## Project Partner Contact Information

<table>
<thead>
<tr>
<th><strong>Centrum Wiskunde &amp; Informatica</strong></th>
<th><strong>SOFTEAM</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Jurgen J. Vinju</td>
<td>Alessandra Bagnato</td>
</tr>
<tr>
<td>Science Park 123</td>
<td>Avenue Victor Hugo 21</td>
</tr>
<tr>
<td>1098 XG Amsterdam, Netherlands</td>
<td>75016 Paris, France</td>
</tr>
<tr>
<td>Tel: +31 20 592 4102</td>
<td>Tel: +33 1 30 12 16 60</td>
</tr>
<tr>
<td>E-mail: <a href="mailto:jurgen.vinju@cwi.nl">jurgen.vinju@cwi.nl</a></td>
<td>E-mail: <a href="mailto:alessandra.bagnato@softeam.fr">alessandra.bagnato@softeam.fr</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Tecnalia Research and Innovation</strong></th>
<th><strong>The Open Group</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Jason Mansell</td>
<td>Scott Hansen</td>
</tr>
<tr>
<td>Parque Tecnologico de Bizkaia 202</td>
<td>Avenue du Parc de Woluwe 56</td>
</tr>
<tr>
<td>48170 Zamudio, Spain</td>
<td>1160 Brussels, Belgium</td>
</tr>
<tr>
<td>Tel: +34 946 440 400</td>
<td>Tel: +32 2 675 1136</td>
</tr>
<tr>
<td>E-mail: <a href="mailto:jason.mansell@tecnalia.com">jason.mansell@tecnalia.com</a></td>
<td>E-mail: <a href="mailto:s.hansen@opengroup.org">s.hansen@opengroup.org</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>University of L’Aquila</strong></th>
<th><strong>UNINOVA</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Davide Di Ruscio</td>
<td>Pedro Maló</td>
</tr>
<tr>
<td>Piazza Vincenzo Rivera 1</td>
<td>Campus da FCT/UNL, Monte de Caparica</td>
</tr>
<tr>
<td>67100 L’Aquila, Italy</td>
<td>2829-516 Caparica, Portugal</td>
</tr>
<tr>
<td>Tel: +39 0862 433735</td>
<td>Tel: +351 212 947883</td>
</tr>
<tr>
<td>E-mail: <a href="mailto:davide.diruscio@univaq.it">davide.diruscio@univaq.it</a></td>
<td>E-mail: <a href="mailto:pmm@uninova.pt">pmm@uninova.pt</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>University of Manchester</strong></th>
<th><strong>University of York</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Sophia Ananiadou</td>
<td>Dimitris Kolovos</td>
</tr>
<tr>
<td>Oxford Road</td>
<td>Deramore Lane</td>
</tr>
<tr>
<td>Manchester M13 9PL, United Kingdom</td>
<td>York YO10 5GH, United Kingdom</td>
</tr>
<tr>
<td>Tel: +44 161 3063098</td>
<td>Tel: +44 1904 325167</td>
</tr>
<tr>
<td>E-mail: <a href="mailto:sophia.ananiadou@manchester.ac.uk">sophia.ananiadou@manchester.ac.uk</a></td>
<td>E-mail: <a href="mailto:dimitris.kolovos@york.ac.uk">dimitris.kolovos@york.ac.uk</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Unparallel Innovation</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Nuno Santana</td>
</tr>
<tr>
<td>Rua das Lendas Algarvias, Lote 123</td>
</tr>
<tr>
<td>8500-794 Portimão, Portugal</td>
</tr>
<tr>
<td>Tel: +351 282 485052</td>
</tr>
<tr>
<td>E-mail: <a href="mailto:nuno.santana@unparallel.pt">nuno.santana@unparallel.pt</a></td>
</tr>
</tbody>
</table>
# Contents

1. **Introduction** ................................................. 2
   1.1 Structure of the deliverable .................................. 2

2. **Validation of the OSSMETER metamodels** ............... 3
   2.1 Validation of the OSSMETER metamodels with respect to Use Case requirements . 4
   2.2 Validation of the OSSMETER metamodels with respect to WP2 requirements ....... 9
   2.3 Summary of unmet requirements .................................. 11
   2.4 Feedback from OSSMETER partners ................................. 12

3. **Revised metamodels** ........................................... 15
   3.1 Base Forge metamodel ........................................ 17
   3.2 Subversion metamodel .......................................... 18
   3.3 Git metamodel ................................................. 18
   3.4 NNTP metamodel ............................................... 19
   3.5 Bugzilla metamodel ............................................ 19
   3.6 GitHub metamodel ............................................. 19
   3.7 Eclipse metamodel ............................................. 24
   3.8 Google Code metamodel ....................................... 27
   3.9 SourceForge metamodel ...................................... 30
   3.10 Redmine metamodel .......................................... 34

4. **Conclusions** .................................................. 38
Document Control

<table>
<thead>
<tr>
<th>Version</th>
<th>Status</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>Document outline</td>
<td>14 August 2013</td>
</tr>
<tr>
<td>0.2</td>
<td>First draft</td>
<td>7 September 2013</td>
</tr>
<tr>
<td>0.7</td>
<td>First full draft</td>
<td>18 September 2013</td>
</tr>
<tr>
<td>0.8</td>
<td>Further editing draft</td>
<td>27 September 2013</td>
</tr>
<tr>
<td>1.0</td>
<td>QA review</td>
<td>30 September 2013</td>
</tr>
<tr>
<td>1.1</td>
<td>Final version</td>
<td>15 October 2013</td>
</tr>
</tbody>
</table>
Executive Summary

Deciding whether an open source software (OSS) meets the required standards for adoption in terms of quality, maturity, activity of development and user support is not a straightforward process as it involves exploring various sources of information. The decision becomes even more challenging when one needs to discover and compare several OSS projects that offer software of similar functionality.

This document presents the process that has been conceived and applied to validate the metamodels presented in D2.1. The aim of such metamodels is to represent the key aspects that are crucial for analysing and comparing OSS projects. The validation process permitted to identify a number of issues in the initial version of the metamodels that have been accordingly refined and extended.
1 Introduction

In OSSMETER metamodels play a key role since they specify the key aspects of OSS projects (currently available from various sources and in different formats) that have to be considered for analysis and comparison purposes. The main challenges related to the definition of such metamodels are the following:

- what are the concepts and attributes more relevant to be represented for analyzing and comparing OSS projects?
- what are the concepts that have to be represented to enable the application of the OSSMETER analysis and comparison tools?
- what is the right level of abstraction that the metamodels should have?
- the boundaries of the OSS projects domain are blurred. Since a metamodel aims at formalizing the concepts of a given domain:
  - what is a precise definition of the OSS projects domain?
  - what is the right trade-off between metamodel expressiveness and domain specificity?

In D2.1 [2] a first version of the OSSMETER metamodels has been defined with the aim of creating models representing in a homogeneous manner different aspects of OSS projects (e.g. types and details of source code repositories, communication channels and bug tracking systems, types of licences, etc.).

In this document, we validate the metamodels in D2.1 and refine them by advancing the work that WP2 is doing according to the following tasks described in the OSSMETER DoW:

Task 2.1: Domain Analysis. Find use cases that capture the most used and relevant information that is used while managing and analysing OSS projects. In particular, given the standard life-cycle of an OSS project, identify what are the elements and artefacts that should be modelled in the context of OSSMETER.

Task 2.2: OSS Project Metamodel Development. Develop a set of EMF/Ecore based metamodels that capture the previously identified elements and that allow a reasonable description of the considered OSS project and the representation of its evolution over time.

Metamodels have been validated with respect to Use Case and WP2 requirements elicited and discussed in D1.1 [5]. Moreover, a short questionnaire has been circulated by emails among the different OSSMETER partners that provided some suggestions to improve the metamodels. The outcome of the validation activities is then considered to refine and evolve the metamodels.

