Code quality measurement: case study

Dmitrii Savchenko, Timo Hynninen and Ossi Taipale

October 26, 2021

Code quality measurement: case study

D. Savchenko[*], T. Hynninen[*] and O. Taipale[*]
[*] Lappeenranta University of Technology, Lappeenranta, Finland
dmitrii.savchenko@lut.fi, timo.hynninen@lut.fi, ossi.taipale@lut.fi

***Abstract* - As it stands, the maintenance phase in the software lifecycle is one of the biggest overall expenses. Analyzing the source code characteristics and identifying high-maintenance modules is therefore necessary. In this paper, we design the architecture for a maintenance metrics collection and analysis system. As a result, we present a tool for analyzing and visualizing the maintainability of a software project.**

Keywords - maintenance, code quality

I. Introduction

Maintenance and upkeep is a costly phase of software life cycle. It has been estimated that maintenance can reach up to 92% of total software cost [1]. Code quality can be analyzed using various existing metrics, which can give an estimate on the maintainability of software. There are several tools and frameworks for assessing the maintainability characteristics of a project. Many tools are included in integrated development environments (IDEs), such as Eclipse metrics [2], JHawk [3] or NDepend [4]. As such the existing tools are specific to platform and programming language, providing quality analysis during development. Considering maintenance also includes activities post-release of a software product, it would be beneficial to perform quality measurement also in the maintenance and upkeep phase of life cycle.

One solution to the post-release monitoring are online data gathering probes, which can be inserted into production code to gather runtime performance data. In order to establish and sustain a commitment for maintenance measurement this work introduces a design for data collection and storage. In this paper we present an architecture for systematically collecting code metrics for maintenance. Additionally, the visualization and analysis of the metrics are explored.

In this study we will focus on the analysis of web-applications. This delimitation is due to the collection of runtime metrics as well as static metrics. The focus on web-applications provides a reasonably standardized measurement interface for runtime performance through the browser's web API. In this paper we also propose the design and implementation of the system called Maintain. The probes for gathering metrics in the system are implemented in both JavaScript and Ruby programming languages.

Rest of the paper is structured as follows. In Section 2, related work in analyzing software maintainability is introduced. Sections 3 and 4 presents the architecture and our implementation for a metrics collection and analysis system, which is main contribution of this work. Evaluation of the system's performance and utility is presented in Section 5. Finally, discussion and conclusions are given in Section 4.

This study was funded by the Technology Development center of Finland (TEKES), as part of the .Maintain project (project number 1204/31/2016).

II. Related research

Software maintenance, as defined by ISO 14764 standard, is the "the totality of activities required to provide cost-effective support to a software system", consisting of activities both during development and post-release [5]. The analysis of software maintainability is by no means a novel concept. Motogna et al. [6] presented an approach for assessing the change in maintainability. In [6], metrics were developed based on the maintainability characteristics in the ISO 25010 software quality model [7]. The study presents how different object oriented metrics affect the quality characteristics.

A study by Kozlov et al. [8] distinguished that particular code metrics (data variables declared, McClure Decisional Complexity) have strong correlations with the maintainability of a project. In the work, the authors analysed the correlation between maintainability and the quality attributes of a Java-project.

In the study by Heitlager et al. [9] a practical model for maintainability is discussed. The study discusses the problems of measuring maintainability, particularly with expressing maintainability as a single metric (Maintainability index).

Studies where different evaluation methods are combined in order to get a more thorough view on the maintainability of a project have been conducted during the past decade. For example, Yamashita [10] combined benchmark-based measures, software visualization and expert assessment. In a similar vein, Anda [11] assessed the maintainability of a software system using structural measures and expert assessment. In general, these studies suggest that visualization systems providing developers and project managers with an analysis of the health of a software project can help distinguish problematic program components, and thus help in the maintenance efforts of software.

