Analysis of Open Source Software Development Iterations by Means of Burst Detection Techniques

Bruno Rossi, Barbara Russo, and Giancarlo Succi

CASE – Center for Applied Software Engineering
Free University of Bolzano-Bozen
Via Della Mostra 4, 39100 Bolzano, Italy
{brrossi,brusso,gsucci}@unibz.it
http://www.case.unibz.it

Abstract. A highly efficient bug fixing process and quick release cycles are considered key properties of the open source software development methodology. In this paper, we study the relation between code activities (such as lines of code added per commit), bug fixing activities, and software release dates in a subset of open source projects. To study the phenomenon, we gathered a large data set about the evolution of 5 major open source projects. We compared activities by means of a burst detection technique to discover temporal peaks in time-series. We found quick adaptation of issue tracking activities in proximity of releases, and a distribution of coding activities across releases. Results show the importance of the application type/domain for the evaluation of the development process.

1 Introduction

The availability of source code and large communities of motivated and restless developers are two key factors at the base of the success of the open source software movement. Large interest has thus gathered the open source development methodology, very often compared critically to traditional software development practices in the quest for an answer to the growing number of failing software projects. Such methodology is mostly based on informal and distributed practices that seem to perform fairly well in domains constituted by turbulent and continuously changing requirements [11]. Practices such as *web-based collaborations*, *peer reviews*, *short cycle iterations*, and *quick releases* - among others - hamper the project management overhead and lead to a leaner development process.

In particular, considering both the *bug fixing process*, and *version release cycles*, many researchers claim that the open source methodology allows a faster bug-fixing process and higher release velocity than proprietary software [1, 6, 13].

Well-known empirical studies in this context are controversial. The *Apache* project was found to be very reactive to bug fixing requests, as well to provide many iterative software releases [9]. This conclusion was found in contrast with the development process of the *Mozilla* web browser, where the process was found equivalent to traditional development practices in terms of adaptability and release cycles [10]. For the *FreeBSD* system, similar in terms of number of core developers and reporters of

C. Boldyreff et al. (Eds.): OSS 2009, IFIP AICT 299, pp. 83–93, 2009.

failures to *Apache*, not enough evidence could be collected to confirm or reject the same hypotheses [5]. *FreeBSD* in the operating system domain, and *Apache* in the web server domain, were found to provide higher velocity in bug-fixing activities than other equivalent proprietary applications. In contrast with this view, *Gnome*, in the graphical user interfaces domain, was found to be less efficient in terms of bug-fixing speed compared to a proprietary application [8]. What appears from these studies is that, indeed, open source software has a faster bug fixing process but specific to particular applications and domains.

With this paper, we investigate the reactivity of the open source software development process when *fixing code defects* and when approaching *version releases*. We used a burst detection technique in time-series analysis to compare the evolution of peak activities during the projects' development.

The paper is structured as follows, in Section 2 we propose the research question, in Section 3 we propose the heuristic for project selection, and the data collection process, Section 4 is devoted to the method, Section 5 proposes the analysis of the datasets, and Sections 6, 7, 8 propose respectively discussion about the results, limitations, future works, and conclusions.

2 Research Question

Our general research question is to evaluate whether there is an increase in activities involving open source software repositories during *version releases* and *bug-fixing* activities. To better investigate the research question, we set-up *five* different hypotheses connected to our research question (Table 1).

The first two hypotheses refer specifically to the velocity of the *bug-fixing process.* We consider coding activities of developers correlated both to the *bug-opening process* (*H1*), and to the *bug-closing process* (*H2*). We expect that a fast bug-fixing process adapts quickly to the opened bug reports, whereas late reaction to users requests will lead to a correlation among coding activities and bug-closing activities.

