Project Number 318772

D3.3 – Language Agnostic Source Code Quality Analysis

Version 1.0
30 October 2014
Final

Public Distribution

Centrum Wiskunde & Informatica

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**

Every effort has been made to ensure that all statements and information contained herein are accurate, however the OSSMETER Project Partners accept no liability for any error or omission in the same.

© 2014 Copyright in this document remains vested in the OSSMETER Project Partners.

Project Partner Contact Information

Centrum Wiskunde & Informatica Jurgen Vinju Science Park 123 1098 XG Amsterdam, Netherlands Tel: +31 20 592 4102 E-mail: jurgen.vinju@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 PortimÃ£o, Portugal Tel: +351 282 485052 E-mail: nuno.santana@unparallel.pt	

Contents

1 Introduction **2**

1.1 Outline of Deliverable 3.3 . . . 2

1.2 The historic viewpoint . . . 2

1.3 Controlling the size factor and acknowledging skewed distributions . . . 3

1.4 From measuring quality to comparing evidence of contra-indicators . . . 4

1.5 Reuse and related work . . . 5

2 Integrating language agnostic metrics into M3 **5**

2.1 Generic M3 . . . 6

2.2 Benefits and pitfalls . . . 6

2.3 Conclusion . . . 7

3 Language agnostic metric providers **7**

3.1 LOC per language . . . 7

3.2 Readability . . . 8

3.3 Comments . . . 9

3.4 License Headers . . . 9

3.5 Type I clones . . . 10

4 Factoids for the OSSMETER quality model **10**

4.1 Percentage of commented-out code . . . 12

4.1.1 Motivation . . . 12

4.1.2 Metrics . . . 12

4.1.3 Example natural language output . . . 13

4.1.4 Code . . . 13

4.2 Comment percentage . . . 13

4.2.1 Motivation . . . 13

4.2.2 Metrics . . . 14

4.2.3 Example natural language output . . . 14

4.2.4 Code . . . 14

4.3 Header use . . . 15

4.3.1 Motivation . . . 15

4.3.2 Metrics . . . 15

4.3.3 Example natural language output . . . 15
4.3.4 Code . . . 15
4.4 Code size . . . 16
4.4.1 Motivation . . . 16
4.4.2 Metrics . . . 16
4.4.3 Example natural language output . . . 16
4.4.4 Code . . . 16
4.5 Code clones . . . 17
4.5.1 Motivation . . . 17
4.5.2 Metrics . . . 18
4.5.3 Example natural language output . . . 18
4.5.4 Code . . . 18
4.6 File readability . . . 19
4.6.1 Motivation . . . 19
4.6.2 Metrics . . . 19
4.6.3 Example natural language output . . . 19
4.6.4 Code . . . 19
4.7 Development team stability . . . 20
4.7.1 Motivation . . . 20
4.7.2 Metrics . . . 20
4.7.3 Example natural language output . . . 20
4.7.4 Code . . . 21
4.8 Development team experience . . . 21
4.8.1 Motivation . . . 21
4.8.2 Metrics . . . 21
4.8.3 Example natural language output . . . 21
4.8.4 Code . . . 22
4.9 Development team experience spread . . . 23
4.9.1 Motivation . . . 23
4.9.2 Metrics . . . 23
4.9.3 Example natural language output . . . 23
4.9.4 Code . . . 23
4.10 Weekend project . . . 24

4.10.1 Motivation 24
4.10.2 Metrics 24
4.10.3 Example natural language output 24
4.10.4 Code 24
4.11 Commit size 25
4.11.1 Motivation 25
4.11.2 Metrics 25
4.11.3 Example natural language output 25
4.11.4 Code 25
4.12 Churn volume 26
4.12.1 Motivation 26
4.12.2 Metrics 26
4.12.3 Example natural language output 26
4.12.4 Code 26
4.13 Commit locality 27
4.13.1 Motivation 27
4.13.2 Metrics 27
4.13.3 Example natural language output 27
4.13.4 Code 27

5 Summary and Conclusions **28**
5.1 Summary 28
5.2 Conclusions 28

Document Control

Version	Status	Date
0.1	Initial draft	10 September 2014
0.5	Further updates	20 October 2014
0.9	For internal review draft	25 October 2014
1.0	QA review updates	30 October 2014

Executive Summary

This deliverable is part of WP3: Source Code Quality and Activity Analysis. It provides descriptions and prototypes of the tools that are needed for source code quality analysis in open source software projects. It builds upon the results of:

- Deliverable 3.1 where infra-structure and a domain analysis have been investigated for Source Code Quality Analysis and initial language-dependent and language-agnostic metrics have been prototyped.
- Deliverable 3.2 where source code activity metrics have been investigated.
- Collaboration with WP2 and WP5, where an integrated quality model is developed which brings together metrics into concise and comparable descriptions ("factoids") of project quality aspects.

In this deliverable we report solely on source code and source code activity metrics which work on any (programming) language. For language specific quality analysis we refer to Deliverable 3.4. The work on Tasks 3.3 and 3.4 has been done in parallel. On the one hand, in order to prevent unnecessary duplication the final report on the satisfaction of the requirements that were identified in Deliverable 1.1 are presented not here, but in Deliverable 3.4 instead. On the other hand, for the sake of cohesion and readability, some general design considerations concerning the metrics and their aggregation is copied between the two deliverable documents in the introduction sections of both documents 1).

The current deliverable does include an update to the Source Code Activity Analysis metrics which were reported on earlier in Deliverable 3.2. The reason is that these metrics are also language independent. The update is necessary to link the results of Deliverable 3.2 into the overall quality model of OSSMETER (see Deliverable 2.4).

In this deliverable we present:

- A brief summary of motivation and challenges for Task 3.3;
- A list of all language-independent source code quality metrics including their motivation in GQM terminology.
- An updated list of all source code activity metrics including their motivation in GQM terminology.
- An overview of the use of the above metrics in the OSSMETER quality model through factoids.

1 Introduction

Language-Agnostic Source Code Quality Analysis (CWI) Tools are needed to create data according to the OSS Quality Attribute Metamodel with data for actual OSS projects.

Language agnostic quality analyses are either motivated by necessity or by natural generality. By necessity, for languages for which we do not (yet) have language specific analysis all we can report are metrics which are language agnostic. Many projects contain different kinds of languages. The OSSMETER project has focused mainly on Java and PHP (see Deliverable 3.4), but for completeness sake we must report on the existence, diversity and volume of source code written in other languages.

By natural generality we have metrics which do not need in-depth knowledge of the syntax or semantics of programming languages. If at any time a metric could be implemented in a language agnostic manner, without loss of accuracy, it probably also should be implemented as such. In this manner we can support more metrics for more languages. As you will read about in this document, the number of metrics we found that are useful in this manner is limited but they are valuable nevertheless.

For consistency's sake and the stratification of the design of OSSMETER all metrics on source code, be it specific or agnostic, are implemented using the Rascal meta-programming language and libraries. The OSSMETER common intermediate model for storing source code information, M3, is used in all cases (See Deliverables 3.1 and 3.2 for details on M3).

1.1 Outline of Deliverable 3.3

- The rest of this introduction describes general design considerations for source code quality analyses. We factor this discussion here to avoid repetitive argumentation in the description of the metrics.
- In Section 2 we describe how we incorporate language agnostic metrics into the OSSMETER platform to unify these metrics with language specific metrics (see also Deliberable 3.4).
- Section 3 motivates and describes each language agnostic metric
- Section 4 motivates and describes all factoids defined on top of the language agnostic metrics.
- We summarize this deliverable in Section 5.

