Open Source Software Information Triangulation: A design Science Study

Fons Wijnhoven
University of Twente
Enschede, The Netherlands *fons.wijnhoven@utwente.nl*

Friso Kluitenberg
HighPerformer.com
Enschede, The Netherlands *friso.kluitenberg@hotmail.com*

Maya Daneva
University of Twente
Enschede, The Netherlands *m.daneva@utwente.nl*

Abstract

Open source components are a promising way for creating and delivering software to the market fast. However, challenges arise when assessing the quality of open source software. While frameworks to assess these components exist, the open source market is neither governed nor regulated and the use of these frameworks is labor-intensive and complex. This research aims to solve this problem by selecting quality indicators for open source software on GitHub and realizing a tool for automatically supporting the evaluation of information about open source software from other available sources. These sources include StackExchange.com for external support and the National Vulnerability and Exposure database for security incident history. Feedback on the developed prototype supports our view that automatic checks of open source software claims is possible and useful.

Keywords: Opensource software, GitHub.com, software quality assessment, information triangulation, design science

1. Introduction

Increasingly more open source software (named OSS in the rest of this paper) is used in products and services [6]. OSS effectively allows to decrease time to market and avoid reinventing the wheel. With the vast amount of available OSS there are risks as well. The OSS market is neither governed nor regulated. However, even though these structures are not in place, the nature of OSS allows for its transparent evaluation. Currently, this is a time-consuming process and thus there is a need for guidelines and an accompanying tool to assess the quality of these components. The present paper responds to this situation. More specifically, we set out to achieve the goals of identifying suitable indicators of OSS quality and to design a related tool that supports the evaluation of OSS but multiple data sources. The use of multiple data sources for opinion formation is what we call information triangulation, and thus we aim at an information triangulator, not an answer machine [24].

OSS are used by a very broad audience, ranging from hobbyists to developers working in enterprises. This translates to a large set of requirements and general recommendations. These OSS components are available through multiple platforms, but to make our study manageable we focus on GitHub, the leading platform for open source components. GitHub also has an Application Programming Interface (API) for easy data consumption and collection of meta-data useful for its assessment. Therefore, the following research question is leading our study: *What are the characteristics of an information triangulator that can improve a user's understanding of the quality of an open source software component?*

In the OSS marketplace, many data sources give potentially valuable information for assessing the information on OSS from GitHub. Professional Online Communities exchange knowledge with varying degrees of professionalism, e.g. LinkedIn groups,

StackExchange.com, and Medium.com. The main difference among these platforms is the level to which discussions are moderated and formalized. These communities often provide a base for additional services as well, such as connecting with businesses looking to hire or establish authorities in a professional area. Open Data Sets provided large companies, government agencies or other parties give answers on questions like (i) Which server or pre-processor versions should be supported? (ii) What are common problems associated with using certain licenses? (iii) Which trends are emerging with regards to cloud computing? The main problem with these data sets is that they require access, resources or expertise to sift through them and find meaning. Online Discussion Boards are moderated platforms by users help each other for reward points or prestige. These boards are tools to both find professional solutions and components, and to support customers in buying decisions [13]. Discussion boards exist in all niches one can think off, e.g. there are boards with a focus on evaluating hosting providers and architecture such as Webhosting Talk. Numerous forums for entrepreneurs are available, like Digital Point. In addition, for software developers, forums like Webmaster Talk exist. Network Collaborations [16] by small and medium-size enterprises (SMEs) share their knowledge on OSS solutions. However, retrieving useful information in a useable form is not unproblematic, e.g. prior research [10] found six challenges with information retrieval in these networks: document understanding, locating relevant architectural knowledge, support for traceability between different entities, support for change impact analysis, assessment of design maturity, and credibility of information.

As we see, lots of open platforms and data sources exist online, but to the best of our knowledge, very few lend themselves to easy and accurate parsing.

2. Research Methodology

In order to allow the automatic support of OSS quality checks, tools need to be developed. Therefore, this research will be a design science study. Following Walls et al [23] and Arazy et al [4], this requires first a search of the literature to identify kernel theories that can guide the selection of requirements, design components, and testable propositions for validating the design. The kernel theories in this context are IT quality frameworks that give relevant indicators of software quality. After establishing this foundation, we design, develop and validate a tool for assessing the quality of open-source components. The design science research model [17] prescribes the following six research steps:

1. Problem Definition & Motivation. The problem definition and motivation has been explored in the previous section. The main problem is that the use of all relevant quality indicators is too labor-intensive to be performed manually only.
2. Objectives of a solution. The objective of the solution is to automatically evaluate software quality and list any ‘red flags’ it encounters. The requirements for this solution are described in section 3.
3. Design & Development. A tool will be developed according to the requirements, the final design and development is addressed in section 4.
4. Demonstration. Section 4.2 demonstrates the tool by its output screens.
5. Evaluation. Section 5 will evaluate the answer to our research question.
6. Communication. The developed tool is made public and included in the project's repository on GitHub.com.

