COOPERATION

OSSMETER

Automated Measurement and Analysis of Open Source Software

Project Number 318772

D5.1 – Platform Architecture Specification

Version 1.0
15 April 2013
Final

Public Distribution

University of York

Project Partners: Centrum Wiskunde & Informatica, SOFTEAM, Tecnalia Research and Innovation, The Open Group, University of L'Aquila, UNINOVA, University of Manchester, University of York, Unparallel Innovation

Project Partner Contact Information

Centrum Wiskunde & Informatica Paul Klint Science Park 123 1098 XG Amsterdam, Netherlands Tel: +31 20 592 4126 E-mail: paul.klint@cwi.nl	**SOFTEAM** Alessandra Bagnato Avenue Victor Hugo 21 75016 Paris, France Tel: +33 1 30 12 16 60 E-mail: alessandra.bagnato@softeam.fr
Tecnalia Research and Innovation Jason Mansell Parque Tecnologico de Bizkaia 202 48170 Zamudio, Spain Tel: +34 946 440 400 E-mail: jason.mansell@tecnalia.com	**The Open Group** Scott Hansen Avenue du Parc de Woluwe 56 1160 Brussels, Belgium Tel: +32 2 675 1136 E-mail: s.hansen@opengroup.org
University of L′Aquila Davide Di Ruscio Piazza Vincenzo Rivera 1 67100 L'Aquila, Italy Tel: +39 0862 433735 E-mail: davide.diruscio@univaq.it	**UNINOVA** Pedro Maló Campus da FCT/UNL, Monte de Caparica 2829-516 Caparica, Portugal Tel: +351 212 947883 E-mail: pmm@uninova.pt
University of Manchester Sophia Ananiadou Oxford Road Manchester M13 9PL, United Kingdom Tel: +44 161 3063098 E-mail: sophia.ananiadou@manchester.ac.uk	**University of York** Dimitris Kolovos Deramore Lane York YO10 5GH, United Kingdom Tel: +44 1904 325167 E-mail: dimitris.kolovos@york.ac.uk
Unparallel Innovation Nuno Santana Rua das Lendas Algarvias, Lote 123 8500-794 Portimão, Portugal Tel: +351 282 485052 E-mail: nuno.santana@unparallel.pt	

Contents

Document Control

Version	Status	Date
0.1	Document outline	1 February 2013
0.2	First draft	14 February 2013
0.7	First full draft	1 March 2013
0.8	Further editing draft	15 March 2013
1.0	QA review	15 April 2013

Executive Summary

This document provides a high level overview and different architectural views the comprise the evolving technical architecture of the OSSMETER platform. It describes the goals of the architecture, the use cases supported by the platform and architectural components that will be implemented to best achieve the use cases. The deliverable outlines the technologies that will be used for the implementation of the platform as well as various architectural constraints that have to be taken into consideration during the implementation phase. A brief discussion on the quality attributes of the proposed architecture is also provided.

1 Introduction

This document provides a high level overview of the evolving technical architecture of the OSSMETER platform. It describes the goals of the architecture, the use cases supported by the platform and architectural components that will be implemented to best achieve the use cases. Moreover, it outlines the technologies that will be used for the implementation of the platform as well as various architectural constraints that have to be taken into consideration during the implementation phase. The development of the architectural description provided in this document was guided by the *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems* [4].

The work presented in this document is done in the context of workpackage (WP) 5, whose objective is to develop the OSSMETER system and integrate the components developed in workpackages 2-4. The results presented in the following sections are related to the following tasks (from the OSSMETER DoW):

Task 2.1: Platform architecture specification. Design an architecture that will host the components developed in work packages 2-4, schedule their execution and store the calculated metrics for the registered OSS project in a scalable manner.

Task 2.2: Project metadata repository design and implementation. Design and implement a scalable repository that will store the metrics calculated by OSS analysis and measurement components developed in work packages 2-4.

1.1 Purpose and Scope of the System

The purpose of the OSSMETER platform is to integrate user-defined quality measurement tools and techniques. The main responsibility of the platform will be the organisation and execution of project metric providers, and the storage of the required data and metric results in a scalable manner. In the context of OSSMETER project, the term *metric provider* can have two different meanings. First, a metric provider can be a software component which extends the OSSMETER platform with additional software metrics and measuring capabilities. Second, a metric provider can be the individual, who extends the platform by contributing to it a new software metric component to the platform.

In more detail the OSSMETER open source extensible platform will:

- Interface with the most widely-used open source hosting forges (e.g. SourceForge, Google Code, Eclipse, Apache, Github);
- Provide extensibility mechanisms for interfacing with additional version control systems, bug tracking systems and communication channels;
- Support incremental monitoring of various aspects of OSS projects;
- Provide support for exploring and comparing the obtained measurements for OSS projects in an intuitive way;
- Produce notifications when health indicators of an OSS project fall below a user-defined level;
- Provide support for exploring the history and evolution of OSS projects;
- Provide support for metric providers to define and register new metrics;
- Provide support for scalable persistence of OSS projects' data.

The OSSMETER platform will be available both as a free hosted service, and as OSS software that users will be able to deploy on their own infrastructure in order to monitor selected OSS and proprietary (internal) software projects.

On top of the OSSMETER platform, a public REST (Representational State Transfer) APIAPI will be provided. The purpose of this API is to allow external developers to retrieve calculated metrics for OSS projects of interest and to exploit them in custom applications. Moreover, an intuitive web-based user interface will be provided through which end users will be able to register new OSS projects to the system, browse through the calculated metrics of projects of interest and compare selected projects side-by-side according to their preferred criteria. The description of the two aforementioned components is out of the scope of this document. This document focuses solely upon the architectural description of the OSSMETER platform, which is the core of the proposed system. Future documents will describe in detail any additional components.

1.2 Stakeholders

One of the main purposes of this architectural description is to serve as a communication vehicle among system stakeholders. The main stakeholders of the OSSMETER platform are the following:

- **Architect**: An individual or organization responsible for system's architecture [4].
- **Developer**: An individual or organization that performs development activities (including requirements analysis, design, testing through acceptance) during the software life cycle process [5].
- **User**: An individual or organization that uses the operational system to perform a specific function [5]. In the context of OSSMETER, a user is an individual or organization, that is using the platform to monitor a specific project or to compare different projects.
- **Augmenter**: An individual or organization that extends or enhances the system. In the context of OSSMETER, an augmenter is an individual or organization, that extends the functionality of the platform by contributing new metric providers or repository connectors.
- **Manager**: An individual or organization that is responsible for the project management.
- **Reviewer**: An individual or organization responsible, for monitoring the progress of the project.

1.3 Overview

The remainder of this document is organised as follows. Section 2 presents the architecture of the system. In particular, in Section 2.1 the different architectural views, which are used in this document are briefly presented. Section 2.2 discusses the architectural view decomposition in more detail. Sections 2.2.1 - 2.2.6 comprise the core of the document. These sections provide a detailed description of the different views used to capture system's architecture. Section 2.2.7 provides a brief discussion on the quality attributes of the proposed architecture and the final section, Section 3, concludes this document.

2 Platform Architecture

In this section, the evolving architecture of the OSSMETER platform will be presented. To develop the OSSMETER platform a typical iterative software engineering process will be used. The process will consist of multiple iterations of an analysis, a design, an implementation, a testing and a deployment phase. A graphical overview of this process is illustrated in Figure 1. The methodology used is based on the concept of evolutionary prototyping as described in [7].

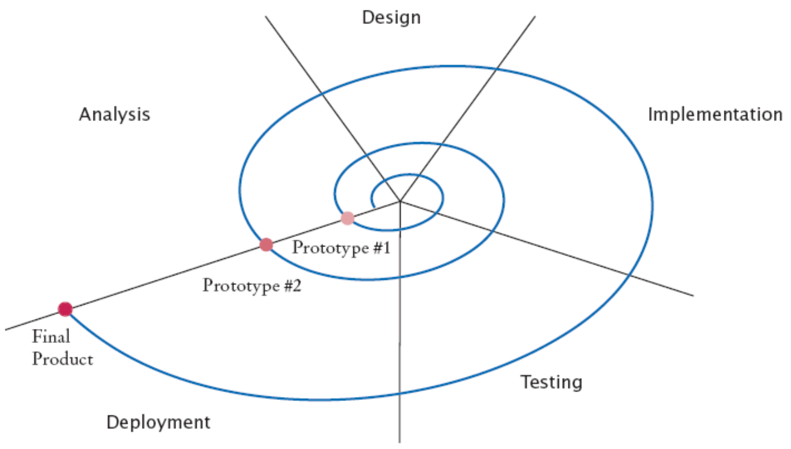


Figure 1: OSSMETER platform development process.

2.1 System Views

This document presents an initial architecture of the system in response to current and expected user needs. The initial prototype of the OSSMETER platform will be delivered using this architecture. In future iterations, the architecture will be evaluated and revised in response to changing requirements and new stakeholder needs.