Table 1. Low-level hypotheses under investigation

Hypothesis	*Rationale*
H1. There is an increase in code-related activities as there is an increase in the creation of bug-reports	immediate response from the developers as soon as large numbers of bug reports are inserted into *issue tracking systems*
H2. There is an increase in code-related activities as there is an increase in the closed bug-reports	a late reaction to users requests will lead to more code activities
H3. There is an increase in bug opening activities in the proximity of a software release date	a rapid increase in bug reports opened as there is a software release
H4. There is an increase in bug closing activities in the proximity of a software release date	a rapid increase in the bug closing process as a release date is approaching
H5. There is an increase in code-related activities in the proximity of a software release	code activities intensify as a release date is approaching

The open source community provides constant feedback to developers. We expect this effect to lead to an increase in bug reports opened in coincidence of a software release (*H3*). We also presume an increase in bug closing activities in correspondence to a software release (*H4*), and a synchronization of code activities with release dates, showing greater coding effort in proximity of version releases (*H5*).

3 Project Selection

To gather knowledge about the current open source landscape, we mined the *Ohloh* (https://www.ohloh.net/) repository. The repository is not only a large collection of metrics and evolutionary data about open source projects, but also a social networking opportunity for developers. As of November 2008, there are *20.590* projects listed including aggregated reports on the website. We used the *Ohloh API* (https://www.ohloh.net/api) to acquire data from the projects, focusing then the analysis on a subset of 5 projects.

We considered *OpenOffice.org*, office automation suite, *KDE*, and *Gnome*, window managers, *Wine-project*, a cross-platform implementation of the Microsoft Windows API, and the *Eclipse* platform for integrated software development. We selected these projects, apart for being rather popular, for the fact that are part of the open source projects with large number of source lines of code (SLOCs) and number of yearly contributing developers. Our heuristic for project selection was complemented by the fact that we limited the number of yearly contributors to a maximum of *500* developers. This boundary excluded thus projects such as the *Linux Kernel 2.6*, and the *Android* project that we considered as outliers compared to other open source projects. We then selected specifically projects based on expectations of obtaining interesting and useful results [2]. According to this rationale, we preferred projects not part of the main cluster according to SLOCs and yearly contributors, and with a wide diffusion in the community (Figure 1).

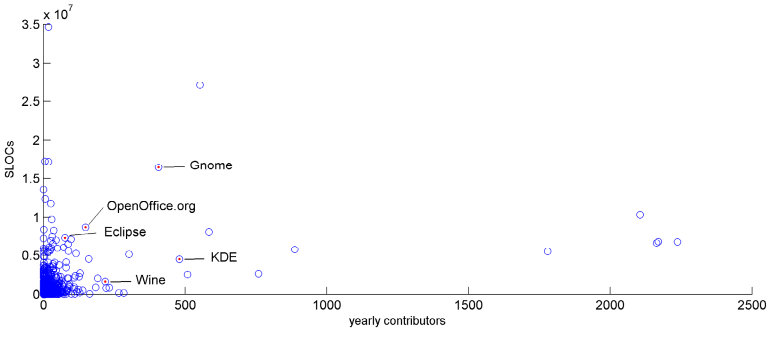

Fig. 1. Projects considered in terms of total number of SLOCs and yearly contributors (9.020 projects from the Ohloh repository

The projects selected range from 76 yearly contributors (*Eclipse*) to 480 (*KDE*), with a size in terms of *SLOCs* from 1,6M (*Wine-Project*) to 16,4M (*Gnome*). Main languages for 4 out of 5 projects are *C* and *C++, with Java* as the main language only for the *Eclipse* project. Information from the *Ohloh* repository encompasses a period from 3 to 15 years (Table 2).

Table 2. Descriptive statistics for the projects

Project	12 months contributors	SLOCs	Main Language	Information start date	Information end date
OpenOffice.org	149	8.721.053	C/C++	2000-07-01	2008-09-01
KDE	480	4.530.775	C	2005-05-01	2008-09-01
Gnome	406	16.467.663	C/C++	1997-01-01	2007-11-01
Wine	218	1.644.154	C	1993-06-01	2008-09-01
Eclipse	76	7.352.744	Java	2001-04-01	2008-09-01

3.1 Process of Data Collection

For the analysis we used 3 different sources of information, respectively for *code activities*, *bug reports*, and *release dates*.

