rating
Security rating
SonarQube
Testing
Condition coverage
Unit test errors
MMT-Network
MMT-Security
Production
Security incidents
Response time
Network bandwidth
CPU Load
EMIT
Software modularity
$\frac{classComplexity}{maintanabilityRating} > threshold$
Requirements quality
$\frac{NumberReopenIssues}{NumberIssues} > threshold$
Code reliability
$IssuesBySeverity == Major\ or\ Critical$
$ReliabilityRating > 1\ Major\ Bug$
Software Security
$SecurityRating > 1\ Major\ Vulnerability$
$SecurityIncidents > max$
Software Performance
$ResponseTime_t > ResponseTime_{t-1}$
$bandwidth_t == bandwidth_{t-1}$
"Reforce the modular design of your development"
"Refine requirements definitions or provide more details"
"There is unsolved major issues in the code, make a code review and check untested scenarios"
"Check code to eliminate exploitable vulnerabilities"
"Optimize the code to improve performance and minimize delays"

Figure 38. Example of Correlation processes.

Based on these definitions and considering that a modular code can be more understandable and maintainable, we can correlate the two metrics and compute the ratio R = class complexity/maintainability rating. If this ratio is more than a specific threshold set by an expert, the recommendation "Reinforce the modular design of your development" will be provided to the software architect and developers.

In the initial state, we can either receive the input related the class complexity denote cc or the maintainability rating denoted mr. The process accesses respectively to the states "cc received" or "mr received". If we receive the same measurement related to the same metric, we update its value and loop on the state. Otherwise, if we receive the complementary metric, we compute the ratio R = class complexity/maintainability rating. If this ratio is less than the defined threshold, we come back to the initial state otherwise, we raise the recommendation.

Timers are used to come back to the initial state if the measurements are too old. For sake of place, only this EFSM is presented in Fig. 7. All the others follow the same principles.

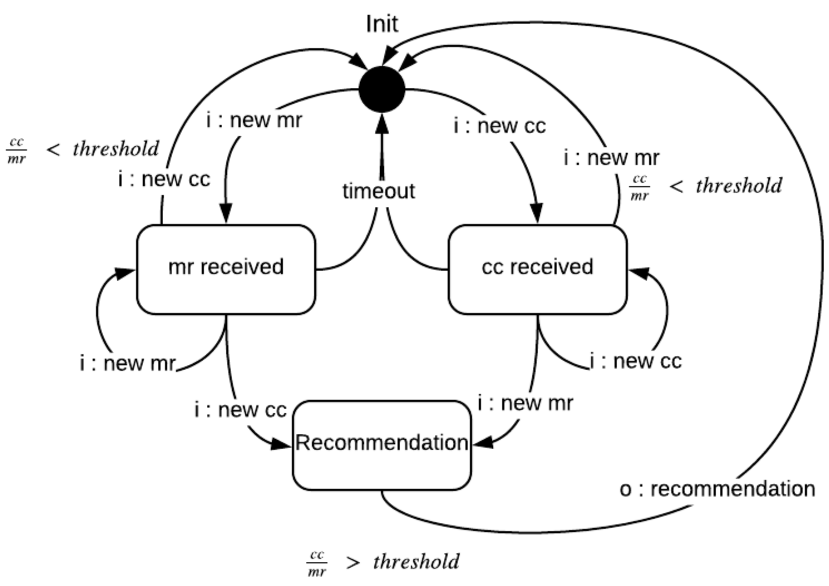


Figure 39. Software Modularity Correlation processes.

Requirements quality

The assessment of the requirements quality can rely on two metrics provided by the SonarQube tool that are the total number of issues and the total number of reopened issues. These numbers are collected during the implementation phase and we can consider that the fact that we reopen an issue many times during the development process can be related to an ambiguous definition of the requirement that needs to be implemented. If we have a ratio R = number of reopened issues/number of issues that is more than a specific threshold, we can consider that the requirements are not well defined and that the development needs more refinement about them. The recommendation "Refine requirement definitions or provide more details" will be provided to the requirements analyst.

Code reliability

The assessment of the code reliability relies on two metrics provided by the SonarQube tool that are the number of issues categorized by severity and the reliability rating. The issues in SonarQube are presented with severity being blocker, critical, major, minor or info and the reliability rating are from A to E: A is to say that the software is 100% reliable and E is to say that there is at least a blocker bug that needs to be fixed. Based on these definitions and considering that a reliable code should be at last free of major or critical issues, we can check that there is no major, critical nor blocker issues and the reliability rating is < C corresponding to 1 Major bug. If this condition is not satisfied, the recommendation "There is unsolved major issues in the code, make a code review and check untested scenarios" will be provided to the software developers and testers.

