Project Number 318772

D3.4 – Language-Specific 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

Document Control

Version	Status	Date
0.1	Initial draft	11 September 2014
0.5	Updated draft	22 October 2014
0.8	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 an integrated architecture for source code analysis and source code activity analysis was presented. The current document adds multi-language support to this architecture.
- 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 a metrics which work for specific programming languages, namely Java and PHP. For language agnostic quality analysis we refer to Deliverable 3.3. 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 here, but not in Deliverable 3.3. 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 (Section 1).

In this deliverable we present:

- A brief summary of motivation and challenges for Task 3.4;
- The infra-structure to support metrics for PHP;
- A build-information (e.g. class paths) recovery tool for Java, called BMW;
- Metrics including their motivation in GQM terminology [3]:
  - Shared metric providers between Java and PHP
  - Metric providers specific for Java
  - Metric providers specific for PHP
- An overview of the use of the above metrics in the OSSMETER quality model through factoids.
- A status update of the full requirements table relevant for WP3.

This document is also good to read if you are a user of OSSMETER.

1 Introduction

The goal of this deliverable is to produce metric providers which report on the quality of source code using language specific source code analyses.

Language specific source code quality analyses are based on source code metrics which rely on concrete models of the syntax and semantics of a particular programming language. This knowledge about the language is expensive to implement and highly useful when applied in the context of metrics. The reason is the source code quality is intimately linked to the language in which the code is written. High level abstract concepts such as separation of concerns, coupling and cohesion, but also low level concepts like size and control flow have language specific interpretations.

For example, one language may import names of modules (compilation units) transitively into a namespace while the other language might not. The impact of a local change is radically different between the two different models so coupling metrics are affected by this semantic difference. Another example, one language may have short-circuit semantics [5] for the boolean conjunctive and disjunctive operators (&& and ||), while the other might not. The number of test cases needed to exercise both operands of these operators is thus different between these two languages.

In the code of the metrics of this deliverable, these differences will not be very explicit. The "magic" is mostly hidden in the construction of a common intermediate model, called M3 in the shape of the abstract syntax trees and the results of name resolution. This does not imply that metric implementations are directly reusable between programming languages. In this project we have tried to make reuse as explicit as possible such that no accidental errors are made in interpreting metrics for the wrong programming language semantics. By accidental metric reuse, the two earlier examples of scoping and short-circuiting could go very wrong.

1.1 Roadmap to Deliverable 3.4

- In the remainder of this section we discuss a number of general design consideration for source code metrics. These are relevant for appreciating the details of the metrics in the other sections;
- Section 2 describes how we generalized the OSSMETER platform to handling multiple programming languages at the same time and to open extensibility with more programming languages. This also includes the descriptions of the Java front-end and the PHP front-end;
- Precise language Java analysis needs compilation, and compilation needs a class paths. In Section 3 we describe how we infer class paths and how we retrieve necessary libraries from the internet;
- All language specific metric providers are motivated and described in Section 4;
- In Section 5 we motivate and describe the factoids which are based on the language specific metrics;
- The requirements on WP3 which were produced in Deliverable 1.1 are repeated and reported on in Section 6.
- Section 7 summarizes Deliverable 3.4.

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 5 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) [10]. 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 [27].

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 [39], 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 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 [21]. 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

[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

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

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).

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 [26, 25], 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/`

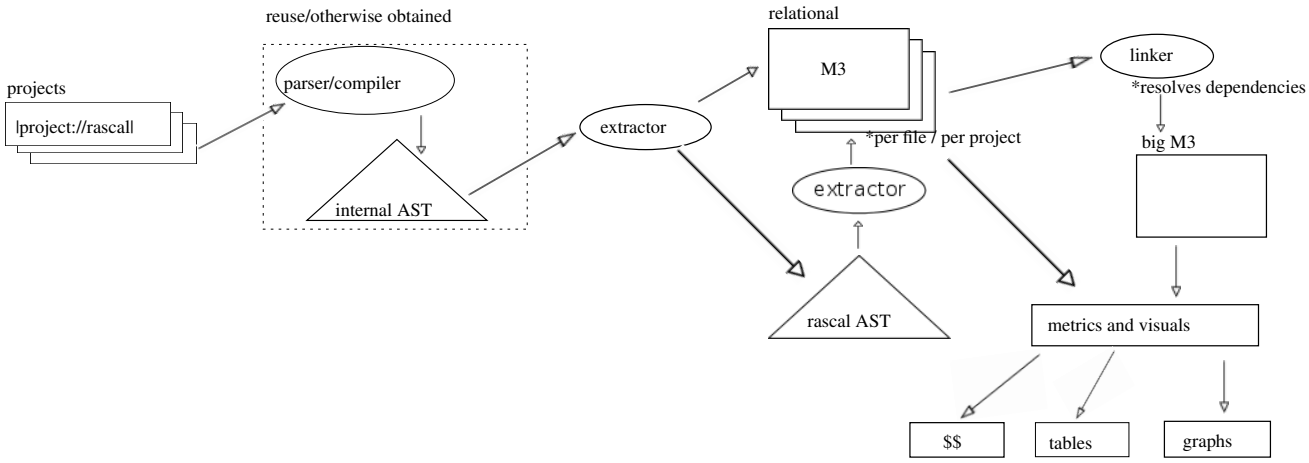

Figure 1: Extracting information from source code into the M3 model, and then using the M3 model for metrics

.

2 Multi-language support for source code quality analysis

For the current deliverable an important contribution to the OSSMETER platform is multi-language support. Having the need for full support for three languages (Java, PHP and the generic language[5]) we have generalized the platform for handling any amount of programming languages.

The general design for WP3 is depicted in Figure 1. For supporting multiple languages we need:

- Language specific parsing and name/type analysis, labelled “model extractors”;
- Intermediate models: M3 relations and abstract syntax trees;
- Metrics to work on these models and ASTs: metric providers;
- Factoids to summarize the metrics as star ratings and natural language.

The OSSMETER platform now features the following variation points with respect to language specific metric support:

- Some model extractors are configured to be active or not in a particular installation of OSSMETER. This configurability is handled via a model extraction OSGI extension point (Section 2.1).
- Some model extractors are applicable for some files and projects and not others. This variability is handled by the model extraction algorithm and the intermediate storage model for all M3 models (Section 2.2);
- Metric providers are applicable on the results of particular model extractors. This variability is managed by meta-data annotations on the metric providers (Section 2.2).

Details regarding the two different model extractors for Java and PHP are discussed in Section 2.3 and Section 2.4 respectively.

[5]For a description of the generic language see Deliverable 3.3.

```
1  @M3Extractor{myLanguage()}
2  rel[Language, loc, M3] myLanguageExtractor(
3    // identifies the project:
4    loc project,
5    // what happened between now and the previous version? for incremental extraction and caching
6    ProjectDelta delta,
7    // where to find the actual files of the project, sorted per unique repository source root:
8    map[loc repos, loc folders] checkouts,
9    // where to store temporary files if necessary, sorted per unique repository source root:
10   map[loc repos, loc folders] scratch) {
11    ... // build the model here
12   return ...; // return per file a tuple <language, file ,m3 model>
13 }
```

Figure 2: Template M3 extractor function with meta data.

```
1  @ASTExtractor{myLanguage()}
2  rel[Language, loc, AST] myLanguageParser(
3    // identifies the project:
4    loc project,
5    // what happened between now and the previous version? for incremental extraction and caching
6    ProjectDelta delta,
7    // where to find the actual files of the project, sorted per unique repository source root:
8    map[loc repos, loc folders] checkouts,
9    // where to store temporary files if necessary, sorted per unique repository source root:
10   map[loc repos, loc folders] scratch) {
11    ... // parse the files here, and annotate the ASTs if needed
12   return ...; // return per file a tuple <language, file, AST model>
13 }
```

Figure 3: Template AST extractor function with meta data.

2.1 Model extraction

A new platform extension "model extractor" introduces an OSGI extension point for registering a model extractor. A model extractor encapsulates the parser, name analysis and type analysis which is needed to translate source code files for a particular languages into M3 models.

Model extractors are written in the Rascal programming language mostly, but can use any other implementation language which can be connected through the JVM [17].

An OSGI extension point `-ossmeter.rascal.extractor-` is used to tag all bundles which contain a model extractor. The `RascalManager` class (see Deliverable 3.2) locates all functions tagged `@M3Extractor` and `@ASTExtractor`. Template AST and M3 extractors are depicted in Figure 2 and Figure 3.

To add a new language to the platform is to:

- Create an OSGI bundle, which is also an Eclipse Rascal project, with one Rascal file in it and tagged with the `extractor` extension point.

- Add a Rascal module with the two aforementioned function templates implemented
- Include a declaration for the new language name in a separate module which may be imported by the metrics.

```
1 data Language(str version="") = myLanguage();
```

- If necessary write library utility functions on top of the M3 and AST models for use within the metrics.

2.2 Model extraction algorithm

The platform support for managing the aforementioned model extractors is implemented in the `RascalManager` class which was explained earlier in Deliverable 3.2. The model extractors are called indivually directly by the `IMetricProvider` instances.