1.2 The historic viewpoint

In general, for all the following metrics, it holds that the *historic* view, i.e. the metric stored per day, is usually an interesting view on the development and evolution of a project.

Any basic metric depicts the status-quo of the current day, perhaps summarized over the last period of time. The status-quo is a good perspective because it explicitly ignores the past. Mistakes that have been repaired do not have an influence on such metrics. However, the status-quo can not give insights in trends or events and especially a causal analysis is hard with such a snapshot view.

Therefore the platform caters for historic metric providers (see also WP5). As a convenience an historic version can be generated automatically for all Rascal based metrics, by simply adding an annotation `@historic` to the metric's definitionl. Please refer to Deliverable 3.1 and Deliverable 3.2 for a description of Rascal metric providers and how they are included in the platform.

The *goals* of the historic view are to allow the human user to interpret the data and find explanations and to formulate predictions. Sudden jumps across different metric values may be associated with specific events in the project's process. Furthermore, trends in metrics may be extrapolated to estimate future risks or to make management decisions for turning the tide.

The *questions* the historic views answer are:

- Are there trends in the development in one or more of the metrics?
- Are there sudden events visible which emerge in peaks or steep slopes in one or more of the metrics?

The visualization work (WP5) allows the OSSMETER user to investigate these trends and events by inspection, and some of the *factoids* described in Section 4 also report on basic upward or downward trends for recent periods in time. We will not come back to the historic perspective in this section however, since the motivation and discussion is the similar for all metrics.

1.3 Controlling the size factor and acknowledging skewed distributions

In all source code metrics which aggregate eventually over entire projects, we have the big confounding factor of size (or volume) [7]. Basically all metrics are influenced by code volume and at the same time aggregation (e.g. sum, mean, median) over code artifacts emphasize the size factor. The reason is that the amount of artifacts in a big software project is large (many files, classes, methods) such that any sum can be dominated by the amount of terms rather than the actual value of the terms in the aggregated sum [17].

The confounding factor of size has two unfortunate consequences:

- Metrics between different projects become incomparable because almost no two projects have the same or comparable size.
- All metrics which use sum or mean to aggregate over entire projects measure the same thing, namely size.

Furthermore, since the distribution of most metrics over code artifacts (for example: cyclomatic complexity over methods) is not Gaussian [20], both sum and mean as aggregation method tend to hide important observations. For instance, a mean for a long tail (exponential) distribution usually ends up around the absolute minimum while a project could contain a large amount of influential "outliers"[1] in the long tail.

For these two reasons, and unlike other metric tools which are available online, in OSSMETER we avoid sum and mean wherever possible. Instead we work with other aggregation methods, trying to normalize for size where possible and trying to capture the essence of specific distribution of the method where possible.

We use quantiles[2] and the Gini coefficient[3] for many metrics. Both statistical methods are easy to explain and both help to characterize a distribution without assuming a particular distribution type and

[1] It is technically wrong to call them outliers in this case, but it gets the point across.
[2] `http://en.wikipedia.org/wiki/Quantile`
[3] `http://en.wikipedia.org/wiki/Gini_coefficient`

without being influenced by the size factor. With quantiles we divide a distribution in four parts to find out which part most of the metric values fall into: the first 25%, 50% or 75% respectively. With this we can answer questions about the support for a quality contra-indicator: is there enough wrong to raise a red flag? The Gini coefficient measures "evenness": are all methods just as complex or do we have just a few methods which hold most of the complexity? The Gini coefficient enables to answer questions about the impact of quality contra-indications (is this problem everywhere in the project or localized to a few files?).

We should note that the Gini coefficient is sensitive to the amount of data points measured. In general, given a number of data points, n, the scale of Gini ranges between 0 and $1 - 1/n$ [1]. This means that for smaller projects the maximum Gini coefficient may differ significantly depending on the actual size of the project. This makes Gini coefficients harder to compare between relatively small projects with fewer than a dozen files or so. However, as soon as projects actually become hard to analyze manually because they have many files, then this problem disappears and the Gini coefficient effectively ranges between 0 and 1. For the larger projects Gini coefficients are an excellent aggregation method, being both size agnostic and distribution-type agnostic.

1.4 From measuring quality to comparing evidence of contra-indicators

The OSSMETER project is about measuring quality of open source source code. Measuring quality can not be done in an absolute fashion, because by definition it is of a subjective nature[4]. For OSSMETER we are looking for quality as defined by Locke as "secondary qualities": i.e. properties which are dependent on the subjective interpretation by a person in a given context [9]. Naturally, from source code we have to start from measuring "primary qualities": attributes which are intrinsic to it and thus observable without considering context.

To lift our primary observations to secondary quality indicators we use two strategies: contra-indication and comparison by the human user. The first is to measure exactly the opposite of quality: practically all the metrics we collect are measuring the lack of quality by first locating common contra indications for quality and then assessing whether or not the size of the indicator or the amount of instances is significant enough on the project level. The underlying assumptions are that problematic projects will show these measurable signs of bad quality and that if two comparable projects do not show any of these signs the choice between them will not be consequential.

The second strategy in OSSMETER is to summarize (aggregate) primary qualities of two comparable projects and leaving the assessment of secondary quality to the user. The (subjective) selection of which projects to compare is also left -intentionally- to the user. OSSMETER's automatic summarization of the projects saves the user a lot of time reading and assessing code, but it leaves the eventual qualitative judgment to the user. The underlying assumption is that projects are faithfully characterized by the descriptions which OSSMETER generates: the right primary qualities are summarized in the right way. The results from Deliverable 1.1, which include the requirements from the project partners who will use OSSMETER, should ensure that this is the case. Also the ability to configure and fine-tune OSSMETER's quality model by the user is important in this respect (see WP5).

[4] http://en.wikipedia.org/wiki/Quality_(philosophy)

1.5 Reuse and related work

As explained in earlier deliverables D3.1 and D3.2 for WP3 we decided to re-implement all metrics from scratch, but based on reusing existing programming language front-ends. A lot of heavy lifting for the correct processing of programming language syntax and semantics is in the parsing, name analysis and type analysis. Not reusing language front-ends would have rendered the OSSMETER project infeasible. Instead we now have high quality and reliable front-ends to depend upon. Clearly reusing a front-end also represents a considerable investment (see Deliverable D3.1 and Deliverable D3.2), but developing one from scratch is much more expensive and error-prone.

By developing each metric from scratch we have complete control over their definition and can make sure this definition is consistent across different language specific and language agnostic implementations.

By developing the metrics in the domain specific language for meta programming, Rascal [16, 15], we can make sure their implementation is concise and easy to maintain, and it is reasonably easy to add a new programming language to the set.

Nevertheless we have taken notice of a number of projects and tools which are used for source code analysis and measurement. These were also reported on in Deliverable 3.1 and now we complete the list for the current deliverable. From some of these we have borrowed thresholds and definitions of metrics (this is mentioned where appropriate in this report).

- CheckStyle - `http://checkstyle.sourceforge.net`
- PMD - `http://pmd.sourceforge.net/`
- Coverity - `http://www.coverity.com/`
- SonarQube - `http://www.sonarqube.org/`
- CodeSonar - `http://www.grammatech.com/codesonar`
- Understand - `http://www.scitools.com/`
- FOSSology - `http://www.fossology.org`
- TXL - `http://www.txl.ca/`
- DMS - `http://www.semdesigns.com/Products/DMS/DMSToolkit.html`
- Bauhaus - `http://www.bauhaus-stuttgart.de/bauhaus/demo/index.html`
- Squale - `http://www.squale.org/`

2 Integrating language agnostic metrics into M3

To integrate language agnostic metrics into OSSMETER we follow exactly the architecture as depicted in Figure 1. The exact same architecture is used for the language specific metrics. First we parse and process all source file to produce M3 models and abstract syntax trees (ASTs). Then we compute metrics on top of these M3 models and AST models.