3. Design Theory and Meta-Requirements

This section first describes quality criteria for OSS. Next, we analyze what information GitHub already provides, maybe with the use of some tools. Finally, we discuss the requirements for our tool.

3.1. OS Quality

Adewumi et al [1] identify multiple OSS quality criteria on basis of the ISO/IEC 25010 standard. Because of the high importance of risk and security issues nowadays in information systems, we add some insights on risks to this list.

ISO/IEC 25010 has eight quality categories:

1. Functional suitability is about completeness, correctness and appropriateness. Both implied and explicitly stated functionality should match actual functionality.

2. Performance efficiency consists of (i) time behavior, e.g. processing speed, (ii) resource utilization, e.g. memory and (iii) communication usage, and capacity, e.g. CPU processor capacity.
3. Compatibility describes how well OSS can expose information to other components and systems. It consists of two aspects: co-existence with other applications on the same architecture or system and inter-operability.
4. Usability is about how well the implementation of a design matches with the expectations of the user. Sub-criteria for usability are the learning curve required to operate the software, aesthetics, error protection and accessibility.
5. Reliability. Criteria on which reliability can be measured are how available the software is, can it recover well and fast if an error happens and how mature it is. If we look at the reliability of OSS, the infrastructure on which it runs must be taken-into-account as well.
6. Security consists of confidentiality, authentication (user authenticity) and accountability. The main advantage of OSS also gives rise to security issues. Whilst the community can review the source code, malicious actors can use this to find security holes as well. Plus, integrating many OSS components introduces risks for the application as a whole.
7. Maintainability consists of modularity, reusability, testability and modifiability. This is facilitated by OSS by encouraging contributions from their users. If a solution is not maintainable, community support will wither, since forking and contributing to an OSS project will be difficult.
8. Portability describes the ease in which an OSS component can be used in other contexts. It consists of ease of installation, adaptability and replicability dimensions.

Navicasoft has developed an OSS maturity model in 2004 [25]. It determines a weighting factor based on business needs and a maturity score is calculated with all these factors. The model describes six categories on which maturity is assessed: Product Software, Support, Documentation, Training, Product Integration, and Professional Services. Professional services are only offered with very large open source projects, as is the case with Redhat Linux.

Moradini et al. [21] identify various categories of risks associated with OSS. These risks are (1) component integration risks, (2) insufficient quality risks, (3) component operation and maintenance risks, (4) legal risks, and (5) security risks. The component integration risks arise when trying to integrate OSS in a solution and to deploy it. Misjudging those may result in missed deadlines. Insufficient quality risk is concerned with a specific component that fails to meet the solution's requirements. Component operation and maintenance risk addresses the lack of support from either the author or from the community, which may result in an abandoned or outdated component. Another pitfall is technological debt (the so-called code debt [22, 26]), occurring when a component has very low entry barriers or implementation barriers, but other challenges arise down the road. E.g. if the API of a solution isn't well defined, but the solution is easily integrated. In this case, if features need to be added, it will be costly for the organization.

Legal risks are concerned with Intellectual Property (IP) [14, 18] like author liability, rules and rights of commercial exploitation, attribution rights when others start using your software, and rights for the protection of your brand as software producer. License violations can occur in various ways. Developers may copy code from professional communities such as LinkedIn Groups or Stack Exchange. Copyright can be violated since it is not clear how this code is licensed or if the individual contributing the solution is the author of the copyright. Contributions by either pull request (code contribution by 3rd parties) or online suggestions can include code from these sources. In addition, these pull requests can be added with malicious intent.

Security threats are also an important topic in OSS development and usage [8]. Three risks are touched here:

1. Web-Facing Application Risks. As OSS will most likely be accessible via the internet or via the local intranet, security issues need to be taken into account. These issues range from Cross-Site-Scripting to SQL Injection attacks.
2. Collaborative Aspect. Malicious third parties can use – and are using [9] – the collaborative aspect of free OSS to spread malware and gain unauthorized access to information systems, e.g. an advisory can introduce bugs deliberately or even try to commit a backdoor to the repository.
3. Open Aspect. OSS gives an advisory more insight into possible attack vectors by exposing the source code. The knife cuts both ways since the community can review the code as well and detect possible security issues.

3.2. GitHub Available Quality Indicators

Our next step is to find indicators which can be used to assess quality. The OSS project code file and metadata can be used to assess quality. Most often, metadata is generated from an open source project's code base and published. Kalliamvakou et al [12] and Aggarwal et al [2] identify the following sources: Version Control meta data, platform and community meta data, and platform support & documentation. Below, we describe these briefly.

