View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Electronic Communications of the EASST (European Association of Software Science and...

Electronic Communications of the EASST
Volume 65 (2014)

Proceedings of the
International Workshop on
Software Quality and Maintainability
(SQM 2014)

# Analyzing Gerrit Code Review Parameters with Bicho

Jesus M. Gonzalez-Barahona*, Daniel Izquierdo-Cortazar†, Gregorio Robles*, Alvaro del Castillo†

12 pages

Guest Editors: Lodewijk Bergmans, Tom Mens, Steven Raemaekers
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

# Analyzing Gerrit Code Review Parameters with Bicho

**Jesus M. Gonzalez-Barahona*, Daniel Izquierdo-Cortazar†, Gregorio Robles*, Alvaro del Castillo†**

* GSyC/LibreSoft, Escuela Tecnica Superior de Ingenieria de Telecomunicacion, Universidad Rey Juan Carlos (Madrid, Spain) † Bitergia (Madrid, Spain)

**Abstract:** Code review is becoming a common practice in large scale software development projects. In the case of free, open source software projects, many of them are selecting Gerrit as the system to support the code review process. Therefore, the analysis of the information produced by Gerrit allows for the detailed tracking of the code review process in those projects. In this paper, we present an approach to retrieve and analyze that information based on extending Bicho, a tool designed to retrieve information from issue tracking systems. The details of the retrieval process, the model used to map code review abstractions to issue tracking abstractions, and the structure of the retrieved information are described in detail. In addition, some results of using this approach in a real world scenario, the OpenStack Gerrit code review system, are presented.

**Keywords:** Code review; software maintenance; software development;

## 1 Introduction

Code review is an accepted practice to improve the quality of software [BB01]. It contributes to the quality of the code, by detecting defects before they enter the production code base [ML09, MBR13], and checks the adherence of proposed changes to the specific policies and general architecture of the project. But it also introduces extra work for experienced developers [HKY+13], and extra delays in time-to-deploy, which is an important metric for continuous deployment projects. Tracking the performance, timing, people involved and other parameters of code review becomes fundamental to detect bottlenecks, need of resources, or just to detect troublesome trends.

Many large, free, open source software (FLOSS) projects are quickly adopting peer code review as a part of their development policies [RGS08]. And many of them are using Gerrit[1] as the system to support the review process. In those projects, the Gerrit repository hosts all the information needed to track code review. Gerrit itself provides both a web interface and an API to select reviews according to different criteria, and interact with them.

The data handled by Gerrit can be massive. For example, for Havana, the latest stable release of OpenStack, more than 21,000 code review processes were started over a period of 6 months, about 115 per day. For each of them, all the changes in state until the review were accepted or declined are recorded, usually after several iterations between several reviewers and the author. This wealth of information can be used to track in detail the parameters that characterize the review process, and to understand how it is performing.

[1] http://code.google.com/p/gerrit/

Code review support systems such as Gerrit can be understood as specialized issue tracking management systems. Each code review process can be modeled as a ticket (issue), moving through different states as one or more code patchsets are proposed, reviewed, approved, rejected, resubmitted, etc. In the work described in this paper, we take advantage of this fact to extend a tool designed to retrieve information from issue tracking systems to also retrieve information from Gerrit.

The extended tool is Bicho, a part of the MetricsGrimoire toolset[2]. Bicho has a modular architecture, with several backends to retrieve information from the issue tracking systems most popular in FLOSS projects, and a common frontend that produces an SQL database with the retrieved data, in a format which is partially common for all of those systems. A Gerrit backend has been built, which produces the same format for code reviews, modeling them as tickets, thus reusing a large part of the code.

To test the usefulness of the approach, we have used this extended version of Bicho to analyze the Gerrit system of the OpenStack project, designing queries on the produced SQL data to obtain some of the the most relevant parameters of the OpenStack review process.