1.1 Structure of the deliverable

The deliverable is structured as follows:

- Section 2 presents the process that has been followed to validate the metamodels presented in D2.1;
- Section 3 presents the new metamodels that have been conceived by refining those in D2.1 in order to fix the issues that have been identified and described in the previous section;
- Section 4 concludes the document.
2 Validation of the OSSMETER metamodels

The work of WP2 mainly concerns the specification and implementation of metamodels that permit to represent in a homogeneous manner different aspects of OSS projects. Having such models will enable different kinds of automatic analysis and will support objective comparisons of OSS alternatives with respect to user needs, and quality requirements. To this end, WP2 is developing two different kinds of metamodels in order to represent:

- key aspects of a given OSS project which are crucial for its analysis and management;
- metrics enabling automated measurement and comparison of OSS projects.

Project metrics metamodels will be the topic of the last deliverable of WP2 even though some activities related to their definition have been already started. In this section we discuss the validation activities we have done on the first version of the OSS forge metamodels, which are structured as shown in Figure 1. By referring to Figure 2 (initially presented in D1.1), this document is mainly about the activities Assessment of the forge metamodel expressiveness and Forge metamodel definition/refinement applied on the first version of the OSS forge metamodels presented in D2.1. More specifically, by means of the Assessment of the forge metamodel expressiveness the OSSMETER metamodels are analysed in order to check if there are unforeseen requirements that are not satisfied, or if the metamodels do not provide modelling constructs, which are relevant to represent selected OSS projects. To this end, the validation has been performed by considering the following sources of information:

- Use Case requirements, defined by the industrial partners of the project (see Table 1)
- WP2 requirements (see Table 2)
- Feedback from the partners of the project

According to the outcome of such a phase, in the Forge metamodel definition/refinement phase, new concepts are added, and the identified issues are fixed as discussed in the next section.

Figure 1: Structure of the initial version of the OSSMETER metamodels as presented in D2.1
2.1 Validation of the OSSMETER metamodels with respect to Use Case requirements

As discussed in D1.1 the industrial Use Case requirements specification is intended to provide a quantitative view of the envisioned OSSMETER solution, stating measurable criteria that should be met during the implementation of the research and development tasks within the whole project. In this respect, since the OSSMETER metamodels play a key role in the project, and underpin the data exchanged and managed by all the tools developed in the technical work packages, it is necessary to validate the expressiveness of the OSSMETER metamodel with respect to the Use Case requirements. In particular, in D1.1 the following requirements categories are identified:

- C1: General requirements
- C2: OSS project level requirements
- C3: Lifecycle related requirements
- C4: Code quality requirements
- C5: OSS forum activity requirements
- C6: Tracking system requirements
- C7: Platform configurability requirements
- C8: Platform extensibility requirements
The requirements category that is strictly related with the work in WP2 and in particular with the definition of the OSSMETER metamodels is C2. The other requirements categories underpin the work of other WPs i.e., C1, C3, C7, C8 are related with the work of WP5, C4 is related with WP3, and C5, C6 are related with WP4.

In the following the OSSMETER metamodels defined in D2.1 are discussed with respect to the requirements in C2. All the requirements are identified in terms of the modalities SHALL, SHOULD, and MAY defined as follows [5]:

- **SHALL** is used to denote an essential requirement. A typical system could not be used, would not work, or cannot be validated if this requirement is not fulfilled. SHALL requirements are of highest priority for validation of the platform.
- **SHOULD** is used to denote a requirement that would help a typical system be easier to use, or to work better, even if it is not essential; in that case a trade-off can be achieved between development costs on the technology side and user benefit on the system side.
- **MAY** is used to denote a requirement that can lead to a benefit in order to fulfil an additional evaluation criterion or increase the usefulness of the technology. The fulfillment of the requirement is interesting but only in view of available resources and research and development partner interests.

The **OSS project level requirements** are shown in Table[1] and in the sequel they are discussed individually in order to validate the expressive power of the OSSMETER metamodels. In particular, the aim of the discussion is to check if the elicited requirements can be satisfied by means of the information provided by the metamodels (in this case the requirement is marked with the symbol ✓). The symbol ⊘ is used to mark requirements that cannot be directly satisfied by the information available in the project metadata and consequently by those in the metamodels. However, there is the possibility to define some procedures able to derive from the available data the information required to satisfy the requirement. The symbol ✗ is used to identify requirements that are not satisfied by the initial version of the metamodels, but will be fixed by the revised version of the OSSMETER metamodels presented in the next section.

**C2.1: Identify the components of the OSS project (✗)**  Project metadata that are typically used by software forges do not contain specific attributes devoted to the direct representation of software components, e.g., user interfaces, database managers. However, bug tracking system typically organize issues by classifying them by software components. The OSSMETER metamodels specify the attributes, which are necessary to have access to the bug tracking system, consequently the identification of software components of a given OSS project is feasible. However, the metamodels as in D2.1 have to be refined in order to give the possibility to represent and organize the components of OSS projects.

**C2.2: Identify the external "libraries" of the OSS project (✗)**  Like in the previous case, OSS forges do not directly represent and manage external "libraries". However, it is possible to identify them by analysing the implementation code. For instance, in case of Eclipse plug-ins it is possible to identify their dependencies and the required external libraries by looking at the plugin.xml file
## Table 1: Use Cases: OSS project level requirements (as in D1.1 [5])

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Overall Priority</th>
<th>Construction Priority</th>
<th>Eclipse Priority</th>
<th>IT Services Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identify the components of the OSS project</td>
<td>SHALL</td>
<td>SHALL</td>
<td>SHALL</td>
<td>SHOULD</td>
</tr>
<tr>
<td>Identify the external &quot;libraries&quot; of the OSS project</td>
<td>SHALL</td>
<td>SHALL</td>
<td>SHALL</td>
<td>SHALL</td>
</tr>
<tr>
<td>Provide the ability to monitor only the activity of selected OSS project components</td>
<td>SHALL</td>
<td>SHALL</td>
<td>MAY</td>
<td>SHALL</td>
</tr>
<tr>
<td>Provide an indicator of the age of the OSS project components</td>
<td>SHOULD</td>
<td>MAY</td>
<td>SHOULD</td>
<td>SHOULD</td>
</tr>
<tr>
<td>Provide a count of the number of downloads of the OSS project</td>
<td>MAY</td>
<td>MAY</td>
<td>MAY</td>
<td>MAY</td>
</tr>
<tr>
<td>Identify the criticality of the OSS project</td>
<td>SHALL</td>
<td>SHALL</td>
<td>SHALL</td>
<td>SHALL</td>
</tr>
<tr>
<td>Provide a measure of the contributors' commitment to OSS</td>
<td>SHALL</td>
<td>SHOULD</td>
<td>SHALL</td>
<td>SHOULD</td>
</tr>
<tr>
<td>Provide an overall indicator of the level of OSS project activity</td>
<td>SHALL</td>
<td>SHALL</td>
<td>SHALL</td>
<td>SHALL</td>
</tr>
<tr>
<td>Provide an indicator of the estimated OSS project effort</td>
<td>SHOULD</td>
<td>MAY</td>
<td>SHOULD</td>
<td>SHOULD</td>
</tr>
<tr>
<td>Provide an indicator of the number of installations</td>
<td>SHALL</td>
<td>MAY</td>
<td>SHALL</td>
<td>SHOULD</td>
</tr>
<tr>
<td>Identify the main contributor/sponsor of the OSS project</td>
<td>SHALL</td>
<td>MAY</td>
<td>SHALL</td>
<td>SHALL</td>
</tr>
<tr>
<td>Provide the capability to set up an alert concerning specific values of a metric for an OSS project</td>
<td>SHALL</td>
<td>MAY</td>
<td>MAY</td>
<td>SHALL</td>
</tr>
<tr>
<td>Provide the capability to be notified in case thresholds are reached for a metric where an alert has been set up for an OSS project</td>
<td>SHALL</td>
<td>MAY</td>
<td>MAY</td>
<td>SHALL</td>
</tr>
<tr>
<td>Provide a count of the number of active contributors to the OSS project</td>
<td>SHALL</td>
<td>MAY</td>
<td>SHALL</td>
<td>SHALL</td>
</tr>
<tr>
<td>Provide a count of the number of releases of the OSS project</td>
<td>SHALL</td>
<td>MAY</td>
<td>SHALL</td>
<td>SHALL</td>
</tr>
<tr>
<td>Provide a count of the number of active committers to the OSS project</td>
<td>SHALL</td>
<td>MAY</td>
<td>SHALL</td>
<td>SHALL</td>
</tr>
</tbody>
</table>
defining all the properties of the considered plug-in including its dependencies. The OSSMETER metamodels specify the attributes related to the data, which are required to have access to the version control system used by a given project to maintain the corresponding software code. Thus, the requirement of identifying external "libraries" of the OSS project can be satisfied once C2.1 is addressed.

