Reassessing Brooks' Law For The Free Software Community

Andrea Capiluppi and Paul J. Adams
Centre of Research on Open Source Software
University of Lincoln,
Brayford Campus,
Lincoln,
LN5 7TS,
United Kingdom
{acapiluppi, padams}@hemswell.lincoln.ac.uk

Abstract. Proponents of Free Software have argued that some of the most established software engineering principles do not fully apply when considered in an open, distributed approach. Among these principles, "Brooks' Law" has been questioned in the Free Software context: large teams of developers, contrary to the law, will not need an increasingly growing number of communication channels. As advocates claim, this is due to the internal characteristics of the Free Software process: the high modularity of the code helps developers to work on comparted sections, without the need to coordinate with all other contriutors.
This paper examines Brooks' Law in a Free Software context, and it studies the interaction of contributors to a large Free Software project, KDE. The network of interactions is analyzed and a summary term, the "compaction", is dynamically evaluated to test how the coordination mechanism evolves over time in the project. This paper argues that the claim of advocates holds true, but with limitations: in the KDE project, the few initial developers needed a significant amount of communication. The growth of KDE brought the need to break the number of overall communication channels to a significant extent. Finally, an established amount of 300 developers currently needs the same amount of communication as when the developers were only 10. We interpret this result by arguing that Brooks' Law holds true among the core developers of any large Free Software project.

1 Introduction and Related Work

In the last decade, the Free/Libre and Open Source Software (FLOSS) development approach has attracted vast and diverse interested audiences, namely software practitioners, software engineering researchers and, in the large, the end users of software systems. Each of these cohorts has brought to attention one (or several) aspects of this approach [26], criticized [10] or advocated it [22], and, at times, compared it with established approaches [1, 30].

At first, the traditional, closed-source development has been questioned in light of this new paradigm: specifically, it was studied whether FLOSS systems show higher quality characteristics than closed source software [28]. Also, in terms of its evolutionary patterns, research papers have reported on whether the average FLOSS project

could be comparable to a traditional software system [5, 13, 11]. It has also been argued that closed- and open-source paradigms represent just the extremes of the software management spectrum, but several other flavors should be also considered (*e.g.*, commercial open-source, community open-source, among others, [7]), which has then posed the basis for the comparison of these hybrid systems with traditional software [21].

More recently, researchers have started analyzing the fundamentals differences between the established software engineering approaches and the FLOSS development style, comparing it both with the traditional development style (*i.e.*, composed of the phases of requirements, design, implementation, testing and maintenance [23]), and finding differences with the development cycles ([24, 6]), and finding similarities with the Agile paradigm [17, 1].

As a further and one of the most serious challenges that the FLOSS approach still has to face is the issue of trust among end-users: specifically, on one side, the internal product and process attributes of the FLOSS have been analyzed to establish its dependability, reliability and overall quality as an alternative to commercial software, both in research papers [33] and in pan-European projects[1]. On the other side, the chance for end-users to benefit proper, centralized support for software created by the community at large should be assessed [19, 3].

In all the above related works on the matter (but specifically for enabling the trust by users), it is absolutely necessary that research studies on FLOSS systems are performed rigorously, with a strong emphasis on the repeatability of experiments, on the rigorous empirical approach in the collection of data, its parsing and the presentation of results, and finally by illustrating any threats to validity of which assumptions were made throughout the experiment. Specifically, comparisons between the termed closed- and open-source approaches should be based on sound empirical basis, avoiding the pitfalls of personal opinions, strong advocacy and the such.

1.1 Brooks Law and FLOSS Systems

Among the comparisons between closed- and open-source processes, much attention has lately been given to the applicability of the so-called *Brooks' Law* to Free Software projects. Within his famous work, "The Mythical Man Month", Fred Brooks introduced a simple premise, that, oversimplifying as he put it, states [4]:

Adding manpower to a late project makes it later

This is, by Brooks' own admission, not a universally applicable premise: still, when a comparison was to be made, emphasis should be given to the empirical analysis of software artefacts, and the approach validated and pinned with validity threats, in order to draw any conclusion.

On a theoretical – and logical – basis, several contributions have underpinned the assumption that Brooks' Law will not apply to FLOSS projects [25, 15, 14]. One of

[1] SQO-OSS: Software Quality Observatory for Open Source Software, `www.sqo-oss.eu`

the preferred arguments to justify this position is based on the concept of "egoless programming": when developers are not *territorial* about their code, but instead they encourage others to improve the systems, improvement is much more likely, widespread and faster [32].

