The Open Source Way of Working: A New Paradigm for the Division of Labour in Software Development?

Juan Mateos Garcia

W. Edward Steinmueller

SPRU – Science and Technology Policy Research

University of Sussex

INK Research Working Paper No. 1

January 2003

Abstract

The interest the Open Source Software Development Model has recently raised amongst social scientists has resulted in an accumulation of relevant research concerned with explaining and describing the motivations of Open Source developers and the advantages the Open Source methodology has over traditional proprietary software development models. However, existing literature has often examined the Open Source phenomenon from an excessively abstract and idealised perspective of the common interests of open source developers, therefore neglecting the very important organisational and institutional aspects of communities of individuals that may, in fact, have diverse interests and motivations. It is the aim of this paper to begin remedying this shortcoming by analysing the sources of authority in Open Source projects and the hierarchical structures according to which this authority is organised and distributed inside them. In order to do so, a theoretical framework based on empirical evidence extracted from a variety of projects is built, its main concerns being the description and explanation of recruitment, enculturation, promotion and conflict resolution dynamics present in Open Source projects. The paper argues that 'distributed authority' is a principal means employed by such communities to increase stability, diminish the severity and scope of conflicts over technical direction, and ease the problems of assessing the quality of contributions. The paper also argues that distributed authority is principally derived from interpersonal interaction and the construction of trust between individuals drawn to the project by diverse interests that are mediated and moderated through participants' common interest in the project's successful outcome. The paper presents several conclusions concerning the governance of open source communities and priorities for future research.

Keywords: open source software, hierarchies, trust, teams, co-operation.

Acknowledgements:
This research project has been funded under National Science Foundation Grant NSF IIS-0112962, for the period 15 August 2001 – 31 July 2003. The able editorial assistance of Cynthia Little is also gratefully acknowledged.

1 Introduction

Our purpose in this paper is to investigate claims about the effectiveness of the methods employed by 'open source' software development communities in organising the efforts of skilled individuals to produce useful software. It is claimed that these methods support the creation of complex and successful products which, compared to their commercial rivals, offer preferred features (e.g. Apache Web server software), are more reliable in certain applications (e.g. Linux), and are not burdened by the enormous development investments of producing a 'first copy.' In the language of the economics of technology, it is hypothesised that open source methods are a radical process innovation for producing software. This working paper is an assessment of the validity and implications of this hypothesis.

Assessment of innovations at an early stage in their history is a hazardous activity. During the infancy of an innovation, neither the magnitude of its effects nor the domain of its primary application is likely to be easily specified (Rosenberg 1976). Moreover, history offers numerous examples of innovations thought to be of great consequence whose principles of operation proved to be either flawed or fraudulent. In the case of open source software this possibility cannot be entirely ignored. The Microsoft Corporation's rapid ascent to the highest ranks of corporations has been extraordinary. A consequence of Microsoft's ascent, however, has been the perceived foreclosure of opportunity for many talented software designers and engineers to realise their visions.[1] Microsoft's efforts are part of a larger expansion of proprietary software over open source. As Bruce Kogut notes:

> It is fundamental in understanding the origins of open source to acknowledge the deep hostility of programmers to the privatization of software. Early software, because it was developed by monopolies such as telecommunication companies, was created in open-source environments and freely disseminated. The creators of these programs were well known in the software community. They wrote manuals, appeared at conferences, and offered help. (Kogut and Metiu 2001)

That the foreclosure of such roles should produce a 'resistance' movement is not surprising. That this resistance movement appears to have created a 'branching' point in the organisation and division of labour in software production is far more striking.[2]

---

[1] In the broader social context, this perception is contestable. By creating a widely accepted operating system platform, Microsoft has contributed to the very broad diffusion of personal computer technology and thus opened numerous opportunities for many types of software and database production. This broader social observation, however, is consistent with a foreclosure of opportunities for individuals to produce certain types of software such as operating systems. Bezroukov (1999) argues that this anti-Microsoft feeling is one of the main motivations underpinning the open source movement.

[2] The reference to 'branching' is used to indicate the possibility of alternative historical developments. Such developments may also have the feature of 'path-dependence,' the possibility that a historical process may be non-reversible and non-ergodic (i.e. the possibility that alternative 'branches' may become unreachable by available change processes such as rational choice) (David 1993). We do not establish here whether the open source paradigm has path-dependent features, as this would require more evidence concerning the dynamics of open source diffusion in use than has become available so far. We do note, however, that the Linux-Apache linkage may provide a fruitful single case for examining this possibility.

The strength of this 'branch' in the history of software creation and use will rely on three interdependent features of the open source 'paradigm' of software production. The first feature involves the process of recruiting and sustaining membership and participation in open source development activities. The surge in membership and participation that might accompany initially successful skirmishes must be sustained and strengthened as the movement extends it reach and accommodates the diversity required for its broader success. The mechanisms for doing this are examined in Sections 2 and 3.

