Project Number 318772

D3.2 – Report on Source Code Activity Metrics

Version 1.0
16 June 2014
Final

EC 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**

Project Partner Contact Information

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

Contents

Document Control

Version	Status	Date
0.1	Initial draft	24 March 2014
0.2	Incomplete draft	3 April 2014
0.3	Complete draft	4 April 2014
0.4	Revised with a lot more detail	16 April 2014
0.5	References for activity metrics	30 April 2014
1.0	Bumped version to 1.0	16 June 2014

Executive Summary

This deliverable is part of WP3: Source Code Quality and Activity Analysis. It provides descriptions and initial prototypes of the tools that are needed for source code activity analysis. It builds upon the Deliverable 3.1 where infra-structure and a domain analysis have been investigated for Source Code Quality Analysis and initial language dependent and independent metrics have been prototyped.

Task 3.2 builds partly on the results of Task 3.1 and partly introduces new infra-structure. This includes:

- the extraction and analysis of the meta data of version control systems (VCS);
- the development of selected source code metrics from Task 3.1 over time.

The following initial measurements of VCS meta data were planned and have been executed, in the context of the SVN and GIT version control systems:

- Number of committed changes;
- Size of committed changes (churn);
- Number of committers;
- Activity distribution per committer (over files).

On top of this we explored analysis of the activity in terms of certain language-specific source code metrics from the previous Task 3.1:

- Number of changed, added, deleted methods and classes to experiment with language-specific activities.
- Measurement of evenness of distributions (Gini coefficient) of the metrics developed in Task 3.1, for the purpose of detecting trends and spikes.

The goal of these additional metrics is to start bridging the gap from code and VCS meta data metrics to the analysis requirements of the project partners.

What makes WP3 in OSSMETER special is its integrated infrastructure that provides a homogeneous view on languages, analyses and metrics. We generate metrics using high level (descriptive) code in the Rascal language.

In this deliverable we present:

- A brief summary of motivation and challenges for Task 3.2;
- A streamlined interface between the platform and the Rascal programming language;
- A mapping from an object-oriented VCS deltas model to a functional VCS delta model;
- Platform support for managing full working copies and source code diffs;
- A description of the rationale, design and implementation of the above metrics.

1 Introduction

A large part of the quality of an OSS project is perceived to be its activity. A young but inactive project may not be attractive. An old but over-active project may indicate future instability. For active projects, some traces of activity may indicate promise of good quality while other traces of activity may indicate risks. The general frame of mind is that of "Software Evolution" [18, 24]: software projects have a tendency to evolve towards being more and more complex, until they become unmanageable and they have to be decommissioned for being too costly to maintain.

The factors that influence the growth and complexity of software over time are plenty. For Task 3.2 we focus on the *effects* of these factors in *source code*. Making these effects measurable and enabling the platform to present these in concert with other aspects of OSS project activity. The basic questions are:

- Who has been doing what?
- Where and how is the code growing/shrinking?
- How is complexity distributed over the system and is this distribution changing?

The goal of metrics for these basic questions is to provide evidence of developer activity and its effect on source code quantity and quality. By human interpretation we can learn from this how active a project is being developed and maintained, and in which parts. We also can observe the effect of this activity in terms of trends and spikes in the basic indicators for quality from Task 3.1.

The resulting information enables the users of OSSMETER to uncover causal relationships by relating events on the time-axis. For example they could make the following observation: "the number of methods with large cyclomatic complexity is rising steadily over time, while at the same time the number of developers is increasing and their activity is more dispersed over the system than before". They might conclude that the project is growing out of control; after which they take appropriate action, such as considering alternative projects or investing in source code quality governance for the given project.

The OSSMETER project is not about automating the analysis of such causality, but all the more to provide an accurate overview based on which a human can make an adequate assessment. Thus OSSMETER will mainly safe time for the assessing individual, but may also enhance the correctness of an analysis task due to automating the large but mundane task of measuring and summarizing quality attributes of many versions of many open source projects.

In order to obtain the relevant properties we need:

1. The previous results from WP3, Task 3.1: the requirements and infra-structure for measuring source code quality;
2. The previous results from WP5: the OSSMETER platform;
3. Information and meta information from VCS systems, such as project deltas on the source code level and author information (here we integrate with the results from WP2);
4. Differencing of models and metrics produced in Task 3.1;

5. Metrics calculators that synthesize the extracted facts to the required metrics.

For this we largely used the Rascal meta-programming language[1] as well as OSSMETER services provided by the other work packages. We streamline the interface between the platform and the Rascal language, and we extend the platformto support checkout and differencing functionality on the source code level for different VCS providers.

1.1 Outline of Deliverable 3.2

Section 2 is an overview of WP3, its design decisions and the Goal-Question-Metric perspective.

Section 4 describes source code activity metrics, their rationale, related work and example application.

Sections 5, 6, and 7 describe infra-structural improvements and additions:

Section 5 streamlines the addition of new metrics and new languages to the platform.

Section 6 describe platform support for managing and differencing working copies.

Section 7 describes which meta data we extract from VCS systems and marshall to Rascal.

Sections 8 and 9 summarize:

Section 8 describes how the results of Task 3.2 map to the general requirements of OSSMETER.

Section 9 summarizes Deliverable 3.2 and identifies future work.

2 WP3 overview

In this section we reiterate some of the design and design decisions of WP3 that have been reported on earlier in Deliverable 3.1[38], as well as introduce the new subject of activity metrics. This will help the reader to position the contributions of the current deliverable which are presented in the following Section 4 on activity metrics and the infra-structural improvements made as described in Sections 5, 6 and 7. For convenience, the contents of this section literally repeats some material and illustrations from Deliverable 3.1 [38].

At the end of this section we present the whole WP3 from the Goal-Question-Metric perspective.

2.1 General perspective of WP3: source code and source code activity

To get an adequate overview of an open-source project, and its quality, certainly the source code is a key factor to take into account. The goal of WP3 is to focus on source code as a source of information and present it to the platform such that it can be combined with the other views on a project (community, bugs).

There are two main aspects to source code analysis and two main associated goals:

[1] http://www.rascal-mpl.org

- Get an overview of the quality of the source code of a particular version of the project (i.e. the current version, or that version that is associated with a particular release date).
- Get an overview of the activity of the development on the source code (i.e. where and how the software has been changed or is changing).

We should note that, on the one hand, absolute metrics of quality (i.e. to measure understandability) are hard to interpret or even impossible to interpret [11], but when these metrics are presented next to each other for comparison they start to make sense:

- We can compare projects or versions of the same projects (benchmarking [4]).
- We can spot trends or outliers [41].
- We can compare source code metrics with metrics on the other factors on open source project quality by aligning them on the time scale [6].

In other word, WP3 provides enabling fact extractors and metrics, but the real value will only show after we combine and integrate the metrics into the platform.

2.2 Technical Non-Functional Requirements

The main technical non-functional requirements that we distilled from the goals and the project partners requirements are the following:

- **FlexibleMetrics:** It should be easy to experiment with new metrics and new aggregrations of metrics. The reason is that there exist a plethora of metrics and metrics suites. It is hard to choose, but at the same time we will have to choose in order not to overwhelm the users. In the initial integration stages we expect to introduce new metrics, throw away old metrics and fine-tine existing metrics. So, a requirement is to easily add metrics and easily adapt them.
- **AddingLanguages:** It should be easy to add new programming languages to the platform. Adding good support for programming languages is difficult in itself and the system should allow the programmer to focus on the language and not on the platform while doing this.
- **MetricReuseAndConsistency:** It should be possible, in principle, to reuse metrics acros programming languages where possible, or to at least make it easy to see that the metrics are comparable (can be interpreted in the same way).
- **AddingVCS**: It should be possible to quickly add support for new types of version control systems.

2.3 Technical Functional Requirements

The WP3 deliverables mention support for a number of predescribed metrics and programming languages (see Section 8 for details). In summary we provide:

- Full syntax and static semantics support for the Java language.
- Prototypical support for the syntax and semantics of the PHP language.
- Prototypes of the language independent volume metrics (lines of code)

- Prototypes of language dependent volume, complexity and dependency metrics (like non-commented lines of code, cyclomatic complexity, coupling/cohesion metrics)
- A set of aggegration methods on top of the metrics.
- Prototypes of basic activity metrics based on VCS meta data.
- Prototypes of activity metrics based on source code differences.

2.4 Key design elements

2.4.1 Reusing abstract syntax trees

More details on specific metrics can be found in Section 4. Here we describe the rationale for WP3 to develop all actual metric computation from scratch on top of abstract syntax trees that are provided by third parties.

Firstly, we require arbitrary language dependent metrics to be computed by the platform. We believe this implies that full abstract syntax trees [1] are required for all supported languages. To obtain correct abstract syntax trees we need parsers, name resolvers and possibly also type resolvers for each language. Such grammarware [14] represents —per language— an enormous investment. Luckily there is a recent development in opening up API for compiler front-ends and therefore we intend to reuse these as much as possible. Examples are the Eclipse JDT, the Oracle open Java compiler and the Roslyn open C# compiler project. It must be noted that such projects represent an enormous investment in design, implementation, testing and maintenance and we are lucky to have access to these open-source projects.

2.4.2 Computing metrics from ASTs ourselves

On the metrics side, the story is rather different. There exist a plethora of open-source, freely available and non-freely available tools and platforms for computing metrics about source code. To name a few: SonarCube[2], SciTools Understand[3], Bauhaus[4], NDepend[5], Eclipse Metrics tools[6], OOMeter [2], Semmle[7], VizzAnalyzer[8], etc. Our argument for not reusing such "end products" is three-fold:

- The heavy lifting is in parsing, name analysis and type analysis anyway. Once you have an AST, computing a metric is a matter of traversal, projection and aggregation.
- Metrics are nor consistently defined in literature, nor implemented consistently in such tools. Sometimes within a tool, sometimes between tools there exist cumbersome differences. Because the semantics of source code metrics is often not precisely defined there exist all kinds of derivatives and interpretations which are not consistent with eachother [19]. The causes of such differences may be rather low brow, i.e. simple bugs or differences of interpretation, yet

[2] http://www.sonarqube.org/
[3] http://www.scitools.com/
[4] http://www.axivion.com/products.html
[5] http://www.ndepend.com/
[6] http://metrics.sourceforge.net/
[7] http://semmle.com/
[8] http://www.arisa.se/projects.php

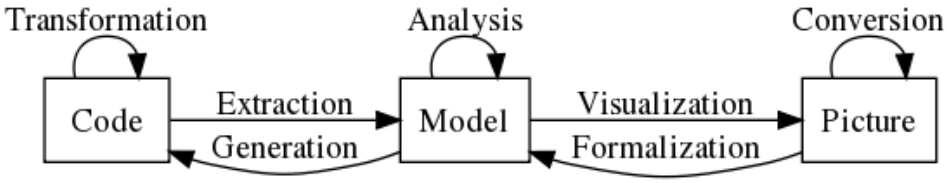

Figure 1: Meta programming domain as serviced by Rascal.

hard to gauge and hard to make explicit nevertheless. It would take a considerable effort to uncover these mundane differences, which do lead to differences in interpretation on the project level [19]. If we would add a new language to the platform via reusing an existing metric provider, this would have to be preceded by an evaluation of the detailed contents of the metric provider and its commonalities and differences with the other existing languages in the platform. This represents a considerable reverse engineerng effort in general.

- Bridging to existing metrics tools is more work than reimplementing the metric, given an AST model of the source code. Every tool comes with its own programming interfaces (or files and database interfaces), while adding a new metric is a matter of reusing the infra-structure of OSSMETER. Also, every tool comes with its separate deployment infra-structure which we would have to integrate as well. We should expect impedenance mismatches as well as configuration overhead there. It would simply be too costly for the expected return-on-investment.