Software security

The assessment of the software security relies on two metrics, one provided by the SonarQube tool that is the security rating and the other is provided by MMT that is the number of security incidents. The security rating in SonarQube provide an insight of the detected vulnerabilities in the code and are presented with severity being blocker, critical, major, minor or no vulnerability. The number of the security incidents provided by MMT reports on successful attacks during operation. The evaluation of security demonstrates that if an attack is successful this means that the vulnerability in the code was at least major because an attacker was able to exploit it to perform its malicious activity. Based on these definitions and considering that a reliable code should be at last free of major vulnerabilities, we can check if there is a major vulnerability and that the number of attacks at runtime are more than a threshold. If this condition is satisfied, the recommendation

"Check code to eliminate exploitable vulnerabilities" will be provided to the software developers and security experts.

Software Performance

The assessment of the software performance relies on two metrics provided by the MMT tool that are the response time and the bandwidth usage. The response time denotes the delay that can be caused by the software, hardware or networking part that is computed during operation. This delay is in general the same for a constant bandwidth (an equivalent number of users and concurrent sessions). Based on this finding, we can correlate the two metrics and compute that the response time is not increasing for during time for the same bandwidth usage. If this response time is increasing, the recommendation "Optimize the code to improve performance and minimize delays" will be provided.

4.3. Metrics Suggester

4.3.1. Description

The Metrics Suggester provides a framework to automate the suggestion of software metrics based on an initial measurement plan. To do so, the framework needs an initial configuration from the user to determine the metrics range to be analysed and the classifier. This framework is available as web application written in Python and using the Python Scikit-learn library for the analysis process.

This tool is based on learning techniques, The SVM (Support Vector Machine) algorithm for the classification or in other words the analysis of the measurements and the RFE (Recursive Feature Elimination) one for the suggestion of a new measurement plan.

This application is integrated to the Measure Platform as the analysis tool: SuggesterTool, by using the MEASURE platform analysis tool integration protocol.

4.3.2. Business added value

Metrics Suggester is an interesting tool allowing to determine (based on the analysis of collected data) and visualise relevant measures, metrics and measurement plans during a specific period of time. This tool is first expected to be used during courses at Telecom SudParis but also to be enriched by other probes, learning techniques (unsupervised) and applied through other diverse metrics (e.g., emotional). IMT also aims to make it sustainable for measuring concrete activities at Telecom SudParis.

4.4. M·ELKI

4.4.1. Description

M·ELKI is a set of web services that makes possible to select, configure, process and visualize results of 4 clustering algorithms provided from the ELKI Java library (https://elki-project.github.io). This library is a state-of-the-art and clustering-focused library in Java. It has been chosen because of these 3 features. It has been chosen instead of the Weka library (https://www.cs.waikato.ac.nz/ml/weka) because its integration within web services has been proved easier than Weka, its memory footprint and its processing efficiency better than Weka[6]. The 4 selected algorithms are drawn out from 4 different clustering algorithm families:

6 Hans-Peter Kriegel, Erich Schubert, Arthur Zimek: The (black) art of runtime evaluation: Are we comparing algorithms or implementations? In Knowledge and Information Systems 2016, DOI: 10.1007/s10115-016-1004-2 (https://elki-project.github.io/benchmarking)

1. DBSCAN, a density-based clustering algorithm
2. KMEANS, a centroid-based clustering algorithm
3. EM, a distribution-based clustering algorithm
4. SLINK, a connectivity-based clustering algorithm

This set of web services are dedicated to the MEASURE platform. In fact, these web services are designed to closely fit the MEASURE analysis tool integration protocol (MeasurePlatform/Analysis-Tool-Integration). Hence the name: M·ELKI.

4.4.2. Business added value

M·ELKI provides state-of-the-art and efficient clustering algorithms to the MEASURE platform. In fact, these are basic algorithms for datamining and bigdata processing. This analysis tool can easily be extended to other algorithms that belong to the ELKI library that same way as the 4 selected algorithms.