The structure of the rest of this paper is as follows. The next section describes the common workflow with Gerrit. Section 3 presents the Bicho tool and its database schema, which will be important to understand how information in code review systems can be mapped to it. After that, Section 4 shows the details of how code review can be modeled as an issue tracking system. Section 5 presents how this was done in the case of Gerrit, so that Bicho could be extended to support it. The paper ends with some examples of the use of the data retrieved from the OpenStack Gerrit repository in Section 6, and some conclusions and further work in Section 7.

## 2 Code review with Gerrit

Code review typically involves a developer submitting a proposed change to the source code of the project, and one or more developers reviewing that change. In the case of Gerrit, the change comes in the form of a "patchset", a set of patches to files in the source code repository. Reviewers may accept the patchset as such, or ask for a new, enhanced patchset addressing their comments, in an iterative process.

If the patch is accepted, it is later automatically merged with the development branch, becoming a part of the code base. If it is not accepted as such, usually a new patchset is required. If no new patchset is provided during a certain period, usually the code review is removed from the active queue, and is no longer considered. When a new patchset is submitted in the context of an existing code review, it is again reviewed with the same possible outcomes, until it is finally approved or the period for submitting a new patchset ends without submission.

In some projects, there is a special kind of reviewers, core reviewers, who have the exclusive right to accept changes to the code base. In this case, only when they signal their acceptance, the code review is finally accepted.

There are several variations of how Gerrit can be used for code review. In our case, we will focus on the practices of the OpenStack project, since it will be used in the examples provided later. In this case, the review process is divided into three steps: verification of the patchset, code

[2] http://metricsgrimoire.github.io

review and approval of the change.

The verification of the patchset is intended for automatic tools to check that it compiles, complies with coding style, etc. If the verification step fails, the code review gets a -1, meaning that the developer should submit a new patchset fixing the detected errors.

When the verification step is passed, the review step starts. During it, any developer can comment or propose improvements to the patchset. They can also vote on the patchset with +1 (should pass as such), 0 (some concerns, but not blocking acceptance), or -1 (a new patchset should be submitted). Core reviewers vote in a similar way, but using -2 or +2 for expressing their status. In OpenStack, a patchset has to be accepted by a core reviewer, thus getting a +2, to be considered for approval. Approval is the end of this step: a core developer marks the patchset as such, and it moves to the last step. At any moment, a new patchset can be submitted, effectively moving the process back to the verification step.

The third step starts when a core reviewer marks the patchset as approved. This triggers another check by automatic tools, similar to the verification step, to ensure the code is ready for merging in the code base. If this step fails, again a new patchset has to be submitted, and the process comes back to the verification step. Otherwise, it is finally merged in the main development branch, and the process ends.

## 3 The Bicho database schema

Bicho is a tool to retrieve information from issue tracking systems, and store it in an SQL database with a structure suitable for queries oriented to analyze its main parameters [RGIH11]. It currently supports Bugzilla, Jira, and the issue tracking systems of some forges (Launchpad, Allura, GitHub, and Redmine). It uses the API provided by these systems, or HTML scrapping from the web pages they provide, or a combination of both, to obtain information about all changes to all tickets. The code in charge of dealing with each specific issue tracking system is the Bicho backend, and in fact there is one backend per supported system. Once Bicho has completed its job, any analysis can be performed just by querying the database it produces.

Each of the supported issue tracking systems has a different model of what a ticket is, and how it is changed. However, all of them share a common model. To accommodate at the same time what is common to all of them, and what is different, Bicho has a set of core tables, used by all the backends, and extended tables, particular to each of them.

A simplified version of the database schema for the core tables shared by all the backends is shown in Figure 1, with the three main tables:

- **Issues:** For each ticket, time when it was opened, opener, summary, description, current status, priority and assigned developer.
- **Changes:** All modifications to fields in tickets (priority, description, resolution status, etc).
- **People:** List of persons that at some point participated in the ticketing process.

In addition, there are two main extended tables, particular for each backend:

- **Issues_Extension:** Modifications to fields that are not in the Issues table, but are relevant for the specific backend.