The second is the effectiveness of the radical process innovation that open source development methods are claimed to represent. To the extent that these methods are effective and cannot be imitated or appropriated by rival approaches, they offer an alternative 'paradigm' for the organisation and division of labour in software production. None of the basic technical methods for producing open source software are unique to the open source community – commercial developers can or do use the same methods for editing, exchanging versions, discussing process and goals, etc. The two most compelling technical advantages of open source software development techniques over proprietary techniques are the opportunities for large numbers of individuals to identify and fix errors in the source code and for individuals to propose *implementations* of features or functionality that they believe will be useful to themselves.[3] Whether these technical advantages are sufficient to assure the effectiveness of open source compared to proprietary software development may be dubious. Either or both of these techniques may be imitated in part by commercial companies (with the aid of non-disclosure agreements and other restrictions on the re-distribution of source code). Thus, if open source does have a sustained advantage as a process innovation the source must be in how the community of developers is recruited, maintained and organised (or self-organised). The specific processes of organisation governing the life of open source projects are examined for clues about the sources of effectiveness in Section 4.

The third feature is a sufficiently broad acceptance of the results to sustain the first two features. We note that the widespread use of the Internet provides the means for distribution of software that is viewed as desirable by users and that, when this software is distributed without a direct charge to the user, the impediments of electronic payment hindering the development of e-commerce are absent. The ability to widely distribute open source software does not, however, assure its take up and use by the user community. For this to happen, an effective participatory development process must either produce or coincide with useful insights into user needs. The participation of users in the development process is one means of achieving this, a point that is central to the framework offered by von Hippel (2002). However, if the methods of open source development are to be extended more broadly, individuals other than those capable of making significant contributions to the coding process must also be included. We believe that it is not possible at this point in history to resolve many of the issues related to open source software demand, but will offer some notes concerning demand in the concluding section (Section 5).

The interaction of the three features described produces several dynamic properties in open source development communities. Clearly, effective techniques will attract entrants and participation although the processes for sustaining the intensity and breadth of involvement must be evaluated. Similarly, the acceptance of open source products will also stimulate the

[3] These are the advantages of 'hordes' identified by Raymond (1999) and user-led innovation identified by von Hippel (2002). Obviously, it is not simply suggestions for revisions or improvements that distinguish open source from proprietary software as the companies sponsoring proprietary software can solicit such suggestions from users and others.

entry and participation of open source community members. It is less clear that the larger community of participants, which emerges because of the employment of productive techniques or the creation of desirable products, will improve the effectiveness of open source development techniques. For this to occur, it is necessary either for effective mechanisms of selection somehow to *emerge* from the operation of open source development efforts or that such mechanisms be *selected* and *implemented* through some sort of choice process. These mechanisms are explored in a preliminary way in Section 5 where we present the conclusions from this paper and indicate the starting point for another investigation (Mateos-Garcia and Steinmueller 2003a). Finally, questions about how the recruitment process and the mechanisms by which open source products prevail over products created by rival techniques are largely questions about outcomes. These outcomes are also shaped by a variety of strategic (e.g. pricing behaviour) and institutional (e.g. rules on intellectual property rights) considerations that are well outside the scope of this working paper.

In summary, several specific features of the organisation of these communities are examined in this paper. These features represent solutions to the specific problems of organising the co-operative efforts that underlie open software development projects. In particular we will examine the social and economic processes that contribute to: a) the recruitment of individuals to participate in projects (Section 2); b) the maintenance of sustained efforts in the development effort, which, in our view, is intimately tied to dispute resolution, the avoidance of decision gridlocks or wars between individuals holding different views about the methods or outcomes that should guide the collective effort (Section 3); and c) the advantages and problems of open source organisational methods at each of several stages in software production (Section 4). In each of these sections, a specific theory and series of contentions are developed that lead to conclusions about 'what we should see' in open source development communities. These sections (Sections 2 to 4) also report on empirical evidence concerning whether these features are observable in the operation of several different open source development communities. Much of this evidence comes from observation of the SourceForge platform – a virtual community that supports a large number of open source development initiatives by providing tools for volunteers to find projects of interest and to communicate and collaborate with others in specific development efforts. SourceForge is described further in Section 2.

In addition to exploring the question of the sources of the effectiveness of the open source software development, our analysis is motivated by a critical concern with the existing research on the 'open source model' and its constituents. In our view this literature, which is discussed throughout this paper, often adopts an idealised and abstract perspective. This perspective leads to neglect of important issues in the social processes of how such communities operate, which needs to be considered in a more complete assessment of the effectiveness, the net effect of not entirely commensurate advantages and pitfalls, of this development model.

In particular, we contend that a lack of attention to the work processes in open source projects obscures the roles of hierarchy and authority operating inside these communities. In addition, the focus on instances of successful open source development efforts has obscured the outcomes within the larger population of open source development initiatives and the 'industrial dynamics' processes (birth, growth, and demise) of such initiatives. Section 5 synthesises the discussion in Sections 2 to 4 and outlines the dynamic issues involved in open source community development (the starting point for Mateos-Garcia and Steinmueller 2003a) and identifies several of the implications of this paper for further analysis and

empirical study. One of these is the application of open source techniques in the development of other information goods, a topic addressed by Mateos-Garcia and Steinmueller 2003b).

2 Building an Open Source Community

A useful starting point for considering the operation of an open source community is the recruitment of individuals to undertake open source software development. This section considers this process following a brief introduction to the precepts governing open source efforts and the existing literature on recruitment issues.