To describe the architecture of the proposed system, we will use a modified version of the *4+1* view model, which is a model for "describing the architecture of software-intensive systems, based on the use of multiple, concurrent views" [6]. The modification to this model is the addition of a sixth view, namely the *Data View* (Figure 2). Since the OSSMETER platform is data-driven, we decided that it is appropriate to include this extra view. The six views used are the following:

- **Use case view**: A small set of use cases is used to illustrate the architecture of the system. The use cases describe interactions between system stakeholders and the system. They serve as a starting point for tests of an architecture prototype.
- **Data View**: The data view is concerned with the type and the organisation of the data used by the system.
- **Logical view**: The logical view is concerned with the functionality that the system provides to system users.
- **Development view**: The development view describes the static organization of the software.

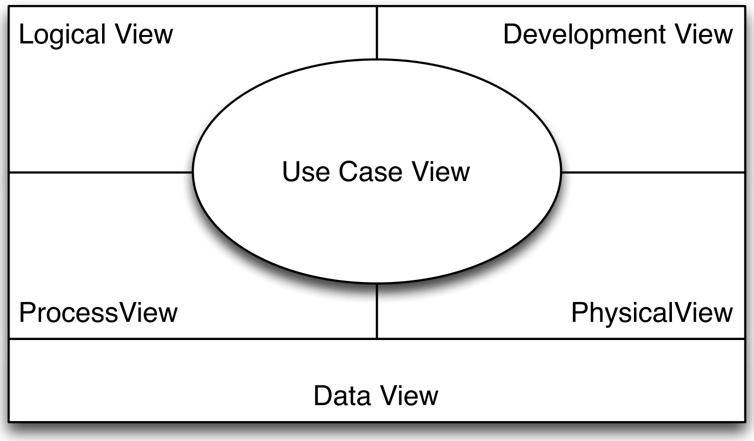


Figure 2: Illustration of the Architectural View Model used in the Context of OSSMETER

- **Process view**: The process view deals with the dynamic aspects of the system and explains the system processes.
- **Physical view**: The physical view is concerned with the topology of software components on the physical layer, as well as the physical connections between these components.

Since different views support different goals and uses, the information needs of each system stakeholder for every view can vary depending on the stakeholder's role. Table 1 shows the information needs of the stakeholders for each of the 6 system views.

Someone new to the project should read the architecture description document to understand how views are documented and what the different views are. Architects are producing this architecture description document. Developers are the primary target of the architecture document and as such most if not all of the views are relevant reading. A top-down reading is to start with the *Use Case* view followed by the *Data*, *Logical*, *Process*, *Development* and *Physical* views. Users should read the *Use Case* view to gain an understanding on the functionality and capabilities of the system. Moreover, they should read the *Physical* view in order to learn how the system can be deployed. The augmenter's main role is to provide additional functionality to the platform. Therefore, augmenters will find useful reading all the sections of the architecture description document. To help with project management and planing, project managers should read mainly the *Logical* and *Use Case* views. Moreover, they should read the *Physical* view in order to understand the project requirements in hardware. Reviewers should read all the sections of this architecture description in order to understand and evaluate the proposed system.

In the following, the technical architecture will be decomposed along the aforementioned views. In addition, the architectural constraints will be presented.

2.2 Architectural View Decomposition

OSSMETER conforms to a 3-tier architecture as it is illustrated in Figure 3. In this architecture, the data access layer is responsible for storing the data of the system, as well as for providing access

Table 1: Stakeholders and their Architecture Documentation Requirements

Stakeholder	**Logical**	**Process**	**Development**	**Physical**	**Use Case**	**Data**
Architect	d	d	s	d	d	s
Developer	d	d	d	s	d	d
User	s	n/a	n/a	s	s	n/a
Augmenter	d	d	d	s	s	s
Manager	s	o	n/a	o	s	o
Reviewer	s	s	o	s	d	o

key: d= detailed information, s= some details, o= overview information, n/a= not applicable (adapted from [3])

mechanisms to the data. On top of the data access layer lies the logic layer. This layer is the core of the system and it consists of the OSSMETER platform, which is responsible for orchestrating the execution of the various metric providers and for providing uniform and seamless access to the data storage, and the metrics/fact providers, which extend the functionality of the system. The final layer of the proposed architecture is the presentation layer, which provides the various project analysis and management services of the system. These services will utilise the information generated and gathered by the functional process logic layer. In this tier the REST API and the web-client will reside. Since their detailed description is out of the scope of this document in the following section we will treat these components as a black box.

2.2.1 Use Case View

This view presents the users perception of the functionality provided by OSSMETER platform. The use cases are derived from the system requirements specified in deliverable D1.1 - *Project Requirements*. The main actors of the OSSMETER platform are the following:

- **Metric Provider**: developers, who extend the platform with additional metrics, fact extractors and connectors are the main users of the OSSMETER platform. Metric providers implement new metrics utilising the functionality provided by the platform for storing and retrieving data from the data persistence in a uniform manner. Moreover, they can use metrics and connectors implemented by other metric providers by using the integration provided by the platform.
- **End-User**: end-users are interested in monitoring and analysing OSS projects. They do not interact directly with the platform, but they interact with it by using the web client or the standalone application.

The first set of use cases is illustrated in Figure 4. The metric providers, which will sit atop the OSSMETER platform, will need to access various technical infrastructure technologies in order to retrieve their contents and perform the required measurements. The result of these measurements will be later used to assess the quality of the registered OSS projects. Table 2 describes in more detail the use cases captured in Figure 4.

During the preliminary domain analyses performed in the context of WP 2 and presented in deliverable D2.1 *"Domain Analysis of OSS Projects"*, three general categories of technical infrastructure

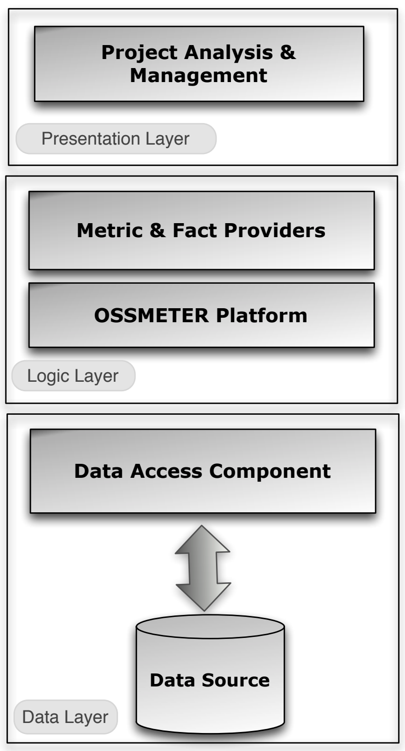


Figure 3: 3-Tier Architecture of OSSMETER System

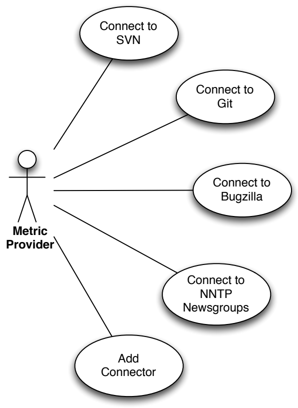


Figure 4: Connecting to technical infrastructure use cases.

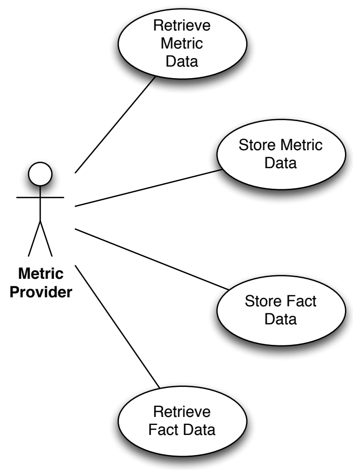


Figure 5: Storing and retrieving metric data use cases.

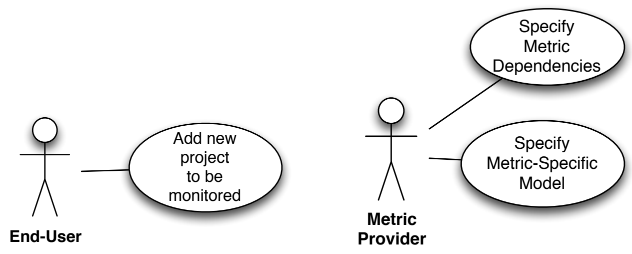
have been identified. The first category contains source code repositories such as *SVN* or *Git*. Moreover, web-based repositories such as SourceForge will be supported. Use cases *UC.1* and *UC.2* in Table 2 present in more detail the scenarios for connecting to *SVN* and *Git* respectively. The second category of infrastructure technologies, which will be supported, contains bugtracking systems such as *Bugzilla*. The use case related to this category is presented in *UC.3*. *UC.4* presents the use case related to the third category of infrastructure technologies, which is communication channels such as *NNTP* newsgroups. Finally, metric providers will need to be able to add additional connectors to the platform. For example, the *Mercurial* version control system might need to be added in order to support analysis of specific projects. This use case is presented in *UC.5*.