**C2.3: Provide the ability to monitor only the activity of selected OSS project components (✗)**
As in the case of C2.1 there is not a direct representation of software components even though it is possible to indirectly identify them. Beyond that, the OSSMETER metamodels permit to specify the data required to have access to the bug tracking systems, communication channels, and version control systems, which represent the main sources of information, to evaluate the activity of a given OSS project, in terms of e.g., number of commits per day, average time between new/closed/resolved bugs, and number of questions/replies. Thus, the requirement can be satisfied once C2.1 is addressed.

**C2.4: Provide an indicator of the age of the OSS project components (✗)**
As said in the description of C2.1, it is possible to identify the components of a given OSS project. The metamodel is able to represent when a given project/component has been created. Consequently, it is possible to indirectly satisfy the requirement by relying on additional information that users can provide to specify when a component has been added. Thus, the requirement can be satisfied once C2.1 is addressed.

**C2.5: Provide a count of the number of downloads of the OSS project (✓)**
This requirement can be satisfied for those forges that maintain the number of downloads of the maintained OSS projects. For instance, SourceForge maintains this information, which is also shown to normal users when visiting the Web pages of projects. The defined OSSMETER metamodels corresponding to SourceForge can capture the number of project downloads.

**C2.6: Identify the criticality of the OSS project (✓)**
The OSSMETER metamodels permits to represent all the issues of a given project. In particular, an abstract metaclass is defined to represent a general bug tracking system. Such a metaclass is extended by other different metamodels each related to a particular bug tracking system technology. In any case, each specific metamodel permits to discover the issues affecting a particular OSS project.

**C2.7: Provide a measure of the contributors’ commitment to OSS (✓)**
It indicates that the committer/contributors are developers with some degree of experience in the OSS community. As previously mentioned, the OSSMETER metamodels include the representation of the data required to have access to the version control systems related to a given OSS project. Consequently, starting from such data it is possible to evaluate the commitment of the involved users, which can play different roles, e.g., administrator, developer, manager, etc. In this respect, OSSMETER metamodels permit to specify the different roles played by the users of a given project. Evaluating the contributors’ commitment across projects is possible if contributors use the same name/login, within the forges analysed.
C2.8: **Provide an overall indicator of the level of OSS project activity (√)**  
As said for C2.3 the OSSMETER metamodels permit to specify the data required to have access to the different systems used to manage OSS project, i.e., bug tracking, version control, and communication channel systems. Consequently, models conforming to the OSSMETER metamodels contain the information that can be exploited by specific metric providers devoted to the calculation of indicators presenting the activity level of OSS projects.

C2.9: **Provide an indicator of the estimated OSS project effort (√)**  
This is related to effort estimation similar to what’s provided by COCOMO. In this respect, the OSSMETER metamodels include the specification of version control systems used in a given project. This is relevant to execute metric providers that perform static analysis on software code.

C2.10: **Provide an indicator of the number of installations (∅)**  
In general, forges do not maintain information related to the number of installations of a given project. In same cases, for example in the case of the Eclipse Market Place, such information is available, and consequently it is possible to define a metric provider able to satisfy the requirement.

C2.11: **Identify the main contributor/sponsor of the OSS project (∅)**  
OSSMETER metamodels include specific metaclasses devoted to the representation of users involved with different roles in OSS projects. Moreover, to identify the main sponsors of a given project it is possible to consider the e-mail addresses (also represented in the metamodels) of committers. In some forges, e.g., SourceForge, there is a direct representation of donations that a given project has received.

C2.12: **Provide the capability to set up an alert concerning specific values of a metric for an OSS project (√)**  
This is related to a feature of the software platform developed in WP5. However, such a requirement is satisfied as long as the considered metrics rely on data that are directly represented by the OSSMETER metamodels or that can be derived from them.

C2.13: **Provide the capability to be notified in case thresholds are reached for a metric where an alert has been set up for an OSS project (√)**  
This is a feature of the software platform of WP5. However, the requirement is satisfied in cases like those in C2.12.

C2.14: **Provide a count of the number of active contributors to the OSS project (√)**  
The users of a given software forge can play different roles and can contribute in different manners e.g., as developer, owner, etc. The OSSMETER metamodels contain the structural features required to represent the users of forges together with their roles. Metric providers devoted to the identification of active contributors can rely on such structural features.

C2.15: **Provide a count of the number of releases of the OSS project (✓)** Some of the relevant forges directly represent releases of a given OSS project (e.g., in the case of Redmine forges). For those cases, the OSSMETER metamodels contain the structural features used to represent them. If such information is not available in the considered forge, it might be possible to deduce information about OSS project releases by looking at the comments of the software commits managed by the used version control system.

C2.16: **Provide a count of the number of active committers to the OSS project (✓)** This is related to C2.7 and the OSSMETER metamodels permit to specify the information required by metric providers devoted to the calculation of active committers of a given OSS project.

### 2.2 Validation of the OSSMETER metamodels with respect to WP2 requirements

As previously done, in this section we use the symbol ✓ to identify WP2 requirements (shown in Table 2) that are already satisfied by the OSSMETER metamodels in D2.1. The symbol × is used to identify WP2 requirements that are not satisfied by the initial version of the metamodels, but will be fixed by the revised version of the OSSMETER metamodels presented in the next section. Additionally, we use the symbol ○ for requirements that have not been addressed yet, and will be addressed in the subsequent WP2 tasks.

**R1 (✓)** The metamodels in D2.1 have been defined by analyzing the GitHub, SourceForge, Eclipse, and Google Code open-source forges. Four metamodels have been defined each representing the project metadata managed by the corresponding forge.

**R2 (✓)** The commonalities of the analysed forges have been represented in a common metamodel, which is extended by four additional metamodels.

**R3 (×)** The metamodels in D2.1 contain both static and dynamic attributes, i.e., attributes whose values do not change over time, and those that are continuously updated (see Section 3 for a better explanation). Thus, a revision of the metamodels is required in order to address the requirement.

**R4 (✓)** By using the approach discussed in D5.1 [3], and D5.2 [4], from the specified OSSMETER metamodels, a number of Java classes are automatically generated to manage the persistence of OSSMETER models in a MongoDB database. An illustrative example is presented at the end of D2.1.