2.1 Linux's Law

Complete availability of information about the logical structure underlying a software program (that is, its source code) is the defining precept of the open source development methodology. By following this precept, it becomes possible for anyone (who has the required technical expertise) to analyse a particular open source program, to find problems in it and to devise solutions. This is not possible in the case of 'closed source' software, provided as a 'black box' whose inner workings are inaccessible to users.[4] One of the main strengths of the open source movement (OSM) is its accessibility. It allows what Raymond (2001) defines as 'Linux's Law': 'given a large enough beta-tester and co-developer base, almost every problem [in a program] will be characterised quickly and the fix obvious to someone'[5] The benefits stemming from the adoption of a holistic approach to design and a communitarian (peer-based) assessment of quality and collaboration in the improvement of a project constitute one of the bases of the success of the open source model.[6]

The availability of the source code in open source software makes it possible for an unlimited number of individuals to collaborate in its development. This characteristic by itself will not improve software design or implementation. The 'possibility' must be realised by the creation and effective operation of a community willing to analyse, test and improve the software. Therefore, an essential component of any open source project is the community of individuals voluntarily engaged in its development. The intensity and quality of effort this community is

---

4 Reconstituting 'source code' (software instructions written in a precise but fairly readily understandable programming language) from compiled or 'object' code (the precise instructions that govern the operation of the computer) is not straightforward. This is because source code typically involves labels and a variety of formatting conventions that enhance comprehensibility and indicate the logical structure of the software. These identifying features are destroyed in the process of 'compiling' object code (the computer does not need this information and its storage is inefficient). Although 'de-compilers' exist that provide limited clues about the original structure of the source code, any reasonably complex program will be extraordinarily difficult to de-compile in a form that is useful for editing and modification, the basis of the conclusion that the inner workings of the software are inaccessible to users.

5 Raymond (2001). Although Linux's law only refers to the issue of debugging and testing a software program, similar axioms are held for its design and enhancement. A larger number of individuals dedicated to improving a program will come up with a greater number of feasible strategies in what Raymond calls a '*parallel exploration of the design space.*' This characteristic of the open source model reduces the risk of adopting sub-optimal design architectures as more possibilities are taken into account at each stage of development.

6 They are what Harhoff, Henkel and von Hippel (2000) define as benefits stemming from 'voluntary information spillovers'. It is possible to consider an open source community as having a network structure whose productivity contribution or value follows Metcalfe's or Reed's law (increases in the number of connections/participants bring more than linear increases in value, see Reed (1999).

willing to invest in a particular project are the basic determinants of its strength and potential for improvements.[7]

2.2 Von Hippel's Interpretation

In addition to the advantages of review and improvement by anyone willing and able to undertake them, another important benefit of the availability of the program's source code in open source software is the opportunity for users to modify programs to address their own needs. According to von Hippel (2002), this user-customisation is one of the defining motivations supporting the growth of the OSM, which he categorises as a 'user innovation network'. von Hippel notes that in many sectors, including software, the existence of 'sticky information' about user needs makes it difficult and expensive for 'intermediate actors' such as innovation manufacturers to develop new technologies suited to user needs: it is probably more efficient for users to create those innovations in-house. von Hippel argues that under certain conditions the benefits to be derived from 'free revelation' of valuable information may outweigh the costs of organising it, making free availability of these innovations, the second characteristic of these networks, a rational strategy. Such conditions include low competition between users, the wish to establish in-house innovations as industry standards, the high cost and low effectiveness of intellectual property protection tools or the pursuit of 'reputation rewards', i.e. recognition by their peers within the community.

Von Hippel also argues that depending on the relative costs of the production and distribution of those innovations, we will find either horizontal innovation networks (when diffusion by users can compete with commercial production) or innovation by commercial manufacturers and distributors (in the opposite case). Open source software, as a result of its low cost reproduction and transmission, falls within the former category.

In other words von Hippel seems to provide an explanation for all three features identified in the introduction. First, the recruitment and retention of developer participation is supported by self-interest in 'customising' the code to the needs of the user-developers. Second, the effectiveness of open source relative to other software development methods in von Hippel's interpretation stems from the relative inefficiency of other methods in translating user needs in software design and implementation – Linux's law complements and reinforces this advantage. Third, the take up or acceptance of the software resulting from this participation is a direct consequence of fact that users have participated in 'customising' it by intervening in the process of development.

There are, however, some problems with von Hippel's interpretation. The assumption that recruitment is achieved through self-interest in 'customisation' leads to contradictions or unexplained problems.

First, it is possible that the user community has convergent needs so that individual interest is at times synonymous with collective interest. This possibility undermines one of the founding assumptions that open source is effective because intermediate producers are unable to effectively anticipate user needs – if such needs are similar across users the ability of commercial producers to anticipate them is less credible.

---

[7] We use the terminology open source community (OSC) to refer to the aggregation of different communities engaged in the development of specific software projects and the term 'community' to refer to individuals involved in specific projects.

Second, it is possible that user needs are divergent but complementary and, hence, do not conflict. In this case, the principal problem is that the resulting collective good, the open source product, will contain features that do not serve the needs of many users but were added to meet the needs of individual users. In this case, there is an incentive for those users who prefer a simpler product to remove the additional features and it is unclear how any sort of closure could be reached in the development process – versions of the software would proliferate creating confusion and uncertainty.