Since language agnostic metrics work on files and treat the contents of files as strings filling in this architecture is easy:

- The parser/compiler for such text files to M3 trivially detects all files and folders and then reads in the contents of all text files.

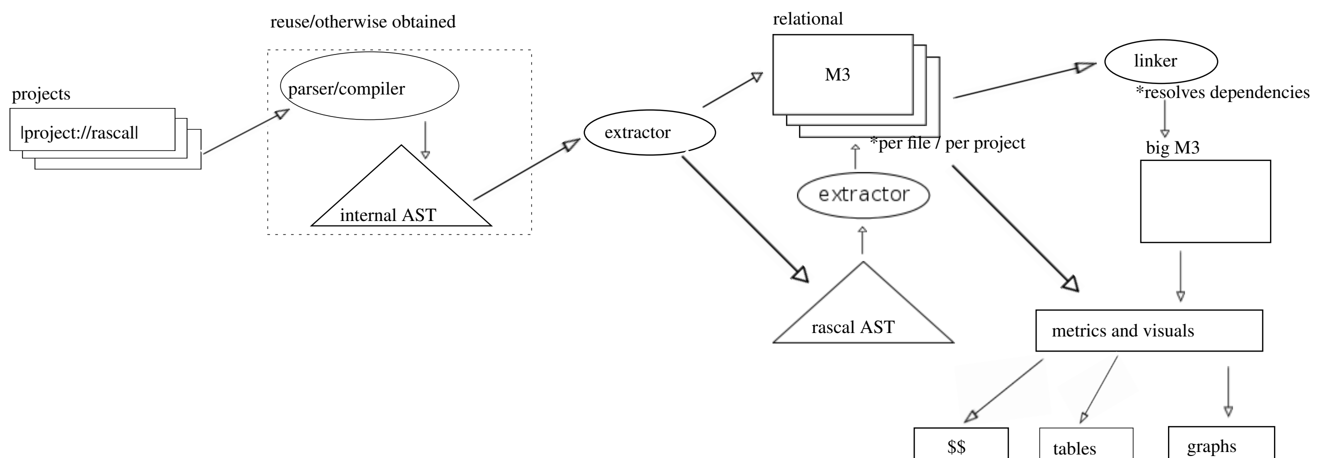

Figure 1: Extracting information from source code into the M3 model, and then using the M3 model for metrics

.

- The model wraps the containment relation of files and folders and the contents on each files.
- All language agnostic metrics work directly on this internal string based representation in M3.
- Linking generic models per file to one big model is done by plain set union.

2.1 Generic M3

In Figure 2 we include the model for any text file as part of M3. This adds the following definitions:

- A name constant to identify the "generic" language. All language agnostic metrics work on M3 and AST models for this language labeled `generic`.
- A wrapper for the contents in the `AST` algebraic data-type
- The use of three of the core relations from M3:

  declarations lists the existence of all files as "compilation units"

  containment for folders and the files and folders they contain

  messages for accidental I/O error while retrieving the contents

2.2 Benefits and pitfalls

The benefits of modeling plain file contents as instances of the M3 model are:

- Reuse of the OSSMETER infra-structure for generating M3: managing cloned working copies, managing version meta data and (eventually) managing incremental updates.
- Stratified and predictable design: all metric implementations look similar.
- Avoid disk I/O: by reading in the files as quickly as possible into the in-memory M3 model analyses will be quick.

```
1  // the generic language is identified by this constant:
2  data Language = generic();
3
4  // ASTs point to the file location and wrap the contents as a list of strings:
5  data \AST = lines(list[str] contents = [], loc file = |unknown:///|);
6
7  // Relevant relations in the M3 core model for the generic language:
8  anno rel[loc name, loc src] M3@declarations; // list all files here
9  anno rel[loc from, loc to] M3@containment; // files and folders
10 anno list[Message]          M3@messages;    // possible I/O errors
```

Figure 2: M3 model elements for the generic language.

And, we have some pitfalls:

- M3 does require in memory representation so OSSMETER worker (virtual) machines should have enough heap space to be able to hold at least one version of the source code of a project.
- The metrics are not optimally efficient due to an explicit intermediate model representation. Some metrics could easily be implemented without storing an intermediate model and actually have a *streaming* implementation (like a Un*x pipe). Thus we pay efficiency and memory space for the acquired consistency and reuse.

2.3 Conclusion

The generic language is now included as a code model in OSSMETER and can be built upon by any metric provider in the platform.

3 Language agnostic metric providers

Here we list all developed *metrics*, motivated by their *goal* and the *questions* they answer. We use the Goal-Question-Metric structure and terminology [4].

3.1 LOC per language

The first metric estimates the programming language from the file extension and counts physical lines of code for each of such file [12].

The *goal* is to understand the risks involved in adopting to or contributing to the project in terms of the sheer volume of code which needs to be maintained, as well as in terms of the required programming language skills and knowledge. Bigger projects are more risky and rarely used programming languages are risky and so are rarely used combinations of languages.

The *questions* are:

- How many different languages are used?

```
1  @memo
2  map[str, str] getLanguageExtensions() {
3  languageExtensions = {
4    <"ActionScript", ["as"]>,
5    <"Ada", ["adb"]>, <"ASP", ["asp"]>,
6    <"ASP.NET", ["aspx", "axd", "asx", "asmx", "ashx"]>,
7    <"Assembler", ["asm"]>, <"C", ["c", "h"]>,
8    <"C#", ["cs"]>, <"C++", ["cpp", "hpp", "cxx", "hxx", "cc", "hh"]>,
9    <"Clojure", ["clj"]>, <"Cobol", ["cob"]>, <"CoffeeScript", ["coffee"]>,
10   <"Coldfusion", ["cfm"]>, <"CSS", ["css"]>,
11   <"CUDA", ["cu"]>, <"Erlang", ["erl", "hrl"]>, <"F#", ["fs"]>,
12   <"Flash", ["swf"]>, <"Fortran", ["f"]>,
13   <"GLSL", ["glsl", "vert", "frag"]>, <"Go", ["go"]>,
14   <"Haskell", ["hs", "lhs"]>, <"HLSL", ["hlsl"]>,
15   <"HTML", ["html", "htm", "xhtml", "jhtml", "dhtml"]>,
16   <"J#", ["jsl"]>, <"Java", ["java", "jav"]>,
17   <"JavaScript", ["js", "jse", "ejs"]>,
18   <"JSP", ["jsp", "jspx", "wss", "do", "action"]>,
19   <"LISP", ["lisp", "cl"]>, <"Lua", ["lua"]>,
20   <"Matlab", ["matlab"]>, <"ML", ["ml", "mli"]>,
21   <"Objective C", ["m", "mm"]>, <"Pascal/Delphi", ["pas"]>,
22   <"Perl", ["pl", "prl", "perl"]>, <"PHP", ["php", "php4", "php3", "phtml"]>, <"PL/I", ["pli"]>,
   <"Python", ["py"]>,
23   <"Rascal", ["rsc"]>, <"Ruby", ["rb", "rhtml"]>,
24   <"Scala", ["scala"]>,
25 <"Shell script", ["sh", "bsh", "bash", "ksh", "csh"]>,
26   <"Smalltalk", ["st"]>, <"SQL", ["sql"]>,
27   <"TCL", ["tcl"]>, <"(Visual) Basic", ["bas", "frm", "cls", "ctl"]>,
28   <"Visual Basic Script", ["vbs", "vbscript"]>,
29   <"XML", ["xml", "xst"]>, <"XSLT", ["xslt"]>
30 };
```

Figure 3: Mapping file extensions to programming language names. The sources used to construct this table are `http://en.wikipedia.org/wiki/List_of_programming_languages` and `http://www.file-extensions.org/`.