The `RascalManager` and `IMetricProviders` implement the following steps together:

1. When the platform boots, `RascalManager` detects and collects all extractor functions;
2. When the first `IMetricProvider` is executed which declares to need a model or an AST for a specific language, it calls *all available extractors* on *all available files*;
3. The common abstract superclass of all Rascal-based `IMetricProviders` caches the extracted models per day (since the whole OSSMETER platform is also synchronized per day). Note that implementations of `IMetricProvider` for specific Rascal metric functions are instantiated automatically by the platform (Deliverable 3.2).
4. All models are passed as an argument to all metric providers which need it as a table (relation) holding the project identifier, the language, the file for which the model is and the actual model.

We decided to call all extractors on all files in step 2 for a number of reasons:

- Sometimes file extensions are not enough to decide which extractor to call. Perhaps parsing is needed or some deeper exploration to find out which language or which language version is used in a particular file;
- To facilitate independent evolution: adding more and newer versions of extractors for existing languages without breaking the previous versions;
- To facilitate extracting the model for the generic language (see Deliverable 3.3), which is always present even if language specific model providers are present.

In step 4, each metric always receives all models which were successfully extracted, so it can decide for itself which model to use and which model to ignore. In the case of the generic model and a language specific model, several metrics use both.

Note that in step 3 all models of all files are stored in memory. We use a weak hash table for this purpose. In case the platform runs out of memory it can throw away the models and they will be computed again from scratch when asked for in the next metric. This behavior is useful for larger projects where the maximum memory footprint is reached when performing deep code analyses.

2.3 Java model extractor

The Java model extractor has been developed for earlier deliverables 3.1 and 3.2 already and is the basis for testing and developing the platform for WP3. Here we repeat the motivation for supporting Java and briefly the design of the extractor.

2.3.1 Motivation

Java is a popular language in the closed source software industry but even more so in the open-source industry. According to the Tiobe language popularity index[6], since 2002 Java has never been other than number 1 or number 2 on the list of most used and most discussed languages. The github analysis by Berkholz[7] also shows that more and more Java projects are being introduced and more and more Java code is being written in the open-source community.

Java is also the primary language on the Eclipse platform and it has good open compiler front-end which can be used as a basis for our model extractors. It is also the primary language our project partners are interested in.

2.3.2 Implementation

As described in Deliverables 3.1 and 3.2 the primary model extractor for Java is based on the Eclipse JDT (Java Development Toolkit) compiler. This open compiler is written for use in the Eclipse IDE and as such stays very close to the source code.

Open compilers which are written for compilation purposes only often takes short-cuts in their initial models to ease processing further downstream. This may introduce inaccurate readings. One example would be the introduction of automatic calls to super(). The compiler should do this, but the initial model which is produced by the parser should not already have this call. The call would be counted in metrics while the programmer would not have typed any in. For this and other such reasons the JDT's compiler is a good choice.

The bridge between Rascal source locations, relations and abstract data-types is a very detailed mapping written in Java against the JDK's API. The current implementation has gone through a number of testing stages and re-designs and bug fixes and has currently been stable for about a year.

One of the open parameters for the front-end is the compiler class-path which is needed to resolve dependencies to get an accurate semantic model of the source code of each Java compilation unit. We discuss how we obtain the values for this parameter per project and per version day in Section 3.

[6] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html
[7] http://redmonk.com/dberkholz/2014/05/02/github-language-trends-and-the-fragmenting-landscape/

2.3.3 Conclusion

We developed a stable, trustworthy, complete and easy-to-measure intermediate model for analyzing Java code in Rascal and developing metrics for the OSSMETER platform. Versions of this Java M3 and AST model were used in:

- The Software Evolution courses in 2012/2013, 2013/2014 and 2014/2015 at Universiteit van Amsterdam;
- The research for this paper: "Empirical analysis of the relationship between CC and SLOC in a large corpus of Java methods." [27];
- The work described in "M3: An Open Model For Measuring Code Artifacts" [22].
- The masters thesis by Maria Gouseti, "A General Framework for Concurrency Aware Refactorings" at Universiteit van Amsterdam.

2.4 PHP model extractor

2.4.1 Motivation

The second language we support and report on for this deliverable is PHP. The choice for a second language was not so easily decided as for the first language Java. The prime alternatives were have been C and C++. It was clear from our Java experience that a choice had to be made for a single language.

On the one hand, supporting the C language would have been possible for sure and could have been motivated from a popularity argument in the open-source community. On the other hand it would not have taught us much about the extensibility and reuse features of the OSSMETER platform since the syntax and semantics could be seen as highly related to and a simplification of Java. One big hurdle in analyzing C is the pre-processor though [12], which makes it a lot harder to implement accurate software metrics. C does have good open compilers available for it nowadays with accurate first-stage parsing and name analysis [28], so for the future we may consider building an OSSMETER extractor based on this.

A step up in language complexity is C++. The language is popular in specific parts of the open-source community, especially in the Linux and BSD communities. However, the state of open compilers for C++ is not so clear as for C and the language is immensely complex. Given the experience with Java and PHP, and an open-source compiler for C++, we are confident an M3 model extractor could be build. This would require a considerable investment (we estimate at least four full man-months of work).

Having considered these alternatives, the next languages we considered were popular scripting languages, such as PHP, Ruby and Javascript. The reason to consider these is not only their relative popularity in the open-source community but also that they are radically different from Java. Adding one of these languages gives the platform a good reason to show its flexibility and generality.

We chose in the end for PHP because is popular, significantly different from but also in some way comparable to Java, and not-to-hard to support:

- PHP is used by 82% of the website who's server-side language we know[8]. About 98% of these sites use PHP 5, and 2% use PHP 4.
- PHP ranks high enough, 4th on TIOBE's community index[9], and it is 4th github language for new (non-forked) projects after javascript (1), ruby (2) and java (3)
- PHP is an interesting language with many dynamic features but also object-oriented so still comparable to Java in some way. This provides opportunity to see how we may share metrics and metric provider code between Java and PHP.
- We found a good open-source parser for PHP, written in PHP and forked and extended for our purposes[10].

[8] http://w3techs.com/technologies/details/pl-php/all/all
[9] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html
[10] https://github.com/cwi-swat/PHP-Parser

2.4.2 Implementation

Since PHP does not have a static type system we should focus on finding a pure PHP parser first. We found PHPParser[11] to be stable and complete and relatively easy to adapt. It is written in PHP itself. To serialize an abstract syntax tree and parse it back into Rascal, we have extended the parser with a general AST visitor and then implemented one visitor for the serialization[12]. Alternatives to PHPParser were the PHC compiler and the PHP Eclipse tools. The first was deemed too big a third-part dependency (with additional demands on deployment) and the second had some serious bugs in AST construction.

The integration into OSSMETER is done via the aforementioned AST. The PHP-analysis library which was developed then analyses this AST to infer a basic M3 model per file. This analysis includes a basic PHP name analysis stage which is used in the metrics described below in Section 4.

Both the PHPParser and the PHP-Analysis components are released as open-source projects from OSSMETER simply because they are used for other purposes as well, such as implementation of security analyses and PHP refactoring tools.

2.4.3 Conclusion

WP3 features detailed models for PHP quality analysis based on an open source parser implemented in PHP itself and its mapping to Rascal M3 and AST data-types.

The PHP models were used in the following instances already:

- "Static, lightweight includes resolution for PHP" [19].
- "An empirical study of PHP feature usage" [18].
- Universiteit van Amsterdam, masters' thesis projects:
  - PHP type inference (to appear);
  - Ioana Rucareanu, "PHP: Securing Against SQL Injection";
  - Dimitrios Kyritsis, "PHP re-factoring: HTML templates";
  - Chris Mulder, "Reducing Dynamic Feature Usage in PHP Code".

The M3 models are used for different purposes outside of OSSMETER as well. As more languages become available outside of the OSSMETER context in the Rascal eco-system, they will also be useful for OSSMETER.

2.5 Summary

We reported on the design and implementation of multi-language support for language specific and language agnostic metric providers. We introduced a new front-end for the PHP language next to the already existing Java front-end.

[11] https://github.com/nikic/PHP-Parser
[12] https://github.com/cwi-swat/PHP-Parser

Based on the models provided we can report on the specific metrics and factoids in Section 4 and Section 5. However, for Java there was a major bottleneck to solve before we could apply the platform to arbitrary projects on `github.com` and `eclipse.org`, namely the inference of the correct classpaths (Section 3).

3 Inferring build parameters for Java projects

The Java front-end will produce ASTs and M3 relations as described in Deliverables 3.1 and 3.2 even if an incomplete class path has been passed to the compiler. However, the more correct the class path the more names can be resolved correctly and the more accurate the deeper (object-oriented) metrics can be. If class path elements are missing or the jar files can not be found on disk, then some of the metrics will "gracefully" degrade in accuracy.

One example of this effect would be depth of inheritance (DIT) [35]. Consider class A which extends class B, but class B is found in a library which can not be resolved. The depth of the inheritance of class A depends on the depth of the inheritance of class B for which the definition is unknown. The DIT metric for class A is at least 2, but it may be more. DIT is not the only metric depending on such information.

Meta information which is needed to correctly build an M3 model is:

- The build class path: jar files and the java run-time environment;
- The source path: all roots of the package hierarchy where compilation units can be found;
- The Java compiler version compliance setting.

For WP3 we targeted to resolve as many dependencies as possible in order to get the most accurate metrics as possible. For this we applied two different mechanisms, one primary and one fallback:

- **META**: Read build meta-information files and retrieve the information from there;
- **DISK**: Infer the meta information from analyzing the files in the checkouts from the version repository system.