Third, it is possible that user needs will be divergent and competing in which case it would be necessary to set priorities and to exclude some features in favour of others. In these circumstances problems of authority immediately surface – who decides about which features to retain and which to discard. In addition, the process of setting priorities interferes with the recruitment and retention of developers as there can be no guarantee that the resulting product will meet their needs. While users might be able to bolt on the additional features that they desire, they are not likely to benefit from wide distribution, examination and review of their additions or efforts to better integrate their contribution into the product. Moreover, the process of version proliferation that would plague the case of complementary/divergent needs would occur.

In other words, it is difficult to imagine how the collective good represented by a particular open source software product can emerge solely through a process of self-interested participants introducing features to the software to customise it to their needs. The problem of negotiating the features of the product in a way that sustains recruitment and retains participation requires a more precise specification. It is possible that effective means for negotiating product features exist and provide non-replicable advantages for the development of open source software.[8]

2.3 Kasper Edwards Interpretation

An alternative to von Hippel's interpretation of the recruitment and other features of open source community operation is offered by Kasper Edwards, who has studied open source communities using the concepts of 'epistemic communities', 'situated learning' and 'legitimate peripheral participation'.[9] Edwards's interpretation provides a useful starting point for analysing issues of authority, hierarchy, recruitment and conflict in open source communities. In this sub-section we briefly summarise Edwards's work, adding a few notes on the role of 'authority' in open source projects, which is more fully developed in Section 4.

An epistemic community can be defined as a group of practitioners that adopts a role as provider of information for decision-making in a context of uncertainty. It is characterised by the existence of a set of beliefs, methodologies, objectives and values that are shared by all its

[8] Open source development methods may prevail over competing techniques even if the negotiations concerning product features are less efficient than the actions taken by private producers to anticipate user needs. The necessary condition is that open source provides specific advantages in developing the software, that is, Linux's law. This case, however, contradicts von Hippel's premise that the innovation network offers unique advantages in better anticipating user needs.

[9] Edwards 2001.

members.[10] These different attributes need not be explicit (for example, be in the form of a code, a set of rules or aims), but are nevertheless internalised by the members of the community and embedded in their behaviours. Thus, instead of beginning with heterogeneous self-interest, the epistemic community starts from a position of shared interests and their accompanying features. The effort to create shared values, notions of validity, beliefs, methodologies, etc. within open source communities is evident in documentation, such as project statements, 'frequently-asked question' (FAQ) lists, open source manifestos, etc.[11] Although the aims, methods and values of an open source community are revealed by such text, perhaps even stronger evidence is provided by experiencing the interaction between members of such communities in which expectations about others' behaviours are tested and re-calibrated against actual behaviour.

The experiential nature of the epistemic community construction is illustrated by how newcomers learn the values and knowledge of the open source community. Such communities are explicitly open to new members and provide recognition of their legitimacy within the community, although not their rights (yet) to participate on an equal footing with existing members. Newcomers slowly assimilate the community's values and learn its methodologies by participating in the social and technical life of the community, beginning from a peripheral position, and eventually becoming 'insiders'. This model of learning as socialisation and participation, known as legitimate peripheral participation, was originally developed by Lave and Wenger (1991) and was adopted by Edwards as the explanation for the recruitment process in open source communities and how these communities reproduce, from a historical perspective.

Two features are missing from Edwards's explanation. First, what is it that motivates individuals to seek to join the community? Second, how are conflicts managed as individuals, upon gaining the legitimacy of becoming an insider, provoke disagreements about either the goals or the process of the community or their own power or status within it? This second question is taken up in Section 4.

The motives for joining open source communities are clearly diverse and constitute a useful subject for further research. However, from observation of the interactions of open source developers in various public and development forums the following are clear candidates.

? To demonstrate technical competency within a community of peers or superiors so as to gauge one's own level of skill and suitability for deeper professional involvement in software design and engineering.
? To counter the social isolation of working professionally as a salaried software designer or developer in projects that are defined by organisational needs over which one has little control. Most software engineers work most of the time on maintaining and extending the functionality of software that has been customised from the vertical system 'solutions' offered by companies such as SAP and ORACLE or that have been developed in-house, usually some time ago. By comparison, participating in the OSM may seem fun or be exciting!

---

[10] Haas (1992). See Cowan, David and Foray (2000) for a directly relevant application of the epistemic community approach to the problems of knowledge codification, of which software is a principal example.

[11] We can, as a starting point, consider them embodied in two main concepts: 'The Hacker Ethic' and the 'Open Source method'. See Himanen, Torvalds and Castells (2001) and Raymond (1999).

? To produce a software product that will be useful in one's professional or recreational use of computers. This motive partially encompasses the user-producer motive of von Hippel but is more open-ended with regard to the issue of 'customisation'. A user-producer may gain beneficial knowledge from participation in the development process regarding the configuration, application and adaptation of a software product that is only tangentially related to their own specificities.
? To gain specific knowledge about a software product that will be of professional value to an individual or an individual's organisation and, perhaps, to influence the development of this product in ways that will be beneficial to the user or their organisation. This motive is also linked to von Hippel's user-producer motivation and may account for explicit corporate sponsorship of participation in open source development activities.
? To participate in a voluntary community that is engaged in a goal that the individual feels is worthwhile and/or operates in a manner which the individual finds congenial. A wide range of more specific motives may fall under this heading, including political commitment to collective action, belief in anarchistic or libertarian principles of voluntary association, or simply excitement about being able to participate in the creation of a particular technology.