We conclude that reusing metric tools would contradict all of our non-functional requirements (**FlexibleMetrics**, **AddingLanguages** and **MetricReuseAndConsistency**). On the other hand reusing language front-ends seems to be the right thing to do and inevitable.

2.4.3 Computing metrics over VCS meta data ourselves

For the metrics over VCS meta data a similar story exists. Most source forges and even the VCSs themselves provide direct access to meta data which can easily be queried and measured. Reusing the actual tools is more work than its worth.

At the same time, by controlling the definitions of the metrics ourselves we can strive for consistency among different providers. This is not an easy task as the meta models for each VCS are significantly different. The integrated meta model which unifies some of the features of different VCSs is one of the products of WP2 that we rely on in WP3.

Please find more detailed information on VCS meta data metrics in Section 4.

2.4.4 Rascal meta programming language

As explained above, all metrics share a similar design. They traverse source files, or abstract syntax tree (AST) representations thereof, then project out different aspects of the source code, count them and then aggegrate them at different levels of abstraction (line, method, class, file, package, system).

In fact measuring software is an instance of the Extract-Analyse-Synthesize paradigm [15] or Extract-Query-Present [43]. The goal of the Rascal meta programming language is to help programmers implement instances of this paradigm (see Figure 1):

- Extract information from source code to generate an arbitrary model;
- Analysis these models;
- Synthesize new source code or data or visualizations as a result.

Rascal was developed in 2009–2013 by the CWI team that is focusing on WP3 of OSSMETER.

As opposed to developing each metric in Java, Rascal code is expected to be in less than 10% of the code. This is primarily due to high level programming concepts such as automated traversal functions [40] and advanced forms of pattern matching [15], and builtin relational calculus primitives [13].

The code of example metric implemented in Rascal can be found in Section 4. Simpler code examples are included in the previous Deliverable 3.1 [38].

The syntax of Rascal is sufficiently close to other programming languages such as Java and Javascript, and therefore most students learn to program in it within a week.

2.4.5 M3 - meta model for metrics

To try and satisfy (mainly) requirement **MetricReuseAndConsistency** a common design decision is to introduce an intermediate model for artifacts extracted from source code.

The idea behind the meta model, inspired by work on [32], [17] and [35], is to represent language specific source facts as relations in the model. Each relation in the meta model represents information that we feel is required to either calculate a metric directly or provide additional information in some metric calculation. During the formulation of the meta model, we identified relations that are exhibited by many programming languages which led us to divide the meta model into two parts, namely the *Core M3 Model* and *Language-specific M3 models* that extend the core for different programming languages. In Sections 2.5 and 2.5.1 we present the M3 Core Model and in Section 2.5.2 we discuss language-specific models and focus on Java.

An essential ingredient of our proposal are *source code locations* that are based on Uniform Resource Identifiers (URIs)[9]. An essential part of an URI is the *scheme* that defines how the information pointed to by the URI has to be interpreted. Typical examples are `http`, `ftp`, and `mailto`. In our proposal we use URI schemes to encode source language and language-element that the URI points to. Examples are:[10]

- `|java+class://P2SnakesLadders/snakes/Game|`: defines Java as source language and denotes a *class* declaration.
- `|java+method://P2SnakesLadders/snakes/Game/setSquare(int,snakes.ISquare)|`: defines Java as source language and denotes a *method* declaration.
- `|java+field://P2SnakesLadders/snakes/Game/squares|`: defines Java as source language and denotes a *field* declaration.

[9] See `http://tools.ietf.org/html/rfc3986`.

[10] We use the syntax for source locations as provided by the `loc` datatype in the RASCAL language, see Section **??**.

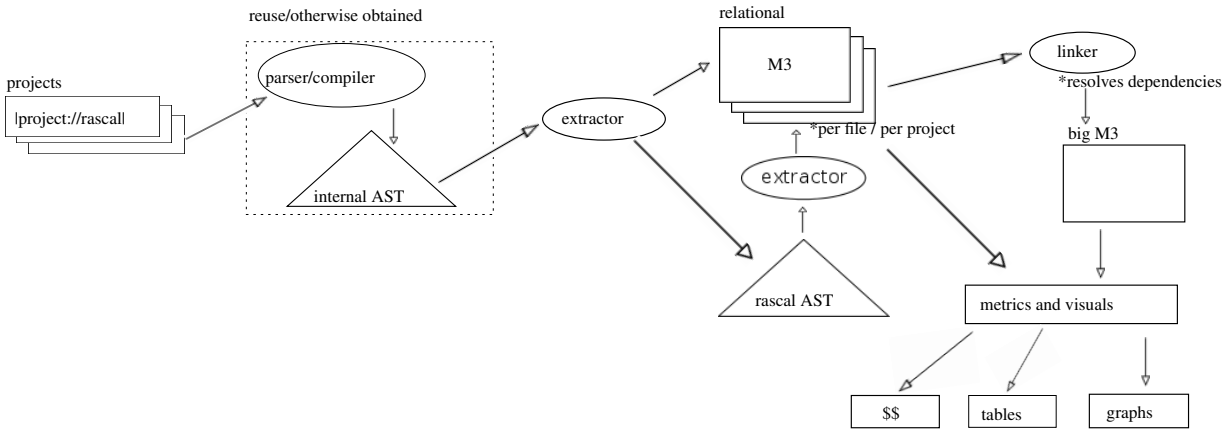

Figure 2: Reference M3 Architecture

- `|project:// P2SnakesLadders/src/snakes/Game.java|(3200,53,<130,41>,<130,94>)` shows how specific source coordinates (line and character information) can be included in source locations.

In addition to providing a completely general and extensible naming scheme, it is also straightforward to provide IDE support for the hyperlinking between extracted data and the original source code.

2.4.6 Architecture

Figure 2, shows how the M3 meta-model is created from source files and provides an indication of how the M3 model will be used by metrics/visualizations. The first task is to extract source facts in the form of an M3 model. We achieve this through reusing parser or compiler for each programming language encountered in the project. For each language supported in OSSMETER, we will need custom extractors. The extractor traverses through the internal Abstract Syntax Tree (AST) representation of each source file in the project, creating the relations in the model and later fuses them together as a single model for a project. The granularity of the M3 model can be per file or per project. In the case the granularity is set to be a project, we get a single M3 model even if multiple languages are found to be present with the difference between the languages being represented in the scheme of the URI's we use to represent source code elements.

The details of how the model can be used to implement metrics will be discussed in Section **??**.

2.5 The Core M3 Model

The definition of a model contains an "id" which defines the project/file for which the model is being created. We then add the following relations to the core model.

- **Declarations**. The declarations relation maps declared language elements to their physical location in a file.
- **Uses**. The uses relation maps uses of declared elements to their physical location in a file.

- **Containment**. The containment relation maps the elements that logically contain other elements to define a structure. For example, in Java files contain classes, classes contain fields and methods etc.
- **Names**. The names relation maps a declared name of an element to the qualified name that the compiler uses to uniquely identify it.
- **Documentation**. The documentation relation contains comments that are mapped to the to a physical location in a file.

The model also contains a list to store compiler generated error/warning messages.

2.5.1 The Core M3 Model in Rascal

The core relations of the M3 model are represented by a central (empty) model (`M3`) to which the relevant relations are attached (using Rascal's annotation operator `@`):

- `M3@declarations`: maps declarations to where they are declared. contains any kind of data or type or code declaration (classes, fields, methods, variables, etc. etc.).
- `M3@types`: assigns types to declared source code artifacts.
- `M3@uses`: maps source locations of usages to the respective declarations.
- `M3@containment`: what is logically contained in what else (not necessarily physically, but usually also).
- `M3@messages`: error messages and warnings produced while constructing a single M3 model.
- `M3@names`: convenience mapping from logical names to end-user readable (GUI) names, and vice versa.
- `M3@documentation`: comments and javadoc attached to declared things
- `M3@modifiers`: modifiers associated with declared things.

The definition of the M3 Core Model in Rascal:

```
1 data M3 = m3(loc id);
2 anno rel[loc name, loc src]                  M3@declarations;
3 anno rel[loc src, loc name]                  M3@uses;
4 anno rel[loc from, loc to]                   M3@containment;
5 anno list[Message messages]                  M3@messages;
6 anno rel[str simpleName, loc qualifiedName] M3@names;
7 anno rel[loc definition, loc comments]       M3@documentation;
```

The core model will only contain facts that are language independent. In addition to these basic facts that we expect to encounter in any type of programming language, we can easily add new relations to the model to incorporate any language independent facts we may find.

2.5.2 Language Specific M3 Model

Each language we provide support for in the OSSMETER platform will have it own M3 model. The language specific model will add new relations to the Core M3 model that will be relevant for metric calculation. As an example we provide Java M3 Model.

2.5.2.1 M3 Java Model The M3 Java Model extends the M3 Core Model and adds the following Java-specific relations:

- **Extends**. Contains the extends relation in Java between classes/interfaces.
- **Implements**. Contains the implements relation between classes and interfaces.
- **MethodInvocations**. Contains all the methods that are called from a method.
- **FieldAccess**. Contains any access to a Java field from anywhere in the source code.
- **TypeDependency**. Contains all the types that a Java element depends on.
- **MethodOverrides**. Contains the relations between methods and the methods that they override.

2.5.2.2 The M3 Java Model in Rascal The M3 Core Model is extended with the following Java-specific relations:

- `M3@extends`: classes extending classes and interfaces extending interfaces.
- `M3@implements`: classes implementing interfaces.
- `M3@methodInvocation`: methods calling each other (including constructors).
- `M3@fieldAccess`: code using data (like fields).
- `M3@typeDependency`: using a type literal in some code (types of variables, annotations).
- `M3@methodOverrides`: which method override which other methods

The definition of the M3 Java Model in Rascal:

```
1 extend m3::Core;
2 anno rel[loc from, loc to] M3@extends;
3 anno rel[loc from, loc to] M3@implements;
4 anno rel[loc from, loc to] M3@methodInvocation;
5 anno rel[loc from, loc to] M3@fieldAccess;
6 anno rel[loc from, loc to] M3@typeDependency;
7 anno rel[loc from, loc to] M3@methodOverrides;
```

An extract of the M3 model for a sample Java project can be found in Deliverable 3.1 [38].

2.5.3 Metrics based on M3 model

The advantage of the metrics based on the M3 model are two fold:

- Metric calculation is abstracted to a higher level and becomes language independent.
- We can change the granularity of the source code facts with ease. The M3 model can be created for a file, folder, project or any combination of these as required and the metric calculation will not need to change.

For use in the OSSMETER project, the platform (WP5) will host all the implemented metrics. The platform contains a Rascal interpreter that will first calculate the M3 model for each project and then pass the model to all the metrics available in its store. See Section 5 for details on how this bridge is designed. This approach allows the users with an easy means to implement their own metric. The user will only need to accept the M3 model (if needed), perform the tasks in the function and return the result back to the interpreter which will store it accordingly. The users only need to focus on what data the metric needs from the model and how it is calculated.

2.6 Goal-Question-Metric plan

Here we cast the work of WP3 from the GQM perspective for reference [5]. We list goals, their subordinate questions and the metrics that should answer them or provide indications. Note that:

- The GQM perspective inspired us to list more metrics than were required by the project plan.
- Some of the identified metrics in the GQM overview have not been implemented yet for this deliverable (see Section 8 for a status report).
- Some of the language dependent metrics have already been prototyped and reported on in the previous Deliverable 3.1, and we reiterate their motivation here.
- Producing the final language dependent and language independent source code metrics are for the next Tasks 3.3 and 3.4.
- Section 4 contains detailed information about the activity metrics for the current deliverable.

2.6.1 Goal: assess maintainability of the source code

The ISO/IEC 9126 norm and its successor ISO/IEC 25010 define the maintainability of a project as a set of attributes that bear on the effort to make modifications. It further divides this "ility" into these categories: analyzability, changeability, stability, testability, and compliance.