- How much code is written in which language?
- Is there a dominant language?

The *metrics* we use are physical lines of code (LOC) and an educated guess of the language by the file extension. To construct an up-to-date table of (contemporary) file extensions we used the information from `http://en.wikipedia.org/wiki/List_of_programming_languages` and `http://www.file-extensions.org/` (see Figure 3). The LOC-per-language metric reports for each language used in the project how many physical lines of code are there.

3.2 Readability

A low-level readability metric checks for the “proper” use of spaces around punctuation in source code files. By punctuation we mean the use of characters such as `','` `';'` `'{'` `'}'`. It has been argued but

never shown experimentally [18, 3] that inconsistent spacing around these characters can lead to hard to understand or easy to misunderstand code. At least it is clear that inconsistency and unpredictability do lead to confusion.

The *goal* of the metric is to assess a possible risk factor: is the layout of this code unpredictable and therefore hard to read? The *question* is whether or not there is enough support for inconsistent spacing in a project for a quality contra-indication.

The basic *metric* we use is to count the locations in a file where we:

- expect whitespace before or after a punctuation mark but there is none;
- have whitespace before or after a punctuation mark but we expected none.

From this number we compute the ratio between the total number of positions around punctuation characters and the above mismatches. The ratio is a size-normalized indicator for the evidence of spacing inconsistent with the norm. What we expect to be typical "expected" use of whitespace we borrowed from the definition of the CheckStyle tool (as required). See the website of CheckStyle: `http://checkstyle.sourceforge.net/config_whitespace.html`.

3.3 Comments

If we assume C-style commenting convention, which is common for the languages we are interested in for OSSMETER, then we may consider the following metrics to be language agnostic. These are all string based, except for recognizing the common comment delimiters such as `'/*'`, `'*\'`, `'\\'`, etc.

According to many, including McConnell [19] and Baecker [3], source code comments are an important factor in source code quality. The *goal* is to find out whether there are quality contra-indicators in the use of source code comments. The relevant *questions* are if the right balance between code and comments has been struck, and whether or not source code comments have been used as a poor man's version management system (i.e. by commenting out code).

The *metrics* we have to answer these questions are:

- commentLOC, commentLinesPerLanguage: how many physical lines of comments?
- commentedOutCode,commentedOutCodePerLanguage: how many physical lines of comments are actually commented out code and not natural language explanations?
- percentageCommentedOutCode: the ratio between normal comments and code which has been commented out
- commentPercentage: the ration between code and source code comments

3.4 License Headers

License Headers are of interest for judging subjective quality of an open-source project. In this case we are not measuring or inspecting the actual contents of the licenses, like for example is done by the FOSSOLOGY project[5] and the MARKOS project[6]. Instead we focus on the maintenance of the actual source text of the license headers rather than what they mean.

[5] `http://www.fossology.org`
[6] `http://www.markosproject.eu`

The *goal* we have with the OSSMETER License Header metrics is to assess whether due diligence has been applied regarding the open-source licensing of the files in the project. The *questions* are if license headers are actually present on every relevant file and if a variety of different licenses have been used or not.

We *measure* the physical lines of code in the headers of all files (headerLOC) and how much of the file consists of header (headerPercentage). Furthermore we measure the number of different headers used (headerCounts) and which headers are used in which files (headerUse).

3.5 Type I clones

For the same reason as source code comments and readability of punctuation, cloning can be considered a language agnostic property of code written in programming languages. By the way, source code comments and punctuation both are used in filters we apply before measuring clones. Source code clones are sometimes considered harmful and sometimes not [14, 13, 6].

The *goal* of measuring the occurence of Type I clones (a.k.a. exact or literal clones) is to find contra-indication of design quality. If a lot of code is cloned within a single project then maintenance may be hard but also generally we see this as a sign that the principles of separation-of-concerns and reuse have not been applied. Such a contra-indication could be confirmed by language specific metrics of coupling and cohesion for example (see Deliverable 3.4).

The Type II and Type III clone types allow for more differences between the copied segments of code, for example by allowing variable names to differ and allowing gaps between clones statements. For Type II and Type III clones it is less clear how often they are quality contra-indicators because Type II and Type III clones are often not maintained synchronously (for example it may have been the intention for the clones to diverge). Because this story about Type I and II clones is unclear we do not feel secure in tagging them as quality contra-indication. So we do not measure their occurrence. Also, Type II and III clone measurement is not language agnostic and computationally more intensive.

The *question* we have is whether Type I code cloning is common practice in the current project. The *metric* is, per language and in general, how many physical lines of code contribute to a clone of at least six physical lines? Luckily to compute this metric we do not actually have to compute all the clone classes [11, 5], but rather have to track only blocks of code of size 6 which occur at least twice. All lines in such blocks are counted and then we compute the ratio between this count and the total size of the system [8].

To give an example of the typical code written to compute a metric such as counting Type I clones, please see Figure 4.

4 Factoids for the OSSMETER quality model

Factoids, in the context of the OSSMETER platform, are textual summaries of measurement results including a star rating on a scale of four. Factoids are an intrinsic element of the OSSMETER quality model (see WP5). Most source code metrics from WP3 will serve as input parameters for one or more

```
1  private alias Block = list[str];
2
3  map[str, int] cloneLOCPerLanguage(rel[Language, loc, AST] asts = {}) {
4    map[str, int] result = ();
5    int BLOCKSIZE = 7; // larger than 6
6
7    langs = asts<0> - { generic() };
8
9    for (lang <- langs) {
10     set[Block] uniqueBlocks = {};
11
12     for (<f, _> <- asts[lang]) {
13       if ({lines(content)} := asts[generic(), f]) {
14         norm = normalizeCode(content, lang);
15
16         if (size(norm) >= BLOCKSIZE) {
17           Block block = [];
18           set[int] linesInClones = {};
19
20           for (i <- [0..size(norm)]) {
21             block += [norm[i]];
22
23             if (size(block) == BLOCKSIZE) {
24               if (block in uniqueBlocks) {
25                 linesInClones += { i - j | j <- [0..BLOCKSIZE] };
26               } else {
27                 uniqueBlocks += {block};
28               }
29
30               block = block[1..]; // remove first element
31             }
32           }
33
34           result["<lang>"]?0 += size(linesInClones);
35         }
36       }
37     }
38   }
39
40   return result;
41 }
```

Figure 4: Rascal code for measuring the amount of physical lines of code contributing to code clones.

factoids. Also source code activity metrics (see Deliverable 3.2) are used for the generation of factoid descriptions and ratings.

In this section we describe the factoids which depend on language agnostic source code metrics and source code activity metrics. Accidentally, all source code activity metrics described here are also language agnostic.

For every factoid we include its rationale and its implementation. The implementation contains the template for the natural language text generated and usually a decision tree (encoded as structural control flow in the Rascal programming language) to compute the star rating.