**R5 (×)** Even though the metamodels have been defined trying to focus only on relevant concepts and attributes, there are still structural features that probably are not necessary for analysis and comparison purposes.
<table>
<thead>
<tr>
<th>ID</th>
<th>Requirement</th>
<th>Priority</th>
<th>Originating WP</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>The forge metamodels shall be defined to support the representation of all the metadata that are used by GitHub, SourceForge, Eclipse, and Google Code to manage projects.</td>
<td>SHALL</td>
<td>WP2</td>
</tr>
<tr>
<td>R2</td>
<td>The forge metamodels shall be defined by distinguishing aspects that are forge independent from those that are forge specific.</td>
<td>SHALL</td>
<td>WP2</td>
</tr>
<tr>
<td>R3</td>
<td>The forge metamodels shall contain only structured aspects of OSS projects. Attributes that can change over time shall be calculated by metric providers.</td>
<td>SHALL</td>
<td>WP2</td>
</tr>
<tr>
<td>R4</td>
<td>The forge metamodels shall be used to define the project data managed by the OSSMETER platform.</td>
<td>SHALL</td>
<td>WP2</td>
</tr>
<tr>
<td>R5</td>
<td>The forge metamodels shall represent only aspects that are relevant for project analysis and comparison purposes. This is necessary to limit the size of the developed metamodels and consequently that of the models conforming to them.</td>
<td>SHALL</td>
<td>WP2</td>
</tr>
<tr>
<td>R6</td>
<td>The concepts represented in the metamodels shall be as concise as possible by deriving as much information as possible. Attributes whose values can be derived by considering the values of other structural features of the models might be not represented in the metamodels.</td>
<td>SHALL</td>
<td>WP2</td>
</tr>
<tr>
<td>R7</td>
<td>Metric metamodels shall be defined by properly organizing the different metamodels supporting the metric providers that will be developed by all the technical workpackages.</td>
<td>SHALL</td>
<td>WP2</td>
</tr>
<tr>
<td>R8</td>
<td>Metric metamodels shall be used to defined the quality and activity metrics that the OSSMETER platform is able to calculate</td>
<td>SHALL</td>
<td>WP2</td>
</tr>
<tr>
<td>R9</td>
<td>A set of OSS projects to be used for validating both the forge and metric metamodels shall be defined</td>
<td>SHALL</td>
<td>WP2</td>
</tr>
<tr>
<td>R10</td>
<td>Metamodels shall be extensible in the sense that concepts shall be added in case of unforeseen requirements that are identified by the other technical workpackages and Use Case providers.</td>
<td>SHALL</td>
<td>WP2</td>
</tr>
<tr>
<td>R11</td>
<td>The forge metamodels shall support the metrics models defined for the code quality and activity analysis.</td>
<td>SHALL</td>
<td>WP3</td>
</tr>
<tr>
<td>R12</td>
<td>The forge metamodels shall provide the information that is needed by metrics providers.</td>
<td>SHALL</td>
<td>WP3</td>
</tr>
<tr>
<td>R13</td>
<td>The OSSMETER model shall support the metrics built upon the outcomes of the text mining workflow.</td>
<td>SHALL</td>
<td>WP4</td>
</tr>
<tr>
<td>R14</td>
<td>Additional forge metamodels may be defined to support the representation of projects managed by additional forges.</td>
<td>MAY</td>
<td>WP2</td>
</tr>
</tbody>
</table>

Table 2: WP2 requirements(as in D1.1 [5])
R6 (X) There are many structural features in the current version of the metamodels that can be dropped to address this requirement. For instance, in the case of GitHub many URLs can be automatically derived from the name of the repository. Currently, the metamodel represent both the name and the URLs of GitHub projects.

R7 (∗) The final definition of metric metamodels and their proper organization is part of the next tasks of WP2 and is the main subject of D2.4.

R8 (∗) This is also part of the last deliverable of WP2, even though the work that has been already done in other WPs concerning the definition of metric providers, partially satisfies the requirement.

R9 (√) In D2.1 a set of representative OSS projects has been defined in order to validate the expressiveness of the forge and metric metamodels.

R10 (√) OSSMETER metamodels have been defined in a modular way in order to distinguish between forge commonalities and forge-specific concepts. Additional refinements have to be done in order to increase the modularity of the metamodels. For instance, the SVN repository concept is replicated in all the forges that make use of it. It might make sense to pull the SVN repository concept out from the forge metamodels and create a specific metamodel for it.

R11 (√) Metric providers devoted to analyzing the quality of software code and the activity of a given project mainly require the information necessary to have access to the version control systems. In this respect, forge metamodels include such information thus the requirement is already satisfied.

R12 (√) This is part of an iterative process, depending on the information required by metric providers, forge metamodels might be extended. In the current version, the information available in the OSSMETER metamodels is enough to support the execution of the metric providers already implemented.

R13 (√) This is also part of an iterative process as previously discussed in R12. The metamodels as defined in D2.1 already support the execution of text mining activities.

R14 (∗) Depending on additional requirements and needs, the forge metamodels will be extended in order to include the support for further forges.

2.3 Summary of unmet requirements

By considering the discussion above the requirements that are not addressed by the OSSMETER metamodels presented in D2.1 are show in Table3. R7 and R8 will be addressed in the next deliver-
D2.2 – Metamodels for Describing OSS projects

<table>
<thead>
<tr>
<th>ID</th>
<th>Requirement</th>
<th>Addressed in Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2.1</td>
<td>Identify the components of the OSS project.</td>
<td>Yes</td>
</tr>
<tr>
<td>C2.2</td>
<td>Identify the external &quot;libraries&quot; of the OSS project.</td>
<td>Yes</td>
</tr>
<tr>
<td>C2.3</td>
<td>Provide the ability to monitor only the activity of selected OSS project components.</td>
<td>Yes</td>
</tr>
<tr>
<td>C2.4</td>
<td>Provide an indicator of the age of the OSS project components.</td>
<td>Yes</td>
</tr>
<tr>
<td>R3</td>
<td>The forge metamodels shall contain only structured aspects of OSS projects. Attributes that can change over time shall be calculated by metric providers.</td>
<td>Yes</td>
</tr>
<tr>
<td>R5</td>
<td>The forge metamodels shall represent only aspects that are relevant for project analysis and comparison purposes. This is necessary to limit the size of the developed metamodels and consequently that of the models con-forming to them.</td>
<td>Yes</td>
</tr>
<tr>
<td>R6</td>
<td>The concepts represented in the forge and metric meta-models shall be minimalistic. For instance, concept attributes whose values can be derived by considering the values of other structural features of the models shall not be represented in the metamodels.</td>
<td>Yes</td>
</tr>
<tr>
<td>R7</td>
<td>Metric metamodels shall be defined by properly organizing the different metamodels supporting the metric providers that will be developed by all the technical work-packages.</td>
<td>No</td>
</tr>
<tr>
<td>R8</td>
<td>Metric metamodels shall be used to defined the quality and activity metrics that the OSSMETER platform is able to calculate.</td>
<td>No</td>
</tr>
<tr>
<td>R14</td>
<td>Additional forge metamodels may be defined to support the representation of projects managed by additional forges.</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 3: Summary of the requirements that are not addressed by the metamodels in D2.1

ables of WP2, especially in D2.4. The other requirements are addressed by means of the metamodels refinements presented in Section 3.

2.4 Feedback from OSSMETER partners

Beyond the Use Case and WP2 requirements discussed in the previous sections, a short questionnaire has been defined to gather further feedback from the partners of the project. The distributed questionnaire consisted of the following questions:

Q1: Do you think there are missing concepts in the current OSSMETER metamodels?
Q2: Why the missing concepts you have identified should be added in the OSSMETER metamodels? Please mention the parts of the case study or metrics providers that would be affected if we do not extend the OSSMETER metamodels.
Q3: Have you spotted concepts, which are not properly represented in the OSSMETER metamodels? How would you better represent them? Why?
D2.2 – Metamodels for Describing OSS projects

For each metamodel, the respondents to the questionnaire had available i) a graphical representation of the metamodels, ii) a textual specification of them, and iii) the deliverable D2.1.

The suggestions that have been collected by means of the previous questions can be summarized as follows:

1. Extend the OSSMETER metamodels by including the Redmine forge since it is used to manage the Modelio project;

2. Introduce separate metamodels for different version control system/communication channel/bug reporting system technologies;

3. Introduce a forge-independent mechanism able to represent the different roles played by the persons involved in a given project;

4. Concerning the Eclipse forge metamodel:
   (a) Introduce the concept of parent project: some projects like the top-level Eclipse project or the Eclipse Platform have sub-projects.
   (b) Introduce the concept of available platform to specify the available platforms where a given project can be deployed (e.g., Ganymede, Galileo, Helios, Indigo, Juno, Kepler, Luna, etc.);
   (c) Introduce the concept of life-cycle to represent the status of the project that can be: pre-proposal, proposal, incubation, mature, top-level or archived;

5. Concerning the GitHub metamodel some information like clone_url, git_url, ssh_url can be inferred from the repository and the user names;

6. Concerning the GoogleCode metamodel it has to be fixed in order to represent that one GoogleCodeProject is related with one VCSRepository only. Currently, it is possible to represent many version control systems used at the same time;

7. Concerning the SourceForge metamodel it has to be fixed in order to enable the representation of Git repositories also for projects managed by SourceForge;

8. Minor issues concerning the cardinalities of associations and compositions have to be fixed.

All the suggestions given by the OSSMETER partners have been implemented as shown in Table 4.