4.5. STRACKER

4.5.1. Description

STRACKER is a web application that aims to increase the quality of software development by tracking and suggesting (thus, the acronym STRACKER) values of various software metrics during the software development process. More precisely, it helps you see the status of the metric values using different charts, and also shows scores assigned to each new record. It also includes a module that predicts future metric values based on values recorded so far.

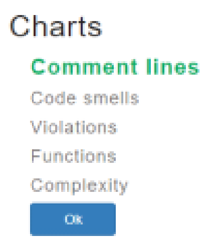
In the future, new metrics, an increase in prediction accuracy and new algorithms for metrics correlation are planned.

4.5.2. How to use the tool

4.5.3. Track software metrics values

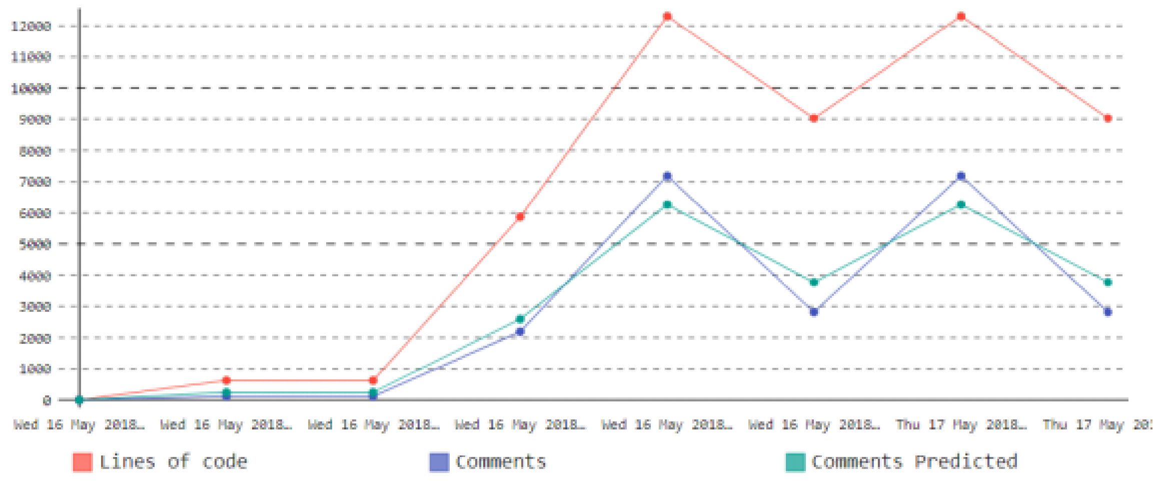
Select a metric that you want to see.

Let us say that we want to see the comment lines metric. We select the comment lines option (option that matches the desired metric) and click ok button.



Then, you can see two charts on the page (the charts are created using pygal library).

First chart may look like the one below:

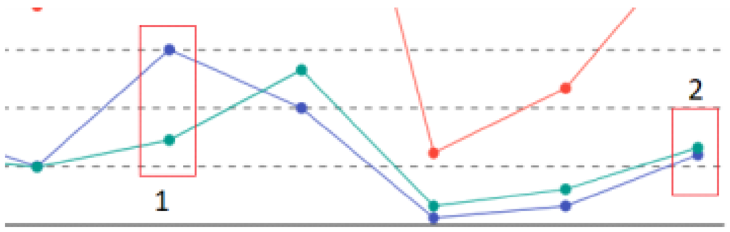


The red line shows the number of lines of code of the project.

The blue line shows the number of comment lines of the project.

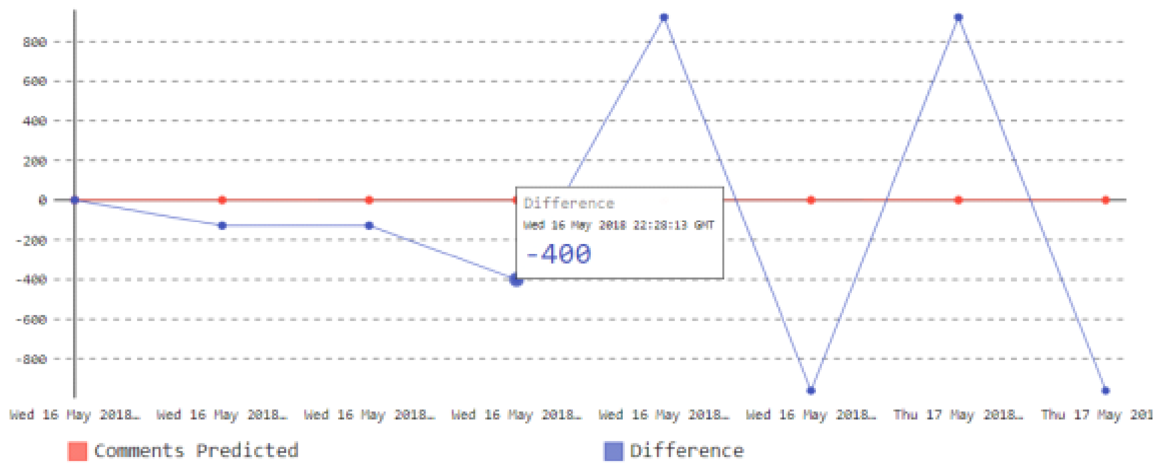
The green line shows the predicted number of comment lines of the project.