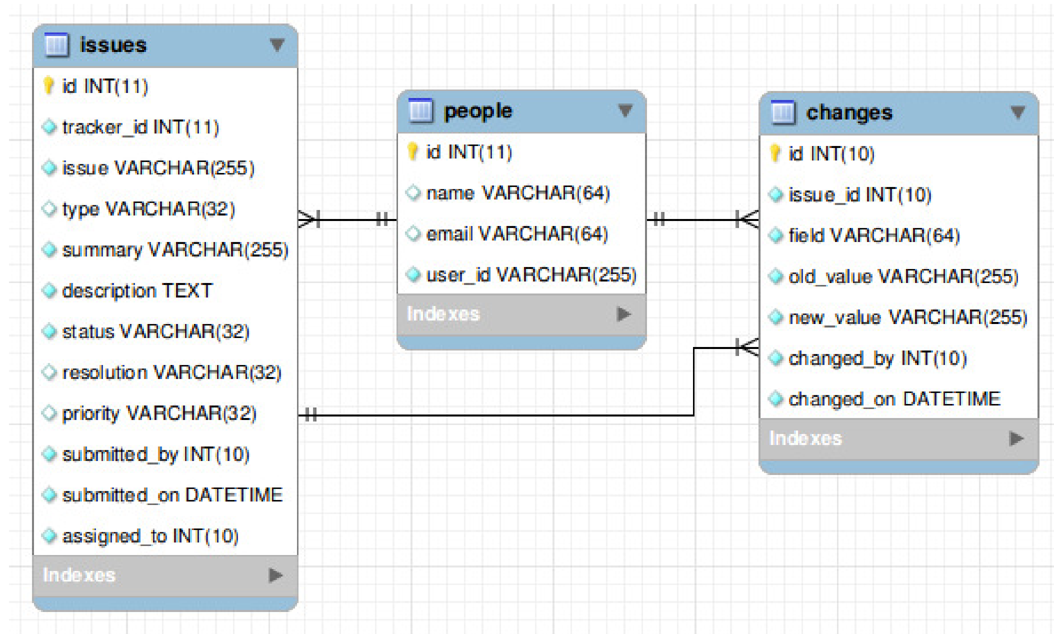


Figure 1: Simplified database schema of the Bicho tool

- **Issues_Log:** Historic status of each ticket over time. It is obtained once the retrieval process is finished, based on the contents of the Issues, Changes and Issues_Extension tables. This table facilitates the analysis of the history of the tickets, and the snapshots at points in time.

## 4 Modeling code review as changes to tickets

The information needed to follow a code review process can be modeled as changes to annotations in a ticket, similar to how tickets evolve in an issue tracking system. In fact, there are cases where projects use a real issue tracking system to assist the review process. One prominent example is the WebKit project which uses annotations in tickets in Bugzilla [GIMR13], or GitHub, where a code review can be implemented using the pull request functionality, which is supported by the issue tracking system with tickets tagged as pull requests.

In the case of Gerrit, modeling with tickets can be as follows:

- Each review code process (change) is modeled as a ticket, with its unique identifier being the change identifier.
- All information needed to track what happens during the code review process is modeled as changes to fields in the ticket.
- The relevant fields are:

  - Submitted (SUBM). Submission of a new patchset: patchset identifier. Increments by one the patchset identifier (first patchset is 1).
  - Verify (VRIF). Result of a verification step: +1 (passed) or -1 (not passed).
  - Review (CRVW). Vote during the code review step: an integer between -2 and +2.
  - Approve (APRV): +1 (approved for merging).
  - Abandoned (ABDN): +1 (marked as abandoned).

- For each change to a field, some extra data will be collected: the current patchset, the time of the change to the field (TIME), and who issued the change (SUBMITTED_BY).

These fields are very similar to those used by, for example, Bugzilla. Some exploration shows that they work the same way as in issue tracking systems: each time a change is produced to a ticket, the change is recorded along with information about who issued the change and when this was done. The only major difference is the existence of the patchset number, which can be considered as a property of a ticket.