First, we retrieved the activity level for each project; we gathered three indicators of development activities during a time-span of one month:

- code-activities = LOCs added + LOCs removed;
- comments-activities = comments added + comments removed;
- blanks-activities = blanks added + blanks removed;

The rationale behind this choice is that for our analysis we needed a pure indicator of activity inside the projects, not an indicator about the growth of the projects in time. In that case considering only additions of lines would have been more appropriate for the focus of analysis.

Second, we retrieved information about *opened* and *closed* bug reports from the *issue tracking systems* of the selected projects; for each project we evaluated the bug reports that were opened and closed at a certain date, by considering the *closing and opening dates* of bugs tagged as *CLOSED, FIXED,* and not marked as *enhancements*.

Third, to retrieve the release dates of each project, we used mainly the official *website* of each project, and where not sufficient, we relied on the integration from third party sources such as *blogs*, *wikis*, etc..

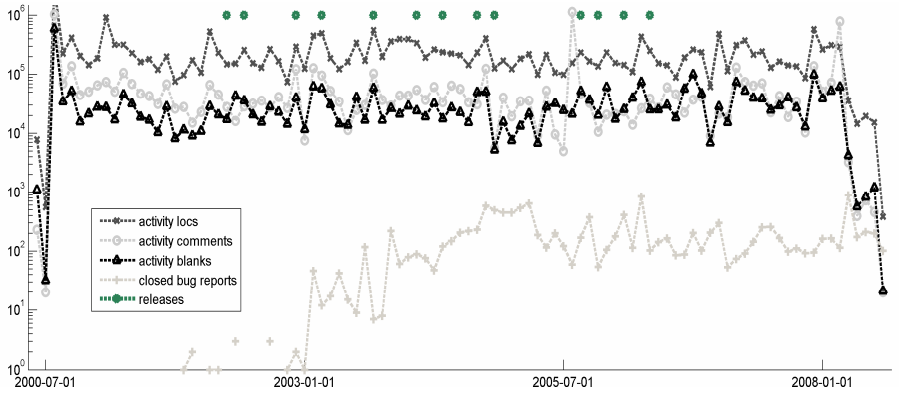

Fig. 2. OpenOffice.org activities, release dates, and bug reports closing dates (log scale)

In Figure 2, we show the typical result of aggregation of the three different data sources, by representing on the same time-line the aggregation of *code-activities* (*code, comments*, and *blanks*), *bug-reports closed*, and *release dates*. The figure refers to the *OpenOffice.org* project.

For the *OpenOffice.org* application, we see a constant trend in all the activities, characterized by some periods where there are bursts, followed by periods of reduced activities. After the first analysis, we decided to drop the indicators *comments-activities*, and *blanks-activities* for the reason that they were highly correlated with the *code-activities* indicator by running non-parametric correlation analysis (Table 3). For this reason in the remaining of this paper we refer to code activity simply as *lines of code added + lines of code removed*.

Aggregated data collected for the projects (Table 4), shows the differences and similarities of the projects in terms of *yearly code activities*, *total commits*, and *total bug reports closed*. The *KDE* project is the monthly most active project (*588 KSLOCs* per month), followed by *Gnome*(*443*), *Eclipse* (*276*), *OpenOffice.org* (*245*), and *Wine* (*35*).