Some research studies have used rather simplistic approaches to validate the above assumption, by for instance using the number of developers and an operationalization of a project's status as a proxy of the complexity of interrelations [27]. One of these studies has shown that in the Sourceforge[2] dataset, the correlation between the "output per person" and "number of active programmers" was very small, It was concluded that, also given to the strict modularization, Brooks' Law does not apply (at least) to the SourceForge data set [18].

Other researchers have maintained a more careful profile: given the assumption that Brooks' Law can be hardly transcended, and will apply anyway to software (and other) collaborating teams, a more appropriate research question is how to distinguish between a core team (where the Brooks' Law does and will definitely apply) and the larger cohort of contributors. On these fringes, in fact, obscure bugs are tracked and fixed, and, more importantly, radical experimentations are implemented and tested in parallel with the main, more important features: in few words, the *core* team can focus on the *core* work [12]. The real goal is therefore how to *diminish* the effects of Brooks Law and have a small improvement of the overall software development process [31].

The paper is articulated as follows: section 2 will introduce the working hypothesis, based on the formulation of Brooks' Law, which has been restated in order to check it empirically. Section 3 will summarize the empirical method and how the metrics were collected and operationalized, while section 4 will show the results of the experiment when applying Brooks' Law to a large collection of related FLOSS projects. Finally, section 5 will identify the main threats to the external and internal validity of the proposed empirical experiment, and section 6 will conclude.

2 Working Hypothesis

Analyzing the thinking behind Brooks' Law, two critical observations can be derived, and potentially operationalized with research questions, metrics and verifiable tests:

- **Issues of communication** – The communication paths between newly joining and existing software developers (hence, not specifically to FLOSS projects) become increasingly complex, creating a bearing overhead; in fact, the number of communication links grows as a square of the number of developers in the project. For a project of n developers, the total number of communication links between them will be $n^2 - n$.
- **Issues of early productivity** – Within a generic software process (again, not specifically to FLOSS projects), new developers, no matter how competent, are not able to work fully productively when they join a new project. Factors such as a need to get to know the team, the underlying process and its codebase all contribute to this.

[2] `http://sourceforge.net`

The focus of this research is on the first point, and it will be tested whether it is generally true or not for FLOSS projects. In addition, the reasoning behind [12] and [31] will be tested too: it will be in fact studied whether the communication paths keep their weight among the core developers of a FLOSS project. For the purpose of creating a strict bound on this research, the second of these points shall be disregarded and proposed as further work, both for FLOSS and traditional software engineering projects.

The reasons for a particular focus given to Free Software development in this research are twofold. Firstly, the manner of Free Software development is often perceived as being radically at odds with conventional wisdom in software engineering practice [29]. Secondly, because of the open nature of Free Software development, a wealth of development-related data is available with which empirical analysis of this aspect of Brooks' Law can be conducted.

3 Method and Operationalization

In the following, an overview of the definitions and the methods used to extract or evaluate the metrics used throughout this paper is reported.

3.1 Source Code Management Systems

Free Software developers have many mediums of communication: email (direct and by list), chat systems, comments in bug tracking systems and comments in source code management systems, for example. Aggregating all of this data over the lifetime of any particular project can be problematic, so this research focuses purely on source code management (SCM).

SCM systems are used by developers to keep a revision history for the software artefacts being produced. In particular, these systems also handle the merging of contributions from different developers working concurrently on the same artefact. As developers make contributions to these artefacts they are usually required by the SCM to leave a short message describing the nature of and rationale behind their commit. This is simply known as the "comment".

In addition to the comment, the SCM will automatically record the developer's username, date and time of commit, the unique file path of the files being modified and, for each path, a note of the nature of the amendment (file added, modified, moved or removed). It is *this* data that is utilized within this research. The Subversion SCM can expose this data in the form of XML and therefore ease the automation of log processing.

3.2 Communication Paths

Given the total number of developers, potentially each one could share the work on some artefact (code, or other) with several other developers. Each one-to-one interaction is here named a communication link: the maximum number of links, given a

number n of developers, is $n^2 - n$, as mentioned above. This value is clearly an upper bound, and summarizes a completely loose approach of collaboration, where each node (*i.e.*, developer) has an interaction with every other node. As it is unlikely that each developer is linked to every other developer extra links may be required to conect one developer to another, via intermediate contributors. A *communication path* is a set of links connecting one developer to another. A communication path may only be one link long; when two developers are directly linked.