The meta information recovery mechanism is applied for every day that a project is analyzed. The reader should realize that the result of the recovery is different possibly for every day and also that the method of recovery may be different. The reason is that projects sometimes switch between different build systems. The build systems we currently support directly are:

- No build system: serviced by the **DISK** mechanism;
- Eclipse projects: serviced by the **META** mechanism;
- Maven projects: serviced by the **META** mechanism;

3.1 The DISK mechanism

The disk mechanism goes through the following steps to infer enough meta-information:

1. Find all files with a `.java` extension, read their `package` declarations to match with the folder structure. The process goes through the file system recursively and stops recursion when the first match is found between the folder structure and the package declarations.
2. Find all files with a `.jar` extension and add them to the classpath;
3. Assume a java version heuristically, based on the year in which the current version of the project was written.

The benefit of this mechnism is that it is fast and works on all checkouts. The pitfall is that if some (unknown) build system is used anyway, then this mechanism is expected to recover significantly less information than could have been recovered from the meta information. Nevertheless, the **DISK** mechanism is always a fallback in case all else fails.

3.2 The META mechanism

To support Eclipse and Maven projects we have to read and interpret the meta information in particular files typically found in Maven projects such as `pom.xml`, and files typically found in Eclipse projects:

- `.project`, declares which builders to use;
- `META-INF/MANIFEST.MF`, which other Eclipse projects and plugins this project depends on;
- `build.properties`, where source files
- `.classpath`, Java class path definition file including external jar file dependencies;
- `.settings/**`, arbitrary builder setting files.

3.2.1 Maven

Maven is a Java dependency and build tool management system. Next to this the Maven central repository is a global repository of versioned jar files where external dependencies can be downloaded from: `http://search.maven.org/`. Using Maven, programmers define dependencies on local or external jar files and if their project layout adheres to a standard definition the Maven file will be very small and elegant.

If the project layout does not adhere to the standard, the Maven developer can define the exceptions. This is done via so called plugins. Using a Maven plugin the XML definitions of a Maven pom.xml file can be extended in arbitrary ways. The semantics of these definitions is defined by the plugin implementations. Plugin implementations are downloaded on demand from the maven central repository as well. This implies that the only way to find out what Maven is doing is by running it.

Note that as a consequence developers do not distribute jar files they depend on with the source code of their Maven projects either: these too are downloaded on demand. This implies that OSSMETER metrics will also download jars from the Maven grand central when needed.

We use one particular Maven plugin as well for OSSMETER, which can print out the actual class path as a list of colon separated absolute file names to (already downloaded) jar files. When we run this plugin, other plugins are triggered and eventually the path is printed to a file on disk. This file is then read back and the contents is passed as parameter to the Java M3 front-end.

The benefit of having this Maven front-end is that most Maven projects will build with very accurate semantic models. The pitfalls is that Maven does arbitrary things to compute the class path, including even sometimes calling the Java compiler and performing a lot of disk I/O in general.

Because of the I/O activity of Maven, it is advisable to install OSSMETER on RAM disks or very fast harddrives, and also to select a file system which is good with a large amount of very small files (such as Reisser FS). Furthermore, it is good to cater for a fast internet connection such that the maven central repository can be reached easily.

3.2.2 Eclipse Java build system

For Eclipse, unfortunately, there exists no straightforward headless Java API for recovering the information we need from these files. The source code which parses and processes this kind of information is intimately tangled with the rest of Eclipse and can not be separated so easily. To parse the files directly ourselves would be an option, but the question is what the semantics of Eclipse really is. This semantics includes OSGI bundles, P2 update sites behavior, the market place, etc.

In other words we could never simulate Eclipse's build system from scratch. Luckily there exists the Tycho project[13]. Tycho is a Maven plugin which ties into a headless version of Eclipse and runs the actual Eclipse code somehow to execute its deployment and build systems. Tycho is configured using extensions to the Maven configuration file. If we can configure Tycho well, then we can use the previous Maven solution to acquire accurate meta information regarding the build parameters of every Eclipse project.

To be able to use Tycho we need to have a `pom.xml` file which can trigger Tycho's behavior. To bootstrap such a situation Tycho comes with its own plugin for inferring a `pom.xml` file from an Eclipse project.

The bridges to Tycho and Maven are all built using calls to command line tools. Even though both are written in Java there exists no proper API for retrieving the same information. Also, both Tycho and Maven are *immense* and *immensely complex* systems. While Maven is running P2 repositories will be contacted as well as the maven repository.

If any unexpected error occurs during either pom file generation or calling maven to collect the class path, then we fall back to the **DISK** mechanism.

3.3 Conclusion

To retrieve the proper class paths for a Java project was much more of a challenge than we anticipated. Especially building Eclipse projects is a huge challenge. We would like to acknowledge the great work of the Maven project and the Tycho project without which we could not have done this. We also acknowledge that this part of OSSMETER has room for improvement due to the large amount of disk and network activity generated by downloading files from P2 repositories and the maven central repository.

This part of WP3 is essential for getting the most accuracy out of the object-oriented quality metrics for Java code. The code is reusable outside of OSSMETER and can be found at `https://github.com/cwi-swat/rascal-java-build-manager` under the EPL license.

[13] `http://www.eclipse.org/tycho/`

4 Language specific metric providers

We have reported on prototypes and experiments with language specific metric providers earlier in Deliverables 3.1 and 3.2. Here we report on all of those and some more but do not include their aggregation into factoids. Please find the factoids in Section 5. Together the metrics and the factoids implement all the requirements for WP3 as identified in Deliverable 1.1. A table with all requirements can be found in Section 6.

4.1 Common Java & PHP metrics

4.1.1 Cyclomatic Complexity

Cyclomatic complexity is a very popular metric which, on executable code units, has a cross-language comparable interpretation. Per unit, the metric counts the number of linearly independent control flow paths. This metric is a good estimate for the number of test cases needed to obtain test coverage of a method or function [31]. In anything we would advise to interpret the metric in this light: it indicates how hard one has to work, per method, to test it.

We have investigated thoroughly the issues regarding interpretation of the cyclomatic complexity metric. For sure, the metric is advertised to measure understandability, but this interpretation has serious issues [40]. From this study we learned that there are many methods which would have really high cyclomatic complexity yet are easy to understand and at the same time there exist very hard to understand methods which have a low cyclomatic complexity.

Still, in the daily work of software quality analysis the metric is avidly used. The SIG maintainability model [15] measures, for example, the number of lines of code in particular groups of methods which have more than a healthy number of linearly independent control flow paths. Using this aggregation method we can find out if a significant part of the systems is “overly complex” and therefore hard to test.

At the same time there exists a body of literature of results claiming that cyclomatic complexity correlates linearly with lines of code [8, 36, 13, 41, 33, 38, 4, 29, 37, 30, 14, 32, 10, 23, 16, 24]. The conclusion would be that it is irrelevant to measure this and we can stick with the easier to explain lines of code. We did a large scale study using OSSMETER technology, trying to confirm this linear correlation [27]. The results of this study show however that cyclomatic complexity is pretty much an orthogonal metric to lines of code and therefore a valuable addition to the OSSMETER metric suite.

We now have for both PHP and Java the following metrics based on CC:

- Plain cyclomatic complexity per method or function: CCPHP, CCJava.
- Weighted method count (WMC) for PHP and Java classes, which is the sum of the CC of the methods in a class.
- Factoids for both languages which aggregate according to the SMM [15], see Section 5.

The code for both languages is depicted in Figure 4 and Figure 5 for Java and PHP, respectively. From this code you can see what it takes to make a metric comparable between two languages. These metrics

```
1  int countCC(Declaration ast) {
2    count = 1;
3    visit (ast) {
4      case \foreach(_, _, _): count += 1;
5      case \for(_, _, _, _) : count += 1;
6      case \if(_, _, _)     : count += 1;
7      case \for(_, _, _)    : count += 1;
8      case \if(_, _)        : count += 1;
9      // the default case has a different name, so we only count real cases:
10     case \case(_)         : count += 1;
11     case \while(_, _)     : count += 1;
12     case \do(_, _)        : count += 1;
13     case \catch(_, _)     : count += 1;
14
15     // infix operators have a funny shape in this AST format:
16     case \infix(lhs, "||", rhs, extendedOperands):
17       count += 1 + size(extendedOperands);
18     case \infix(lhs, "&&", rhs, extendedOperands):
19       count += 1 + size(extendedOperands);
20
21     // for embedded declarations we have already counted the nested
22     // methods (because visit is bottom-up),so lets remove them again.
23     // this should not happen too often, so no performance penalty.
24     case \newObject(_, _, _, Declaration nested)     : count -= countCC(nested);
25     case \newObject(_, _, Declaration nested)        : count -= countCC(nested);
26     case \declarationExpression(Declaration nested) : count -= countCC(nested);
27   }
28   return count;
29 }
```

Figure 4: Implementation of CC for Java.

consider not just a different accidental shape of the abstract syntax trees, but truly deep semantic differences between both languages.

Also the use of the **`visit`** statement makes it easy to compute the metric because it abstracts over the nodes that have to be recursively traversed but do not contribute to the count. This automation entails for Java an over-count in case of nested anonymous classes which needs to be corrected. Source code metrics are full of subtleties.