Version control enables developers to work together on the same software and keep track of changes. In version control, extra information used for collaboration is added to code. The most popular version control system, Git, exposes branches, commits and tags that can be used to assess software quality. A branch diverts the code base from the original version, enabling a developer to make changes based on the main version. Various branching models and naming conventions exist. These are important to make sure the project remains structured, maintainable and bug-free. Git's best practices dictate that developers should never push code directly to a master-branch. Instead, a pull-request should be made and other developers should review the code before it gets merged with the master branch. In addition, features should have their own branch and be prefixed as such. Bug fixes and hot fixes should be labeled in this way as well. Branches should be single purposed with low branching activity [20]. Projects using a good branching strategy will be highly maintainable, since every change, feature and bug-fix is identifiable and merge-able into the main project [11]. A tag is a bookmark for a specific commit, e.g. a release version. Tagging is important for making a distinction between various milestones and versions of a project. How the tag is structured is an important aspect for the package manager. An example of a recommended tagging system is Semantic Versioning which divides a version number up in API-breaking changes, major changes, and minor changes like bug-fixes. Changes in one of these three values can be used to detect how well a project is maintained or how likely it is to introduce API-breaking changes. Analyzing commit messages thus provides insight into maintainability [3].

GitHub.com provides us with much meta-data collected from its users and developers. It also ranks high-quality solutions higher in search results. Plus, it gives a project 'pulse' that indicates how active the project community is. GitHub gives many indicators:

1. Stars. The number of times developers mark the repository as favorite. The number of stars of a repository is also correlated with how often a component is integrated into an application [5] and thus is an indicator for usability.
2. Watchers. The reasons why people 'watch' a repository are mostly to receive updates about future functionality or to receive information about bug fixes. According to Sheoran et al [19] 'watchers' are likely to become contributors in the future as well.
3. Traffic. The number of times that users visit a repository. This is a measure of overall popularity, but does not conclusively give indications about the quality of an open-source component. The source of the traffic (referrer), however, may indicate the demographics of users of the open source component.
4. Clones. The number of times users clone or download the content of the repository. A higher number of clones does not necessarily indicate a higher number of users per se but gives an indication of interest in the OSS.
5. Opinions. If a developer is opinionated, he/she advocates to use the OSS component.

OSS projects allow other members of the platform to support other users, provide bug-fixes or indicate issues. Repository owners and contributors are marked as such in conversations, pull-requests, and issues, so they carry a higher authority. Documentation is essential for a project to be reusable by other developers. Good support and documentation are not a given for every OSS project, therefore it is important to evaluate the volume and depth of documentation and support provided. GitHub provides the following artefacts to aid developers in their evaluation:

1. Link to demos to demonstrate various use cases. Most often, demos increase the ease of implanting a library and seeing the benefits. In addition, it helps shortening the learning curve.
2. A Readme.md file in markdown syntax. This file includes most often, the license, contribution guidelines and any opinions the framework might have. We can detect various sentiments conveyed by this file.
3. A wiki may be used for larger components to convey more use cases and information regarding the component.

4. GitHub pages use Jekyll CMS, which may convey the same information as a Wiki but its input is Markdown syntax and content is stored under version control (git) as well.
5. The presence of a Package.json file. This is a file with meta-data to the repository. It includes build scripts, dependencies and licensing information. In addition, test scripts are also listed in this file.

3.3. Summary of requirements

In this study information from GitHub is evaluated. Various OS quality frameworks have been identified and evaluated. The following categories from current quality control methodologies have been deemed relevant and will be evaluated with the help of the tool: Maintainability, Security, Support, and Documentation. Towards this end and drawing on the sources in the previous sections, we formulate the following requirements for the tool. It should 1) provide an interface to search GitHub.com, 2) evaluate OSS quality accurately as described in the literature review and for each category the literature review deemed relevant, 3) be able to parse and analyze commit messages, 4) be able to parse and analyze the working directory, 5) be able to parse and analyze issues on GitHub.com, and 6) be able to report a quality indication per category

4. Design Propositions of the quality assessment tool

4.1. Goal, scope, input and output

The tool's objective is to help assess OSS components quality. The tool will process the following types of input:

1. OSS component inputs from GitHub. Due to GitHub's version control system a lot of data are available consisting of various types of natural language, structured files, and open source data. These Natural Language files contain for example commit messages, readme's and documentation, which are hard to interpret in an automatic way. However, it is possible to perform pattern and natural language processing onto this data. The Structured Data Files can be interpreted by machines in an exact way and can be good indicators of quality, e.g. built scripts and dependencies.
2. Open Source Meta Data can come from many different sources like support topics on stackexchange.com, security data from the Common Vulnerability and Exposure database and meta-data from GitHub.

As output, the system should be able to produce reports regarding the quality of the software on maintainability, security, support, documentation, and integration risks. In order to check maintainability, we ascertain that the OSS Project does not contain empty commit messages, that it has descriptive commit messages, that it has valid Package.json files, and that it has build scripts (grunt, gulp, phing or other).

For software security there are tools which can test security, called 'F*uzzers* or security testers. This depth of security test is outside the scope of this paper. Our tool will use other metrics, i.e., the tool will access security incident data from the National Vulnerability Database to check any history of security vulnerabilities. The cross-reference is done by a query on a component's name and will then list all the security incident data including a description available.