The second set of use cases is illustrated in Figure 5. These use cases capture the fact that metric providers need to store and retrieve data related to the measurements they make. First, they need to store data necessary for the measurement of the metrics. For, example to compute a set of metrics on a Java file, the Abstract Syntax Tree (AST) of the Java code needs to be extracted and stored. In the OSSMETER terminology, this type of data is called *fact*. The second kind of data that metric providers need to store and retrieve is the actual measurements performed. Tables 3 and 4 describe these use cases in more detail. Figure 7(a) illustrates the use case in which an end-user of the system adds a new project to be monitored. Figure 7(b) illustrates two use cases, which are related to specifying metrics. Each metric, which will be contributed to the platform will have their own data model. This data model will be defined by the metric provider. This use case is described in *UC.11* of Table 5. Moreover, dependencies between different metrics are supported. Data calculated by one metric provider should be visible by other metric providers, as long as this dependency is defined in the metric, which is requesting this data. This use case is described in *UC.11* of Table 5.

Table 2: Use cases related to connecting to technical infrastructure

Use Case Element	Description
Name	ConnectToSVN
Number	UC.1
Priority	High
Description	Metric provider connects to an SVN repository.
Primary Actor	Metric Provider
Pre-condition	There is an active Internet connection or repository is local.
Post-condition	Metric provider has an active connection to the repository and can access its contents.
Name	ConnectToGit
Number	UC.2
Priority	High
Description	Metric provider connects to a Git repository.
Primary Actor	Metric Provider
Pre-condition	There is a local Git clone.
Post-condition	Metric provider has an active connection to the local Git repository and can access its contents.
Name	ConnectToBugzilla
Number	UC.3
Priority	High
Description	Metric provider connects to a Bugzilla isntance.
Primary Actor	Metric Provider
Pre-condition	There is an active Internet connection or Bugzilla instance is local. Morover, metric provider has valid access credentials.
Post-condition	Metric provider has an active connection to the Bugzilla instance and can access its contents.
Name	ConnectToNNTPNewsgroup
Number	UC.4
Priority	High
Description	Metric provider connects to a NNTP newsgroup.
Primary Actor	Metric Provider
Pre-condition	There is an active Internet connection or local copy of the newsgroup. Morover, metric provider has valid access credentials.
Post-condition	Metric provider has an active connection to the newsgroup and can access its contents.
Name	AddConnector
Number	UC.5
Priority	High
Description	Metric provider extends the platform by adding a new connector.
Primary Actor	Metric Provider
Pre-condition	The infrastructure technology has an API for connecting to it and retrieving its contents.
Post-condition	Metric provider extends the platform by contributing a new connector.

Figure 6: Project and metric specific use cases


End-User
Add new project to be monitored
Metric Provider
Specify Metric Dependencies
Specify Metric-Specific Model

(a) Add new project to the platform use case (b) Metric specific use cases

Table 3: Use cases related to storing and retrieving metric data

Use Case Element	Description
Name	StoreMetricData
Number	UC.6
Priority	High
Description	Metric provider stores a measurement in a database.
Primary Actor	Metric Provider
Pre-condition	There is an active connection to a local or remote database.
Post-condition	Metric provider stores a measurement of a metric unit in a database.

Use Case Element	Description
Name	RetrieveMetricData
Number	UC.7
Priority	High
Description	Metric provider retrieves a measurement from a database.
Primary Actor	Metric Provider
Pre-condition	There is an active connection to a local or remote database and the required data exist in the database.
Post-condition	Metric provider retrieves a measurement from a database.

Table 4: Use cases related to storing and retrieving fact data

Use Case Element	Description
Name	StoreFactData
Number	UC.8
Priority	High
Description	Metric provider stores fact data related to this metric in a database.
Primary Actor	Metric Provider
Pre-condition	There is an active connection to a local or remote database and fact data are computed.
Post-condition	Metric provider stores the computed fact data in a database.

Use Case Element	Description
Name	RetrieveFactData
Number	UC.9
Priority	High
Description	Metric provider retrieves fact data from a database.
Primary Actor	Metric Provider
Pre-condition	There is an active connection to a local or remote database and the fact data exist in the database.
Post-condition	Metric provider retrieves fact data from a database.

Table 5: Project and metric specific use cases

Use Case Element	Description
Name	AddNewProject
Number	UC.10
Priority	High
Description	End-user adds a new project to the database.
Primary Actor	End-user
Pre-condition	There is an active connection to a local or remote database and the project specific meta-data is available.
Post-condition	A new project to be monitored is added to the database.

Use Case Element	Description
Name	SpecifyMetricDependencies
Number	UC.11
Priority	High
Description	Metric provider defines dependencies to other metrics.
Primary Actor	Metric Provider
Pre-condition	There metric of interest is available.
Post-condition	Metric provider defines dependencies to other metrics, so that their data can be accessed.

Use Case Element	Description
Name	DefineMetricSpecificModel
Number	UC.12
Priority	High
Description	Metric provider defines the metric-specific data model.
Primary Actor	Metric Provider
Pre-condition	There is an active connection to a local or remote database and the required data exist in the database.
Post-condition	Metric provider retrieves a measurement from a database.

2.2.2 Data View

The *Data View* describes the data model of the OSSMETER platform. This view can act as a blueprint for creating the database structure. Moreover, it can provide a conceptual description of the objects in the system's domain model.

The first and most important decision in the context of the *Data View* has to do with the choice of the database that will be used for the system. Concerning data storage, there are the following requirements:

- **Open source**: the database used by the OSSMETER platform, should be open source. Similarly, the APIs associated with the database should be open source as well.
- **Scalable**: since the OSSMETER platform is expected to manage large datasets. Therefore, a database, which scales well, should be used.
- **Schema-less**: since OSSMETER should support heterogeneous and complicated metrics, the flexibility provided by schema-less databases is desired.
- **Deployable on different platforms**: the database should be deployable on different platforms, in order to support adequately the OSSMETER platform.

After considering the aforementioned requirements, we decided to use MongoDB[1] as the database of the OSSMETER platform. Apart from satisfying the above requirements, MongoDB is a mature choice. Its development started in 2007, while in 2009 it was open sourced with a AGPL license. MongoDB's Java drivers are released under the Apache Software License, so as long as we don't modify the code of MongoDB this is compatible with EPL, under which the OSSMETER platform will be distributed. Moreover, MongoDB is one of the most popular databases, which means that there is a wide user-base. This is proved by the Big Data Index published by Jaspersoft[2], in which MongoDB is the first database in downloads. MongoDB is a NoSQL database, which stores structured data as JSON-like documents with dynamic schemas. Moreover, it provides automatic horizontal scaling by supporting sharding[3].

The OSSMETER platform needs to store the following data in the database:

- Project-specific meta-data
- Metrics and metric-specific data associated with each project registered to the platform.

A MongoDB instance can have several databases. In the context of OSSMETER, there is one database called OSSMETER, in which project specific meta-data is stored. This meta-data captures information required by the platform in order to assess the quality of a project such as repository URLs, bug tracking systems, communication channels, etc. Such meta-data will conform to domain-specific metamodels, which will be defined in the context of WP 2 and presented in deliverable D2.2 *"Metamodels for Describing OSS projects"*. Apart from the OSSMETER database, there is a database for each of the registered projects, which are named after the project they correspond to. These databases capture calculated metrics and information required by the metrics. This structure

[1] http://www.mongodb.org/
[2] http://www.jaspersoft.com/
[3] Sharding distributes a single logical database system across a cluster of machines.

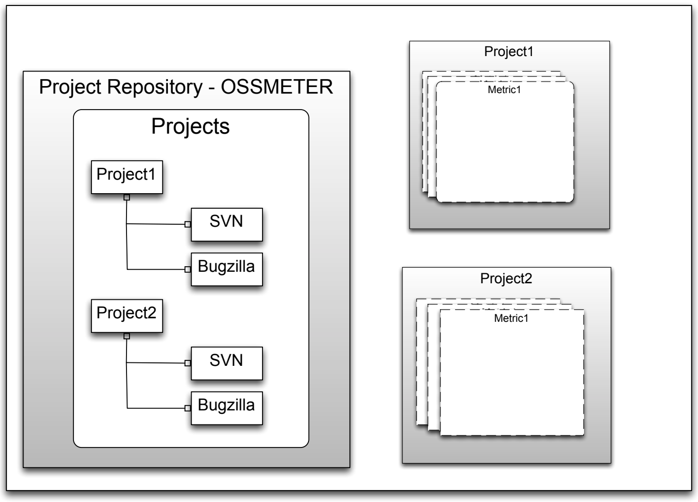


Figure 7: Project repository data model.

is illustrated in Figure 7. The project repository contains a single collection named *Projects*. This collection contains a document for each of the registered projects. This document is a JSON-like structure, which captures the project specific meta-data and it is named after each project.