4.1.2 Object-oriented metrics

The OO characteristics are most interesting. Are object-oriented modeling features used in a way that is beneficial? Or is object-orientation simply an extra layer of accidental complexity for a particular project? Can we see this from measuring the code?

There exists a large body of literature on metrics for object-oriented systems. Not every metric is easy to interpret, especially when aggregated to the project level. We have chosen to be complete and implement the full CK [7] and MOOD [9] metrics suites for object-oriented measurement, next to

```
1  int countCC(list[Stmt] stats) {
2    count = 1;
3    visit (stats) {
4      case \do(_, _)                  : count += 1;
5      case \for(_, _, _, _)           : count += 1;
6      case \foreach(_, _, _, _, _)    : count += 1;
7      case \while(_, _)               : count += 1;
8      case \if(_, _, elseIfs, _)      : count += 1 + size(elseIfs);
9      case \tryCatch(_, catches)      : count += size(catches);
10     case \tryCatchFinally(_, catches, _): count += size(catches);
11   }
12   return count;
13 }
```

Figure 5: Implementation of CC for PHP.

what was required (Section 6). When necessary, the availability of these metrics, also in a historic perspective, will allow the user of OSSMETER to drill down into causes when for example events or trends are detected in other more shallow metrics (code activity and code volume). This drilling down assumes the user understands these metrics and how to interpret them.

Although the set of factoids can be relatively easily extended to aggregate and combine any of the provided metrics, we highly recommend *not to mix and match or combine* the aforementioned metrics using basic arithmetic or statistical functions. The reason is that for most metrics the unit of measure or even the conceptual dimension is unknown. Combines such metrics into a single number is meaningless and may lead to misrepresentation of the quality of an open-source project. Briand et al. [6] also argue that the different metrics accidentally may measure the same dimension so combining them may give more weight to one dimension than another, again accidentally. Just do not computationally combine these metrics. Otherwise they can be very useful in finding where problematic code may be, especially in combination with source code activity and committer statistics.

In the aggregation to factoids we are much more selective for simplicity's sake (Section 5).

The metrics provided for both PHP and Java are:

1. Abstractness
2. Reuse ratio
3. Specialization ratio
4. Depth of inheritance tree
5. Number of children
6. Coupling between objects (our preferred "Coupling" metric [20])
7. Data abstraction coupling
8. Message passing coupling

```
1  map[loc, int] Ce(rel[loc package, loc \type] pkgTypes, rel[loc depender, loc dependee] typeDeps) {
2      packages = domain(pkgTypes);
3
4      otherPkgDeps = typeDeps o invert(pkgTypes) − invert(pkgTypes);
5
6      return (p : ( 0 | it + 1 | t ← pkgTypes[p], otherPkgDeps[t] != {} ) | p ← packages);
7  }
8  map[loc, int] Ca(rel[loc package, loc \type] pkgTypes, rel[loc depender, loc dependee] typeDeps) {
9      otherPkgDeps = typeDeps o invert(pkgTypes) − invert(pkgTypes);
10
11     typesDependingOnPackage = pkgTypes o invert(typeDeps);
12
13     return (p : size(typesDependingOnPackage[p]) | p ← domain(pkgTypes));
14 }
15 map[loc, int] CBO(rel[loc type1, loc type2] typeDeps, set[loc] allTypes) {
16     coupledTypes = typeDeps + invert(typeDeps);
17
18     return ( t : size(coupledTypes[t]) | t ← allTypes );
19 }
```

Figure 6: Metric implementations for afferent, efferent coupling and coupling between objects which are reused between CC and PHP, uses relational calculus operators, reducers and map comprehensions for conciseness.

9. Afferent coupling
10. Efferent coupling
11. Instability
12. Response for class
13. Method inheritance factor
14. Attribute inheritance factor
15. Method hiding factor
16. Attribute hiding factor
17. Polymorphism factor
18. Lack of cohesion in methods
19. Lack of cohesion in methods “4” (our preferred “Cohesion” metric [34])
20. Tight class cohesion
21. Loose class cohesion
22. Number of methods

23. Number of attributes

These metrics are where the M3 model shines. Most metrics implementations here share a core implementation by reuse. The mapping is made by mostly by the front-end mapping into the M3 meta model and not by the metric implementations (contrast this with the Cyclomatic Complexity method).

The code of three example implementations of reusable metrics is included in Figure 6. The other MOOD and CK metrics are very similarly implemented.

4.2 Java specific metrics

4.2.1 Style anti-patterns

The popular CheckStyle[14] and PMD[15] are open-source projects report on the occurrence of a large number of "code smells" [11]. Code smells are called "anti-patterns", typical idioms that may either indicate bad quality or expected to cause problems later (such as bugs, inefficiency or hard-to-maintain code). The requirements of Deliverable 1.1. mention a number of the categories of anti-patterns detected by both CheckStyle and PMD.

In line with our intent to have the semantics of all metrics "in hand" to control, adapt and extend it, we re-implemented a large set of anti-patterns, inspired by both CheckStyle and PMD. These anti-patterns, although valuable are too numerous to present to the user of OSSMETER as metrics one-by-one. Instead we grouped the anti-patterns using the ontology provided by the CheckStyle documentation. In this documentation anti-patterns are grouped by related type.

We count per category:

- the total number of occurrences of anti-patterns in the code per category;
- the count per file;
- the spread of the occurrences through the project.

4.2.1.1 Interpreting style anti-pattern metrics The interpretation of these metrics is difficult because every anti-pattern is different and thus they are largely incomparable amongst each other. Adding up the occurrences of anti-patterns in code as a metric is questionable. Nevertheless, such a number when compared between projects may indicate something important. If one project would have few issues while another has a lot, this can confirm a suspicion made by other metrics. We advise to use style metrics in this way, rather than dismissing a project based solely on some number of arbitrary style violations. Always use these metrics to compare projects and never to pinpoint problems with the quality of a single project.

4.2.1.2 Categorized anti-patterns The current implementation of style checks in OSSMETER has the following anti-patterns in the following categories. Each category is divided in several sub-categories which are given by the CheckStyle documentation:

[14] `http://checkstyle.sourceforge.net`
[15] `http://pmd.sourceforge.net`

Category	Error Proneness	Inefficiencies	Understandability
	blockCheck	string	namingConvention
	coding		metric
			sizeViolation
			imports
			classDesign

This table would be easy to extend when more anti-pattern are required. We do recommend not to introduce many more categories as including the categories into a quality model may be hard to motivate. Some categories provided by CheckStyle and PMD overlap with metrics we already provide elsewhere, such as code clones. We have not re-implemented or included them here again in order to maintain some kind of orthogonality between different metric providers.

For each of the sub-categories we implemented the following patterns:

Subcategory	Anti-patterns
blockCheck	EmptyBlock, NeedBraces, AvoidNestedBlocks
coding	AvoidInlineConditionals, MagicNumber, MissingSwitchDefault, SimplifyBooleanExpression, SimplifyBooleanReturn, StringLiteralEquality, NestedForDepth, NestedIfDepth, NestedTryDepth, NoClone, NoFinalizer, ReturnCount, DefaultComesLast, FallThrough, MultipleStringLiterals
string	StringInstantiation, StringToString, InefficientStringBuffering, UnnessaryCaseChange, UseStringBufferLength, AppendCharacterWithChar, ConsecutiveLiteralAppends, UseIndexOfChar, InefficientEmptyStringCheck, AvoidStringBufferField
namingConvention	AbstractClassName, ClassTypeParameterName, ConstantName, LocalFinalVariableName, LocalVariableName, MemberName, MethodName, MethodTypeParameterName, PackageName, ParameterName, StaticVariableName, TypeName
metric	BooleanExpressionComplexity, ClassDataAbstractionCoupling, ClassFanOutComplexity
sizeViolation	ExecutableStatementCount, FileLength, LineLength, MethodLength, ParameterNumber, MethodCount
imports	UnusedImports, ImportOrder, ImportControl
classDesign	VisibilityModifier, FinalClass, MutableException, ThrowsCount

As you can see these are quite a number of checks to implement. A first implementation was rather nice and clean and traversed the AST once for every anti-pattern. This was too slow. We re-implemented the whole set as one big traversal where data is collected on the way down and counted on the way back.

A code example of one of the anti-pattern checks is depicted in Figure 7.

