Evaluating Free / Open Source Software Projects

Jesus M. Gonzalez-Barahona

Published with GitBook

Table of Contents

evaluating-foss-projects

Evaluating Free / Open Source Software Projects (Book)

Introduction

Free / open source software projects may be very different from each other. Evaluating them either to compare them, or just to understand better how to deal with them is currently a kind of black art. This book intends to compile some information that helps towards providing background, methodologies and procedures for converting it into predictable engineering. Time will say if we succeeded in our effort.

The book is structured in three main areas. The first one, consisting of the chapters "Before evaluating", and "Sources of information" provides the basic background for understanding what is the evaluation of free, open source software (FOSS) projects, how it can be done, and what can be expected from it. The second one deals with the specifics of the different aspects of evaluation of FOSS projects, and is composed by three chapters: "Evaluating the community", "Evaluating development processes", and "Evaluation models". The third are is the practical one, presenting some tools for assisting in the evaluation, and includes three chapters too: "Evaluation dashboards", "Quantitative evaluation with Grimoire", and "Using Grimoire". The last two describe the Grimoire technology as an example of how tools can assist in evaluation, specially on quantitative evaluation, of FOSS development projects.

This book is a collaborative effort, in which the original authors like to see us as the facilitators of a large endeavor: the evolution, improvement and maintenance of the book over time. This can be done only with the collaboration of readers, experts in the fields, and in general anyone interested in the book. To make it easier to contribute, we decided to explore the GitBook technology, which is a FOSS project itself.

Therefore, this book is available from GitBook, which mirrors it from a GitHub repository. Different versions of the book (for reading online, for downloading in several formats) can be obtained from GitBook, while the "source code" (based on Markdown files) can be obtained in the GitHub repository. Contributions are accepted in several ways, the most convenient being using GitHub pull requests to that repository. You are encouraged to fork that repository and submit your pull requests suggesting improvements, if you want to collaborate. You can also use GitHub issues in the same repository to report any problem or make any suggestion.

We, the authors, hope that you find this book useful, and that it contributes to make your life easier and happier.

Before evaluating

In this chapter we deal with some preliminary issues, before describing how the evaluation of specific aspects of FOSS products and projects can be done.

Steps in evaluation

Most evaluation methods follow the following steps:

- Conduct a market research, to decide the subjects (products or projects) to evaluate.
- Define the evaluation criteria.
- Perform the actual evaluation, producing the evaluation results.

The first two can be preformed in any order, and both lead to the actual evaluation. In some cases, the process is iterative:

- some subjects are selected, and evaluated according to certain criteria
- based on the results, the criteria are refined, and the list of subjects redefined
- a new evaluation is performed, which leads to more precise criteria and subjects, until enough data for a final decision is obtained

Market research

Before starting the actual evaluation, the set of subjects to evaluate must be defined. For that, an extensive research for potential subjects is needed. But this research is not only a matter of finding all options. It would be desiderable to find all the products or projects that could have the slightest possibility of being the most appropriate ones, but that is usually impractical. A more cost-effective approach is to perform an extensive market research, but then follow some criteria to produce a short list for evaluation.

Depending on the resources available for the evaluation, on how expensive that evaluation will be per subject, and on the expected benefits, the size of the this list can be shorter or longer.

In some cases the subjects are already decided beforehand, and this step is skipped. This happens, for example, when a community decides to evaluate itself, or when a company wants to evaluate the products on which they relay for a service they provide.

Defining evaluation criteria

The evaluation criteria will determine the information to be obtained after the evaluation process. It is important to fit those criteria to the objectives of the evaluation. It is quite different, for example, to evaluate some products with the aim of selecting the most mature one, than the one with most perspectives of being improved for a long period.

Therefore, the evaluation criteria are tightly tied to the objectives of the evaluation, and it is necessary to make this relationship explicit. One way is to start by clearly stating the objectives, then map them to specific characteristics of the subject to evaluate, and finally define a procedure to evaluate those characteristics.

The mapping of objectives to characteristics is not easy, since it is not always obvious which characteristics will have a greater impact on the objectives. For example, if the objective is to select a project with a community easy to join, which characteristics would be evaluated? Maybe how long does it to newcomers to reach the core development team? Maybe how many contributions were performed by newcomers in the past? Maybe the details of formal policies encouraging the participation of newcomers? Or maybe the friendliness of conversations in mailing lists and forums in which newcomers participate?

It is clear how for making this mapping of objectives to characteristics, we needed a model of how the latter lead to the former. Those can range from informal, simple, rule of thumb models, to formal, theory-backed, and empirically-tested ones. In any case, we need some expertise on how characteristics of projects and products may influence the objectives.

The definition of procedures to evaluate the characteristics can also be tricky. You need to know what can be evaluated, and how you can apply that knowledge to the characteristics of interest.

It is very rare that you just find the exact evaluation for your characteristic of choice. For example, how could you characterize expertise in a development community? At least two dimensions are involved: the individual expertise of individuals, and how that expertise permeates the community to be useful to newcomers. But both dimensions are difficult to evaluate, and you usually need to relay on proxies, such as how long have developers stayed in the community, and how experienced people collaborate with newcomers to solve issues and take decisions.

Because of all these reasons, it is a hard job to start from scratch when evaluating. It is much better if we can find an evaluation model that fits our needs, and we just map it to our specific objectives. A large part of this text will describe some existing models that you could find useful.

In addition, it is convenient to explain how these models can be produced in systematic way. For that we will introduce the goal-question-metric (GQM) method in a later section. It will serve both as an illustration of how models can be produced, and as a tool to produce new models, if you need that. But you can use GQM not only to derive your own models, if you have the expertise and resources for that, but also to adapt existing ones to your specific environment.

Performing evaluation

Once the characteristics to evaluate are clear, and the methods to evaluate them too, you can start the actual evaluation. Depending on how you defined the evaluation methods, several actions may be performed, such as:

- Surveys, for example to know about perceived quality by users, or about effort estimation of their own work by developers.
- Interviews with experts, for example to know about how mature a certain product is considered.
- Study of documentation, to learn about how a product interoperates with others, and to which extent that is described.
- Analysis of source code, to characterize some quality parameters.
- Analysis of messages in mailing lists, to characterize the flow of information in the project.
- Study of the project bylaws, to determine how formal decisions are taken.

These are just examples: many other actions are possible. But whatever the case, with the information gathered from those actions, the evaluation is performed.

Evaluation results

Depending on the objectives of the evaluation, one of three kinds of resulting information are provided in most evaluation models:

- Tags. These are binary valued parameters that result of evaluation. For example, for a certain definition of "mature", a project may be defined as "mature" or "not mature".
- Scales. Parameters with values that are numbers or elements from a finite set. For example, an scale can be defined as from 0 to 100, trying to show how close to "100%" is the value. For example, a parameter "closed-bugs" could be "78%", meaning that of all bugs reported during a certain period, 78% were closed. Or with a real or integer number. For example, "median of time to close" can be defined to be "178", meaning that the median of time to close a certain set of tickets is 178 hours. Or with values in a set of strings. For example, "maturity" could have values in the set "mature", "close to maturity", "inmature".
- Metadata. This is usually detailed information about the parameter, from which usually either scales or tags can be produced. For example, detailed metadata for a parameter could be a list of its main statistics, or even a complete list of all its values. That way, from detailed metadata on "time to close" consisting on the time to close all tickets, the

above mentioned scale "median of time to close" could be inferred.

In addition, free text evaluation can be useful as well, such as the detailed analysis by an expert.

A very specific case of free text evaluation are factoids. Factoids are predefined pieces of text that describe with natural language some quantitative situation. They are selected based on the results of the quantitative evaluation, but have the appearance of free text. To some extent, they can be better for novices, since they provide an explanation in "common words" of the tags, numbers or scales.

Factoids shown by OpenHub for the git project, circa June 2015

We can classify the results of the evaluation process in two categories:

- Quantitative evaluations. They produce a quantitative description of the evaluated characteristic. Tags, scales and metadata are cases of quantitative evaluation.
- Qualitative evaluations. They produce a description of the quality of the evaluated characteristic. Free text evaluations produced by an expert are an example of qualitative evaluations.

Qualitative evaluations can be converted into quantitative ones by using the descriptions to select from a scale of valuers. That allows for easier comparison, but usually some information is lost, the kind of shadows and details that qualitative descriptions provide.

Quantitative evaluations can be converted in qualitative by producing the alreaready mentioned, predetermined "factoids". That allows for easier interpretation, but it is convenient to remember that those are "syntetized" qualities: the underlying information is quantitative, and the derived factoids are just descriptions of those quantities.

NOTE: TODO. Example of both conversions

Goal-question-metric

The baseline rationale of the goal-question-matric GQM metric for deteriming the characteristics of subjects to evaluate, and how to evaluate them can be summarized as follows:

> "The Goal Question Metric (GQM) approach is based upon the assumption that for an organization to measure in a purposeful way it must first specify the goals for itself and its projects, then it must trace those goals to the data that are intended to define those goals operationally, and finally provide a framework for interpreting the data with respect to the stated goals."
>
> The Goal Question Metric Approach.

In other words, you first have to state the goals of the evaluation. Then, you have to map those goals to characteristics of the subject of evaluation, and how they are going to be evaluated. Finally, you have to find out how to interpret the results of the evaluation with respect to the intended objectives.

Using the terms proposed by GQM, you have to define:

- Goals, at the conceptual level.
- Questions, at the operational level.
- Metrics, at the quantitative level.

Definition of goals

Definition of questions

Definition of metrics

What is different in FOSS evaluation

Evaluation of FOSS products is different for the following reasons:

- The easy access to the product to evaluate.
- The quantity and quality of available information
- The importance of the community
- The special case of open devlopment
- The competing market for deep support

Access to the product

In the case of non-FOSS, the first barrier to evaluate is the access to the product. For FOSS, the evaluator is usually one download away from evaluating any FOSS product which is adequately packaged. For non-FOSS, just accessing may mean signing a contract, paying for a regular non-exclusive license, or obtaining a usually limited evaluation version.

This means that with FOSS, evaluating the real thing promplty, to any detail, withouth strings of any kind attached is much more simple.

Available information

For FOSS products, not only the exectuable version of the software is availalble. Per definition, source code is available as well, which allows for its inspection, and the evaluation of aspects of quality that need access to it, such as code quality.

For some non-FOSS, source code may be available, either for all potential users or for those with a certain negotiation power. But it is a rare event.

In addition, if the development model is open, the development information for the FOSS product is kept avaible to anyone with the devloping community. Even when a single company drives the development of a FOSS product, they may decide to run all development in the open. When it is produced by a community, the rule is that the development information is available.

Therefore, the evaluation by third parties of the development processes is possible in the case of FOSS using open development models.

TBD: repositories where the information about development is available.

Community

Development and user communities are usually key factors for FOSS products. Healthy development communities ensure the future survivability of the product even better than strong companies. Large, involved user communites ensure the needed pressure to keep the product in the leading edge.

Therefore, the evaluatilon of communities is of great importance in the case of FOSS.

Open development

Some FOSS projects are developed "behind the curtains", not different from traditional projects. For those, little or no infornmation about the development process is available. But fortunately, these projects are the exception in the FOSS world. The usual case is that an open development model is used.

A simple definition of open development is:

> Open development is an emerging term used to describe the community-led development model found within many successful free and open source software projects.
>
> Avoiding abandon-ware: getting to grips with the open development method, by Paul Anderson

That kind of development, because of its very nature, usually provides publicly a lot of information about the internals of their development processes. That allows for an evaluation of those processes, something that is impossible in the traditional, closed development cases.

Competing market

The existence of a competing market, with many providers of in-depth support, independent of each other, is possible in the case of FOSS. In the case of non-FOSS, due to the strict control granted by maintaining all copyright rights, only companies in agreement with the producer can provide this kind of services, and therefore no real competing market exists.

In the case of FOSS, that market can exist. But it does not always exists. In fact, for many FOSS products no specific provider of in-depth support can be found. This is the case for most volunteer-driven and in-house projects, when an organization develops the software for its interenal needs. In both cases, until there is enough commercial use of the software, there is no demand for commercial support. Unfortunately, this means that new commercial actors interested in using the software will have more difficulty in doing so, for the very reason they cannot find support.

TBD: Generic support companies can be useful here. Detail the case of a single-provider.

TBD: importance of evaluating if such a competing market existis or not, and how it is.

The importance of transparency

Free, open source software communities are a matter of trust. All participants want to feel that the rules of the community are fair, and that everyone is considered on the value of their contributions, with no bias due other factors. For that, it is very important that the information about what happens in the community is transparent, and available to anyone. This is one of the main reasons for adopting open development models in FOSS communities.

But having the data available is not enough. Specially in large projects, the community needs some means to understand what is happening. The quantity of data may be really large, and it is not easy to extract from it useful information. Therefore, transparency is not only providing the data, but providing the data in a way that it is useful for the community, in a way that helps it to understand what's happening at several levels of detail.

The many facets of transparency

Criteria for evaluation

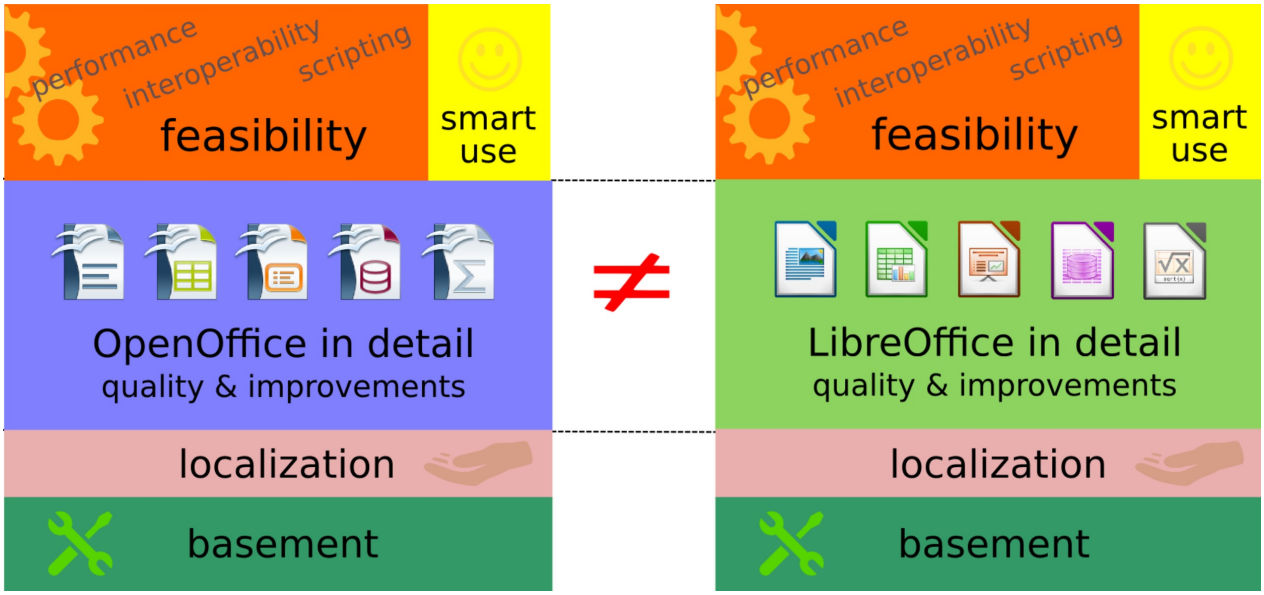
- Intangible factors
- Risk
- Functionality
- ...

Evaluation of functionality

This is one of the more commons evaluations when selecting tools, either to use or to integrate with others. Usually, this is done in the context of a product acquisition procedure, and considers mainly compliance with requirements, quality, and adaption to certain needs. The evaluation can be used to balance against cost, or to select among products that could fit the requirements.

Most of this evaluation is not different for FOSS and non-FOSS programas. Only the easy of access to the elements to evaluate make a difference. In the case of FOSS, source and binary code for the program are easily available for evaluation. Source code may be convenient to understand how a certain feature works, or to better evaluate performance. In some specific cases, such as evaluating security features, the availability of source code allows for deep inspection. But usually we can just use the general functional evaluation models. Therefore, instead of entering into details we will just sketch how functional evaluation can be done, illustrating with an example. This example is Comparing LibreOffice with Apache OpenOffice, a comparative functional evaluation of both products.

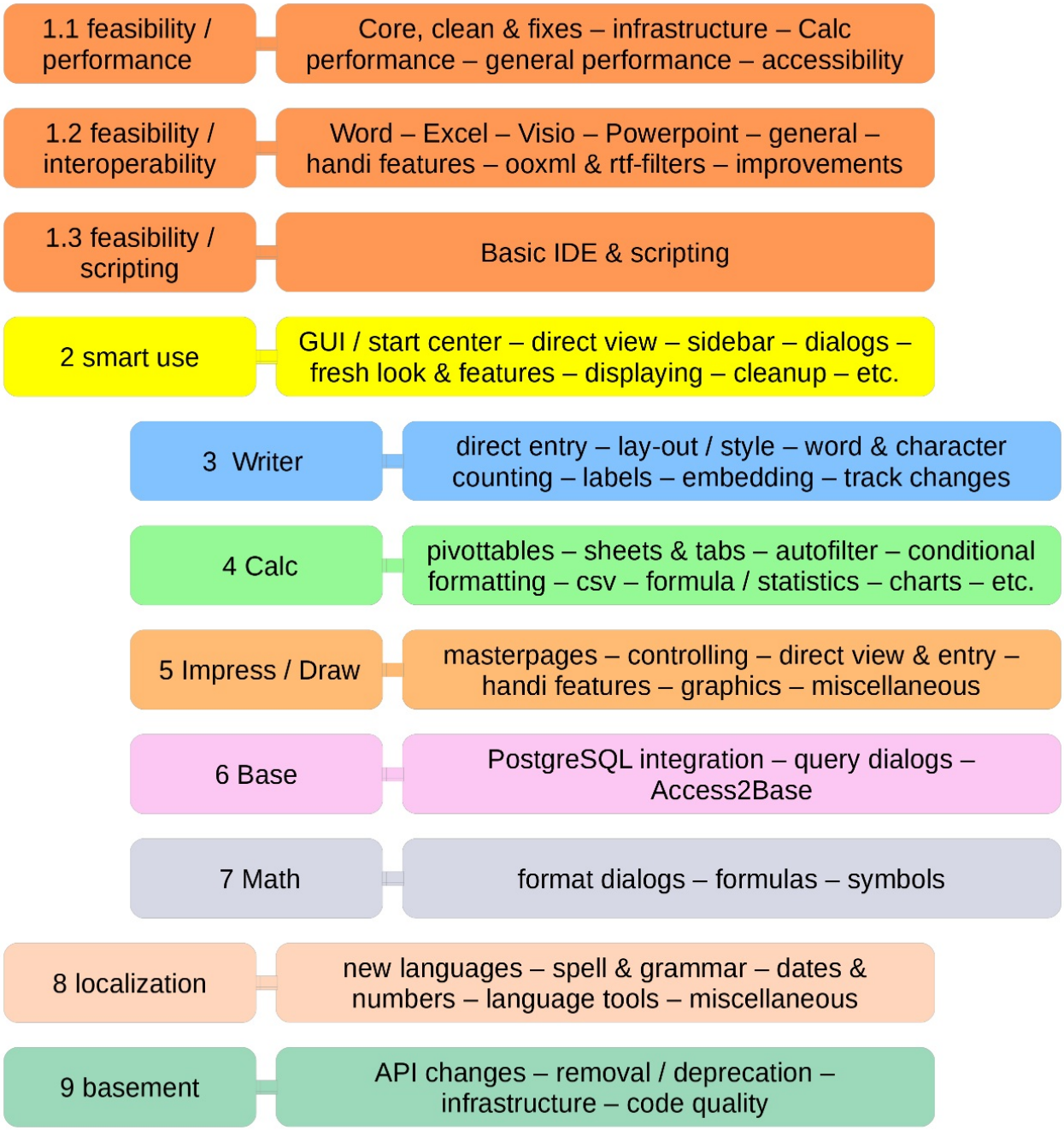
The evaluation starts by defining a model of the product to evaluate, and a grouping of its more relevant characteristics. The next picture shows a functional model of the LibreOffice and Apahe OpenOffice case.



Commparing LibreOffice and Apache OpenOffice: functional model. All the evaluation in the report is based on this model.

The functional model defines the main functional components of the software to evaluate. Now, we can define functional features of relevance, and evaluate each of them. The evaluation can be quantitative or qualitative. In the former case, boolean (the feature is available or not) and fractional (the feature is available to a certain fraction of some "ideal" feature) evaluations can be performed. For example, a certain feature can be present in a product, or "80% present" with respect to some ideal feature. In the latter case, an expert provides a detailed description of each feature. In many cases, both evaluations can be present, since both can be relevant.

In the case of the comparison of LibreOffice and Apache OpenOffice, the next picture shows the main relevant features for evaluation, grouped according to the modules defined in the functional model.



Comparing LibreOffice and Apache OpenOffice: identification of relevant features

Which are later refined in specific features, which are evaluated to be present or not ("+" or "-" in the next figure). In this case, the functionality is related to the corresponding changes in the source code during a certain period. But this is not necessarily the case. A very similar evaluation can be performed just by defining functional aspects and then verifying if they are present or not in a certain product.

localization		
8 WORLDWIDE NATIVE USE AND TOOLS (SINCE VERSION 3.4)	OFFICE SUITE	
spell and grammar	AOO 4.1	LO 4.4
Updated German spell check dictionaries, hyphenation and thesaurus. [3.4 | 3.4]	+	+
Duden Korrektor extension 5.0 and 6.0 added. [3.4 | 3.5]	+	+
Possessive genitive case and/or partitive month names if provided by a locale's locale data. [-.- | 3.5]	-	+
Corrections to Polish [pl-PL], Portuguese [pt-PT and pt-BR], Slovenian [sl-SI], and Latin [la-VA] locale data, esp. date formats. [-.- | 3.5]	-	+
Bundled LightProof grammar checkers for English, Hungarian, and Russian. [-.- | 3.5]	-	+
Checks: punctuation, a/an article, word spacing, paragraph capitalization, simple word duplication etc.. [-.- | 3.5]	-	+
Longer explanations, using Wikipedia articles. [-.- | 3.5]	-	+
Translated key names on Windows for Asturian, Catalan, Estonian, Slovenian and Spanish. Translated key names on Linux for Slovenian. [-.- | 4.0]	-	+

Comparing LibreOffice and Apache OpenOffice: evaluation of features

This information can be later used to produce a report on the functionality found, in a comparison between different products, etc.

Evaluation of suitability

Example: OpenBRR

Evaluation of quality

Example: QSOS, Qualoss

Evaluation of maturity

Example: Polarsys Maturity Model

Evaluation of community and development processes

Example: The Bitergia evaluation

Sources of information

When evaluating FOSS projects there are many potential sources of information, which in many cases, and specially when the project follows the open development paradigm, are public. Some of these sources are:

- Source code management systems (SCM), such as git, Mercurial or Subversion. All changes to the source code, and in some cases, to documentation and other related artifacts, are stored in them. From these systems we can extract any past version of the source code, which allows for code analysis, code inspection, etc. SCM systems store metainformation for every change, which usually includes who authored and committed it, when, the files involved with the corresponding diffs, etc.
- Issue tracking systems (ITS), also names ticketing systems or bug reporting systems, although they are used for much more than reporting bugs. Most projects use them for tracking how bugs are reported and fixed, and how new functionality is proposed, defined and built. But they can also use ITS for tracking the policy decision or the code review process, or even for tracking how the infrastructure is managed. Their repositories are usually modeled as tickets, which experiment changes in state until they are closed. Some examples are Bugzilla, Lanuchpad, GitHub Issues, Allura Tickets, Trac or RedMine.
- Code review systems (CRS). Used to track the code review process. They are usually modeled very similar to ITS, but specialized for code review, and in many cases integrated with SCM. Some examples are Gerrit and GitHub Pull Requests.
- Asynchronous communication systems (ACS). The most classical ones are mailing lists and forums. But more recently others are emerging, such as question/answer systems, of which StackOverflow and Askbot are good examples. The repositories for ACSs are usually modeled as messages, grouped in threads, either implicit or explicit. Each message includes content, but also metainformation such as author, date, etc.
- Synchronous communication systems (SCS). Some examples of them are IRC or more recently, Slack.
- Testing and continuous integration systems (CI). Some examples are Jenkins, Hudson and TravisCI.
- Web sites and other repositories for content related to the project. They may include documentation, download areas with binaries ready to run, etc.

All these systems usually offer means for persons to interact with them, which can be used to get a first hand impression of how the project is using them. Qualitative evaluations can benefit from this kind of browsing of information. This can be done, for example, by reading comments in commit records, messages in mailing list archives or IRC logs, or the history of tickets in ITSs.

It is also possible to use tools to retrieve information from them, usually via APIs designed for that matter. This allows for the automated retrieval of information for performing quantitative evaluation, or for storing all the data in a database for further analysis.

Source code management systems

Information in SCMs is usually organized in "changes", which in most systems are named "commits". The information stored with each change is different for different SCM systems, but it always incude the change itself (some way of identifying which lines were affected by the change, and how), and some metainformation. The metainformation is about who and when produced the change, and some other information related to it. For example, in the case of git, that metainformation associated with each change includes:

- A unique identifier for the commit (dubbed the "commit hash").
- The commit id of the previous commit(s) in the history of the repository.
- An identifier for the author (the person writing the change).
- An identifier for the committer (the person committing the change to the repository).
- Dates for authoring and committing actions.
- A comment produced by the author of the change.

- Complete diff with the changes (differences introduced by the change with respect to the previous situation).

An example of a git commit record

To illustrate how to obtain this information, see below an example. It is the information in a git commit record, as produced using the command `git show --pretty=fuller` for a commit in the CVSAnalY repository (excluding the diff):

```
commit 364f67f13b0046c0a0a688b30a1341ff9946ac26
Author:     Santiago Dueñas <sduenas@bitergia.com>
AuthorDate: Fri Oct 11 12:55:44 2013 +0200
Commit:     Santiago Dueñas <sduenas@bitergia.com>
CommitDate: Fri Oct 11 12:55:44 2013 +0200

    [db] Add author's commit date

    Some SCMs, like Git, make a distinction between the dates when
    the committer and author pushed the changes.
    ...
```

The first line shows the commit hash, the next four are the authoring and committing identifiers and dates. In this case, author and committer are the same person, and authoring adn committing dates are the same. After those come several lines with the comment produced by the author, describing the change. After the comment comes the diff, with the list of changes (which were ommitted in the snippet, see below).

The raw information in the commit record can be obtained with `git show --pretty=raw`:

```
commit 364f67f13b0046c0a0a688b30a1341ff9946ac26
tree c121da67fcba250490b6b326deae46f041b76626
parent 99acc4d7762e3773751f17ae9f0b58169f5e4de0
author Santiago Dueñas <sduenas@bitergia.com> 1381488944 +0200
committer Santiago Dueñas <sduenas@bitergia.com> 1381488944 +0200

    [db] Add author's commit date

    Some SCMs, like Git, make a distinction between the dates when
    the committer and author pushed the changes.
    ...
```

This format is harder to read for humans, but includes more information (such as the previous commit, or "parent"), and is therefore more useful for mining data.

The first lines of the diff that was omitted for the above commit are shown below:

```
diff --git a/pycvsanaly2/DBContentHandler.py b/pycvsanaly2/DBContentHandler.py
index 579e103..4b0066d 100644
--- a/pycvsanaly2/DBContentHandler.py
+++ b/pycvsanaly2/DBContentHandler.py
@@ -149,7 +149,7 @@ class DBContentHandler (ContentHandler):
             self.actions = []
             profiler_stop ("Inserting actions for repository %d", (self.repo_id,))
         if self.commits:
-            commits = [(c.id, c.rev, c.committer, c.author, c.date, c.message, c.composed_rev, c.repository_id) for c
+            commits = [(c.id, c.rev, c.committer, c.author, c.date, c.author_date, c.message, c.composed_rev, c.reposi
             profiler_start ("Inserting commits for repository %d", (self.repo_id,))
             cursor.executemany (statement (DBLog.__insert__, self.db.place_holder), commits)
             self.commits = []
diff --git a/pycvsanaly2/Database.py b/pycvsanaly2/Database.py
index dacf406..9b02909 100644
--- a/pycvsanaly2/Database.py
+++ b/pycvsanaly2/Database.py
```

The first lines of the diff shows how to invoke the diff command to produce the output for the first file changed by the commit (a refers to the situation before the change, b to the situation after the change). Lines changed are those starting with - (removed) or + (added). In this case, a change in a line is modeled as removing the old line and adding a new one with the change.