The tool will triangulate support data with Stack exchange which is a popular professional community for software developers. The data returned will consists of the amount of questions per day and the date of the last question. How well a solution is supported by the community can be estimated by checking if support is provided on Stackoverflow.com, how many contributors are active on GitHub, the number of issues opened, closed and the average issue life time, and the presence of a Wikipedia page about the OSS.

Documentation allows users to solve challenges they face with the OSS. This tool section will quickly evaluate the depth of information the documentation provides. The depth of the documentation will be evaluated by checking the existence of a Readme file and if the readme file has a link to an external documentation or the readme file contains a "getting started", "Usage", "API Section" and a section about how to get support.

The risks found in section 3 will be evaluated with the following criteria per category.

Component integration risk can be assessed by finding how well a developer is able to determine implementation efforts required. Readme files enable to find if use cases for the actual use of the software exist and if the software has a close match with the user's problem.

The presence of a demo allows a third-party developer to quickly judge if the component is a fit for their project. A closer match reduces risks. A package file (package.json or composer.json) is frequently used to describe the dependencies of an open source component. If this file is available, there should also be a section of build scripts that allow a developer to easily rebuild the software from scratch. This allows a party who is using the component, to easily make little or big adjustments. This fact can be ascertained by looking for a 'build' or 'dist' directory

Risks of insufficient software quality entails that a component may not be suited for an intended use case. Morandini identified loss of control which manifests itself in OS projects often through loss of community support. The loss of community support can be measured through search volume on Google Trends [7]. Metrics of Stackexchange are used as an indicator for developers searching for alternative solutions on external platforms. In addition, risk of insufficient quality lends itself for static code analysis.

All systems and product are subjected to natural cycles of introduction, growth, maturity and decline [15]. If a component in the declining phase, this could lead to increased maintenance or technical debt. Just as with real debt, this technological debt needs to be paid off in some point of time. It is therefore useful to know if a component is continuously integrated in solutions.

Legal risk can be checked by the presence of licenses. Security risks can be assessed by the previously mentioned National Vulnerability Database.

4.2. Architecture and Prototype

UML architecture diagram Fig. 1 describes how the tool's components tie together and what the role of the user in the system is.

The user supplies input for the tool in the form of a link to a GitHub repository. (In the future this could be automated as a Google Chrome or Firefox extension which provides users that info when they browse for repositories on GitHub.com.) The prototype will than query basic information about a repository, such as name and author from GitHub and return that to the user. After retrieving this data, the tool retrieves data from all previously defined sources and uses this data to measure OS components on quality.

For long-running background operations or extra data, the tool will send that data in the form of JSON to the client. The client, running in the user's browser will parse and update the view accordingly. One relation missing in figure 1 is how the tool receives data from the Security Vulnerability Database. This however, is done through a local lookup in a database dump which is downloaded in advance in XML format. The advantage of this is that we are not relying on extra HTTP requests for API calls, however the database does need to be updated regularly.

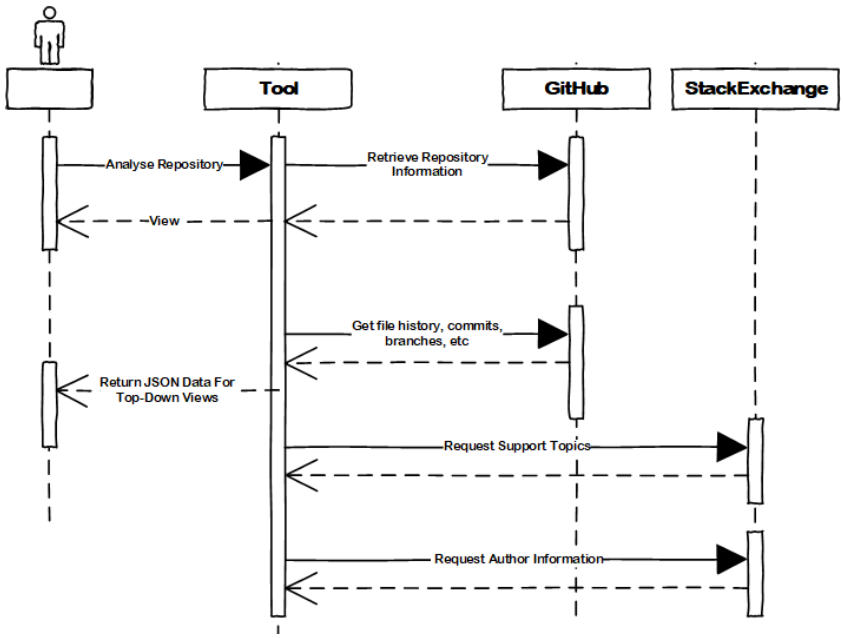

Fig. 1. UML architecture diagram of the tool

To speed up the development of this prototype, we selected relevant components which they were proficient in. These are the following:

1. Laravel Model View Controller. This PHP based controller provides structure out of the box. We used it to route user requests between various pages and organize logic in Model, View, Controller and service layers. https://GitHub.com/laravel/laravel.
2. Laravel GitHub. This extension allows a developer to consume the GitHub API. The tool uses non-authenticated request (request without oAuth authentication) in order to retrieve data such as commits and tags https://GitHub.com/GrahamCampbell/Laravel-GitHub.
3. VueJS. This library is used to render views in a reactive way. It integrates well with PHP and Laravel due to its ability to consume JSON. https://GitHub.com/vuejs/vue
4. XHTTP. A small frontend JavaScript component for asynchronous JavaScript and xml request. This library is used to fetch support data from Stackexchange.com and retrieve and renew data without refreshing the page. https://GitHub.com/Mitranim/xhttp.

In addition, the usual front-end tooling – based on NPM – was used. The non-exhaustive list of components used are WebPack, Babel and various Babel loaders. By leveraging these OSS components, other developers can easily extend our tool to include more or other criteria to assess software quality.

The final implementation of our tool is available via GitLab https://gitlab.com/Arevico/oss-quality-triangulator for anyone to use, review and contribute to. Because of space limitation, in this paper we only briefly describe the tool's functionality. It includes two quality modules: one for maturity evaluation and one for risk evaluation. The first uses as input (1) built scripts and commit messages to evaluate maintenance, (2) incident history to evaluate the extent of security, (3) information on issues and developers' documents to evaluate the level of support and the quality of documentation. The second module uses information about test cases and components operation, availability of various files, and search history of developers for the purpose of effort and risk estimation. Aggregating the evaluation output from both modules, the tool's user is presented with a report to support his/her decision-making process.

We further realized the tool in the following screens (See Figures 2-8).

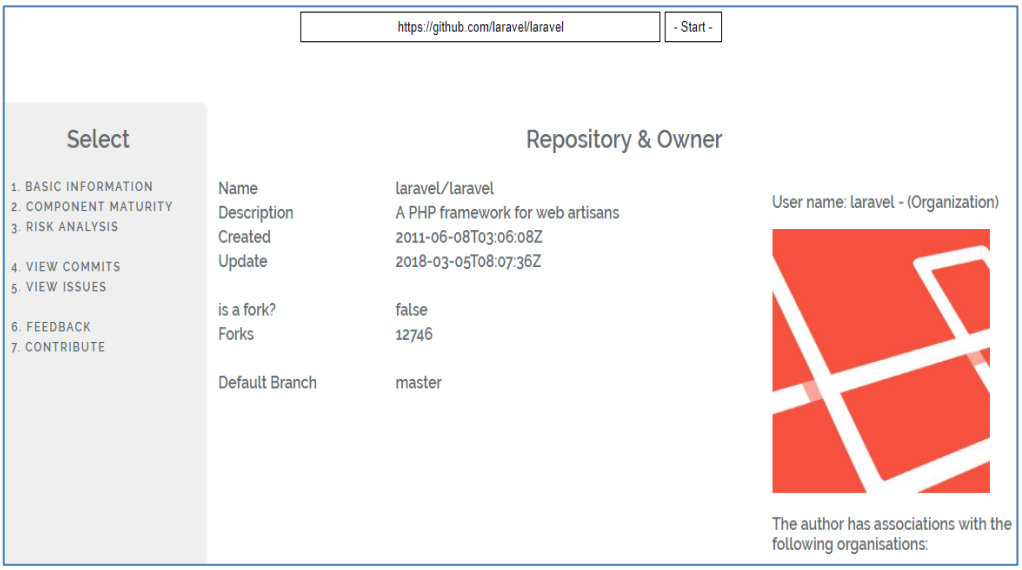

Fig. 2. The first triangulation screen of the https://github.com/Laravel/Laravel project

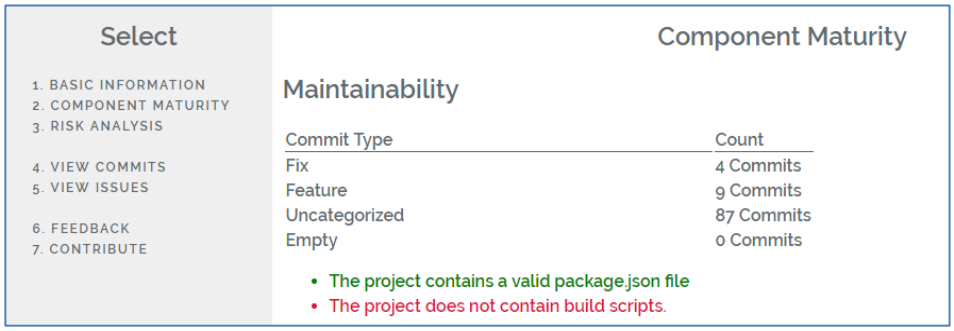

Fig. 3. The maintainability triangulation of the https://github.com/Laravel/Laravel project