4.2.1.3 Coverage over CheckStyle and PMD

The output of our Rascal-implemented style checks which should overlap with the semantics of CheckStyle have been tested using the XML machine interface of CheckStyle.

```
1  list [Message] emptyBlock(\catch(_, body), list [Statement] parents , M3 model) =
2          isEmptyStatement(body) ? [blockCheck("EmptyBlock", body@src)] : [];
3
4  list [Message] emptyBlock(\do(body, _), list [Statement] parents , M3 model) =
5          isEmptyStatement(body) ? [blockCheck("EmptyBlock", body@src)] : [];
6
7  list [Message] emptyBlock(\while(_, body), list [Statement] parents , M3 model) =
8          isEmptyStatement(body) ? [blockCheck("EmptyBlock", body@src)] : [];
9
10 list [Message] emptyBlock( \for(_, _, _, Statement body), list [Statement] parents , M3 model) =
11         isEmptyStatement(body) ? [blockCheck("EmptyBlock", body@src)] : [];
12
13 list [Message] emptyBlock( \for(_, _, Statement body), list [Statement] parents , M3 model) =
14         isEmptyStatement(body) ? [blockCheck("EmptyBlock", body@src)] : [];
15
16 list [Message] emptyBlock( \try(Statement body, _), list [Statement] parents , M3 model) =
17         isEmptyStatement(body) ? [blockCheck("EmptyBlock", body@src)] : [];
18
19 list [Message] emptyBlock( \try(body, _, Statement \ finally ), list [Statement] parents , M3 model) =
20         isEmptyStatement(body) ? [blockCheck("EmptyBlock", body@src)] : [];
21
22 list [Message] emptyBlock(\ initializer (Statement initializerBody ), list [Statement] parents , M3 model) =
23         isEmptyStatement( initializerBody ) ? [blockCheck("EmptyBlock", initializerBody@src )] : [];
24
25 list [Message] emptyBlock(\if(_, Statement thenBranch), list [Statement] parents , M3 model) =
26         isEmptyStatement(thenBranch) ? [blockCheck("EmptyBlock", thenBranch@src)] : [];
27
28 list [Message] emptyBlock(\if(_, Statement thenBranch, Statement elseBranch ), list [Statement] parents
29                                 , M3 model) =
30         (isEmptyStatement(thenBranch) ? [blockCheck("EmptyBlock", thenBranch@src)] : []) +
31         (isEmptyStatement(elseBranch) ? [blockCheck("EmptyBlock", elseBranch@src)] : []);
```

Figure 7: Block check anti-pattern implemented in Rascal.

Strictly speaking, from CheckStyle we re-implemented 50 out of 132 anti-pattern checks. The code is much more short than its implementation in Java for CheckStyle, which is due to the domain specific nature of Rascal as a meta programming/scripting language.

From PMD we implement 55 out of 229 checks provided by PMD. PMD provides a lot of checks which overlap with other metrics of OSSMETER. For this overlap we do not have statistics.

4.2.1.4 Conclusion We provide a large number of anti-patterns, as inspired by the OSSMETER requirements, CheckStyle and PMD. Adding more patterns is easy.

The aggregation of style violation instances (occurrences of anti-patterns) is based on a categorization found in the CheckStyle documentation and a further abstraction into classes of interest: efficiency, understandability, and error proneness.

4.2.2 Test coverage

The next important set of metrics is around the quality of the tests. We focus completely on the concept of automated *unit testing* and we assume all unit tests are implemented using the JUnit framework, version 3 or 4[16].

If not most of the code of a project is exercised by its automated unit tests, then we have an indication that the code is probably not so well tested. This does not necessarily mean the code is buggy, but it does mean the code will be hard to maintain (refactor, fix, or extend).

Code coverage from running unit tests is a dynamic property. Tools like Clover[17] measure dynamically which lines of code or which branches have been executed while running all the tests. Code coverage is then the percentage of lines which have been executed divided by the total lines of executable lines of code.

We can not run the code in OSSMETER for a number of reasons; for security's sake, for the sake of efficiency, and for the sake of completeness. It may be impossible to automatically run the tests on some days in the history of a project accidentally, but still we want to know how the tests would cover the code if they could have been run.

Heitlager and Visser describe how one could statically approximate dynamic code coverage by estimating the methods which are called starting from the JUnit test methods [2]. The results of their experiment show how an accuracy of around 70% can be achieved. Moreover, the trend of static code coverage does follow the trend of actual code coverage for the systems they tested on (see Figure 9).

4.2.2.1 Implementation The static coverage metric we implemented works directly on the M3 models we extracted. It uses call information and method overloading to compute a transitively closed call graph. Combined with the lines of code metric we can then compute how much of the code is covered by the test.

Inaccuracy in the metric is caused directly by dynamic dispatch. We assume every method which could be called, will be called by the test.

The lions share of the implementation of the metric is depicted in Figure 8.

4.2.2.2 Conclusion Static test coverage is a valuable metric for assessing how well the code is being tested over time. Particularly over time it is interesting to see trends or spikes, which is what we will focus on in constructing the factoid (see Section 5).

This metric is unique for OSSMETER as far as we know, next to its use by the authors of the original paper [2] at the Software Improvement Group[18].

[16] `www.junit.org`
[17] `https://www.atlassian.com/software/clover/overview`
[18] `http://www.sig.eu`

```
real estimateTestCoverage(rel[Language, loc, M3] m3s = {}) {
  m = systemM3(m3s);
  implicitContainment = getImplicitContainment(m);
  implicitCalls = getImplicitCalls(m, implicitContainment);
  fullContainment = m@containment + implicitContainment;

  liftedInvocations = {};
  for (<caller, callee> ← m@methodInvocation) {
    if (isMethod(caller)) {
      liftedInvocations += { <caller, callee> };
      continue;
    }
    inverseContainment = m@containment<1,0>;
    if (caller.scheme == "java+initializer")
      caller = getOneFrom(inverseContainment[caller]);
    if (caller.scheme == "java+class" || caller.scheme == "java+anonymousClass"
        || caller.scheme == "java+enum") {
      for (meth ← fullContainment[caller], meth.scheme == "java+constructor")
        liftedInvocations += { <meth, callee> };
    }
  }
  fullCallGraph = liftedInvocations + implicitCalls + m@methodOverrides<1,0>;
  allTestMethods
    = getJUnit4TestMethods(m) + getJUnit4SetupMethods(m);
  interfaceMethods
    = { meth | <entity, meth> ← m@containment, isMethod(meth), isInterface(entity) };
  reachableMethods
    = { meth | meth ← reach(fullCallGraph, allTestMethods), meth in m@declarations<0> }
    − allTestMethods − interfaceMethods;
  totalDefinedMethods
    = (0 | it + 1 | meth ← m@declarations<0> − allTestMethods − interfaceMethods, isMethod(meth));
  return (100.0 * size(reachableMethods)) / totalDefinedMethods;
}
```

Figure 8: Implementing the static code coverage metric.

4.2.3 Advanced language features

As "advanced language features" we interpret features of Java which are harder to understand when used and may lead to hard-to-maintain code. There are of course also advanced language features which make the code easier to understand and maintain (such as generics). Given the discussion in the introduction, we focus on contra-indicators of quality here as well.

The simple metric counts occurrences of the use of:

- type parameter wildcards
- type parameter upperbounds and lowerbounds
- union types
- anonymous classes

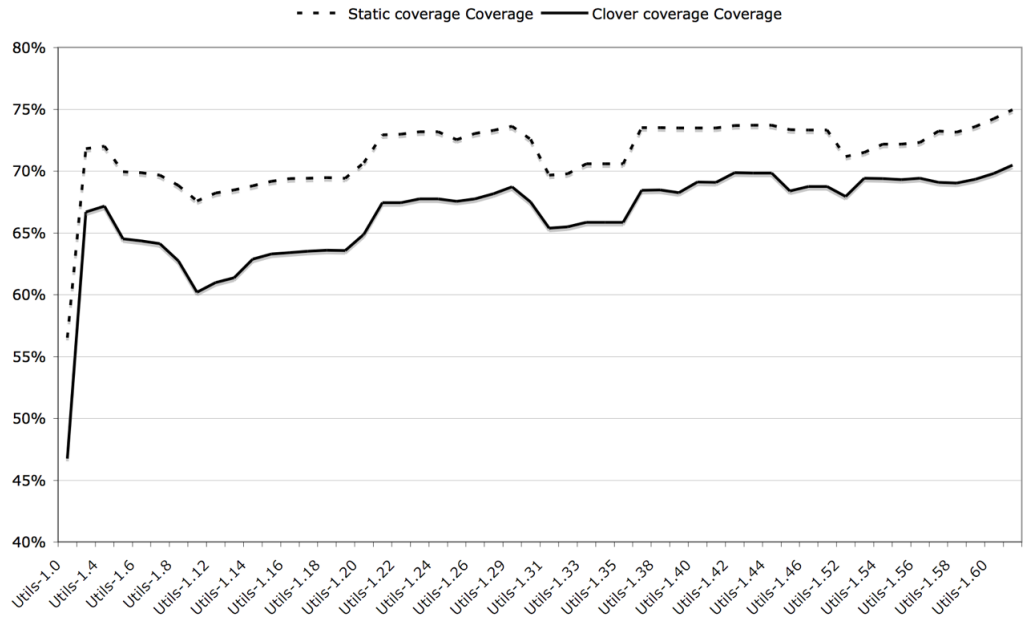

Figure 9: Graph taken from Heitlager and Visser [2], showing accuracy of static code coverage over time for a single project.

.

The metric could be seen as a part of the style metrics discussed earlier, but e decided to single it out because it was such an explicit requirement in Deliverable 1.1.

4.3 PHP specific metrics

PHP is a dynamically typed object-oriented programming language, unlike Java which is statically typed. This means that no accurate name resolution can be done statically for PHP and also that some OO metrics can both over-approximate or under approximate the actual values. We add some metrics here to be able to get an impression of this effect on the CK and MOOD metrics. The metrics in their own right also have a qualitative interpretation.

4.3.1 Name specificity

In a running PHP program it will be dynamically decided which name leads to calling which function or method, which name leads to including which other file, which name refers to which attribute name, etc.

For the current metric we are not talking about dynamic expressions which compute these names, but rather static/literal names for which statically it is unknown what the name actually points to.