Notes on using information from SCM systems

Identifiers for authors and committers are usually a name and an email address. But in some cases, such as Subversion, only a user name is available.

In decentraliized SCMs, such as git, dates for authoring and committing include local timezones, usually those of the computer in which the corresponding action was performed. This allows for timezone analysis to infer regions where developers work. In addition, since times are local, studies on daily schedules can also be performed. But in the case of centralized SCMs, such as Subversion, those dates are for the server where commits took place, which means that no information about local time for developers is avilable.

The information in the SCM allows for the reconstruction of the whole history of the repository. In the case of git, the information about the previous commit or commits (in the case of merges, there is nore than one previous commit) allows for the recovery of the full history of the repository. Using that information and some other hints, you can decide to which branch a change was committed.

The complete diff whcih is available for each change allows for the complete reconstruction of the code modifications, which is the ultimate reason for storing it. But it can be used for infering the files involved, the size of the change, and other parameters.

It is important to note that, from a historical point of view, the information provided by the SCM system is now always reliable. For example, in the case of git, developers can "rewrite" history, when they ammend or rebase. Therefore, a current retrieval of information from a git repository may show different data for some commits in the past than a similar retrieval done some time ago, or even a dfferent list of commits. For systems which only commit with an automated tool after code review, which checks and forbids history rewrites, this is not an issue. But most projects do not have specific rules or technical measueres to avoid history rewrittings, and therefore any results about past history need to be understood as "current past history".

Issue tracking systems

Information in issue tracking systems is usually organized in "tickets". This is why they are also called ticketing systems. In fact, the job of the ITS is to track the changes to each ticket. Therefore, most ITS maintain both a record for each ticket with its current state, and a history of all changes to their state, or a list of past states.

The information usually found for a ticket is:

- Identifier. Unique identifier for each ticket.
- Summary. A one line text summarizing the purpose of the ticket.
- Description. A longer description of the purpose of the ticket. This can be reporting a bug, requesting a feature, or starting a new task, for example.
- Opening date. When the ticket was opened.
- Ticker opener. An identifier for the person opening the ticket. Depending on the ITS, this can be a full name, an email address, or a user name.
- Ticket asignee. The person asigned by the project to deal with the ticket.
- Priority. A number or a text informing about the priority for the ticket. This is usually set by the ticket opener, but can be later adjusted by developers.
- State. A tag with the current state of the ticket. Examples of states are "open" (ticket opened, but still not dealt with by the project), "assigned" (ticket already assigned to some developer), "fixed" (ticket is considered to be fixed), "closed"

(the issue is considered to be done).

Almost all fields in the ticket are subject to changes. When this happens, the change is recorded, including information about who made the change, when, and what the change was about.

In addition, a ticket usually have an associated list of comments. These comments are posted by the different persons who contribute to close the ticket. Some of them may be from the ticket opener, such as when a clarification is posted. Some others may be from people interested in the ticket, such as other people experiencing the same problem. Some others may be by developers trying to solve the issue. Each comment is composed by some text (the comment itself), the posting date, the author of the comment, and maybe some other fields.

But despite having a similar structure, tickets in different ITS may be presented to users in very different ways, as is shown in the next examples.

Example of ticket in GitHub

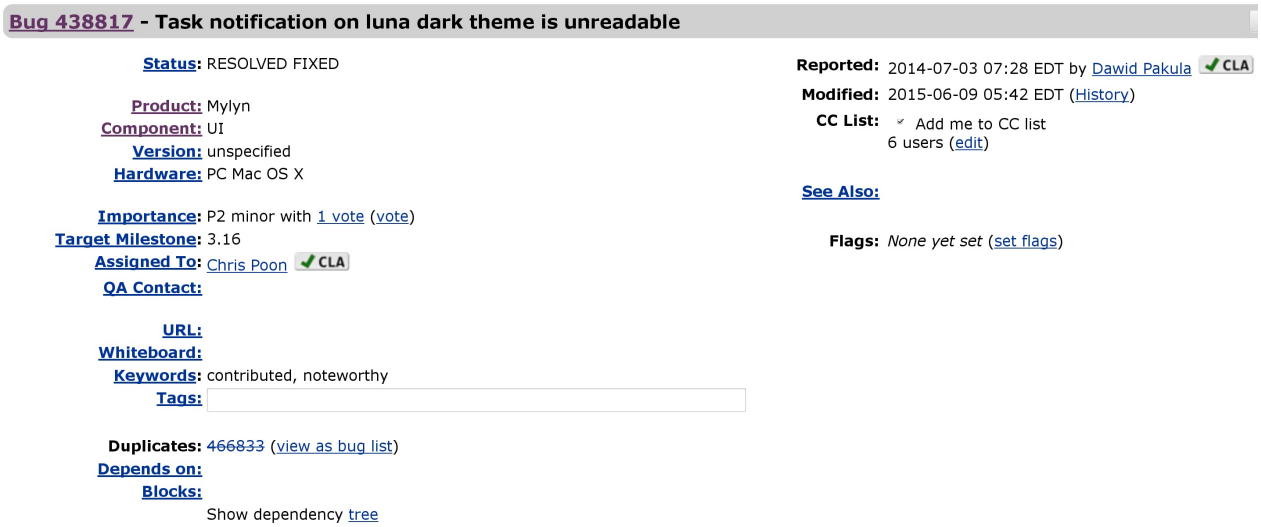
In GitHub, tickets are called "issues". They are presented in a single HTML page, showing the description, history and comments in it. Most of the current state is shown in the right column.

Example of ticket: GitHub issue from the Bicho project. The most relevant fields can be observed. The first text on the left column is the description of the ticket, the second one is a comment.

GitHub also provides the same information via the GitHub Issues API, which is easier to use for automated retrieval of information.

Example of ticket in Bugzilla

In Bugzilla tickets are called "bugs". Bugzilla is one of the earliest free software ITS, and is currently in use by some very large communities, such as Eclipse. In the following example, the status information for a ticket is shown. Below that area, the list of comments to the ticket is found, starting by its description.



Bug 438817 - Task notification on luna dark theme is unreadable

Status: RESOLVED FIXED
Product: Mylyn
Component: UI
Version: unspecified
Hardware: PC Mac OS X
Importance: P2 minor with 1 vote (vote)
Target Milestone: 3.16
Assigned To: Chris Poon CLA
QA Contact:
URL:
Whiteboard:
Keywords: contributed, noteworthy
Tags:
Duplicates: ~~466833~~ (view as bug list)
Depends on:
Blocks:
Show dependency tree

Reported: 2014-07-03 07:28 EDT by Dawid Pakula CLA
Modified: 2015-06-09 05:42 EDT (History)
CC List: Add me to CC list
6 users (edit)
See Also:
Flags: *None yet set* (set flags)

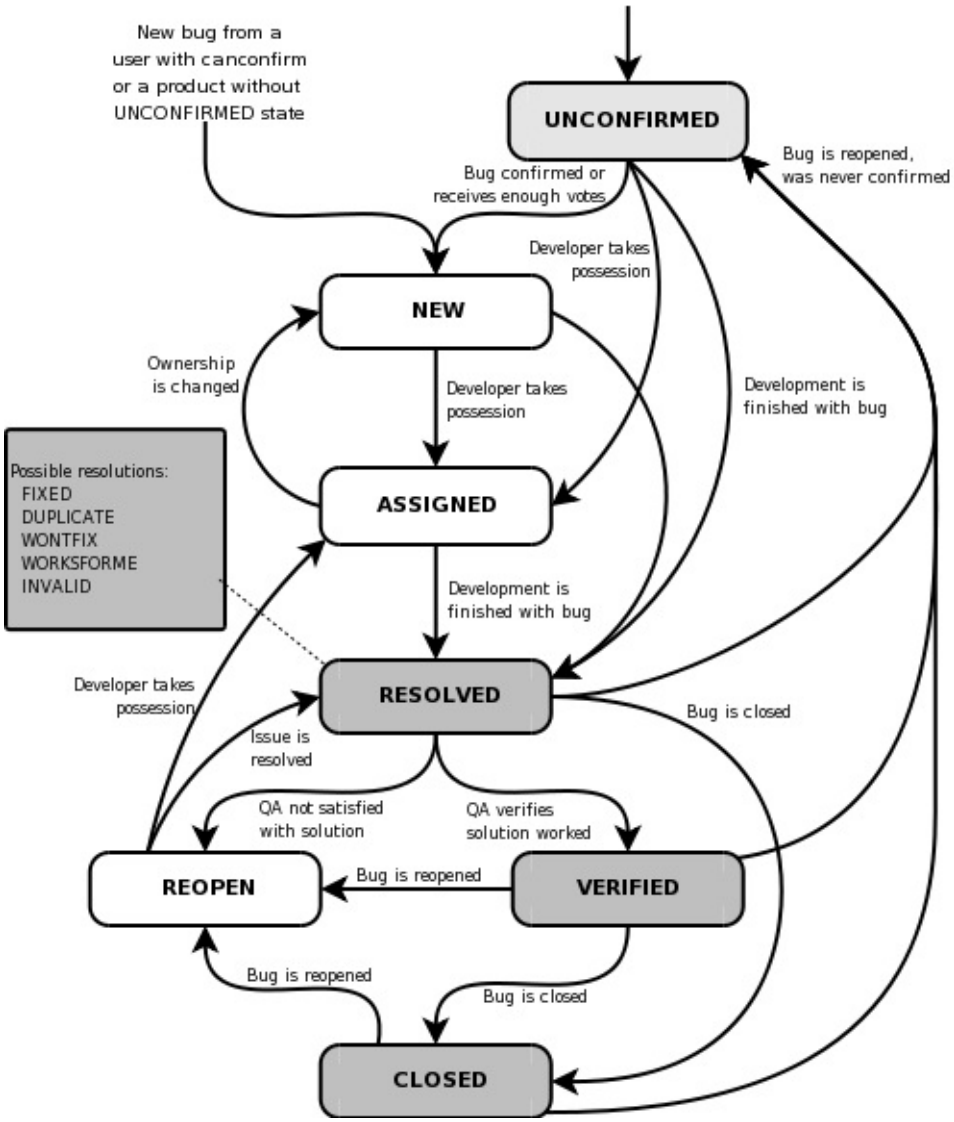
Example of ticket: Bugzilla issue from the Eclipse Mylin project. Comments come below this status information.

Bugzilla provides similar information via an XML file, which is more suitable for automated retrieval.

Notes on using information from an ITS

First of all, it is important to notice that projects may use tickets for many different kinds of actions. The most known one is reporting a bug, and that's the reason why these systems are also called "bug reporting systems". But tickets may refer to feature requests, tasks being performed, or even policy discussions. It is up to the policies and uses of a project to decide what kind of communication they channel through the ITS, and that varies a lot from project to project. Therefore, counts on for example "open tickets" are different from counts on "open bugs".

ITSs are also very different on which kind of information they store, and what does it mean. For example, all of them include some kind of encoding the "state" of the ticket. But while GitHub uses tags for that, Bugzilla uses a "status" field. In addition, each project may use this information in different ways, and in many cases they define their own tags, status fields, or whatever the ITS uses for this. That makes it very difficult to know at first sight even when a ticket is a bug, or when a bug is actually fixed. The workflow for tickets, and how and who can move them from state to state, is usually project-defined,although the ITS may constrain or recommend about it. For example, the next figure shows the recommended workflow for Bugzilla.



Recommended workflow in Bugzilla. Original figure illustrates Life Cycle of a Bug, in The Bugzilla Guide - 3.6.13 Release distributed under the GNU Free Documentation License.

This said, ITSs provide a rich information about how the project is dealing with some of its most important development processes. For example, they allow for the calculus of time-to-fix for bugs, or time-to-implementation for feature requests. They allow as well for the identification of the persons carrying on most of the maintenance, or the key developers in implementing some kind of new functionality.

In some cases, the ITS carries too processes such as code review or documentation management, which have their own peculiarities.

Code review systems

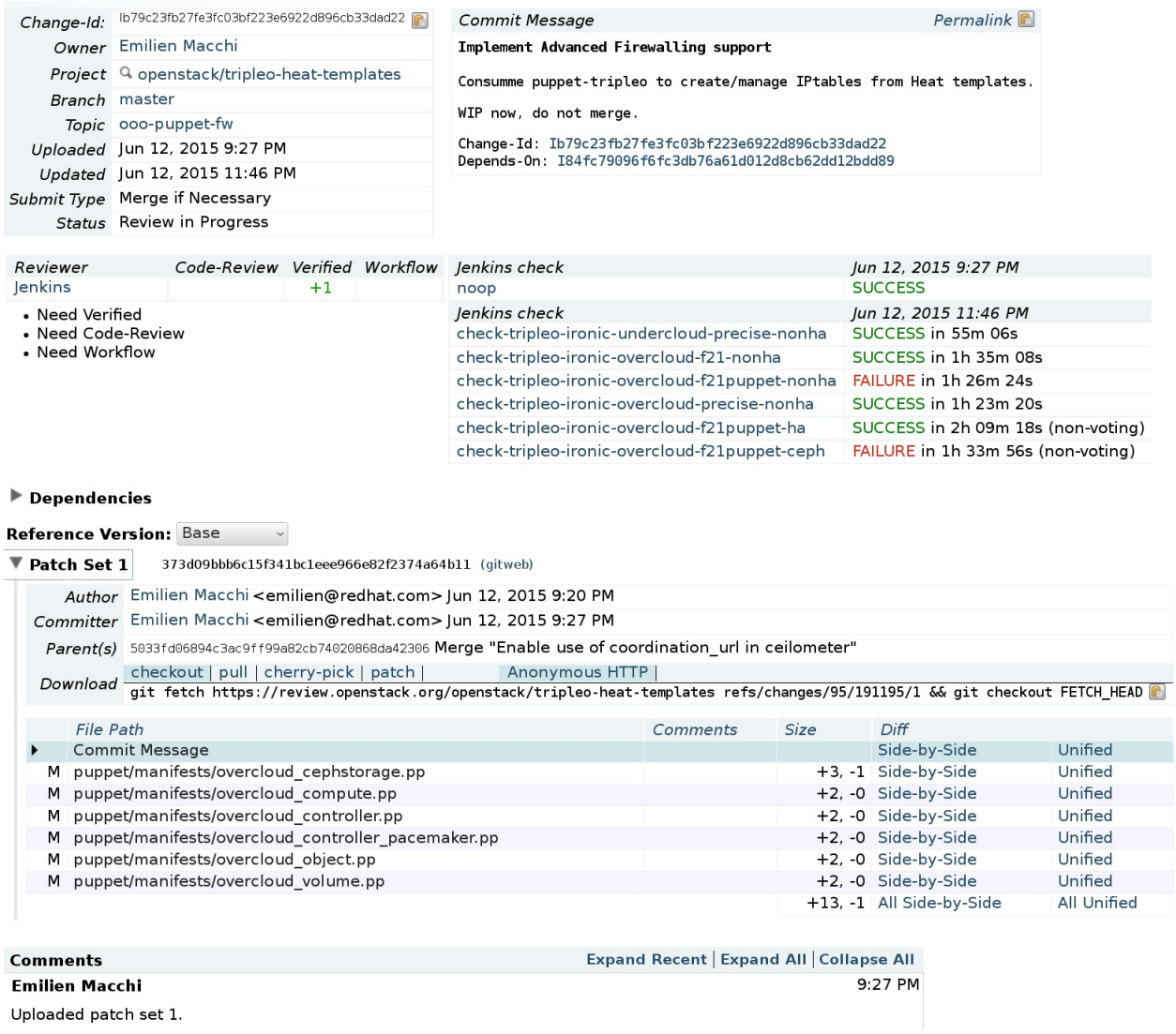
Code review systems are used, as their name implies, for reviewing source code. Changes to code are organized as patches, which are submited by developers to the system, and then commented and reviewed by reviewers. Depending on the project, maybe only some developers can review, or only some can accept or decline changes.

The information structure of a CRS is quite similar to that of ITS. The role of tickets is taken by proposed patches. Proposed patches go through changes in state as the review cycle progresses. Some reviewers propose to accept the change, some others may ask for a new version, finally the change is accepted or maybe abandoned. All of them (reviewers and change proposers) can write comments as well.

Due to this similarity, some projects use the ITS for code review. This is for example the case of WebKit, which uses Bugzilla. But during the last years, specialized systems for code review have been adopted in most cases. The most prominent cases are Gerrit and GitHub Pull Requests.

Example of code review in Gerrit

Gerrit tracks "changes", usually in combination with git. Each change is a commit, which is reviewed by flagging it with +1 (proposal for accepting the change) or -1 (proposal for asking for a new version of the change). Each version of a change is called "patchset", and the developer is expected to submit one patchset after another until reviewers are happy with the change. In this case, the acceptance of the change is signalled by flaging it with +2. Gerrit can be tuned to the specific policies of a project, so that for example a certain number of +2 is needed to accept (merge) a change, or that only core reviewers can flag a change with +2.



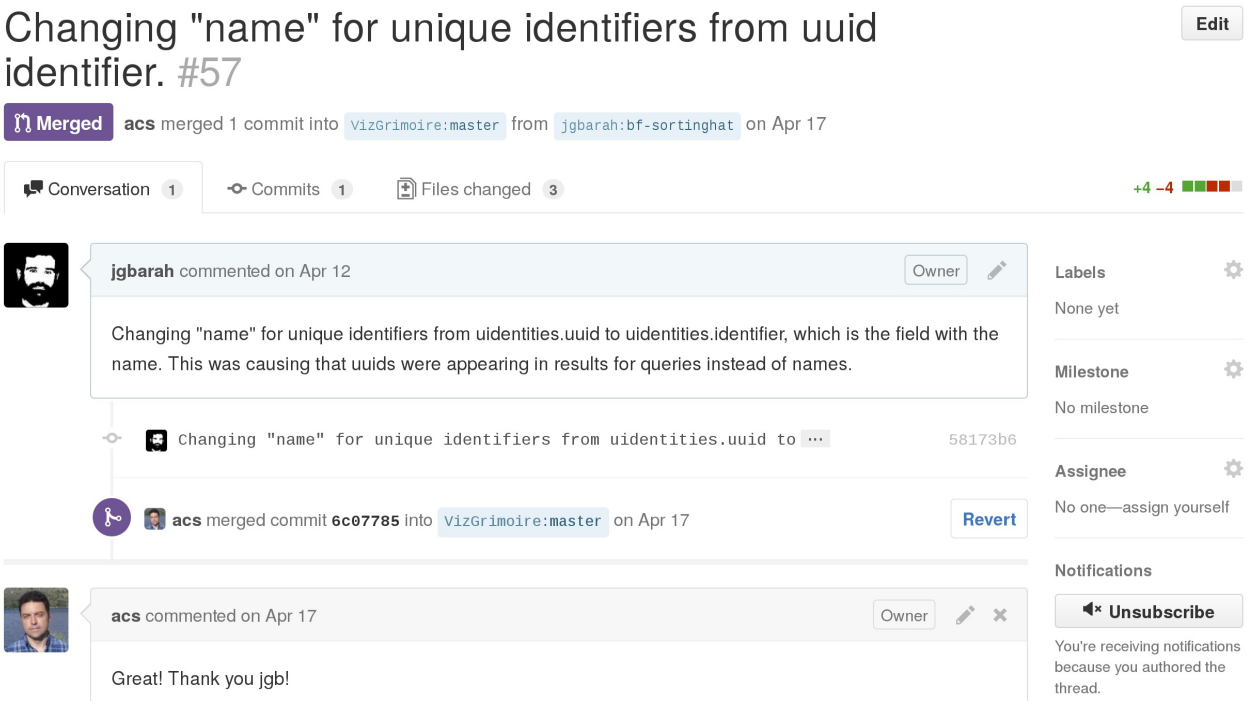
Example of code review: Gerrit code review for OpenStack

Gerrit can also be connected with testing and continuous integration systems, so that automated tests are run before and after a change is reviewed. A change submitted to Gerrit finishes its review process, possibly after several patchsets are submitted, when it is accepted (merged), or when the developer decides to withold it (abandon).

Example of code review in GitHub

In GitHub, code review is done via "pull requests", which are a kind of specialized tickets (issues). The process starts when a developer submits a change. To prepare for that, they usually have previously forked the corresponding git repository, and

committed the proposed change to it. Then, that commit can be proposed for pull request via the GitHub web interface. Once it is proposed, it appears to GitHub users as a ticket, with the peculiarity that the corresponding change can be explored, commented, and merged.



Example of code review: GitHub pull request

GitHub does not allow for tracking different versions of the change. In fact, different versions can be submitted, by changing the commit (eg, ammending it). But that information is lost when the corresponding ticket is browsed: only the last version of the change is available.

Since a pull request is really a ticket, it can also be labeled, assigned, closed, etc.

Notes on using information from a CRS

CRS are very important when tracking processes in software development. When a project uses mandatory code review, any new piece of code has to be through the CRS, which tracks times, people involved, etc. Therefore the information mined from it can be used to learn about how long does code review lasts, and who is responsible for delays in code review: developers who fail to submit quiclly new versions of a change, or reviewers who are slow in reviewing it.

People involved in review processes is also a very interesting information. Commit records only keep information about the author and maybe the person merging the commit. Code review provides a much more detailed information: all people commenting, or proposing approval or decline of chaneges are tracked. In addition, CRS usually provide links with testing and continuous integration (CI) systems, which may say a lot about how good proposed changes are.

In many cases, tickets refering bug reports or feture requests corresponding to the change are linked as well. This allows for more complete analysis of all the development process. For example, that allows to track, since the moment a bug is reported to when a fix is proposed, how it evolves to the final merged change, how it finally passes the CI tests, until it is ready for deployment in production environments.

Asynchronous communication systems

In the early ages of FOSS projects, most communication was asynchronous. The most popular tools were email (using mailing lists) and USENET News (in some cases, connected to a mailing list). With time, web forums became important in some communities too, and the relevance of USENET News declined, to the point of fading away. Some special-purpose

forums, such as StackOverflow or Reddit, emerged during the last years as important points of communication, even if the projects didn't decide on using them.

Each of of these systems organize information in a different way, although they have some aspects in common. All of them are organized around "messages" (email messages, posts in a forum, questions in StackOverflow, etc.) which have an author, a date, and in most cases a one-line summary. In most cases messages are related in threads, usually as they reply or mention each other. But common aspects stop here. The structure of messages, how threads are organized, and other ancillary information are different from system to sytem.

Example of ACS: email message in Mailman

Mailman is a mailing list manager. You can create mailing lists, and manage them via a website. It has been one of the most usual choices for handling mailing lists during the 2000s. It provides an HTML interface, which gives access to all messages. But it also offers archives in mbox format, which is much easier to mine. The information stored in those archives includes usually most of the headers in the original messages. Below you can find an example of one of those messages, as seen via the HTML interface.

[Metrics-grimoire] Logo for MetricsGrimoire github organisation?

Luis Cañas-Díaz lcanas at bitergia.com
Mon Mar 2 09:32:19 CET 2015

- Next message: [Metrics-grimoire] Logo for MetricsGrimoire github organisation?
- **Messages sorted by:** [date] [thread] [subject] [author]

```
On 28/02/15 11:13, Andy Grunwald wrote:
> Hey,
>
> the organisation page of https://github.com/MetricsGrimoire looks quite
> sad without logo.
> I am not a big designer or creative in designing logos. Maybe you got an
> idea for an organisation logo and apply this to the organisation?
>
> With a logo this looks much more happy :)

You're right. I think Santi had something for Grimoire with a logo with
a book of enchantments. Santi, don't you?

Best.
>
> Andy
>
> PS: This applies for https://github.com/VizGrimoire also.
```

Example of asynchronous communication: email message in Mailman

The main fields of information can be seen in this message: the subject (a summary of the message), the sender (including email address, slightly mangled), the date, and the contents. Some other headers may be available.

One important detail is about the date. In most cases, this includes at least the date as stamped by the mail server (usually, Mailman itself). But it can include as well the date of the mailer program user to send the message, usually set to the local time of the sender. This means that both analysis by local time and by universal time are possible.

The contents of the message can be huge, since they may include attachments. Depending on the configuration, attachments may be available or not.

Mailman deals with the list of subscribers to the mailing list, but it doesn't track its history. Therefore, the current list of subscribers can be retrieved, but it is not possible to obtain past lists for a historical analysis.

Privacy sets may interfere with mining, even for public mailing lists. For example, archives can be available only for subscribers, or email addresses may be mangled. Both cases make mining a bit more complicated, or make some analysis impossible.

A final comment: public mailing lists can always be subscribed to. This mean that a miner can subscribe, and produce an archive with all details present in incoming messages. The history before the subscription won't be available, fut from that point on, all details can be easily accessed.

Notes on using information from ACS

It was Apache the first FOSS community to explicitly state that "If it didn't happen on a mailing list, it didn't happen". Since them, but also before them, many others have put in practice this principle, even without stating an specific policy for enforcing it. This means that mailing lists and forums are of great importance to track the coordination activities of projects, and how they discuss and take decisions.

But archives for forums and mailing lists are not always available, or they are available only in part, or they are available in ways that are difficult to mine. For example, some projects don't keep archives of some periods of their history, or rely on systems that even when they perform archiving, are very mining-unfriendly. A notable case is Google Groups, which is used as the ACS by many projects. This system doesn't have an API for mining information, which means that web spidering is the only way to retrieve information. On the contrary, several mailing lists software (such as Mailman) and systems (such as GMane) have very good facilities for automatically retrieving archived information.

Privacy of email addresses is a bit problem for mailing lists archives, but also for mailing list miners. To prevent spam, may archives mangle mailing lists, in a way that it is impossible to know the actual email address of the sender of a message. This means that identification of developers cannot be done based on email addresses, as it can be done for example for git repositories. Therefore, it is not possible to know if a certain poster is the same person that authored a commit. Some types of analysis and evaluation, which rely on this identification, can therefore not be done in those cases. An example is the analysis of how many developers participate in a mailing list.

Forums have also their own problems. First of all, each forum has its own peculiarities. Some of them have APIs which make mining very simple. But some others have APIs not really designed for mining, lacking fundamental capabilities like incremental searching of posts, which makes it complex and resource-demanding to retrieve their information. In some other cases, APIs simply don't exist, and the only way to mine messages is to get database dumps for all the infomation they store, which is obviously difficult.

Synchronous communication systems

The traditional synchronous communication system of choice in FOSS projects has been IRC (Internet Relay Chat). Those are used for casual conversation, support to users, quick discussions between developers, and even for socializing.

Other communication channels used by FOSS projects have been those based on XMPP, such as those provided by Jabber, and more recently, Slack.

The information obtained from all these systems can be organized in a similar way. The unit of communication is the message, which can be related to its author, and to the date it was posted. In most of these systems, communication is organized in channels, and therefore messages can be related to a channel as well.

Example of SCS: IRC log

IRC channels on Freenode are one of the most popular synchronous communication channels used by FOSS projects.

People participating in the channels use some software to connect to Freenode. That software lets them see all messages posted to the channel while they are connected, and send messages as well. The software also informs about who is connected to the channel, and produces notifications when new people join or leave it.

To log all these interactions, usually IRC bots are used. Those are programs that connect to the channel, and record all interactions received from it. These bots produce files with those logs that the project uses to make conversations public, and to preserve them for the future.

*** RichardRaseley has joined #openstack-operators		[17:28]
*** belmoreira has joined #openstack-operators		[17:30]
mdorman	anybody know if the rax keystone extension for api keys is open sourced? http://docs.rackspace.com/openstack-extensions/auth/RAX-KSKEY-service-devguide/content/	[17:34]
*** zul has quit IRC		[17:38]
*** zul has joined #openstack-operators		[17:40]
*** harlowja has quit IRC		[17:58]
*** harlowja has joined #openstack-operators		[18:02]
jlk	I don't see it in the normal spot	[18:02]
jlk	github.com/rackerlabs	[18:02]
mdorman	kk	[18:09]
mdorman	seems like that might be a proprietary thing, was just curious	[18:09]
mgagne	AFAIK, I think you need the client auth plugin too anyway to make it work	[18:10]
mdorman	makes sense	[18:11]
*** bradjones has quit IRC		[18:23]
*** bradjones has joined #openstack-operators		[18:24]

Example of SCS: Log of the #openstack-operators IRC channel of the OpenStack project.