Therefore, this information can be retrieved from a Gerrit system, and be stored in a very similar way to how it would be stored in a database for an issue tracking system. By querying that database, the main parameters of the code review process could be calculated. Some examples:

- For any review and patchset pair, the period from a change from SUBM (first submission of a patchset) to the first change to CRVW will be its time-to-attention (time until first review is obtained).

- The number of unique people changing CRVW during a certain period will be the number of code reviewers.

- The number of tickets opened before a certain date, but with no change to APRV or ABDN before that date, is the backlog of review processes still open at that moment.

## 5 Extending Bicho to support Gerrit

To show in practice how code review abstractions can be mapped to issue tracking system abstractions, a Bicho backend for Gerrit was designed and developed. Using the Gerrit API all the needed data is retrieved. Following the model described in the previous section, that data is fed into the Bicho database as follows:

- For each change (code review process), a new entry in the Issues table is opened, with the change as ticket identifier.

- For each change, patchsets are identified, and recorded in the Changes table using an identifier for each of them.

- For each patchset, the review process history is modeled as entries in the Changes table for the intended fields (SUBM, VRIF, CRVM, APRV, ABDN), each of them tagged with the corresponding patchset identifier.

| | Files | Lines of code |
|---|---|---|
| Bicho (included all backend) | 27 | 6,621 |
| Bicho backends (9 backends) | 12 | 4,104 |
| Gerrit backend (gerrit.py) | 1 | 231 |

Table 1: Number of files and lines of code (as analyzed by the CLOC tool) for Bicho, Bicho backends, and the Gerrit backend for it. All code is in Python.

- An Issues_Log table, built from the previous ones.
- People and other tables are built with data obtained for each event in the review process.

The abstraction which was more difficult to map to the Bicho model was the patchset. Unfortunately, we could not find an abstraction in an issue tracking system capable of capturing the idea of a patchset. Fortunately, an unused field (for the case of Gerrit) in the Changes table could be used to store this patchset identifier. In the future, probably this should be implemented as an extension to the Changes table, specific to the Gerrit backend.

The extension of Bicho to support Gerrit represented a relatively small quantity of work, as can be shown in Table 1. Thanks to the use of the Bicho frontend, and the already available common facilities provided by the program, code to retrieve data from Gerrit consists of a mere 231 lines of Python code.

Once the backend was complete, we could test it by running queries similar to those for issue tracking systems (such as time to reach certain states, or people involved in tickets/changes). The experience has shown that the queries are similar, and a lot of the expertise in mining databases with issue tracking information can be used to mine code reviews. In addition, queries specific to code review systems (such as number of patchsets per change, or time waiting for review) were devised, and tested. The next section shows some examples of all these analyses.

# 6 Examples of analyses of the information

To illustrate how the Bicho database with the Gerrit backend can be used to analyze code review practices, this section shows the details of some analyses: number of people asking for reviews and reviewing, and oldest reviews still open. They have been performed on the Gerrit data retrieved by Bicho from the OpenStack project as of November 21st 2013.

## 6.1 Number of people involved in reviews

The size of the community involved in the code review process, and how many of them act as change proposers and as reviewers are important parameters to determine how large the development community is, and what roles developers are assuming in the review process.

The number of people proposing code changes each month can be obtained from the Issues table, in which the `submitted_by` field is the identifier for the person proposing the change mapped to the ticket. The (slightly simplified) SQL code is:

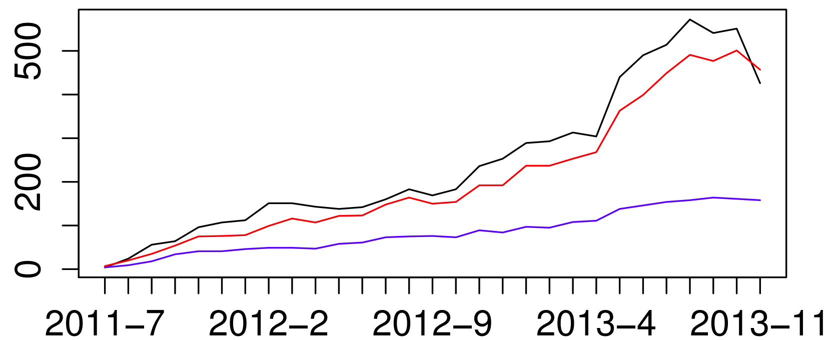


Figure 2: Number of change proposers (black), reviewers (red) and core reviewers (blue) per month in the OpenStack Gerrit system. Studied period: from July 2011 to November 2013.

```
SELECT YEAR(changed_on),
       MONTH(changed_on),
       COUNT(DISTINCT(submitted_by))
         AS submitters
FROM issues
GROUP BY YEAR(submitted_on),
         MONTH(submitted_on);
```

A similar query (also slightly simplified) can be used to get the number of people reviewing patchsets each month, using information from the Changes table. Now, changes to the CRVW field, which corresponds to votes during the review step, are tracked:

```
SELECT YEAR(changed_on),
       MONTH(changed_on),
       COUNT(DISTINCT(changed_by))
         AS reviewers
FROM changes
WHERE field='CRVW'
GROUP BY YEAR(changed_on),
         MONTH(changed_on);
```

Figure 2 plots the evolution of those two parameters over time, showing how the number of people involved in the review process is increasing quickly, but still keeping balanced. This is typical of a true peer review process where everyone ends up as change proposer and reviewer.

We have also plotted core reviewers, which in OpenStack are those who can finally accept or reject a patchset. This is a smaller group of people, with a different evolution. It can be seen how their number grew proportionally to reviewers during a large part of the history of the project, but during the last months it is no longer growing that way. This could cause an increasing workload on those most experienced developers, signaling that probably the project should find a way of increasing the number of these type of developers.

### 6.2 Top oldest reviews in Gerrit still open

To detect problems and bottlenecks, detecting which changes are staying longer in the review process is of interest. For getting this information, data in the Issues and Changes tables can be crossed to find the oldest changes among those still not approved or abandoned. We therefore can use the following query:

```
SELECT issues.issue AS review,
       issues.summary AS summary,
       TIMESTAMPDIFF (HOUR, times.min_time,
              times.max_time) AS opened
FROM (SUBQUERY) times,
     issues
WHERE times.issue_id = issues.id
ORDER BY opened DESC
LIMIT 10;
```

With SUBQUERY being:

```
SELECT changes.issue_id AS issue_id,
       MIN(changes.changed_on) AS min_time,
       MAX(changes.changed_on) AS max_time
FROM changes, issues
WHERE changes.issue_id = issues.id AND
      (issues.status='NEW' OR
       issues.status='WORKINPROGRESS')
GROUP BY changes.issue_id
```

For example, on November 21st 2013 the oldest change still open has the identifier 25,882[3]. Table 2 shows the ten oldest change proposals that are still open, and for how long the have been open.

### 6.3 Time to close a review

The time to close a change is important from a management point of view. This time is defined as the period since a change is opened in Gerrit, to the moment one of the patchsets proposed

[3] https://review.openstack.org/#/c/25882/

| Change | Summary | Days open |
|---|---|---|
| 25882 | Add listing tested APIs | 225 |
| 31068 | Sync common db and db.sqlalchemy code from Oslo | 173 |
| 33236 | Run DB API tests on a given DB backend | 151 |
| 33473 | Add SSL certificate verification by default | 148 |
| 30755 | The use of the class variables | 148 |
| 34291 | API extension to list supported scheduler hints | 146 |
| 34519 | Add API schema for v3 keypairs | 145 |
| 36207 | Use common Oslo database session | 134 |
| 36291 | added Neutron incompatibility note for simple IP management | 133 |
| 36197 | added tab showing all servers assigned to a hypervisor | 132 |

Table 2: Top 10 oldest changes still open on November 21st 2013 in the OpenStack Gerrit system. Change column shows the Gerrit identifier for the change, time open is measured in complete days.

in it is finally approved and merged in the development branch, and the change is considered to be done. The study of the evolution of this parameter helps to understand what can be expected when submitting a new change proposal, how reactive the project is to these proposals, and how long it is taking to review the code, once it has been written and proposed as a new change.