The design of factoids for WP3 is based on the following principles:

- Avoid weighted averages between incomparable metrics. The only time when we would allow (weighted) averages between metrics if we know their dimension and unit of measure to be exactly equal.
- Use as few metrics as possible to compute the star rating. Many metrics compute attributes in a similar or overlapping dimension (although the unit of measure is probably different). We can compute the star rating based on a canonical or representative metric, and use natural language, tables and lists to report the other metrics in the same factoid. This avoids any confusion without loss of completeness.
- Use decision trees to compute star ratings from multiple metrics. Suppose two metrics are used to compute the star rating, one will be the first used to choose between good (3 or 4 stars) and bad (1 or 2 stars), then the next metric can be used to fine tune and choose the exact star rating. This makes it easy to relate the resulting rating back to the original metrics.
- Optimize for basic statistical methods, i.e. use nothing fancy. The more involved statistical computations are the harder they are to interpret. So we do not apply power transforms or use advanced correlation tests, etc. The set of mathematical functions we use is min, max, gini, median, and quantiles and percentiles.

In the following each subsection reports on a single factoid including motivation, the metrics used, an example natural language output and the code used to compute the factoid.

4.1 Percentage of commented-out code

4.1.1 Motivation

It's bad to have a lot of commented-out code in the project. It's a tell-tale sign of experimentation (as opposed to production code) or that the version management system is not used in a pragmatic way.

4.1.2 Metrics

This factoid is based on `locPerLanguage` and `commentedOutCodePerLanguage`.

4.1.3 Example natural language output

"The percentage of commented out code over all measured languages is <totalPercentage>%. The percentages per language are <commaSeparatedTable>."

4.1.4 Code

```
1  Factoid percentageCommentedOutCode(map[str, int] locPerLanguage = (), map[str, int] commentedOutCodePerLanguage = ())
2    // only report figures for measured languages
3    languages = domain(locPerLanguage) & domain(commentedOutCodePerLanguage);
4
5    if (isEmpty(languages)) {
6      throw undefined("No LOC data available", |unknown:///|);
7    }
8
9    totalLines = toReal(sum([ locPerLanguage[l] | l <- languages ]));
10   totalCommentedLines = toReal(sum([ commentedOutCodePerLanguage[l] | l <- languages ]));
11
12   totalPercentage = (totalCommentedLines / totalLines ) * 100.0;
13
14   stars = four ();
15
16   if ( totalPercentage >= 5) {
17     stars = \one();
18   } else if ( totalPercentage >= 2) {
19     stars = two();
20   } else if ( totalPercentage >= 1) {
21     stars = three ();
22   }
23
24   txt = "The percentage of commented out code over all measured languages is <totalPercentage>%.";
25
26   languagePercentage = ( l : 100 * commentedOutCodePerLanguage[l] / toReal(locPerLanguage[l]) | l <-
   languages );
27
28   otherTxt = intercalate (", ", ["<l[0]> (<languagePercentage[l]>%)" | l <- languages]);
29   txt += " The percentages per language are <otherTxt>.";
30
31   return factoid ( txt , stars );
32 }
```

4.2 Comment percentage

4.2.1 Motivation

We expect a ratio of about 9% comments for any number of lines of code. Over-documentation leads to inconsistencies and under-documented code is hard to understand. According to Capers Jones [10] the ideal comment density is 1 comment per 10 statements. Either less or more is worse than this optimum. This amounts to around 9% if all lines would be either statements or comments.

Riehle et al. [2] produced hard data about what to expect in open-source projects in terms of comment density: the average comment density of in over 5000 open-source projects was 18.67%.

4.2.2 Metrics

We used `locPerLanguage` and `commentLinesPerLanguage` for this factoid.

4.2.3 Example natural language output

"The percentage of lines containing comments over all measured languages is <totalPercentage>%. Headers and commented out code are not included in this measure."

4.2.4 Code

```
1  Factoid commentPercentage(map[str, int] locPerLanguage = (), map[str, int] commentLinesPerLanguage = ()) {
2    // only report figures for measured languages
3    languages = domain(locPerLanguage) & domain(commentLinesPerLanguage);
4
5    if (isEmpty(languages)) {
6      throw undefined("No LOC data available", |unknown:///|);
7    }
8
9    totalLines = toReal(sum([locPerLanguage[l] | l <- languages ]));
10   totalCommentedLines = toReal(sum([ commentLinesPerLanguage[l] | l <- languages ]));
11
12   totalPercentage = (totalCommentedLines / totalLines ) * 100.0;
13
14   stars = \one();
15
16   if ( totalPercentage < 25) {
17         if ( totalPercentage > 8) {
18                 stars = four ();
19         }
20         else if ( totalPercentage > 6) {
21                 stars = three ();
22         }
23         else if ( totalPercentage > 4) {
24                 stars = two();
25         }
26   }
27   else {
28         stars = three ();
29   }
30
31   txt = "The percentage of lines containing comments over all measured languages is <totalPercentage>%.
32         'Headers and commented out code are not included in this measure.";
33
34   languagePercentage = ( l : 100 * commentLinesPerLanguage[l] / toReal(locPerLanguage[l]) | l <-
languages );
```

```
35
36     otherTxt = intercalate (", ", ["<l> (<languagePercentage[l]>%)" | l <- languages]);
37     txt += " The percentages per language are <otherTxt>.";
38
39     return factoid ( txt , stars );
40 }
```

4.3 Header use

4.3.1 Motivation

To be open-source, all code needs an open-source license. We need to know if the developers have put the licenses consistently. It is also good to know if within a project a variety of licenses is present.

4.3.2 Metrics

We use `headerCounts` and `headerPercentage`

4.3.3 Example natural language output

- “The percentage of files with a header is <headerPercentage>The largest group of similar headers spans <highestSimilarity>
- “Only <headerPercentage>% of the files contain a header.”
- “No headers found.”;

4.3.4 Code

```
1  Factoid headerUse( list [ int ] headerCounts = [], real headerPercentage = -1.0) {
2
3          if (headerCounts == [] || headerPercentage == -1.0) {
4                  throw undefined("Not enough header data available ", |tmp:///|);
5          }
6
7          stars = 1;
8          message = "";
9
10         if (headerPercentage > 50.0) {
11                 stars += 1;
12
13                 if (headerPercentage > 95.0) {
14                         stars += 1;
15                 }
16
17                 highestSimilarity = max(headerCounts) / sum(headerCounts);
18
19                 if ( highestSimilarity > 0.8) {
20                         stars += 1;
21                 }
```

```

            message = "The percentage of files with a header is <headerPercentage>%." +
                      "The largest group of similar headers spans <highestSimilarity>% of the files.";
        }
        else if (headerPercentage > 0.0) {
            message = "Only <headerPercentage>% of the files contain a header.";
        }
        else {
            message = "No headers found.";
        }

        return factoid(message, starLookup[stars]);
}
```

4.4 Code size

4.4.1 Motivation

How much code is written in which language for this project? Language choice is an important (subjective) maintenance factor and too many different languages may be confusing.

4.4.2 Metrics

We only use `locPerLanguage` for this factoid.

4.4.3 Example natural language output

“The total size of the code base is <totalSize> physical lines of code. The main development language of the project is <mainLanguage>, with <locForMainLanguage> physical lines of code. The following <size(sorted) - 1> other languages were recognized: <tableWithOtherSizes>.”