The figure above shows a fraction of the log produced by one of these bots. It can be seen how both messages and join/leave notifications are recorded, and how for each of them the time is available (the date is implicit, since one log file is produced per day). The identifier for the person sending each message or entering or leaving the channel is recorded as well.

In IRC servers, person identifiers can be reserved, so it is usual that frequent participants use always the same name. But this is not always the case: non registered identifiers are available for anyone to use.

Notes on using information from SCS

One of the main trouble with SCS systems is that it is not usually easy to track the identities of people participating in the channels. Nothing like email addresses is available. Although the regulars in the channels are usually identifiable, for most participants this is not the case.

In some cases there are privacy concerns with logging SCS channels. Although they are usually public, to comply with the openness that most project mandate, they are not perceived by some developers as mailing lists are. Due to their immediacy, people may be inclined to forget that what they say is public, and may find themselves saying words that they wouldn't really say in public. When bots are used for recording, some people may be reluctant to participate, for fearing that those cases are recorded and linked with them in public. However, although these concerns exist, public channels are really public, and most projects are dealing with them exactly as they do with public mailing lists: recording, archiving, and making archives public.

SCS recordings are a good data source to check for people with high involvement in the project. Being available for participating in discussions with anyone joining the channel, or answering quick questions, are signals that can be interesting to track.

Testing and continuous integration systems

During the last years more and more projects are including an infrastructure for performing automatic testing. Continuous integration, when used by projects, usually require that those tests are run for each change, so that automatic tools can decide whether it is safe to integrate the change. When the project is using code review, automated testing is usually a part of any review cycle, to spare time to reviewers, who can focus on those changes that passed the testing process.

Some of the most used systems for continuous integration in FOSS projects are Jenkins / Hudson and Travis.

Impact on the infrastructure of the projects

Mining may have a significant impact on the performance of the infrastructure of the project. In fact, some mining activities can be identified by the sysadmins for that infrastructure as a kind of DoS (denial of service) attack. But even when they are not, those activities can cause a log of stress in the project infrastructure. For example, retrieving all information from the ITS, means querying it for all tickets, and the for each ticket obtain all changes and comments. This is not only a significant effort for the ITS, but also an effort for which it was not really designed.

ITS should provide quickly listings of open tickets, or recent tickets assigned to some developers, or the most recent status for a certain ticket. But the actions performed by programs retrieving information for mining are very different from this "usual behavior".

There are some practices that should be followed to ensure that impact on the project infrastructure is minimum:

- Design the retrieving tools as much repository-friendly as possible. For example, use delays between operations, so that the system is not "hijacked" by the program, letting users perform their tasks
- Contact the project for advice in case it is anticipated that the infrastructure is going to be stressed. Project sysdamins can help to design retrieving scenarios that cause as little harm as possible.
- Whenever possible, use archives with previously retrieved information. In this case, the information is retrieved only once, but can be later consumed for many different studies by different parties. If the information is reliable enough, there is no need to stress the infrastructure once and again. In addition, you don't have to retrieve the information yourself, and can concentrate in designing and performing your analysis.

Evaluation of FOSS communities

For many FOSS projects, the communities supporting them are the main responsibles for the evolution of the project. Evaluating the communities is therefore fundamental to evaluate the project.

Different scopes: developers, contributors, users...

FOSS communities are diverse, and may include many different actors. But in general, attending to the scope, the following, usually overlapping, communities can be defined:

- The development community, composed by people in charge of developing and maintaining the software produced by the company.
- The contributing community, composed by all the people actively contributing, not only with code. Examples of contributors are: submitters of bug reports, participants in discussions in mailing lists, translators, writers of documentation, etc. The contributing community includes developers too.
- The user community, composed by users of the software participating somehow in the community. This can be by asking questions, by attending events, by joining social network groups with interest in the project. etc. Usually the user community includes the development and contributing communities, since they are also users of the system.
- The ecosystemn community, composed by all stakeholders not only in the project itself, but in all the ecosystem of projects related to it. The user community is a part of the ecosystem community.

We define these communities to hightlight different populations that may be relevant for an evaluation. It is important to realize that their borders are fuzzy, and people move from one to another as time passes.

For each of these scopes, different evaluation means can be used. Despite the apparent diversity, we can also identify some techniques and parameters that are useful for all of them. The rest of this chapter will enter into the peculiarities of each community, and will show as well what they have in common and the techniques that are usful for evaluating all of them.

Development community

The development community is composed of the persons developing and maintaining the products produced by the project: software and related artifacts, such as documentation.

In the case of FOSS projects with open development models based on coordination tools, there is a lot of information available about them. Usually, data can be collected from the following repositories:

- Source code management repository. Almost all the information in is produced by the development community, since they are mainly changes to source code. In fact, one of the ways of defining the development communuity is as "those people who have contributed at least one change".
- Code review system. All reviewers can be considered as a part of the development communities. Most of the submitters of change proposals are also developers, or they are acquiring that status.
- Issue tracking system. Developers participate in ITS by opening, commenting and closing tickets. They are not the only ones opening or commenting, but usually only they can fix issues, and close tickets.
- Asynchronous and synchronous communication. It is very usual to have separate channels for developers, which allow for a separate tracking of this community.

In summary, most of the data in those SCM, CRS and ITS repositories are related to development activities, and developers usually have separate channels in ACS and SCS. Therefore, the evaluation of the development community in open projects, where all these repositories are public, can be very detailed.

Contributing community

The contributing community is a bit in between the development community and the user community. Contributors are usually users that are in the road to become developers. But they may or may not walk that path towards the development community. This makes it difficult to specifically track contributors who are not developers.

- Issue tracking system. Tickets are opened by contributors, be them developers or not. Since we consider both feature requests and bug reports as valuable contributions to the project, everything happening in the ITS is performed by the contributing community. However, the "responisve" part of the action is carried on by developers.
- Asynchronous and synchronous communication. Contributors may join development channels. But being they users as well, contributors are also present in user channels. This makes it difficult to track their activity, except that they can be identified in ITS, and their identity linked to ACS and SCS.

Since the difference between "contributor" and "developer" is a specially fuzzy one, in many cases both communities are considered as one. However, in some sense contributors are the pool where developers come from. People usually become developer after contributing to the project for a while. Therefore, for estimating the future of the development community, and the engagement of people who could become developers, studying the contributor community is specially interesting.

Users community

The community of users of a FOSS projects is much more difficult to evaluate than those of contributors and developers. In fact, even estimating the number of users is usually difficult. Usage is in most cases passive, in the sense that almost no interaction with the project is needed to become user. In most cases of non-FOSS software, to become user implies purchasing a license, which is an action that can be tracked. But in the case of FOSS software, it is enough to get the software somehow, and start using it. No red tape is involved.

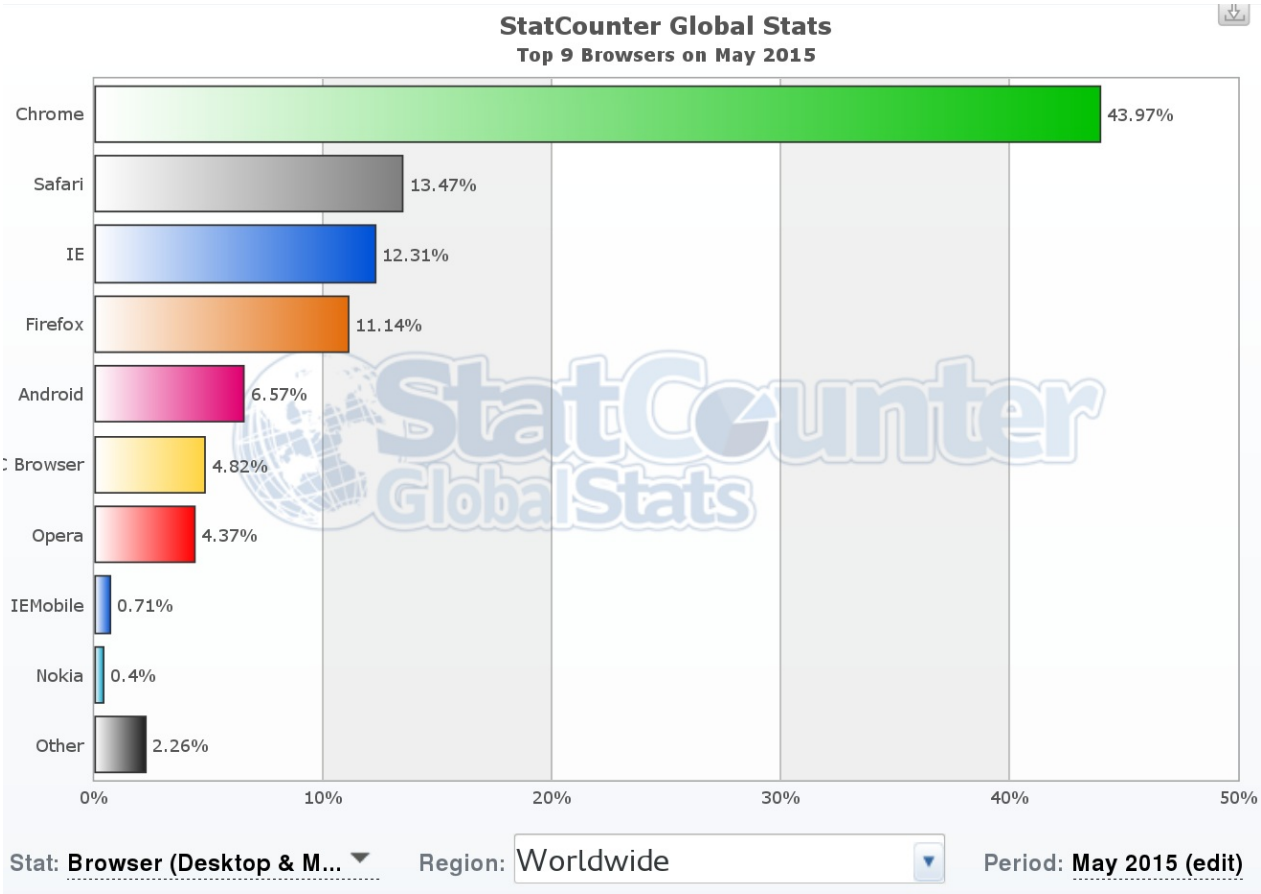
Therefore, the source of information to estimate the size of the community of users are indirect:

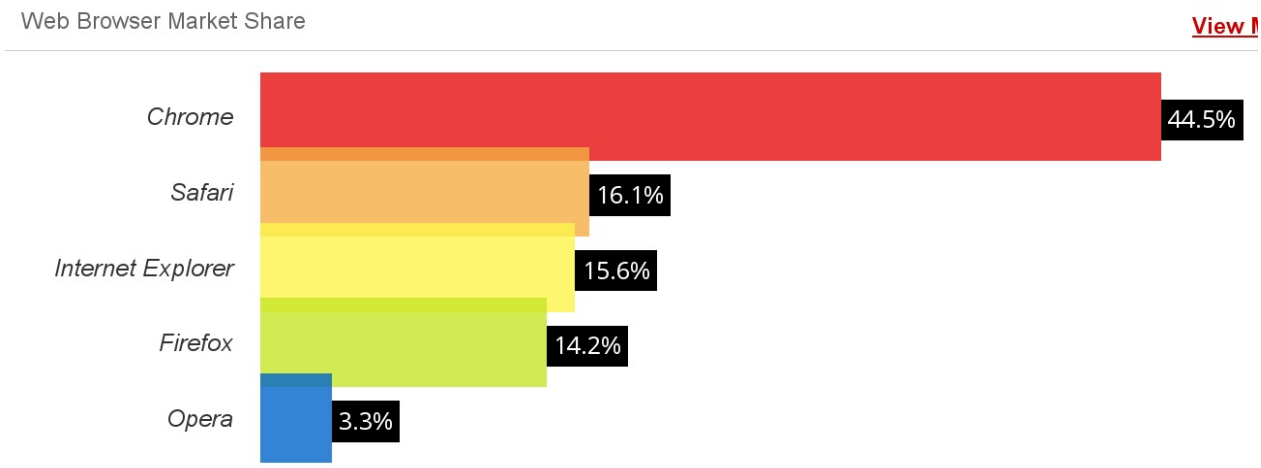
- Downloads. Many projects maintain a download area. When the primary usage of the product is via those areas, the number of downloads can be an estimator of the number of users. Of course, downloads of different releases have to be taken into account, and some model on when users reinstall with a newer version are needed. But this method can be enough to estimate trends and order-of-magnitude numbers. Of course, if most of the usage is not by direct download, the numbers are much less precise. This happens, for example, when the software is mainly available through FOSS distributions or via third party download areas.
- Questions and comments in user forums. Given that the ratio of users to contributors is very large in most cases, it can be assumed that questions and comments about the product in third party forums are mainly by users. Therefore, the number of those questions and numbers can be a proxy for estimating the user population. Some models to convert those numbers into number of users are needed, but again trends over time can be somewhat accurate.
- Presence in FOSS distributions. Some FOSS distributions maintain their own statistics about package (and therefore, product) installation. For example, in Debian the opt-in Popularity Contest maintains accurate stats of installed packages. From those numbers, and estimating the total population of Debian users, total usage in Debian can be estimated. From there, estimations for other distributions can be extrapolated. These numbers are probably not very accurate, but can provide an order-of-magnitude estimation.
- Answers to polls and surveys. Polls and surveys to specific populations or to the population in general can also be a source of information. This is a general technique to know about user adoption, which compared to the others has the main drawback of its cost. Only the really popular FOSS products will appear in general surveys, but in some cases this is enough to have an idea. For example, the usage of Firefox or Chrome web browsers can be estimated this way.
- Raw numbers in the Internet. Some services, such as Google trends, provide some information about how popular terms are in the web. In some cases, those numbers can be used to estimate trends in usage, assuming that the more popular a product, the more people appearences it will have in the global web.

There are some specific cases when a more concrete estimation, or at least a lower watermark for an estimation can be established:

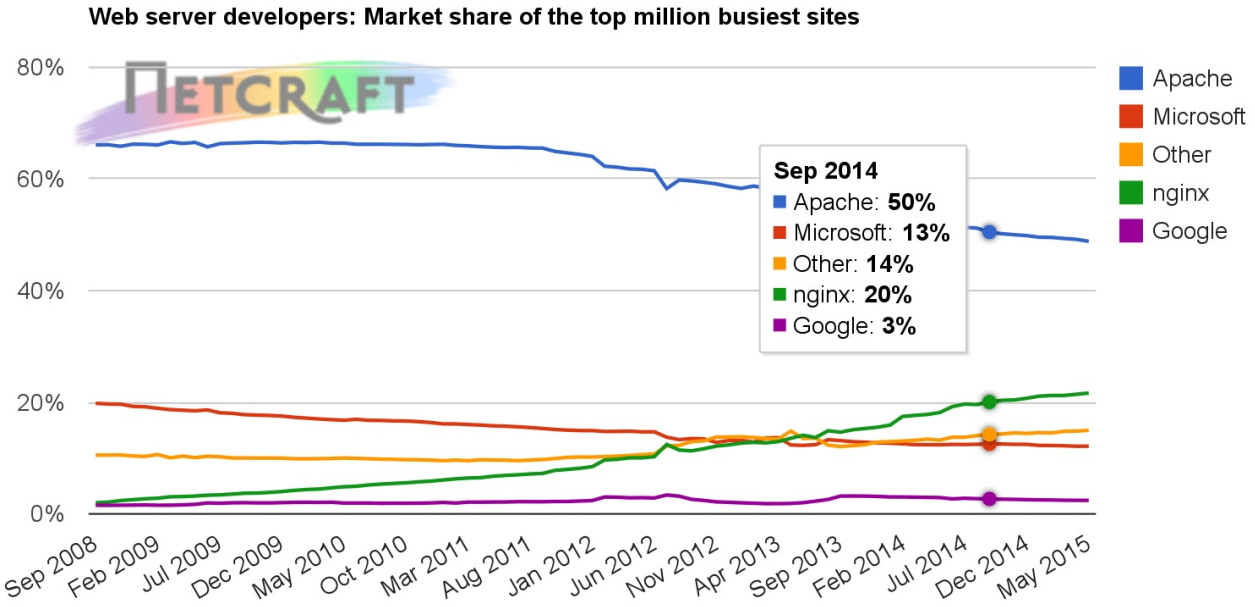
- Software sending "beacons" to the project. This can be desktop or mobile software connecting to a certain location with a "Here I am" message, or a web product including components that are downloaed from a certain website. In both cases, since FOSS software can be changed, maybe there are versions of the product with the beacon removed. In addition, maybe there are products being used withouth Internet connection. But when these cases can be neglected, the estimation of usage can be very good. A very specific case is when the software, as a part of their normal functioning, identifies itself somehow. For example, web browsers send identification strings to web servers. These strings can be used to estimate usage.
- Software which answers when queried. This a very specific, but very accurate case, when the software can be located and queried. The most well known case is the estimation of web servers, where the user base of Apache or nginx is tracked periodically by querying web servers all over the world for their identification string.
- Software distributed through markets. When the product is distributed mainly through a market (a mobile or a distribution app market), usually it provides detailed numbers about installations, deinstallations, etc.

These three cases are rare, but when they happen, estimations can be very accurate. The next pictures show some cases (web browsers, web servers) for which these methodologies can be used. That allows for the usage estimation of some FOSS products, such as Apache HTTP Server, nginx, Chrome or Firefox.





Example of usage estimation: StatCounter top 9 browsers (May 2015), top, and W3Counter web browser market share (May 2015), bottom. Both surveys are performed by using the identification information from web broswers in large collections of web sites. It is interesting noticing how they differ, even when they seem to use similar methodologies.



Developer	April 2015	Percent	May 2015	Percent	Change
Apache	491,868	49.19%	488,313	48.83%	-0.36
nginx	214,310	21.43%	216,433	21.64%	0.21
Microsoft	121,210	12.12%	121,274	12.13%	0.01
Google	24,293	2.43%	24,170	2.42%	-0.01

Example of usage estimation: Netcraft web server survey (May 2015). This survey is performed by querying web servers (in this case, the top million busiest sites) for their identification string.

Ecosystem community

The ecosystemm community is a kind of mega-community, including all the communities relevant to the project under evaluation. All the above comments for developer, contributing and user commities apply, since all of them are representer in this ecosystem community. But at the same time, there are more overlappings, since many developers, contributors or

users may be in many of the communities in the ecosystem.

The ecosystem community is difficult to study because it is usually large, and is spread through many different infrastructures. In fact, the first problem to address is to find out all the projects that form a part of it, since interrelations between components can be complex. However, this mega-community is very important for the long term sustainability of the project under evaluation. Usually, resources for most projects, including developers and users, come primarily from their ecosystem communities. Some of them can work as attractors, bringing new resources to the ecosystem community from the outside world. Of course, identifying those projects that create and nurture an ecosystem community is very important to understand long term trends in FOSS techologies.

Just as an example, when studying the ecosystem for a GNOME application, all the GNOME ecosystem has to be taken into account, because it will be relevant for the future of the project. For example, if basic GNOME libraries stop evolving, it is difficult that the application keep pace with puture needs.

From another point of view, the definition of the ecosystem, and therefore of the ecosystem community, is something that depend on the objectives of the evaluation. The ecosystem can be defined only for those projects with strong ties and great dependency, or for all those on which the project depends to some extent. The former case is the most usual, since dependencies and relationships are easy to perceive. But the latter can lead to important conclussions, such as when many projects discoverd that they were hit by the Heartbleed bug, deep in a software produced by a handful of developers, which was included in many, many very popular programs.

Common techniques

In addition to the analysis of the sources of information detailed in the previous chapter, there are some techniques that can be applied, mainly to find out about the non-developer communities.

Surveys and interviews

Surveys and interviews can help to obtain information that cannot be inferred from the project repositories. This can be because the needed information is not available in such repositories, or because the target population is not using them.

An example of the first case is the effort devoted to develop software. It an evaluation parameter is the effor that a project puts into development and maintenance of the software, for example in aggregated person-months, that is not an information that can be reliablily extracted from repositories. But it can be obtanined if developers answer a simple survey.

An example of the second one is user satisfaction. In some cases, a web-based survey where users rank a software with a five-stats schema may be enough. The main trouble here is to reach a sample of users representative enough of the user population for the survey to be statistically significant.

Some projects do this kind of surveys on a regular basis, publishing their results. The main trouble with such surveys is that they are not comparable from project to project, which means that usually new surveys are needed, which is a time consuming and expensive procedure. It would be very coonvenient for evaluators that FOSS projects agreed on some common questions to developers and users, and standarized some surveys that could be used for the most common evaluation scenarios.

Interviews to experts on a project are also a good source of data. Open or directed interviews provide mainly qualitative information about a community, which can enrich or complement quantitative information obtained by other means.

Traces in collaboration systems

All communication systems which can be analyzed can provide useful information for some kinds of community evaluation. It was already mentioned how ACS and SCS, as defined by the project, can be valuable sources of data. But other external communication channels can also help. For example, mentions in Twitter or other social networks, and even sentiment

analysis on those mentions can say a lot about how the project and its community are perceived.

Evaluation of documentation and third party studies

Documentation can also provide details about the community. In some cases, the very availability of docuemtation, or certain kinds of documentation, is a relevant fact for users. In some others, user-generate documentation can also e a proxy for estimating some parameters, such as user involvement in the project.

The documentation generated by the project itself in some cases describes some details about the community. For example, developer documentation usually describes the repositories and communication channels used by the project, which can be the source for empirical analysis of those repositories. Documentation can be useful as well for determining project policies with respect to participation, structure and decission making in the community.

Evaluating activity

One of the aspects of a community that are most usually evaluated is activity. In this context, evaluating activity refers to finding signs and traces of activity performed to make the project advance towards its goals. Activity can be of different kinds, such as:

- committing patches to the source code management system
- reporting, commenting or fixing bugs in the issue tracking system
- submitting patches or reviewing them in the code review system
- sending messages to mailing lists or synchronous communication systems

Not all the activity is observable, and suitable for evaluation. For example, in mailing lists it is easy to know when a message is sent: it is enough to explore the list archive. But it is difficult to know who received that message (the list recipients is usually not public), and almost impossible to know who read it (reading is a private activity performed in your own mailer program).

But the observable activity is usually good enough to know about the heartbeat of the project, about how many people is active in different roles, and about the general trends.

There are several analysis of activity suitable for evaluation. The most common are:

- Parameters reflecting activity for a certain period. For example, number of changes to the source code for the whole history of the project, or number of messages sent to mailing lists of the project during a certain week.
- People active for a certain period. For example, people fixing bugs during a release period, or people providing code review advise during the last month.
- Evolution of any of them. For example, new tickets per month for the whole live of the project, or messages sent to IRC channels per week.
- Trends for any of them. For example, increase (or decrease) in number of messages posted in the forums for the project from December 2014 to January 2015.

Number of commits per month for Puppet, as shown by Grimoire Dashboard, circa June 2015. Trends for the last year, month and week are shown as well.

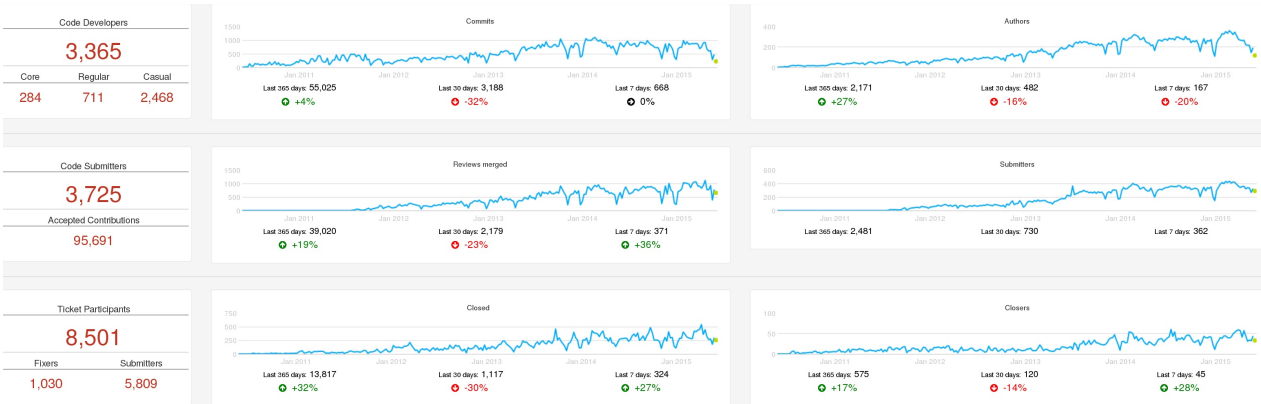
Parameters by themselves only provide a first hint. Saying that a project is performing 2,303 commits in one month is a first indicator about how active is the project, but doesn't provide too much information. Putting it into context starts to make things more interesting. For example, comparing two projects with similar functionality, but one of them committing five times the other, is a first step towards comparing their activities.

However, commit patterns may be very different from project to project, and a simple comparison may be misleading. For example, one of them may be committing a very single change proposal, just to improve them later. Another one, meanwhile, may be following an stringent code review process, committing only after several iterations that improved change proposals. The first pattern will produce much more commits than the second. The same can be said for other parameters.

Comparisons within the same project are usually much more interesting and fair. If the project didn't change policies nor patterns during the last two months, comparing activity parameters will provide a good idea of trends. Comparisons over larger periods of time will allow for detecting the impact of changes in policies, tools or patterns. For excample, changes of the source code management system, or the introduction of code review, or policies on closing old tickets are reflected in the long-term charts about activity. And of course, growth, stagnation or decrease in activity can be clearly perceived over time.

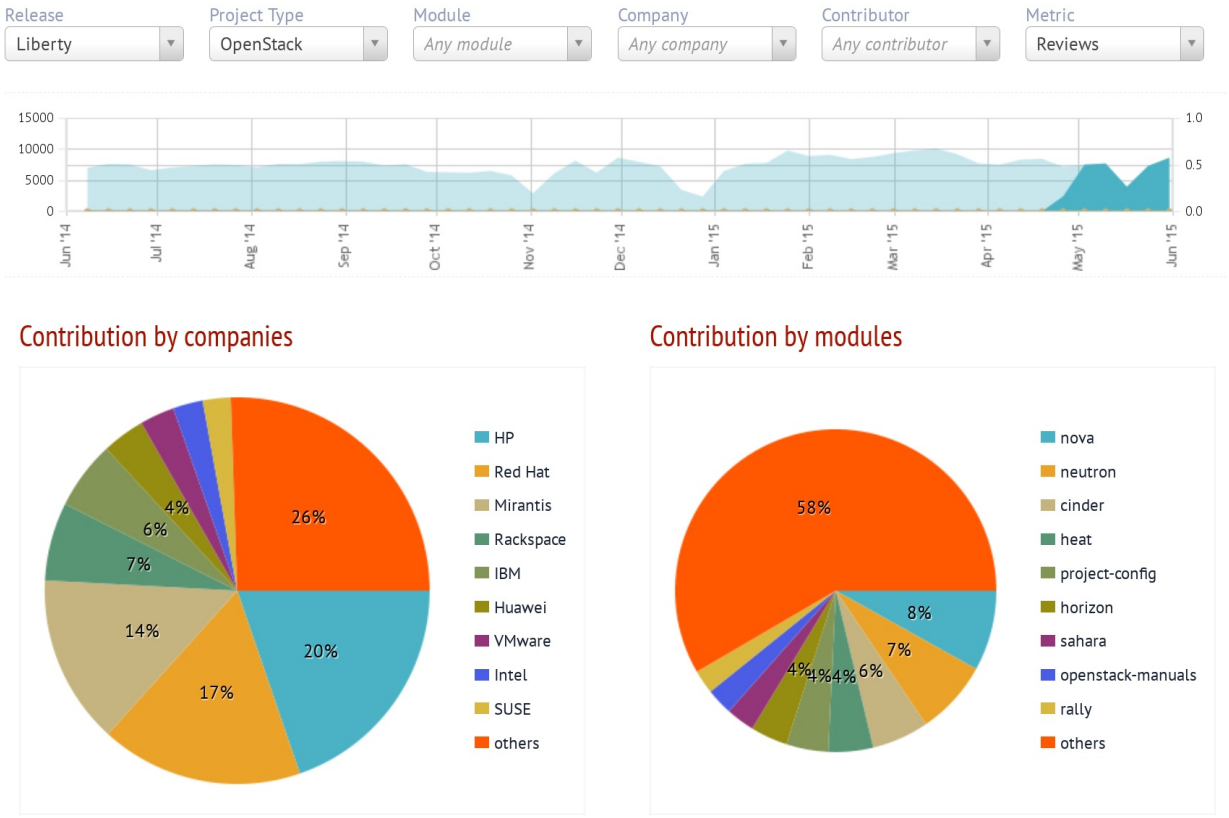
In addition to the raw parameters on activity, the parameters related to persons performing that activity are also relevant. They allow for a first characterization of the active community in several areas. An exponential growth in code authors, or a steady decline in bug fixers will certainly be interesting subjects of further analysis.