Table 3. Spearman Rank Order correlation between code-activities, comments-activities, and blanks-activities, significant at 0.01 two-tailed

Correlation with Code Activities		
Project	*Code_comments*	*Code_blanks*
OpenOffice.org	0.7147	0.70408
KDE	0.9064	0.9702
Gnome	0.7821	0.8615
Wine	0.8930	0.9286
Eclipse	0.8578	0.8159

Table 4. Aggregated data for projects considered during the analysis

Project	Code Activity (KSLOCs)	Total commits	Total bug reports closed in the period	Number of months
OpenOffice.org	24.284K	168.121	12.692	99
KDE	24.111K	92.433	5.824	41
Gnome	62.475K	258.989	4.812	141
Wine	6.560K	50.148	5.559	185
Eclipse	24.898K	167.893	32.350	90

4 Method of Analysis

Once we gathered all information from the three different sources, we started investigating the relation between *code activities*, *releases*, and *bug-fixing* activities. We used a burst detection technique, the same technique used in [12] to identify similarities between temporal online queries and similar to the one used in [3, 4] to analyze developers' behavior.

The technique identifies in a systematic way peaks in the temporal evolution and compares them with peaks in other time-series. Given a time-series defined as a sequence of time-points $t=(t_1,t_2,\ldots,t_n)$, a burst or peak is defined as a set of points that exceed the observed behavior in other points of the time-series. More formally, the approach is as follows:

1. calculate the moving average MA_l of the time-series, where $l>0$ is the lag of the moving average;
2. calculate the *cut-off* value for a given l as $mean(MA_l)+x*std(MA_l)$; this gives a threshold to use for peak detection. In our case we considered $x=0.5$ as an appropriate value for the detection technique applied to our dataset; we must consider that higher x values will increase the *cut-off* level and thus lead to a detection of only the strongest peaks in the time-series;
3. determine bursts with $MA^i_l>cut\text{-}off$, where i is the time interval considered;
4. project the bursts on a temporal line; this is to identify the time points corresponding to the bursts;
5. compare the overlap of the bursts for the different activities on the temporal line;

Figure 3 illustrates visually the steps *1-4*, by showing a random dataset, the moving average, the cut-off point, and the peaks identified.

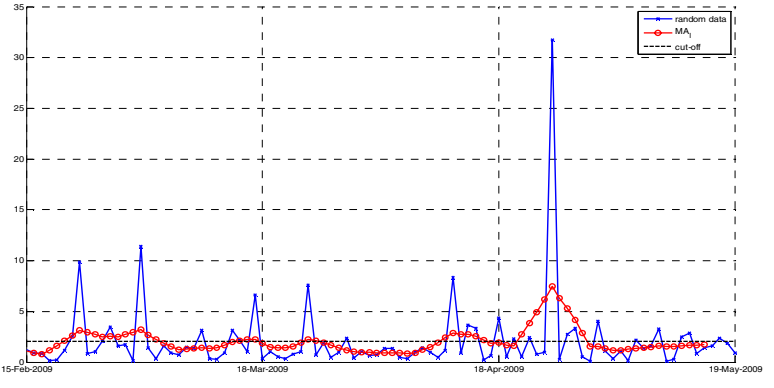

Fig. 3. Burst-detection method. Random dataset with peaks, moving averages, cut-off line, and bursts areas identified

The cut-off line gives an indication of the data points to consider as part of a peak. Once peaks have been identified, they are plotted on a line to ease the comparison with peaks identified in other time series. For this reason we needed then a metric to compare the identified bursts (Figure 4). To compare different regions of activities we defined the following metrics:

- number of peaks identified by the approach, we identify all peaks as t'_i; this gives a first evaluation of the *burstiness* of the time-series;
- number of intersections between peaks, computed as $t'_A \cap t'_B$: gives a raw indication about the intersection of peaks, but it is not a reliable measure as we

can have a hypothetical time-series with peaks that span over the entire period that gets a perfect intersection with any time-series with at least one peak. For this reason we defined the next metric;

- a measure of recall, defined as $\frac{(t'_A \cap t'_B)}{t'_A}$, that gives information about the specific coverage of peaks that intersect between the two time-series;