In this case example, we have considered separately those changes that get merged into the code base, and those that are abandoned. The query for obtaining the times for merged changes is shown below. The one for abandoned changes is almost the same, with ABANDONED instead of MERGED in the subquery.

```
SELECT TIMESTAMPDIFF (HOUR, times.min_time, times.max_time) AS opened
FROM (SUBQUERY) times,
     issues
WHERE times.issue_id = issues.id
```

With SUBQUERY being:

```
SELECT changes.issue_id as issue_id,
       MIN(changes.changed_on) AS min_time,
       MAX(changes.changed_on) AS max_time
FROM changes, issues
WHERE changes.issue_id = issues.id AND
      issues.status = 'MERGED'
GROUP BY changes.issue_id
```

To understand the query, it is important to notice that the Issues table keeps the status field, which is initialized as NEW, and can later be WORKINPROGRESS (both stated imply that the

| Type | Min | 1st Qu. | Median | Mean | 3rd Qu. | Max |
|---|---|---|---|---|---|---|
| Merged | 0 | 0 | 17 | 97.13 | 95 | 7,161 |
| Abandoned | 0 | 0 | 11 | 135.7 | 116.2 | 7,322 |

Table 3: Basic statistics of the distribution of the time to close (merge or abandon) changes. Time is in integer hours (but means or medians can be fractional because divisions are implied). Mean is not very representative due to the skewness of the distribution.

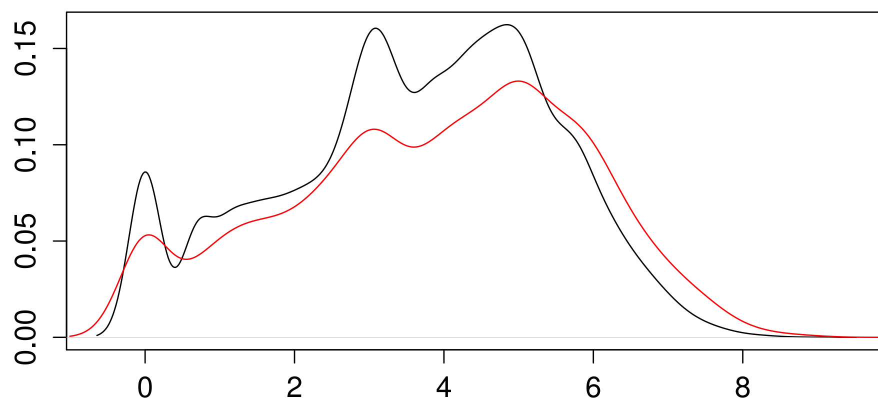


Figure 3: Distribution of time to close for merged (black) and abandoned (red) changes. Time, in X axis, is in natural logarithm scale (unit is integer hours).

review process did not finish yet), MERGED (was merged into the code base) or ABANDONED (was marked as such).

As shown in Table 3, 25% of all the finally accepted changes were merged in less than 1 hour, 50% in less than 16 hours, and 75% in less than 95 hours (roughly 4 days). This also means, of course, that 25% of the changes that were approved took more than 4 days to merge. It is important to notice that in many of these cases several patchsets were submitted and reviewed before the change was finally approved (and the last of these patchsets was merged). And that this process takes place sequentially: for a given review if a new patchset is uploaded, the previous one is automatically deprecated. The numbers for abandoned changes are similar: a bit shorter for the quickest changes to be abandoned, a bit larger for those taking more time.

Figure 3 shows the distribution of time to close both types of changes. It can be graphically shown how most of the times are between $e^2$ (approximately 7 hours) and $e^6$ (about 400 hours, or 17 days). The difference between the distribution of abandoned and merged changes can also be better understood by viewing it.

## 7 Conclusions and further work

In this paper we have shown how systems supporting code review processes can be modeled as a specialized kind of issue tracking systems. This theoretical model has been used to extend the functionality of Bicho, a tool to retrieve information about tickers from issue tracking systems, to retrieve information about code review processes from Gerrit, one of the most popular tools to assist in code review in FLOSS projects.