Maintainability is reported to be of utmost importance since a large part in the cost-of-ownership of software is to maintain it [29]. This includes perfective, corrective, preventive and adaptive maintenance as defined in ISO/IEC 14764. The dominant factor in maintainability is how easy it is for programmers to understand the existing design. If they can quickly navigate to points of interest or find the causes of a bug or assess the impact of a feature request, this makes a project easier to adapt to changing requirements and as such it is more reliable, not to mention more fun to contribute to.

The SIG maintainability model [12] is a "practical model for measuring maintainability" which tries and covers the ISO 9126/25010 attributes in a straightforward manner. The metrics used in this model you will also find in our lists. A key lesson from this work is to select metrics which can be related back to visible factors in the source code.

We should emphasize that for metrics per unit, such as methods and classes, we expect the platform to present the results as distribution histograms. In that manner two histograms for different versions of the same project or two histograms for latest version of two different projects can be compared.

Q: How large is the project?

- M: total lines of code. This basic language independent metric gives a indication of the size of a project [12].
- M: total non-commented, non-empty lines of code. This language dependent metric produces a more accurate indication of the size, while normalizing for certain layout idioms [12].

Q: How complicated is the code in this project?

- M: total lines of comments. This metric indicates how much to read next to the code to understand it. The metric is tricky to interpret, since often people comment out dead code [12]. Still, it is very basic metric that can not be ignored.
- M: ratio of lines of non-commented code to lines of comments. A high ratio could mean that a lot of code has been commented out, which is an indication of bad quality, or that a lot of explanation is needed, which indicates hard-to-understand code, or that the code is commented redundantly, which is an indication of inexperienced programmers. This ratio is argued for in related work as well [42].
- M: number of lines of code in units with low, medium, high cyclomatic complexity. Taken from the SIG maintainability model [12] this metric aggegregates risks to the project level. It can be compared to the cyclomatic complexity per unit to find the cause of a risks.
- M: SIG star rating [12] aggregrates over size and complexity and testing metrics to provide a 5 star rating. It is handy from the management perspective as an executive summary. Nothing more.

Q: How is complexity distributed over the design elements of project?

- M: cyclomatic complexity per executable unit (method) [23]. Indicates per unit how hard to test a method is (how many test cases you approximately would need) and is a proxy for understandability in that sense as well. Without aggregation this metric provides insight in the quality of specific design elements. It can also be considered to be a volume metric in that sense [7].
- M: gini coefficient of cyclomatic complexity over methods [41]. Provides a quick overview in case a trend towards more bad code being spread, or a sudden event that changed the balance of complexity.
- M: coupling and cohesion metrics per unit from the Chidamber & Kemerer suite [8]: coupling between objects, data abstraction coupling, message passing coupling, afferent coupling, efferent coupling, instability, weighted method per class, response for class, lack of cohesion, class cohesion. Source code complexity is influenced by separation of concerns, i.e. how well units can be analyzed separately from their context. The C&K metrics provide indications of how well concerns may have been separated. Its hard to automatically aggegrate these metrics to project level, so these are typically presented as histograms or gini coefficients.

Q: How is size distributed over the design elements of the project?

- M: non-commented lines of code per class, method. This metric provides insight in the cause of a large volume and the quality of the design. Big classes and big methods are code smells [10].
- M: gini coefficient of lines of code over classes, methods [41]. Provides a quick overview in case a trend towards more bad code being spread, or a sudden event that changed the balance of complexity, for example a lot of generated code being pushed to the repository.
- M: number of methods per class. A common and simple object-oriented metric [8, 20, 9], more basic than a code smell.

- M: number of attributes per class. A common and simple object-oriented metric [8, 20, 9], more basic than a code smell.

Q: How well does the code adhere to certain coding standards?

- M: number of code smell detections [10]. Code smells have been shown to be harmful [44].
- M: number of violations of industry standards, such as MISRA-C [**?**]. Many tools such as Coverity[11], SonarCube[12] implement such analysis for the C language. For Java, "CERT Oracle Secure Coding Standard for Java" could (partly) be implemented. This is still under investigation. Some coding standards are mostly about code layout which has been shown to have a large impact on understandability [25].
- M: depth of inheritance tree per class[8, 20, 9], a high number indicating overly complex code. It is common to try and avoid this.
- M: the MOOD metrics suite for object-oriented design[9]: Method Hiding Factor, Attribute Hiding Factor, Method Inheritance Factor, Attribute Inheritance Factor, Polymorphism Factor, Coupling Factor. These summarize the quality of the OO design.
- M: how many lines of code of the project are in clones bigger than 6 non-commented, non-empty lines. A metric from SMM [12], which indicates how much copy/paste programming has been done in the current project.

Q: Is the code tested automatically?

- M: check for existence of common unit test framework usage, such as JUnit[13]. This is just of the top of our heads.
- M: static testing coverage metric [3]. A difficult and expensive metric to compute, but highly valuable since it indicates a prime factor of maintainability as depicted by ISO/IEC 9126 and 25001. We have to experiment with this one to see if it will be feasible.

Q: Is the code easy to build and run?

- M: check for existence of common build infra-structure, such as Ant[14], Maven [15], or common IDEs such as Eclipse [16], Netbeans[17], GNU autotools[18], etc. This is just of the top of our heads. It may also be that natural language communications give a better indication of this aspect.

[11] www.coverity.com
[12] http://www.sonarqube.org/
[13] http://www.junit.org
[14] http://ant.apache.org/
[15] http://maven.apache.org/
[16] http://www.eclipse.org
[17] http://netbeans.org
[18] http://www.gnu.org

2.6.2 Goal: assess activity of the developer community

In combination with information from other sources, such as bug trackers and community forums, we need to obtain a view of the activity of a project. The activity may give hints about its health, and about possible risks involved in depending or contributing to the project. "Software Process Mining" [30] is the act of retrieving knowledge from the traces that developers leave while interacting in one way or another with the project. For WP3 we focus on the traces developers leave in source code and in the meta data of VCS repositories.

Note that the OSSMeter platform (see WP5) analyzes software projects with the granularity of a day at minimum. The following questions are all relative to the previous point in time (i.e. revision number) for which the platform analyzed the source code and the VCS meta data.

Q: How much code was changed?

- M: Number of changed files
- M: Churn per file
- M: Churn per project
- M: Declaration churn, i.e. how many added/removed classes or methods. This is a first derivative of the earlier language dependent volume metrics.

Q: How many people are active?

- M: number of committers per day. This can be aggregrated later dynamically for selected periods of time.
- M: number of core committers per day.

Q: How did the changes affect the maintainability of the system?

- M: all the maintainability metrics from above. By plotting these on a time axis we can see their development [6]. Metrics per unit would be presented by collections of histograms.
- M: the first derivative of all the maintainability metrics from above, plotted over time, such that we can see clearly when important things happened and what is normal development activity.

These questions are all relative to a certain time-frame, for example "the last 3 months". This time frame will be a parameter of the platform's UI.

Q: who is changing most of the code?

- M: churn per committer
- M: core committer list, committers ordered by churn. We avoid a threshold and just present the list in ordered fashion.

Q: how are changes distributed over the design elements of the project?

- M: earlier mentioned Gini coefficients measured for size distributions plotted over time.

2.6.3 Goal: compare open source projects on internal quality

This goal represents our intention to put projects next to each other in the UI of OSSMeter. To measure quality is oxymoronic: measurements are quantities. Only by comparison we can judge and interpret metrics. The metrics provide an adequate summary, and by comparing the summaries of two projects we can asses which project we like better or which version of a project we like better.

> Q: How do the latest versions of selected projects (i.e. in the same domain) compare in terms of maintainability?

- M: present the earlier mentioned quality metrics for both projects.
- M: present differences between the quality metrics for the projects.

2.6.4 Goal: assess the viability of an open source project

Again, we focus here on juxtapositioning versions and projects for allowing the user to make a qualitative judgment based on summaries provided by metrics. We want to spot trends in activity to try and predict whether or not a project has a healthy (short-term) future.

> Q: How do selected projects (i.e. in the same domain) compare in terms of activity for a given time frame?
>
> - M: present the earlier mentioned activity metrics for both projects next to each other over a period of time.

2.7 Summary

We explained the goals of WP3 and its basic functional and non-functional requirements. Key design decisions include reusing existing front-ends, implementing metrics from scratch in the Rascal language and a language parametric intermediate representation called **M3**.

We used the GQM perspective to motivate a number of existing metrics explained earlier in Deliverable 3.1, to motivate new metrics explained here for Deliverable 3.2 (see the next Section 4 for more details, and to identify possible metrics to experiment for in the future. An exact report on what was planned, and what is to be done is in Section 8.

3 Quality Metrics with Rationale

Here we will enumerate again the metrics considered previously in Task 3.1. for quality analysis in WP3, explain their rationale and why they are implemented from scratch.

Existing implementation of source code metrics can be found online in frameworks such as SonarCube, Bauhaus, and Understand, yet we choose not to reuse them. We chose to reuse language front-ends that produce ASTs and not the particular source code metrics implementations. There exist for every definition of a source code metric in literature quite a few implementations that differ from it. The reasons are opaque. Sometimes metrics are not designed for new programming language features and they should be interpreted in some way or another. Some metrics are not easy to calibrate with a threshold. In general the statistical properties of a metric over a large corpus of today's source code are unknown, so arbitrary changes to the metric definitions are unfounded. For OSSMETER we intend to implement published metrics as directly as possible and to name any experiments with derived or adapted metrics differently. In this way we can clearly communicate to the user what the semantics of each metric is.

We hope that controlling each implementation of a metric and having it in a similar form enables experimentation and cross-language consistency. Most metrics are only a few lines of Rascal code. In general the motivation for implementing metrics from scratch in OSSMETER is to control their definition and correct implementation. Note that to implement the metrics we *do reuse* existing front-ends for programming languages and libraries for VCS providers:

- Eclipse Java Development Tools
- PHPParser
- JGit
- SVNKit

In the remaining parts of this section we would like to introduce some aspects of working with quality metrics that were thought up of while working on Task 3.2.

3.1 Differencing M3 Models

The **M3** intermediate model programming language syntax and static semantics model was introduced for Task 3.1 [38] and forms the basis for analyzing source code activity as well. The metrics based on the **M3** model can often be expressed generically, given the appropriate mapping from programming language specific abstract syntax trees to the core part, or the object-oriented extension of **M3**. The **M3** model and its producers are a pivotal results of OSSMETER, with plenty applications outside of the project as well. Since **M3** capture the core static semantics of a programming language it used as a reusable language parametric front-end for other applications such as IDE construction for domain specific languages, static analysis and source code refactoring tools.

An **M3** model is basically a set of relations, where a relation is a set of tuples. By extracting an M3 model for two consecutive versions we can compute a number of interesting metrics without much effort. Example facts from an **M3** model are "declarations": a relation from declared entities to their source code locations. The *set difference* operator between two sets of declarations $D_1 \backslash D_2$

(and vice versa) represents the difference between what is declared between two projects. This is how we measure for example "number of added/deleted methods", "number of added/deleted classes" etc. Logical/language dependent size metrics are all implemented in this fashion.

3.2 Distributions and Aggregation

The semantically deeper metrics, such as cyclomatic complexity and documentation density, can be computed for the new model and compared directly to the old model. Here we see that aggregation or visualization is of importance. The development of the distribution of these metrics over time is interesting. Are more and more classes dependent on more and more other classes? Or are a few code "God" classes becoming connected to all others? These are relevant questions which can not easily be seen from differencing the basic metric data.

For Task 3.1 we focused on producing the raw activity data based on incrementally computing the basic metrics (see next subsection). However, eventually the platform will provide adequate visualizations for the metrics, and allow side-by-side or superimposed images of metric distributions.