Several of these parameters together show a multifaceted view of the project. As an example, next figures show a summary of activity of the same project, OpenStack, as shown in three different dashboards: Grimoire, Stackalitics, and Open Hub.



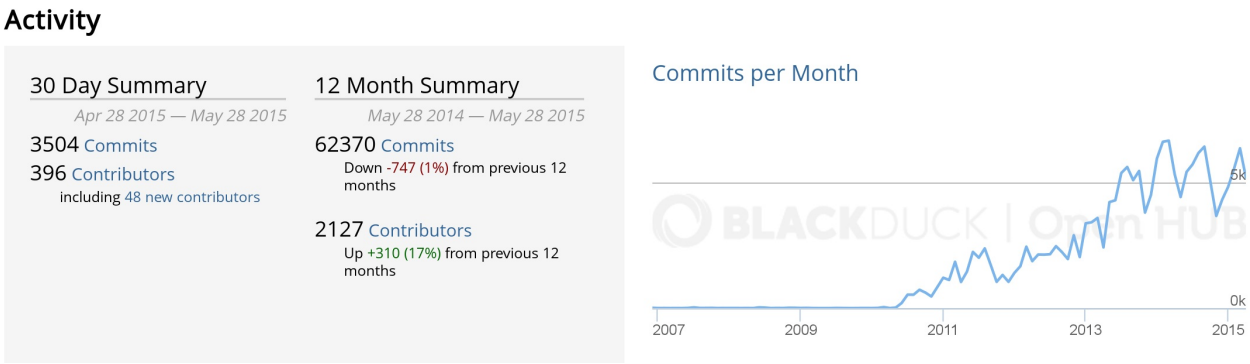
Activity in OpenStack: summary of activity in serveral repositories over time, as shown by Grimoire Dashboard, circa June 2015.

The Grimoire Dashboard shows activity in each kind of repository, which allows for easy comparison, while at the same time the general trends of activity in the project are visible. It shows some metrics about the people active in different roles.



Activity in OpenStack: summary of code merges over time, and split by company and module, as shown by Stackalytics, circa June 2015.

Stackalytics focuses on changes merged, although it shows other activity as well. The summary includes activity by company and by module.



Activity in OpenStack: summary of activity over time, as shown by Open Hub, circa June 2015.

Open Hub shows a chart with the history of the activity, and some factoids about it, with a focus on activity in the source code management system.

Activity in source code management

Activity in source code management reflects how the project is producing changes for the products they build. Source code management stores "commits", each of them being a change (or "patch") to the source code. Each change is different in nature, size, complexity, etc., which makes it difficult to compare individual changes. However, when we consider large collections of changes, trends become apparent.

In particular, when a project has pre-merge code review, this is a metric very difficult to cheat. If a developer tries to split a commit in several, for increasing the personal commit count, code reviewers would complain. Usually, the very possibility that this happens is enough for discouraging developers who could be tempted to split commits.

However, it is important to realize that a single commit can be very important for the project, and be the result of a great effort. This is speciall the case when we're looking at the numbers of a specific persons, instead of aggregate numbers. Therefore, commit counts should not be used as a basis of rewarding systems, for example.

When used properly, commit counts can be a good estimator for total effort. Recent studies show how above a certain number of commits per month, it is very likely that the developer works full time in the project. Numbers below that threshold can be prorated to estimate a fraction of full-time effort. This threshold is dependent on the project, but is usually around 10-15 commits per month, in complex systems with code review and continuous integration.

Activity in code review and ticketing systems

In CRS, activity is usually measured as the number of completed (or started) review processes per period. In the case of core reviews where different versions of the proposed change can be submitted, the total number of versions submitted for review is also significant.

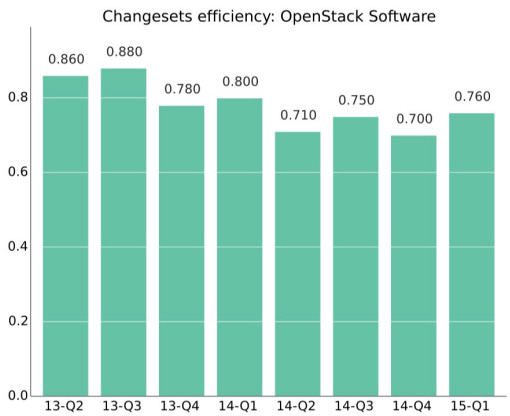
It is important to notice that when pre-merge code review is in place, the number of code review processes ending in a merge is equal to the number of commits. However, the metrics may show some differences depending on the dates considered. For reviews, either the date of the starting or finishing of the process may be considered. For commits, the date of authorship, or the date of commit (usually the date of the merge).

The number of review processes are good indicators of the volume of the review process. Even when not all reviews require the same effort or have the same complexity, aggregated numbers tell a lot about the resources needed for the code review processes.

The number of review processes ending in merge or abandon is also important. In fact, the difference between new code reviews, and merged plus abandoned code reviews for a period of time is an indicator of whether the project is coping with all proposed changes, or not.

In the case of ITS, activity is measured either in terms of open tickets, or closed tickets. In fact, both are important indicators, and their difference shows whether the project is coping well with new tickets. The number of state changes, and the number of comments to tickets, are good indicators of activity as well.

For both systems, efficiency in closing is an important factor. It is the ratio of new tickets or review processes to closed tickets or finished (merged or abandoned) review processes. When this number is larger than one, that means that the project is opening more than closing, which is a problem in the long term. If it is lower than one, the project is "recovering old workload", by closing more issues than they are being opened.



Period	(Aband. and Merg.)/Subm.
13-Q2	0.86
13-Q3	0.88
13-Q4	0.78
14-Q1	0.8
14-Q2	0.71
14-Q3	0.75
14-Q4	0.7
15-Q1	0.76

Example of efficiency: new versus closed review processes in OpenStack, 2015, first quarter.

Activity in communication systems

Activity in communication systems is usually measured in messages. But messages in different kinds of communication systems may be very different. For example, it is usually much longer to write an email message than to write a one line comment in an IRC channel. This means that metrics from one system cannot be compared with metrics from another one, even if they are similar.

But nummbers for different points in time can be compared, whcih allows for detecting trends, and even estimate the amount of effort needed to track all communication channels for a project. Since core developers may need to track all of them, this is a first estimation of how the communication cost is hitting producitivity.

Active persons

In addition to activity itself, knowing who are the persons causing that activity is very important when evaluating a community. Active persons in a project can be measured in several repositories:

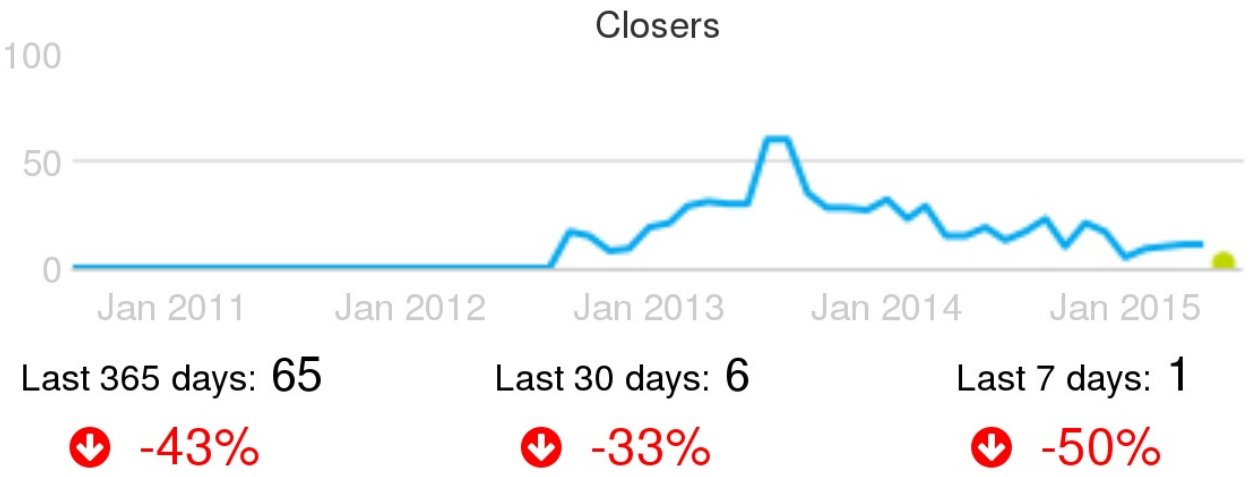
- SCM: In modern SCM systems, such as git, authors and committers can be counted separately. Authors are persons authoring changes to source code. Committers are persons commiting those changes to the repository. Authors can be also committers, if they have commit rights. But if they don't, usually they send their changes to committers, who are those merging them into the repository. However, sites such as GitHub ot GitLab make things more complex, since when the changes are contributed via the web interface, as pull requests, authors are considered as committers even if they don't have commit access to the repository.

Example of active persons: Active authors per month in the Puppet project, circa June 2015.

- CRS: Code review systems allow for the identification of several populations of active persons: change proposers (initiators of review processes), reviewers, rejecters (reviewers rejecting changes, asking for new versions), accepters (reviewers accepting changes), and abandoners (submitters abandoning a proposed change).

Example of active change submitters: Active summitters of proposed changes for code review, per month, in the Wikimedia projects, circa June 2015.

- ITS: The main active populations to track are ticket openers and closers, and people participating by changing states or commenting. Openers are persons contributing with new bug reports or feature requests. In many cases they are not developers, but people hit by a bug, or needed a new feature, and engaged enough with the project to devote some time to file the new ticket. People changing stated and closing tickets are very likely developers in the project. People commenting are either developers, or non-developers (maybe the one who submitted the ticket) collaborating in the bug-fixing or feature-implementation processes.



Example of active ticket closers: Active people closing tickets, per month, in the CloudStack project, circa June 2015.

- Communication channels: In most communcation channels, be them synchronous or asynchronous, active persons that can be measured are senders (or posters). In most of these systems the information about who is actually reading, or even receiving, those messages sent is not easy to obtain, or does not exist. In the systems that allow for it, the number of persons actively answering or following-up to a message is interesting as well.

Example of active senders in IRC channels: Active people sending messages in IRC channels, per week, in the OpenStack project, circa June 2015.

Of course, in addition to the raw numbers of active persons, the ratio of any parameter showing activity to the numbers of the group causing that activity is specially relevant. For example, the ratio of commits to authors of those commits over time shows quickly if the number of commits per author are growing or not.

Merging identities

For most of the studies based on tracking persons, it is important to merge all identities that a single person may have in the repositories in a single merged identity. That can be done at four levels:

- The repository level. That consists of merging all the identities of the same person in a given repository. For example, meging all your identities in a certain git repository. This is useful, for example, to count the real number of people working in that repository.
- The repository kind level. In this case, all the identities of the same person, across all the repositories of the same kind for a certain project, will be merged. This is useful for studies for all repositories of the same kind. For example, for counting the total number of developers contributing to source code, and thus to git repositories, of a certain project.
- The project level. In this case, all identities of the same person, across all the repositories of any kind of the same project will be merged. This is needed to know about persons at the project level, such as for evaluating the population of contributors of a project across all its repositories of any kind.
- The global level. All identities for a certain person, in any repository of any analyzed project, is merged into a single merged identity. This is useful when tracking people working in several projects. For example, for finding developers working both in project X and project Y.

In some cases, the projects keep some information to track the multiple identities of developers. But in most cases, you can only relay on heuristics and in manual comparison and merging of identities. There are many heuristics that can be used, but they can be tricky depending on the circumstances, over- or underperforming in specific projects. One example is comparison of email addresses when the complete name string matches. To illustrate this heuristic, let's use the following email addresses:

```
Jesus M. Gonzalez-Barahona <jgb@bitergia.com>
Jesus M. Gonzalez-Barahona <jgb@gsyc.es>
```

A heuristics finding exact matches in names would correctly merge these two indentities. But now consider the same heuristics applied to these two addesses:

```
John Smith <john@somecompany.com>
```

```
John Smith <js@someothercompany.com>
```

Given that John Smith is a very common name, it could perfectly be the case, specially in a large community, that those identities correspond to different persons, and therefore shouldn't be merged.

In general, this happens with any heuristics you may find out. That is the reason why usually the merging of identities is really a mix of applying heuristics and manual check of the identities. Of course, when the project itself is involved, and the real persons whose identities are merged collaborate, the process can be reviewed by them. This is the better way of ensuring accuracy.

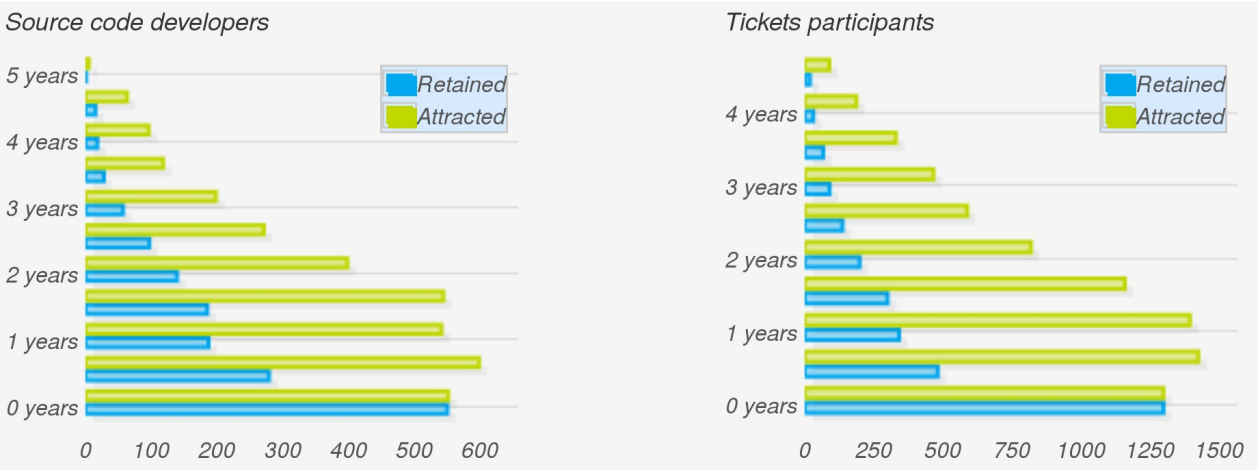
As was commented at the beginning of this section, the most accurate this merging process is, the better estimation of parameters that depend on identfying persons, and not identities.

Aging

Of the many aspects to explore in the community of a FOSS project, turnover and age structure are some of the more important. Turnover shows how people are entering and leaving the community. It tells how attractive is the community, and how it retains people once they join. Age structure, understanding age as "time in the project" shows for how long members have joined it. It tells how many people are available in different stages of experience, from old-timers to newbies. Together, both can be used to estimate engagement, to predict the future structure and size of the community, and to detect early potential problems that could prevent a healthy growth.

The community aging chart

Both turnover and age structure can be estimated from data in software development repositories. A single chart can be used to visualize turnover and age structure data obtained from these repositories: the community aging chart. This chart resembles to some extent the population pyramid used to learn about the age of populations. It represents the "age" of developers in the project, in a way that provides insight on its structure.



Example of aging charts: Community aging charts for authors of code, as found in git repositories of the OpenStack project (left) and ticket participants, as found in the OpenStack Launchpad (right), circa June 2015. Each pair of blue and green bars corresponds to a generation of six months.

In the aging chart, each pair of two horizontal bars shows how a "generation" is behaving. The Y axis represents how old is each generations, with younger ones at the bottom. For each generation, the green bar (attraction) represents the number of people that joined it. In other words, how many people were attracted to the community during the corresponding period (say, first semester of 2010). Meanwhile, the blue bar (retention) represents how many people in that generation are still active in the community. In other words, how many of those that were attracted are still retained.

One chart, many views

The aging chart shows many different aspects of the community. Let's review some of them.

The ratio of the pair of bars for each generation is its retention ratio. For the newest generation, it is 100%, since people recently entering the community are still considered to be active (but that depends on the inactivity period, see below). A ratio of 50% means that half the people in the generation are still retained. Comparing the length of each pair of bars, we can quickly learn about which generations were most successfully retained, and which ones mostly abandoned the project.

The evolution of green bars tells us about the evolution of attraction over time. Most successful projects start with low attraction, but at some point they starts to become very attractive, and the bars grow very quickly. When a project enters maturity, usually its attraction becomes more stable, and can even start to decline, with the project being still extremely successful, just because it is no longer "sexy enough" for potential newbies.

The evolution of blue bars tells us about the current age structure of the community. If bars in the top are large, but those in the bottom are small, the community is retaining early generations very well, but having difficulties with retain new blood. On the contrary, if bars in the top are small while those in the bottom are large, newcomers are staying, while experienced people already left. Blue bars can only be as large as green bars (you cannot retain more people, for a certain generation, that those that you attracted to it). Therefore, "large" and "small" for blue bars is always relative to green bars.

Different charts for different information

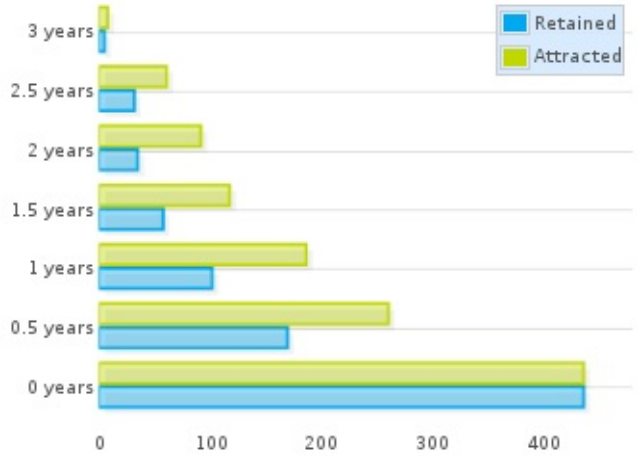
To build the community aging chart, three parameters have to be considered: generation period, inactivity period, and snapshot date:

- The generation period defines how long generations are: that is the granularity of the chart. It is usually one year, or maybe six months for younger communities. People in the community is going to be charted according to their generation, using this granularity.
- The inactivity period is how long we wait before considering that somebody left the community. We don't know if persons really left the community: maybe they are on vacation, or on a medical leave. So, we have to estimate that "if somebody was not active during the last m months, we consider that person as a departure from the community". That m is the inactivity period, which is usually equal to the generation period, but could be different.
- The snapshot date is when we consider as "now". That is, we can calculate the the community aging chart for today, but also for any time in the past. In fact, comparisons of charts for different snapshot dates say a lot about the evolution of the attraction and retention of the project over time.

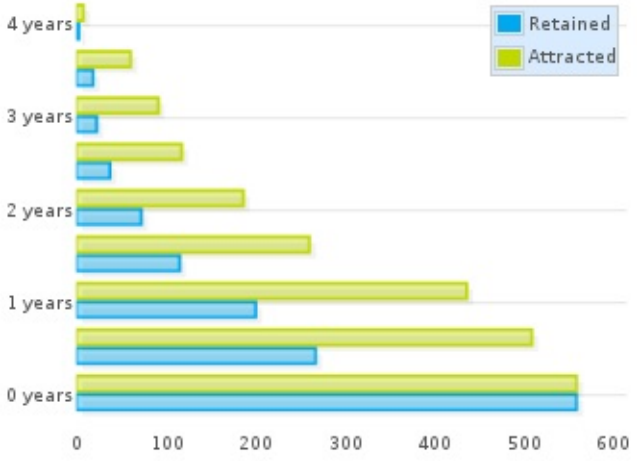
Comparing a community aging chart from the past with the current one let us compare the potential we had some time ago with the reality now. In most development communities, people inactive for a long period are very unlikely to show up again. That means that the sum of the retention bars in the chart snapshoted two years ago are the maximum population that the community is going to have two years later, save the generations entering during these two years.

One example and some comments

As an example, we can compare the aging chart for OpenStack in July 2014 with the same chart for July 2013. Both charts show six-month generations and use six month inactivity period as well. Obviously enough, the latter includes two bars less, those corresponding to the two last generations, who still had not joined the project in July 2013. Green bars corresponding to generations more than one year old in July 2014 are exactly the same as those in the chart for July 2014, only shifted by one year. If a generation attracted a number of people, that does not depend on when we set the snapshot.



Aging chart for authors in OpenStack git repositories in July 2013



Aging chart for authors in OpenStack git repositories in July 2014

If we focus now on the one-year-old generation for July 2013 (the third one, counting bottom-up), we can see how it is represented one year later. From a total of about 190 persons attracted, about 100 were still retained in July 2013. That means that in July 2014 we could expect at most 100 persons still retained in that generation. Now, fast forward to the future: in the chart for July 2014, about 70 persons are still retained from the (now) two-years-old generation. In other words, the project lost a much higher share of the generation during the first year than during the second one, even if we consider the latter case relative to those that still were in the project in July 2013.

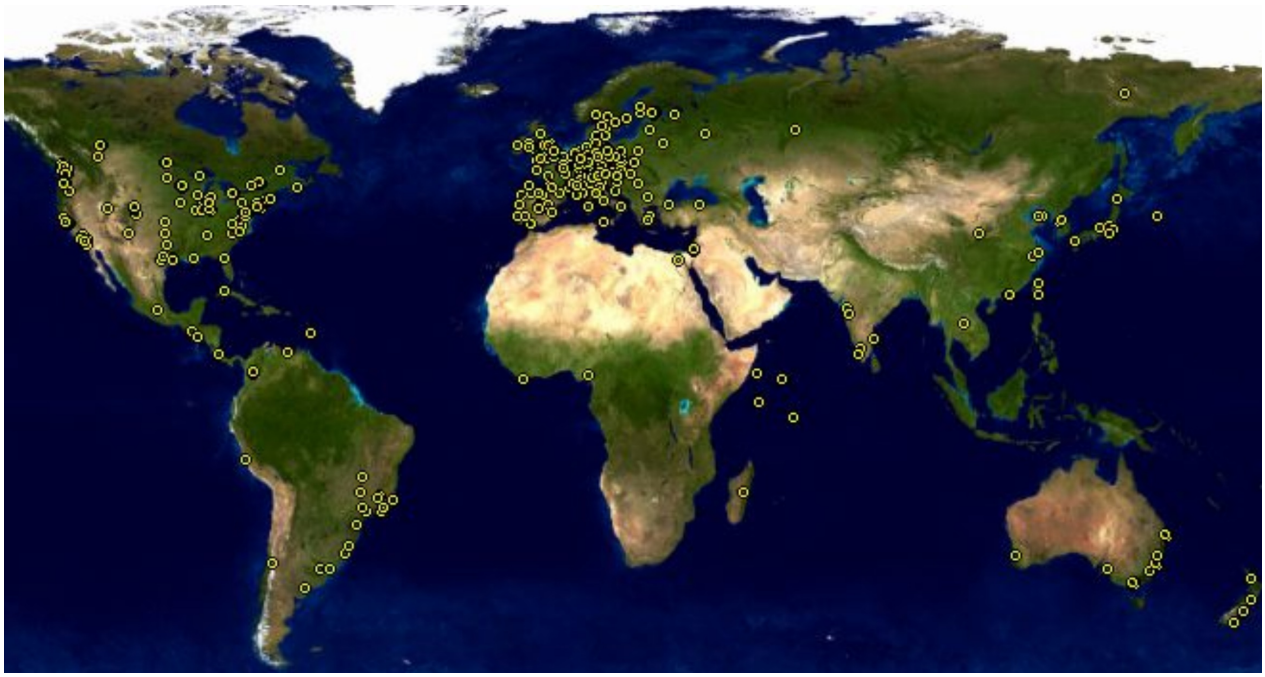
This is a very common fact found in most projects: they lose a large fraction of attracted persons during the first year, but are more likely to retain them after that point. This depends as well on the policies of the project, and how you enter the community. Retention ratio for the first year usually reflects more than anything how difficult it is to enter the community. The more difficult it is to get in, usually the most engaged people are, and the less likely to leave quickly. But the more difficult it is to get in, the less people in the newer generation are going to be attracted. Therefore, projects with different entry barriers can attract very different quantities of people, but maybe the retained people after one year is very similar. Of course, volunteers and hired developers have different entry / leave patterns too, that influence these ratios.

We can also read the future a bit. Assuming the current retention rates per generation, we could estimate the size of the retention bars for the future, and from it the total size of the community with a certain experience in the project. For example, all those staying more than two years in the project in one year from now, are in the blue bars corresponding to generations currently older than one year. This allows for the prediction of shortages of developers, or of experienced developers, for example.

In fact, any policy oriented to improve attraction or retention of people can be easily tracked with these aging charts, by defining the ideal charts for the future, and then comparing with the actual ones.

Time zones and other geographical information

Knowing about the geographical location of the members of the community is difficult. In some projects, when people register in the project, they can enter some geographical information. That can be the country or city of residence, or even their coordinates. As an example, see below the map of Debian developers (well, in fact, the map of some of the Debian developers, who specified their location).



Example of geographical information for a community: Map showing Debian developers location, for those developers who registered their coordinates.

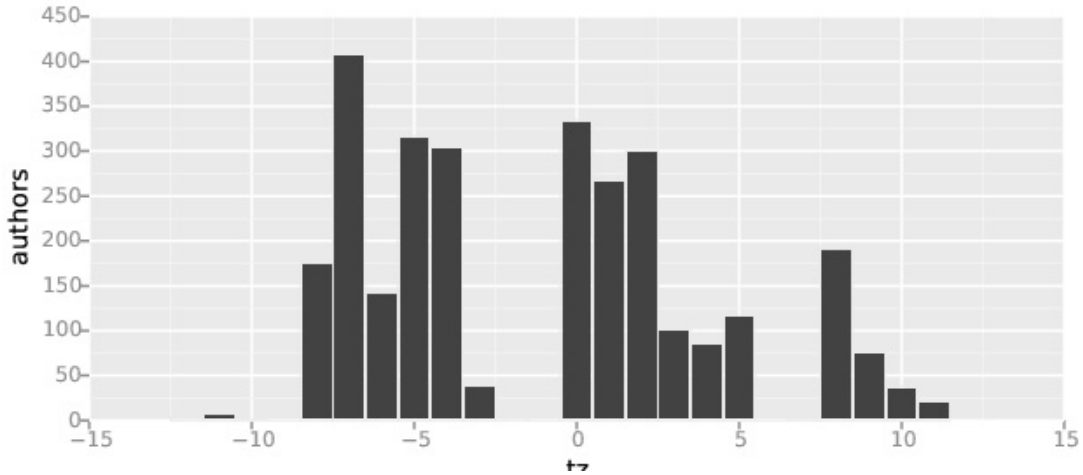
But having this level of information is unusual, and usually incomplete, since it covers only community members who want to fill in this information.

When the project records IP addresses accesing its infrastructure, they can do IP geo-location on them. Since different types of access can be tracked (access to the development repositories, to the downlowad area, to the forums, etc), those projects can track with detail the location of different actors in the community. But again, this is unusual. Most projects don't have these capabilities, or don't want to put this tracking in place.

For projects willing to have some information about the geographic location of their community, but not using the former techniqueus, there is still something to be done: time zone analysis.