4.4.4 Code

```
Factoid codeSize(
        map[str, int] locPerLanguage = (),
        int projectAge = -1,
        int numberOfActiveCommittersLongTerm = -1) {
  if (isEmpty(locPerLanguage)) {
    throw undefined("No LOC data available", |unknown:///|);
  }

  // generic() is already removed in locPerLanguage
  lrel[str, int] sorted = sort(toRel(locPerLanguage),
    bool (tuple[str, int] a, tuple[str, int] b) {
        return a[1] > b[1]; // sort from high to low
    });

  totalSize = ( 0 | it + locPerLanguage[l] | l <- locPerLanguage );
```

```
16
17     mainLang = sorted [0];
18
19     txt = "The total size of the code base is <totalSize> physical lines
20            'of code. The main development language of the project is
21            '<mainLang[0]>, with <mainLang[1]> physical lines of code.";
22
23     if (size(sorted) > 1) {
24       otherTxt = intercalate (", ", ["<l[0]> (<l[1]>)" | l <- sorted [1..]]);
25
26       txt += "The following <size(sorted) - 1> other languages were
27              'recognized: <otherTxt>.";
28     }
29
30     if (projectAge > 0) {
31            txt += "The age of the code base is <projectAge> days.";
32     }
33
34     if (numberOfActiveCommittersLongTerm > 0) {
35       txt += "In the last 12 months there have been
36              '<numberOfActiveCommittersLongTerm> people working
37              'on this project.";
38     }
39
40     stars = 1;
41
42     if (totalSize < 1000000) { // 1 star if LOC > 1M
43       stars += 1;
44
45       if (totalSize < 500000) {
46         stars += 1;
47       }
48
49       if (size(locPerLanguage) <= 2) {
50         stars += 1;
51       }
52     }
53
54     return factoid(txt, starLookup[stars]);
55   }
```

4.5 Code clones

4.5.1 Motivation

We measure, independent of the size of a project, the percentage of its source lines which contribute to exact copies (clones) of at least six lines of code. A high percentage indicates copy/paste programming, meaning a bad software design which may lead to maintenance issues in the near future.

4.5.2 Metrics

We use `locPerLanguage` and `cloneLOCPerLanguage` for this factoid.

4.5.3 Example natural language output

“The measured percentage of source code in Type I clones larger than 6 lines is <totalClonePercentage>%. The percentages of clone code per language are <tableWithPercentages>”

4.5.4 Code

```
1  Factoid cloneCode(map[str, int] locPerLanguage = (), map[str, int] cloneLOCPerLanguage = ()) {
2    measuredLanguages = domain(locPerLanguage) & domain(cloneLOCPerLanguage);
3
4    if (isEmpty(measuredLanguages)) {
5      throw undefined("No LOC data available", |unknown:///|);
6    }
7
8    clonePercentagePerLanguage
9      = { <"<l>", (100.0 * cloneLOCPerLanguage[l]) / locPerLanguage[l]> | l ← measuredLanguages };
10
11   lrel[str, real] sorted = sort(clonePercentagePerLanguage,
12     bool (tuple[str, real] a, tuple[str, real] b) {
13         return a[1] > b[1]; // sort from high to low
14     });
15
16   totalLOC = sum([ locPerLanguage[l] | l ← measuredLanguages ]);
17   totalCloneLOC = sum([ cloneLOCPerLanguage[l] | l ← measuredLanguages ]);
18
19   totalClonePercentage = (100.0 * totalCloneLOC) / totalLOC;
20
21   stars = \one();
22
23   if (totalClonePercentage < 10.0) {
24           stars = four();
25   } else if (totalClonePercentage < 15.0) {
26       stars = three();
27   } else if (totalClonePercentage < 20.0) {
28       stars = two();
29   }
30
31     txt = "The measured percentage of source code in Type I clones larger
32           ' than 6 lines is <totalClonePercentage>%.";
33
34   if (size(sorted) > 1) {
35     otherTxt = intercalate(", ", ["<l[0]>: <l[1]>%" | l ← sorted]);
36
37     txt += " The percentages of clone code per language are <otherTxt>.";
38   }
39
```

```
40    return factoid(txt, stars);
41  }
```

4.6 File readability

4.6.1 Motivation

We measure the placement around comma's, semi-colons and curly braces because it has been shown these influence readability and therefore understandability of the code. Here you can see how many of the files in the project are easy to read and how many are harder to read.

4.6.2 Metrics

We use only `fileReadability` for this factoid.

4.6.3 Example natural language output

"In this project, <percentage> of the files readability issues. 'An average file has <med> of the spaces in the wrong place (either no space or too many spaces).";

4.6.4 Code

```
1   Factoid readabilityFactoid(map[loc, real] fileReadability = ()) {
2     if (isEmpty(fileReadability)) {
3       throw undefined("No readability data available.", |tmp:///|);
4     }
5
6     re = {<l, fileReadability[l]> | l <- fileReadability};
7
8     // percentages per risk group
9     lowPerc = [ r | <_,r> <- re, r >= 0.90];
10    medPerc = [ r | <_,r> <- re, r < 0.90 && r >= 0.75];
11    highPerc = [ r | <_,r> <- re, r < 0.75 && r >= 0.50];
12    veryHighPerc = [ r | <_,r> <- re, r < 0.50];
13
14    med = 100.0 * median(medPerc + highPerc + veryHighPerc);
15
16    total = size(fileReadability);
17    star = \one();
18
19    txt = "For readability of source code it is import that spaces around delimiters such as [,;{}] are used.\n";
20
21    if (size(veryHighPerc) == 0 && size(highPerc) == 0 && size(medPerc) <= (total / 10)) {
22      star = four();
23      txt += "In this project less than 10% of the files have minor readability issues.
24             'An average file in this category has <med>% of the expected whitespace.";
25    }
26    else if (size(veryHighPerc) == 0 && size(highPerc) <= (total / 5) && size(medPerc) <= (total / 15)) {
```

```
27        star = three ();
28        txt += "In this project less than 20% of the files have moderate readability issues .
29              'An average file in this category has <med>% of the expected whitespace.";
30     }
31     else if ( size (veryHighPerc) <= ( total / 5) && size(highPerc) <= ( total / 25)) {
32        star = two ();
33        txt += "In this project less than 30% of the files have major readability issues .
34              'An average file in this category has <med>% of the expected whitespace.";
35     }
36     else {
37        star = \one ();
38        txt += "In this project , <percent( size (veryHighPerc) + size (highPerc) + size (medPerc), total )>% of the files have r
39              'An average file in this category has <med>% of the expected whitespace.";
40     }
41     else {
42        star = \one ();
43        txt += "In this project , <percent( size (veryHighPerc) + size (highPerc) + size (medPerc), total )>% of the files have r
44              'An average file has <med>% of the expected whitespace.";
45     }
46
47     return factoid ( txt , star );
48  }
```

4.7 Development team stability

4.7.1 Motivation

We observe developer activity as the number of commits and their date per developer. From this we compute activity in the last two weeks and the last 6 months to see how large the active team of developer is and if this team is growing or shrinking.

4.7.2 Metrics

We use `maximumActiveCommittersEver`, `numberOfActiveCommitters.historic`, `numberOfActiveCommitters`, `sizeOfDevelopmentTeam`, and `numberOfActiveCommittersLongTerm`.

4.7.3 Example natural language output

One of the following four:

- "In the last half year the development team was stable and active."
- "The project has seen hardly anybody developing in the last half year"
- "People have been leaving the development team in the last half year."
- "In the last half year the development team has been growing."

and then: "The total number of developers who have worked on this project ever is <totalDevs>. The maximum number of active developers for this project during its lifetime is <maxDevs>, and in the last two weeks there were <activeDevs> people actively developing, as compared to <longTermActive> in the last twelve months."