We also experimented summarizing entire distributions using the Gini coefficient. A gini coefficient is a measure of "evenness" of a distribution. An example: does each class have the same number of methods, or is the distribution of methods over classes heavily skewed? How is the skewness developing over time?

Gini coefficients give insight in:

- radical changes in the shape of such distributions indicating radical changes in the design of a system;
- Trends in the shape of such distributions indicate creeping risks or incremental improvements over time.

By computing the Gini coefficients we can summarize an entire distribution in a single number and plot it over a large number of revisions.

3.3 Modular/Incremental Model Extraction

The basic **M3** model can be produced on a file-by-file basis: i.e. each source code file produces a single model. To get an overview of a project, we can:

- compute metrics per file model and aggregate externally, or
- merge models of all files and compute metrics over the unified model

Some metrics can be computed in either way, some metrics need the global view to make sense, some metrics are incorrect if considered on a file-by-file basis. The reason for the latter is that there may be duplication between different file-based models that is removed after merging the models.

To optimize metric calculation the current Rascal metric providers calculate M3 models per file and store them on disk. Merging models is not done for any of the current providers, but will be

necessary for computing some of the language dependent metrics (e.g. coupling metrics) for the future Deliverable 3.4.

We currently compute only new models for the files which have changed. This makes the model extraction faster, but care must be taken since it depends on the eventual metrics computed from the models whether or not the models for *depending* files have to be recomputed as well. The good news is that every **M3** core model explains dependency in detail, such that the dependencies of the old models may be used to compute the "damage" for computing the new model. This is future work for Deliverable 3.4.

3.4 Conclusions and future work

The **M3** model is an extensible infra-structure that can be used for computing metrics and comparing (diffing) semantically rich programming language models.

To use Gini coefficients for summarizing distributions is a core idea for the platform which we will explore further when test driving the platform. The existing Gini metrics give a good starting point for experimentation and more of these are added easily in the future.

M3 models can be computed incrementally, but optimization is still necessary and we need an eye on the correctness of incrementally updated models. For the future we may consider developing a generic (language-parametrized) damage computation based on dependencies between declarations.

4 Measuring source code activity

4.1 Background

A number of big success stories in OSS project, GNU/Linux probably being the biggest, had led to researchers trying to understand what is it that make an OSS project successful. A survery of the current literature points out user and developer interest [36], [26]) and project activity [37] as key success measures. Midha and Palvia [26] also note that OSS projects need to keey themselves less complex and more modular, since more complex projects deter user involvement while modularity enhances it.

4.1.1 People

The importance of people in OSS projects has been noted by Markus et al [21]. They mention the critical need for OSS projects to attract contributors on an on-going basis to be sustainable. Mockus et al [27] point out that for a successful project, it needs a group of people, larger by an order of magnitude than the core, to repair defects and even a larger group should be there to report problems. Raymond [31] emphasize the importance of large groups for successful projects. Schweik and English [33] however have noted that it is team activity and not size that is a factor for success.

Nakakoji et al [28] classify the community of an OSS project into eight roles: Passive User, Reader, Bug Reporter, Bug Fixer, Peripheral Developer, Active Developer, Core Members and Project Leader. We will be using these definitions whenever we talk about the people involved with a project.

4.1.2 Activity

Mattila and Mehtonen [22] highlight projects need keep moving and be active or the developers loose interest.

4.2 Introduction

Open source software projects usually make their project source code avaiable through various version control systems. The version control system allows us to answer who, when, why, what information regarding changes to an open source software project. It provides us with not only the activity information but also the people involved with those activities. Our focus in WP3 when we mention we measure source code activity is to answer these type of questions, some of which can be answered using the versioning system data alone while the others will need to be answered with the data from the versioning system as well as the data from the source code of the project.

The goal is to get insight into not only how much is happening (quantitatively), but also what is happening to the source code (qualitatively). For Task 3.1 we produced a number of quality indicators for source code based on metric values. To see these metrics develop in time, for consecutive versions of a project, should produce insight in how a project is developing in terms of quality.

Metric	Per unit	Rationale
Lines churn	file, committer, commit	Where is work done? By whom? When?
Number of committers	file	How is responsibility distributed over the project?
Number of files	commit	What is the commit culture? big steps or small steps?
Declaration churn	classes, methods, ...	Compare with volume metrics for an idea of design quality.
Core committers	project	Who is taking responsibility of the project?
Gini coefficients	CC over methods, LOC over class, LOC over files	For all metrics it is interesting how they distribute over a project. Sudden changes indicate a sudden change in design, trends tend creeping design degradation or incremental refactoring
(Partial) SIG maintainability model	project	What is the executive summary on the source code quality of the project? The SMM aggregates basic ISO 9126 quality indicators a four dimensional scale and finally to a five-star rating.

Table 1: Summary of metrics

The challenge is to summarize and visualize the development of the metrics such that an overview can be achieved. Most metrics are on a unit basis: per method, per class, per file. Since distribution types of the metrics over the units are either unknown or heavily skewed (exponential, log normal), normal aggregation methods such as computing a mean or selecting a median will not inform what is actually happening with the source code but rather hide important aspects of the quality of a system.

We use the meta data from VCS systems as described in 7 received from the OSSMETER platform which is converted into the model in Figure 9 and passed to each metric function to calculate its value.

4.3 Activity Metrics

Mattila and Mehtonen [22], have noted that explicit metrics are missing from almost all the papers. The only exception was Schweik and English [34] where the authors developed a classification system for describing the success and abandonment of OSS projects. The metrics that we have compiled for Task 3.2 have been done in line with trying to identify the people and to highlight the activity that has taken place in a project.

A summary of activity metrics that were thought of during the implementation and testing of Task 3.2. These are the depicted in Table 1 and have been prototyped in the platform.

Below we present all the metrics that have been compiled to show activity in OSS projects.

4.3.1 Number of commits

The number of commits metric counts the number of changes that have take place since the last measurement.

- Rationale
  The number of commits that have occured between the two measurement interval provides an indication of the activity that have occured recently. We can aggregate the data from this metric to provide an estimation of the effort that has been put into the project.
  Significant changes in any direction (more changes/less changes than normal) could be an indicator of change in developer interest in the project.

4.3.2 Number of committers

The number of committers metric counts the number of people who have contributed to the project since the last measurement.

- Rationale
  Indicates the size of developer base of a project. Changes in the number indicates the change in the interest among the people involved with the project. As pointed out in [21], constant increase in the number would indicate that the project in sustainable, while decreases could be an early indicator of project failure (resulting from the decrease of interest in the project).

4.3.3 Churn

We have introduced a number of different metrics to measure the churn in a system. The idea of measuring churn is to try to find out what type of activity is taking place. The current implementation will need to be augmented with the works in [16] to be able to try to identify what kind of activity is taking place.

1. Churn per commit

   This metric shows us how many lines of code have changed between the last measurement and now. Currently the metric returns a single number (sum of lines added and deleted, which is the definition of churn) per revision. In the future, we could change this so that we get lines added and deleted separately (maybe even lines changed).

   - Rationale
     Indicates the scale of each activity. This allows us to see how much work was done per change. We would like to characterize the activity depending on the churn data.

   - Source code

```
1 @metric{churnPerCommit}
2 @doc{Count churn}
3 @friendlyName{Counts number of lines added and deleted per commit}
```

```
4  map[str revision , int churn] churnPerCommit(ProjectDelta delta , map[str, loc] workingCopyFolders, map[str, l
5    = (co. revision : churn(co) | /VcsCommit co := delta)
6    ;
7
8  int churn(VcsCommit item)
9    = (0 | it + count | /linesAdded(count) := item)
10   + (0 | it + count | / linesDeleted (count) := item)
11   ;
```

2. Churn: per committer

   This metric shows us how many lines of code has been changed by each committer.

   - Rationale
     Indicates the activity associated with a committer. This allows us to associate the changes to the people who made the changes (add accountability). This also allows us to provide a guess about the type of activity that each person is dedicated to.

   - Source code

```
1  @metric{churnPerCommitter}
2  @doc{Count churn per committer}
3  @friendlyName{Counts number of lines added and deleted per committer}
4  map[str author, int churn] churnPerCommitter(ProjectDelta delta
5                                               , map[str, loc] workingCopyFolders
6                                               , map[str, loc] scratchFolders )
7    = (co.author : churn(co) | /VcsCommit co := delta );
8
9  int churn(VcsCommit item)
10   = (0 | it + count | /linesAdded(count) := item)
11   + (0 | it + count | / linesDeleted (count) := item);
```

3. Churn: per file

   This metric shows us how many lines of code has been changed for each file.

   - Rationale
     Indicates the activity associated with a file. Decrease in the metric over the life of a file could be an indicator of files becoming more stable.

   - Source code

```
1  @metric{churnPerFile}
2  @doc{Count churn}
3  @friendlyName{Counts number of lines added and deleted per file }
4  map[str file , str churn] churnPerFile( ProjectDelta delta
5                                         , map[str, loc] workingCopyFolders
6                                         , map[str, loc] scratchFolders )
7    = (co.path : churn(co) | /VcsCommitItem co := delta );
8
9  int churn(VcsCommitItem item)
10   = (0 | it + count | /linesAdded(count) := item)
11   + (0 | it + count | / linesDeleted (count) := item);
```

4. Churn: per class

   This metric counts the number of elements of a class has changed since the time it was last measured.

   - Rationale
     Churn per class (with collaboration with lines of code) would allow us to provide a clue about the design quality of the system. As an example, if we see that a lot of lines of code have been added to the system but the churn per class hasn't been affected much, we could probably conclude that the design of the system isn't very good.

   - Source code

```
1  @metric{classChurn}
2  @doc{classChurn}
3  @friendlyName{classChurn}
4  int getClassChurn( ProjectDelta delta , map[str, loc] workingCopyFolders
5                   , map[str, loc] scratchFolders ) {
6    int churnCount = 0;
7    visit (classChurn) {
8      case classContentChanged(loc changedClass, set[loc] changedContent):
9        churnCount += size(changedContent);
10     case classModifierChanged( locator , oldModifiers , newModifiers):
11       churnCount += 1;
12     case classDeprecated(loc locator): churnCount += 1;
13     case classUndeprecated(loc locator): churnCount += 1;
14     case addedClass(loc locator): churnCount += 1;
15     case deletedClass(loc locator): churnCount += 1;
16   }
17   return churnCount;
18 }
```

5. Churn: per method

   Counts the number of methods that have changed since the last measurement.

   - Rationale
     Similar to churn per class, we could use the measure the design quality of a system from the perspective of division of the methods in the system.

   - Source code

```
1  @metric{methodChurn}
2  @doc{methodChurn}
3  @friendlyName{methodChurn}
4  int getMethodChurn(ProjectDelta delta , map[str, loc] workingCopyFolders
5                   , map[str, loc] scratchFolders ) {
6    int churnCount = 0;
7    visit (methodChurn) {
8      case unchanged(loc locator): churnCount += 0;
9      case returnTypeChanged(loc method, TypeSymbol oldType, TypeSymbol newType):
10       churnCount += 1;
11     case signatureChanged(loc old, loc new): churnCount += 1;
```

```
12        case modifierChanged(loc method, set[Modifier] oldModifiers, set[Modifier] newModifiers):
13          churnCount += 1;
14        case deprecated(loc locator): churnCount += 1;
15        case undeprecated(loc locator): churnCount += 1;
16        case added(loc locator): churnCount += 1;
17        case deleted(loc locator): churnCount += 1;
18      }
19      return churnCount;
20    }
```

6. Churn: per field

   Counts the number of fields that have been changed since the last measurement.

   - Rationale
     Similar to the churn per class and method metrics this metric will allow us to provide an indication of the design quality of the system by looking at the changes to the fields that change.