The databases dedicated to the registered projects can contain a collection for each applicable metric. The data model for each metric is specified by the metric provider as it is described in Section 2.2.5. An example of the structure of the data in a metric repository is illustrated in Figure 8, where it is shown how it is arranged for the *Lines of Code* metric. The first collection named *SVN Repo Data* captures the measurements for total lines of code of all the files in an SVN repository. This collection consists of different documents (SVN1, SVN2) for each SVN repository used by the project under consideration. Each document contains 3 fields. The *URL* field captures the URL address of the SVN, the *Revision No.* field captures the revision number of the head of the repository and finally the *Lines of Code* field captures the total lines of code for that SVN. The second collection named *Loc1* captures the measurements for the lines of code for every single file in the repositories of the project under consideration. In this collection there is a document for each file and each document has an attribute which represents the URL of the file, an attribute which captures its revision number and an attribute to capture the number of lines of code of the file.

2.2.3 Logic View

In the *Logic View*, the OSSMETER platform is decomposed into a set of non-overlapping and collaborating components. The main focus of this view is on the functional architecture of the platform. A component diagram, which shows the aforementioned decomposition is illustrated in Figure 9.

Platform : at the crux of the proposed architecture is the *Platform* component. This is the main component, which is responsible for managing the overall functioning of the platform. The *Platform* component is responsible for setting up the other components, as well as for their execution.

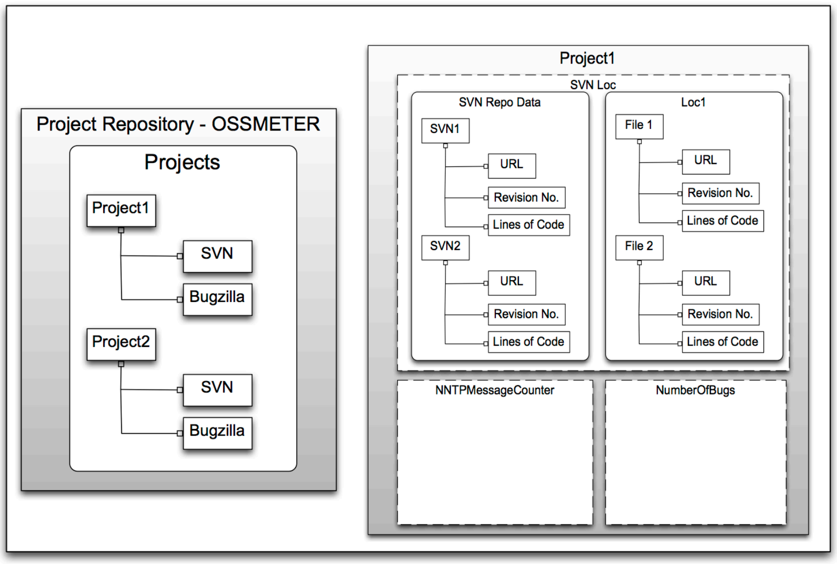


Figure 8: Example of a metric repository data model.

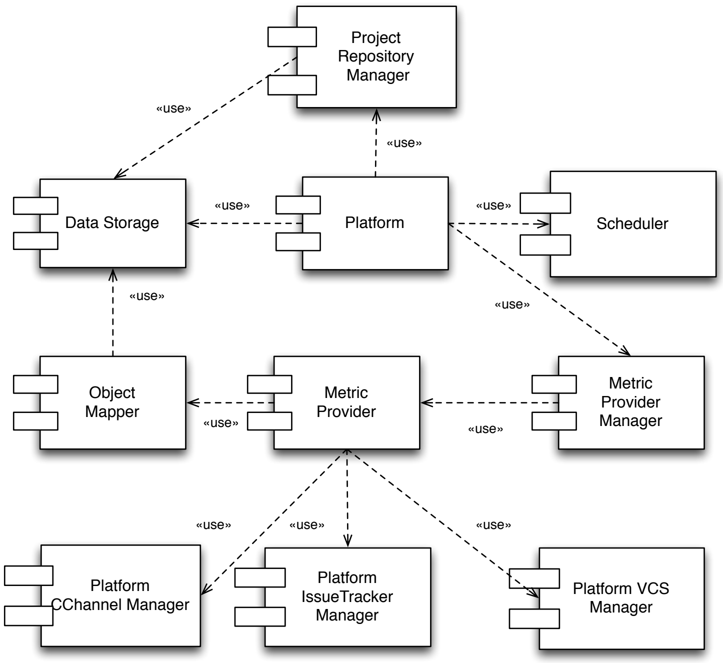


Figure 9: Component Diagram of OSSMETER Platform

Scheduler : the *Scheduler* component is responsible for orchestrating the execution of the various metrics. The goal of the scheduler will be to utilise scheduling techniques in order to maximise throughput (number of metrics that complete their execution per unit of time) as well as to minimise turnaround (total time between the start of the execution of a metric and its completion).

ProjectRepositoryManager : the main responsibility of the *ProjectRepositoryManager* component is to manage new and existing OSS projects in the database. This component provides mechanisms to add new projects and their associated meta-data into the database, as well as to delete or update projects, which are already in it.

DataStorage : this component provides the data storage facilities of the platform. The platform uses this component to store project meta-data, measurements of metrics or data which are required for the computation of metrics.

MetricProviderManager : the main responsibility of this component is to manage the metric providers build atop the OSSMETER platform. This component discovers the user-defined metric providers and then returns them to the platform in order to be executed.

MetricProvider : this component consists of the built-in and user defined metrics of the platform. These metrics measure different quality attributes of OSS projects and their measurements are stored in the data storage.

ObjectMapper : the *ObjectMapper* component provides an abstraction layer above the data storage. It enables the developers of metric providers to interact with the data storage without using low-level database operations.

PlatformVCSManager : this component is responsible for handling connections to a VCS repository database regardless of the type of the repository (e.g. SVN, Git, Mercurial, etc). Moreover, it provides operations for calculating a set of changes between two given revisions.

PlatformIssueTrackerSManager : this component is responsible for handling connections to an issue tracking system regardless of its type (e.g. Bugzilla, Googlecode issue tracker, etc). Moreover, it provides operations for calculating a set of changes between two given moments in time.

PlatformCChannelManager : this component is responsible for handling connections to a communication channel system regardless of its type (e.g. NNTP newsgroup, etc). Moreover, it provides operations for calculating a set of changes between two given revisions.

Table 6 summarizes all the system components and their dependencies.

2.2.4 Development View

The development view focuses on the actual software design of the OSSMETER platform. The static structure of the system is illustrated in Figure 10. This structure focuses on the core of the system only. In Table 7 the classes comprising the system are described briefly.

The basic idea behind the proposed architecture is that the platform is using the scheduler to execute the quality metrics built atop the platform. These metrics are discovered by the platform by utilising

