OpenHub: A Scalable Architecture for the Analysis of Software Quality Attributes

Gabriel Farah
Universidad de Los Andes
Cra 1E #19-40,
Bogotá - Colombia
57 1 339-4949
g.farah38@uniandes.edu.co

Juan Sebastian Tejada
Universidad de Los Andes
Cra 1E #19-40,
Bogotá - Colombia
57 1 339-4949
js.tejada157@uniandes.edu.co

Dario Correal
Universidad de Los Andes
Cra 1E #19-40,
Bogotá - Colombia
57 1 339-4949
dcorreal@uniandes.edu.co

ABSTRACT

There is currently a vast array of open source projects available on the web, and although they are searchable by name or description in the search engines, there is no way to search for projects by how well they perform on a given set of quality attributes (e.g. usability or maintainability). With OpenHub, we present a scalable and extensible architecture for the static and runtime analysis of open source repositories written in Python, presenting the architecture and pinpointing future possibilities with it.

Categories and Subject Descriptors
D.2.7 [**Software Engineering**]: Distribution, Maintenance and Enhancement.

General Terms
Algorithms, Measurement, Experimentation.

Keywords
GitHub, Python, Quality Attributes, Architecture

1. INTRODUCTION

Currently, the web hosts millions of freely accessible open source repositories across different services such as GitHub, BitBucket, Sourceforge, etc. Finding the right repository to use for a project amounts for a big part in the development pipeline. This includes; searching repositories by name or description, and then continuously integrating and testing projects until a fit meets a desired set of quality attributes (e.g. performance, testability or usability)[5]. This is a time consuming task since there is no proper tool to search for repositories by how well they perform on a set of given quality attributes.

OpenHub aims to be a system that easily provides access to information about millions of open source repositories and their performance on different quality attributes. It also aims to be easily extensible, allowing the analysis of different quality attributes and technologies/languages beyond the ones covered in this paper.

Prior to the release of this paper, we had more than 1.7 million projects crawled and more than 140,000 Python repositories analyzed for a total of 9.8TB of data processed using the OpenHub architecture. The resulting 3GB dataset collected is available for download. In this paper, we present the general design; go through the challenges and limitations of working with the dataset and outline research opportunities that emerge from it.

2. DATA COLLECTION

The crawler component was implemented to constantly crawl all public repositories available through GitHub, store or update (if they already existed) the repositories in a MongoDB data store, and eventually, push them to a queue for quality attribute analysis. Figure 1 presents the BSON document schema used to store the repository data, the instructions for loading them into MongoDB can be found in: goo.gl/fXqVWq. The type of JSON documents stored inside the MongoDB instance are exactly the same as provided by the GitHub API (http://developer.github.com/v3/repos/), with additional fields that are added for every quality attribute analyzed.

Figure 2 presents an overview of the process and the main components present in OpenHub.

The first challenge faced in the data collection process was that GitHub imposed an API limit of 5,000 requests per hour for authenticated requests. Given the always-growing number of repositories in GitHub, a single account would not suffice to go through the whole dataset quickly enough, for this reason, OpenHub was designed to perform queries in parallel using multiple accounts in different machines while maintaining a consistent state throughout all of them, and so avoiding the problem of over-querying.

In the website http://openhub.virtual.uniandes.edu.co there are two separate files available for download. One MongoDB dump with 140,000 Python repositories containing the GitHub API information plus the code metrics obtained using the OpenHub platform. Additionally, the second download available is the full 1.7 million GitHub repositories API information presented in MongoDB dump. This download includes the 140,000 python repositories mentioned.

MSR'14, May 31 – June 1, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2863-0/14/05...$15.00
http://dx.doi.org/10.1145/2597073.2597135