The amount of such "unclear" names and the number of possible matches for target code artifacts is what we want to measure. A large number in these cases indicates code for which it is hard to predict for a human being which code will be executed and which data will be referred to. Not knowing which code is executed is a threat to the security of PHP websites for example, and also a threat to fixing bugs consistently, etc.

We consider the following separate categories of names and create histograms of how ambiguous the names are:

- type names
- method names
- attribute names (fields)
- included filenames

The analysis necessary for computing these histograms is based on a single pass name resolution algorithm developed specifically for this purpose. It will resolve only names which have obviously only one candidate, and will report all candidates if there are more. Basically the name is searched for in the context of the project, in the right namespace, and some straightforward constraints are applied. For example, when searching for methods a minimal number of parameters must be present.

The better a PHP programmer makes use of explicit name spaces and the better the names are chosen (i.e. more unique), the fewer candidates will be generated by the above analyses.

It must be admitted that these are fairly new metrics and an experimental validation of their validity and their usefulness is still underway. Nevertheless we did investigate support for inaccurate name resolution in a large number of PHP projects and are trying to find solutions to statically resolve them efficiently [19].

For other dynamically typed languages, such as javascript and ruby we expect similar metrics to be useful.

4.3.2 Use of dynamic features

The power of PHP is also its Achilles' heel: dynamic typing. Similar to reflection in Java, where most static analyses have to give up, in PHP there exist a number of features where static analysis techniques have a hard time. These features do allow for powerful frameworks to be developed, but they also lead to security flaws and hard-to-understand code.

We investigated where and how dynamic features are used in PHP [18]. It appears the most harmful features are used in localized ways, but in sufficient amounts to be relevant.

We measure the following basic facts and correct for the project size factor later in the factoid:

- numberOfDynamicFeatureUses
- numberOfEvalCalls
- numberOfFunctionsWithDynamicFeatures

4.3.3 Missing libraries

PHP deployment is done usually by copying the right source code into the scope of the PHP interpreter. Sometimes this deployment strategy is visible in the version repository of the project itself (the dependencies are copied in). This makes projects look bigger than they really are.

Usually we expect that PHP projects do not include the code of third-party dependencies and this will lead to unresolved file includes. We estimate this number to give the user of OSSMETER an indication of the external dependencies. Adding external dependencies or including more version repository roots to complete the project may also change the name resolution scores described earlier.

4.4 Conclusion

We described all the basic metrics for Java and PHP that have been implemented for WP3. Their aggregation into factoids is described in the next Section 5 and the table of requirements, where also metrics are explicitly mentioned, is included in Section 6.

5 Language specific factoid descriptions

Factoids are a combination of natural language and a four star rating to summarize a qualitative aspect of an open-source project. The goal of a factoid is indeed to accurately reflect the state of the project and make different projects comparable.

We select the "right" metrics for inclusion into a factoid and are careful to aggregate in a manner which is faithful to the data and its expected distribution (see Section 1).

The goal is to have a small set of factoids which summarize a project, such that users can zoom into specific metrics later if they require more detail.

The current section lists all language dependent factoids. The language independent factoids are described in Deliverable 3.3.

5.1 OO Complexity

5.1.1 Motivation

A lot of classes with deep inheritance is a sign of bad design and hard to understand and maintain code. The code will suffer from the fragile base class problem and it may be hard for the developers to find out which code is executed when.

5.1.2 Metrics

We used depth of inheritance for both Java and PHP.

5.1.3 Example natural language output

"The percentage of <language> classes with a problematic inheritance
depth is <badPercentage>%."

5.2 Coupling

5.2.1 Motivation

Too high coupling is a sign of bad design, hard to independently evolve components and generally hard-to-understand code. According to Sahraoui, Godin, and Miceli [34] a CBO metric above 14 is a good threshold to pinpoint bad design. We measure the percentage of classes over this threshold.

5.2.2 Metrics

We use coupling between objects for this factoid. There are many more metrics for different kinds of coupling, but it is unknown whether or not they measure different things and how these metrics correlate. Therefore it is safer to focus on a single most general metric.

5.2.3 Example natural language output

"The percentage of <language> classes with problematic coupling is <badPercentage>%."

5.2.4 Code

```
1  Factoid Coupling(str language, map[loc, int] cbo) {
2          if (cbo == ()) {
3                  throw undefined("No CBO data", |file://|);
4          }
5
6          numClassesWithBadCoupling = ( 0 | it + 1 | c ← cbo, cbo[c] > 14 );
7          badPercentage = numClassesWithBadCoupling * 100.0 / size(cbo);
8
9          stars = four();
10
11         if (badPercentage > 20) {
12                 stars = \one();
13         }
14         else if (badPercentage > 10) {
15                 stars = two();
16         }
17         else if (badPercentage > 5) {
18                 stars = three();
19         }
20
21         txt = "The percentage of <language> classes with problematic coupling is <badPercentage>%.";
22
23         return factoid(txt, stars);
24 }
```

5.3 Cohesion

5.3.1 Motivation

We measure a lack of cohesion for all methods. Any class which has one or more methods with lack of cohesion should be refactored. If there exist many such classes, the project has a high technical depth and will be harder to understand and harder to change.

5.3.2 Metrics

We use LCOM4 for both PHP 4 and 5: lack of cohesion in methods "4".

5.3.3 Example natural language output

"The percentage of <language> classes with problematic cohesion is <badPercentage>%."

5.3.4 Code

```
1  Factoid Cohesion(
2          str language,
3          map[loc, int] lcom4) {
4          if (lcom4 == ()) {
5                  throw undefined("No LCOM4 data", |file: // /|);
6          }
7
8          numClassesWithBadCohesion = ( 0 | it + 1 | c ← lcom4, lcom4[c] != 1 );
9
10         badPercentage = numClassesWithBadCohesion * 100.0 / size(lcom4);
11
12         stars = four();
13
14         if (badPercentage > 20) {
15                 stars = \one();
16         }
17         else if (badPercentage > 10) {
18                 stars = two();
19         }
20         else if (badPercentage > 5) {
21                 stars = three();
22         }
23
24         txt = "The percentage of <language> classes with problematic cohesion is <badPercentage>%.";
25
26         return factoid(txt, stars);
27 }
```

5.4 Cyclomatic Complexity

5.4.1 Motivation

Good OO design should avoid complex control flow in the bodies of methods. When many method bodies become large and complex, we have a contra-indication for software quality and maintainability. We measure how many lines of code in methods are influenced by control flow complexity larger than 10 and larger than 100. These percentages can also be used to compare the understandability of methods between projects.

5.4.2 Metrics

Cyclomatic Complexity, aggregated in percentiles with specific thresholds [15].

5.4.3 Example natural language output

"The cyclomatic complexity footprint of the system's <language> code shows a <very high, high, moderate, low> risk. The percentages of methods with moderate, high and very high risk CC are respectively <perc1>%, <perc2>% and <perc3>%.";

5.4.4 Code

```
public Factoid CCFactoid(map[loc, int] methodCC, str language) {
  if (isEmpty(methodCC)) {
    throw undefined("No CC data available", |tmp://|);
  }

  thresholds = [10, 20, 50]; // moderate, high, very high

  counts = [0, 0, 0];

  for (m <- methodCC) {
    cc = methodCC[m];
    for (i <- [0..size(thresholds)]) {
      if (cc > thresholds[i] && (i + 1 == size(thresholds) || cc <= thresholds[i + 1])) {
        counts[i] += 1;
      }
    }
  }

  numMethods = size(methodCC);

  percentages = [ 100.0 * c / numMethods | c <- counts ];

  rankings = [ [25, 0, 0], [30, 5, 0], [40, 10, 0] ];

  stars = 1;

  for (i <- [0..size(rankings)]) {
    if (all(j <- [0..size(percentages)], percentages[j] <= rankings[i][j])) {
      stars = 4 - i;
      break;
    }
  }

  txt = "The cyclomatic complexity footprint of the system\'s <language> code
        'shows a <["", "very high", "high", "moderate", "low"][stars]> risk.
        'The percentages of methods with moderate, high and very high risk CC
        'are respectably <percentages[0]>%, <percentages[1]>% and
        '<percentages[2]>%.";

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

5.5 Understandability

5.5.1 Motivation

We observe common problems with hard-to-understand code and count the number of occurrences and the spread of these issues across the project. Examples are violated naming conventions, not enough use of spaces in the code, overly complex or large files and methods, and large coupling.

5.5.2 Metrics

Naming conventions: classNames, methodNames, AsbtractClassNames, ClassTypeParameter, LocalVariable, LocalFinalVariable, MethodTypeParameter, Imports: redundant, unused, star, illegal, SizeViolations: ExecutableStatements, FileLength, MethodLength, ParameterNumber, MethodCount, Class design violations: visibility, final class, mutable exception, throwscount, simple metrics: boolean complexity, ClassDataAbstractionCoupling, ClassFanOutComplexity,

5.5.3 Example natural language output

A number of different messages are generated, many combinations are possible, but we list some examples here: “Currently, there is hardly any understand code in this project and this situation is stable in the last six months” or “Currently, hard to understand code practices are widely spread throughout the project, but the situation has been improving over the last six months”.

5.5.4 Code