   - Source code

```
1   @metric{fieldChurn}
2   @doc{fieldChurn}
3   @friendlyName{fieldChurn}
4   int getFieldChurn(ProjectDelta delta, map[str, loc] workingCopyFolders
5                     , map[str, loc] scratchFolders) {
6     int churnCount = 0;
7     visit (fieldChurn) {
8       case fieldModifierChanged(locator, oldModifiers, newModifiers):
9         churnCount += 1;
10      case fieldTypeChanged(loc locator, _, _): churnCount += 1;
11      case fieldDeprecated (loc locator): churnCount += 1;
12      case fieldUndeprecated(loc locator): churnCount += 1;
13      case addedField(loc locator): churnCount += 1;
14      case deletedField(loc locator): churnCount += 1;
15    }
16    return churnCount;
17  }
```

4.3.4 Contributors

Through the use of these metrics we would like to identify the different roles that the people involved with the project have been playing. We feel that we could be able to identify few roles (core member, active developer, peripheral developer) based on the information we received from the version control system, while for other roles (bug reporter, bug fixer) we might need to work in tandem with our partners in indentifying them.

1. Core Contributors Counts the LOC changes produced by each contributor over the history of the project. The people with the highest contributions appear ahead of the rest in the resutling list.

[28] define core contributors as people who have been involved with the project for a relative long time and have made significant contributions to the project. To get an initial idea of the core contributor we have deviated from the definition of core contributors in that we only count the total contributions. As future work we would like to align our implementation with the definition in [28].

- Rationale
  Identifying the core group of contributors to answer who are the most important group for a project. A core contributor becoming inactive for a long period of time could be an early indicator of risk to the project (specially if we can see from other metrics that there are certain part of the source code where other members are little to no knowledge).

- Source code

```
1  @metric{coreCommitters}
2  @doc{Finds the core committers based on the churn they produce}
3  @friendlyName{Core committers}
4  list[str] coreCommitters(ProjectDelta delta
5                           , map[str, loc] workingCopyFolders
6                           , map[str, loc] scratchFolders) {
7    map[str author, int churn] committerChurn
8      = churnPerCommitter(delta, workingCopyFolders, scratchFolders);
9    map[str author, int churn] olderResult = ();
10
11   loc coreCommittersHistory = |home:///ossmeter/<delta.project.name>/corecommitters.am3|;
12
13   if (exists(coreCommittersHistory)) {
14     olderResult = readBinaryValueFile(#map[str, int], coreCommittersHistory);
15     for (str author <- olderResult) {
16       committerChurn[author]? 0 += olderResult[author];
17     }
18   }
19
20   writeBinaryValueFile(coreCommittersHistory, committerChurn);
21
22   list[int] churns = reverse(sort(range(committerChurn)));
23   map[int, set[str]] comparator = invert(committerChurn);
24
25   return [author | authorChurn <- churns, author <- comparator[authorChurn]];
26 }
```

2. Active committers

   Using the definition of [28], active developers(committers) are poeple who regulary contribute features and fix bugs. For our implementation, we have decided that active committers are defined as people who have at least one commit in the last 15 days. The number of days to decide the status will have to be parametized since other researchers [30] have used activity for 30 days to decide if a committer is active.

   - Rationale

Shows who have been most active in the last measurement period. This metric could be used to indicate how the committers community has been changing. Steady/drastic decrease or increase in the number could be an indicator of the project becoming less or more interesting to people.

- Source code

```
1  @metric{activeCommitters}
2  @doc{activeCommitters}
3  @friendlyName{activeCommitters}
4  str activeCommitters( ProjectDelta delta , map[str, loc] workingCopyFolders, map[str, loc] scratchFolders ) {
5    list [ str ] activeAuthors = [];
6    datetime today = delta . date ;
7    writeBinaryValueFile (|home:///ossmeter/<delta . project . name>/activecommitters_<printDate(today . justDate , "y
     delta ]);
8    list [datetime] activePeriod = dateRangeByDay(createInterval(decrementDays(delta . date , 15), today ));
9
10   for (datetime d ← activePeriod) {
11     loc activeCommittersForDay = |home:///ossmeter/<delta . project . name>/activecommitters_<printDate(d. justDa
12
13     if ( exists (activeCommittersForDay)) {
14       activeAuthors += readBinaryValueFile (# list [ str ], activeCommittersForDay);
15     }
16   }
17
18   map[str, int ] dist = distribution ( activeAuthors );
19
20   list [ int ] activityCount = reverse ( sort (range( dist )));
21   map[int, set[ str ]] comparator = invert ( dist );
22
23   return intercalate (", ", [author | numActivity ← activityCount, author ←
     comparator[numActivity]]);
24 }
```

3. Inactive committers

Counts the number of people who have no active commits in the last 3 months but at least one commit in the 6 months prior to that.

- Rationale This could be used as an indicator to measure if the project is losing contributors. If the developer interest drops in the project and the project isn't able to attact new developers could be an indicator of the project heading towards failure.

- Source code

```
1  @metric{inactiveCommitters}
2  @doc{inactiveCommitters}
3  @friendlyName{inactiveCommitters}
4  list [ str ] inactiveCommitters( ProjectDelta delta , map[str, loc] workingCopyFolders, map[str, loc] scratchFold
5    set[ str ] threeMonthsActive = {};
6    set[ str ] activeBeforeThreeMonths = {};
7    datetime today = delta . date ;
```

```
8     writeBinaryValueFile (|home:///ossmeter/<delta.project.name>/activecommitters_<printDate(today.justDate, "
    delta]);
9     datetime threeMonthsAgo = decrementMonths(delta.date, 3);
10    list[datetime] activePeriod = dateRangeByDay(createInterval(threeMonthsAgo, today));
11
12    for (datetime d <- activePeriod) {
13      loc activeCommittersForDay = |home:///ossmeter/<delta.project.name>/activecommitters_<printDate(d.justD
14
15      if (exists(activeCommittersForDay)) {
16        threeMonthsActive += {*readBinaryValueFile(#list[str], activeCommittersForDay)};
17      }
18    }
19
20    activePeriod = dateRangeByDay(createInterval(decrementMonths(threeMonthsAgo, 6), threeMonthsAgo));
21
22    for (datetime d <- activePeriod) {
23      loc activeCommittersForDay = |home:///ossmeter/<delta.project.name>/activecommitters_<printDate(d.justD
24
25      if (exists(activeCommittersForDay)) {
26        activeBeforeThreeMonths += {*readBinaryValueFile(#list[str], activeCommittersForDay)};
27      }
28    }
29
30    return [*(activeBeforeThreeMonths - threeMonthsActive)];
31  }
```

4.3.5 Committers: per file

The count of the number of people who have contributed to a single file.

- Rationale To understand how the responsibility between the contributors is distributed over the project. Files that have only one contributors working on them would be prime candidate for risk points in case the contributor maintain the file suddenly becomes inactive.

- Source code

```
1  @metric{NumberOfCommittersperFile}
2  @doc{Count the number of committers that have touced a file}
3  @friendlyName{Number of Committers per file}
4  map[str file, int numberOfCommitters] countCommittersPerFile(ProjectDelta delta, map[loc, loc] workingCopyFolder
5    map[str, set[str]] result = committersPerFile(delta, workingCopyFolders, scratchFolders);
6
7    return (key : size(result[key]) | key <- result);
8  }
9
10 map[str, set[str]] committersPerFile(ProjectDelta delta, map[loc, loc] workingCopyFolders, map[loc, loc] scratchF
11   map[str file, set[str] committers] result = ();
12   set[str] emptySet = {};
13   for (/VcsCommit vc <- delta, vc.author != "null") {
14     for (VcsCommitItem vci <- vc.items) {
15       // Need to check that the committer is not already counted
16       result[vci.path]? emptySet += {vc.author};
```

```
17        }
18      }
19
20      return result ;
21    }
```

4.3.6 Files: per commit

The count of the number of files that have changed per commit.

- Rationale: To understand what the commit culture is for a project, whether people make commits in big steps or small. Another use of the metric would be to identify parts of the system that tend to get changed together.

- Source code

```
1  @metric{filesPerCommit}
2  @doc{Counts the number of files per commit}
3  @friendlyName{Number of files per commit}
4  map[str revision , int count] numberOfFilesPerCommit(ProjectDelta delta , map[loc, loc] workingCopy, map[loc, loc] s
5    map[str revision , int count] result = ();
6
7    for (/VcsCommit vc ← delta) {
8      result[vc.revision]? 0 += size(vc.items);
9    }
10
11   return result ;
12 }
```

4.3.7 Distributions

For all metrics it is interesting how they distribute over a project. Sudden changes indicate a sudden change in design, trends tend creeping design degradation or incremental refactoring.

1. Distribution: committers over files

   Shows how the contributions of committers are distributed over the files in the project.

   - Source code

```
1  @metric{committersoverfile}
2  @doc{Calculates the gini coefficient of committeroverfile}
3  @friendlyName{committersoverfile}
4  real giniCommittersOverFile(ProjectDelta delta , map[str, loc] workingCopyFolders, map[str, loc] scratchFolder
5    rel[str , str] filesCommitters = {< commitItem.path, vcC.author > | /VcsCommit vcC ←
   delta, commitItem ← vcC.items};
6
7    committersOverFile = distribution (filesCommitters <1,0>);
8    distCommitterOverFile = distribution (committersOverFile);
9
10   return gini([<0,0>]+[<x, distCommitterOverFile[x]> | x ← distCommitterOverFile]);
11 }
```

2. Distribution: LOC over files

   Shows how the LOC is distributed over the files.

   - Source code

```
@metric{ locoverfiles }
@doc{ locoverfiles }
@friendlyName{ locoverfiles }
real giniLOCOverFiles(ProjectDelta delta, map[str, loc] workingCopyFolders, map[str, loc] scratchFolders ) {
  map[str, int] locMap = countLoc(delta, workingCopyFolders, scratchFolders );

  distLOCOverMethods = distribution (locMap);

  return gini ([<0,0>]+[<x, distLOCOverMethods[x]> | x ← distLOCOverMethods]);
}
```

3. Distribution: LOC over classes

   Shows how LOC is distributed over classes.

   - Source code

```
@metric{ locoverclass }
@doc{ locoverclass }
@friendlyName{ locoverclass }
real giniLOCOverClass(ProjectDelta delta, map[str, loc] workingCopyFolders, map[str, loc] scratchFolders ) {
  map[str class, int lines] result = ();
  map[str, list [str]] changedItemsPerRepo = getChangedItemsPerRepository(delta );

  for (str repo ← changedItemsPerRepo) {
    list [str] changedItems = changedItemsPerRepo[repo];
    loc workingCopyFolder = workingCopyFolders[repo];
    loc scratchFolder = scratchFolders [repo ];

    for (str changedItem ← changedItems) {
      if ( exists (workingCopyFolder+changedItem)) {
        loc scratchFile = scratchFolder +changedItem;
        M3 itemM3 = readBinaryValueFile(#M3, scratchFile [ extension = scratchFile . extension+".m3"]);
        if (!( unknownFileType(_) := itemM3)) {
          result += (lc . path : sc . end. line − sc . begin . line + 1
                     | <lc, sc> ← itemM3.model@declarations, isInterface ( lc )
                     || isClass ( lc ) || lc . scheme == "java+enum");
        }
      }
    }
  }

  distLOCOverClass = distribution ( result );
  return gini ([<0,0>]+[<x, distLOCOverClass[x]> | x ← distLOCOverClass]);
}
```

   - Results
     Figure 3 shows a sample of the evenness of the distribution of lines of code over classes image plotting the gini coefficients on the y axis and the date on the x axis.