---

http://projects.eclipse.org/projects/eclipse
http://projects.eclipse.org/projects/eclipse.platform
<table>
<thead>
<tr>
<th>Suggestion</th>
<th>Sections implementing the suggestion</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sec. 3.10</td>
</tr>
<tr>
<td>2</td>
<td>Sec. 3.1, 3.2, 3.3, 3.4, 3.5</td>
</tr>
<tr>
<td>3</td>
<td>Sec. 3.1</td>
</tr>
<tr>
<td>4.a</td>
<td>Sec. 3.1, 3.7</td>
</tr>
<tr>
<td>4.b</td>
<td>Sec. 3.7</td>
</tr>
<tr>
<td>4.c</td>
<td>Sec. 3.7</td>
</tr>
<tr>
<td>5</td>
<td>Sec 3.6</td>
</tr>
<tr>
<td>6</td>
<td>Sec. 3.8</td>
</tr>
<tr>
<td>7</td>
<td>Sec. 3.9</td>
</tr>
<tr>
<td>8</td>
<td>Sec. 3</td>
</tr>
</tbody>
</table>

Table 4: Implementation of the suggestions given by the OSSMETER partners
3 Revised metamodels

In this section we present the OSSMETER metamodels, which have been refined to fix the issues identified during the validation activities as described in the previous section. The metamodels have been restructured as shown in Figure 3. In particular, the Base Forge metamodel represents all the main concepts, which are typically involved when managing software forges. The concepts represented in such a common metamodel are extended by other metamodels, in order to represent specificities of particular forges (i.e., GitHub, SourceForge, GoogleCode, Eclipse, and Redmine), of different version control systems, communication channels, and bug reporting technologies.

During the refinement process we identified different kinds of information that will be detailed in the next sections, and that can be classified as:

- **Static**: it refers to attributes, e.g., project name, and urls of repository servers, that do not change over time, and that have to be necessarily represented in order to have access to resources of the considered projects and analyse them;
- **Dynamic**: it refers to attributes whose value changes over time, e.g., the size of a given project, or the persons involved in it with some roles. Such attributes can be represented in the metamodel, even though their value has to be lazily updated;
- **Derived**: it refers to project properties that can be derived from static ones, thus storing them my not be necessary. For instance, given a GitHub repository rep of a given user, the corresponding url to download the whole archive of the repository is https://github.com/
user/rep/archive/master.zip;
- **Service**: it refers to attributes shared by different metrics providers. Thus instead of replicating them in the data model of different metric providers, they are represented once in the corresponding forge metamodels. For instance, if many metric providers require the number of downloads of a given project as starting point for their analysis, it might make sense to add an attribute downloads in the metaclass devoted to the representation of software projects.
The identification of Service attributes is not straightforward and depends on the available metric providers and on the information they require. As shown in Figure 4, there are at least two ways for supporting the execution of many metric providers requiring the value of the same project attributes. A trivial way is that each metric provider tries to retrieve the value of the required value directly from the considered software forge (see Figure 4a). This is not always possible especially when the considered forge limits the number of APIs requests. For instance, the rate limit for the GitHub APIs is 5,000 requests per hour for authenticated users, 60 requests per hours for unauthenticated users. The adoption of service attributes can overcome such difficulties as shown in Figure 4b. In particular, metric providers retrieve the value of the required attribute from the OSSMETER platform, which in turn lazily updates its value.

As previously discussed the metamodels are part of an iterative process aiming at refining the existing concepts and adding new ones according to the needs of the other technical work packages: if some metrics provider or source code analysis requires concepts, which are currently missing in the metamodel, then they will be properly added in the metamodel and a new version will be released.
In this respect, the exact identification of service attributes will be finalized once a representative set of metrics providers will be identified and developed.

In the remaining of the section all the metamodels shown in Figure 3 are discussed in detail.

### 3.1 Base Forge metamodel

There are a number of concepts that, regardless of the considered forge, are always involved in the management of OSS projects. In particular, as show in the metamodel in Figure 5 open source projects (see the metaclass `Project`) are typically associated with one or more version control systems (VcsRepository), a number of communication channels, like forums, and wikis (CommunicationChannel), and bug tracking systems (BugTrackingSystem). Importantly, each project has also a number of licenses (License) that specify the terms and conditions under which the source code of the considered project can be used, modified and/or shared. Finally, a number of persons with different responsibilities and roles (e.g., developer, owner, mentor, and leader) are involved in a given open source project (see the metaclasses Person and Role).

In addition to the concepts already captured in the base forge metamodel presented in D2.1 [2], Figure 5 shows additional information related to the management of metrics provider. In particular, each project can be analysed by different MetricProviders each identified by a corresponding
id. The last execution of a given metric provider is also stored (see the attribute `lastExecuted`). Moreover, as presented in D5.2 [4], metric providers can be historic or transient (see the enumeration `MetricProviderType`).

It is important to remark that, by considering the classification given above, all the structural features of the base forge metamodel are static.

### 3.2 Subversion metamodel

![Figure 6: Subversion metamodel](image)

The Subversion metamodel represents the minimal information required to obtain access to a given SVN repository, i.e., the `username`, `password`, and the inherited `url` of `VcsRepository` representing the address of the svn server. The `browse` attribute represents the url of the Web page, through which users can browse the content of the considered svn repository. All the structural features of the Subversion metamodel are static.

### 3.3 Git metamodel

![Figure 7: Git metamodel](image)

Similarly to the previous section, Figure 7 represents the minimal information required to obtain access to a given git repository, i.e., the `username`, `password`, and the inherited `url` of `VcsRepository` representing the address of the git server.
3.4 NNTP metamodel

Among the communication channels that are typically used by software forges, newsgroups play a key role since they permit to share relevant information about the projects of interest. The Network News Transfer Protocol (NNTP) is the application protocol that is typically used for transporting news articles between news servers and client applications. Figure 8 shows the static information required to obtain access to a given news server. In particular, beyond the url attribute inherited by the CommunicationChannel metaclass, the username and password are also represented. These attributes are necessary when the considered news server requires user authentication. The attributes name and description represent the name and description, respectively as it appears on the news server.

![Figure 8: NNTP metamodel](https://example.com/image)

3.5 Bugzilla metamodel

Bugzilla is among the most representative bug tracking systems currently used by software forges. The main concepts involved in the management of bugs by means of Bugzilla are products and components. Products are the broadest category in Bugzilla, and tend to represent real-world products, whereas components are subsections of a product. Figure 9 shows the static information required to have access to a Bugzilla server, i.e., the inherited url attribute from the abstract BugTrackingSystem metaclass, username, and password. Queries on the considered Bugzilla server can be done by considering specific products and components of the project under investigation, and by means of the CGI program specified by the attribute cgiQueryProgram.

3.6 GitHub metamodel

Figure 10 formalizes the concepts which have been identified during the analysis of GitHub forge[4] and subsequently refined after the delivery of D2.1. In particular, we identified many attributes whose values are derived from those of static ones. In the following, all the metaclasses of the

GitHub metamodel are described and the distinction of static, dynamic, and derived attributes is also presented.

**GitHubRepository**  In GitHub each project is defined by means of a number of static and derived attributes as described in the following.