```
1  @metric{ understandabilityFactoid }
2  @uses=(" spreadOfUnderstandabilityIssues ":" spreadOfUnderstandabilityIssues "
3         ," filesWithUnderstandabilityIssues ":" filesWithUnderstandabilityIssues "
4         ," filesWithUnderstandabilityIssues . historic ":" filesWithUnderstandabilityIssuesHistory "
5         )
6  @doc{Explains what the impact of style violations is for the project .}
7  @friendlyName{Spread of style violations over files }
8  @appliesTo{java()}
9  Factoid understandabilityFactoid ( real spreadOfUnderstandabilityIssues = 0.0
10                                  , int filesWithUnderstandabilityIssues = 0
11                                  , rel[datetime, int] filesWithUnderstandabilityIssuesHistory = {}
12                                  )
13   = genericFactoid ("hard to read"
14                     , spread= spreadOfUnderstandabilityIssues
15                     , files = filesWithUnderstandabilityIssues
16                     , history = filesWithUnderstandabilityIssuesHistory );
17
18
19 private Factoid genericFactoid (str category
20                              , real spread = 0.0
21                              , int files = 0
22                              , rel[datetime, int] history = {}
23                              ) {
24    sl = historicalSlope ( history , 6);
25
26    expect1 = "";
27    expect2 = "";
28
29    switch (<sl < 0.1, -0.1 >= sl && sl <= 0.1, sl > 0.1>) {
30      case <true , _ , _ > :
31        expect1 = "and its getting worse in the last six months";
32        expect2 = "but issues have been spreading in the last six months";
```

```
33        case <_       , true    , _       > :
34          expect1 = "and this  situation  is  stable ";
35          expect2 = "but the  situation  is  stable ";
36        case <_       , _       , true    > :
37          expect1 = "but the  situation  is improving over the  last  six  months";
38          expect2 = "and the  situation  has been improving in  the  last  six  months"; }
39      }
40
41      switch (<spread > 0.5, spread < 0.2,   spread == 0.0>) {
42        case <_,_, true > :
43          return factoid ("Currently , there  is  no <category> code in  this  project
44                           '<expect1>.", \four ());
45        case <_, true , _> :
46          return factoid ("Currently , <category> code is  localized  to  a  minor
47                           ' part  of  the  project  <expect2>.", \three ());
48        case <true , _, _> :
49          return factoid ("Currently , <category> code  practices  are  widely  spread
50                           ' throughout  the  project  <expect1>.", \one ());
51        default           :
52          return factoid ("Currently , there  is  a  some small  amount of <category>
53                           '  code spread through  a  small  part  of  the  project
54                           '<expect2>.", \two ());
55      }
56  }
```

5.6 Error-proneness

5.6.1 Motivation

Common style errors are associated with misunderstandings and bugs, such as badly indented code, deeply nested control flow and overly complex boolean expressions. We collect a number of these issues and count how often they occur in a project. Are these issues wide spread throughout the project or do we have a minor number of exceptions? Have programmers been solving such problems or is more of this being introduced recently. This gives an indication of the future robustness of the system regarding bugs that could have been avoided.

5.6.2 Metrics

We used spreadOfErrorProneness, filesWithErrorProneness, fileWithErrorPronenessHistory, blockCheck, coding, AvoidInlineConditionals, MagicNumber , MissingSwitchDefault , SimplifyBooleanExpression , SimplifyBooleanReturn , StringLiteralEquality, NestedForDepth , NestedIfDepth , NestedTryDepth , ReturnCount DefaultComesLast , MultipleStringLiterals , etc.

5.6.3 Example natural language output

A number of different messages are generated, many combinations are possible, but we list some examples here: “Currently, there is no error pone code in this project and this situation is stable in the

last six months" or "Currently, error prone code practices are widely spread throughout the project, but the situation has been improving over the last six months".

5.6.4 Code

See also the code for the understandability factoid for the definition of the `genericFactoid` function.

```
1  Factoid errorProneFactoid ( real spreadOfErrorProneness = 0.0
2                            , int filesWithErrorProneness = 0
3                            , rel[datetime, int] fileWithErrorPronenessHistory = {}
4                            )
5    = genericFactoid ("error prone"
6                      , spread=spreadOfErrorProneness
7                      , files = filesWithErrorProneness
8                      , history = fileWithErrorPronenessHistory );
```

5.7 Inefficiencies

5.7.1 Motivation

Java beginners sometimes use simple coding style which quickly leads to very inefficient code. We track a number of such common "mistakes" and report on their frequency and spread as an indication of the maturity of the project.

5.7.2 Metrics

We used the following metrics: stringInstantiation, stringToString, inefficientStringBuffering, unnnessaryCaseChange, useStringBufferLength, appendCharacterWithChar, consecutiveLiteralAppends, useIndexOfChar, inefficientEmptyStringCheck, avoidStringBufferField

5.7.3 Example natural language output

A number of different messages are generated, many combinations are possible, but we list some examples here: "Currently, there is no obviously inefficient code in this project and this situation is stable in the last six months" or "Currently, obviously inefficient code practices are widely spread throughout the project, but the situation has been improving over the last six months".

5.7.4 Code

```
1  Factoid inefficientStringsFactoid ( real spreadOfInefficiencies = 0.0
2                                    , int filesWithInefficiencies = 0
3                                    , rel[datetime, int] filesWithInefficienciesHistory = {}
4                                    )
5    = genericFactoid ("inefficient string usage"
6                      , spread= spreadOfInefficiencies
```

```
7                   , files = filesWithInefficiencies
8                   , history = filesWithInefficienciesHistory );
```

5.8 Java Unit Test Coverage

5.8.1 Motivation

Unit testing is an important method of quality management. Next to counting the number of methods which are actually tested directly, we also statically approximate how much of the code in the entire project is exercised by the unit tests. Note that this (over-)approximation may look better than the real dynamic code coverage, but the trend can be trusted to indicate if its going better or worse with the project's unit tests.

5.8.2 Metrics

We used testOverPublicMethods, testCoverage, and TestCoverage.historic.

5.8.3 Example natural language output

"The percentage of methods covered by unit tests is estimated at <testCoverage>%. This situation is <stable|getting worse|improving> over the last six months. The estimated coverage of public methods is <testOverPublicMethods>%"

5.8.4 Code

```
1  Factoid JavaUnitTestCoverage(
2    real testOverPublicMethods = -1.0
3    , real testCoverage = -1.0
4    , rel[datetime, real] history = {}) {
5    sl = historicalSlope(history, 6);
6
7    expect = "";
8
9    switch (<sl < 0.1, -0.1 >= sl && sl <= 0.1, sl > 0.1>) {
10     case <true, _, _>:
11       expect = "The situation is getting worse in the last six months";
12     case <_, true, _>:
13       expect = "This situation is stable";
14     case <_, _, true>:
15       expect = "This situation is improving over the last six months";
16   }
17
18   if (testOverPublicMethods == -1.0 || testCoverage == -1.0) {
19     throw undefined("Not enough test coverage data available", |tmp://|);
20   }
21
22   stars = 1 + toInt(testCoverage / 25.0);
```

Compliance
Full
Partial
None

Table 1: Coding scheme for compliance.

Priority
SHALL
SHOULD
MAY

Table 2: Coding scheme for priority.