4.7.4 Code

```
1  Factoid developmentTeamStability(rel[datetime day, int active] history = {}, int maxDevs = 0, int totalDevs = 0, int act
2    sl = historicalSlope(history, 6);
3
4    stability = \one();
5    team = "";
6
7    if (−0.1 < sl && sl < 0.1 && longTermActive > 0) {
8       stability = \three();
9      team = "In the last half year the development team was stable and active.";
10   }
11   else if (−0.1 < sl && sl < 0.1 && longTermActive == 0) {
12      stability = \one();
13     team = "The project has seen hardly anybody developing in the last half year.";
14   }
15   else if (sl < 0 && longTermActive > 0) {
16      stability = \two();
17     team = "People have been leaving the development team in the last half year.";
18   }
19   else if (sl > 0) {
20      stability = \four();
21     team = "In the last half year the development team has been growing.";
22   }
23
24   txt = "<team>
25         'The total number of developers who have worked on this project ever is <totalDevs>.
26         'The maximum number of active developers for this project during its lifetime is <maxDevs>,
27         'and in the last two weeks there were <activeDevs> people actively developing, as compared to
28         '<longTermActive> in the last twelve months.";
29
30   return factoid(txt, stability);
31 }
```

4.8 Development team experience

4.8.1 Motivation

We measure developer experience within the scope of a project to find out if only newbies are currently developing or at least a few more experienced people are still working on the project. Some projects are abandoned by the lead developers which is a contra-indicator for the future viability of the project.

4.8.2 Metrics

We use `firstLastCommitDatesPerDeveloper` and `commitsPerDeveloper`.

4.8.3 Example natural language output

- "There were no experienced committers working for the project in the last 6 months."

- “There was only one experienced committer working for the project in the last 6 months. Overall, he/she contributed <thisMuch> commits.”
- “The number of experienced committers working for the project in the last 6 months is <numExperiencedCommitters>. Their average overall number of commits is <thisMuch>.”

4.8.4 Code

```
1  Factoid developmentTeamExperience(
2    ProjectDelta delta = \empty(),
3    map[str, tuple[datetime first , datetime last ]] firstLastCommitDates = (),
4    map[str, int ] commitsPerDeveloper = ())
5  {
6    if ( delta == \empty() || commitsPerDeveloper == ()) {
7      throw undefined("No delta available ", |tmp:// /|);
8    }
9
10   today = delta . date ;
11   sixMonthsAgo = decrementMonths(today, 6);
12
13   committersInLastHalfYear = { name | name <- firstLastCommitDates, firstLastCommitDates [name]. last > sixMonthsAgo };
14
15   experiencedCommittersInLastHalfYear = { name | name <- committersInLastHalfYear,
16     firstLastCommitDates [name]. last > decrementMonths(firstLastCommitDates[name]. first , 6),
17     (commitsPerDeveloper[name]?0) > 24 }; // at least 1 commit per week on average
18
19   numExperiencedCommitters = size(experiencedCommittersInLastHalfYear );
20
21   stars = numExperiencedCommitters + 1;
22
23   if ( stars > 4) {
24     stars = 4;
25   }
26
27   txt = "";
28
29   if ( stars == 1) {
30     txt = "There were no experienced committers working for the project in the last 6 months.";
31   }
32   else if ( stars == 2) {
33     txt = "The was only one experienced committer working for the project in the last 6 months.";
34     txt += " Overall , he/she contributed
35            '<commitsPerDeveloper[getOneFrom(experiencedCommittersInLastHalfYear)]>
36            ' commits.";
37   }
38   else {
39     txt = "The number of experienced committers working for the project in
40           ' the last 6 months is <numExperiencedCommitters>.
41           ' Their average overall number of commits is
42           '<mean([commitsPerDeveloper[d] | d <- experiencedCommittersInLastHalfYear])>.";
43   }
44
45   if ( size (committersInLastHalfYear ) == numExperiencedCommitters) {
```

```
46        txt += " There were no other committers active in the last 6 months.";
47      }
48      else {
49        txt += " In total , <size(committersInLastHalfYear)> committers have
50               ' worked on the project in the last six months.";
51      }
52      return factoid ( txt , starLookup[ stars ]);
53    }
```

4.9 Development team experience spread

4.9.1 Motivation

Are the developers working on smaller islands of functionality or is everybody up-to-date with everything? The spread of committers over a project indicates longer term stability of the project.

4.9.2 Metrics

We used `developmentTeamExperienceSpread`, `committersPerFile`, `countCommittersPerFile`, and`committersoverfile`.

4.9.3 Example natural language output

“Developers are spreading out over the entire project and it can be expected that most files have more than one contributor.”

4.9.4 Code

```
1   Factoid developmentTeamExperienceSpread(real developmentTeamExperienceSpread = 0.0, map[loc,int] perFile = ()) {
2     list [int] amounts = [ perFile [ i ] | i <- perFile ];
3     if (amounts == []) {
4       throw undefined("No commit data available .", |tmp:// /|);
5     }
6
7     med = median(amounts);
8     maxi = List::max(amounts);
9
10    if (developmentTeamExperienceSpread >= 0.5) {
11      if (med >= 2) {
12        return factoid (\four (), "Developers are spreading out over the entire project and it can be expected
13                                  ' that most files have more than one contributor .");
14      }
15      else {
16          return factoid (\three (), "Developers are spreading out over the entire project but most files will
17                                     'have only one contributor .");
18      }
19    }
20    else {
```

```
21        if (maxi > 1) {
22          return factoid (\two(), "Developers are mostly focusing on their own files in the project , but there is
23                                  ' definitely some collaboration going on." );
24        }
25        else {
26          return factoid (\one(), "Developers are mostly focused on their own files
27                                  'in the project ." );
28        }
29      }
30    }
```

4.10 Weekend project

4.10.1 Motivation

Assuming there exist projects which are mostly developed in free time over the weekend against projects which are funded by working hours, we try to find out in which of the two categories a project falls. Weekday projects are valued more because, we assume, more monetary investment is involved, indicating longer term project stability.

4.10.2 Metrics

We use `percentageOfWeekendCommits` and `commitsPerWeekday`.

4.10.3 Example natural language output

- "Over the entire lifetime of this project, commits have been done usually over the weekend."
- "Over the entire lifetime of this project, commits are done usually during the week, and hardly during the weekend."
- ...

4.10.4 Code

```
1   Factoid weekendProject(int percentageOfWeekendCommits = -1) {
2     if (percentageOfWeekendCommits == -1) {
3       throw undefined("No weekend commit data available .", |tmp:// /|);
4     }
5
6     if (percentageOfWeekendCommits > 75) {
7       return factoid (\one(), "Over the entire lifetime of this project , commits have been done usually over the weekend.")
8     }
9     else if (percentageOfWeekendCommits > 50) {
10      return factoid (\two(), "Over the entire lifetime of this project , commits are done mostly over the weekend, but also
11    }
12    else if (percentageOfWeekendCommits > 25) {
13      return factoid (\ three (), "Over the entire lifetime of this project , commits are done mostly during the week, but also
14    }
15    else {
```

```
16      return factoid (\four (), "Over the entire lifetime of this project , commits are done usually during the week, and har
17    }
18  }
```

4.11 Commit size

4.11.1 Motivation

Commits should be frequent and small as opposed to rare and big. Smaller increments mean more opportunities for testing and feedback. Also the commit locality makes sure it is easier to track te introduction of bugs. We consider small incremental commits to be a sign of a professional development team.

4.11.2 Metrics

Metrics used are `churnPerCommitInTwoWeeks`, `churnPerCommitInTwoWeeks.historic`, and`churnPerCommitInTwoWeeks`.