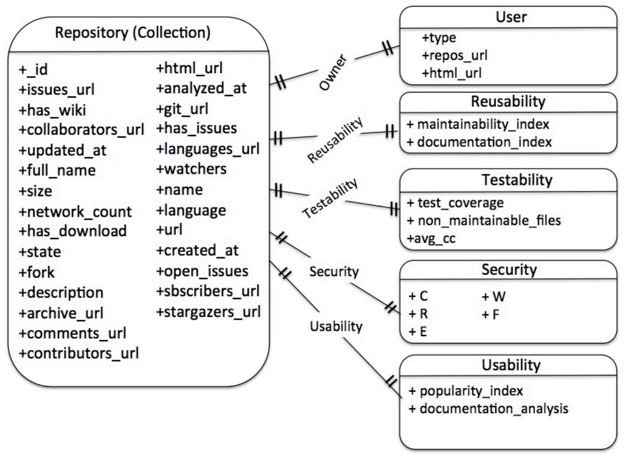

Figure 1. BSON Schema used by OpenHub

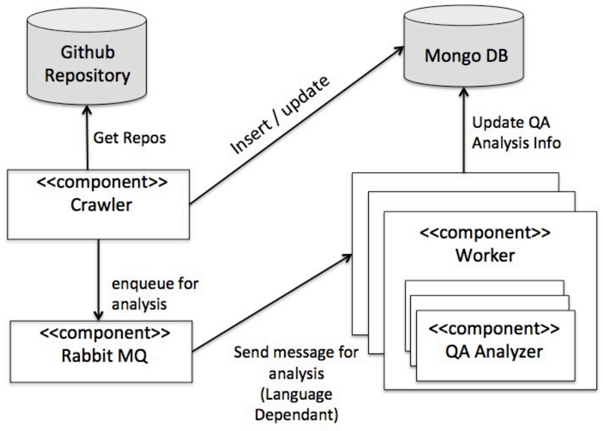

Figure 2. OpenHub main components

3. DATA ANALYSIS

The distributed messaging queue used was RabbitMQ, which allowed us to concurrently receive repository analysis messages from the crawler and assign them to any available QA Analysis worker for asynchronous analysis, as well as create different queues for repositories in different languages.

3.1 Repository Selection and Allocation

It was decided to analyze only quality attributes for repositories written in Python for the scope of this project, given the ease of development. However, this imposed some limitations in the implementation of performance and scalability analyzers given Python's dynamic typing, which ultimately resulted in the omission of such analyzers.

The deployed infrastructure contains three worker machines with a Worker component inside, whose function is to analyze a software repository. It contains a Manager module, which is constantly listening to repository analysis messages from the corresponding language queue to which it is configured. Upon reception of the message, the manager then downloads the code for the repository and passes it to any present quality attribute [1] analyzers inside the worker component for analysis, and finally updates the repository's document inside the database with the newly found analysis data returned from each quality attribute analyzer.

This process was designed to be highly extensible by allowing newly developed quality attribute analyzers for a language to be easily added to any Worker component; analyzers are simply Python modules that reside inside a package named after a quality attribute. The analyzer module must implement a *run_test* function to initiate the analysis and can return any kind of object, and as long as it resides inside any quality attribute package, the Worker Manager will automatically pick it up. Figure 3 presents the internal structure of OpenHub in terms of packages and software components.

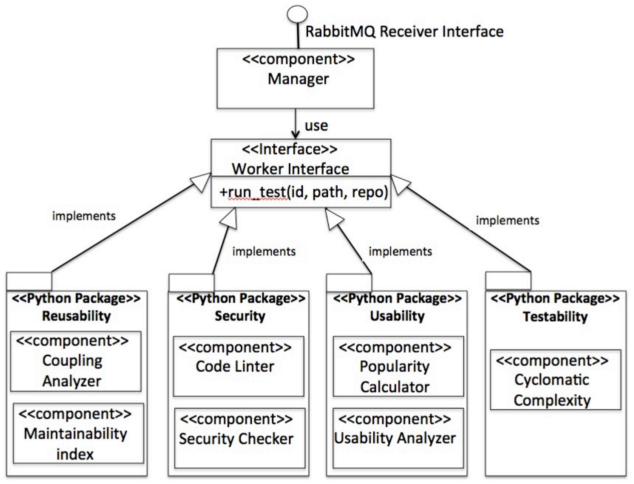
RabbitMQ Receiver Interface
<<component>> Manager
use
<<Interface>> Worker Interface
+run_test(id, path, repo)
implements
implements
implements
implements
<<Python Package>> Reusability
<<component>> Coupling Analyzer
<<component>> Maintainability index
<<Python Package>> Security
<<component>> Code Linter
<<component>> Security Checker
<<Python Package>> Usability
<<component>> Popularity Calculator
<<component>> Usability Analyzer
<<Python Package>> Testability
<<component>> Cyclomatic Complexity

Figure 3. OpenHub´s internal packages and modules

3.2 Analyzers Implementation

In OpenHub, workers contain a list of independent software packages to be called dynamically for the analysis of the data. In our implementation, there is a mix of analyzers; some created for OpenHub and some open source available in GitHub. Tables 1 presents the list of the quality attribute analyzers implemented for Python repositories used in this project.