**Static attributes:**
- *private*, it is a boolean representing if a repository is private or not. Creating private repositories requires a paid GitHub account;
- *homepage*, it is the root URL of the considered GitHub forge (e.g., https://github.com/);
- *languages*, it specifies the programming languages used to implement the considered project;
- *master_branch*, a typical aspect which is common in version control systems is that of branches, i.e., developers can create new branches of the project in order to add hotfix, new functionalities, etc. Once the branches code has been properly tested, it can be merged with the production branch (referred with the *master_branch* attribute);

**Dynamic attributes:**
- *fork*, on GitHub it is possible to create repository forks, i.e., developers that want to contribute on a project can easily create a "copy" of it that can be modified and extended independently. Thus, if a repository is a fork of another one then the attribute *fork* is true. To maintain all the forks of a given repository, the reference *forks* is used;
- *size*, it represents the size in Kb of the repository;
- *downloads*, it refers to the artefacts that users can download (see the metaclass *GitHubDownload* below);
- *milestones*, it refers to the milestones defined by the owner of the repository (see the metaclass *GitHubMilestone* below)

People involved in a given project can play different roles that can be summarized as follows:
- *collaborators*, they are used to grant access to a repository owned by a personal account. This enables a collaborative, even though properly regulated, project development;
Figure 10: GitHub metamodel
D2.2 – Metamodels for Describing OSS projects

– **stargazers**, similarly to social networks, registered GitHub users may specify when they like a repository. In other words, starring is basically a way to bookmark interesting repositories. Given a GitHub repository, **stargazers** permits to refer all the users that like it;
– **watchers**, it is used to refer all the GitHub users that want to receive email or web notifications about a repository;
– **owner**, it refers to the owner of the considered repository;

The representation of such roles is performed by exploiting the metaclasses **Person** and **Role** of the base metamodel shown in Figure 5.

**Derived attributes:**

– **full_name**, it is a string representing the full name of the considered repository. In particular, given a repository named **grit**, owned by the user **mojombo**, then the value of the attribute **full_name** is **mojombo/grit**;
– **html_url**, it is the URL pointing to the home page of the considered repository. It is derived by concatenating the values of the **homepage** and **full_name** attributes (e.g., https://github.com/mojombo/grit);
– **clone_url**, it is the URL that developers can use to create a local clone of the repository. In this way, the content of the considered repository is locally downloaded and developers can start working with it. Depending on the remote server, different transport protocols can be used to clone repositories. Git supports ssh, git, http, and https protocols, and consequently different URLs are given, i.e., ssh_url, git_url, and svn_url. Such urls are all derived. For instance, given a repository with the **full_name** as **octocat/Hello-World**, the following urls are derived:
  – clone_url: https://github.com/octocat/Hello-World.git
  – git_url: git://github.com/octocat/Hello-World.git
  – ssh_url: git@github.com:octocat/Hello-World.git
  – svn_url: https://svn.github.com/octocat/Hello-World
– **mirror_url**, it is the URL of the repository mirror. In particular, a repository can have a mirror, which is a complete copy of the project that can be used for different purposes, mainly for backup. Given a repository with the **full_name** as **octocat/Hello-World**, and **git.example.com** the url of the server for the mirror, the corresponding **mirror_url** is git://git.example.com/octocat/Hello-World.

**GitHubMilestone**  As typically done in software engineering and project management, a number of milestones are defined by the owner of a repository in order to specify the most important events or functionalities to be achieved during the project. To represent milestones of projects maintained in GitHub the following structural features are given:

**Static attributes:**

– **number**, it is an integer used to identify a milestone;
– **title** and **description** contain the title and the description, respectively, provided by the creator of the milestone.
– *created_at* and *due_on* represents when the milestone has been created and when it is due, respectively;

**Dynamic attributes:**

– *state*, it represents if the considered milestone has been achieved or not;
– *closed_issues* and *open_issues* refers to the closed and open issues, respectively (see the metaclass GitHubIssue below).

**GitHubContent**  A GitHubRepository contains a number of files and directories, which are represented by means of the metaclass GitHubContent consisting of the following attributes:

**Static attributes:**

– *name*, it is a string representing the name of the represented content;
– *type*, it specifies if the considered content is a file or a directory;
– *path*, it is the absolute path of the content with respect to the root directory of the considered repository;

**Dynamic attributes:**

– *size*, it is an integer representing the size in Kb of the represented content;
– *sha*, it is an identifier of the considered content;

**GitHubDownload**  It represents some repository elements, which can be directly downloaded. The main attributes of such a metaclass are the following:

**Static attributes:**

– *id*, it is an internal identifier of the download element;
– *name*, it is the name of the represented download element;
– *description*, it is a description of the provided download element;
– *content_type*, it represents the type of the download element (e.g., jpg, txt, png, etc.).

**Dynamic attributes:**

– *size*, it represents the size of the considered download element;
– *download_count*, it represents how many times the considered element has been downloaded;

**Derived attributes:**

– *html_url*, it is the URL of the represented download element. It is an attribute whose value is derived from the *full_name* and *homepage* attributes of the corresponding repository and, the *name* attribute of the considered download element. For instance, given the repository having [https://github.com](https://github.com) as *homepage*, mojombo/grit as *full_name*, and the download element named grit-1.0.1.gem, the corresponding *html_url* is [https://github.com/downloads/mojombo/grit/grit-1.0.1.gem](https://github.com/downloads/mojombo/grit/grit-1.0.1.gem);
– **url**, it is like the previous one, even though instead of using the name of the download element, it makes use of its internal ID. For instance, https://api.github.com/repos/mojombo/grit/downloads/5 is the value of the **url** attribute of the download element having the **id** attribute with value 5;

**GitHubUser** Modifications on GitHub repositories can be performed only by authenticated users, which are uniquely represented by a **login** and have associated the derived **html_url** linking to the user home page. For instance, given the user **mojombo**, the corresponding **html_url** is https://github.com/mojombo.

Users can follow and/or be followed by other users (see the reference **followers**). Finally, the meta-class **GitHubUser** is the type of different references in the metamodel in Figure[10] meaning that users can play different roles in the GitHub forge.

**GitHubBugTracker** It represents the bug tracking system of GitHub and refers the **issues** of the considered repository.

**GitHubIssue** It represents an issue affecting a given repository and the following structural features are given:

Static attributes:

– **number**, it is an integer used to identify an issue;
– **title** and **body**, contain the title and the description, respectively, provided by the creator of the issue;
– **created_at** and **closed_at** represent when the issue has been created, and closed, respectively;

Dynamic attributes:

– **state**, it represents if the considered issue is still open or not;
– **comments**, it represents the comments that logged users are given to the considered issue.
– **updated_at**, it represents when the issue has been updated;

A logged user can be involved in a given issue with different roles. In particular, he can be i) the **creator**, ii) one of the **assignee** involved in its resolution, iii) mentioned in the issue, or simply iv) **subscribed** since he is interested in the issue and he wants to be updated about its resolution.

### 3.7 Eclipse metamodel

Figure[11] formalizes the concepts which have been identified during the analysis of the Eclipse ecosystem. Unfortunately, Eclipse does not provides well defined APIs as in the case of GitHub, consequently the formalized concepts have been identified by looking at the Web pages of different projects belonging to the Eclipse ecosystem, and other technical papers about the topic. In the following all the metaclasses shown in Figure[11] are described in detail.

[^1]: [http://www.eclipse.org](http://www.eclipse.org)
Figure 11: Eclipse metamodel
EclipseProject It extends the abstract metaclass Project of the metamodel in Figure 5 and a number of structural feature are added as described in the following:

**Static attributes:**

- **shortName**, it is a string used to represent the short name of the considered project;
- **projectplanUrl**, it represents the URL pointing to an XML file containing the project plan in an Eclipse defined standard format;
- **updatesiteUrl**, it contains the URL of the project’s main update site used by the update mechanism provided by Eclipse to simplify the local installation and update of projects;

Moreover, people can be involved in a given Eclipse project with different roles, namely **mentor**, **committer**, and **leader**. In particular, when a new project is proposed it is required to have at least **two mentors**. Mentors must be members of the Architecture Council and are required to monitor and advise the new project during its incubation phase, but are released from that duty once the project graduates to the mature phase.

The roles of the people involved in a given Eclipse project are represented by means of the metaclasses Project and Role of the base metamodel shown in Figure 5.

**Dynamic attributes:**

- **status**, it specifies the status of the project that can be **prepoposal**, **poposal**, **incubation**, **mature**, **toplevel**, and **archived**;
- **platform**, it specifies the platforms where the project can be deployed, e.g., Ganymede, Galileo, Helios, Indigo, and Juno.