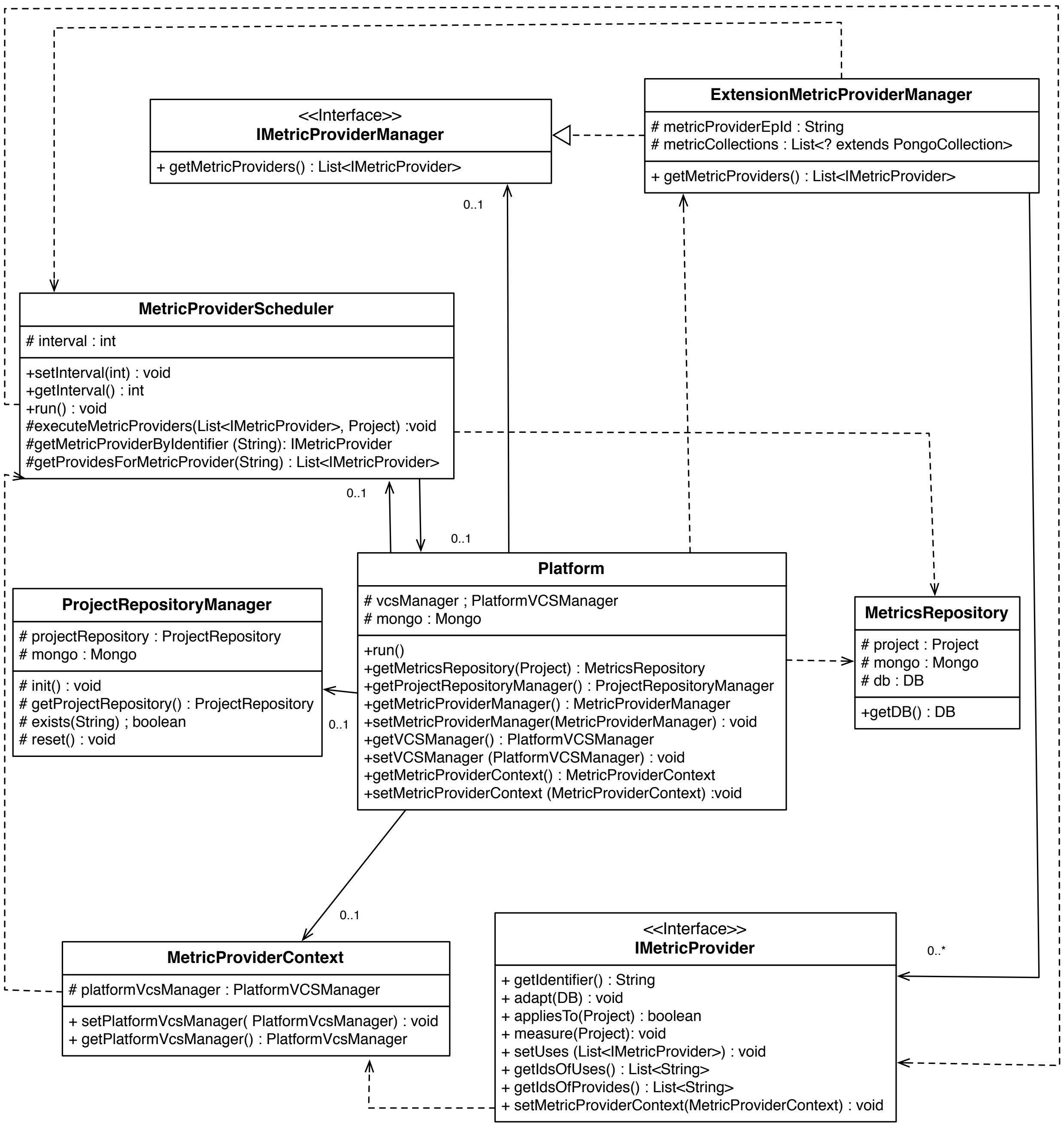

<<Interface>>
IMetricProviderManager
+ getMetricProviders() : List<IMetricProvider>
ExtensionMetricProviderManager
# metricProviderEpId : String
# metricCollections : List<? extends PongoCollection>
+ getMetricProviders() : List<IMetricProvider>
MetricProviderScheduler
# interval : int
+setInterval(int) : void
+getInterval() : int
+run() : void
#executeMetricProviders(List<IMetricProvider>, Project) :void
#getMetricProviderByIdentifier (String): IMetricProvider
#getProvidesForMetricProvider(String) : List<IMetricProvider>
Platform
# vcsManager ; PlatformVCSManager
# mongo : Mongo
+run()
+getMetricsRepository(Project) : MetricsRepository
+getProjectRepositoryManager() : ProjectRepositoryManager
+getMetricProviderManager() : MetricProviderManager
+setMetricProviderManager(MetricProviderManager) : void
+getVCSManager() : PlatformVCSManager
+setVCSManager (PlatformVCSManager) : void
+getMetricProviderContext() : MetricProviderContext
+setMetricProviderContext (MetricProviderContext) :void
ProjectRepositoryManager
# projectRepository : ProjectRepository
# mongo : Mongo
# init() : void
# getProjectRepository() : ProjectRepository
# exists(String) ; boolean
# reset() : void
MetricsRepository
# project : Project
# mongo : Mongo
# db : DB
+getDB() : DB
MetricProviderContext
# platformVcsManager : PlatformVCSManager
+ setPlatformVcsManager(PlatformVcsManager) : void
+ getPlatformVcsManager() : PlatformVcsManager
<<Interface>>
IMetricProvider
+ getIdentifier() : String
+ adapt(DB) : void
+ appliesTo(Project) : boolean
+ measure(Project): void
+ setUses (List<IMetricProvider>) : void
+ getIdsOfUses() : List<String>
+ getIdsOfProvides() : List<String>
+ setMetricProviderContext(MetricProviderContext) : void
0..1
0..1
0..1
0..1
0..1
0..*

Figure 10: OSSMETER Platform static structure

Table 6: System Components

#	Component Name	Type	Component Dependecies
1	Platform	Subsystem	2, 3, 4, 5
2	Scheduler	Subsystem	
3	ProjectRepositoryManager	Subsystem	4
4	DataStorage	External System	
5	MetricProviderManager	Subsystem	6
6	MetricProvider	Subsystem	7, 8, 9, 10
7	ObjectMapper	External System	4
8	PlatformVCSManager	Subsystem	
9	PlatformIssueTrackerSManager	Subsystem	
10	PlatformCChannelManager	Subsystem	

an extension point mechanism. Every metric provider must define its extension point in order to be visible by the platform. The various manager entities provided by the platform provide access to the data storage facility which can be used to store and retrieve data.

Table 7: System Element Description

Element Type	Name	Description
Class	**Platform**	This is the main class of the OSSMETER platform. It is responsible for setting up the various components of the platform and for executing it.
Attribute	vcsManager: PlatformVcsManager	Declares the manager which handles connections to VCS repositories such as SVN or Git.
Attribute	mongo : Mongo	Declares the database which will be used for storing and retrieving OSSMETER related data.
Method	getMetricsRepository(. . .)	Returns the metrics repository associated to a specific project.
Method	getProjectRepositoryManager()	Returns the manager, which is managing the project repository.
Method	getMetricProviderManager()	Returns the manager, which is managing the metric providers of the platform.
Method	setMetricProviderManager(. . .)	Sets the manager, which is managing the metric providers of the platform.
Method	getVCSManager()	Returns the manager which handles connections to VCS repositories such as SVN or Git.
Method	setVCSManager(. . .)	Sets the manager which handles connections to VCS repositories such as SVN or Git.
Method	getMetricProviderContext()	Returns the manager which handles connections to VCS repositories such as SVN or Git for a specific MetricProvider.
Method	setMetricProviderContext(. . .)	Sets the manager which handles connections to VCS repositories such as SVN or Git for a specific MetricProvider.

Table 7: Continued

Interface	**IMetricProviderManager**	Defines the IMetricProviderInterface. In this implementation metric providers are discoverable by the platform by using the extension point mechanism of Eclipse.
Attribute	getMetricProviders()	Returns a list of all metric providers, which have been registered to the platform.
Class	**ExtensionMetricProviderManager**	Implements the interface of IMetricProviderManager.
Attribute	metricProviderEpId : String	Declares the id of the extension point.
Attribute	metricCollections : List<?>	Declares the collections in the database which belong to a given metric provider.
Method	getMetricProviders()	Returns a list of all metric providers, which have been registered to the platform. This method utilises the extension point mechanism of Eclipse.
Class	**MetricProviderScheduler**	The scheduler class is responsible for scheduling the execution of the platform's metrics.
Attribute	interval : int	The execution of the metrics takes place at regular intervals. This attribute defines this interval.
Method	setInterval(. . .)	Sets platform's interval.
Method	getInterval()	Returns platform's interval.
Method	executeMetricProviders(. . .)	Executes the metric providers registered to the platform.
Method	getMetricProviderByIdentifier(. . .)	Returns the metric provider with the given identifier.
Method	getProvidesForMetricProvider(. . .)	Returns a list of metric providers, which contribute to a given metric provider. It is used to discover dependencies between metric providers.
Class	**ProjectRepositoryManager**	This manager class is responsible for registering projects to the platform as well as for managing already registered projects.
Attribute	projectRepository : ProjectRepository	This attribute represents the repository, where project metadata are stored.
Attribute	mongo : Mongo	Declares the database which will be used for storing and retrieving OSSMETER related data.
Method	init()	Initialises the project repository.
Method	getProjectRepository()	Returns the project repository.
Method	exists(. . .)	Checks if a given project exists in the repository.
Method	reset()	Resets the project repository.
Class	**MetricsRepository**	This class represents the repository for the data associated with a given project.
Attribute	project :	Declares the project to which the repository belongs.
Attribute	mongo : Mongo	Declares the database which will be used for storing and retrieving OSSMETER related data.
Attribute	db : DB	The repository associated to a given project
Method	getDB()	Returns the repository of a given project

Table 7: Continued

Class	**MetricProviderContext**	This class defines the context of a metric provider. This context can be connectors to use or other relevant classes.
Attribute	platformVcsManager : PlatformVcsManager	This attribute represents the generic manager, which handles connections to VCS repositories.
Method	setPlatformVcsManager(. . .)	Define that a given connector is in the context of a metric.
Method	getPlatformVcsManager()	Returns the manager, which handles connections to VCS repositories, for a given metric.
Interface	**IMetricProvider**	This interface must be implemented by anyone who wants to extend the platform with additional metrics.
Method	getIdentifier()	Returns the unique identifier of a metric provider.
Method	adapt(. . .)	Defines the data in the database to which the metric provider has access.
Method	appliesTo(. . .)	Checks if a metric provider is applicable for a given project.
Method	measure(. . .)	This is the main method of a metric provider. It calculates the metric for a given project.
Method	setUses(. . .)	Defines dependencies between metric providers. It specifies which other metric providers the given one uses.
Method	getIdsOfUses()	Returns the ids of the metric providers upon which the given one is dependant.
Method	getIdsOfProvides()	A metric provider can provide data and functionality to other metric providers. This method returns the ids of these metric providers.
Method	setMetricProviderContext(. . .)	Sets the context of the metric provider.