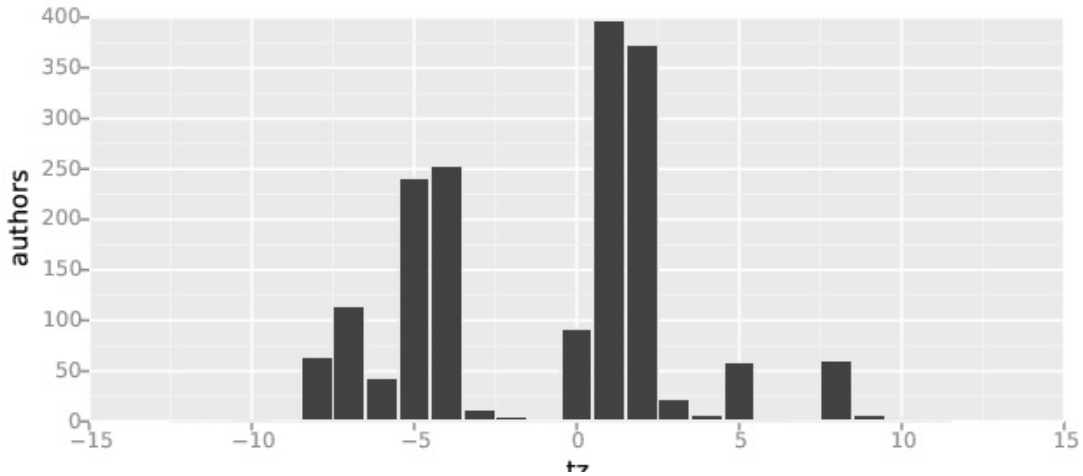
The main advantage about time zone analysis is that it uses geographic information that individuals provide when using some repositories. Well, as is usual in these cases, it is not exactly individuals, but the software they use. The most two widespread cases are git and mailers:

- git clients include the local date when commits are created. When those commits are acccepted, merged in other repositories, etc. the time (including the time zone tag) are not altered in most cases. Please, note that we said "in most cases": some actions on commits will alter their time, usually setting the time zone tag to that of the person performing that action. But still, the information is reliable enough to know about the time zones for commit authors.



Example of timezone analysis: Number of git authors per time zone, repositories for the OpenStack project, during 2014.

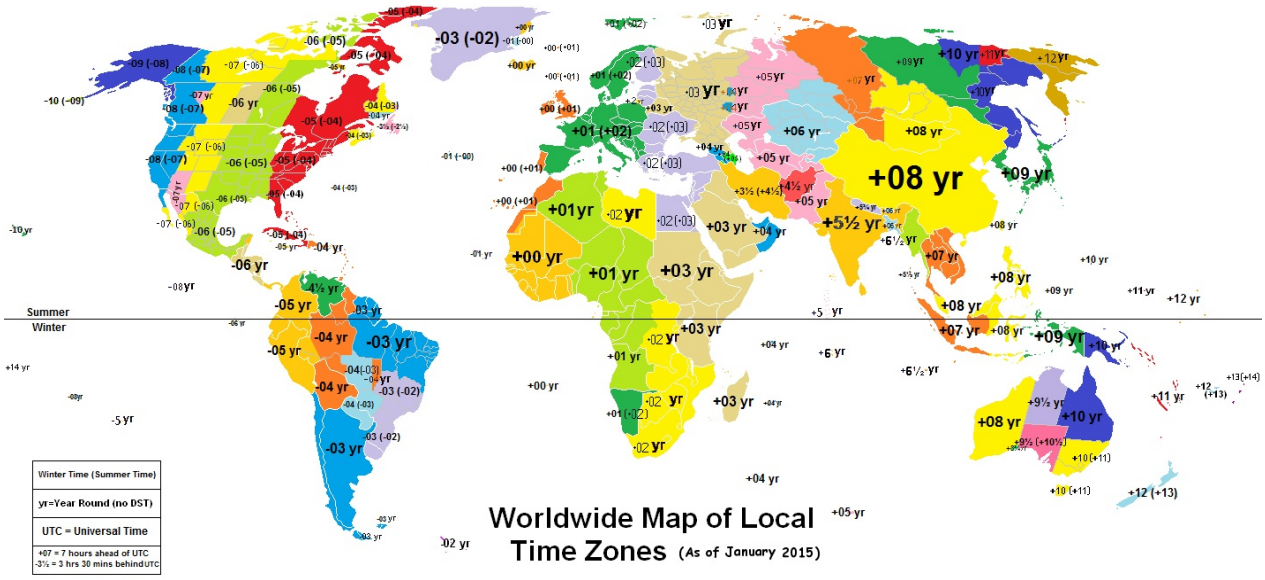
- Mailers include the local time, including time zone tags, in messages sent. In many cases, mailing list software keep this time. When that is the case, the analysis of mailing list repositories permit the identification of time zone for senders.



Example of timezone analysis: Number of messages per time zone, sent to Eclipse mailing lists during 2014

In both cases, it is important to notice that there are at least three sources of trouble with this time zone analysis:

- Bots that perform commits or send messages. They can have their local time zone set to whatever is convenient for the machine where they reside. Since in some projects bots can do a lot of these actions, the number of messages or commits per time zone can be greatly affected because of this.
- People setting their time zone to something else than their time zone of residence. For example, frequent worldwide travellers, or persons with intense interactions with people in other timezones, may have their time zone set to UTC+0 (universal time, formerly Greenwich time). This means that the time zone corresponding to UTC+0 can be overrepresented because of this fact.
- Many countries are in fact in two timezones, since they change time in Summer (Summer savings time).



Map of world time zones. Original: Worldwide Time Zones (including DST), by Phoenix B 1of3, Creative Commons Attribution-Share Alike 3.0 Unported

Due to the distribution of population on the Earth, timezone analysis provides a very high level glimpse of the geographical distribution of the community. There is no way of telling European from African contributors, for example, since they are in the same timezones. But you can roughly identify persons from several regions (the list is not exact, look at the map for details and a more accurate description):

- UTC+12: New Zealand
- UTC+10, UTC+11: Australia
- UTC+9: Japan, Korea.
- UTC+7, UTC+8: China, Eastern Russia, Indochina.
- UTC+6: India (in fact, it is UTC+5:30).
- UTC+3 to UTC+5: Western Russia, East Africa, Middle East.
- UTC+0 to UTC+2: Western and Central Europe, West Africa.
- UTC-2, UTC-3: Brazil, Argentina, Chile.
- UTC-4 to UTC-6: North America Central and East Coast (US, Canada, Mexico), Central America, South America West Coast.
- UTC-8, UTC-7: North America West Coast (US, Canada).

For some uses, this split in regions is enough. For example, in the above chart about OpenStack git auhtors it is clear how most of the developers are from North America and Western Europe, with some participations of the Far East and other regions. But the distribution of the Eclipse mail senders is even more centered in Western Europe, with a large participation from North America, a only some presence from the rest of the world.

This kind of study is enough to assess the results of policies for increasing geographical participation, or to know where developers come from to decide on a meeting location.

Time of collaboration

In global communities, knowing when people is working is important for many issues. For example, for deciding on coordination synchrounous distance meetings, for estimating to which extent the work on issues may be continuous because there are people working at any time, or to know about work timing patterns in the project.

People work at different times of the day due to being in different time zones, but also due to different working habits. For example, many people working for companies tend to follow the office hours schedule, from 8 to 5 or similar. But volunteers tend to work on their spare time, that is in the evenings and during the night. The same can be said about day of the week:

office workers tend to be active from Monday to Friday, while volunteers tend to work also during weekends. There are similar differences in vacation periods too.

These work patterns can be estimated from dates in almost any of the repositories that we can use as data source, since activity is usually tagged with a time. This allows for very detailed analysis of when people perform that activity.

Affiliation

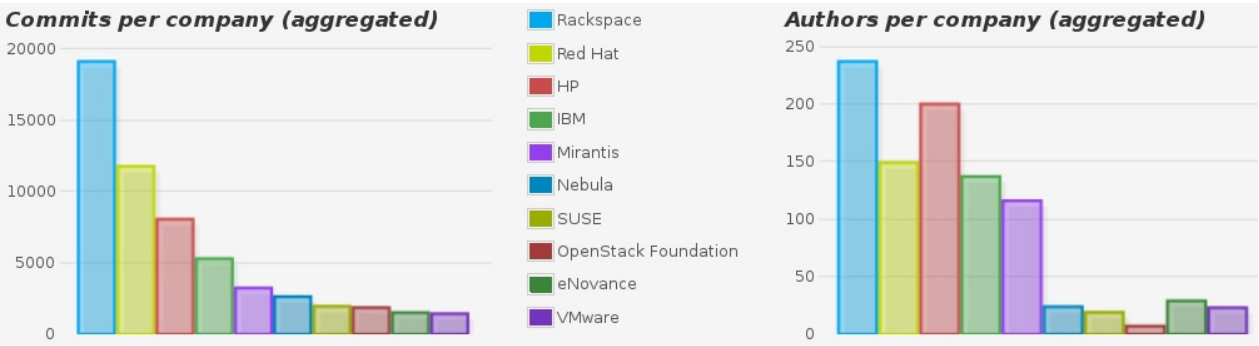
In FOSS communities, many develpers are not working as volunteers, but as paid workers. In this case, it may be important to know for which organization each of these developers is working. Knowing it allows for several kinds of higher level studies, such as diversity in organizations contributing to a project, or how each organziation collaborates with others. In general, any study that can be performed at the person level, can be performed at the organization level just by aggregating the activity of all their employees.

The basis of these analysis is therefore identifying to which organziation is affiliated each developer, if any, and during which time period. In some cases, this information is maintained to some extent by the projects themselves. For example, the Eclipse Foundation and the OpenStack Foundation maintain detailed affiliation information for all their committers. But even in those cases, there are other people, such as casual posters to mailing lists, that cannot be identified in a coompulsory way, and who have little motivation to collaborate in any affiliation tracking schema maintained by the projects.

There are other techniques that may work in a certain fraction of the cases to track affiliations. It is important to notice how this problem is related to the merging of identities, which was mentioned earlier in this chapter. Assumming that the merging of identities is already done and accurate, some of the techniques for finding affiliation information are:

- Using domains in email addresses to identify companies. Not all domains are useful for this. For example, @gmail.com or @hotmail.com refer to the mailing system that the person is using, and has nothing to do with the organization for which they are working. But many other addresses, such as @redhat.com, @ibm.com or @hp.com can be easily tracked to Red Hat, IBM or HP. An specific case when this technique doesn't work is when there is a project policy or tradition of using project addresses. Thuius is the case, for example, of Apache, with @apache.org addresses, customarily used by Apache developers in activities related to the project. Obviously, those addresses have no use for fiinding affiliations.
- Getting listitgs from involved companies. Companies contributing to a project may maintain such listings for their own interest. If they are willing to share them, they are a precious source of information to assign affiliation. An specific case of this is when the project itself tries to track affiliation for all contributors, as was commented earlier.
- Internet searches. People can be usually found in social networks, web pages, etc. From the information found in those places, in many cases their affiliation (at least their current afiliation) can be inferred.

Once affiliation information is available, any community study can be done by organziation. For example, the next chart shows the most active companies (by changes merged) in the OpenStack project.



Example of organization analysis: Top ten organizations in OpenStack by number of commits and number of authors of commits for the whole history of the project up to November 2014.

Diversity

In FOSS projects, diversity is important. Diversity ensures that dependencies on certain people, or on certain companies, are not a risk for the future of the project. Diversity ensures that control of the project is not in the hands of a few people. Diversity helps to have a healthy community, and lowers the barriers for new people to join the community.

There are several metrics for diversity, some of them are:

- Bus factor: Number of developers it would need to lose to destroy its institutional memory and halt its progress.
- Apache Pony Factor: Diversity of a project in terms of the division of labor among committers in a project.

Bus factor

The bus factor tracks the concentration of unique knowledge on the software in specific developers. The name "bus factor" comes from the extreme scenario "What would happen to the project if a bus hits certain developers?". The first formulation of this question is attributed to Michael McLay, who in 1994 asked what would happen to Python if Guido van Rossum (its original author, and leader of the project) were hit by a bus.

If the knowledge on the project is very concentrated on a small group of developers, the trouble for the project if those developers leave is very high. On the contrary, if the knowledge is evenly spread through all the developers, even if a large number of them leave, the project can survive the shock more easily.

A maybe naive, but very practical simplification of the metric is to assume that the amount of code authored by developers is a good proxy for their knowledge on the system. Therefore, the distribution of the lines of code authored per developer in the current version of the software would allow for calculating the bus factor.

A more complete view has into account other sources of information, such as bugs fixed in certain parts of the code, participation in design and decission making, etc.

In any of those cases, some simple metrics that can be defined to get a number representing the bus factor are:

- Minumum number of developers who have a certain fraction of the knowledge. For example, minumum number of developers authoring 50% of the current version of the software.
- Maximum fraction of the knowledge that have a certain number of developers. For example, maximum fraction of source code authored by 10% of the developers.

Even when these metrices are very simple, they capture a part of the "bus factor" idea. More complex metrics, having into account the complete distribution of knowledge (or source code authorship) across developers, are possible. Factoring these metrics by the time of activity in the project is also interesting (assuming that the longer the experience in the project, the larger the knowledge).

Apache Pony Factor

The pony factor was proposed by members of the Apache Software Foundation. The first detailed explanation about it was by Daniel Gruno. It is a number that shows the diversity of a project in terms of the diversity of labor among its developers. From this point of view it is a concrete implementation of the bus factor.

The pony factor is defined as "the lowest number of developers whose total contribution constitutes the majority of the code base". A derived metric is the augmented pony factor, which takes into account if a developer who contributed to the code base is still active or not. In this case, contributions by inactive developers are ignored.

The "contribution" considered for calculating the pony factor is the number of commits. Therefore, we can rephrase the definition of the augmented pony factor as:

> "The augmented pony factor of a project is the lowest number of active developers whose total commit count is at least 50% of the total number of commits to the project".

This single number tries to capture how many active developers have "half of the knowledge on the project". The larger this number, the more diverse it is, and the more people should be affected by the bus factor for the project to experience trouble.

Evaluating development processes

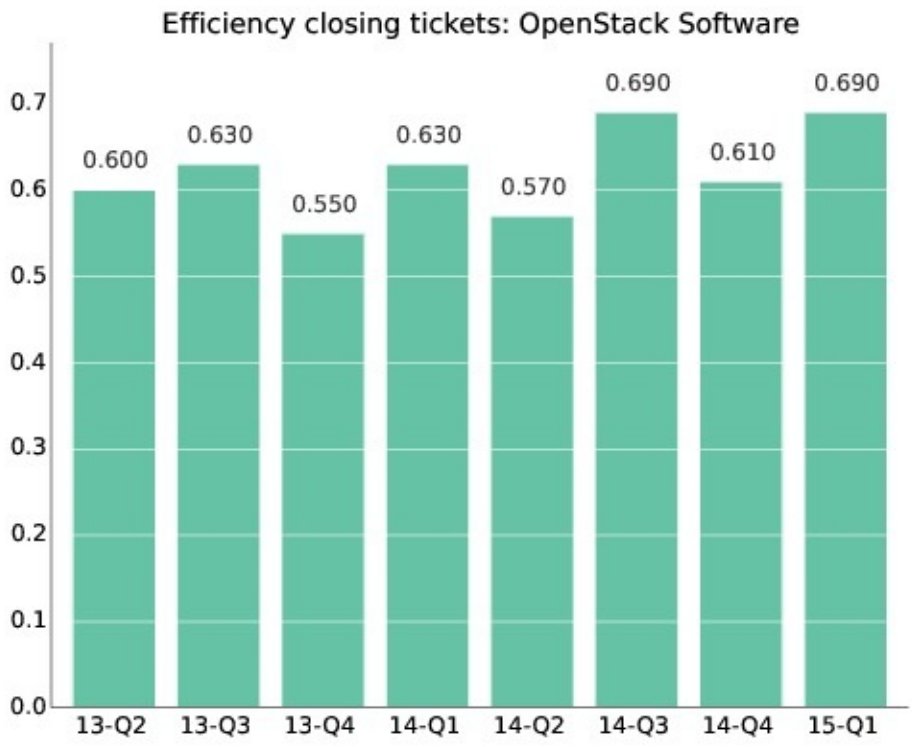
Processes are fundamental in software development. We can model as processes most actions in development projects, from implementing a new feature, to fixing a bug, or even to make a decision about how to implement a feature. Many of these processes can be tracked using information available in software development repositories.

From this point of view, FOSS projects are not different from any other software development project. But, as we discussed in the case of activity, when the project follows an open development model, most of the information needed to track processes is public. Therefore, any third party can use it to evaluate how those processes are being completed.

Performance

There are several metrics for evaluating performance in processes. Some of the most useful are:

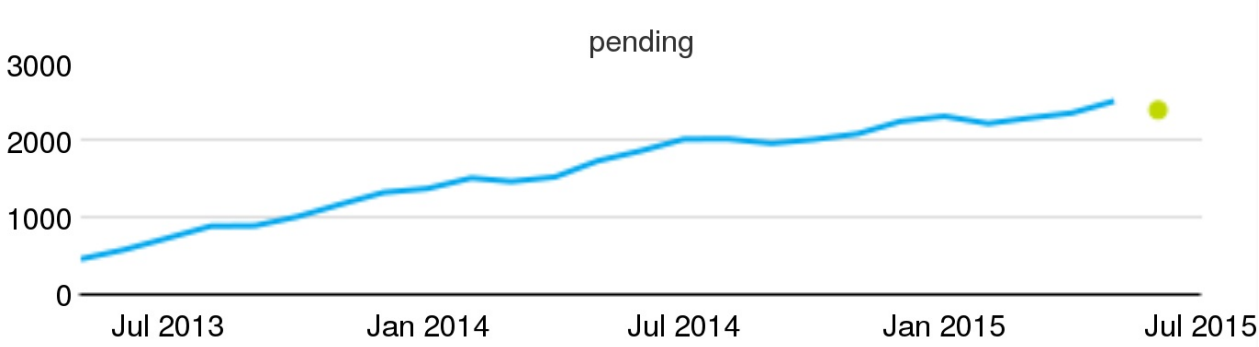
- Efficiency. Defined as the ratio between finished processes and new (started) processes for a certain period. For example, efficiency in dealing with tickets can be defined as tickets closed / tickets opened per month. Efficiency lower than one means that the project is not coping with new processes: more processes are starting that the project is finishing. Every period that happens, the backlog of open processes will increase. The evolution over time of efficiency allows to understand the long-term trends, and whether a certain efficiency is something temporary, or a permanent trend.



Example of efficiency: Ratio of closed to opened tickets per quarter for the OpenStack project. From the OpenStack Community Activity Report, January-March 2015, by Bitergia.

- Backlog of open processes. Defined as the number of processes currently open at a certain moment. For example, the backlog of code review processes still in process on a certain date. The backlog of open processes is the workload the project has to deal with, assuming no new workload appears. If the backlog increases, efficiency is lower than one, and the project is not coping with new processes. Of course, other backlogs are possible, for processes in other states

different from "open".



Example of backlog: Pending code review processes in Wikimedia projects, evolution per month, circa July 2015.

- Time to attend. Defined as the time since the moment a process is open, to the time it is first attended by the project. For example, the time to attend a certain bug report. Statistics about time to attend say about how responsive the project is, in regard of providing some early feedback to the initiator of the process. For some cases, this early action on the process may be automatic, performed by a bot. Even when this is still interesting, since in the end the opener gets some feedback, usually it is important when a human is dealing with the process for the first time.
- Time to finish. Defined as the time since the moment a process is open, to the time it is finished.

Both for time to attend and time to complete, the mean for a collection of processes is not significant, because the distribution of times is usually very skewed. Medians or quantiles can be more useful to characterize the time to attend for a collection of processes. You can remember what the median and quantiles mean:

> "A collection of processes has a median t of time-to-something if the longest of the 50% of processes with shortest time-to-something is t. In other words, 50% of the processes were shorter than t, and 50% were longest than t."
>
> "A collection of processes has a 0.95 quantil of time-to-something equal to t if the longest of the 95% of processes with shortest time-to-someting is t. In other words, 95% of the processes were shortest than t, and 5% were longest than t.

You can quickly notice that the median is the 0.5 quantil.

As an example, if the median of time-to-close tickets for a certain month was 23.34 hours, that means that 50% of the tickets were closed in less than 23.34 hours, or than 50% of the tickets took more than 23.34 hours to close.

As another example, if the 0.95 quantil of time-to-review a change is of 4.34 days, that means that 95% of the changes were reviewed in less than 4.34 days, but 5% of the changes were reviewed in more than 4.34 days.

You should notice as well that you can only measure time-to-X if X already happened. For example, time-to-close can only be measured for a ticket if it was already closed. Otherwise, you can only know that time-to-close will be longer than the time it has been open up to now, but nothing else. This is important, because it can cause counter-intuitive situations.

Assume for example a project with 100 still open tickets and 50 already closed tickets. All 50 closed tickets had a time-to-close of 2 days. All the 100 open tickets were open 60 days ago. If we measure the median of time-to-close now, it will be of 2 days, since it only applies to closed tickets. Now, the 100 still open tickets are closed during today. At the end of the day, we have 50 tickets with 2 days of time-to-close, and 100 with 60 days of time-to-close. That is, our median of time-to-close raised to 60 days, even when we closed a lot of old tickets. In other words, closing a lot of tickets raised time-to-close, which could be interpreted as a decrease in performance, when it is exactly the other way around.

To avoid these effects, there are some other metrics, such as:

- time-active. It is defined as time since the process started, only for processes which still didn't finish. In some sense, it is a complement to time-to-finish: while time-to-finish considers only finished processes, time-active provides a similar

information, but only for processes still open.

In the former example, time-active before closing the old 100 tickets would be 100 days foro all tickets to consider (those still open). After closing those 100 tickets, there are no tickets to calculate time-active, since all of them are closed.

- time-open. It is defined as time-to-close for processes already closed, and time-active (time since opened) for processes still open.

In the former example, time-open before closing the old 100 tickets would be of 2 days for the closed tickets, and of 60 days for still open tickets. That would be a median of 60 days. After closing the tickets, it would still be of 2 and 60 days, now keeping the median in 60 days, which at least remains constant.

- aggregated-time-open. It is defined as the sum of time-open for all tickets.

In the former example, before closing the tickets aggregated-time-open is of

```
50 x 2 + 100 x 60 = 6100 days
```

- aggregated-time-open-diff. We define this as the difference with the previous time-open for all tickets. This allows us to have a comparison about how aggregated-time-open varies over time, if it is measured periodically.

exactly as it will be after closing the 100 old tickets. This allows for a monotonic metric, which produces more intuitive results.

Example: regular and burst processes

Another example can illustrate a different scenario. Assume now that a project is opening 3 processes every day, and is closing them after two days (at the end of the second day). Metrics will be evaluated at the end of each day. In this case, metrics will evolve as follows:

Day	New	Finished	Open	Closed	TTF (median)	TA (median)	TO (median)	ATO	ATOD
1	3	0	3	0	N/A	1	1	3	N/A
2	3	3	3	3	2	1	1.5	9	6
3	3	3	3	6	2	1	2	15	6
4	3	3	3	9	2	1	2	21	6

Table describing an scenario of a project opening and closing processes. Each row represents a day. New: number of new processes open during the day. Finished: number of processes finished during the day. Open: number of processes still open at the end of the day (this is the backlog of still open processes). Closed: number of processes already closed at the end of the day. TTF: time-to-finish (or time-to-close) for all closed processes at the end of the day. TA: time-active, for open processes. TO: time-open, for open and closed processes, at the end of the day. ATO: aggregrated-time-open at the end of the day. ATOD: aggregated-time-open-diff at the end of the day. For simplicity, we assume that new processes start at the beginning of the day, and finished processes finish at the end of the day. All times are in days.

The median for time-to-finish quickly moves from N/A to 2 once the project starts to finish processes, and remain there since the time it takes to close tickets is constant. Time-active remains stable at 1, since at the end of the day, the processes from yesterday were closed, and only those opened when the day started are still active. Time-open reflects a bit more closely what is happening in days 1, 2, and 3, moving from 1 to 2 as more new processes start. Aggregated-time-open-diff shows the regularity of the system as well.

Now, let's see how the metrics reflect a peak in new processes. Let's assume that on day 5, in addition to the 3 new

processes that are finished in two days, 10 new processes start, and they are not finished during the following days.

Day	New	Finished	Open	Closed	TTF (median)	TA (median)	TO (median)	ATO	ATOD
5	13	3	13	12	2	1	1	37	16
6	3	3	13	15	2	2	2	53	16
7	3	3	13	18	2	3	2	69	16

On day 5, we have 13 new processes: the 3 "regular" ones, and that peak of 10 more. At the end of the day, we have closed the 3 processes that started on day 4. This means that 13 processes (all that started during the day) remain open at the end of the day. We have a total of 12 closed processes (we had 9 on day 4, plus three more we finished today). The median for time-to-finish remains at 2, since all closed processes took 2 days to finish. All processes still open were opened at the beginning of the day, therefore time-active is 1 for all of them, and the median too. The median for time-open, on the contrary, and maybe surprisingly, went down to 1. The accounting is as follows: 12 closed processes took 2 days to finish, while 13 open processes have been open for one day. Therefore, more than 50% of the process have a time-open of 1 day. Aggregated-time-open rose to 37: it was 21, plus 3 more days for the processes finished during the day, plus 13 more days for the new processes that started today. Aggregated-time-open-diff rose to 16 days (37 - 21).

On day 6, we have only three new "regular" processes, and we finish the three "regulars" that started on day 5. That means that the number of processes still open at the end of the day remains at 13, since none of the 10 "extra" processes that started on day 5 finished. The number of closed processes is rises to 15, with the 3 from day 5 that were finished. Time-to-finish remains at 2, since still all finished processes took 2 days to finish. Time-open now rises to 2, with the following accounting: 15 closed processes took 2 days to finish; 10 processes that started on day 5 have been open for 2 days; 3 processes open today were open for 1 day. In short: for 25 processes time-open is 2, for 3 it is 1. Therefore, the median is 2. With respect to time-active, for the three projects opened at the beginning of the day, it is 1, and 2 for the 10 projects opened on day 5. Therefore the median of time-active is 2. From all this accounting it is clear that aggregated-time-open is 53, and aggregated-time-open-diff is 16.

On day 7, new, finished and open remains as on day 6, since only 3 "regular" new processes start. Closed processes are increased with the 3 that are closed today. Time-to-finish remains at 2. We have 13 open processes, being 10 of them 3 days old, and three 1 day old: median is therefore 3. Time-open is calculated as follows: 18 processes took two days to close, 10 processes were open for 3 days, 3 processes were open for 1 day. Therefore, the median of time-open remains at 2. From these numbers, aggregated-time-open is 36 + 30 + 3, that is 69. Aggregated-time-open-diff is 16 once again.

From this scenario, we can learn several lessons. Time-to-finish does not reflect new processes that are still not finished. They can last for long periods, but will not be reflected in time-to-finish until they are finished. That means that time-to-finish can grow quickly when old processes are finalized, which is natural as we defined the metric, but is maybe not what some people expect when consider a longer time-to-finish as a worse metric, when comparing. Time-active, meanwhile, reflects how old processes still active age, but completely ignore (by design) how long it took to close processes.

Another lesson is that time-open can be masked by a large population of closed processes. In the example, assuming the pattern of new processes includes only "regular" processes, the median for time-open will remain at 2, even when a large amount of open tickets are unattended.

Aggregated-time-open and specially aggregated-time-open-diff reflect much better what is happening. Aggregated-time-open-diff, in particular shows how we have a continuous "lag" in dealing with processes, those 16 days of "increase" every day. That metric rose immediately when new processes entered, and will only go down when they are finished. It reflects to some extent the "amount of work still open".

Metrics for periods, metrics for snapshots

To better understand how the above metrics evolve over time, it is important to consider how exactly they are defined when we want periodic samples of them. The key is characterizing the collection of processes used to calculate the metric. In

short, some metrics are defined for collections corresponding to periods, and some others are defined for collections fulfilling some property in given snapshots (cuts in time). Depending on whether they are defined on periods or on snapshots, they behave differently.

Efficiency and time-to metrics are defined on collections of processes defined over periods. Backlog is defined on collections of processes defined on a point in time, a "snapshot" of the processes. Since snapshots are easier to understand, let's start explaining them.

When we want to analyze the evolution of the backlog over time, we define the sampling rate (say, once per week), and the starting point for the time series (say, January 1st at 00:01). What we do after that is to measure the backlog at the given points in time, by selecting the collection of open processes (if this is the backlog of open processes), and counting it:

Snapshot	Collection	Number
2015-01-01 00:01	Processes open	34
2015-01-08 00:01	Processes open	23
2015-01-15 00:01	Processes open	37
2015-01-22 00:01	Processes open	46
2015-01-29 00:01	Processes open	51

Collections based on periods are a bit trickier. If we consider for example efficiency, it is defined as the ratio of opened to closed processes. For comparing how the system is evolving over time, we need to define comparable collections of processes as time passes. But it is not useful to define those collections as "opened processes" and "closed processes" at some snapshots. The reason is clear: at a certain point in time, the collection will contain usually zero, or maybe one (if it was exactly opened or closed at that point in time) process. Which doesn't make sense for studying the evolution.

To avoid this problem, we define collections on periods. For example, all tickets opened during the first week of the year, and all tickets closed during the first week of the year.