```
23
24    if ( stars > 4) {
25       stars = 4;
26    }
27
28    txt = "The percentage of methods covered by unit tests is estimated at
29           '<testCoverage>%. <expect>. The estimated coverage of public methods
30           ' is <testOverPublicMethods>%";
31
32    return factoid ( txt , starLookup[ stars ]);
33  }
```

6 Detailed requirements from Deliverable 3.1

We use the coding scheme shown in Table 1 and Table 2. These requirements have been identified early in the project and explained in Deliverable 1.1.

The requirements for WP3 have been reported on previously in Deliverable 3.1 as well, from a planning perspective. Here we add a report on the status quo in the fourth column:

ID	Requirement	Priority	Expected compliance	Status quo
13	Metrics for software quality shall be defined that are independent of any programming language (language-agnostic metrics).	SHALL	Full	Full
14	Fact extractors shall be available that extract from source code the facts that are needed for computing language-agnostic metrics.	SHALL	Full	Full for Java and PHP.
15	Language-specific metrics for software quality shall be defined for Java.	SHALL	Full	Full
16	The facts needed to compute language-specific metrics for Java shall be extracted.	SHALL	Full	Full
17	Language-specific metrics for software quality may be defined for other languages (PHP, Python, C).	MAY	Partial	Partial, only for PHP. No other languages.

18	The facts needed to compute language-specific metrics for other languages (PHP, Python, C) may be extracted.	MAY	Partial	Partial, only for PHP. No other languages.
19	Calculation of software quality metrics should, where possible, be the same across all languages and paradigms.	SHOULD	Full	Full: The M3 model enables this where possible.
20	Development activity shall be measured by the number of committed changes.	SHALL	Full	Full
21	Development activity shall be measured by the size of committed changes.	SHALL	Full	Full: for SVN and GIT
22	Development activity may be measured by the distribution of active committers.	MAY	Full	Full
23	Development activity may be measured by the ratio between old and new committers.	MAY	Full	Full
24	History of some metrics should be captured to summarize quality evolution during development.	SHOULD	Full	Full
25	A model shall be designed to represent quality and activity metrics.	SHALL	Full	Full
34	Provide a rating of the quality of code comments of the OSS project	SHALL	Partial	Full
35	Provide a well-structured code index for the OSS project	SHALL	Full	Full
36	Provide a rating of the use of advanced language features for the OSS project	SHOULD	Full	Partial: for clarity's sake we focused only on language features which may be considered contra-indicators for quality.
37	Provide a rating of the use of testing cases for the OSS project	SHALL	Partial	Full
38	Provide an indicator of the possible bugs from empty try/catch/finally/switch blocks for the OSS project	SHALL	Full	Full, included in style checks

39	Provide an indicator of the dead code from unused local variables, parameters and private methods for the OSS project	SHALL	Partial	Partial: in general this analysis is very expensive in terms of run-time and memory behavior and possibly too much work to enable to every programming language. For Java we have the warnings from the Java compiler in the M3 model which can serve as a useful proxy.
40	Provide an indicator of the empty if/while statements for the OSS project	SHALL	Full	Full, included in style checks
41	Provide an indicator of overcomplicated expressions from unnecessary if statements and for loops that could be while loops for the OSS project	SHALL	Full	Full, included in style checks
42	Provide an indicator of suboptimal code from wasteful String/StringBuffer usage for the OSS project	SHALL	Partial	Full, included in style checks
43	Provide an indicator of duplicate code by detecting copied/pasted code for the OSS project	SHALL	Full	Full
44	Provide an indicator of the use of Javadoc comments for classes, attributes and methods for the OSS project	SHALL	Full	Partial: basic fact extraction done.
45	Provide an indicator of the use of the naming conventions of attributes and methods for the OSS project	SHALL	Full	Full, included in style checks.
46	Provide an indicator of the limit of the number of function parameters and line lengths for the OSS project	SHALL	Full	Full: included in style checks
47	Provide an indicator of the presence of mandatory headers for the OSS project	SHALL	Partial. The implications of this requirements have to be further explored.	Partial, checking for headers and header consistency, not the content.
48	Provide an indicator of the use of packets imports, of classes, of scope modifiers and of instructions blocks for the OSS project	SHALL	Full	Full

49	Provide an indicator of the spaces between some characters for the OSS project	SHALL	Partial	Full
50	Provide an indicator of the use of good practices of class construction for the OSS project	SHALL	Partial	Full: included in the style checks.
51	Provide an indicator of the use of multiple complexity measurements, among which expressions for the OSS project	SHALL	Full.	Full: several orthogonal views provide indicators of this aspect.
52	Provide an indicator of the cyclomatic complexity for the OSS project	SHALL	Full	Full

7 Summary and Conclusions

This deliverable reported on the inclusion of a new language (PHP) to the platform, the generalization of the platform to arbitrary numbers of languages, the inference of build parameters for Java, all the language dependent metrics for both Java and PHP and the aggregation of these metrics into factoids.

The style checks, static approximation of test coverage and name resolution effectivity metrics are highlights in this deliverable which show the strength of doing a deep source code analysis.

The main issue with the platform is speed of acquiring the metrics, which is currently mostly limited by I/O and heap space. The platform scales out nicely, so we expect we can work around the limitations by adding more machines.

In Section 6 we reported how most of the requirements of Deliverable 1.1, with motivated exceptions, are satisfied including full satisfaction of a number of requirements which were expected only to be partially fulfilled.

References

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

[2] Tiago L. Alves and Joost Visser. Static estimation of test coverage. volume 0, pages 55–64, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

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

[4] Victor R. Basili and Barry T. Perricone. Software errors and complexity: an empirical investigation. *Communications of the ACM*, 27(1), 1984.

[5] Jan A. Bergstra and Alban Ponse. Proposition algebra and short-circuit logic. In Farhad Arbab and Marjan Sirjani, editors, *Fundamentals of Software Engineering*, volume 7141 of *Lecture Notes in Computer Science*, pages 15–31. Springer Berlin Heidelberg, 2012.

[6] Lionel C. Briand, Jürgen Wüst, John W. Daly, and D. Victor Porter. Exploring the relationship between design measures and software quality in object-oriented systems. *J. Syst. Softw.*, 51(3):245–273, May 2000.

[7] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[8] Bill Curtis, Sylvia B. Sheppard, and Phil Milliman. Third Time Charm: Stronger Prediction of Programmer Performance by Software Complexity Metrics. In *Proceedings of the 4th International Conference on Software Engineering*, ICSE '79, pages 356–360, Piscataway, NJ, USA, 1979. IEEE Press.

[9] F. Brito e Abreu. The mood metrics set. *ECOOP 95 Workshop on Metrics*, 1995.

[10] 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.

[11] Eva Van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *WCRE*, pages 97–, 2002.

[12] J. Feigenspan, M. Schulze, M. Papendieck, C. Kästner, R. Dachselt, V. Köppen, M. Frisch, and G. Saake. Supporting program comprehension in large preprocessor-based software product lines. *Software, IET*, 6(6):488–501, Dec 2012.

[13] Alan R. Feuer and Edward B. Fowlkes. Some Results from an Empirical Study of Computer Software. In *Proceedings of the 4th International Conference on Software Engineering*, ICSE '79, pages 351–355, Piscataway, NJ, USA, 1979. IEEE Press.

[14] Geoffrey K. Gill and Chris F. Kemerer. Cyclomatic Complexity Density and Software Maintenance Productivity. *IEEE Transactions on Software Engineering*, 17(12):1284–1288, December 1991.

[15] 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.

[16] Israel Herraiz and Ahmed E Hassan. Beyond lines of code: Do we need more complexity metrics? In *Making Software What Really Works, and Why We Believe It*, chapter 8, pages 126–141. O'Reilly Media, 2010.

[17] Mark Hills, Paul Klint, and Jurgen Vinju. Rlsrunner: Linking rascal with k for program analysis. In *International Conference on Software Language Engineering (SLE)*, LNCS. Springer, 2011.

[18] Mark Hills, Paul Klint, and Jurgen J. Vinju. An empirical study of php feature usage: a static analysis perspective. In Mauro Pezzè and Mark Harman, editors, *ISSTA*, pages 325–335. ACM, 2013.

[19] Mark Hills, Paul Klint, and Jurgen J. Vinju. Static, lightweight includes resolution for php. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 503–514, New York, NY, USA, 2014. ACM.

[20] Martin Hitz and Behzad Montazeri. Measuring coupling and cohesion in object-oriented systems. In *Proceedings of International Symposium on Applied Corporate Computing*, pages 25–27, 1995.

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

[22] A. Izmaylova, P. Klint, A. Shahi, and J. J. Vinju. M3: An Open Model For Measuring Code Artifacts. Technical Report arXiv-1312.1188, CWI, December 2013.

[23] Graylin Jay, Joanne E. Hale, Randy K. Smith, David P. Hale, Nicholas A. Kraft, and Charles Ward. Cyclomatic Complexity and Lines of Code: Empirical Evidence of a Stable Linear Relationship. *Journal of Software Engineering and Applications*, 2(3):137–143, 2009.

[24] Ahmad Jbara, Adam Matan, and Dorg G. Feitelson. High-MCC Functions in the Linux Kernel. *Empirical Software Engineering*, pages 1–38, 2013.

[25] 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.

[26] 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.

[27] 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.

[28] Chris Lattner. LLVM and Clang: Next generation compiler technology. In *The BSD Conference*, pages 1–2, 2008.

[29] H.F. Li and W.K. Cheung. An Empirical Study of Software Metrics. *IEEE Transactions on Software Engineering*, SE-13(6):697–708, June 1987.

[30] Randy K. Lind and K. Vairavan. An Experimental Investigation of Software Metrics and Their Relationship to Software Development Effort. *IEEE Transactions on Software Engineering*, 15(5):649–653, May 1989.

[31] Thomas J. McCabe. A complexity measure. In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, pages 407–, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[32] Michael B. O'Neal. An Empirical Study of Three Common Software Complexity Measures. In *Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing: States of the Art and Practice*, SAC '93, pages 203–207, New York, NY, USA, 1993. ACM.

[33] M. Paige. A metric for software test planning. In *Conference Proceedings of COMPSAC 80*, pages 499–504, October 1980.

[34] H.A. Sahraoui, R. Godin, and T. Miceli. Can metrics help to bridge the gap between the improvement of oo design quality and its automation? In *Software Maintenance, 2000. Proceedings. International Conference on*, pages 154–162, 2000.

[35] Frederick T. Sheldon, Kshamta Jerath, and Hong Chung. Metrics for maintainability of class inheritance hierarchies. *Journal of Software Maintenance*, 14(3):147–160, May 2002.

[36] Sylvia B. Sheppard, Bill Curtis, Phil Milliman, M. A. Borst, and Tom Love. First-year results from a research program on human factors in software engineering. In *AFIPS Conference Proceedings*, volume 48, pages 1021–1027, New York, NY, USA, June 1979.

[37] Martin Shepperd. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal*, 3(2):30–36, March 1988.

[38] Takeshi Sunohara, Akira Takano, Kenji Uehara, and Tsutomu Ohkawa. Program complexity measure for software development management. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 100–106, Piscataway, NJ, USA, 1981. IEEE Press.

[39] 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.

[40] Jurgen J. Vinju and Michael W. Godfrey. What does control flow really look like? eyeballing the cyclomatic complexity metric. In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE Computer Society, 2012.

[41] M. R. Woodward, M. A. Hennell, and D. Hedley. A measure of control flow complexity in program text. *IEEE Transactions on Software Engineering*, 5(1):45–50, January 1979.