4.11.3 Example natural language output

“In the last two weeks the average size of a commit was <ratio>, this is <more,less> than the normal <ratioMedian> in the last six months. The trend is <going down, stable, going up>”

4.11.4 Code

```
1   Factoid commitSize(rel[datetime, int] ratioHistory = {}, int ratio = 0) {
2     ratioSlope = historicalSlope ( ratioHistory , 6);
3     ratioMedian = historicalMedian ( ratioHistory , 6);
4
5     msg = "In the last two weeks the average size of a commit was <ratio>, this is <if ( ratio > ratioMedian) {>more<}els
6           ' than the normal <ratioMedian> in the last six months. The trend is <slopeText( ratioSlope , "going down", " stabl
7
8     if ( ratio < 500) {
9       if (ratioMedian > 250) {
10        return factoid (msg, \two ());
11      }
12      else if (ratioMedian > 125) {
13        return factoid (msg, \three ());
14      }
15      else {
16        return factoid (msg, \four ());
17      }
18    }
19    else {
20      return factoid (msg, \one ());
21    }
22  }
```

4.12 Churn volume

4.12.1 Motivation

Churn is define as the amount of added, removed or changed lines of code. It is a measure of developer output which is not biased for adding code only. Next to the activity of the developers we present the total output of the project per two weeks period. This gives an indication of the result of the development team, how much work there is actually done. The thresholds for the star rating are somewhat arbitrary here since what is expected churn depends on so many factors.

4.12.2 Metrics

This factoid is based on `churnInTwoWeeks.historic` and `churnInTwoWeeks`.

4.12.3 Example natural language output

"In the last two weeks the churn was <churn>, this is <more,less> than the normal <churnMedian> in the last six months. The trend is <going down, stable, going up>."

4.12.4 Code

```
1  Factoid churnVolume(rel[datetime, int] churnHistory = {}, int churn = 0) {
2    churnSlope = historicalSlope (churnHistory, 6);
3    churnMedian = historicalMedian (churnHistory, 6);
4
5    msg = "In the last two weeks the churn was <churn>, this is <if (churn > churnMedian) {>more<}else{>less<}>
6          'than the normal <churnMedian> in the last six months.
7          'The trend is
8          '<slopeText(churnSlope, "going down", " stable ", "going up")>.";
9
10   if (churn > 0) {
11     if (churnMedian < 100) {
12       return factoid (msg, \two ());
13     }
14     else if (churnMedian < 500) {
15       return factoid (msg, \three ());
16     }
17     else {
18       return factoid (msg, \four ());
19     }
20   }
21   else {
22     return factoid (msg, \one ());
23   }
24 }
```

4.13 Commit locality

4.13.1 Motivation

The locality of a commit indicates a design quality: if commits are local it means the system is easy to fix and easy to extend. If commits are not local, this means that either a big restructuring is taking place or that the design has problems. Commit locality is related to coupling and cohesion metrics, so if commits are non-local we advise to have a look at the coupling and cohesion results as well. filesPerCommit.historic

4.13.2 Metrics

We used `filesPerCommit` and its historic version.

4.13.3 Example natural language output

- "Commits are usually local to a single file."
- "Commits usually include between 5 and 10 files."

4.13.4 Code

```
1  Factoid commitLocality(rel[datetime day, map[loc, int] files] filesPerCommit = {}) {
2     counts = [ d[f] | <_, map[loc, int] d> <- filesPerCommit, loc f <- d];
3     if (counts == []) {
4        throw undefined("No commit data available.", |tmp:///|);
5     }
6
7     med = median(counts);
8
9     if (med <= 1) {
10      return factoid("Commits are usually local to a single file.", \four());
11    }
12    else if (med <= 5) {
13      return factoid("Commits are usually local to a small group of files.", \three());
14    }
15    else if (med <= 10) {
16      return factoid("Commits usually include between 5 and 10 files.", \two());
17    }
18    else {
19      return factoid("Commits usually include more than 10 files.", \one());
20    }
21 }
```

5 Summary and Conclusions

5.1 Summary

In Deliverable 3.3 we have reported on:

- Important design considerations for language agnostic metrics;
- The modeling of all unknown languages as the "generic" language, such that the entire architecture for source code analysis in OSSMETER which is based on the M3 model and the Rascal language can be reused;
- A number of language agnostic source code metrics;
- The development of "factoids" based on language agnostic source code metrics;
- The development of "factoids" based on source code activity metrics which were reported on in Deliverable 3.2.

We refer to Deliverable 3.4 for language specific metric providers and the factoids they enable. In this document you will also find how we detect which language model to generate.

5.2 Conclusions

Language agnostic metric providers and factoids are limited in number, but they are valuable and easy to interpret across different languages. We've included the source code activity metrics in this report because they too are language agnostic.

References

[1] Paul D. Allison. Measures of inequality. *American Sociological Review*, 43:865–880, December 1978.

[2] O. Arafat and D. Riehle. The comment density of open source software code. In *Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*, pages 195–198, May 2009.

[3] Ronald M. Baecker and Aaron Marcus. *Human Factors and Typography for More Readable Programs*. ACM, New York, NY, USA, 1989.

[4] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. The goal question metric approach. In *Encyclopedia of Software Engineering*. Wiley, 1994.

[5] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 368–, Washington, DC, USA, 1998. IEEE Computer Society.

[6] Xiliang Chen, Alice Yuchen Wang, and Ewan Tempero. A replication and reproduction of code clone detection studies. In *Proceedings of the Thirty-Seventh Australasian Computer Science Conference - Volume 147*, ACSC '14, pages 105–114, Darlinghurst, Australia, Australia, 2014. Australian Computer Society, Inc.

[7] Khaled El Emam, Saïda Benlarbi, Nishith Goel, and Shesh N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Transactions on Software Engineering*, 27(7):630–650, 2001.

[8] Ilja Heitlager, Tobias Kuipers, and Joost Visser. A practical model for measuring maintainability. In *Proceedings of the 6th International Conference on Quality of Information and Communications Technology*, QUATIC '07, pages 30–39, Washington, DC, USA, 2007. IEEE Computer Society.

[9] Ted Honderich. *The Oxford Companion to Philosophy*. Oxford University Press, 2005.

[10] Capers Jones. *Software Assessments, Benchmarks, and Best Practices*. 2000.

[11] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, July 2002.

[12] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[13] Cory Kapser and Michael W. Godfrey. "cloning considered harmful" considered harmful. In *Proceedings of the 13th Working Conference on Reverse Engineering*, WCRE '06, pages 19–28, Washington, DC, USA, 2006. IEEE Computer Society.

[14] Cory J. Kapser and Michael W. Godfrey. "cloning considered harmful" considered harmful: Patterns of cloning in software. *Empirical Softw. Engg.*, 13(6):645–692, December 2008.

[15] Paul Klint, Tijs van der Storm, and Jurgen Vinju. EASY Meta-programming with Rascal. In *Post-proceedings of GTTSE'09*, volume 6491 of *LNCS*, pages 222–289. Springer, 2011.

[16] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Proc. SCAM'09*, pages 168–177. IEEE, 2009.

[17] Davy Landman, Alexander Serebrenik, and Jurgen Vinju. Empirical analysis of the relationship between CC and SLOC in a large corpus of Java methods. In *30th IEEE International Conference on Software Maintenance and Evolution, ICSME 2014*, 2014.

[18] Steve McConnell. Layout and style. In *Code Complete*, pages 399–492. Microsoft Press, 1993.

[19] Steve McConnell. Self-documenting code. In *Code Complete*, pages 453–491. Microsoft Press, 1993.

[20] Bogdan Vasilescu, Alexander Serebrenik, and Mark van den Brand. By no means: A study on aggregating software metrics. In *Proceedings of the 2Nd International Workshop on Emerging Trends in Software Metrics*, WETSoM '11, pages 23–26, New York, NY, USA, 2011. ACM.