In Figure 10, the core of the OSSMETER platform was presented. Another interesting aspect of the platform is the definition of generic connectors and how these connectors can be used by the platform in order to increase throughput and minimise turnaround. These connectors abstract away the particularities of similar sources of information such as different types of VCS repositories. Moreover they minimise the number of connections of the different, independently-developed metric providers to such sources in order to economise network resources. They do this by caching locally those parts of the information source, which have changed between two points in time. Figure 11 illustrates the design of system, which is responsible for handling connections to VCS repositories.

The VCS manager provides a uniform way to connect to VCS repositories. Repository-specific connectors (for examples connectors for SVN or Mercurial) can extend the *AbstractVcsManager* class. When a repository-specific connector is specified and when a repository of this type is available for a given project, then the connector connects to the repository during the execution of the platform. There are two different options. First, if it is the first time this repository is accessed, then the connector caches locally the contents of the repository and these are made available to the platform's metric providers. In this way only one connect-fetch-disconnect cycle is required and therefore we economise network resources. On the other hand, if the repository has been accessed in the past,

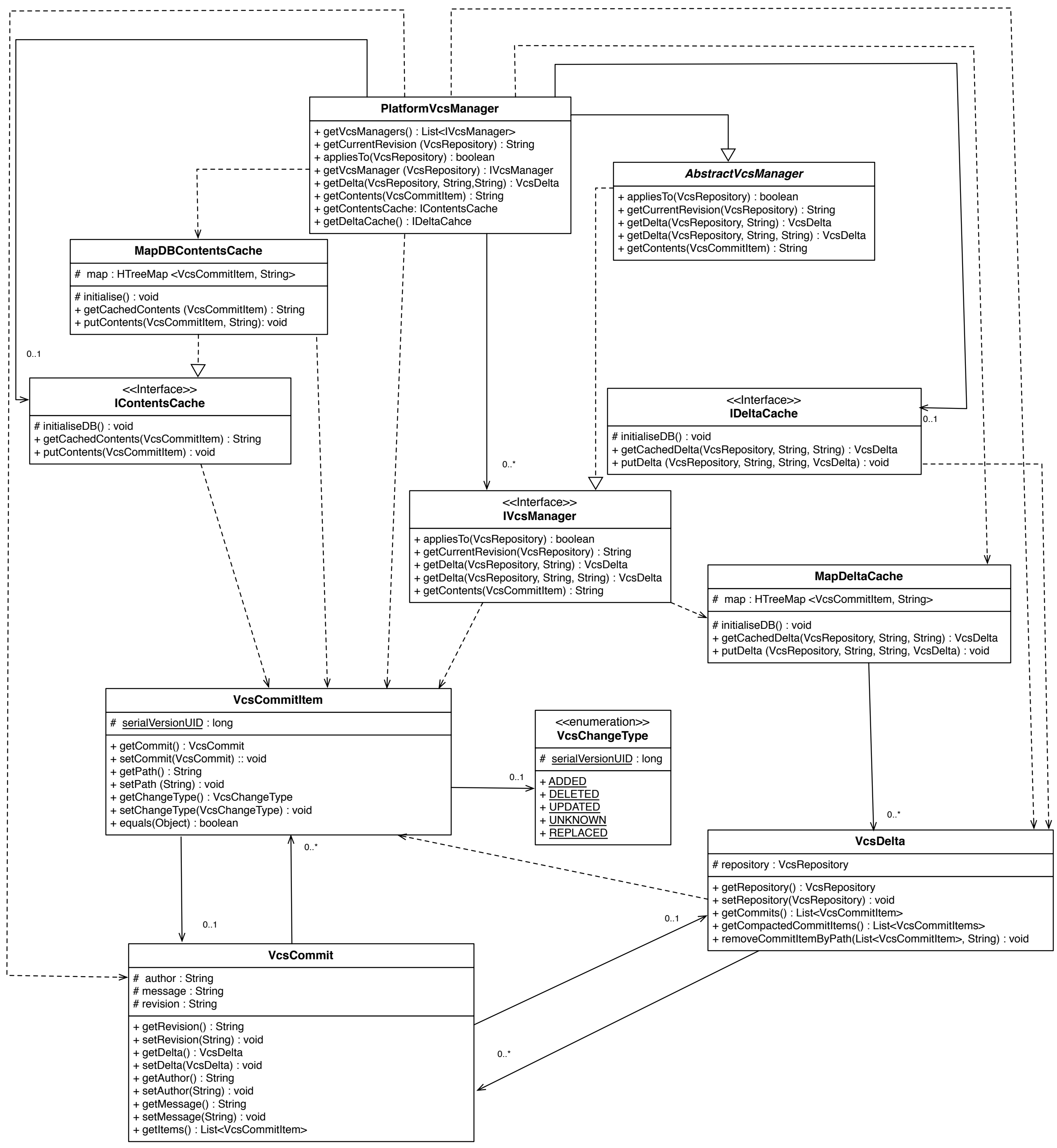

PlatformVcsManager
+ getVcsManagers() : List<IVcsManager>
+ getCurrentRevision (VcsRepository) : String
+ appliesTo(VcsRepository) : boolean
+ getVcsManager (VcsRepository) : IVcsManager
+ getDelta(VcsRepository, String,String) : VcsDelta
+ getContents(VcsCommitItem) : String
+ getContentsCache: IContentsCache
+ getDeltaCache() : IDeltaCahce
AbstractVcsManager
+ appliesTo(VcsRepository) : boolean
+ getCurrentRevision(VcsRepository) : String
+ getDelta(VcsRepository, String) : VcsDelta
+ getDelta(VcsRepository, String, String) : VcsDelta
+ getContents(VcsCommitItem) : String
MapDBContentsCache
# map : HTreeMap <VcsCommitItem, String>
# initialise() : void
+ getCachedContents (VcsCommitItem) : String
+ putContents(VcsCommitItem, String): void
0..1
<<Interface>>
IContentsCache
# initialiseDB() : void
+ getCachedContents(VcsCommitItem) : String
+ putContents(VcsCommitItem) : void
<<Interface>>
IDeltaCache
# initialiseDB() : void
+ getCachedDelta(VcsRepository, String, String) : VcsDelta
+ putDelta (VcsRepository, String, String, VcsDelta) : void
0..1
0..*
<<Interface>>
IVcsManager
+ appliesTo(VcsRepository) : boolean
+ getCurrentRevision(VcsRepository) : String
+ getDelta(VcsRepository, String) : VcsDelta
+ getDelta(VcsRepository, String, String) : VcsDelta
+ getContents(VcsCommitItem) : String
MapDeltaCache
# map : HTreeMap <VcsCommitItem, String>
# initialiseDB() : void
+ getCachedDelta(VcsRepository, String, String) : VcsDelta
+ putDelta (VcsRepository, String, String, VcsDelta) : void
VcsCommitItem
# serialVersionUID : long
+ getCommit() : VcsCommit
+ setCommit(VcsCommit) :: void
+ getPath() : String
+ setPath (String) : void
+ getChangeType() : VcsChangeType
+ setChangeType(VcsChangeType) : void
+ equals(Object) : boolean
0..1
<<enumeration>>
VcsChangeType
# serialVersionUID : long
+ ADDED
+ DELETED
+ UPDATED
+ UNKNOWN
+ REPLACED
0..*
VcsDelta
# repository : VcsRepository
+ getRepository() : VcsRepository
+ setRepository(VcsRepository) : void
+ getCommits() : List<VcsCommitItem>
+ getCompactedCommitItems() : List<VcsCommitItems>
+ removeCommitItemByPath(List<VcsCommitItem>, String) : void
0..*
0..1
0..1
VcsCommit
# author : String
# message : String
# revision : String
+ getRevision() : String
+ setRevision(String) : void
+ getDelta() : VcsDelta
+ setDelta(VcsDelta) : void
+ getAuthor() : String
+ setAuthor(String) : void
+ getMessage() : String
+ setMessage(String) : void
+ getItems() : List<VcsCommitItem>
0..*

Figure 11: OSSMETER Platform VCS manager.

then to optimise even more the performance of the platform, only the delta since the last time the repository was accessed is cached locally. To this end the revision numbers provided by the various VCS systems are very helpful. Table 8 describes briefly the elements of the *PlatformVcsManager* component.