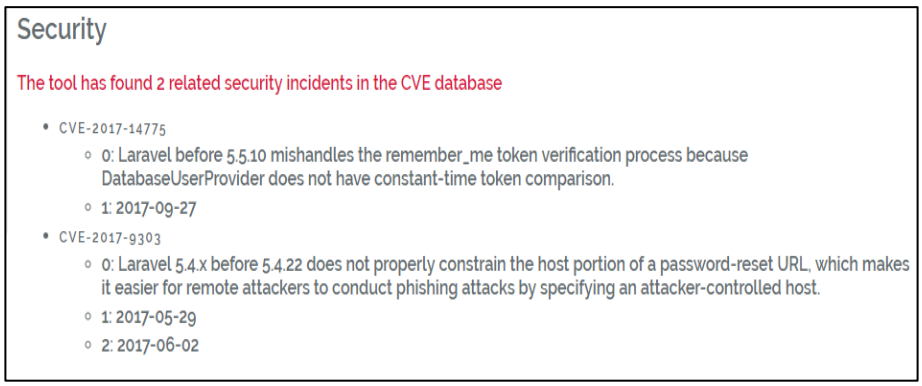

Fig. 4. The security triangulation of the https://github.com/Laravel/Laravel project

Fig. 5. The support triangulation of the https://github.com/Laravel/Laravel project

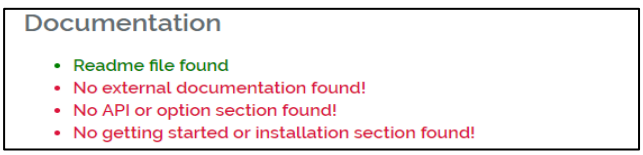

Fig. 6. The documentation triangulation of the https://github.com/Laravel/Laravel project

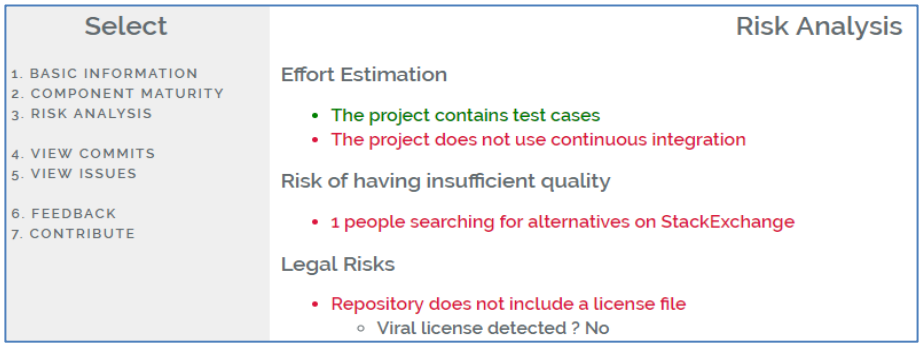

Fig. 7. The risk triangulation of the https://github.com/Laravel/Laravel project

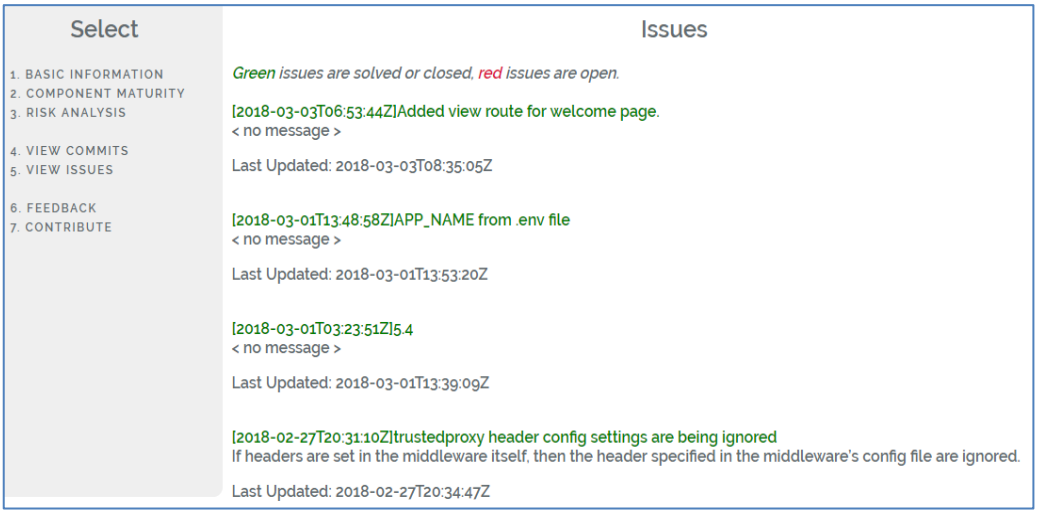

Fig. 8. The security triangulation of the https://github.com/Laravel/Laravel project

4.3. Design Propositions

The main hypothesis in our action research process is that we can accurately measure software quality by the input-to-output mapping provided in this section. This mapping will consolidate functional requirements with the inputs that an OSS component provides. The resulting output will be an evaluation of OSS. In addition, potential risks will be evaluated as well. The tool uses a variety of metrics and communicates those to the users to support their decision making about whether to use a specific open source component. To test if the tool really supports decision-making, the following hypothesis are established:

- P1: Users are able to interpret the quality metrics of an open source component.
- P2: Users gather new insight by using a tool which automatically reviews open source component quality.
- P3: Users change their perception and decision whether or not to use the component after viewing the metrics provided by the prototype.
- P4: The metrics used were found to be useful by the end user.