**Derived attributes:**

Many attributes of the EclipseProject metaclass are derived from the **shortName** attribute as explained in the following:

- **paragraphUrl**, it is a string containing the URL of a HTML file in the project’s web directory that contains a concise (e.g., 50-100 word) description of the project. For instance, given the modeling project emft (Eclipse Modeling Framework Technology) the corresponding paragraphUrl is http://www.eclipse.org/modeling/emft/project-info/project-page-paragraph.html;
- **descriptionUrl**, it is a string containing the URL of a HTML file in the project’s web directory that contains a short (e.g., 2-3 paragraph) description of the project. For instance, in the case of the modeling project emft, it’s http://www.eclipse.org/modeling/emft/project-info/summary.html;
- **downloadsUrl**, it represents the URL that points to the project’s downloads. For instance, in the case of the modeling project emft, it’s http://www.eclipse.org/modeling/emft/downloads/?project=emft;
- **homePage**, it represents the URL of the project home page (e.g., http://www.eclipse.org/modeling/emft/).
**Release**  It is used to represent a specific release of a given project. Each release is defined by its dynamic attribute *status* (i.e., *completed*, *scheduled*, and *tentative*), and by the static attribute *date* when it has been issued.

**Article**  Typically a number of articles or other interesting links related to the considered project are maintained. In this respect, the metaclass *Article* is provided and it consists of the static attribute *url* pointing to the represented material.

**Review**  All major changes to Eclipse projects must be announced and reviewed. For instance, major changes include the introduction or exclusion of significant new technology or capability. For each review, the project leadership prepares documentation for, and receives feedback from, the Eclipse membership. The metaclass *Review* is used to represents a given project review which is defined by means of the review *type*, *state*, and its *end date*.

For each Eclipse project a number of communication channel are provided as described in the following.

**MailingList**  Multiple mailing lists can be associated to a given project. Each mailing list is represented by means of the static attributes:

- *name*, it represents the name of the newsgroup as it appears on the news server;
- *description*, it contains the optional description that helps visitors to decide which newsgroup has to be visited;
- *type*, generally a project has a single *main* newsgroup even though multiple ones can be available;
- *archiveUrl*, it represents the address that points at the archive of the considered mailing list.

**Wiki and Documentation**  They are used to represent the Wiki and the project’s documentation. The *url* attribute inherited from the *CommunicationChannel* metaclass is used to points to them.

### 3.8 Google Code metamodel

Figure 12 formalizes the concepts which have been identified during the analysis of the Google Code forge. The analysis has been performed by examining the Web pages of different Google Code projects, the Google IssueTracker API[^6] and other relevant technical material. In the following all the metaclasses shown in Figure 12 are described in detail.

**GoogleCodeProject**  It represents a Google Code project and a number of structural features are given as described below:

**Static attributes:**

[^6]: https://code.google.com/p/support/wiki/IssueTrackerAPI
Figure 12: Google Code metamodel
D2.2 – Metamodels for Describing OSS projects

- wiki, forum, concerning communication channels, a Google project has associated a wiki and a forum, which are reachable by following the content of the attribute url inherited from the metaclass CommunicationChannel of the common forge metamodel shown in Figure 5.

**Dynamic attributes:**

- stars, users can specify when they like a project by adding a start to it. The attribute stars contains the total number of stars that a given project currently has;
- downloads, it refers all the files (e.g., packages, bundles, documents) that can be downloaded (see the metaclass GoogleDownload below);

A logged user (represented in the metamodel by means of the metaclass GoogleUser) can be involved in a given project with different roles. In particular, he can be owner, contributor, and committer. The main differences are is that owner has a complete access to all the elements of a project. Contributors can contribute only to the communication channels, whereas committers can modify also the code. Such roles are not directly defined in Figure 12 since they are represented by means of the metaclasses Person and Role of the base metamodel shown in Figure 5.

**GoogleDownload** It represents a file which is available for download. The following structural features are given:

**Static attributes:**

- fileName, it represents the name of the file;
- uploaded_at, updated_at, they represent when the file has been uploaded and when updated, respectively;
- size, it represents the size of the represented file;

**Dynamic attributes:**

- starred, the considered file can be starred if a user likes it;
- downloadCounts, it represents how many time the file has been downloaded;

**GoogleIssueTracker** As previously mentioned, for each project a bug tracking system is provided in order to manage all the issues of a project.

**GoogleIssue** The issues of a given project are represented by means of the following structural features:

**Static attributes:**

- owner, it refers to the user that created the issue;
- created_at, updated_at, they represent when the issue has been created and updated, respectively;
- priority, all the issues have assigned a priority, which can be low, medium, and high;
– type, when creating an issue the user can specify if it is a defect or if its resolution represents an enhancement of the project;
– component, it refers to the particular component in the project which is affected by the issue;
– summary, labels, they are used to describe the issue and properly tag it;

Dynamic attributes:
– status, it represents the status of the issue, which can be new, started, accepted, reviewed, and acknowledged;
– stars, the users that consider the issue of relevant importance can add a start to it. The attribute stars contains the total number of stars that the considered issue currently has;
– comments, users can comment an issue with the aim of simplifying its resolution;

GoogleIssueComment The comments for a given issue are represented by means of the creation date and by a text containing the body of the comment.

GoogleLabel Issues and downloads can be tagged by means of user-defined labels, which can be used for organization and search purposes.

3.9 SourceForge metamodel

Figure [13] formalizes the concepts which have been identified during the analysis of SourceForge. The analysis has been performed by examining the Web pages of different SourceForge projects, the SourceForge API[7] and other relevant technical material. In the following all the metaclasses shown in Figure [13] are described in detail.

SourceForgeProject It represents a given SourceForge project and a number of structural features are given as described below:
Static attributes:
– created, it represents the creation date of the project;
– projectId, it is an integer used to represent the identifier of the project;
– private, it is a boolean attribute used to specify if a project is private or not. The source code of private projects can be viewed only by the owner and by invited people. Moreover, the source code is not be exposed to search engines or other people without the proper viewing rights;
– shortDesc, it contains a short description of the project as shown in the project summary page;
– programmingLanguages, it represents all the programming languages, which have been used to develop the project code;
– os, it represents the operating systems, which are supported by the project;
– environments, it represents all the running environments of the project (e.g., Eclipse, or if it Web based);

Figure 13: SourceForge metamodel
– *topics, audiences, categories*, projects can be tagged for better organizing them and for searching purposes. In this respect, the structural features *topics, audiences and categories* refers to all the tags, which have been associated to the project;

– *donation*, it refers to information about the donation status of the project (see the metaclass *Donation* below);

**Dynamic attributes:**

– *percentile, ranking*, they are measures of the project activity. The higher the values of such attributes, the higher the project activity is;

– *featureRequests*, it refers to the feature requests done by users (see the metaclass *FeatureRequest* below);

– *supportRequests*, it refers to the support requests done by users (see the metaclass *SupportRequest* below);

– *discussions*, it refer to the occurring discussions about relevant aspects of the project;

People can be involved in a given project by playing the following roles:

– *developers*, it represents project developers, which are represented by means of the *name*, and *homePage* (see the metaclass *Person* in Figure 5);

– *maintainers*, it represents the maintainers of the project;

Such roles are represented by means of the metaclass *Role* of the common metamodel shown in Figure 5.

**Derived attributes:**

– *summaryPage*, it represents the URL of the Web page showing a summary about the project, i.e., a short description, information about its activity, links to the download page, user ratings, etc. For instance, given the project named *texstudio*, the corresponding *summaryPage* is *http://sourceforge.net/projects/texstudio/*;

– *homePage*, it represents the URL of the home page of the project. Sometimes it is that in *summaryPage*;

– *downloadPage*, it represents the URL of the Web page showing the files and packages that can be downloaded. It is derived from the *name* of the project. For instance, given the project named *texstudio*, the corresponding *downloadPage* is *http://sourceforge.net/projects/texstudio/files/*;

– *mailingList*, it represents the URL of the web page showing the link to the mailing list of the project. For instance, given the project named *texstudio*, the corresponding *mailingList* is *http://sourceforge.net/p/texstudio/mailman/*;

– *supportPage*, it represents the URL of the Web page showing some other places where users can look for information about the project. For instance, given the project named *texstudio*, the corresponding *supportPage* is *http://sourceforge.net/projects/texstudio/support.*
Donation  It represents the donation status of the project as described by means of the following attributes:

- `status`, it specifies if the project is accepting or not donations;
- `comment`, in case the project is accepting donations, a comment is shown in the corresponding donation page;
- `charities`, it refers to already received charities;

Discussion  It represents a discussion about some relevant topic related to the project. A discussion is represented by the URL of the Web page where the discussion is occurring.