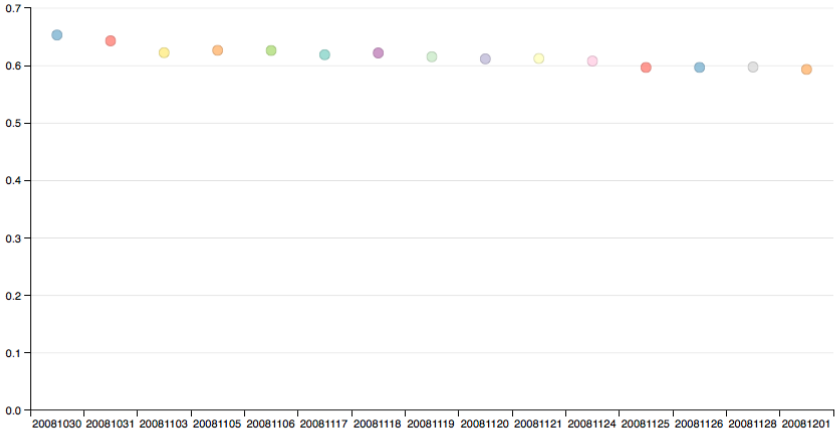

Figure 3: Distribution of evenness of lines of code over classes

4.4 Conclusions and Future Work

We have implemented an initial set of metrics that we feel tells about the people and the activity related to a project. The list of metrics is not complete and we hope to extend it with more metrics pertaining to activities on going on the project. As can be evidenced from most of the metric sources provided, we feel adding metrics that related to activity in the project alone are trivial.

```
1 @metric{metricName}
2 @doc{Long metric description}
3 @friendlyName{short friendly metric description}
4 map[loc, num] myMetric(ProjectDelta delta, map[str, loc] wc, map[str, loc] scratch) {
5   ... // rascal code;
6 }
```

Figure 4: Template for introducing a new metric provider in Rascal: no boilerplate code.

5 Adding new source code and activity metrics

The initial prototype of metric providers for Task 3.1 has been extended to be able to cover more different kinds of metrics easily. We noticed that experimentation is necessary and we predict that for the final deliverables in M24 still new metrics or derivates of existing metrics will be introduced. This is reasonable since only then the platform will be fully integrated and support a dashboard where different metrics come together. From the initial prototypes we expect feedback and fine-tuning requests that point back to the metric providers from Task 3.1 and Task 3.2.

To enable experimentation and fine-tuning we opted for a no-configuration, no-boilerplate design of integrating Rascal metrics into the platform. First we explain the result of the new design from the programmer's perspective, then we explain how it works internally.

5.1 Programmer perspective

To add a new Rascal metric the OSSMETER programmer can choose to create a new plugin project, or add a new metric to an existing project. We propose to group similar metrics into the same project, where "similar" means for the same programming language and for the same measured dimension. Language agnostic metrics can also be grouped together as one.

To add a new metric project, the programmer should:

- Create a new folder in the OSSMETER repository and initialize it as an Eclipse plugin project.
- Create a `plugin.xml` file containing:
  `<extension point="OSSMETER.rascal.metricprovider"></extension>`
- Create a `META-INF/RASCAL.MF` file containing:
  `Main-Module: MyMetric`
- Create an empty `src/lang/LanguageName/metrics/MyMetric.rsc` containing: **module** MyMetric. Or for a language agnostic metric we use `src/metrics/MyMetric.rsc`.

To add a new metric to a project, the programmer can simply add a function to the `MyMetric` module as depicted in Figure 4. From this code example you can read that the programmer writes a single Rascal function to add a new metric. Such functions will be called with three arguments containing a model of what happened since last time OSSMETER analyzed the project (see Section 7), locations of the relevant working copies of the source code of the project (see Section 6) and for convenience, locations of additional folders to store transient or non-transient extracted facts.

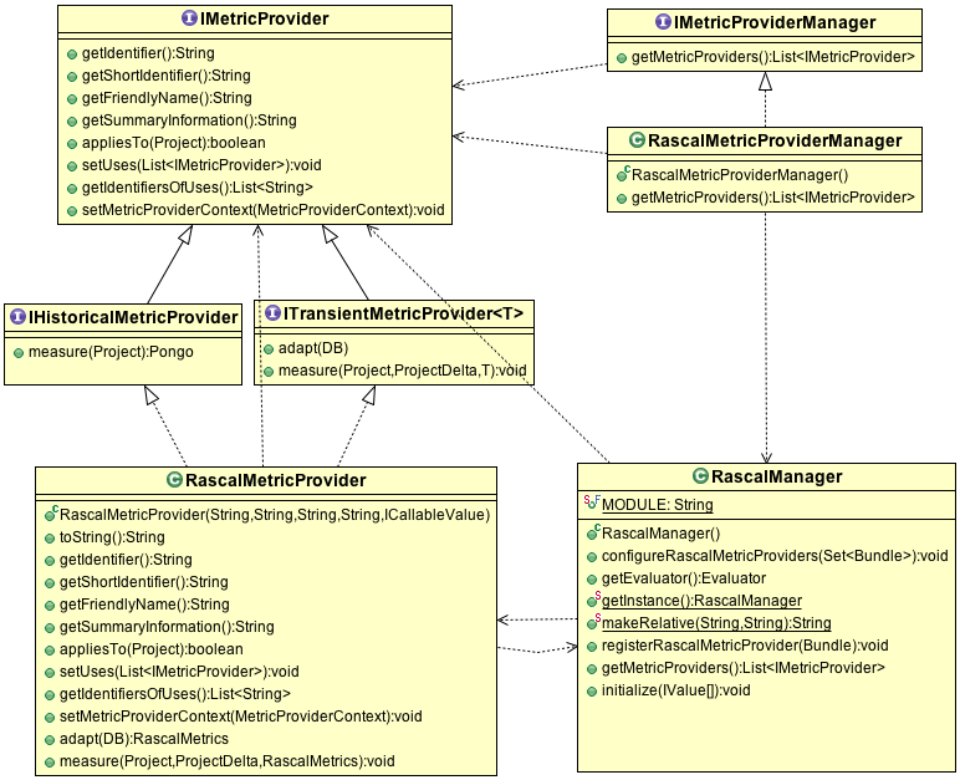

Figure 5: UML class diagram showing how Rascal is bridged to the OSSMETER platform

Note that the disk space is expected to be volatile, and serves only as a means for optimization. The results of metric computations are always stored in the OSSMETER database via the platform. Here we only cater for making metric computations incremental by allowing them to cache artefacts on disk if so required.

The `@` annotations provide obligatory meta information to the platform about each metric, which is used in registering the metric to the platform as well as presenting its results to the user.

5.2 Design and implementation

The OSSMETER platform was extended with a feature to register multiple metric providers at once (See Deliverable 5.3, Section 4.2). Its `metricProviderManager` extension point now accepts implementations of `IMetricProviderManager` with one method `IMetricProvider getMetricProviders`. This feature enables us to register as many metric providers as needed when the platform boots. The idea is to instantiate one IMetricProvider per Rascal function as mentioned above.

Figure 5 shows a class diagram with the essential elements of the bridge between Rascal and the OSSMETER platform. The interfaces `IMetricProvider`, `IHistoricalMetricProvider` and `ITransientMetricProvider` are provided by the platform. `RascalMetricProvider` is a Java class which holds a reference to a given Rascal function. This class takes care of computing the parameters to the Rascal function by using the working copy creation facilities (Section 6) and the VCS meta data

providers (Section 7), then calling the function, and finally marshalling its resulting metrics into the platform database. The marshalling is done using generic helper classes generated by Pongo[19] (See Figure 6).

The `RascalManager` abstracts the Rascal run-time. If the run-time changes, for example when the Rascal compiler is released, this is where to expect changes. As you can see this manager is coupled with `RascalMetricProvider`, so this class is expected to have to change along with it. The manager holds a reference to a Rascal `Evaluator`, and loads a general `Manager` utility module and instantiates `RascalMetricProviders` by passing in references to `ICalleableValue` at construction time.

When the platform asks for metric providers by calling the method on `IMetricProviderManager`, the `RascalMetricProviderManager` will first look for OSGI projects which extend the tag extension point `rascalMetricProvider`. This is just to filter all present OSGI plugins. For the filtered plugins, the metric provider will search for the `META-INF/RASCAL.MF` meta data, load the Rascal modules into memory and use API of the Rascal run-time to enumerate all Rascal functions that are tagged with `@metric`. For each such function an instance of `RascalMetricProvider` is instantiated and the resulting collection is the return value of `IMetricProvider::getMetricProviders`.

As for separation of concerns: the platform is still oblivious to how each metric is implemented, and the metrics are oblivious as to how the input parameters are obtained. The contract for source code metrics is based solely on the availability of a working copy on disk, for which the location is provided as a parameter, and on the abstract representation of VCS meta data.

Concerning the results of each metrics we should also explain a deviation from the general design of OSSMETER. In general we generate new model classes for each metric provider's results using Pongo. For the Rascal situation this is not necessary because its type system is more flexible than Java. So instead of introducing new classes for each Rascal-based metric, we offer a number of generic reusable models that link input to output metric in the database. Figure 6 shows their UML diagram. We expect some more of these classes to be added in the future, and we use Pongo itself to do this.

5.3 Conclusion and future work

This way of adding new metrics to the OSSMETER platform has no boilerplate or overhead whatsoever. The wiring code is a single point of change for Rascal based metrics in case of significant future design changes in the platform, which mitigates the risk of platform evolution.

Note that the metrics produced for Task 3.1 have been updated to reflect this new design.

Changes to be expected in this interface, if any, would be in the parameters as they are passed to each metric. In future versions of the platform we may use optional keyword parameters instead of obligatory ordered parameters to make the metrics more robust against platform changes. Also, for M3-based metrics we may factor out the acquisition of these models for certain metrics such that the user does not have to call the M3 model creation manually for each metric.

[19] `https://code.google.com/p/pongo/`

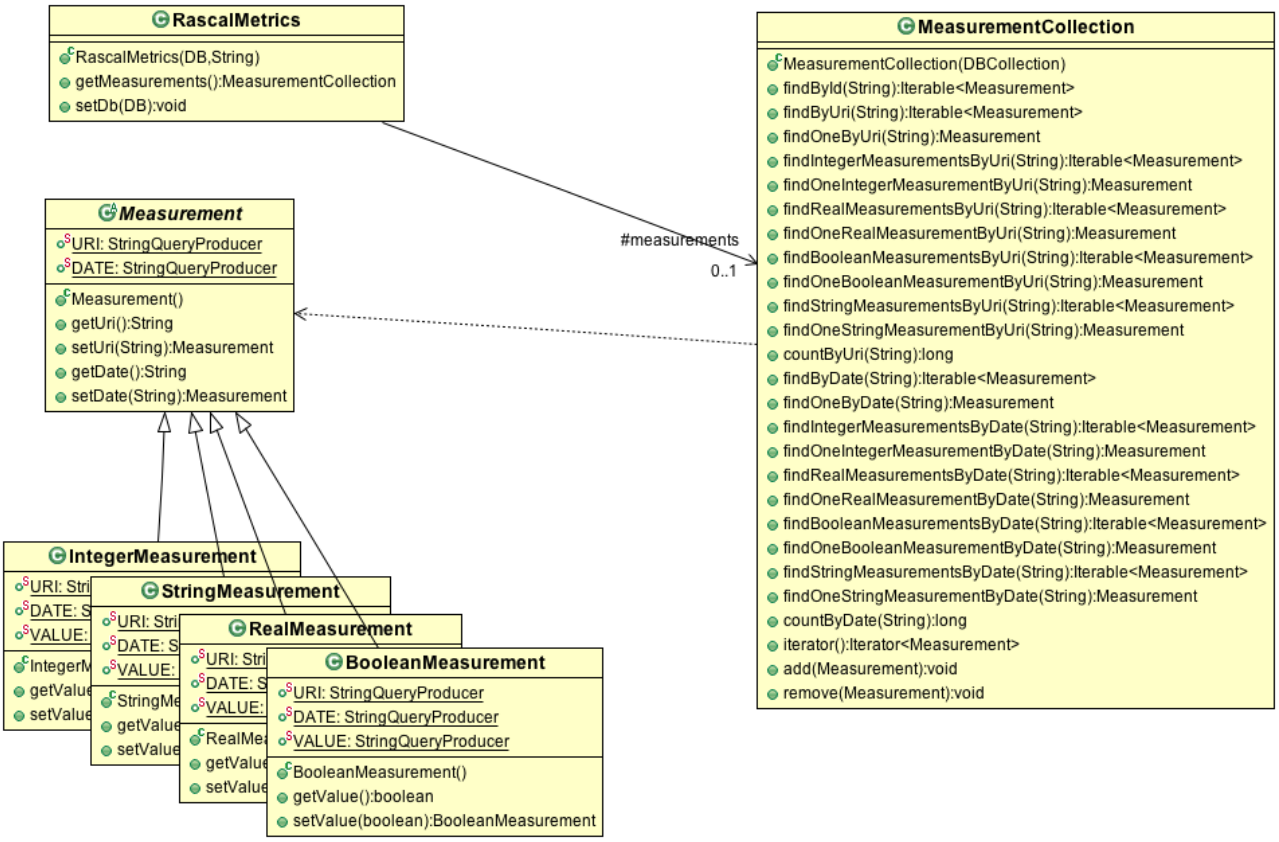

Figure 6: Pongo-generated model classes for generic metric results as computed by Rascal-based metric functions.