It is important to note that all of these motives may be present and that, at this moment, none can be clearly identified as dominant within the actual community of individuals participating within open source communities. In effect, the motives for joining and participating in open source communities are likely to be heterogeneous, while the process of participation may serve to align diverse individual motives in a set of shared values and practices that establish the basis for regulating and governing the collective activity. To observe the processes of regulation and governance as well as the outcome of processes of recruitment and participation it is useful to have specific empirical evidence and the next section introduces one source of such evidence.

2.4 The SourceForge Development Platform

SourceForge is an online platform whose stated mission is to 'enrich the Open Source community by providing a centralised place for Open Source Developers to control and manage Open Source Software Development'.[12] That is, Sourceforge provides a location and a set of resources for making collaboration between open source developers easier and more fruitful by providing tools for development, by supporting the exchange of technical data and by creating forums for discussing objectives and purposes of development efforts. The services offered by SourceForge include:[13]

- Web space for the storage of open source projects as well as current version systems (CVS) development tools and file download facilities.
- A 'compile farm' of hardware and operating system environments where open source projects under development can be tested in different conditions.
- User-friendly tools ('trackers') for the reporting of problems ('bugs') found in programs, the submission of solutions ('patches'), and the request for additional features and support.

---

[12] http://sourceforge.net/docman/display_doc.php?docid=6025&group_id=1. Accessed 24/9/2001. The SourceForge portal is a version of a product for distributed software development commercialised by VA Software.

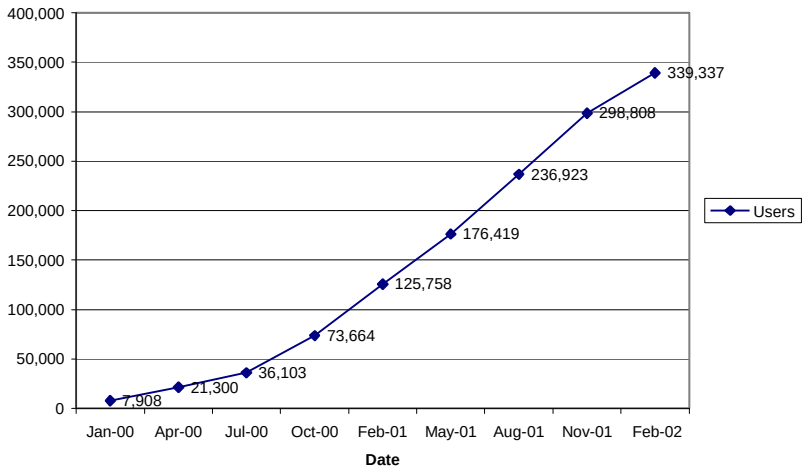
[13] A more detailed overview of the services provided by SourceForge can be found at http://sourceforge.net/docman/display_doc.php?docid=753&group_id=1 Accessed 25/01/2002.

- Mailing lists and forums for the exchange of information between developers both in the shape of 'project forums' and topic-based 'foundries'.
- A search engine for the localisation of projects and individuals according to different characteristics, as well as a 'project-help wanted' forum.

As of 25th September 2002, SourceForge was hosting 46,762 projects and had 486,148 registered users.

There are two key points to be made about SourceForge regarding recruitment. The first is that it has been a successful means of recruiting individuals to participate in the open source movement. Figure 1 depicts the growth in membership over the past several years. In a two-year period SourceForge succeeded in attracting over 300,000 users.

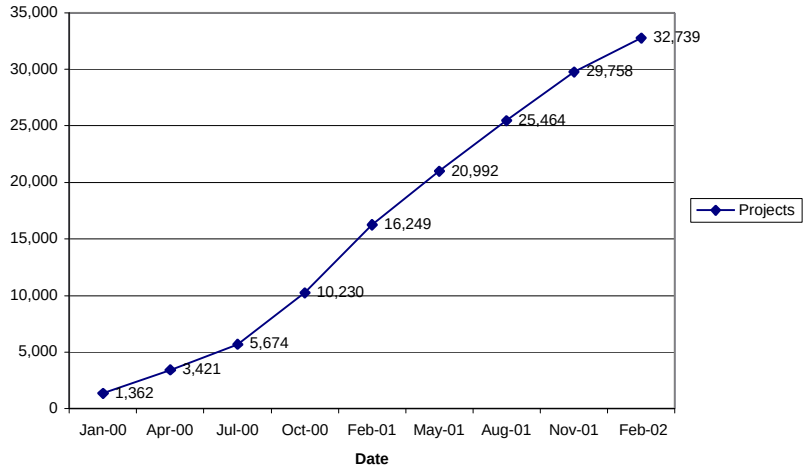
Figure 1 SourceForge Registered Users



The second point to be made about the SourceForge recruitment process is that the distribution of user participation is highly skewed. Although the number of projects has increased in parallel with the number of registered users (see

Figure 2), the leading projects attract the majority of participants with a very large number of projects with only one participant or author being essentially ‘bids’ for project participation.