3.3 Community Graphing

As previously mentioned, the number of *potential* communication links grows exponentially as new developers join a team. It can be perceived, however, that only for small teams working on small projects is it important that all of these links are utilized. For larger projects with a larger number of developers it may not be required to make use of all these communication linkss. The first stage of this research introduces a method for graphing the communication links and paths exposed by the SCM for a given project. The Community Graph is a simple, undirected, weighted graph in which nodes represent accounts with the SCM system and edges represent a relationship between the nodes where this relationship may be interpreted as "has worked on the same artefact as". The weight of the edge is the count of artefacts on which the pair of SCM accounts has collaborated. A similar approach to community network analysis was taken by Martinez-Romo et al [20].

Figure 1 shows the Community Graph for the KDE Marble[3] project. The graph has been laid out using the Kamada-Kawai algorithm [16]. Within this particular implementation of that algorithm, edges of higher weight are made shorter. The result, here, being that SCM accounts which have worked together frequently, will appear closer together. One of the consequences of this is that the graph representation reveals distinct, concentric "levels" of contribution. These are similar to the central levels of contribution as hypothesized by Crowston et al [8].

3.4 Graph Processing

Having produced a graph representing all the utilized communication paths amongst developers in a project, a method is required by which we can measure how compact the communication is. In order to achieve this, a slightly modified version of the Floyd–Warshall algorithm [9] for finding the shortest paths between all pairs of nodes is utilized. As described above, the weight of edges within the graphs created as part of this research increase as developers work more with each other on the same resource. It is, therefore, important to establish the *shortest* path between nodes in order to measure how compact the communication within a project is. Of course, given that edge weight denote how often two developers have worked together, higher path weights are desirable.

[3] KDE Marble: `http://edu.kde.org/marble/`

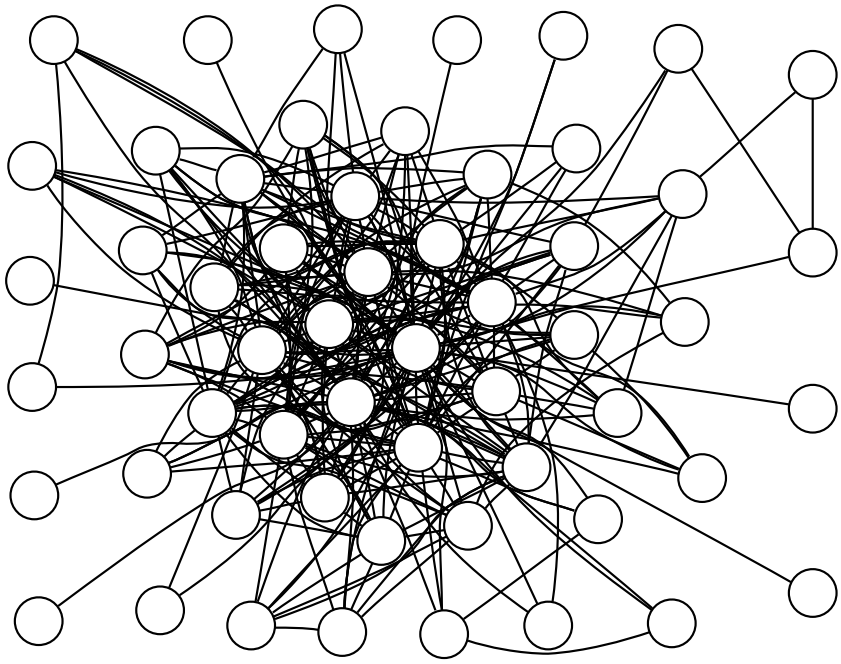

Fig. 1. Community Graph for KDE Marble (edge weights removed)

In order to establish how growth within a project affects the communication complexity, community graphs can be produced and, for each, it is possible to measure the number of developers and mean weight of the longest paths. These two measurements can then be plotted against each other in order to establish any trends. In order to do so, the average weight of the path between nodes in the communication graph will be defined here as the "compaction" of a community graph

3.5 Project Selection

In this research analysis was conducted for the entire KDE[4] project. KDE was specifically chosen for this research as it has previously been shown to be a project with more than 10 years of development, a very large and active community [2] producing good quality code [1]. It is therefore reasonable to interpret KDE as a successful project.