6 Working copies and source code differences

The platform as developed in WP2 provides access to version control meta data. This models information about what projects exists and in which repositories the code is stored. WP2 also provides the necessary meta data for some of the metrics that we define in this deliverable (Section 4). This is information about which versions have been committed in a certain time frame and by whom. This model is called `ProjectDelta`, and it is described elsewhere [**?**].

Next to the metrics that run directly on the VCS model, we also need metrics to be defined on top of source code differences. This requires a lower level of abstraction and the actual source code of the versions to be compared to be available. Source code metrics for OSS projects are based by necessity on full working copies of the project at a certain time. Acquiring working copies requires knowledge for each kind of VCS provider. For activity analysis furthermore, the metrics need information about which source code (lines) changed and which did not, and by whom they were changed.

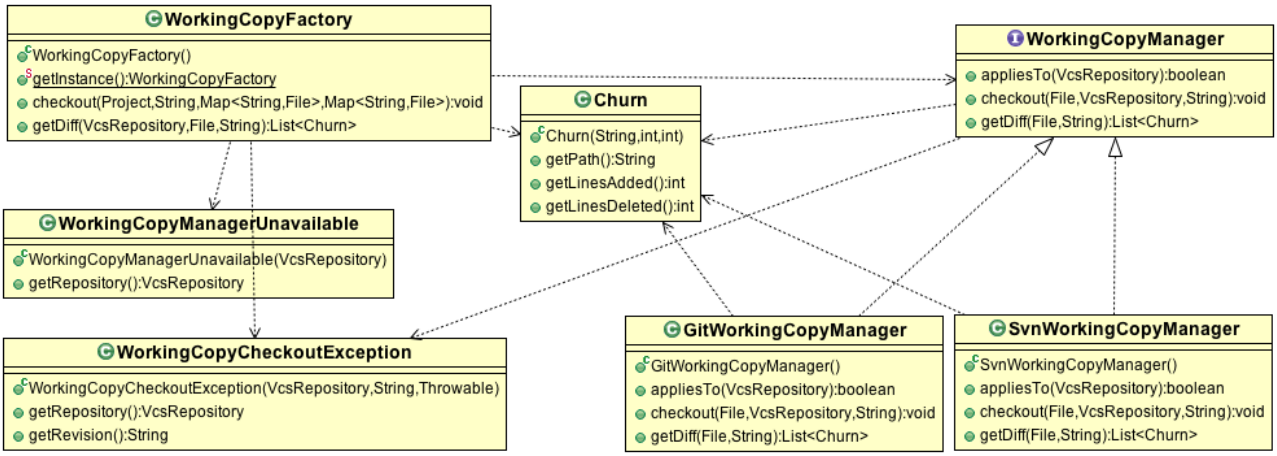

Figure 7: UML Class Diagram showing the design of acquiring working copies and source code differences.

6.1 Requirements

The first observation is that many metrics may reuse the same working copy. Since creating working copies is IO intensive it is necessary to actually implement this reuse.

The second observation is that a single project as analyzed by OSSMETER may reference several VCS repository locations. This implies that several working copies have to be made available to a single metric.

Finally, differencing source code in the past is an expensive (server-side) operation for some VCS systems both in terms of CPU and IO. Care must be taken to make this work as fast as possible.

6.2 Design

6.2.1 IWorkingCopyManager

We decided to make a separate OSGI extension point for obtaining working copies and for comparing these working copies to arbitrary versions on the VCS server side. This extension point is called `workingCopyManager` and its contributions should implement `IWorkingCopyManager` for every specific VCS system. Note that this extension to the platform is a reusable and generic service which is not limited to WP3 or Rascal-based metric providers in any sense.

Figure 7 depicts an UML class diagram. I.e. next to the `IVcsManager` interface delivered by WP2, we add an `IWorkingCopyManager`. Currently we provide two prototype implementations: one for Git and one for SVN.

We decided not to extend the existing `IVcsProvider` interface (which currently provides VCS meta data, and not VCS data) such that the platform implementers can incrementally add more support for different VCSs, and such that different ways of creating working copies and diffs can be easily

experimented with. Perhaps in a future verison of the platform when all revelant providers have been implemented to the two extension points can be merged into a single interface.

6.2.2 Churn

Figure 7 shows that to model the source code differences between versions, we introduced a model class named `Churn`. `Churn` holds meta data of which lines where added or removed. Some VCSs provide this information quite directly in their own machine interfaces for other VCSs this information must be recovered for example by using the Un*x 'diff' tool and parsing its output. In general, the more support from the VCS, the faster the differencing will be.

The `WorkingCopyManager` interface contains a method `getDiff` to retrieve a set of `Churn` objects, in which the mapping from the VCS capabilities to the OSSMETER platform is done. The `Churn` data is later included in the VCS meta data when it is sent to the Rascal function (see Section 7).

In a specific implementation of `getDiff`, we can use the information for the generic `IVcsManager` interface to find out which files have changed, and then run the `diff` functionality of the VCS on the working copy. Or, we may simply run the `diff` functionality of the VCS directly. The choices to be made here are a matter of efficiency and need to be experimented with at scale.

6.2.3 WorkingCopyFactory

Finally, a generic `WorkingCopyFactory` singleton loads `WorkingCopyManagers` using an OSGI extension points and clients call two methods to obtain working copies and to obtain `Churn` objects. To satisfy the requirement of one project holding many repositories: the method to obtain working copies will fill a map to match a working copy for each VCS repository that a project contains.

One client of the `WorkingCopyFactory` is the aforementioned `RascalMetricProvider` (see Section 5).

6.3 Implementation

We currently have working copy managers for SVN and GIT. These two use the commandline API to "checkout" or "clone" repositories into the assigned working copy locations.

For differencing we also use the commandline interfaces of SVN and GIT, directly parsing their output and mapping it to `Churn` objects. We use the existing `VcsProvider` interfaces for both to acquire meta information about code committers.

6.4 Conclusion and future work

We extended the platform with an incrementally extensible way of obtaining working copies and differencing them. Support for other VCSs can be added easily. The data that is produced by this part of the design will be consumed by Rascal metric providers as described in Section 5.

In the future we will rewrite the commandline based versions of the working copy providers by Java based versions, for uniformity and deployment reasons.

7 Extracting meta data from VCS systems

7.1 Requirements

In principle most VCS systems store a first class representation of what is different between consecutive versions of a project. The systems facilitate looking back in history to help a programmer understand the cause of a systems behavior, or to facilitate releasing an update of a previously released version, and last but not least, to share versions between project team members and users of the project. Most VCSs also support authorship information, on different levels of granularity. A "commit" is generally understood as the act of a single person storing a new version of the project in a central VCS.

Please note the VCS jargon is inconsistent between different VCSs and synonyms do introduce confusion. A "commit" in SVN is different from a "commit" in GIT. A "committer" is an "author" in SVN while in GIT "author" may be somebody else from the "committer". In CVS each file has a different "version" number, while in SVN a whole project shares the same "version" number.

The goal of the VCS model in OSSMETER is to unify as much as possible the jargon into a common vocabulary and API. This will allow metrics using this API to work independently from the specific VCS. At the same time, the VCS model will allow specific features for specific VCS systems, so as to not loose information which may be relevant to the quality of an OSS project.

For Task 3.2 we wish to have all meta information about a version of a project readily available next to the full source code to be able to mix and match metrics on both levels.

7.2 Design and Implementation

The object-oriented part of the design and implementation of acquiring meta data from VCSs is described in Deliverable 5.3 [39]. This component is a core of the integration effort since much of OSSMETER is centered around the versions of the projects that it is analyzing.

The infra-structure described in Deliverable 5.3 produces a unified view on the meta data provided by a VCS. Most importantly, it delivers a `ProjectDelta` model which contain `RepositoryDeltas` for every repository which contain `CommitItems` which eventually link to source code deltas and author identities.

This is the basic meta data that can be combined with source code analysis or summarized in itself to generate metrics. To easily integrate source code analysis with VCS meta data analysis, we wish to implement all metrics in the same context. This is the Rascal language. Hence the VCS delta information is transformed to Rascal data types and passed as a parameter to each metric function (see above).

The mapping from object-oriented delta models to function delta models in Rascal is clear and simple. Since the OO model is basically a container tree from Project, via Repository to CommitItem and Author, the entire model can be mapped naturally to abstract data types with an occasional embedded list (see Figure 9 for the target Rascal model and Figure 8 for the class that executes the conversion).

Notice from Figure 9 that the information about `Churn` is directly integrated in the meta model. This is produced by the `WorkingCopyProvider` extensions for SVN and GIT (explained earlier) for reasons of efficiency and reuse.

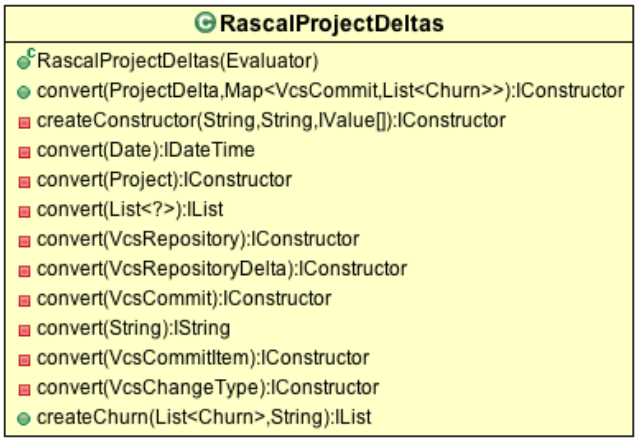

Figure 8: Converting the VCS Delta model to Rascal data types using a method for each level in the input model.

7.3 Conclusions and Future Work

The basic facts about what is happening to the source code are represented precisely in the `ProjectDelta` model.

The `Churn` data is a priori aggregated to number of lines added and deleted per commit item. In the future we may go down one level of abstraction, i.e. which lines were added and which were changed. This can enable a tighter integration with other source code metrics, but for now there are no requirements to be satisfied with such a more detailed model which can not be satisfied otherwise.

```
1  data ProjectDelta
2    = projectDelta (datetime date, Project project, list [VcsRepositoryDelta] vcsProjectDelta )
3    | \empty();
4
5  data Project = project (str name, list [VcsRepository] vcsRepositories );
6
7  data VcsRepository = vcsRepository(str url );
8
9  data VcsRepositoryDelta
10   = vcsRepositoryDelta(VcsRepository repository, list [VcsCommit] commits, str lastRevision );
11
12 data VcsCommit
13   = vcsCommit(datetime date, str author, str message, list [VcsCommitItem] items, str revision );
14
15 data VcsCommitItem = vcsCommitItem(str path, VcsChangeType changeType, list[Churn] churns );
16
17 data VcsChangeType = added() | deleted () | updated() | replaced () | unknown();
18
19 data Churn = linesAdded(int i) | linesDeleted (int i );
```

Figure 9: Rascal model for VCS project deltas, mirroring OO VcsDelta model [39].