Documentation Analyzer: a text based classifier using as input the ".md" (Markdown files) present in every GitHub repository was implemented. To train the model, a training data set of 1500 ".md" files was manually built and classified using the following attributes:

Class: "Good"

- Introduction to the repository
- Dependencies
- Examples/Getting started
- Pointer to further readings
- License
- (Above attributes in an orderly manner)

Class: "Average"

- Some of the attributes presented in the "Good" class

Class: "Bad"

- None of the attributes presented in the "Good" class
- Inexistent ".md" file in the repository

The implementation was conducted using the Python ML library sckit-learn.

Table 1. Quality attributes analyzers used in OpenHub

Quality Attribute	Analyzer	Description
Usability	Documentation Analyzer	Analyses the a repositories README file for the quality of its documentation and usage instructions
	Popularity Index Analyzer	Calculates the popularity on the web of a given repository.
Testability	Cyclomatic Complexity Analyzer	Calculates the cyclomatic complexity for a given repository.
	Code Linter Analyzer	Lints repository code and returns stats about how well the code is written and how vulnerable it is
	Basic security Analyzer	Checks basic security standards for repositories that are web applications or web frameworks.
Reusability	Maintainability Index Analyzer	Calculates the maintainability index for a given repository
	Coupling Analyzer	Calculates inner dependencies in the repository and how decoupled it is.

Coupling Analyzer: This analyzer calculates the dependency graph for any given *Python* repository considering dependencies only present within the repository, i.e. not external libraries. The dependency graph is a directed graph in which each outgoing edge represents a dependency from one Python module to another, considering dependencies as import statements between modules.

Given the dependency graph for any given repository, this analyzer determines the total number of independent components, where an independent component is defined as:

$$IC = \{n \in NODES \mid \forall (n,m) \in EDGES, m \in IC\}$$

That is, a set of nodes in the graph such that every outgoing edge of every node inside the set points to another node inside the set, i.e., there are no outgoing edges pointing to a node outside the set. Semantically, this represents a group of modules which only have dependencies between themselves and are independent from other modules in the repository, and as such could be used independently outside the project. The analyzer uses a simple linear time algorithm based on a depth-first traversal of the graph to find the number of independent components in the graph.

Given the number of independent components, this analyzer returns a coupling index between 0 and 1 that determines if the repository is highly coupled or not given the following formula:

$$1 - \left(\frac{\#\ of\ independent\ components}{\#\ of\ nodes}\right)$$

This index provides a measurement of coupling, where a highly coupled repository is represented by an index of 1, and a highly decoupled repository is represented by an index of 0.

Cyclomatic Complexity Analyzer: This analyzer calculates three custom attributes using as base the McCabe formulation for every repository static code. The underline implementation use the Radon Python library.

Maintainability Index Analyzer: In this module, two metrics are obtained; one is the Maintainability Index, which measures how maintainable (easy to support and change) the source code is. The maintainability index is calculated as a factored formula consisting of SLOC (Source Lines Of Code), Cyclomatic Complexity and Halstead volume.

Code Linter Analyzer: This analyzer uses the Python code linter *pylint* to do a static code analysis of the quality of the code. The analyzer counts and returns the number of *pylint* messages found of each type according to *pylint* docs.

The full code implemented for each analyzer, libraries used and documentation can be obtained at:

https://github.com/gabrielfarah/OpenHub/tree/master/worker

3.3 Results Consolidation

After a Worker finishes running all of the quality attribute analyzers for a given repository, it grabs all of the returned metrics and persists them in the database.

If any failures arise during the execution of any quality attribute analyzer, the Worker Manager will automatically skip the analyzer and mark the repository as pending for a full analysis in the database. If none of the analyzers can be executed, the Manager will mark the repository as failed in the database. Any repositories marked as failed or pending will be pushed to the queue for analysis when the Crawler re-encounters them.

4. CHALLENGES AND LIMITATIONS

We noticed that the time and resources it takes to process a repository is extremely hard to predict; none of the variables available before the processing phase (repository size, past processing time, etc.) can explain in a polynomic relation, the actual time or resources consumed. The inability to predict resources leads to bottlenecks in the architecture decreasing performance and oversaturating the queue. To address this issue, we implemented a time out for each of the individual analyzers inside a worker, if the timeout is used, the worker will interrupt the subprocess and will continue with the rest, notifying the manager about the issue and leave it in a consistent state for a future attempt.