But periods are not "points on time", and therefore we have to be careful on how we define them. If they are too short, they are going to capture too many occasional effects, and periods are going to be difficult to compare. But if they are too large, they are going to miss seasonal effects, masked by the "mean behavior".

If they are not homogeneous enough, they can be misleading. Consider for example two consecutive periods of 10 days each, but one capturing one weekend (Tuesday 7 to Friday 16), and the second one capturing two (Saturday 17 to Sunday 26). If processes are affected by lower activity during weekends, as is usually the case, the second period will appear less active when in fact maybe it is not, considering the seasonal effect of weekends. There are statistical tools that help with these effects, but a good selection of periods can minimize this effect. Days, weeks, months, quarters of years are usually good periods, when the granularity is good enough for the kind of process being analyzed.

Another effect to have into account is which processes we consider as being included in the period collection. For example, if we are measuring opened versus closed tickets for a given month, say January 2015, we can consider processes closed during that month, or processes closed at any point in time, but opened during that month. The first definition provides information on how the project is performing during a month, in terms of finished work (processes closed) versus new work (processes opened). The second definition provides information about how much work that started that month was already completed. Both are interesting, but both are very different.

Because of these reasons, it is important not only a careful selection of the period, and inclusion criteria for deciding the collections corresponding to the period. It is important as well to define carefully which processes are interesting to select, according to what is interesting to measure.

Some remarks about performance in finishing processes

In the end, when you are interested in performance in finishing processes, you should consider both the backlog and some statistics (usually the median or some quantile) of time-to-finish or time-open. The backlog will tell you about how much work is pending. The time-to-close about how long did it take to finish the processes.

All these metrics have to be considered in the context of activity. This applies specially to efficiency and backlog, but affect other metrics too. For example, in the context of a project where activity is growing quickly, it is relatively normal that efficiency is less than one, but still the project is healthy in the long term. When activity is growing quickly, usually the project is receiving more resources, and its community is growing accordingly.

But allocating new effort to deal with processes may take some time, while the growth in activity is directly linked to the opening of new processes. Therefore, it is usual that there is a certain lag between needs to close processes, and new people dealing with them. As the project stabilizes, it will start to create new processes more slowly, efficiency will increase, and backlog will start to decrease.

New features, bug fixing and code review

Among the tickets in the ITS, most issues deal usually with either feature requests or bug fixes. In fact, many projects require that the process towards any change in the code, either to fix a bug or to implement some new functionality, start with filing a ticket.

In the case of new features, the person starting the process may be a developer, and in that case we usually talk about a proposed feature. But it can be a user as well, and then it becomes a feature request. Both developers and users file report bugs by opening a bug report ticket.

From a traditional software engineering point of view, telling features from fixes may be considered important. This is because they signal two different activities: "real" development, when adding new functionality, or maintenance, when fixing bugs. In traditional environments this could be done at different stages in time, and even by different teams with primary responsibilities either in producing the next release (development) or in fixing problems with past releases (maintenance). In many modern projects, specially when they are using continuous release practices, this difference is less important.

From a practical point of view, be it interesting or not, there are practical problems for doing specific analysis for features or bugs. The main one is how to tell feature requests and proposed features from bug fixes. In some cases, the ITS provides a flag to make a difference, but even in those cases, it is only an indication that the ticket could be referring to a feature or a bug. On one side, bug fixing can evolve to feature implementation, and the other way around. On the other, it is not always easy to decide if a certain activity is improving the system by adding some missing functionality, or fixing a bug.

Consider for example the case of o form not working properly when using a touchpad. Solving it can be understood as fixing a bug (the form should work always, and it was not working in certain circumstances) or as implementing a new feature (support for touchpad). When you can match against a detailed list of requirements, this could be solved by deciding if the form was intended to work with touchpads or not, and then classify the action as a fix or as new functionality (by adding a new requirement). But most FOSS projects are not that formal, and even when they are, this is in many cases a matter of how requirements are interpreted.

To complicate matters further, in some projects there are more activities being carried on in the ITS. Those can include discussions on requirements, on the policies of the project, or requests related to the use of the development infrastructure.

And still a step beyond in making things difficult for the analyst, some projects use the ticketing system for code review. This has been a natural evolution, when specialized code review systems didn't exist. In fact, writing comments with opinions on a patch linked to a ticket, or to a commit that closed a ticket, are two examples of coded review which can be found in many projects, even when code review was not formally adopted by them. When some projects decided to adopt formal code review procedures, they started by using what they had handy: the ITS. That's how projects such as WebKit defined workflows in the ITS (Bugzilla in their case) to deal with code review. Other projects used workflows defined on Jira.

With time, specialized systems such as Gerrit emerged. Even when they are focused on code review, they still use a model

quite similar to ticketing systems, with each code review cycle being modeled as a ticket. Other systems, such as GitHub pull requests, are even closer to tickets, to the point that the interface they offer is almost the same.

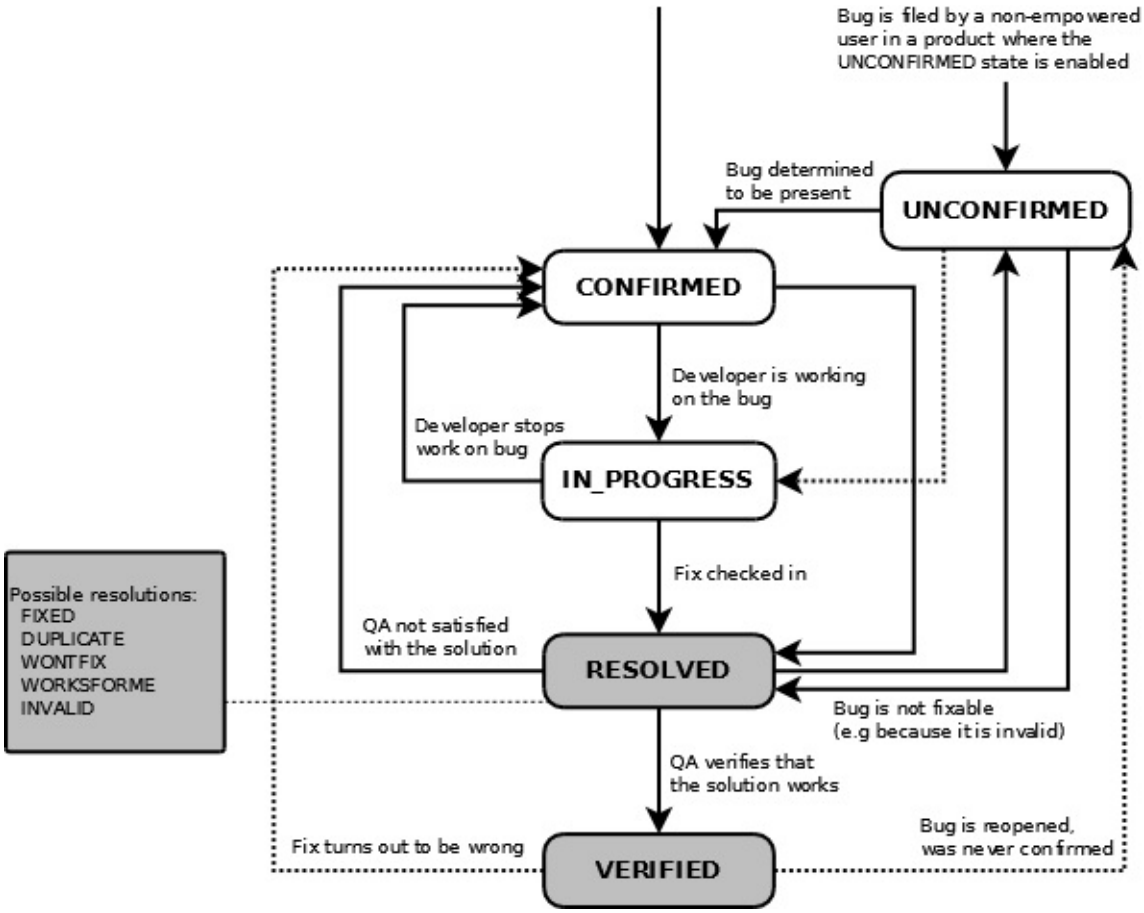
From the analytics point of view, the good news are that given these similarities, the analysis of bug reports, feature implementation and code review are quite similar. That is the main reason why in most of this chapter we talk about processes instead of tickets or code review. However, there are some important differences too, for example in the workflows, or in some metrics which can be unique to some of these cases but not to the others, but still are very important for the cases that apply.

Workflow patterns

Projects use different workflows to deal with processes. In many cases, even they can have specific policies, such as defining allowed state transitions to deal with tickets, or review patterns to deal with proposals of change to the source code. When the different states and transitions are recorded in some development repository, the complete process can be tracked. This allows for the analysis of the real workflows, and their compliance with good practices or project policies. In addition, time between transitions can be measured, to detect bottlenecks and compute time for different workflows.

Workflows for tickets

The workflow of tickets depend a lot on the specific ITS used. In addition, most of them have a basic workflow defined, but either the administrators of the system, or in some cases its users, can customize it. For example, the next figure shows the default workflow for Bugzilla, whcih different projects adapt in different ways.



Default workflow for a ticket in Bugzilla 4.0.1, as shown in Life cycle of a bug, in the Bugzilla manual.

However, there are some aspects in which most, if not all ITS behave teh same way. The life of a ticket starts when it is opened. In some cases there is a specific "new" or "open" state, in which it stays until the first actions are taken, usually assigning it to someone. This process is called "triaging the ticket". It is very important, because until the ticket is assigned,

it is very unlikely that someone starts to work with it. Therefore, "time-to-triage" is a meaningfull metric. The process of triaging may involve confirming a bug, or asking more details about it, or discussing whether a feature request makes sense.

Once the ticket is "ready to be worked on", some actions are taken until it is resolved. That means that the person in charge considers that the work is done. But in most cases, that has to verified, so that a third party, maybe the person who open the ticket, considers that indeed the issue is solved. Therefore, time-to-resolved and time-to-verified are interesting metrics, telling about how developers are working, and about how good their work is (in terms of satisfying third parties verifying the solutions).

In any state, the ticket can be closed. For example, an old untriaged ticket can be closed because no further action can be done on it. Or an unsolved ticket can be closed because new versions render the ticket void. Usually, tickets are closed after being verified. But even in this case, they can be reopen if new evidence arises suggesting that the work is not really done.

Workflows in code review

In code review, the workflow can be divided between two states: waiting for review and waiting for new change proposals. For entering into details, let's consider two of the most usual CRS these days: Gerrit and GitHub Pull Requests.

For Gerrit, the workflow starts with a developer submitting a proposal for a change (a change). This proposal is composed by a patchset (in fact, the contents of a git commit), and a comment (which may include a reference to the ticket which originated the change proposal). Once the patchset is submitted, anyone can review it, by submitting comments and tags. Tags can range from -2 to 2. Usually +1 means "I agree with this change", and -1 means "I don't agree with this change". -1 should be accompanied by a comment stating what should be changed in the patchset to make that reviewer happy. -2 and +2, when used, usually refer to special reviewers, that in some projects have the decision power. In a given patchset is approved (+1 or +2) it gets merged in the main branch, and is considered to enter "production-ready code". If it gets rejected (-1, -2), the developer is expected to send a new patchset, addressing the concerns of reviewers. In addition, automated testing can also be the cause of rejection. If change submitters come to a point when they don't want to produce new patchsets to address reviewers concerns, they can abandon the change.

In this workflow we can measure the total time to accept, but also the time waiting for review: since a patchset was uploaded to when a reviewing decision on it was made. And the time waiting for the developer: since a new patchset was requested to when it was uploaded. Some other timings, related to how long does it take the automated testing process can also be considered. And finally, the number of cycles, or patchsets, that were needed for a change proposal to be accepted can be an interesting metric too.

For GitHub Pull Requests, the workflow is a bit more simple. The process starts by submitting a pull request, which is a collection of commits. Anyone can comment on the pull request, and comments can include requests for changes to the commits. The developer can change the commits to have those comments into account. When someone with commit rights considers that the pull request is ready, can merge it in the code base. Automated testing can be used too, which usually annotates the pull request ticket with comments or flags.

This process is more flexible than the Gerrit one, but also more difficult to track in an uniform way. For example, is very difficult to tell when a change in the commits was made, or even if it was made at all, since no tags are mandatory. This means that measuring time waiting for reviewer or for developers are difficult to measure, or even to define.

You can see more details about how a specific project uses Gerrit for code review in the OpenStack Developers Manual. For details on GitHub Pull Requests, you can read Using pull requests.

Participation

We can also analyze who participated in specific processes in a project. From that point on, we can assess on several dimensions, discussed in sections below, such as diversity in participation or neutrality.

The relevant participants are different depending on the kind of processes. For example, for ticket closing:

- Openers or submitters. Those that file the tickets, either reporting a bug, requesting or proposing a feature, or raising any other issue.
- Commenters. Those commenting on the ticket. Comments may be requests for more information, provisions of that requested information, reporting of the progress, proposals to receive feedback, etc.
- State changers. Those changing the state of the ticket. In some ITS, they can be just flagging the ticket in some way.
- Closers. Those closing the tickets.

But in fact, for any transition of state, an actor can be defined, which leads to very specific kind of actors, depending on the project. This means as well that the identification of states and transitions in the workflow with tickets is fundamental for the identification of relevant participants.

As a side note, the identification of relevant participants in specific actions in the ITS may be tricky. For example, in the case of OpenStack, the person considered to be closing a ticket is not really the one changing the state to closed, but the owner of the ticket at that time (the owner is the person assigned to it).

In the case of code review, the most relevant participants are:

- Submitters. The persons submiting proposals for change.
- Reviewers. The persons reviewing the changes.
- Commenters. The persons providing comments. This can be reviewers, commenting on how to improve the change proposal, oor submitters, commenting on how the will address reviewers' suggestions, or third parties.
- Core reviewers. In some projects, there is a special kind of reviewer, elected by the project, whith have the power to accept or reject changes.

Diversity in participation

There are several aspects about participation in processes where diversity has a role:

- Participation from diverse time zones in specific process, which may help to speed the process up, or to slow it down, depending on how is that participation.
- Participation by organizations, which is an indicator over the level of control that a single actor, or a small group of them, maybe have over a project. Or the level of dependency that the project has on those actors.

To analyze diversity, it is needed to carefully determine the relevant actors, and then characterize them from any of these points of view (geographical area, affiliation to organziations, etc.).

Neutrality

Neutrality means how neutral the community is to the different individuals or groups that work together in it. Neutrality is important to the community, because it ensures that all actors are considered equal with respect to the characteristics that are not related to their capacities or skills. For example, it ensures that no company intentionally delays fixing bugs that were reported by some other company, or that code proposed by people from a certain region is not taking longer to review, given other aspects are equal.

Once the diversity analysis is performed, and the relevant diversity characteristics to consider are defined, the neutrality analysis produces metrics for each of the individuals or the groups identified, to allow for comparisons. But those metrics have to be considered with some care. For example, a neutrality analysis can show how time-to-review for developers of company A is taking twice than for company B. This could easily to the conclusion that company A is being discriminated by reviewers with respect to company B. But it could happen as well that developers from company A are much less carfull, or

less experienced, or less trained, than those of company B. And that could lead to employees of A submitting much worse change proposals than those of B. Which would explain perfectly the difference in time-to-review, since reviewrs could be much more reluctant to decide on their code because of those reasons.

Therefore, the main goal of the neutrality analysis is to provide metrics that are at the same time fair and relevant. That is, that the differences in them are really related to discrimination and lack of neutrality, and not to different skills or expertise.

Metrics for tickets

Based on all the information above, in this section we study which metrics are more relevant for tickets. Let's start with some timing metrics for feature requests:

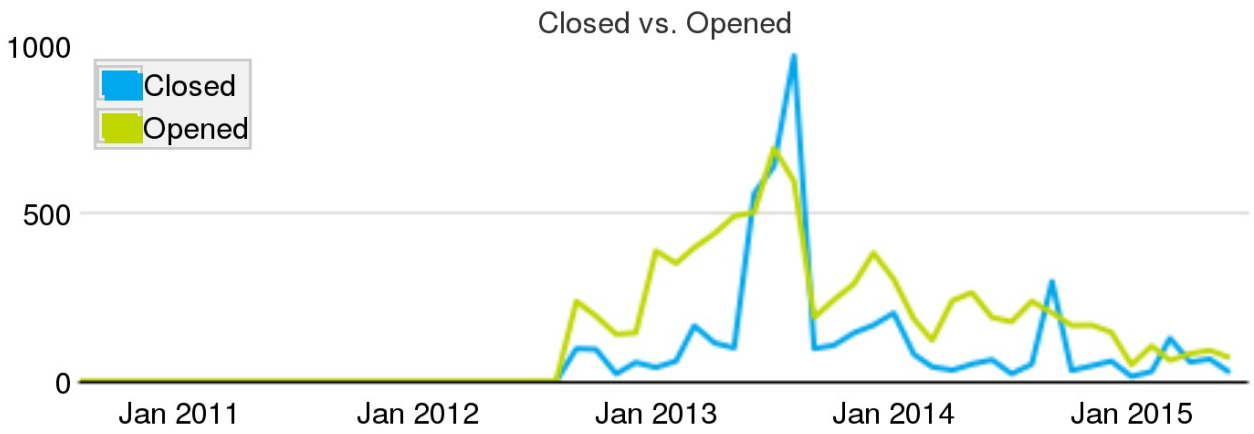
- Time to attend: Up to the moment there is some comment by a developer, usually commenting on the feasibility of the request, and maybe assigning to a developer for implementation.
- Time to first patch: Up to the moment a patch implementing the feature is attacjed to the ticket.
- Time to final patch: Up to the moment a patch lands in the code base intended for the next stable release. In some cases, this is equal to time to first patch, because there is no further process once the patch is produced. But in others, code review or automatic testing is in the middle of landing into the code base.
- Time to release: Up to the moment the patch is included in a public release. If the system is following a continuous release policy, this can be exactly equal to time to final patch. But when point releases are produced at discrete moments, this time can be considerably longer.
- Time to deployment: Up to the moment the patch is deployed into production systems. In many cases, the FOSS project doesn't have a direct reference to this moment, since it happens downstream, in the institutions using the software.

Ratios between new feature requests and final patches or released patches are interesting to evaluate the performance of the project in terms of how much it is coping with new requests.

In any case, the metrics discussed for new features can be applied to bug fixes too. There are some other intersting metrics as well:

- Reopenend bug reports. Those tickets that after being considered closed, because it seemed that the bug was fixed, have to be reopened because it was not fixed at all.
- Ratio of reopened to closed. Give an idea of how effective is the bug fixing process. If the ratio is high, that means that many bugs are assummed to be fixed when they are not, and therefore have to be reopened.

Because of the difficulty of differentiating between feature requests and bug reports, in many cases it is useful to obtain the metrics for all tickets together.



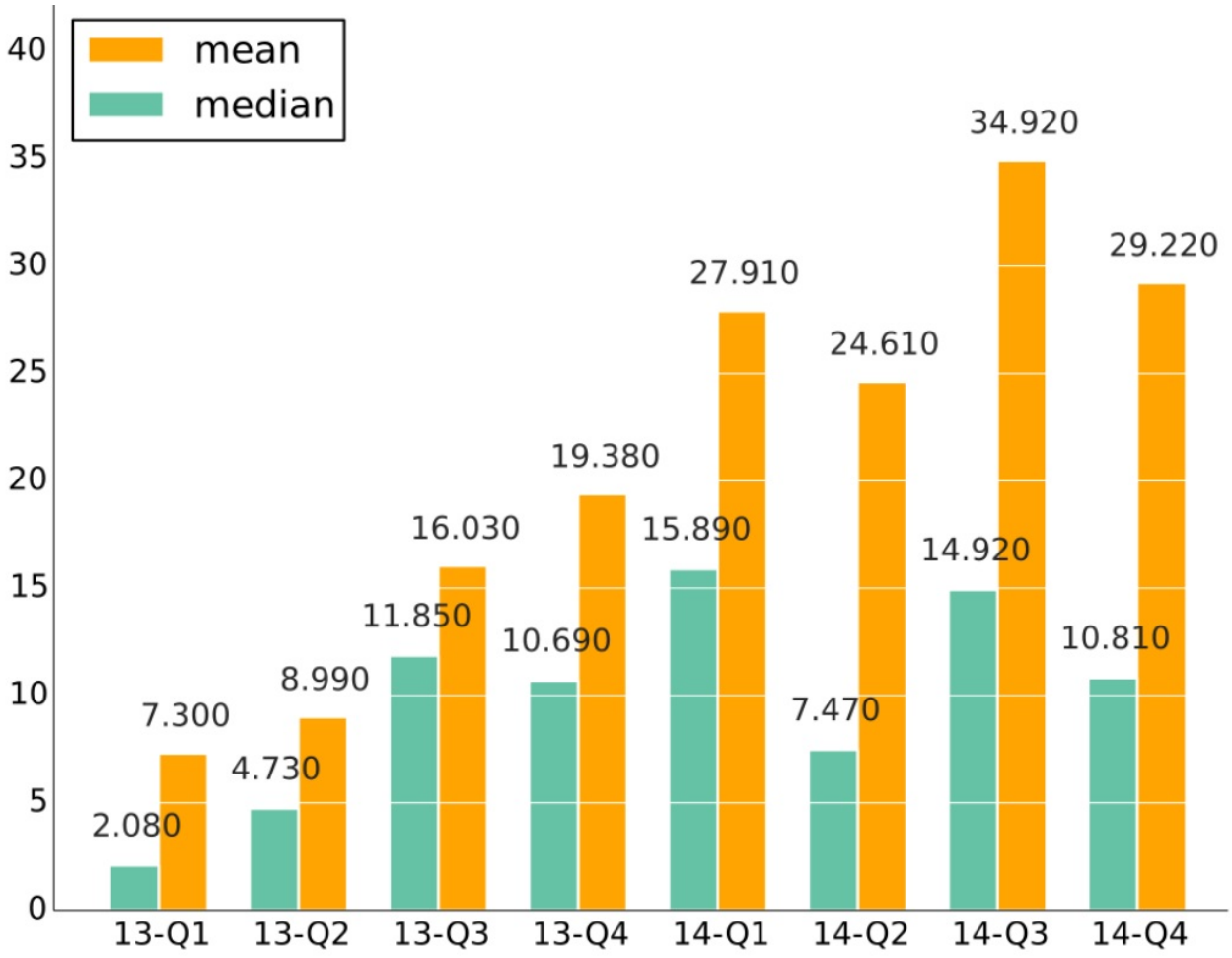
Evolution of closed and open tickets per month for the Apache Cloudstack project circa July 2015

For example, the chart above shows the evolution of closed and opened tickets per month for a FOSS project. In this case, it can be observed how the number of opened tickets is larger than closed tickets almost for every month. This situation, which is very common in real projects, means that every month the project is not coping with all the new work they have. From time to time, the project can run "closing parties", or use bots just to close old bugs that are never going to be fixed, and maybe are no longer bugs. The blue spike in Summer 2013 could be one of such cases.

Metrics for code review

Most of the metrics used for issue tracking systems (either bug fixes or feature requests) can be applied to code review, if we consider the submission of the patch to review as the starting point of the metric. Therefore, we have:

- Time to attention: Up to the moment a reviewer starts taking action on the code review system.
- Time to review: Up to the moment reviewers take a decision on the review of the code, such are accepting it, or requesting some changes from the submitter.
- Time to new submission: From the moment some changes were requested by reviewers, to the moment a new patch is sent.
- Time to merge: From the moment a code review process starts, to the moment the corresponding change lands in the code base.



evolution of time to merge in Nova (an OpenStack subproject), by quarter, during 2013 and 2014.

Based on these times, aggregated times for each review process can be computed:

- Reviewer time: Aggregated periods while the submitter is waiting for review by reviewers. This metric captures the amount of delay in the process which is under the responsibility of reviewers.
- Submitter time: Aggregated periods while reviewers are waiting for a new submission. This metric captures the amount

of delay which is under the responsibility of submitters.

Therefore, a long time to review may be due to a long reviewer or submitter time. The problems the project is facing in both cases, and the means to solve them, are very different.

Not only time is interesting:

- Number of review cycles: Number of reviews (from submission of a change to acceptance or request for a new change) that are needed to finish a review process.

In addition, metrics about the effectiveness of the code review process are useful too. For example:

- Ratio of abandoned to merged changes. Gives an idea about how many of the submitted proposals for change end nowhere, with respect to how many end in the code base.
- Ratio of merged to submitted changes. This is a kind of a success ratio, showing how many review processes finish with a change in the code base with respect to how many were started.

Metrics for mailing lists

In mailing lists and other asynchronous communication systems process metrics can be useful as well. Communication channels can be used by users to solve problems or report them. This interaction can be monitored in several repositories. In our experience, this can be done mainly in ITS, but also in asynchronous communication systems. From those, it can be known:

- Who asks for help, or in general comments on issues related to the project.
- Who participates in solving those issues.
- Who of those who participate are developers.

And some timing information, such as:

- Time to first answer. From the moment a question is made, to the moment the first answer arrives.
- Time for whole discussion. Which is the timespan from the first message in a thread to the last one.

Evaluation models

This chapter describes some of the evaluation models that can be used for FOSS projects. We will focus on quantitative models, since they are easier to apply and replicate, and probably, more useful. This does not mean that qualitative models are not important. But since they are more based on qualitative perception, they are very dependent on the expertise and familiarity of the expert performing the evaluation. Quantitative models try to be more independent from the person doing the analysis, but defining quantitative data that tries to capture the relevant aspects of the project being evaluated.

Of course, there are shadows in a continuum ranging from pure quantitative to pure qualitative. In fact, the models mentioned in this section may have some aspects which are at least partially qualitative.

Basics of quantitative evaluation

Quantitative evaluation is based on the identification of quantitative parameters that can be significant, and the definition of measurement models for them.

Given the number of evaluation models that exist, Stol and Ali Babar have proposed a comparison framework to evaluate them. In order to do so, the most relevant evaluation models were identified. The result of this identification process, after screening around 550 research papers, is provided in following table, with 20 approaches. The column "Orig" shows if the initiative is the result of a research (R) or an industrial (I) effort. Models have been classified as industrial if they are associated to at least one company. The column "Method" indicates the completeness of the methodology, meaning that if all required activities, tasks, inputs and outputs are outlined, the assessment methodology offers a complete guide to evaluation. If a mere set of evaluation criteria are proposed, then the authors labeled the methodology as not complete.

Name	Year	Orig	Method
Capgemini Open Source Maturity Model	2003	I	Yes
Evaluation Framework for Open Source Software	2004	R	No
A Model for Comparative Assessment of Open Source Products	2004	R	Yes
Navica Open Source Maturity Model	2004	I	Yes
Woods and Guliani's OSMM	2005	I	No
Open Business Readiness Rating (OpenBRR)	2005	R/I	Yes
Atos Origin Method for Qualification and	2006	I	Yes
Selection of Open Source Software (QSOS)	2006	R	No
Evaluation Criteria for Free/Open Source Software Products	2007	R	No
A Quality Model for OSS Selection	2007	R	Yes
Selection Process of Open Source Software	2007	R	Yes
Observatory for Innovation and Technological transfer on Open Source software (OITOS)	2007	R	No
Framework for OS Critical Systems Evaluation (FOCSE)	2007	R	No
Balanced Scorecards for OSS	2007	R	Yes
Open Business Quality Rating (OpenBQR)	2007	R	Yes
Evaluating OSS through Prototyping	2008	R	No
A Comprehensive Approach for Assessing Open Source Projects	2008	R	Yes

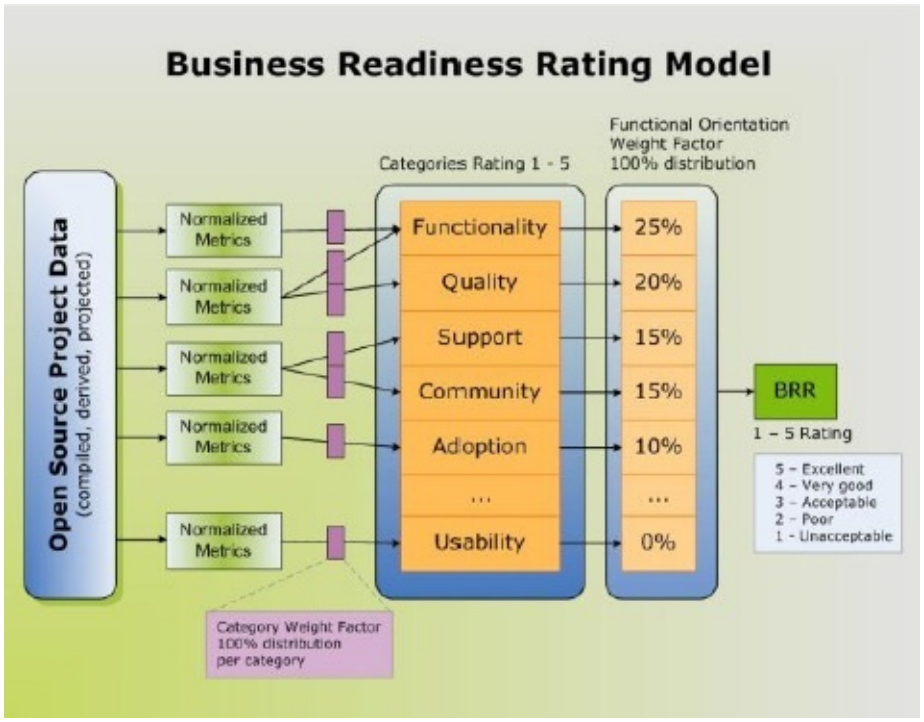
Software Quality Observatory for Open Source Software (SQO-OSS)	2008	R	No
An operational approach for selecting open source components in a software development project	2008	R	No
QualiPSo trustworthiness model OpenSource Maturity Model (OMM)	2009	R	No

Of all these models, we have selected some that we describe in some more detail in the following sections.