[4] KDE Project: `http://www.kde.org/`

4 Results

For the whole KDE project, the number of developers within the project is plotted against the average weight of path between nodes in the communication graph (known, now, as the "compaction") for each fortnight over the lifetime of the project. The higher the compaction, the stronger the communication paths. Figure 2 shows the results obtained for the KDE project. The results shown represent the 262 fortnights from 27 April, 1997 until the fortnight starting 29 April, 2007.

These results show us that whilst the number of active developers is relative small (fewer than 10) some of the project's higher compaction scores (values greater than 10) are being achieved. This certainly follows Brooks' Law and conventional wisdom in both the process-driven and Agile communities. This confirms what reported by earlier works [31, 12]: small FLOSS teams behave as traditional software engineering teams, and the need for communication among core developers follows almost inevitably the stated Brooks' Law.

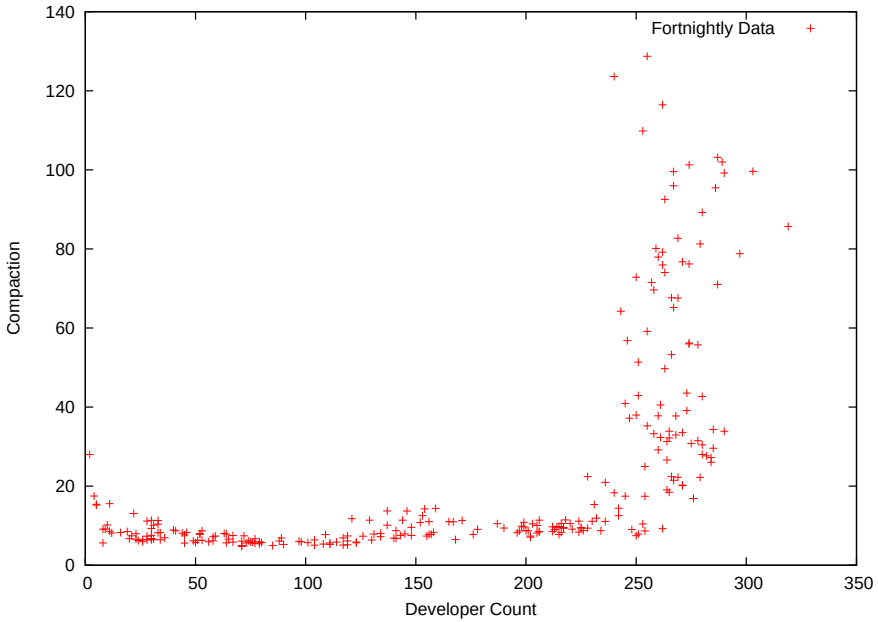

Fig. 2. Communication Compaction in KDE

As the KDE project has grown it has, as expected, seen a dilution of its communication paths. Efficient management of the community and regular restructuring of the project into teams has ensured that whilst KDE was between 50 and 230 developers in size, communication was still half as compact as it was for around 10 developers.

The most striking aspect of the KDE results is revealed as the project grows above 250 developers in size. At this point a critical mass is achieved allowing communication to become, at times, six times more compact as when the project had only 10 developers or fewer. The most likely explanation for this phenomenon is that certain developers create strong links between teams within the KDE project by contributing to more than one part of the overall project.

5 Threats To Validity

In assessing Brooks' Law within the Free Software community the thoroughness of this research may be challenged in two clear manners.

1. Internal validity – this research has not addressed the "ramp up" time for new developers, as mentioned in Section 2. It is reasonable to assume that new developers, when joining a new project, will start with a lower productivity than more established and skilled developers. Albeit this could minimally affect the above results in terms of evolving community graphs, one of the lemmas of Brooks' Law could be severely affected.
2. Construct validity – although this research has aimed to produce an accurate picture of the *actual* communication paths through a project, only one type of data source has been used: email, chat and bug tracking have all been disregarded. Again, this is a serious threat especially when dealing with the analysis of new developers joining a project: it is arguable that their contribution starts at first by being acquainted with the existing developers on the mailing lists, or by submitting bug reports at first. The actual code contribution could be in some ways correlated with the duration of this ramp-up period.