III. Architecture

Maintain system architecture is presented at the figure 1. System consists of the following components:

- *Probe* is a program that gathers some valuable data from the software (static or dynamic). Each probe should have an associated analyzer;
- *Data Storage* – data storage that stores the raw data from the probes. It also has REST interface that receives the data from the probes;
- *Analyzer* is a program that gets the raw data from the associated probe and creates a report, based on this data;
- *Report Storage* – data storage that stores reports from analyzers;
- *Report Visualizer* is a component that creates a visual representation of the report.

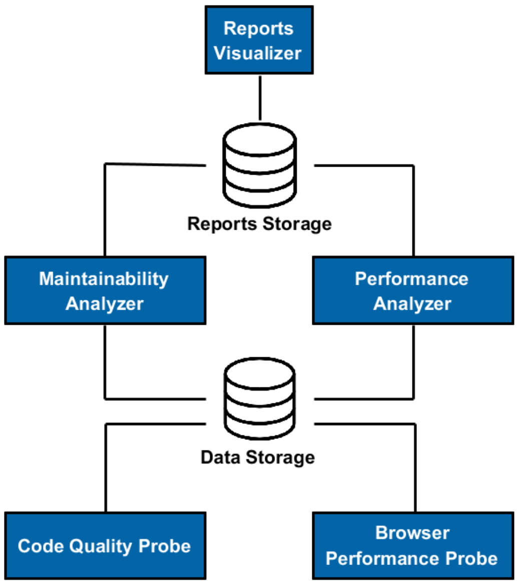


Figure 1. System architecture

Workflow of the system is centered around the Data Storage. Generally, it looks like this:

- Probes gather the information from the source code, it might be some static analysis results or dynamic performance data;
- Gathered and normalized data is sent to the Data Storage. Probe can have different data types, data structure is defined by analyzer;
- When new data is received by Data Storage, the associated analyzer is called. It requests the data from the Data Storage, produces report (object, that contains current status of the analyzed application aspect and a set of time series for the end user);

When end user requests the report, Reports Visualizer generates a visual representation of the time series, that were created by analyzers.

IV. Implementation

Maintain system was implemented using Ruby on Rails framework and hosted on Heroku cloud platform. Project details page is shown on Figure 3. This page provides the information about the current state of the project, that is described as a set of 8 scores, based on quality characteristics, described in ISO/IEC 25010 [7]. Those scores are visualized as a polar chart with 8 axis for each quality characteristic respectively. Score calculation is based on the report statuses – each report has an associated probe, and each probe has a set of associated quality characteristics. Quality characteristics are set by the project administrator.

System class structure is organized as pictured in figure 2. As system gathers the data using REST API, it is generally impossible to predefine all possible probes and probe types and set their quality characteristics in advance. That's why we decided to let user define the quality characteristics for probe when it is created or modified. Result score is based on statuses of last reports for each probe respectively.

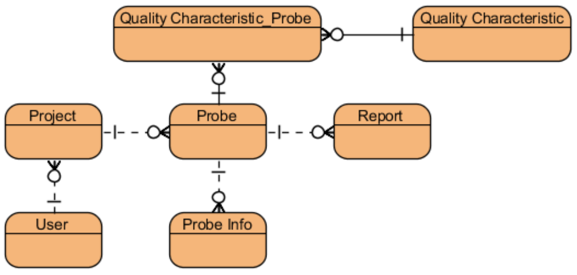


Figure 2. System entity-relationship diagram

A. Probes

As a case study, we have implemented four probes: HAML, JavaScript and Ruby code quality probes, and browser performance probe. JavaScript and Ruby code quality probes are based on maintainability index, which is calculated using the following formula:

```
maintainability = 171 -
                (3.42*Math.log(effort))-
                (0.23*Math.log(cyclomatic))-
                (16.2*Math.log(loc))
```