Compliance
Full
Partial
None

Table 2: Coding scheme for compliance.

Priority
SHALL
SHOULD
MAY

Table 3: Coding scheme for priority.

8 Satisfaction of OSSMETER Requirements

8.1 Summary

In this section we report on whether or not the OSSMETER requirements for WP3 have been met by delivering Task 3.1 and 3.2, and what needs to be done further to complete WP3. In general, the upfront conclusion is that:

- The infra-structure is completely functional, and enables both satisfying current requirements as well as unexpected new requirements.
- The metrics align with the requirements, as far as possible.
- Aggregation and visualization is partially satisfied, and future work remains.
- Delta model providers for more VCS systems are necessary in the future.
- M3 model providers for more programming languages are necessary in the future.
- AST-based coding convention checkers are future work for Deliverable

For meeting the end goals for the actual use cases, we identified that the basic metrics are indeed available or being made available, and at the same time there will be a need for good dynamic aggregation (meaning while the user is browsing the data) over time frames. To meet the requirements of the end user, there will also be a need to correlate metrics from different providers relating them on the time axis [**?**].

8.2 Detailed requirements from Deliverable 3.1

We use the coding scheme shown in Table 2 and Table 3. 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	Partial: M3 provides language agnostic metrics after producing the model. Some basic language agnostic metrics need to be added to cover for "unknown" languages so as to produce some meaningful volume metrics at least.
14	Fact extractors shall be available that extract from source code the facts that are needed for computing language-agnostic metrics.	SHALL	Full	Partial: For Java the fact extractors are complete, for PHP there is an initial prototype, for the unknown language files we will add a basic fact extractor.
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. We plan to define PHP metrics.	Partial: a prototype M3 provider exists for which severable M3-based metrics work. Evaluation and extension is still under way.
18	The facts needed to compute language-specific metrics for other languages (PHP, Python, C) may be extracted.	MAY	Partial. We planned to extract PHP metrics.	Partial: Partial: a prototype M3 provider exists for which severable M3-based metrics work. Evaluation and extension is still under way.
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: committers per file
23	Development activity may be measured by the ratio between old and new committers.	MAY	Full	Partial: basic fact extraction done, but need to interpret what this metric means for the future.

24	History of some metrics should be captured to summarize quality evolution during development.	SHOULD	Full	Full: historical metric providers are an integral part of the platform.
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	None: we don't know how to measure comment quality internally, but we may introduce quality of the distribution of comments over the code.
35	Provide a well-structured code index for the OSS project	SHALL	Full	None: this is future work, but the Rascal libraries contain precise abstract file system model and extraction API already, and M3 models contain full mappings between logical entities and their physical representation on disk. Satisfying this requirement will help linking back metric results to source code.
36	Provide a rating of the use of advanced language features for the OSS project	SHOULD	Full	Partial: this is future work, but half of it is done. The key enabler is to be able to detect each language feature precisely. The AST model in M3 satisfies that requirement, which we have for PHP and Java.
37	Provide a rating of the use of testing cases for the OSS project	SHALL	Partial	None: In principle, test cases have to be executed to determine this. We have to explore how a weaker, but meaningful, metric can be defined. An simple idea is to test for the existence of test cases and their relation to the source code. The M3 models for test files would contain explicitly dependencies on the to be tested code [3].
38	Provide an indicator of the possible bugs from empty try/catch/finally/switch blocks for the OSS project	SHALL	Full	None: future work

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	None: future work, based on M3 ASTs
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	None: future work, based on M3 ASTs
42	Provide an indicator of suboptimal code from wasteful String/StringBuffer usage for the OSS project	SHALL	Partial	None: future work based on M3 ASTs, and clarification required.
43	Provide an indicator of duplicate code by detecting copied/pasted code for the OSS project	SHALL	Full	None: future work based on M3 models for language specific clone detection and (re)use text-based clone detection tools otherwise.
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	Partial: fact extraction done, but compliance testing is future work.
46	Provide an indicator of the limit of the number of function parameters and line lengths for the OSS project	SHALL	Full	Partial: fact extraction done.
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.	None: to be done, as simple word bag comparison of the first comment in a source file may work.
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	None: future work to find out exactly what the implications of this requirement are.
50	Provide an indicator of the use of good practices of class construction for the OSS project	SHALL	Partial	Partial: basic fact extraction done, in terms of M3 ASTs. Coding conventions still need to be formalized.
51	Provide an indicator of the use of multiple complexity measurements, among which expressions for the OSS project	SHALL	Full.	Partial: basic complexity measures are in place, but an aggregation is still to be explored.
52	Provide an indicator of the cyclomatic complexity for the OSS project	SHALL	Full	Full

9 Summary, Conclusions and Future Work

9.1 Summary

For Task 3.2 we revisited results from Task 3.1 to streamline the addition of new metrics. More importantly, we introduced:

- reusable platform support for multi-repository working copy creation;
- reusable platform support for differencing source code line-by-line;
- language agnostic activity metrics on top of VCS meta data from WP2;
- language specific activity metrics on top of existing software quality metrics from WP3;
- Gini coefficient based aggregation of source code activity.

9.2 Conclusions

The number of new metrics produced for Task 3.2 was not big, but the platform support underneath it all for WP3 required significant engineering effort. The platform is ready now for test driving on larger projects and for larger numbers of revisions. Initial studies can be done using the platform investigating open questions in software engineering.

9.3 Future Work

We expect to invest in fine-tuning and optimizing the platform and its metrics before the first full evaluations of its functionality in M24. For Tasks 3.4 and Tasks 3.5 (language agnostic and language specific metrics) we have a good number of prototype metrics, but we expect that initial evaluations will provide feedback and result in a round trip.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, , and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson, 2006.

[2] Jarallah S. Alghamdi, Raimi A. Rufai, and Sohel M. Khan. Oometer: A software quality assurance tool. In *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering (CSMR'05)*, pages 190–191, 2005.

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

[4] T.L. Alves, C. Ypma, and J. Visser. Deriving metric thresholds from benchmark data. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10, Sept 2010.

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

[6] Martin Brandtner, Emanuel Giger, and Harald Gall. Supporting continuous integration by mashing-up software quality information. In *IEEE CSMR-WCRE 2014 Software Evolution Week (CSMR-WCRE)*, pages 109–118, Antwerp, Belgium, February 2014. IEEE.

[7] M. Bruntink. Testability of object-oriented systems: a metrics-based approach. Master's thesis, 2003.

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

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

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

[11] N. Fenton. Software measurement: A necessary scientific basis. *IEEE Transactions on Software Engineering*, 20(3):199–206, 1994.

[12] 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.

[13] Paul Klint. Using Rscript for Software Analysis. In *Working Session on Query Technologies and Applications for Program Comprehension (QTAPC 2008)*, 2008.

[14] Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering Methodology*, 14(3):331–380, 2005.

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

[16] J.M. Kraaijeveld. Exploring Characteristics of Code Churn. Master's thesis, TU Delft, 2013.

[17] Adrian Kuhn and Toon Verwaest. FAME, a polyglot library for metamodeling at runtime. *In Workshop on Models at Runtime*, pages 57–66, 2008.

[18] Meir M. Lehman. Programs, life cycles, and laws of software evolution. *Proc. IEEE*, 68(9):1060–1076, September 1980.

[19] Rüdiger Lincke, Jonas Lundberg, and Welf Löwe. Comparing software metrics tools. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 131–142, New York, NY, USA, 2008. ACM.

[20] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics*. Prentice Hall, 1994.

[21] M. Markus, B. Manville, and C. Agres. What makes a virtual organization work? *Sloan Management Review*, 42(1):13–26, 2000.

[22] Anna-Liisa Mattila and Tanja Mehtonen. Measuring open source software success and recognising success factors. *Open Source Software Development Seminar 2013*, 2013.

[23] 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.

[24] Tom Mens and Serge Demeyer. *Software Evolution*. Springer-Verlag, 2008.

[25] Richard J. Miara, Joyce A. Musselman, Juan A. Navarro, and Ben Shneiderman. Program indentation and comprehensibility. *Commun. ACM*, 26(11):861–867, November 1983.

[26] V. Midha and P. Palvia. Factors affecting the success of open source software. *Journal of Systems and Software*, 85(4):895–905, 2012.

[27] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of open souce software development: Apache and mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):309–346, 2002.

[28] K. Nakakoji, Y. Yamamoto, Y. Nishinaka, K. Kishida, and Y. Ye, editors. *Evolution Pattters of Open-Source Software Systems and Communities*, Proceeding of 2002 International Workshop on Principles of Software Evolution, 2002.

[29] Thomas M. Pigoski. *Practical Software Maintenance: Best Practices for Managing Your Software Investment*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[30] W. Poncin, A. Serebrenik, and M. van den Brand. Process mining software repositories. In *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*, pages 5–14, March 2011.

[31] E.S. Raymond. *The Cathedral & The Bazaar*. O'Reily, 2001.

[32] S. Demeyer S. Tichelaar, S. Ducasse and O. Nierstrasz. A meta-model for language-independent refactoring. In *Proc. Int'l Sym. Principles of Software Evolution*, pages 157–169. IEEE Computer Society, 2000.

[33] C. M. Schweik and . English. Perliminary steps toward a general theory of internet-based collective-action in digital information commons: Findings from a study of open source software prrojects. *Internal Journal of the Commons*, 7(2):234–254, 2013.

[34] C.M. Schweik and R. English. Identifying success and abandonment of free/libre and open source (floss) commons: A premilinary classification of sourceforge.net projects. *The European Journal fo the Informatics Professional*, 7(6):54–59, 2007.

[35] M. Scotto, A. Sillitti, G. Succi, and T. Vernazza. A relational approach to software metrics. In *Proceedings of the Software Applied Computing (SAC'2004)*, pages 1536–1540, March 2004.

[36] K.J. Stewart, A.P. Ammeter, and L.M.Maruping. Impacts of license choice and organizational sponsorship on user interest and development activity in open source software projects. *Information Systems Research*, 17(2):126–144, 2006.

[37] C. Subramaniam, R. Sen, and M.L. Nelson. Determinants of open source software project success: A longitudinal study. *Decision Support Systems*, 46(2):576–585, 2009.

[38] SWAT research group. *D3.1 - Domain Analysis of OSS Quality Attributes*. Technical report, Centrum Wiskunde & Informatica, March 2013.

[39] University of York. *D5.3 - Component Integration (Interim)*. Technical report, University of York, March 2013.

[40] Mark van den Brand, Paul Klint, and Jurgen J. Vinju. Term rewriting with traversal functions. *ACM Trans. Softw. Eng. Methodol.*, 12(2):152–190, 2003.

[41] R. Vasa, M. Lumpe, P. Branch, and O. Nierstrasz. Comparative analysis of evolving software systems using the gini coefficient. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 179–188, Sept 2009.

[42] Kurt D. Welker, Paul W. Oman, and Gerald Atkinson. Development and application of an automated source code maintainability index. *Journal of Software Maintenance*, 9(3):127–159, 1997.

[43] Ken Wong. *Untangling safety-critical source code*. PhD thesis, University of British Columbia, 2005.

[44] Aiko Yamashita and Leon Moonen. To what extent can maintenance problems be predicted by code smell detection? - an empirical study. *Information and Software Technology*, 55(12):2223–2242, 2013.