6 Conclusions and Further Work

This paper has proposed an empirical evaluation of a general principle of software engineering (the so-called Brooks' Law) in the context of Free Software. Several earlier works have hindered the possibility that this principle, when applied in the distributed and open approach as advocated by the FLOSS proponents, will not apply as in traditional software engineering. The contribution that this work aimed to establish was a level of confidence in the conclusions by means of an empirical evaluation.

Using well known attributes of the graph theory (also applied by the earlier *community network analysis* [20]), this paper formulated a research hypothesis: the paths of communication among developers in FLOSS projects will be less than the theoretical ones ($n^2 - n$, when n developers are present in the project). According to a previous contribution [12], this paper refined the hypothesis, claiming that the communication overheads apply to FLOSS projects too, but only within the core team of developers.

At first, the community graph of a large FLOSS system (the KDE project) was extracted both for the overall system, and for each subproject contained within it. A well

known characteristics of graphs (*i.e.*, the compaction, or the average weight of path between nodes) was then studied for the overall system, and plotted in its evolutionary trend. Three observations could be drawn from this visualization: at first, it is visible that in the earlier inception of this project, few developers achieved a high compaction, following the needs to interact more within the core team. It was established that teams of around 10 FLOSS developers, similarly to any traditional or Agile software team, will incur inevitably in the overheads of Brooks' Law.

The second observation was based when the project started growing steadily: efficient restructuring of the code and modularization in several subprojects achieved a sustainable lower compaction than its initial phase, to a minimum of one third of its original value. The third observation was finally made in the latest stage: as a mature project, the number of active developers has reached more than 300, but the compaction is still the same as when the developers were only 10. This is a further indication that was postulated in the past for FLOSS systems applies for the applicability of Brooks' law: larger FLOSS projects will not follow at large this law, but only a subset of core developers will be in need of a constant communication.

Two main works will be conducted in the future: at first, these results will be compared with another, less organised repository of FLOSS projects: a sample of projects from SourceForge will be analyzed, its compaction evaluated and compared to the KDE results. Our working hypothesis will state that large SourceForge projects will display a compaction pattern similar to the found one. The second strand we'll pursue is the enlargement of this research to the second aspect of Brooks' Law: it will be therefore studied whether FLOSS developers have a "ramp-up" period, where their contribution levels are lower but their individual compaction is relatively higher than that of established developers.

Acknowledgements This work was partially supported by the European Community's Sixth Framework Programme under the contract IST-2005-033331 "Software Quality Observatory for Open Source Software (SQO-OSS)".

References

1. Adams, P.J., Capiluppi, A.: Bridging the gap between agile and free software approaches: The impact of sprinting. International Journal of Open Source Software and Processes **1**, 58–71 (2009)
2. Adams, P.J., Capiluppi, A., de Groot, A.: Detecting agility of open source projects through developer engagement. In: Proceedings of the Fourth International Conference on Open Source Systems. Open Source Development, Communities and Quality (2008)
3. Bonaccorsi, A., Rossi, C.: Why open source software can succeed. Research Policy **32**(7), 1243–1258 (2003). DOI 10.1016/S0048-7333(03)00051-9
4. Brooks, F.P.: The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition. Addison-Wesley Professional (1995)
5. Capiluppi, A.: Models for the evolution of OS projects. In: Proceedings of ICSM 2003, pp. 65–74. IEEE, Amsterdam, Netherlands (2003)