The second restriction was the download process of the code itself in order to be analyzed. After several tests, performing a git clone over the repository was faster and less error prone than downloading the repository zipped code via http.

The final restriction corresponds to the quality analyzers that we were able to implement. Given the choice of Python as the programming language of the repositories, it was impossible to implement quality attribute analyzers for performance or scalability. The dynamically typed nature of the Python language prevents the arbitrary execution of its code since the types of parameters for functions are unknown. This limitation however, will not be present in languages like Java, where the types are known.

5. RESEARCH OPPORTUNITIES

OpenHub's architecture allows researchers to develop and test new quality attribute analyzers on the fly either for static or runtime analysis over the big array of open source repositories available. This approach, combined with the open presentation of the results via Web/ REST interfaces, will open the way for additional studies such as:

1. The average and specific differences in software quality attributes among programming languages.
2. The variability over time in quality attributes in software repositories while the project matures.
3. The relationship between the social formation of a open source community and the implemented quality attributes.
4. Quality attribute variability in same language implementation projects targeting different topics (Web, Mobile, Visualization, Math related applications, etc.).

We believe there are plenty of opportunities for researchers and developers to integrate more metrics for open source repositories into the current build. We encourage researchers, wanting to test a quality attribute analyzer to fork the GitHub repository and submit pull requests. Once approved and tested, the development team will download the new source and the results will be available for query instantaneously after over the Web/ REST interfaces.

For the submission of this paper, the dataset provide software metrics for the python language repositories. With the analyzed data, statistical studies can be performed paring the mentioned metrics with additional data to perform studies such as:

1. Augment the GitHub search engine with the addition of software quality metrics and make it available to the general public. If expanded to all of the languages supported in GitHub, these metrics can be a useful tool to compare similar repositories in terms of how good is the quality of the code presented.
2. Give researchers additional features over the GitHub API to add to their machine learning algorithms either for repository segmentation, repository classification among others.
3. Allow researchers, visualization of the change over time of the code metrics. Making it easier the discovery of insights explaining what variables affect this changes and how.

6. RELATED WORK

Our proposal uses a similar approach as the described in the GHTorrent Dataset and Tool suite paper [3]. More specifically, the use of a queuing system was an idea we used in OpenHub for the selection and distribution of the repositories. Similarly, RabbitMQ and MongoDB were used due to theirs easy deployment and integration. In the GHTorrent work, the architecture is used to pull out information about the repositories given the GitHub notification web service, we took it a step further by downloading and analyzing the static code of the repositories. Additionally, we implemented several static code analysis as proposed by the literature [3] [4], however our work differs in terms of the parallelization and error management we implemented in our queue architecture.

7. CONCLUSIONS

The implementation of OpenHub produced several positive results including the creation of a highly extensible system that allowed us the analysis of millions of open source GitHub repositories. As future work, we would like to expand into several more programming languages such as Java, C++, PHP and JavaScript, with this, OpenHub will cover the majority of github.com, allowing the comparison between similar components implemented in different languages.

Once OpenHub cover the majority of GitHub, we plan to index and analyze similar open source host such as Google Code, Bitbucket and SourceForge. On parallel, release the information collected to the general public, using a web interface, allowing software architects and programmers to query and review, the additional information collected for the open source repositories.

The source code for the project can be obtained at https://github.com/gabrielfarah/OpenHub. More information can be found at http://openhub.virtual.uniandes.edu.co.

8. REFERENCES

[1] Bass, L. and Clements, P. and Kazman, R.(2012). Software Architecture in Practice. Pearson Education. Isbn:9780321815736.

[2] Gousios, G. (2013). The GHTorrent Dataset Tool Suite. MSR '13: Proceedings of the 9th Working Conference on Mining Software Repositories

[3] Kannavara, R. (2012). Securing Opensource Code via Static Analysis. *ICST, page 429-436. IEEE.*

[4] Novak, J. (2010). Taxonomy of static code analysis tools. MIPRO, 2010 Proceedings of the 33rd International Convention. Pages:418-422. SIBN:978-1-4244-7763-0

[5] Tao Xie and Jian Pei. 2006. MAPO: mining API usages from open source repositories. In *Proceedings of the 2006 international workshop on Mining software repositories* (MSR '06). ACM, New York, NY, USA, 54-57. DOI=10.1145/1137983.1137997 http://doi.acm.org/10.1145/1137983.1137997