Table 8: VCS Manager Subsystem Element Description

Element Type	Name	Description
Interface	**IVcsManager**	This is the interface that defines the methods that developers of repository-specific connectors need to implement.
Method	appliesTo(...)	Checks if the particular connector applies to a specific repository
Method	getCurrentRevision(...)	Returns the latest revision number of th repository.
Method	getDelta(...)	Calculates and returns a delta from the given revision until the last one.
Method	getDelta(...)	Calculates and returns a delta between two given revision numbers.
Method	getContents(...)	Returns the contents of a given commit item.
Class	**AbstractVcsManager**	This abstract class implements the IVcsManager interface.
Method	appliesTo(...)	Checks if the particular connector applies to a specific repository
Method	getCurrentRevision(...)	Returns the latest revision number of th repository.
Method	getDelta(...)	Calculates and returns a delta from the given revision until the last one.
Method	getDelta(...)	Calculates and returns a delta between two given revision numbers.
Method	getContents(...)	Returns the contents of a given commit item.
Class	**PlatformVcsManager**	Extends and provides concrete implementation to the AbstractVcs-Manager abstract class.
Method	appliesTo(...)	Checks if the particular connector applies to a specific repository.
Method	getCurrentRevision(...)	Returns the latest revision number of th repository.
Method	getDelta(...)	Calculates and returns a delta from the given revision until the last one.
Method	getDelta(...)	Calculates and returns a delta between two given revision numbers.
Method	getContents(...)	Returns the contents of a given commit item.
Method	getContentsCache()	Returns the contents cache associated with this manager.
Method	getDeltaCache()	Returns the delta cache associated with this manager.
Method	getVcsManagers()	Returns all the VCS managers available.
Method	getVcsManager(...)	Return manager associated with a specific type of repository.
Interface	**IContentsCache**	This interface defines the methods related to caching the contents of a repository.
Method	initialiseDB()	Initialises the cache.
Method	getCachedContents(...)	Returns the contents from the cache for a given commit item.
Method	putContents(...)	Stores the contents of a given commit item into the cache.
Class	**MapDBContentsCache**	This implementation of the IContentsCache uses the MapDB, which provides provides concurrent TreeMap and HashMap backed by an embedded database engine.
Attribute	map: TreeMap	This is a collection of commit items and their contents.
Method	initialiseDB()	Initialises the cache.
Method	getCachedContents(...)	Returns the contents from the cache for a given commit item.
Method	putContents(...)	Stores the contents of a given commit item into the cache.
Interface	**IDeltaCache**	This interface defines the methods related to caching the deltas of a repository.
Method	initialiseDB()	Initialises the cache.
Method	getCachedContents(...)	Returns the delta from the cache.
Method	putContents(...)	Stores a delta into the cache.
Class	**MapDBDeltaCache**	This implementation of the IDeltaCache uses the MapDB, which provides provides concurrent TreeMap and HashMap backed by an embedded database engine.

Table 8: Continued

Attribute	map: TreeMap	This is a collection of commit items and their contents.
Method	initialiseDB()	Initialises the cache.
Method	getCachedContents(...)	Returns the delta from the cache.
Method	putContents(...)	Stores a delta into the cache.
Class	**VcsCommitItem**	This class represents a single commit item.
Attribute	serialVersionUID : long	This is a unique identifier of the commit item.
Method	getCommit()	Returns the commit to which a commit item belongs.
Method	setCommit(...)	Sets the commit of a commit item.
Method	getPath()	Returns the path of the commit item in the repository.
Method	setPath(...)	Sets the path of a commit item in the repository.
Method	getChangeType()	Returns the change type of the commit item.
Method	setChangeType(...)	Sets the change type of a commit item.
Method	equals(...)	Compares two commit items.
Class	**VcsCommit**	This class represents a repository commit.
Attribute	author: String	This is the individual performing the commit.
Attribute	message: String	This is the message attached to a commit.
Attribute	revision : String	This captures the revision number of a commit.
Method	getRevision()	Returns the revision of a commit.
Method	setRevision(...)	Sets the revision of a commit.
Method	getDelta()	Returns the delta that is created by the commit.
Method	setDelta(...)	Sets the delta of the commit.
Method	getAuthor()	Returns the author.
Method	setAuthor(...)	Sets the author.
Method	getMessage()	Returns the message of a commit.
Method	setMessage(...)	Sets the message of a commit.
Method	getItems()	Returns a list of the commit items of the commit.
Class	**VcsDelta**	This class represents the delta of a commit.
Attribute	repository: VcsRepository	This is the repository the commit coresponds to.
Method	getRepository()	Returns the repository for a given delta.
Method	setRepository(...)	Sets the repository for a given delta.
Method	getCommits()	Returns the commits associated with a given delta.
Method	getCompactedCommitItems()	Returns the committed items associated with a given delta.
Method	removeCommitItemByPath()	Removes a commit item from a list by a given path.
Enumeration	**VcsChangeType**	This enumeration defines the different change types for commit items.
Literal	ADDED	A commit item is added to the repository.
Literal	DELETE	A commit item is deleted from the repository.
Literal	UPDATED	A commit item is modified in the repository.
Literal	REPLACED	A commit item is replaced by another one in the repository.
Literal	UNKNOWN	An unknown change type.

2.2.5 Process View

The process view deals with the dynamic aspects of the OSSMETER platform. It explains at a high level of abstraction how the various metrics of the system are executed. The sequence diagram illustrated in Figure 12 models the runtime behavior of the platform.

The *Platform* component is responsible for setting up the platform and for initializing its execution. When the scheduler is initialised, it requests from the platform all the projects, which have been

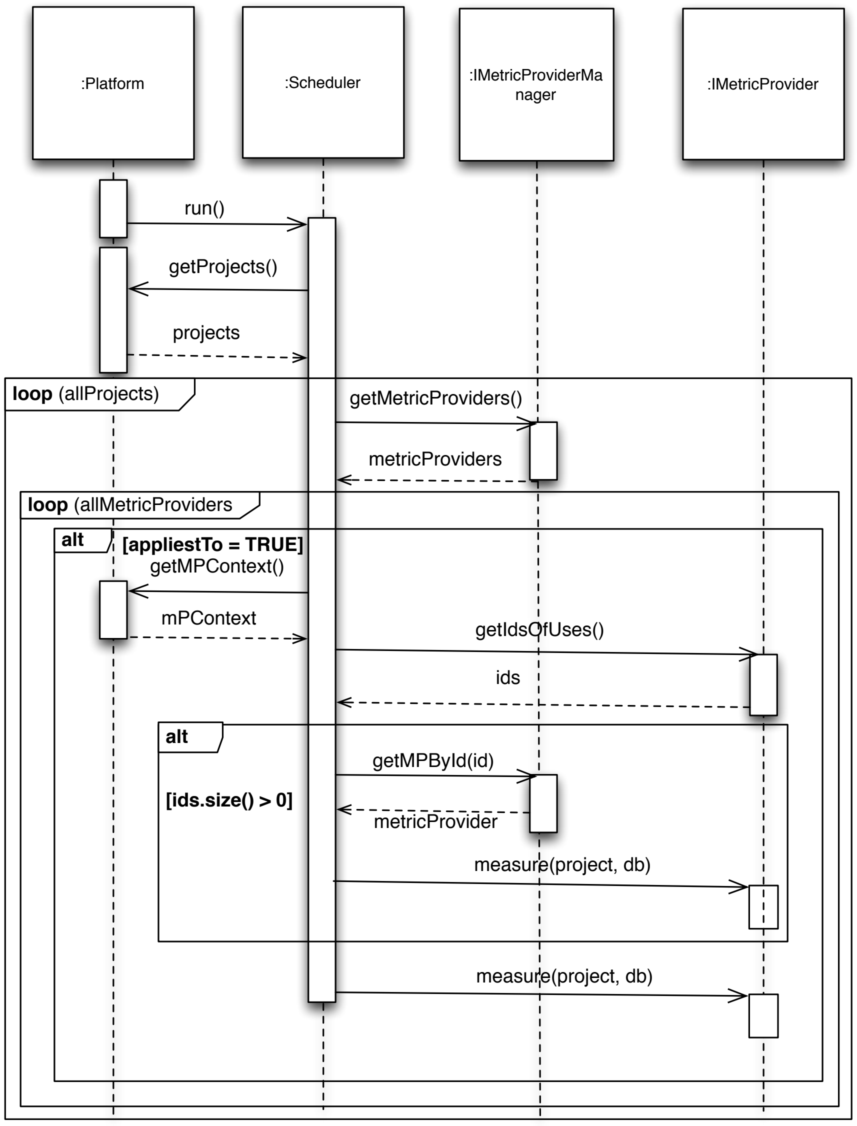


Figure 12: OSSMETER Platform runtime behavior

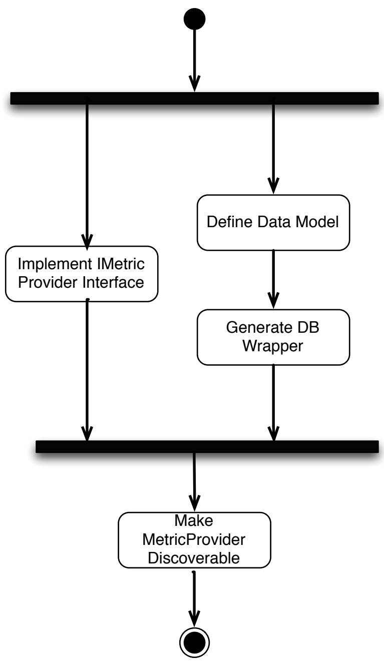

Define Data Model
Implement IMetric Provider Interface
Generate DB Wrapper
Make MetricProvider Discoverable