We note that the resulting output of the tool is not a final judgement for a developer on selecting OSS components. The tool will not decide for a user which component is the best in each situation. Rather, it is the goal of this tool to complement the knowledge that the user already has and draw his/her attention to potential issues, pitfalls and establish validity of claims made by the author. Where possible, the tool will link to other sources so that a decision maker or developer can get more information about a potential issue.

5. Evaluation Results

We carried out a first evaluation by using practitioners' feedback to understand if the tool meets its goals and if it does what it is supposed to do. Feedback about the tool has been collected through various channels and platforms, such as Stackoverflow, Reddit (Open Source and PHP

groups) and LinkedIn groups, and direct email. The respondents include freelance developers that have created relevant topics in the professional communities discussed before. The feedback was delivered via a feedback tab that we implemented in the prototype. Therein, we asked the following questions:

1. How useful is the information provided by this tool in a scale from 1 to 5?
2. Did the tool influence decisions about using this component or not (no or yes)?
3. What can we do to improve this tool / are there other things you want to share?

There were ten people leaving quality feedback via the feedback tab in the tool. In addition, three responses were received in response to direct mail or on open communities (Reddit, LinkedIn) instead of people using the feedback form.

Over the feedback period (21 days), 265 unique users accessed the tool (traffic log on request). The geographic distribution of users is mainly in the Netherlands, United States and the European Union. The users rated the usefulness of the information provided about the component on average at 3.1 on a scale of 5. Seven ratings of three, one rating of four and one rating of 3.5. Four people indicated that the tool influenced their decision about the tool, but they did not or could not provide details what the result of that change is. One person indicated that his/her decision about using the component did not change, but their overall view on the component became more positive

Users indicated that they would like to use the tool to compare various components, instead of just analyzing one. In addition, the tool should work in a recursive manner, looping over all dependencies a project might have analyzing their quality.

In addition, users provided as feedback that static code analysis is a good way to get more in-depth knowledge about the quality of a component. Users did not provide feedback about which metrics they are looking for in static code analysis.

6. Conclusions and Future Work

This paper addressed the difficulty of assessing quality of OSS components and the usefulness of an automatic tool to assist in this. Using a design science research method, we analyzed the practical problem of software quality evaluation in OSS, and proposed an artefact (the Triangulator tool) as the remedy to this problem.

As a conclusions and answer to our research question “What are the characteristics of an information triangulator that can improve a user’s understanding of the quality of an open source software component?”, we have given many requirements, a design and prototype of such a triangulator. The users in our evaluation additionally delivered multiple suggestions for improvement and further testing with a larger set of users is needed.

In the evaluation with OSS practitioners we found the following: First, the practitioners signaled that while the tool did provide good data collection, the judgement is left to the end-user. This is inherent to a triangulator and values that may differ per user. The design propositions are evaluated and we find the following.

The interest in a tool that automatically evaluates quality of open source components (P1) can be confirmed by the responses of users trying out the prototype. Users indicated that they learned new things from the tool and found the way the prototype processes data from multiple sources useful.

Some users were able to interpret the significance for their final product (P2), but most users would like the tool provide some interpretation as well. Linking to the theory behind the tool was not considered enough, so the implications of failing and passing certain metrics needs to be included in a future iteration of the tool.

Users change their perception after viewing the metrics provided by the prototype (P3). Some users suggested that this perception building would be easier of multiple outcomes could be in one screen.

The metrics used were found to be useful by the end user (P4). The security incident data was well perceived by the participants in our evaluation. However, the opinions about risk data were divided. One user indicated that he would like to have it compared to a benchmark. Furthermore, users provided as feedback that static code analysis is a good way to get more in-depth knowledge about the quality of a component. To incorporate this in the prototype, more research is needed on predictors for code quality in various programming languages, so that these results can become of the tools knowledge base.

This research has some implications for practice and research. First, it indicated a need and market demand to assess the quality for OSS quality automatically. Our Triangulator tool can

potentially help OSS practitioners run a quick quality evaluation that allows them to leverage OSS information that is dispersed across multiple repositories. Knowing the evaluation, an OSS company may decide for or against using a component. Second, our tool can provide a base for researchers and interested individuals to expand on with more quality and risk indicators. Specifically, we suggest two lines for future research. First, the implications of maturity on component quality should be further studied and communicated to the user. Second, more risk factors should be explored and evaluated and potential impact of those risks should be communicated to the end-user if the component is at risk.