Figure 2 SourceForge Projects



Krishnamurthy (2002) finds that the modal number of developers participating in a sample of the 100 most active SourceForge projects was 1 and that from the projects he selected the highest level of participation was 42. These statistics are consistent with the relationship between projects and developers where an even division of the developers over projects would imply approximately 10 developers per project. The skew towards unity in the modal (and to 4 in the median) number of participants in the typical (modal and median) project reported by Krishnamurthy (2002) requires that some projects receive significantly more users. Krishnamurthy (2002) suggests that the model of open source development process offered by Raymond (1999), in which a large 'horde' of co-developers is essential to achieving the advantages of software improvement associated with the previously described 'Linux's Law', may not actually be widespread within the open source movement. Moreover, the generally relatively small number of participants in a project suggests that the processes of governance and authority within the community and, indeed, participation of others than the progenitor of the project, require careful examination. This topic, the problem of sustaining open source project involvement, is examined in more detail in Section 3.

3 Sustaining Open Source Development Effort

The existence of a set of shared norms, beliefs and methodologies in the open source movement does not preclude disagreement and conflict. Disagreements are likely to arise when comparing the merits and downfalls of different technical alternatives. The existence of distinct bodies of specialised knowledge among the members of a community, of uncertainty about the relative merits and efficiency of different approaches in achieving the aims of the community, and questions concerning the aims and purposes of the community are potential sources of discord and disagreement. These disagreements must be resolved in a way that preserves the continuity of effort in the community. Dispute resolution requires an institutional framework, which may be either chosen or emerge from interpersonal interaction. In either case, rules, norms and practices will be available for resolving disputes. The simplest dispute resolution mechanism is the constitution of 'higher' authority and we begin the discussion in this section of sustaining community efforts with an examination of the origins and use of authority. This discussion leads to the conclusion (to be found in Section 3.2) that, to a meaningful extent, individuals other than the progenitor or leader of the project, may accumulate authority. The consequences of this distributed authority for project development are examined in Section 4.

3.1 Integrators and Authority

Authority is exercised in deciding, for example, which of two alternative patches that solve the same problem will be included in a release, which implementations of a feature are more effective, etc. When the community reaches this point, it may delegate responsibility to the project leader to decide on the technical direction to be followed by the project. It might be argued that such decisions could be taken objectively, but in our opinion, this would imply an excessively idealised view of the commensurability of different technical solutions in a context of uncertainty. It is our contention that the decisions taken frequently involve a degree of arbitrariness. In the case of many important open source projects, we find rules-of-thumb and heuristics aiding and guiding this decision-taking process. Some examples of these rules would be 'make it work first and then improve it' or 'keep interfaces clean' or 'usability above complexity' (or the reverse). But even these different rules emerge as an exercise of authority in establishing criteria for measuring and comparing the efficiency of different solutions. On many occasions, we find that it has been the project administrator that has been the one to establish or make these rules explicit.[14]

The arbitrariness/subjectivity of the project administrator's decisions is, in turn, governed in the open source model by the possibility of 'forking'. Forking is the special characteristic of the GPL (General Public Licence) (and other open source licences) governing open source software that makes it possible for individuals dissatisfied with the decisions taken by a project administrator to fork that project. That is, to start a new development branch following a different technical direction. Having reached this point, members of the supporting community will decide which of the two branches they will support, in an implicit

[14] Linus Torvalds and the guidelines he set for the contribution of patches, enhancements and functionality to the Linux OS are the most obvious example of this case. For example, Moody (2002) quotes Torvalds as saying: 'I've never been all that concerned with design. I have a very pragmatic approach: What works and what people want to use is good'.'

exercise of democratic participation.[15] However, although forking would appear to be an available and legitimate way of challenging the decisions of an incumbent project administrator, it is a very rarely exploited. Raymond (1999) tries to explain the surprising stability of an open source community from a cultural perspective, arguing that forking, in the open source cultural milieu, constitutes a taboo, as it violates a project leader's 'ownership' of a project and weakens the supporting community.

Raymond argues that project progenitors or leaders have ownership within a property rights framework. By founding the project, an individual establishes an 'intellectual territory' that others will be wary of trespassing. For Raymond the existence of taboos, rules and customs inside the community of practice of software designers and developers and the potential for punishing untoward behaviour are the basis of the enforceability of property rights and hence the leader's authority. Thus, authority over a project, e.g. the right to distribute modified versions of the software program being developed inside it, is an extension of the 'property right' of the leader. If projects are the 'intellectual property' (in loose terms) of their founder, authority can be legitimately conveyed to a successor (inheritor). Ownership and authority over a project may also be obtained by recovering it from abandonment (that is, in the case that the previous owner has stopped working on it). Other mechanisms for establishing authority are, for Raymond, of lesser consequence.

A similar framework is adopted by Lerner and Tirole (2000), again taking the role of the leader as central in defining the authority structure governing the operations of the community. Unlike Edwards (who uses, but contests, the validity of the term 'leader' given the voluntary nature of participation), Lerner and Tirole assume that the leader has a directive role within the community. In Lerner and Tirole's model, the leader not only creates the 'vision' for the project but also plays a key role in directing the division of labour. As Edwards (2000) notes, this conceptualisation differs markedly from open source community practice where user involvement is engaged less through statements of vision and more through the distribution of working code that provides the basis for extension and amendment, the proximate activities of contributors. That this process is also accompanied by discussion and interaction between the leader and the contributors, or among contributors, is to be expected. It is not, however, helpful to see the leader as a 'master planner'.

If the role of the project leader is not directive, but instead involves a co-ordinating role and the retention of some right to intervention or arbitration over the co-operative activities within the project, how does authority operate within open source projects? Answering this question involves returning to the issue of what motivates individuals to participate in these activities.