Extending an existing tool instead of writing a new one for retrieving information from Gerrit has proved to be a very efficient approach, as it could be implemented with not much more than 200 lines of Python code. The result is a completely functional Gerrit data retriever, that offers information organized in an SQL database ready to be queried to analyze any relevant parameter.

The paper also shows how this database can in fact be mined to obtain such parameters in a real world scenario; in this paper we have shown it with the OpenStack project, with several tens of thousands of changes proposed and reviewed. This has allowed us to comment on the characteristics and the quality of several aspects of the review process. These aspects included both extensive properties, such as the time to close changes (review processes) or size of the communities involved in code review, and intensive properties, such as the oldest reviews at a certain point in time.

In the future, authors intend to rearchitect Bicho so that it becomes a tool for retrieving information both from issue tracking and code review systems. The work presented in this paper shows that its architecture is already close to that goal, but we intend to redesign it so that Gerrit and other code review systems can be better integrated as first-class citizens in the context of Bicho structure.

## Reproduceability

Bicho can be retrieved from the MetricsGrimoire project at GitHub[4]. The exact version used for the work presented in this paper, along with the MySQL dump of the database retrieved by Bicho from the OpenStack Gerrit repository and the R script used to produce the graphics and the data in this paper can be retrieved from the reproduceability package for this paper[5].

## Acknowledgments

The work of Jesus M. Gonzalez-Barahona and Gregorio Robles in the study presented in this paper has been funded in part the Spanish Government under project SobreSale (TIN2011-28110). The work of Daniel Izquierdo-Cortazar has been funded in part by the Spanish Government under the Torres Quevedo program (PTQ-12-05577).

[4] http://metricsgrimoire.github.io
[5] http://gsyc.es/~jgb/repro/2014-sqm-bicho-gerrit

# Bibliography

[BB01] B. Boehm, V. R. Basili. Software Defect Reduction Top 10 List. *Computer* 34(1):135–137, Jan. 2001.
doi:10.1109/2.962984
http://dx.doi.org/10.1109/2.962984

[GIMR13] J. M. Gonzalez-Barahona, D. Izquierdo-Cortazar, S. Maffulli, G. Robles. Understanding How Companies Interact with Free Software Communities. *IEEE Software* 30(5):38–45, 2013.

[HKY+13] K. Hamasaki, R. G. Kula, N. Yoshida, A. E. C. Cruz, K. Fujiwara, H. Iida. Who Does What During a Code Review? Datasets of OSS Peer Review Repositories. In *Proceedings of the 10th Working Conference on Mining Software Repositories*. MSR '13, pp. 49–52. IEEE Press, Piscataway, NJ, USA, 2013.
http://dl.acm.org/citation.cfm?id=2487085.2487096

[MBR13] M. Mukadam, C. Bird, P. C. Rigby. Gerrit Software Code Review Data from Android. In *Proceedings of the 10th Working Conference on Mining Software Repositories*. MSR '13, pp. 45–48. IEEE Press, Piscataway, NJ, USA, 2013.
http://dl.acm.org/citation.cfm?id=2487085.2487095

[ML09] M. Mantyla, C. Lassenius. What Types of Defects Are Really Discovered in Code Reviews? *Software Engineering, IEEE Transactions on* 35(3):430–448, 2009.
doi:10.1109/TSE.2008.71

[RGIH11] G. Robles, J. M. Gonzalez-Barahona, D. Izquierdo-Cortazar, I. Herraiz. *Tools and Datasets for Mining Libre Software Repositories*. Volume 1, chapter 2, p. 24–42. IGI Global, Hershey, PA, 2011.
doi:10.4018/978-1-60960-513-1
http://www.igi-global.com/book/multi-disciplinary-advancement-open-source/46171

[RGS08] P. C. Rigby, D. M. German, M.-A. Storey. Open source software peer review practices: a case study of the apache server. In *Proceedings of the 30th international conference on Software engineering*. Pp. 541–550. 2008.