Figure 13: Process for the implementation of a custom metric provider

registered. Then for every project, which has been registered, the metric provider manager returns all the metric providers. Next, for each applicable metric provider, the scheduler fetches its context and its dependencies. The context of a metric provider is an appropriate connector to a data source. For example the context of a metric provider, which needs to access a VCS repository, is a VCS manager. The VCS manager is part of the *PlatformVCSManager* component. Metric providers in OSSMETER platform can rely on other metric providers for pieces of data. These dependencies are provided by the metric provider itself by using the *getIdsOfUses* method, which returns a list of the ids of the dependencies. If the list returned by the *getIdsOfUses* method is not empty, then the scheduler ensures that the dependencies of a metric provider are executed before the metric provider itself. In the future, more intelligent scheduling policies will be investigated in order to optimise throughput and turnaround. Moreover a mechanism to detect cyclic dependencies will be added.

The activity diagram illustrated in Figure 13 shows how the platform can be extended by implementing custom metric providers. The first task that must be done by the developers of metric providers is to implement the *IMetricProviderInterface*. In section 2.2.4 more details about the methods and parameters of the interface are provided.

The second task for developing a custom metric provider is the definition of the data model of the metric and the generation of the data base wrapper classes. In the context of OSSMETER we use Pongo [4] to define the data model. Pongo is a template based Java POJO generator for MongoDB. Instead of using low-level database objects to interact with the MongoDB database, with Pongo metric provider developers can define the data/domain model using Emfatic [2] and then generate strongly-typed Java classes, which can then be used to work with the database at a more convenient level

[4]https://code.google.com/p/pongo/

of abstraction. There are many different alternatives to Pongo. Such alternatives include Morphia [5], MJORM [6] and Jongo [7]. Despite the fact that different alternatives exist, in the context of OSSMETER we have chosen to develop and use an in-house built Object Document Mapper (ODM). This decision was taken mainly because of the maturity of the existing solutions. Existing ODMs are not mature enough, and since Pongo is an important component of the OSSMETER platform we needed a solution which we could control and modify depending on our needs.

The final task for developing a custom metric provider is to make the metric provider discoverable by the platform. The OSSMETER platform is built atop Eclipse [8]. The plug-in architecture of Eclipse makes it easy for developers to extend the functionality of the runtime system, which is based on the Equinox [9] implementation of the OSGi specification [1]. To enable developers to extend the functionality of the OSSMETER platform, we take advantage of the extension mechanism of Eclipse. An example of the definition of an extension point is shown in Listing 2.2.5. Once such an extension point is defined for a custom metric provider, then this provider can be detected by the OSSMETER platform at runtime.

```
<extension point="org.ossmeter.repository.metricprovider">
  <metricProvider provider="org.ossmeter.metricprovider.loc.LocMetricProvider"/>
</extension>
```

2.2.6 Physical View

This view describes the multi-tier physical architecture of the OSSMETER platform, which is illustrated in Figure 14. The role of its tier is the following:

- In a Web application, the client-tier consists of an Internet browser that submits HTTP requests and downloads HTML pages from a Web server. The OSSMETER platform can be also deployed locally, therefore custom clients can be developed which communicate directly with the functional logic tier.
- The Web tier runs a Web server to handle requests and responds to these requests by querying the functional logic using the REST API provided by the OSSMETER system.
- The Functional Logic tier is the core of the system. It can reside on an application server or it can be deployed locally on the machine of the user. It supplies a number of services such as registering and managing OSS projects or executing metric providers. It communicates with the database using the Pongo ODM.
- The final tier of the system is the Data storage tier, which the OSSMETER must access to offer its services. This is the tier, where the MongoDB resides.

[5]https://code.google.com/p/morphia/
[6]https://code.google.com/p/mongo-java-orm/
[7]http://jongo.org/
[8]http://www.eclipse.org/
[9]http://www.eclipse.org/equinox/

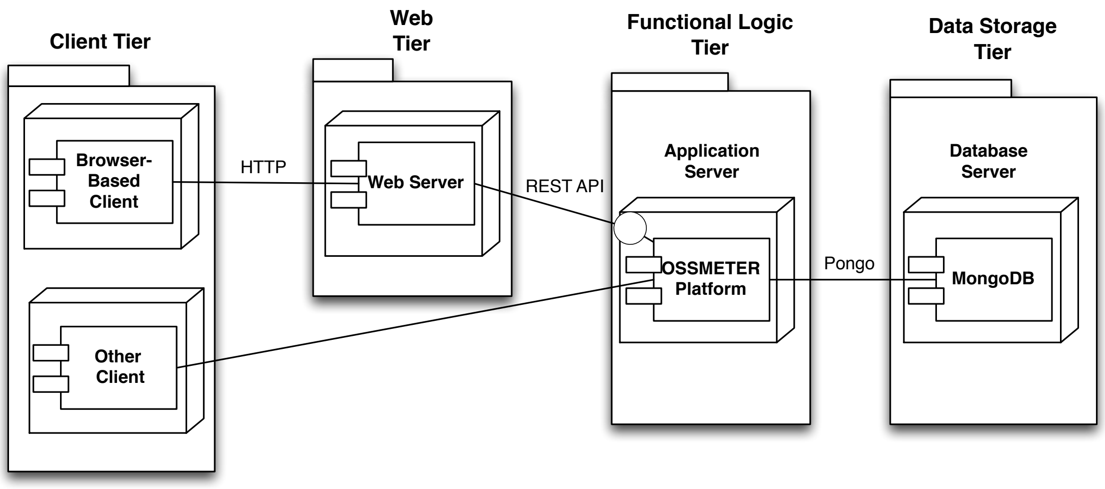


Figure 14: OSSMETER Platform deployment architecture

2.2.7 Quality Attributes

Table 9 summarises how the OSSMETER platform architecture supports key quality attribute requirements for the overall system architecture. Moreover, the approaches and tactics used to achieve them are presented briefly.

Table 9: System quality attribute requirements

Requirement	How Achieved	Related Tactics
Performance	Minimise connections per execution by using local caches. Schedule metric execution in a way to minimise access to the database.	Introduce concurrency, reduce demand and increase available resources.
Extensibility	Use of plug-in based approach (OSGi). Use of a schema-less database.	Anticipate extensions and use of Web Services.
Scalability	Use of database, which supports sharding. Use of caching	Load balancing, keep multiple copies of data and increase available resources.
Portability	Implementation based on Eclipse, which is Java-based and provision of a REST API. Use MongoDB, which can be deployed on multiple platforms.	Abstract common services.
Implementation Transparency	Provide complete transparency of implementation details by providing a REST API so that client programs are independent of implementation details such as operating system or component location.	Semantic coherence.

3 Conclusion

This document presented the evolving architecture of the OSSMETER platform. In the future, the architecture will be evaluated and revised in response to changing requirements and new stakeholder needs.

For the description of the system's architecture, we have used a modified version of the *4+1 view model*. Due to the data-driven nature of the OSSMETER platform, an additional view was added to this model. More specifically, a data view was added in order to capture the architecture of the data of the system as well as the data storage requirements.

Finally, in addition to the use of the modified *4+1 view model* for the description of the system's architecture, we have briefly presented the key quality attribute requirements.

References

[1] OSGI Alliance. OSGi Service Platform, Core Specification, Release 4, Version 4.2. Technical report, OSGI Alliance, September 2009.

[2] Miguel G. Bigeardel. Emfatic. http://wiki.eclipse.org/Emfatic, 2010.

[3] Paul Clements, David Garlan, Len Bass, Judith Stafford, Robert Nord, James Ivers, and Reed Little. *Documenting Software Architectures: Views and Beyond.* Pearson Education, 2002.

[4] Standards Committee. IEEE recommended practice for architectural description of software-intensive systems. *IEEE Std 1471-2000*, 2000.

[5] International Organization for Standardization. Systems and Software Engineering - Software Life Cycle Processes. *IEEE Std 12207-2008*, 2008.

[6] Philippe Kruchten. The 4+1 view model of architecture. *IEEE Softw.*, 12(6):42–50, November 1995.

[7] Luqi. Software evolution through rapid prototyping. *Computer*, 22(5):13–25, May 1989.

Abbreviations

API	Application Programming Interface
AST	Abstract Syntax Tree
OSS	Open Source Software
REST	Representational State Transfer