3.2 The Interaction between Participation and Authority

Raymond (1999) proposes that the motivations of individuals for contributing to open source development, centre on the accumulation of status in a reputation game. The rules of this game dictate that those who 'contribute more' to a project achieve respect from their peers and increase their status, and thereby derive satisfaction. This status is not directly linked to authority (which emerges from the property rights scheme described above), although Raymond does seem to acknowledge that reputation conveys some degree of authority. Raymond's account seems in a sense limited because it only acknowledges the 'reputation'

[15] The similarity with the framework of Hischman (1970) who called attention to the responses of exit, voice and loyalty to organisational stress is notable.

factor driving individuals to collaborate in open source development and ignores the other economic political, ethical and personal motivations proposed elsewhere.[16]

In examining the construction of authority, the participants' motivations are important. In our view, these motivations (see outline of motivations in Section 1) stem from three distinct sources: 1) the capability to direct the knowledge development process to meet specific needs; 2) the value of contributing to developing knowledge and the opportunities this affords to acquire knowledge; and 3) the desire to socially interact with others engaged in software design and development. Only the first of these motivations necessarily involves a direct quest for authority to influence the project's development, i.e. to contest the progenitor's presumed right to take decisions about the solutions sought and the means of implementing them.

The second and third motivations, however, may lead to the building of reputation through a process of interaction with others. As a result of these interactions, an individual gains the respect of other community members, which implies that a new leader would win the allegiance of these members in adopting a fork in the development of the project. In other words should forking occur, the original leader or progenitor might be displaced and authority over the project seized by someone else.

This is a very divisive and confrontational description of the social dynamic by which reputation is translated into authority or influence. It ignores the socialisation process described by Edwards (2001). This socialisation process serves to align the incentives with the interests of participants and is likely to encourage exit from the community by all but the most obdurate dissenting or opportunistic individuals. Indeed, it is likely an individual will only react to the second motive by acting 'in public' as if they are motivated by the first or third. Similarly, more directly self-interested rewards such as 'reputation' (Raymond) or 'indirect economic rewards' (Lerner and Tirole) can be obtained as a by-product of operating as if it were the second and third motives that governed an individual's behaviour. In following these motives visibility and the acknowledgement of the skill and capabilities of an individual are obtained and these lead to the desired benefits of reputation and the recognition from outside observers that may translate into grants, job opportunities, venture capital, etc.

The same blunting of the explicitness of power seeking is likely to occur for those individuals whose main motivation for participating in an open source project is to develop a product that better fulfils their needs. Someone participating, learning from and becoming influential in an open source project could be expected to be more capable of determining its technical direction and better able to achieve an outcome that satisfies his or her requirements or specific commercial objectives.[17] A similar strategy is as relevant for those individuals following a political or ideological agenda, because authority in a project, acquired through participation, will make it easier for them to steer the project in a particular political or ideological direction. [18]

---

16 Economic motives include signaling, see Lerner and Tirole (2000) and von Hippel (2002), political motives include resistance to proprietary software and ideological support of Free Software, see Bezroukov (1999), Levy (2001). Ethical and personal motivations as well as political motivations are examined in Himanen, Torvalds et al. (2001).

17 Participation of commercial companies in open source development could be seen as a way of ensuring that the programs being produced are well suited to their requirements/objectives (i.e. compatibility with proprietary complementary assets, such as hardware).

18 The collaboration of Linux commercial distributors and more political groups such as the FSF in development activities can be seen from this perspective.

Finally, it is possible to pose a 'craft' hypotheses in which individuals contribute to open source because they find software development activities intrinsically satisfying and enjoyable. We could simply say that 'individuals of this type contribute to an open source project because they like to contribute', without introducing any extra variable into the argument. In this case, as Raymond notes, 'Status (Authority) may come as an unexpected outcome.' Even then, we can still consider authority to be an extra incentive for these individuals, inasmuch as acquiring it is bound to make it possible for them to take decisions at higher levels of the project and steer it in a direction that they find more 'technically exciting'.

3.3 Distributed Authority?

If the individuals within a community accumulate a degree of respect and mutual recognition from their peers that is, in effect, reputation, does this reputation become 'authority'? Authority, as we employ the term here, means ability/power to make decisions about the aims of the community and the decisions about their implementation including the methods to be employed. Distributed authority would imply that participants other than the project leader or progenitor would have a role in taking these decisions.

In order to conclude that authority within a community is distributed it is necessary to look more closely at the process by which software code is developed within such communities. The most typical model (see Edwards (2000) and Raymond (1999)) is where open source projects begin from a kernel of working code that is *extended* as well as being edited through individual contributions. If editing (e.g. bug fixing, cleaning of code to make it function more efficiently, documentation of source code, etc.) were the only function of the larger participant community it would simply not motivate most participants. It is the process of extending the code, adding functionality or features, making it more flexible and configurable and integrating it more smoothly with other software and hardware systems, that provides of the interest and satisfaction for skilled programmers. This process of extension is also, however, the source of potential conflict since the community programmers may not agree that all these extensions are worthwhile. Thus, disagreement is intrinsic to the process of open source software development and its most important and innovative features would be lost in the absence of workable arrangements for distributing and changing influence within the community.