The scope of the chart is to show to a software project manager the difference between the current metric (blue line) and the predicted metric (the green line). Since the prediction is based on a model taking into account many software projects, the predicted value is, in a sense, showing the „best practices".



Zooming in (see figure above), we notice that the red box 1 shows a situation when the difference is rather big whereas in red box 2, the metric values and predicted values are almost similar.

The second chart uses the predicted metric as a baseline:

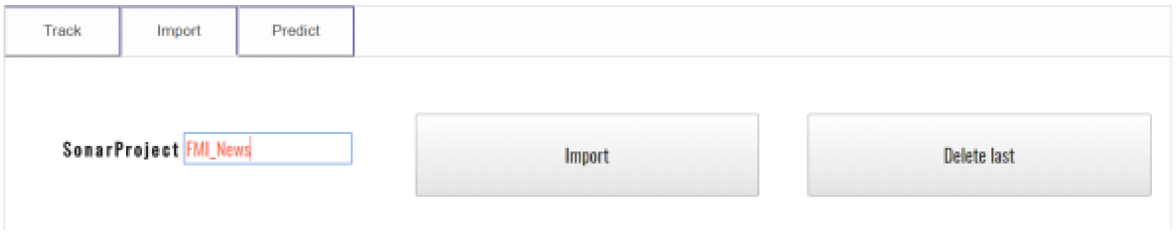


Here we can more clearly see the difference between the number of predicted comment lines and the number of comment lines of the project.

Also, if we put the cursor over a point, we can see the exact value of that point.

4.5.4. Import new metrics values

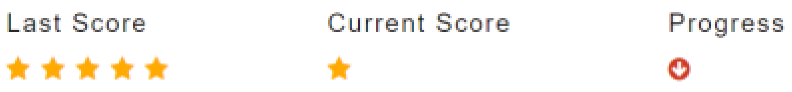
In order to add a new project to the current project, you need to upload your project in SonarQube and then Stracker will make the import from there.



Go to "Import tag". Write the project name from SonarQube and click Import button. Now your new record is added.

Also, here you can delete your last record by clicking the Delete last button.

Also, you can see if your new record is better or not than the last one, by looking at the score.



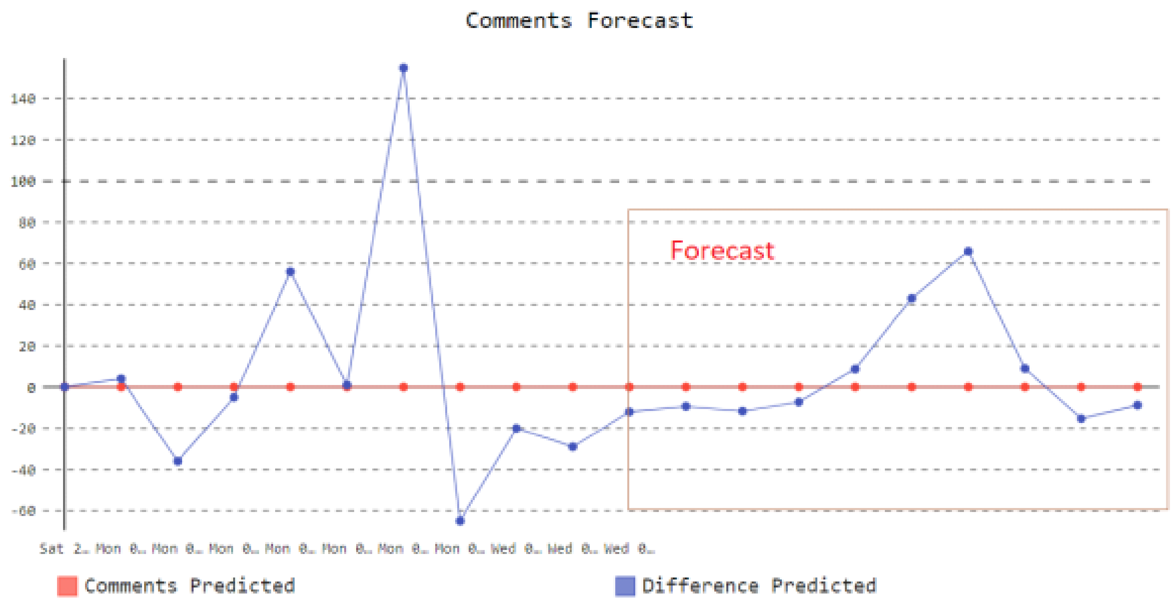
Current score represents the score of the last added value, calculated based on the difference between our metric value and predicted value.