6. Capiluppi, A., González-Barahona, J.M., Herraiz, I., Robles, G.: Adapting the "staged model for software evolution" to free/libre/open source software. In: IWPSE '07: Ninth international workshop on Principles of software evolution, pp. 79–82. ACM, New York, NY, USA (2007). DOI 10.1145/1294948.1294968
7. Capra, E., Wasserman, A.: A framework for evaluating managerial styles in open source projects. In: B. Russo, E. Damiani, S. Hissam, B. Lundell, G. Succi (eds.) Open Source Development, Communities and Quality, pp. 1–14. IFIP International Federation for Information Processing (2008)
8. Crowston, K., Annabi, H., Howison, J., Masango, C.: Effective work practices for software engineering: free/libre open source software development. In: Proceedings of the ACM workshop on Interdisciplinary software engineering research (2004)
9. Floyd, R.W.: Algorithm 97: Shortest path. Communications of the ACM **6**, 345 (1962)
10. Fuggetta, A.: Controversy corner: open source software-an evaluation. J. Syst. Softw. **66**(1), 77–90 (2003). DOI 10.1016/S0164-1212(02)00065-1
11. Godfrey, M.W., Tu, Q.: Eovlution in open source software: A case study. In: Proeedings of 16th IEEE Int. Conf. on Software Maintenance, pp. 131–142 (2000)
12. Goldman, R., Gabriel, R.: Innovation Happens Elsewhere: How and Why a Company Should Participate in Open Source. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2004)
13. Herraiz, I., González-Barahona, J.M., Robles, G.: Determinism and evolution. In: A.E. Hassan, M. Lanza, M.W. Godfrey (eds.) Mining Software Repositories, pp. 1–10. ACM (2008). DOI 10.1145/1370750.1370752
14. Johnson, J.P.: Economics of open source software (2001). URL `http://opensource.mit.edu/papers/johnsonopensource.pdf`
15. Jones, P.: Brooks' law and open source: The more the merrier? IBM DeveloperWorks (2002). URL `http://www-106.ibm.com/developerworks/library/merrier.html`
16. Kamada, T., Kawai, S.: An algorithm for drawing general undirected graphs. Information Processing Letters **31**, 7–15 (1989)
17. Koch, S.: Agile principles and open source software development: A theoretical and empirical discussion. In: Extreme Programming and Agile Processes in Software Engineering: Proceedings of the 5th International Conference XP 2004, no. 3092 in Lecture Notes in Computer Science (LNCS), pp. 85–93. Springer Verlag (2004)
18. Koch, S.: Profiling an open source project ecology and its programmers. Electronic Markets **14**(2), 77–88 (2004)
19. Lakhani, K.R., von Hippel, E.: How open source software works: "free" user-to-user assistance. Research Policy **32**(6), 923–943 (2003). DOI 10.1016/S0048-7333(02)00095-1
20. Martinez-Romo, J., Robles, G., Gonzalez-Barahona, J.M., Otuño-Perez, M.: Using social network analysis to study collaboration between a floss community and a company. In: Proceedings of the Fourth International Conference on Open Source Systems. Open Source Development, Communities and Quality (2008)
21. Mens, T., Fernandez-Ramil, J., Degrandsart, S.: The evolution of Eclipse. In: Proc. International Conference on Software Maintance (ICSM). IEEE Computer Society Press (2008)
22. Perens, B.: The emerging economic paradigm of open source. First Monday **10**(SI-2) (2005)
23. Pressman, R.S.: Software engineering: a practitioner's approach (2nd ed.). McGraw-Hill, Inc., New York, NY, USA (1986)
24. Rajlich, V.T., Bennett, K.H.: A staged model for the software lifecycle. IEEE Computer pp. 66–71 (2000)
25. Raymond, E.S.: The Cathedral and the Bazaar. O'Reilly & Associates, Sebastopol, CA, USA (1999)

26. Scacchi, W.: Understanding Open-Source Software Evolution, pp. 181–205. Wiley (2006)
27. Schweik, C.M., English, R.C., Kitsing, M., Haire, S.: Brooks' versus linus' law: an empirical test of open source projects. In: dg.o '08: Proceedings of the 2008 international conference on Digital government research, pp. 423–424. Digital Government Society of North America (2008)
28. Stamelos, I., Angelis, L., Oikonomou, A., Bleris, G.L.: Code quality analysis in open source software development. Information Systems Journal **12**(1), 43–60 (2002). DOI 10.1046/j.1365-2575.2002.00117.x
29. Succi, G., Russo, B., Weiss, D., Hapke, M.: D3.1 analysis of the mutual relationships between libre software development and development done using agile methods (2006). URL `http://bl.ul.ie/calibre/deliverables/D3.1.pdf`
30. Tirole, J., Lerner, J.: The Simple Economics of Open Source. SSRN eLibrary (2000). DOI 10.2139/ssrn.224008
31. Weber, S.: Why open source works. Ubiquity **5**(11), 1–1 (2004). DOI 10.1145/991104.991105
32. Weinberg, G.M.: The Psychology of Computer Programming. John Wiley & Sons, Inc., New York, NY, USA (1985)
33. Zhao, L., Elbaum, S.: Quality assurance under the open source development model. Journal of Systems and Software **66**(1), 65–75 (2003). DOI 10.1016/S0164-1212(02)00064-X