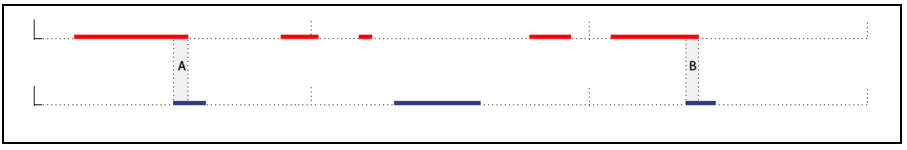

Fig. 4. Overlapping of bursts areas detected between two time-series

The whole process has been implemented as a *Java* application that mines the projects' repository and generates *CSV* files. Such files are then the input of a *Matlab (*http://www.mathworks.com*)* script that handles calculation and plotting of the overlapping regions. Information about version releases and bug reports is still saved manually to *CSV* files and then used as an additional data source for the script.

From this point we will label code-activities as *CA*, the activities of bug reports opening as *BO*, and the bug reports closing activities as *BC*. We will also label the peaks for *opening bugs* activities as t'_{BO}, *closing bugs* activities as t'_{BC}, and *code-activities* as t'_{CA}.

5 Analysis

We run burst detection on the *CA*, *BO*, and *BC* time-series for all projects. We used a window size of $l=8$, as we found heuristically that parameter to work better with our dataset than a shorter window of $l=2$, as peaks identified for *BC* and *CA* were respectively *+25,9%,* and *+25%*. A better fitting was also confirmed visually by inspection of the generated burst regions.

5.1 Bug Reports Opening and Closing versus Code-Activities

First we run a comparison between *CA*, and *BC* (Table5). In 3 out 5 projects, the process has a higher *burstiness* for code-related activities rather than bug-closing activities.

In only 2 projects, *KDE* and *Gnome*, there is an increase in *CA* that seems related to *BC* activities. This specifically means that for these two projects, peaks in code activities are highly correlated to the activities of closing bug reports in the same period.

We run the same comparison, this time taking into account the *BO* activities (Table 6). In two projects, *Gnome*, and *Wine*, peaks in the bug reports opening are related with peaks in code-activities, while for the other projects the behavior is different.

Table 5. Comparison between code activities and bug closing activities, l=8, x=0.5

Project	t'_{BC}	t'_{CA}	$\frac{(t'_{BC} \cap t'_{CA})}{t'_{BC}}$
OpenOffice.org	28	20	0.00
KDE	8	12	0.62
Gnome	8	57	0.87
Wine	9	46	0.0
Eclipse	34	25	0.0

Table 6. Comparison between code activities and bug opening activities, l=8, x=0.5

Project	t'_{BO}	t'_{CA}	$\frac{(t'_{BO} \cap t'_{CA})}{t'_{BO}}$
OpenOffice.org	23	20	0.09
KDE	11	12	0.09
Gnome	29	57	0.86
Wine	40	46	0.45
Eclipse	35	25	0.0

According to the relation between code, and bug reports activities, we can categorize the projects in 4 categories: in the first category there is no apparent connection in bursts between time-series *BC-CA*, and *BO-CA* (*OpenOffice.org, Eclipse*), in second category there is a relation in bursts *BC-CA* (*KDE*), in the third category there is a relation *BO-CA* (*Wine*) and in the final category there is a strong relation in bursts between time-series *BC-CA*, and *BO-CA(Gnome).*

5.2 Software Releases versus Code, Bug-Closing, Bug-Opening Activities

As a next step, we compared software releases with *CA*, *BO*, and *BC* activities (Table 7). We wanted to evaluate the behavior of developers in proximity of software releases. In all projects, and with different degrees, releases are related to increases in *bug reports opening*, *bug-closing*, and to a minor extent to *code-activities*.

Bug reports opening activities are related to new version releases: as a new version is released there are peaks in the generation of bug reports. Especially for the *Wine* and *Eclipse* projects this effect seems relevant.

Also *bug closing* activities are stronger in presence or in proximity of a *release date*. This effect seems particularly strong for the *Eclipse* project.

Comparing the two effects, we can state that in presence of a release, *bug opening* activities are more bursty than bug *closing activities*, this can be an indication that while *bug closing* activities are more gradual in time, *bug opening* activities are more subject to bursts at a software release date.