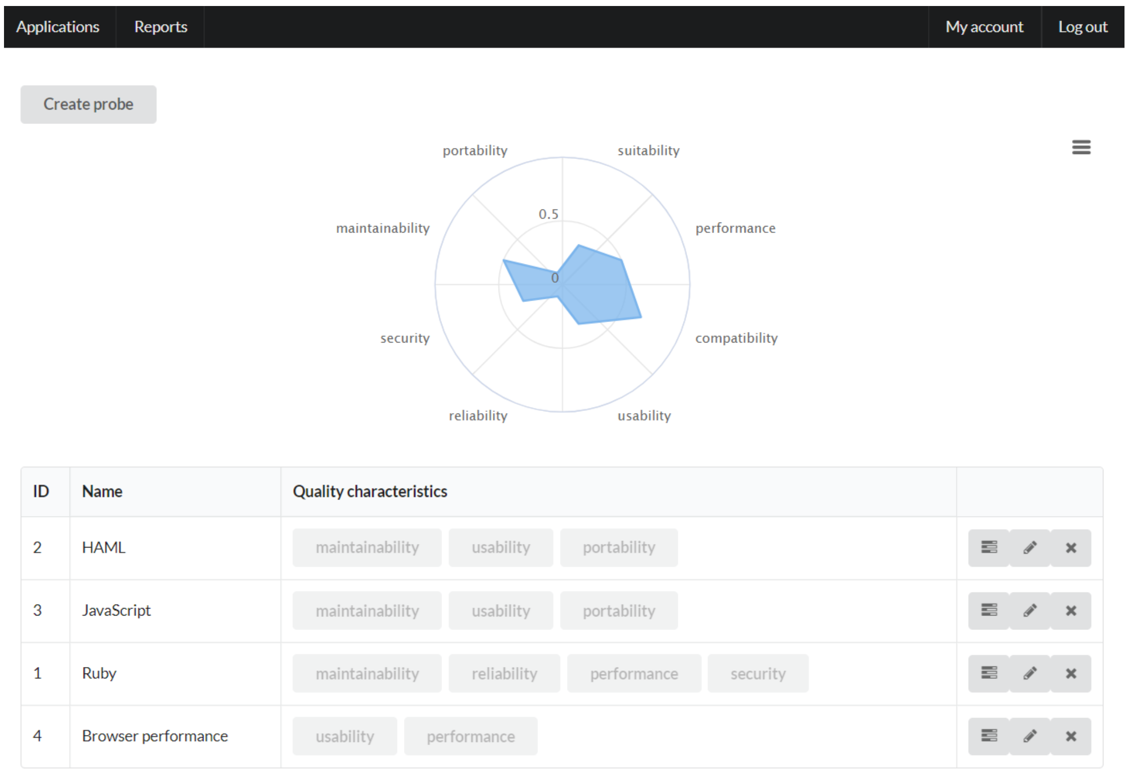
HAML maintainability index uses recursive formula, based on linter report:

```
Maintainability = a*maintainability
```

where **a** is 0.9 for linter error and 0.99 for linter warning

Code quality probes produce the following data for Data Storage:

```
{
    maintainability: M,
    revision: R,
    datetime: D,
    modules: Ms
}
```



ID	Name	Quality characteristics	
2	HAML	maintainability usability portability	
3	JavaScript	maintainability usability portability	
1	Ruby	maintainability reliability performance security	
4	Browser performance	usability performance	

Figure 3. Project page example

where M is average maintainability index for the whole project, R is current Git revision, D is current date and time, and Ms is a list of maintainability index for project files and their names. Browser performance probe generates report in different format:

```
{
    page: P,
    timing: T,
    datetime: D
}
```

Where P is an URL of the current page (without query), T is the time between page load start time and DOM ready event time in milliseconds, and D is current date and time.

B. Analyzers

Currently we have implemented two different analyzers - maintainability analyzers for Ruby, JavaScript and HAML probes, and performance analyzer for browser performance probe. Workflow for maintainability analyzer works is described below:

- Data from Data Storage is grouped by days, maintainability index for each day calculated as a median of indices for day. If no data presented for day, analyzer sets the value for the previous day (fallback for weekends);
- List of maintainability indices are smoothed using exponential moving average method, those values are used as a time series for visualizer;
- Linear regression for last five days is used as a status of the project source code quality: if it is less than zero, then code quality is bad.