Last score is the previously added score.

Progress is positive if the current score is better than the previous one, and negative otherwise.

3. Forecast software metric values

We can also see a graph with forecasted values of our project metrics data.



Thus, you can get the possible values of your metric in the future (using time series prediction algorithms).

4.5.5. Business added value

There are two important features of the Stracker tool that may be valuable to a software project manager.

First, she can compare various values of the metrics with “expected” values from a “best practices” point of view. Large deviations may raise some alarms (we will implement this feature in the future) and check the situations where the difference is large.

Second, if she receives predicted values for her metrics, she can better plan the project (e.g., if the predicted number of tests is high you may increase the man-power from testing point of view).

5. References

[1] https://github.com/ITEA3-Measure/

[2] Rainer Gerhards: “The Syslog protocol”, Request for Comments (RFC) 5424, DOI: 10.17487/rfc5424, 2015.

[3] https://www.iso.org/standard/35733.html

[4] K. P. Bennett and A. Demiriz, “Semi-Supervised Support Vector Machines,” Advances in Neural Information Processing Systems, pp. 368– 374, 1999.

[5] S. Dahab, S. Maag, A. Bagnato, and M. A. A. da Silva, “A learning based approach for green software measurements,” in Proc. of the 3rd International Workshop on Measurement and Metrics for Green and Sustainable Software Systems, MeGSuS 2016, co-located with the 10th International Symposium on ESEM 2016, 2016, pp. 13–22

[6] S. Dahab, S. Maag, X. Che,“A Software Measurement Framework guided by Support Vector Machines.” The 5th International Workshop on Network Management and Monitoring, NetMM 2017, Taiwan

[7] Hawk: towards a scalable model indexing architecture Konstantinos Barmpis, Dimitris Kolovos University of York, Heslington, York, UK Proceeding BigMDE '13 Proceedings of the Workshop on Scalability in Model Driven Engineering Article No. 6

[8] Integration of a graph-based model indexer in commercial modelling tools - Antonio García-Domínguez, Konstantinos Barmpis, Dimitrios S. Kolovos, Marcos Aurélio Almeida da Silva, Antonin Abherve, Alessandra Bagnato MoDELS2016 DOI: 10.1145/2976767.2976809

[9] MONDO Project Web Site *http://www.mondo-project.org/*

[10] MONDO Deliverable 5.5 D5.5 Model Indexing Framework Final Version *Model Indexing Framework*

[11] Hawk Model Indexer in GitHub, *https://github.com/mondo-project/mondo-hawk*

[12] Modelio Modeling Tool, www.modelio.org

[13] T. Stoenescu, A. Stefanescu, S. Predut, F. Ipate. RIVER: A Binary Analysis Framework using Symbolic Execution and Reversible x86 Instructions. In Proc. of 21st International Symposium on Formal Methods (FM'16), LNCS 9995, pp. 779-785, Springer, 2016.

[14] T. Stoenescu, A. Stefanescu, S. Predut, F. Ipate. Binary Analysis based on Symbolic Execution and Reversible x86 Instructions, published in Fundamenta Informaticae 153, 2017.

[15] C. Paduraru, M. Melemciuc, A. Stefanescu. A distributed implementation using Apache Spark of a genetic algorithm applied to test data generation. published in Proc. of PDEIM'17, workshop of GECCO’17, ACM 2017.