Tracker  Trackers on SourceForge are used to track bug reports, requests for new features or proposed patches as detailed below.

Request  It represents different kinds of request that users can do about the project. Requests are represented by means of the following structural features:

**Static attributes:**

- `summary`, it represents a short description about the request;
- `creator`, it represents the user that has done the request;
- `created_at`, it represents the creation date of the request.

**Dynamic attributes:**

- `status`, it represents the status of request, i.e., if it is still open, or if it has been closed;
- `updated_at`, it represents the update date of the request.

FeatureRequest  It represents a request that a given user has done since he is interested in a new feature, which is currently missing in the project. Other users that are interested in the same feature can express their vote. The dynamic attribute `votes` maintain the total number of users that would like to have the requested feature added in the project.

SupportRequest  It represents a request of support related to some issues that users have experienced during the usage of the project;

Patch  It represents a patch proposed by a given user about some bugs of the project.
**Bug** It represents a single bug maintained by the bug tracking system of SourceForge. In particular, the following structural features are given:

*Static attributes:*

- `description`, it represents a description about the bug;
- `assignee`, it refers to the user that has to solve the bug;
- `submitted`, it refers to the user that has identified the bug and has created a new entry in the system;
- `priority`, it represents the priority thus the importance of the bug;
- `bugVisibility`, it specifies if the bug is public or not;

*Dynamic attributes:*

- `status`, it represents the status of the bug, thus if it is still open or if it has been closed;
- `resolutionStatus`, it specifies if the bug has been fixed or not;

### 3.10 Redmine metamodel

Figure 14 formalizes the concepts which have been identified during the analysis of Redmine forges. The analysis has been performed by looking at the Web page of the Redmine project\(^8\), its APIs\(^9\), and other relevant technical material. In the following all the metaclasses shown in Figure 14 are described in detail.

**RedmineProject** It extends the abstract metaclass `Project` of the metamodel in Figure 5 and a number of structural features are added as described in the following:

*Static attributes:*

- `identifier`, it is a string that identifies the project;
- `description`, it is a string containing a short description of the project;
- `created_on`, it represents when the project has been created;
- `wiki`, it refers to the home of the wiki pages associated to the project.
- `queryManager`, it refers to the custom queries visible by the user (public and private queries) for the project;
- `issueTracker`, it refers to the bug tracking system used to manage the project’s bugs;

*Dynamic attributes:*

- `updated_on`, it represents when the project information has been updated;
- `members`, it refers to the project’s members that can have different roles has specified below (see the metaclass `RedmineUser`);
- `versions`, it refers all the versions available for the project;

\(^{8}\) [http://www.redmine.org/](http://www.redmine.org/)
\(^{9}\) [http://www.redmine.org/projects/redmine/wiki/Rest_api](http://www.redmine.org/projects/redmine/wiki/Rest_api)
Figure 14: Redmine metamodel
RedmineWiki  It is used to represent the Wiki related to the project. The url attribute inherited from the CommunicationChannel metaclass is used to point to it.

RedmineQueryManager and RedmineQuery They are related to the management of queries that users can submit.

RedmineProjectVersion  It represents a particular version of the project and it is defined by means of the following attributes:

Static attributes:
   - name, it is a string used to identify the version;
   - description, it is a short description of the version;
   - created_on, it specifies when the version has been created.

Dynamic attributes:
   - updated_on, it specifies when the version has been updated;
   - status, it gives additional information about the stage of the considered version, which can be open, locked, or closed.

RedmineUser  It extends the abstract metaclass Person of the base metamodel shown in Figure and it is defined by means of a login attribute and a number of roles played by the user. People can play different roles, e.g., administrator, manager, contributor, and developer. The representation of such roles is performed by means of the metaclass Role of the base metamodel shown in Figure.

RedmineBugIssueTracker  It represents the bug tracking system of Redmine and refers the issues of the considered project.

RedmineIssue  It represents an issue affecting a given project and it is defined in terms of the following attributes:

Static attributes:
   - category, it refers to the element representing the category of the issue, e.g., UI, and database;
   - description, it is a string containing the description of the issue;
   - feature, it refers to the element representing the feature of the considered project that is affected by the issue;
   - priority, it is possible to specify the priority of the issues in order to deduce the order of their resolution;
   - author, it refers to the Redmine user that identified the issue and described it;
   - template, each issue has a corresponding template that can be defectReportFromUser, defectReportFromDeveloper, and reviewRequest;
   - relations, it is possible to specify relations between existing issues. There are different types of issue relations as defined by the enumeration RedmineIssueRelationType;
– *start_date*, and *due_date*, they are strings representing when the issue has been identified, and when it has to be solved, respectively.

**Dynamic attributes:**

– *status*, it represents the current status of the issue, which can be at least *new* or *closed*;
– *update_date* it is a string representing when the issue has been updated.
4 Conclusions

In this document we have presented the process we have followed to validate the metamodels presented in D2.1 and subsequently, the revision that has been performed to address the issues that have been identified. The metamodels have been implemented as EMF/Ecore [1] models. The metamodels are already integrated in the OSSMETER platform being implemented in WP5. Without going into the details of the architecture of the OSSMETER platform (which is the main subject of D5.1 [3] and D5.2 [4]) it is important to note that for each metamodel a corresponding Eclipse/OSGI plug-in project has been developed. Each project contains a number of Java classes obtained out of the meta-classes defined in the corresponding metamodel. Such Java classes permit to create, save, load and query models conforming to the corresponding metamodel.

The subsequent steps of WP2 mainly concern the creation of models conforming to the developed metamodels. In this activity a number of issues we have already identified have to be managed. In particular, importing into the platform projects, which are managed by existing forges implies the development of dedicated forge-specific importers. There are at least two main cases that have to be managed with respect to the different interfaces, which are provided for accessing the data:

- forges like GitHub or SourceForge providing open APIs or Web Services that can be used to extract the data in a structured way;
- forges like Eclipse providing no further information than the project pages in HTML.

Furthermore, having the problem of dealing with a large amount of data, an efficient way to retrieve and manage them it is needed. The preferred strategies to get the data will be the following:

- **open API**: if a structured API is provided it will be the preferred method (GitHub, SourceForge). In this case, the process has to take care of possible APIs restrictions that forges can apply. For instance, the rate limit for the GitHub APIs is 5,000 requests per hour for authenticated users, 60 requests per hours for unauthenticated users (see Section [3]);
- **database exports**: if there are no open APIs, an export of the database will be requested to the forge owner (this has been the first attempt with the Eclipse forge). A program will be developed to retrieve the data from the export and persist it in the OSSMETER database;
- **parsing the project Web pages**: if there are no other options, specific parsers will be developed. The development of such parsers will consist of standard Java classes (like BufferedReader, Document or XPath) and the process will be i) getting the projects web pages, ii) transforming HTML code to XML documents, iii) parsing the XML documents and extracting the data that will be in turn stored into the platform database.

Since there is a huge amount of data to be managed in efficient and scalable ways, a NoSQL database will be used. In this respect, WP5 identified MongoDB as a candidate database manager. Moreover, to make the development process easier, WP5 has developed a Java POJO generator for MongoDB, that uses annotated EMF files to generate all the infrastructure to persist, search and read the data from a MongoDB database [4].

It is important to recall that the OSSMETER metamodels will be continuously refined to address unforeseen requirements and to extend their expressiveness with respect to the information required by the metric providers that will be developed.
References