Peaks in *code activities* are also correlated to the proximity of a release date. This means that in proximity of a release date there is a burst in code development activities. The *Wine* project, and, again the *Eclipse* project do not follow this behavior.

Table 7. Comparison of software releases with bug reports closing activities and code activities, *l=8*, *x=0.5*

Project	$\frac{R \cap t'_{BO}}{R}$	$\frac{R \cap t'_{BC}}{R}$	$\frac{R \cap t'_{CA}}{R}$
OpenOffice.org	0.38	0.31	0.23
KDE	0.33	0.33	0.33
Gnome	0.5	0.06	0.5
Wine	1.0	0.22	0.09
Eclipse	1.0	1.0	0.0

A question that still remains open is how much skewed are the peaks between code activities and bug closing activities. One approach would be to consider the distance between *lagged time series*, but this would compare all periods without focusing on the peaks. Remaining in the context of our approach, it would mean to find *k*, periods of lags, such that $Max(t_{Ak} \cap t_B)$, with time-series lagging as validation. We leave this step as future work, as for the significance of the result we need a finer granularity of data analysis based on data points on a daily scale.

6 Discussion

Falling back to our initial research hypotheses we can state the following:

H1. There is an increase in code-related activities as there is an increase in the creation of bug-reports; we did not get enough evidence to support this hypothesis, for only *two* projects out of *five* we derived some evidence of a relation of code activities and bug reports creation; according to our rationale, this means that the projects do not adapt quickly to the new reports that are issued;

H2. There is an increase in code-related activities as there is an increase in the closed bug-reports: also in this case, we could not find evidence to support the hypothesis. Only for *two* out of *five* projects there is indication that in coincidence with peak code activities there is a peak activity in the closed bug reports;

H3. There is an increase in bug opening activities in the proximity of a software release date: we report that this hypothesis is supported by all the projects, with *two* out of *five* projects where the behavior is particularly evident;

H4. There is an increase in bug closing activities in the proximity of a software release date: we can state that for mostly of the projects analyzed there are bursts of bug closing activities in coincidence of software releases;

H5. There is an increase in code-related activities in the proximity of a software release: for *three* out of *five* projects this holds, there is a more or less limited coincidence of peaks in code activities with software releases;

In accordance with our initial observation about many and contrasting empirical results about the speed of the development process to adhere to bug-fixing requests, a generalization of results across projects is difficult to obtain (Table 8).

We see that each application has a different pattern in answering our hypotheses, we suspect that the reason is due to the different type of application and domain.

Our interpretation of the general findings is that open source projects examined are subject to limited peaks in code development activities during the early phases of the bug reporting process (*H1*). As soon as an user/developer issues a bug report, the activity starts to be frenetic in order to solve the issue. Solving many issues is then incremental: there are limited bursts in activities as the bug reports are closed (*H2*). The fixing of defects in code is a process that is distributed in time.

From another viewpoint, release dates are connected with peaks in the bug opening process and the closing process (*H3/H4*): approaching a software release, bug reports will be closed with a bursting activity, as well with a new release the users will tend to increase their reporting activities. Confirming *H1*, bursts of code activities are not detected – or slightly detected - when there is a code release (*H5*). This seems to confirm a more distributed effort during the development process.

Table 8. Comparison of projects according to hypothesis of higher development speed (+ supports the hypothesis, - is against the research hypothesis)

Project	H1	H2	H3	H4	H5
OpenOffice.org	-	-	+	+	+
KDE	-	+	+	+	+
Gnome	+	+	+	-	+
Wine	+	-	++	+	-
Eclipse	-	-	++	++	-

7 Limitations and Future Works

Limitations of current work are threefold:

- the major limitation is the monthly granularity of the time series considered, a finer granularity is opportune in consideration of the technique used;
- another limitation is the restricted number of projects analyzed, although we considered relevant projects. We plan to extend the analysis to a larger number of projects;
- we did not consider in this work intensity of peaks. Adding this information, once normalized, to the calculation of peaks' distance gives more information about the actual similarities of peaks;

Future work will go in the direction of addressing these limitations, in particular extending the current analysis to other projects, and, more important, to collect this information throughout single projects data-mining. Results from the queries of the *Ohloh* repository will be used as a relevant source for support and validation. Moreover, other aspects such as the impact of the development of open source components that are the basis for the development of a complex system and the techniques used to improve their quality through an open testing process [7] can be investigated.