Workflow for browser performance analyzer is different:

- Performance data is grouped by five minutes, value for each section is calculated as a 95th percentile of all values for section;
- If values for all sections are less than 2 seconds, then browser performance is good.

V. Maintain System Usage Example

Maintain system was evaluated using a proprietary web application, that was implemented using Ruby on Rails as a backend, and CoffeeScript on top of React.JS as a frontend. This project is on maintenance phase, so we decided to analyze historical data and compare Maintenance system results with the feedback from the project manager, who managed the analyzed project. Application was used by 5 administrators and about 10000 users. Maintenance system was deployed in Heroku cloud, while probes were running on local PC, that had 1.8 GHz 2-core CPU and 4 Gb RAM. We gathered the code quality information for all the previous commits to make picture more consistent.

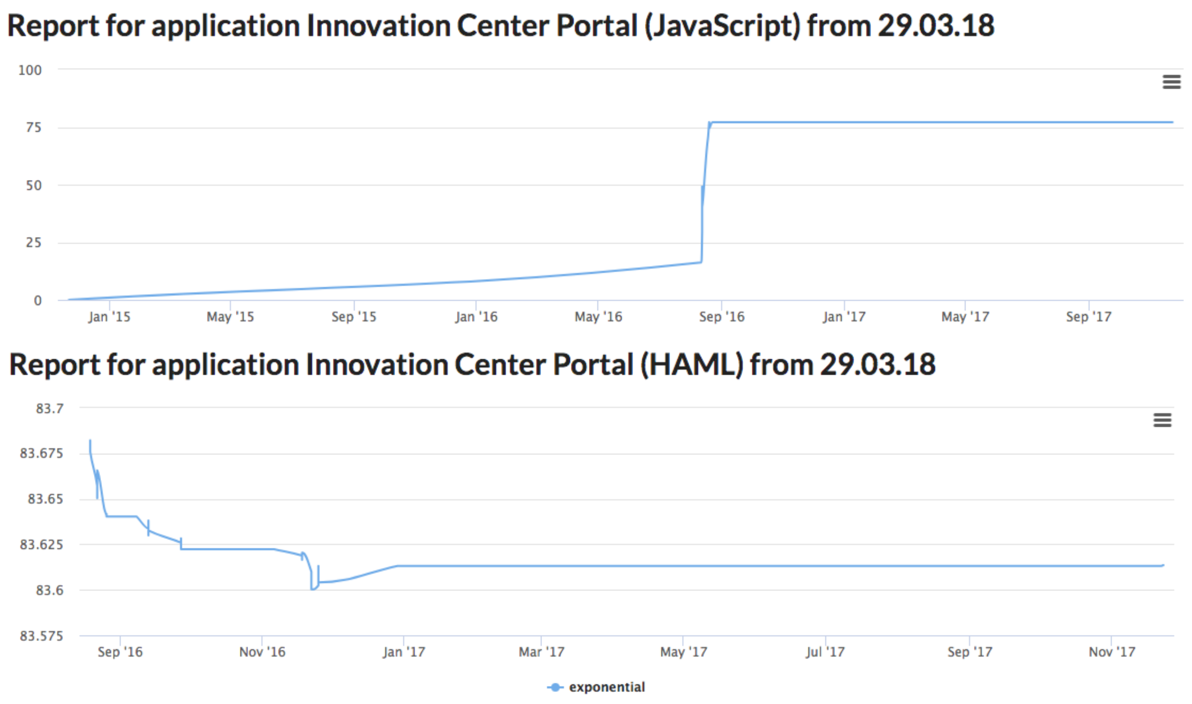


Figure 4. Project code quality measurements

Figure 3 illustrates the general 'health' of the analyzed application at the last Git revision at Master branch. Figure 4 shows the JavaScript (CoffeeScript) and HAML code quality. The project was started as a pure backend solution, while frontend development started at the beginning of September 2016. As shown in the graph, HAML code quality was decreasing from September 2016, until December 2016, then it was stable. This behavior can be explained by a deadline of the project, that was at the end of the year 2016. After the deadline, the project active development stopped. Project manager evaluated the results and stated, that such an 'early warning' system could notify the team and save some development resources.