OpenBRR

The OpenBRR (Open Business Readiness Rating) is an evaluation method proposed in 2005 and sponsored most notably by Carnegie Mellon and some industrial partners (CodeZoo, SpikeSource and Intel) The OpenBRR white paper. The goal of this method is to provide an objective manner to assess community-driven projects, offering a final quantitative mark that is intended to provide a measure of its readiness to be deployed in a business environment.

Following figure provides an overview of the how OpenBRR should be applied. As can be seen, OpenBRR involves a multi-step evaluation process, that can be adjusted by the evaluator to adapt the assessment to the specific needs of the organization that wants to deploy the software under study-



OpenBRR is based on gathering metrics and factual data on up to following ten categories:

- Functionality
- Usability
- Quality
- Security
- Performance
- Scalability
- Architecture
- Support
- Documentation
- Adoption
- Community
- Professionalism

For each category, a set of criteria and metrics are proposed. These inputs are then weighted and each of the above introduced categories are given a rating that ranges from 1 to 5. Then, depending on the final usage the software will be given, adopters may weight these categories, obtaining an overall rating of the project. Hence, not all categories are weighted equally, and for some scenarios a category may not be considered at all for the final rating (in that case, its weight factor would be 0%).

To help in the assessment, OpenBRR offers a spreadsheet template that can be used in the evaluation process. Many of the input data in this model are to be obtained by external tools or from the Internet. As an example, the quality category considers the following inputs:

- Number of minor releases in past 12 months
- Number of point/patch releases in past 12 months
- Number of open bugs for the last 6 months
- Number of bugs fixed in last 6 months (compared to # of bugs opened)
- Number of P1/critical bugs opened
- Average bug age for P1 in last 6 months

These inputs are rated as well from 1 to 5, and the evaluator may then weight them in a posterior step.

Udas et al. discuss in a report how to apply OpenBRR in real environments based on their experience in the evaluation of Learning Management Systems. The 31 page report is very exhaustive and provides some general guidelines to be followed when using OpenBRR. It also gives an idea of how difficult and time-consuming it is.

The OpenBRR website provided a set of examples of use of the evaluation model. Of these, the most known assessed Moodle and Sakai, two well-known learning management systems that were widely used in industry and academic institutions. As they introduce the OpenBRR assessment process very well, we will show them here in detail. You can browse the OpenBRR spreadsheet for Moodle and the OpenBRR spreadsheet for Sakai for more details.

The first step in the process is to select and weigh the criteria to be use in the evaluation process. In the case of Moodle and Sakai, the evaluators chose to use the following:

Rank	Category	Weight
1	Functionality	25%
2	Usability	20%
3	Documentation	15%
4	Community	12%
5	Security	10%
6	Support	10%
7	Adoption	8%
Total		100%

leaving out five criteria: Quality, Performance, Scalability, Architecture, and Modularity.

In the following step, each criteria is evaluated on its own. As an example, for the evaluation of the functionality, a list of 27 standard functionality items (that include from discussion forums to surveys or automatic testing) are included, which have been obtained from the edutools.info on-line portal. Depending on the grade of its completeness, each functionality is scored and weighted from 1 to 3 as shown in the following table. Additional 8 extra functionalities (such as LaTeX support or the inclusion of video) are rated in the same fashion.

Weight & Test Score Specification	Score

Very important	3
Somewhat important	2
Not important	1

In order to obtain a total score for the functionality criteria, the total weights of the standard functionality items is summed up in W. Then the score for the assessed tool is obtained by adding all the scores, both from the standard and extended functionality, as T. Depending on the completeness of T related to W (in percentage), a final rating score is provided, using the cutoff values provided in following table.

Rating Score Table	Percentage Cutoff	Score
Unacceptable	0%	1
Poor	65%	2
Acceptable	80%	3
Good	90%	4
Excellent	96%	5

In our case studies, Sakai obtains a 3 out of 5 (its percentage is 86.67%, as it has a total score of 52 out of a total weight of 60), while Moodle obtains 5 out of 5 (its percentage is 106.67% with a total score of 64 out of a a total weight of 60.

Once this is done with all evaluation criteria, the score of each of the criteria is introduced in a spreadsheet and the final score is calculated. It should be noted that when doing so the previously defined weights are to be taken into consideration. For instance, the results of this step is provided in the following table for Moodle and Sakai. The total score of 4.19 for Moodle and of 3.23 for Sakai is finally obtained by summing up all the weighted scores for each of the categories.

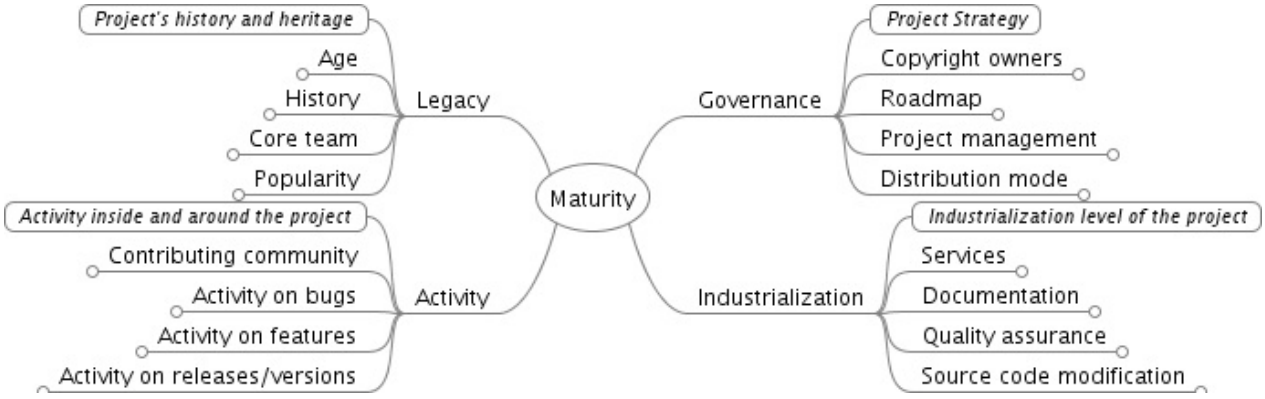
Rank	Category	Moodle Unweighted	Sakai Unweighted	Weight	Moodle Weighted	Sakai Weighted
1	Functionality	5	3	25%	1.25	0.75
2	Usability	4	4	20%	0.8	0.8
8	Quality	0	0	0%	0	0
5	Security	4.2	3.4	10%	0.42	0.34
9	Performance	0	0	0%	0	0
10	Scalability	0	0	0%	0	0
11	Architecture	0	0	0%	0	0
6	Support	4	1.5	10%	0.4	0.15
3	Documentation	3.1	3.1	15%	0.47	0.47
7	Adoption	4.4	4.2	8%	0.35	0.34
4	Community	4.2	3.2	12%	0.5	0.38
12	Professionalism	0	0	0%	0	0

Although OpenBRR is one of the most known assessment models, it has not achieved to create a thriving community and currently it seems to have come to a halt.

QSOS

QSOS (Qualification and Selection of Open Source software) is an assessment methodology proposed by ATOS Origin in 2004 and updated in 2013. It is composed of a formal method that describes a workflow to evaluate projects, a set of tools that help to apply the QSOS workflow and a community. The proposed process is shown in the figure below. It is divided in four iterative steps and is iterative in nature, meaning that it can be applied with different granularity levels, becoming more detailed in every iteration.

The first step is concerned with defining the evaluation criteria in three axes: type of software, type of license and type of community. The type of software axis is composed by two additional criteria: a maturity analysis and a functional coverage analysis. The next figure shows a diagram with the specific items that are to be considered when assessing the maturity of a project. These items can be obtained in general from any free software project.



The second item for the type of software is related to the functionality of the project and depends on the software domain.

The type of license criterion evaluates the software licenses for three aspects: if the license is a copyleft license, if copyleft is only bounded to the module level and if the license allows to add restrictions.

Finally, the type of community criterion addresses the :

- A single developer working on his own on the project
- A group of developers, without formal processes
- A developer organization with formalized and respected software life cycle, roles and a meritocratic structure
- A legal entity (such as a foundation) that manages the community and acts as legal umbrella for the project
- A commercial entity: a company that employs some of the core developers and tries to obtain revenues from the development of the project

The second step involves the evaluation of the projects by obtaining data and measures from the project; raw evaluations

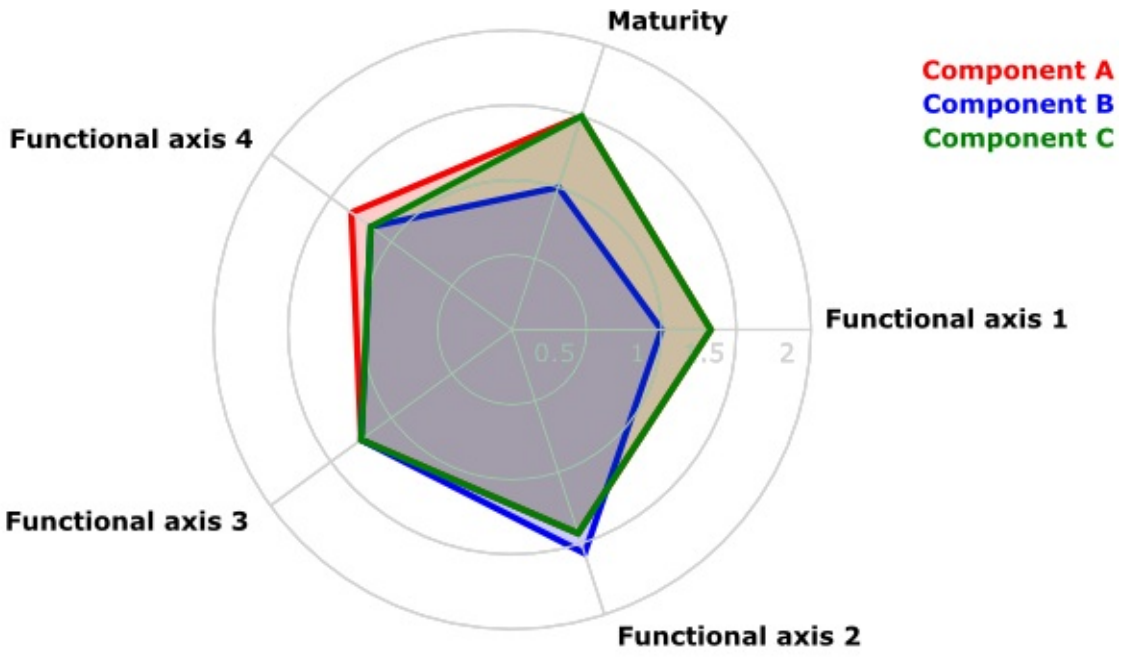
are the output of this step. For each of the criteria, a score between 0 and 2 is given. The following table provides the scoring rule in the case of the assessment of functionality:

Score	Description
0	Functionality not covered
1	Functionality partially covered
2	Functionality fully covered

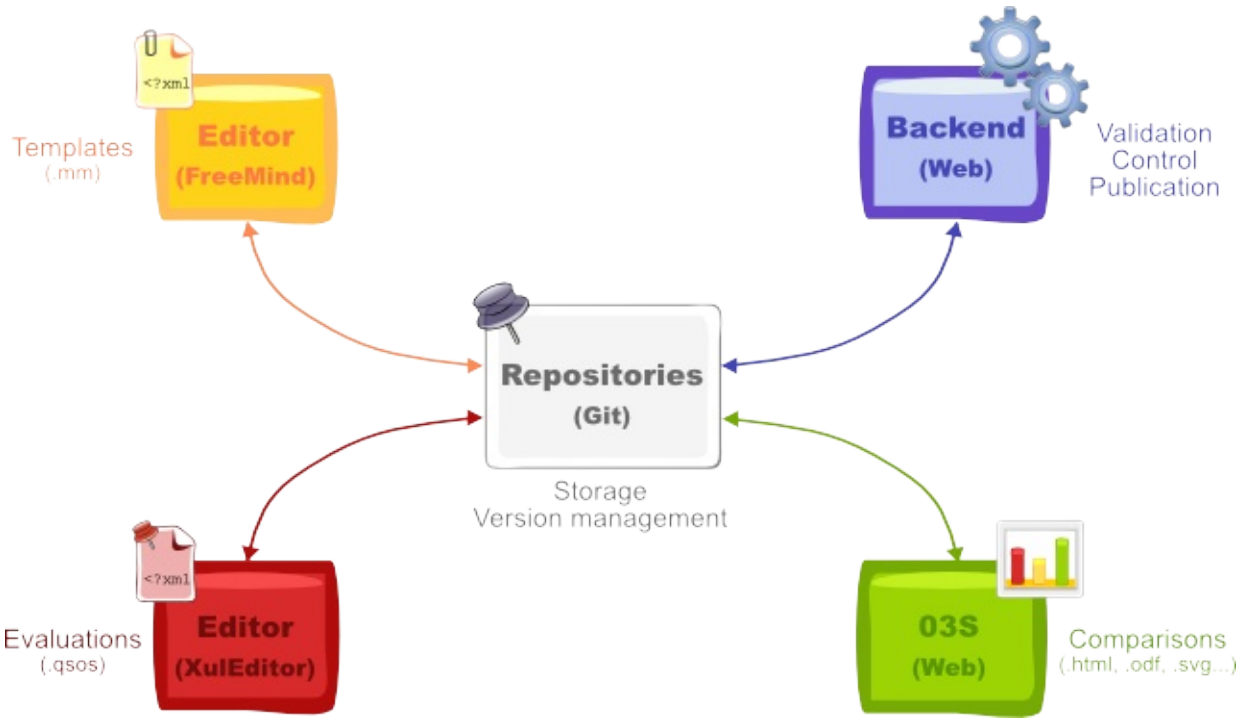
The results of the second step are then weighted depending on the context and the requirements under which the software will be used; specifically this is done by setting weights and filters in advance. So while in the first step, for instance all functionality may be assessed independently of its importance for its adoption and deployment, in this step the degree of relevance of each functional aspect will be translated into a weighting value. In the case of functionality, this means that functionalities may be considered required, optional or not required.

The final step is the selection of the most relevant software solution, by comparing the result obtained by several candidate software projects. QSOS offers two different modes of selection: strict and loose selection. Within the loose selection process, all software projects under assessment are evaluated for all criteria, and obtains a final rating. Within the strict selection process, as soon as a software does not comply with a relevant criteria of the evaluator, it is eliminated from the evaluation process. So, for instance, if a software does not require the required functionalities it is not further considered. It should be noted that with the strict selection procedure, and depending on the demands of the user, it may happen that no software meets the conditions.

The final result of QSOS can be shown and compared graphically by several means. One of them is using a radar format as shown in the next figure.



The QSOS framework offers a set of tools that help users follow the assessment process. Among them, there is an editor (the Freemind well-known mind-mapping tool) the create evaluation templates. These templates can then be used for the evaluation of a project using a Firefox extension or a stand-alone application. QSOS offers a web backend service where templates and evaluations can be made public and shared. Finally, O3S is a web-based tool that allows to manipulate evaluations, perform comparisons and export them in various formats.



Finally, the QSOS is an open project by itself, offering support to users and acting as a repository of templates and evaluations in several languages.

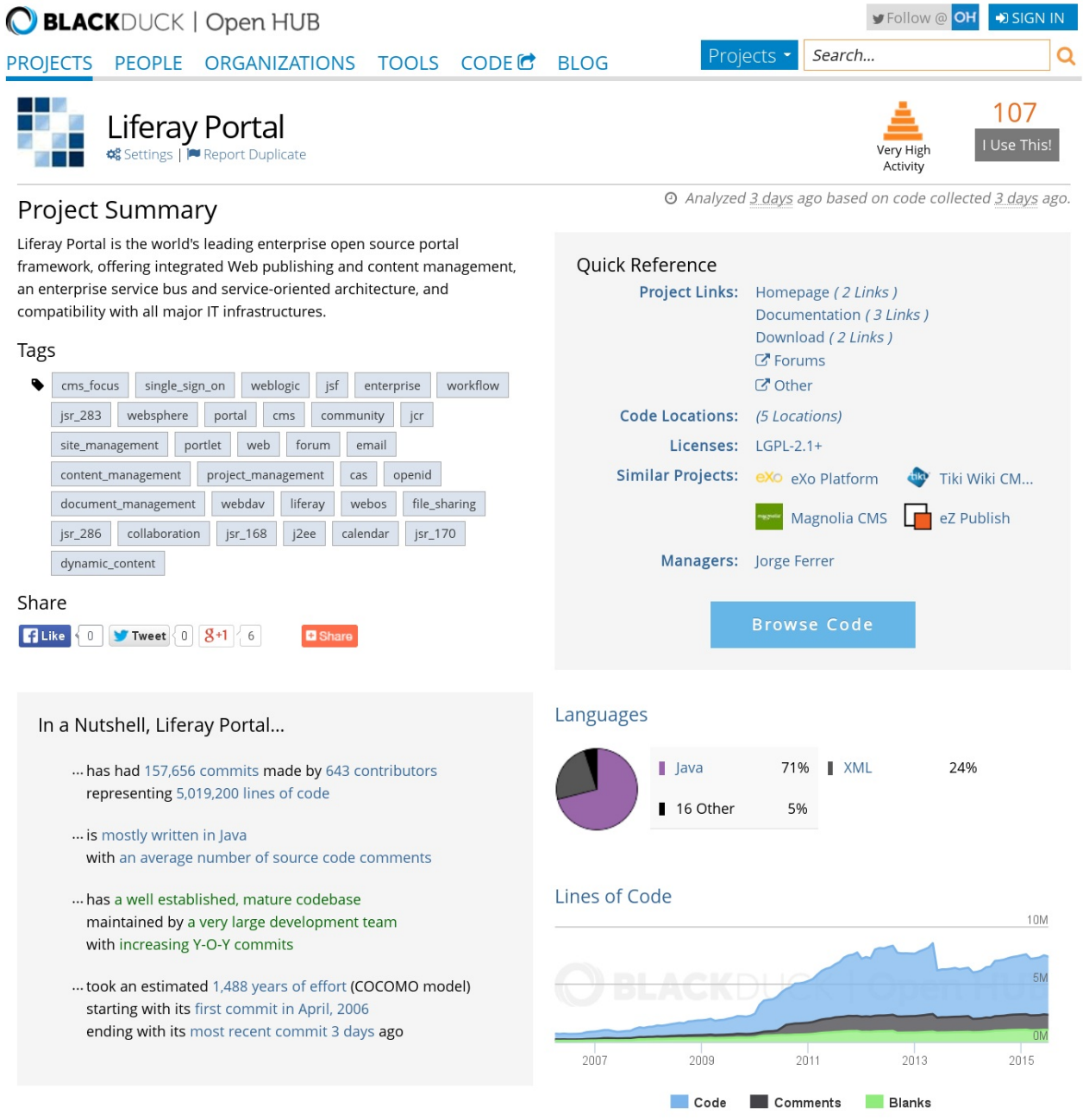
Evaluation dashboards

There are several software products or services that provide dashboards useful for evaluation. Each of them have different characteristics, and are useful in different parts of the evaluation process, or for different areas of evaluation. In this chapter we introduce some of them.

The Open Hub dashboard

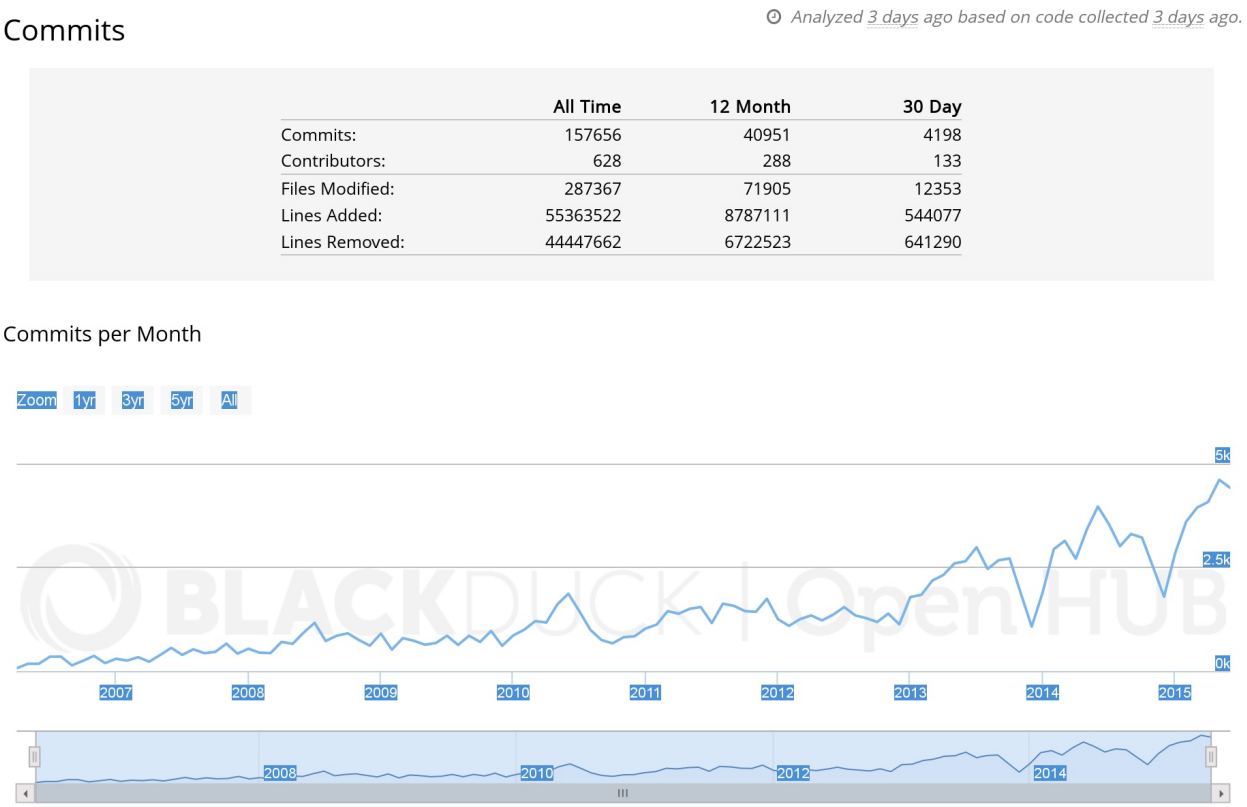
Open HUB, formerly known as Ohloh, is a website maintained by BlackDuck. Among other services, it provides a software development dashboard for a very large collection of projects (at least, tens of thousands of them).

All the information provided by Open HUB is based on the analysis of SCM. It includes both the analysis of the contents (licensing, lines of code, programming languages) and the metainformation (commits, committers, etc).



Main page of the Open HUB dashboard for Liferay Portal, as of July 2015.

The dashboard offers several views, each showing a different aspect of the project. The main page is a summary of the parameters of the project, and in some cases provides enough parameters for a basic assessment about its activity, and the size of the code produced. Then, there are specific panels showing analysis by programming language, and by lines of code, activity and developers.



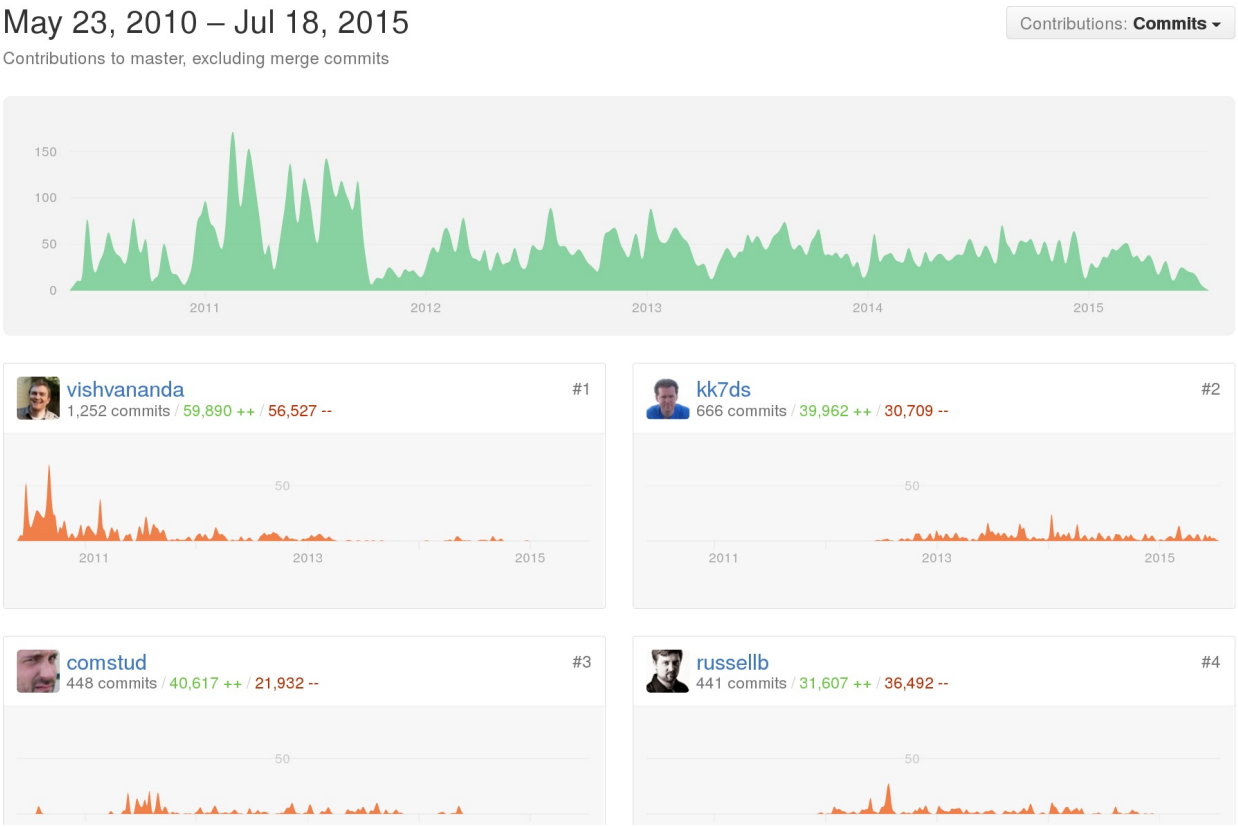
Commits panel of the Open HUB dashboard for Liferay Portal, as of July 2015.

The main interest of Open HUB is probably the huge number of projects for which they offer information. It is very like that if you look for any even minimally known project, Open HUB maintains a dashboard for it.

The main problems with this dashboard lie on the lack of support for data sources other than the SCM, on its relative simplicity, and on the fact that being proprietary software, only they can improve and customize it.