8 Conclusions

The open source development methodology is considered to be highly efficient in the bug fixing process and to propose generally quick version release cycles. We proposed an empirical investigation of this assertion studying 5 large and well-known open source projects by studying temporal evolution of *source code activity*, *issue tracking repositories activities*, and *release dates*. By using temporal burst-detection to evaluate peaks in time-series, we compared the periods of highest activity in the different time-series.

We found that peaks or bursts in *code activities* are not related to peaks in *bug reports closing activities* inside *issue tracking systems*, instead we found peaks in *bug reports opening/closing* to be synchronized with version release cycles. *Code activities* seem more distributed across version releases.

Our conclusions show that the open source development methodology quickly adapts to the changing environment, but also that such velocity depends on the application and specifically on the projects' domain.

References

1. Challet, D., Du, Y.L.: Closed Source versus Open Source in a Model of Software Bug Dynamics, Cond-Mat/0306511 (June 2003), http://arxiv.org/pdf/cond-mat/0306511
2. Christensen, C.M.: The ongoing process of building a theory of disruption. Journal of Product Innovation Management 23(1), 39–55 (2006)
3. Coman, I., Sillitti, A.: An Empirical Exploratory Study on Inferring Developers' Activities from Low-Level Data. In: Proceedings of 19th International Conference on Software Engineering and Knowledge Engineering (SEKE 2007), Boston, MA, USA, July 9-11 (2007)
4. Coman, I., Sillitti, A.: Automated Identification of Tasks in Development Sessions. In: Proceedings of 16th IEEE International Conference on Program Comprehension (ICPC 2008), Amsterdam, The Netherlands, June 10-13 (2008)
5. Dinh-Trong, T., Bieman, J.: Open source software development: a case study of FreeBSD. In: Proceedings of 10th International Symposium on Software Metrics, pp. 96–105 (2004)
6. Feller, J., Fitzgerald, B.: Understanding Open Source Software Development. Addison-Wesley Professional, Reading (2001)
7. Gross, H.G., Melideo, M., Sillitti, A.: Self Certification and Trust in Component Procurement. Journal of Science of Computer Programming 56, 141–156 (2005)
8. Kuan, J.: Open Source Software as Lead User's Make or Buy Decision: a Study of Open and Closed Source Quality. Stanford University (2002)
9. Mockus, A., Fielding, R., Herbsleb, J.: A case study of open source software development: the Apache server. In: Proceedings of the 22nd international conference on Software Engineering, pp. 263–272. ACM, Limerick (2000)
10. Mockus, A., Fielding, R., Herbsleb, J.: Two case studies of open source software development: Apache and Mozilla. ACM Trans. Softw. Eng. Methodol. 11, 309–346 (2002)
11. Scacchi, W.: Is open source software development faster, better, and cheaper than software engineering. In: 2nd ICSE Workshop on Open Source Software Engineering (2002)
12. Vlachos, M., Meek, C., Vagena, Z., Gunopulos, D.: Identifying similarities, periodicities and bursts for online search queries. In: Proceedings of the 2004 ACM SIGMOD international conference on Management of data, pp. 131–142. ACM, Paris (2004)
13. Weinstock, C.B., Hissam, S.A.: Making Lightning Strike Twice? In: Feller, J., Fitzgerald, B., Hissam, S., Lakhani, K. (eds.) Perspectives on Free and Open Source Software, pp. 93–106. MIT Press, Cambridge (2005)