VI. Discussion and Conlusion

The objective of this study was to facilitate the systematic collection and analysis of maintenance metrics, in order to reduce the effort required in the maintenance phase of software already during development. To realize the goal we designed and implemented an architecture for a system which can be used to collect both static and runtime metrics of a software project. We then implemented analysis tools to visualize these metrics, and display the most high-maintenance modules in a project repository.

The novelty of the presented work is the extendibility and modularity of the architecture. The architecture is not platform specific. New probes and corresponding analyzers can be added at any stage, using the REST API with any programming language or platform. The data storage and reporting system provide a common interface for the systematic collection of quality metrics, allowing the developers of a project to establish and sustain a commitment for quality measurement.

Providing a platform to establish the measurement commitment is important, because previous research shows that the quality assurance and testing practices of developers do not necessarily line up with measurement possibilities distinguished in academic research. For example, the recent study by Garousi and Felderer distinguishes that the industry and academia have different focus areas on software testing [12]. Likewise, Antinyan et al. show in [13] that existing code complexity measures are poorly used in industry. In this work, we used the maintainability index as an indicator for code quality, as it has been used in both academia and industry. In future, we should work on evaluating whether quality metrics presented in academic publications could be implemented into our system as probes providing reliable measurements.

Additionally, in future work we aim to develop more measurement probes in the system. We should evaluate the different metrics to distinguish which measurements provide the most useful information about software maintainability.

References

[1] "The Four Laws of Application, Total Cost of Ownership." Gartner, Inc., 2012.

[2] Eclipse Metrics Plug-in, http://sourceforge.net/projects/metrics. (accessed 5th Feb 2018).

[3] JHawk, http://www.virtualmachinery.com/jhawkprod.htm. (accessed 5th Feb 2018).

[4] NDepend "http://www.ndepend.com", (accessed 5th Feb 2018).

[5] ISO/IEC, "ISO/IEC 14764: Software Engineering - Software Life Cycle Processes - Maintenance." 2006.

[6] S. Motogna, A. Vescan, C. Serban, and P. Tirban, "An approach to assess maintainability change," in 2016 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR), 2016, pp. 1–6.

[7] ISO/IEC, "ISO/IEC 25010: Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models." 2011.

[8] D. Kozlov, J. Koskinen, J. Markkula, and M. Sakkinen, "Evaluating the Impact of Adaptive Maintenance Process on Open Source Software Quality," in First International Symposium on

Empirical Software Engineering and Measurement (ESEM 2007), 2007, pp. 186–195.

[9] I. Heitlager, T. Kuipers, and J. Visser, "A practical model for measuring maintainability," in Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the, 2007, pp. 30–39.

[10] A. Yamashita, "Experiences from performing software quality evaluations via combining benchmark-based metrics analysis, software visualization, and expert assessment," in 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2015, pp. 421–428.

[11] B. Anda, "Assessing software system maintainability using structural measures and expert assessments," in Software Maintenance, 2007. ICSM 2007. IEEE International Conference on, 2007, pp. 204–213.

[12] V. Garousi and M. Felderer, "Worlds Apart: Industrial and Academic Focus Areas in Software Testing," IEEE Software, vol. 34, no. 5, pp. 38–45, 2017.

[13] V. Antinyan, M. Staron, and A. Sandberg, "Evaluating code complexity triggers, use of complexity measures and the influence of code complexity on maintenance time," Empirical Software Engineering, pp. 1–31, 2017.