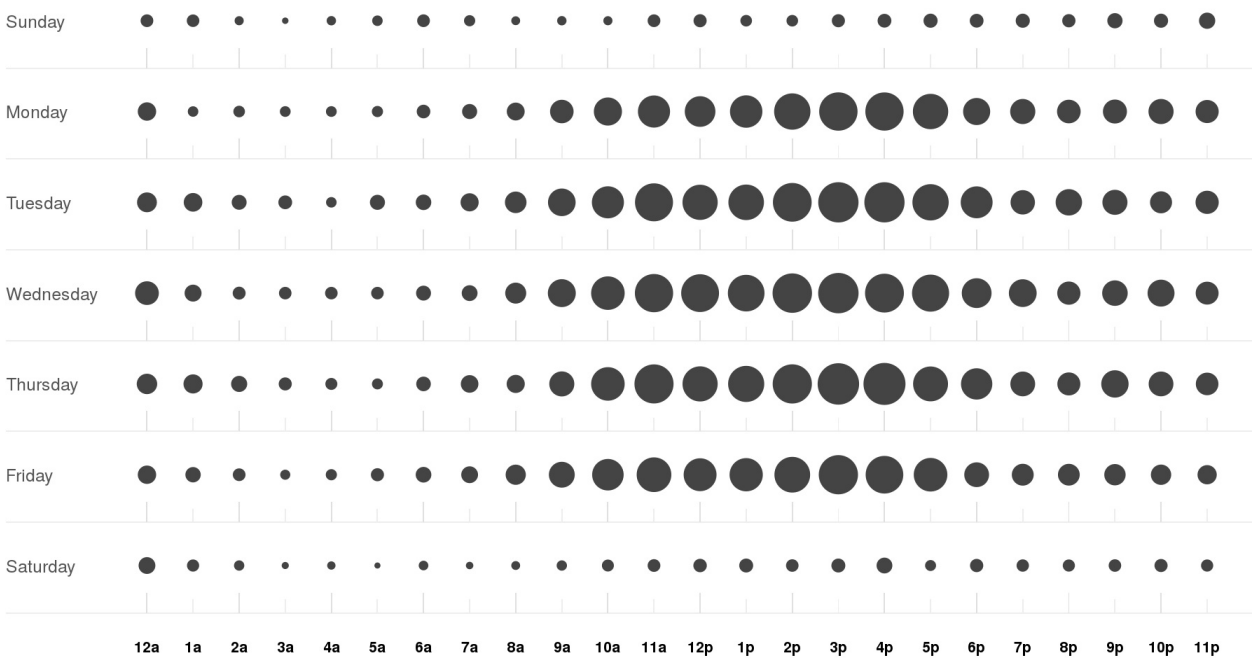
GitHub statistics

GitHub provides some statistics for the repositories it hosts. Most of it is based on the activity in the git repository, although it provides some basic data about issues and pull requests as well.



Contributors panel of the GitHub dashboard for OpenStack Nova, as of July 2015

Even when they don't use the term "dashboard", they provide a simple one. It includes information on contriutors and activity, with a focus on the historical evolution of contributions.



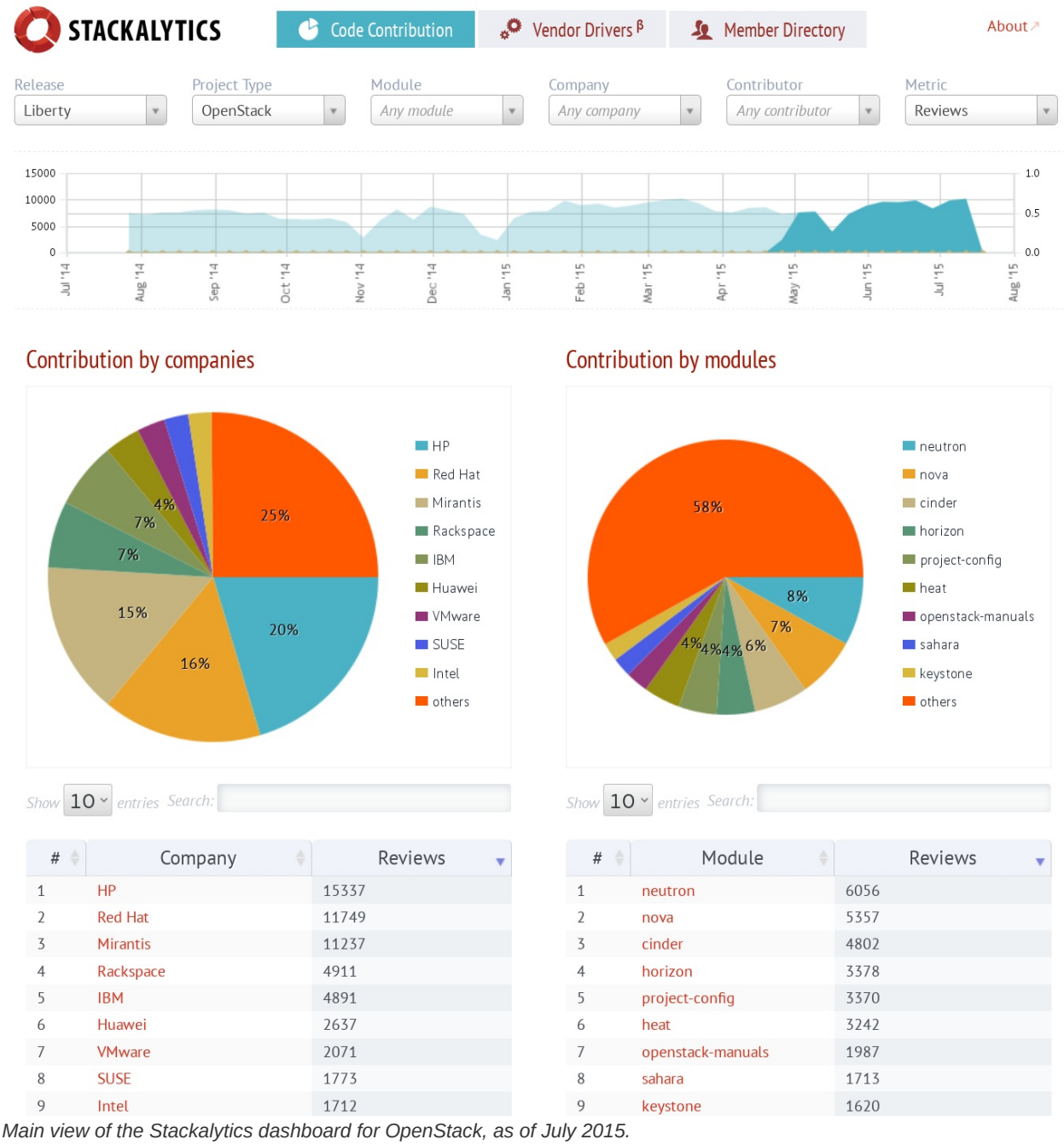
Activity punchcard for OpenStack Nova, shown by the GitHub dashboard, as of July 2015

The dashboards provided by GitHub are intersting because, even being simple, they are available for all the repositories the site hosts. And they host most of the FOSS being developed these days. Therefore, as in the case of Open HUB, it is very likely that if you need some simple metrics for almost any project, you can find a repository in GitHub with it, and therefore a dashboard providing it.

On the other hand, some of the main drawbacks are that the metrics provided are just a few, and in general simple, that they don't provided aggregated metrics, for example at the level of a whole GitHub organization, and that they are proprietary software, meaning that only they can improve their dashboard.

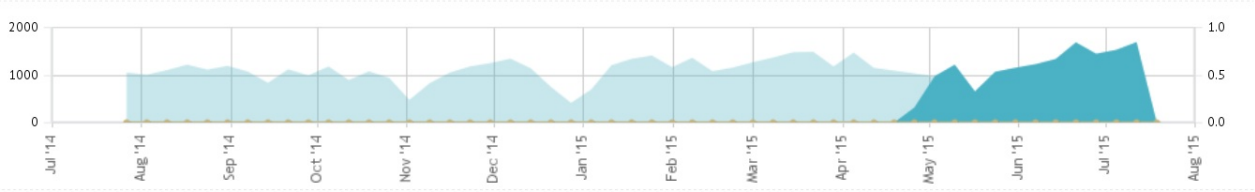
Stackalytics

Stackalytics is a service provided by Mirantis to the OpenStack community. It is based on FOSS, and for that reason it has been considered by other communities to gather and visualize their development metrics.

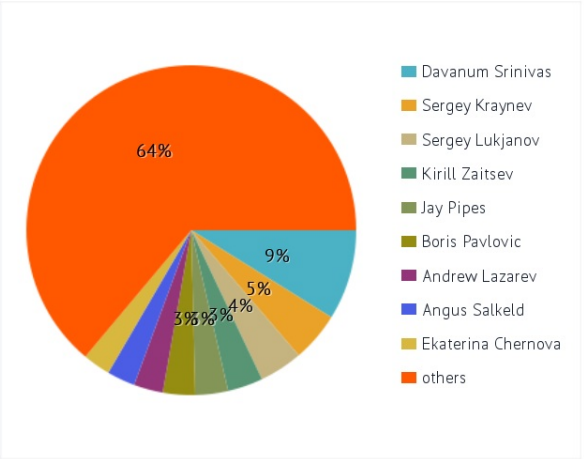


Main view of the Stackalytics dashboard for OpenStack, as of July 2015.

The Stackalytics dashboard shows a summary the activity and the community of the project, including the evolution of several parameters over time, and the current shares of contribution for companies, subprojects and developers.



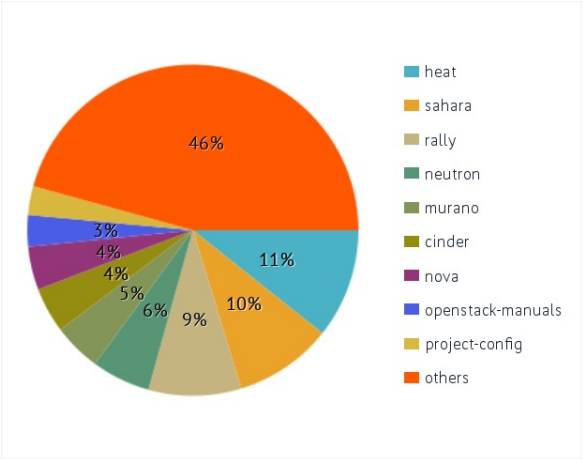
Contribution by contributors



Show 10 entries Search:

#	Contributor	-2|-1|+1|+2|A (+ ratio)	Reviews
1	Davanum Srinivas	0|91|172|732|567 (90.9%)	995
2	Sergey Kraynev	1|122|66|356|96 (77.4%)	545
3	Sergey Lukjanov	5|42|13|417|278 (90.1%)	477
4	Kirill Zaitsev	1|115|172|101|39 (70.2%)	389
5	Jay Pipes	1|109|79|172|53 (69.5%)	361
6	Boris Pavlovic	3|159|18|174|92 (54.2%)	354
7	Andrew Lazarev	0|61|8|253|3 (81.1%)	322

Contribution by modules



Show 10 entries Search:

#	Module	Reviews
1	heat	1202
2	sahara	1076
3	rally	1018
4	neutron	644
5	murano	526
6	cinder	498
7	nova	476
8	openstack-manuals	345
9	project-config	316

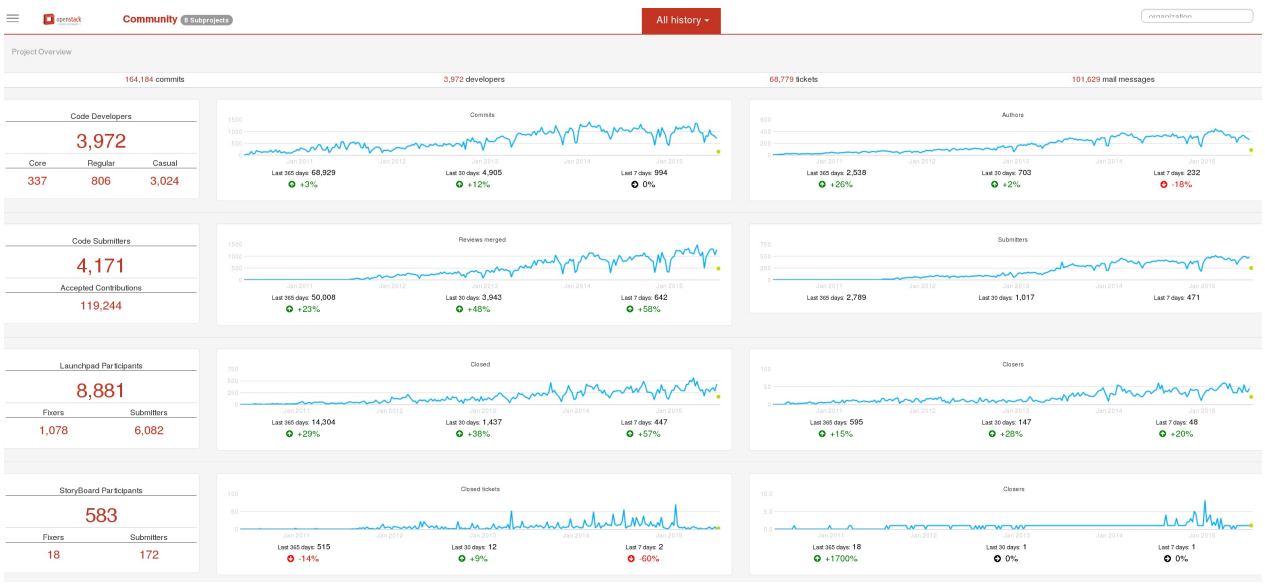
View for the activity of Mirantis, Stackalytics dashboard for OpenStack, as of July 2015.

Stackalytics is tailored to the specific needs of the OpenStack community. It provides information not only about commits in the git repositories, but also about tickets, code review and mail messages, making it a comprehensive tool that assists on the understanding of the OpenStack project. Being FOSS, the tool can be adapted to special needs, and is in fact being considered by other communities for providing a dashboard service to their developers.

In its current form, its main drawback is related to its main feature: bein specifically tailored to the needs and characteristics of OpenStack, it can be difficult to adapt to other projects.

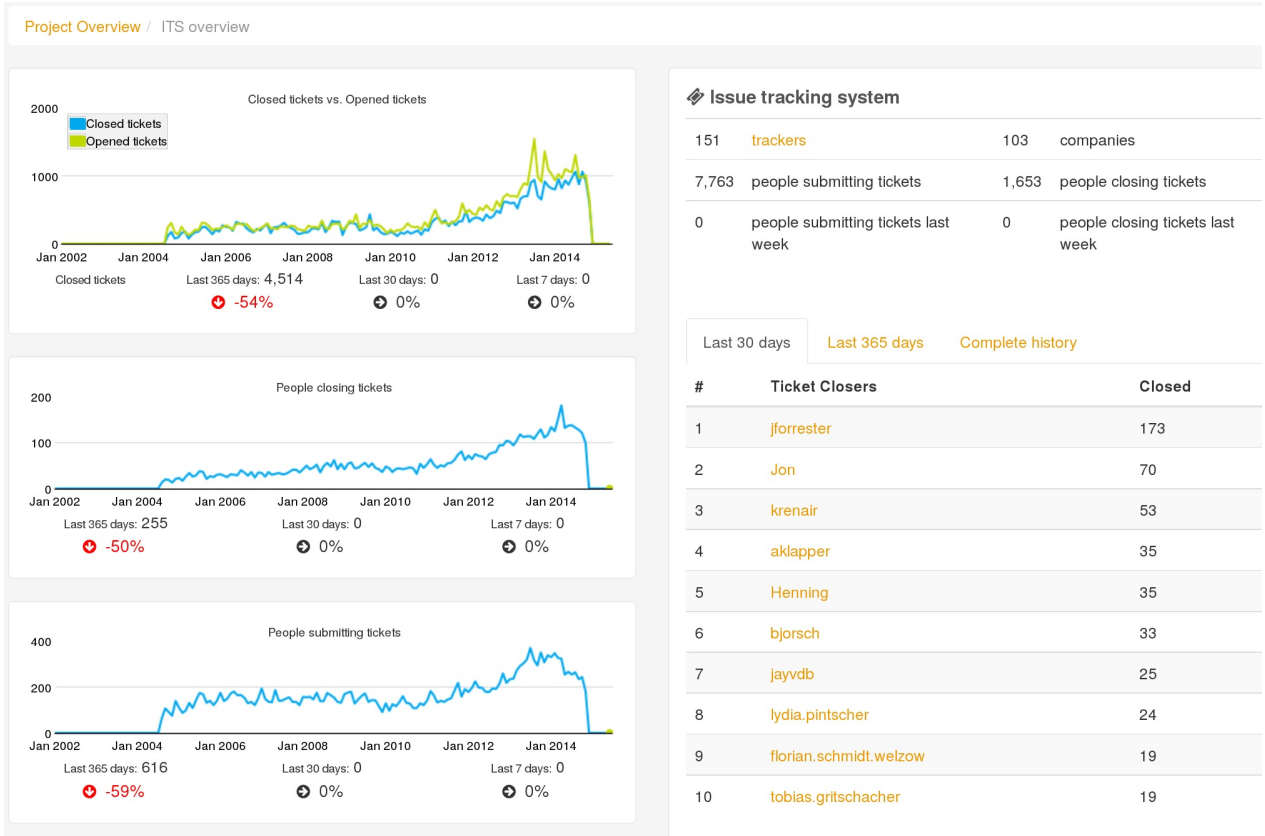
Grimoire Dashboard

Grimoire is a software system designed to retrieve information from software development repositories, store in a database, and then use that information for producing dashboards and other visualizations. Grimoire is FOSS, and can be adapted to many different needs, one of them being the grimoire dashboard.



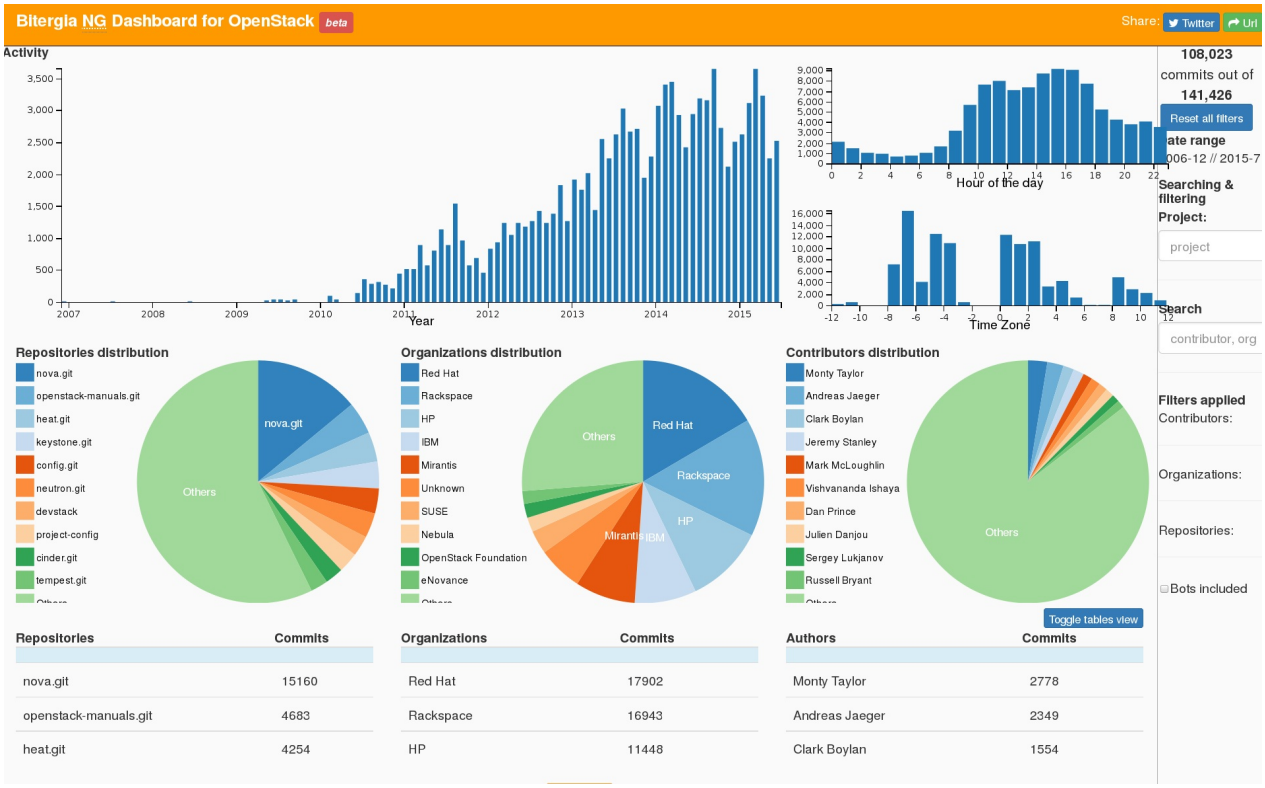
Main page of the Grimoire dashboard for OpenStack, as of July 2015

Grimoire is capable of extracting information from many different kinds of repositories related to software development. The dashboard uses the information retrieved from them in several panels, which show activity, community data, analysis of processes, diversity reports, and other kinds of studies.



Tickets panel for the Grimoire Dashboard for the Wikimedia Foundation projects, as of July 2015.

Grimoire is offered as a maintained service by Bitergia, but being FOSS it can also be deployed and customized by anyone. Currently one of its main drawbacks is that it is not easy to deploy, and that the capabilities for interacting with the data provided are limited. These problems are expected to be mitigated by the new generation of Grimoire, which is under development during Summer 2015.



Main view of the new generation of the Grimoire dashboard for OpenStack, in a preview of work in progress, July 2015.

Quantitative evaluation with Grimoire

[This chapter is still to be written]

Data retrieval

MetricsGrimoire

The database

Analysis and visualization

vizGrimoire

Analysis library

GrimoireLib

Visualization library

vizGrimoire JavaScript library.

The Grimoire dashboard

Automation of the whole process

Automator

Some details

Unique identities

Affiliations

Project hierarchies

Using Grimoire

[This chapter is still to be written]

Example of complete cases of use of Grimoire to analyze real projects.

Analysis of a git repository

Analysis of a GitHub project

Analusis of a medium project

Analysis of a large project

Acknowledgements and reuse of materials

Jesus thanks both Bitergia and Universidad Rey Juan Carlos, for the opportunity of participating in writing this book.

The section "Aging" of the chapter "Evaluating the community" is based on a draft of the article "Measure your open source community's age to keep it healthy", publised in O'Reilly Radar.

To probe further

This is an still disorganized list of random links. It will evolve hopefully in an organized list of useful resources.

Evaluation of community growth

- Brackets Health Report
- Polarsys Maturity Model
- Apache Maturity Model
- Open Business Readiness Rating (OpenBRR): website
- Comparing OpenBRR, QSOS, and OMM Assessment Models
- Modèle de maturité OSS, Smals Recehrche. Evolution of OpenBRR.
- Qualipso Open Source Maturity Model (OMM). Entry in Wikipedia, details in Qualipso-related website
- Open-source software assessment methodologies, article in Wikipedia.

Software evaluation (mainly functional, but not only):

- Software Evaluation Guide, Criteria-based Software Evaluation and Tutorial-based Software Evaluation, by the Software Sustainability Institute.

- Quantitative Methods for Software Selection and Evaluation, by Michael S. Bandor, September 2006

Bibliography

Paul Anderson. Avoiding abandon-ware: getting to grips with the open development method (HTML, visited on 2015-04-29).

Victor R. Basili, Gianluigi Caldiera, H. Dieter Rombach. The Goal Question Metric (PDF, visited on 2015-04-26).

Barend Jonkers, Cor Nouws. Comparing LibreOffice and Apache OpenOffice (PDF, visited on 2015-06-08; blog post linking to it, visited on 2015-06-08).

Klaas-Jan Stol, Muhammad Ali Babar. A comparison framework for open source software evaluation methods. Open Source Software: New Horizons, 389-394. (PDF, visited on 2015-06-24).

Carbon, R., Ciolkowski, M., Heidrich, J., John, I., and Muthig, D. Evaluating Open Source Software through Prototyping, in St.Amant, K., and Still, B. (Eds.) Handbook of Research on Open Source Software Technological, Economic, and Social Perspectives (Information Science Reference, 2007), pp. 269-281.

Assessment of the degree of maturity of Open Source open source software, (PDF.)

Lavazza, L. Beyond Total Cost of Ownership Applying Balanced Scorecards to OpenSource Software. Proc. International Conference on Software Engineering Advances (ICSEA) Cap Esterel, French Riviera, France, 2007, pp. 74-74.

Cruz, D., Wieland, T., and Ziegler, A. Evaluation criteria for free/open source software products based on project analysis, Software Process Improvement and Practice, 2006, 11(2).

Lee, Y.M., Kim, J.B., Choi, I.W., and Rhew, S.Y. A Study on Selection Process of Open Source Software. Proc. Sixth International Conference on Advanced Language Processing and Web Information Technology (ALPIT), Luoyang, Henan, China, 2007.

del Bianco, V., Lavazza, L., Morasca, S., and Taibi, D. Quality of Open Source Software The QualiPSo Trustworthiness Model. Proc. Fifth IFIP WG 2.13 International Conference on Open Source Systems (OSS 2009), Skövde, Sweden, June 3-6, 2009.

Golden, B. Succeeding with Open Source (Addison-Wesley, 2004).

Wasserman, A.I., Pal, M., and Chan, C. The Business Readiness Rating a Framework for Evaluating Open Source, 2006, Technical Report.

www.openbrr.org Business Readiness Rating for Open Source, RFC 1, 2005.

Majchrowski, A., and Deprez, J. An operational approach for selecting open source components in a software development project. Proc. 15th European Conference, Software Process Improvement (EuroSPI), Dublin, Ireland, September 3-5, 2008.

Polančič, G., and Horvat, R.V. A Model for Comparative Assessment Of Open Source Products. Proc. The 8th World Multi-Conference on Systemics, Cybernetics and Informatics, Orlando, USA, 2004.

del Bianco, V., Lavazza, L., Morasca, S., and Taibi, D. The observed characteristics and relevant factors used for assessing the trustworthiness of OSS products and artefacts, 2008, Technical Report no. A5.D1.5.3.

Ciolkowski, M., and Soto, M. Towards a Comprehensive Approach for Assessing Open Source Projects Software Process and Product Measurement (Springer-Verlag, 2008).

Taibi, D., Lavazza, L., and Morasca, S. OpenBQR a framework for the assessment of OSS. Proc. Third IFIP WG 2.13 International Conference on Open Source Systems (OSS 2007), Limerick, Ireland, 2007, pp. 173-186.

Cabano, M., Monti, C., and Piancastelli, G. Context-Dependent Evaluation Methodology for Open Source Software. Proc. Third IFIP WG 2.13 International Conference on Open Source Systems (OSS 2007), Limerick, Ireland, 2007, pp. 301-306.

Sung, W.J., Kim, J.H., and Rhew, S.Y. A Quality Model for Open Source Software Selection. Proc. Sixth International Conference on Advanced Language Processing and Web Information Technology, Luoyang, Henan, China, 2007, pp. 515-519.

Ardagna, C.A., Damiani, E., and Frati, F. FOCSE An OWA-based Evaluation Framework for OS Adoption in Critical Environments. Proc. Third IFIP WG 2.13 International Conference on Open Source Systems, Limerick, Ireland, 2007, pp. 3-16.

Atos Origin Method for Qualification and Selection of Open Source software (QSOS) version 1.6, 2006, Technical Report.

Petrinja, E., Nambakam, R., and Sillitti, A. Introducing the OpenSource Maturity Model. Proc. ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development (FLOSS 09), Vancouver, Canada, 2009, pp. 37-41.

Woods, D., and Guliani, G. Open Source for the Enterprise Managing Risks Reaping Rewards (OReilly Media, Inc., 2005).

Duijnhouwer, F., and Widdows, C. Open Source Maturity Model, Capgemini Expert Letter, 2003.

Koponen, T., and Hotti, V. Evaluation framework for open source software. Proc. Software Engineering and Practice (SERP), Las Vegas, Nevada, USA, June 21-24, 2004.

Samoladas, I., Gousios, G., Spinellis, D., and Stamelos, I. The SQO-OSS Quality Model Measurement Based Open Source Software Evaluation. Proc. Fourth IFIP WG 2.13 International Conference on Open Source Systems (OSS 2008), Milano, Italy, 2008.

Polančič, G., Horvat, R.V., and Rozman, T. Comparative assessment of open source software using easy accessible data. Proc. 26th International Conference on Information Technology Interfaces, Cavtat, Croatia, June 7-10, 2004, pp. 673-678.

Udas, K., and Feldstein, M. Apples to Apples: Guidelines for Comparative Evaluation of Proprietary and Open Educational Technology Systems. May 2006. SUNY Learning Network at the State University of New York, USA. (PDF, visited on 2015-07-15).