References

1. Adewumi, A., Misra, S., Omoregbe, N.: A Review of Models for Evaluating Quality in Open Source Software. IERI Procedia. 4 88–92 (2013)
2. Aggarwal, K., Hindle, A., Stroulia, E.: Co-evolution of project documentation and popularity within github. In: Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014. pp. 360–363. (2014)
3. Alali, A., Kagdi, H., Maletic, J.I.: What's a typical commit? A characterization of open source software repositories. IEEE Int. Conf. Progr. Compr. (June 2014), 182–191 (2008)
4. Arazy, O., Kumar, N., Shapira, B.: A Theory-Driven Design Framework for Social Recommender Systems. J. Assoc. Inf. Syst. 11 (9), 455–490 (2010)
5. Borges, H., Hora, A., Valente, M.T.: Understanding the factors that impact the popularity of GitHub repositories. In: Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on. pp. 334–344. IEEE (2016)
6. de Bruin, T., Raybaud, P., Toulhoat, H.: A DFT Chemical Descriptor to Predict the Selectivity in α-Olefins in the Catalytic Metallacyclic Oligomerization Reaction of Ethylene According to the (Hemi)labile Ligand Coordinating to Titanium. (2008)
7. Choi, H., Varian, H.: Predicting the present with Google Trends. Econ. Rec. 88 (special issue SI), 2–9 (2012)
8. Cowan, C.: Software security for open-source systems. IEEE Secur. Priv. 99 (1), 38–45 (2003)
9. Goodin, D.: Kernel.org linux repository rooted in hack attack, *The Register*, http://www.theregister.co.uk/2011/08/31/linux_kernel_security_breach/, Accessed: December 10, 2018, (2011)
10. De Graaf, K.A., Liang, P., Tang, A., Van Vliet, H.: How organisation of architecture documentation affects architectural knowledge retrieval. Sci. Comput. Program. 121 75–99 (2016)
11. Jarczyk, O., Gruszka, B., Jaroszewicz, S., Bukowski, L., Wierzbicki, A.: GitHub Projects. Quality Analysis of Open-Source Software. In: Lecture Notes in Computer Science. pp. 80–94. Spinger (2014)
12. Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D.M., Damian, D.: The promises and perils of mining GitHub. In: Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014. pp. 92–101. (2014)
13. Lee, M.K.O., Cheung, C.M.K., Lim, K.H., Ling Sia, C.: Understanding customer knowledge sharing in web-based discussion boards. Internet Res. 16 (3), 289–303 (2006)
14. Lerner, J., Tirole, J.: The Scope of Open Source Licensing. J. Law, Econ. Organ. 21 (1), 20–56 (2005)
15. Michelle Grantham, L.: The validity of the product life cycle in the high-tech industry. Mark. Intell. Plan. 15 (1), 4–10 (1997)
16. Möller, K., Svahn, S.: Crossing East-West boundaries: Knowledge sharing in intercultural business networks. Ind. Mark. Manag. 33 (3), 219–228 (2004)
17. Pfeffers, K., Tuunanen, T., Rothenberger, M.A., Chatterjee, S.: A Design Science Research Methodology for Information Systems Research. J. Manag. Inf. Syst. 24 (3), 45–77 (2007)
18. Rosen, L.: Open source licensing: Software freedom and intellectual property law. Prentice Hall, Upper Saddle River, New Jersey (2004)
19. Sheoran, J., Blincoe, K., Kalliamvakou, E., Damian, D., Ell, J.: Understanding "watchers" on GitHub. In: Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014. pp. 336–339. ACM digital library, Hyderabad, India (2014)

20. Shihab, E., Bird, C., Zimmermann, T.: The effect of branching strategies on software quality. In: Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement - ESEM '12. p. 301. (2012)
21. Siena, A., Morandini, M., Susi, A.: Modelling Risks in Open Source Software Component Selection. In: Conceptual Modeling. pp. 335–348. Springer Nature Zwitserland (2014)
22. Tom, E., Aurum, A., Vidgen, R.: An exploration of technical debt. J. Syst. Softw. 86 (6), 1498–1516 (2013)
23. Walls, J.G., Widmeyer, G.R., El Sawy, O.A.: Building an Information System Design Theory for Vigilant EIS. Inf. Syst. Res. 3 (1), 36–59 (1992)
24. Wijnhoven, F., Brinkhuis, M.: Internet information triangulation: Design theory and prototype evaluation. J. Assoc. Inf. Sci. Technol. 66 (4), 684–701 (2015)
25. Zahoor, A., Mehboob, K., Natha, S.: Comparison of open source maturity models. Procedia Comput. Sci. 111 348–354 (2017)
26. Zazworka, N., Shaw, M. a., Shull, F., Seaman, C.: Investigating the Impact of Design Debt on Software Quality. Work. Manag. Tech. Debt. 17–23 (2011)