The extent to which such influence is explicitly recognised as 'authority' with regard to taking decisions as to direction, deciding about contesting contributions, or releasing new fixes, patches, or versions will depend upon the personalities involved in a particular community and history of the community's development. For example, Apache's democratic committee is logical if we take into account that after its inception, development was done co-operatively as a series of patches, put together communally, by a core of elite programmers. On the other hand, Linux's model of benevolent dictatorship can be seen as the result of the asymmetry in the distribution and capabilities for the exchange of knowledge between Linus Torvalds and other initial contributors. Still other models in which explicit authority is devolved into working groups or held by a 'central committee' are employed in various other projects. In all of the examples, however, some distribution of authority is inevitable as the project admits the innovative contributions of its participants, a condition for maintaining their participation and avoiding the risk of forking.

Raymond (1999), retaining his focus on the 'property rights' explanation of the structure of open source communities, acknowledges that authority is distributed within a project in the following way:

> The subsystem-owner role is particularly important for our analysis and deserves further examination. Hackers like to say that 'authority follows responsibility.' A co-developer who accepts maintenance responsibility for a given subsystem generally gets to control both the implementation of that subsystem and its interfaces with the rest of the project, subject only to correction by the project leader (acting as architect). We observe that this rule effectively creates enclosed properties on the Lockean model within a project, and has exactly the same conflict-prevention role as other property boundaries. Raymond (1999)

Raymond's view of the project leader as the master architect is, however, problematic. He does not demonstrate how delegation of responsibility for a major sub-system to a sub-system owner allows the project leader to retain meaningful control of the 'architecture' of the system as such systems are modular. It is a bit like saying that the glazing contractor is given control of what and where windows will be installed in a building, but arguing that because the architect has specified that the building shall have windows that the architect retains 'ownership' and control of the design.

A somewhat different view of the process is the following account of the revisions process is in the GNOME project:

> Even though any developer with CVS [current version system] access can modify any given module, it is expected that only a small group of them can do it directly. These developers have demonstrated to the maintainers of the module that their patches can be trusted. GNOME works, as many other open source projects, on the basis of merit. At the beginning of his or her contributions, a new developer is excepted to send patches (or deltas [changes] from the current version of the module) to the maintainer of the module, until the maintainer acknowledges that the new developer can directly apply his or her patches to the CVS repository, and therefore, those patches will no longer require approval by the maintainers. German (2002)

In the gap between the master architect and the maintainers, who have authority over the implementation of sub-systems (the system is a collection of sub-systems) and who allow 'trusted' developers to directly make patches into the current version, there is considerable scope for distributing authority within the project. It is not at all clear from any single developer's or community's experience that there is a 'master model' of how authority is distributed within a project between the progenitor and the contributors. In fact, we must expect a variety of models, depending on the development of individual communities. In other words, at this stage of research into the OSM, it is not appropriate to conclude that there is a single model of distributed authority. What can be said is that participation creates reputation and reputation in turn can be translated into a degree of authority over how the project proceeds and is implemented. What this more complex and negotiated model shares with Raymond's property rights model is consensus that the aim of sharing authority is to preserve the stability of the community against the possibility of 'forking.'

At this point then, the idea of 'distributed authority' is a conjecture, supported primarily by assumption and rhetoric with only some suggestive evidence from the GNOME project.

Section 4 offers corroborating evidence in the course of a deeper exploration of what makes the 'open source' development method effective, the second feature of the OSM, suggesting that it may constitute a historical branch in the organisation of software production.

4 Avoiding Demise and Disintegration of Open Source Development Efforts

The conjecture that 'distributed authority' may be a fundamental instrument in open source communities for resolving disagreements and taking decisions is linked to assessment of the OSM's effectiveness as a means of dividing and organising labour in software development. This section carries forward the analysis of the interactive social processes underlying distributed authority while adding the complementary elements of Linux's law, which argues that 'openness' contributes to effectiveness by increasing the intensity of the search for incremental improvements and error fixes to a body of software code.

As in the previous section, a pivotal element of our analysis is the coincidence of the opportunities for 'forking' and the relative infrequency of such opportunities being exploited. Distributed authority offers one explanation, and opens the door to further hypothesising about how hierarchies emerge through processes of socialisation and interaction. This explanation, however, is not inherently more plausible than that of Raymond who maintains that the progenitor-leader of a particular community retains ultimate authority and 'ownership' of the endeavour and that any observed hierarchy involves a delegation of this authority, which may, at any time, may be reconfigured as the progenitor sees fit. Distinguishing between these two interpretations requires examination of interactions within open source communities and case studies of how conflict and crisis are managed in particular projects.

4.1 The Challenge to Authority

In the previous section we made the conjecture that in the open source context, the reputation of an individual derives from her or his knowledge of the project, acquired through participation in its development, and that this reputation provides the basis for authority. We now examine challenges to the authority of the progenitor, or to the 'inner circle' of individuals with the highest reputation, conducting our analysis inside the same framework employed previously. The aim is to illustrate how the way in which a project evolves, and the conflicts that emerge in the process, can be seen to arise as a result of the diversity of objectives, perspectives, schemes and methodologies of the participants. Moreover, new entrants must go through a catching up process in the course of which they acquire enough knowledge to be able to challenge the decisions of the leader or the 'inner circle' of individuals with the greatest influence.

Heterogeneity of knowledge and visions. We have already mentioned that in a context of great uncertainty and complexity it is unrealistic to expect technical decisions to be completely objective. Computer science does not provide deterministic nor optimal solutions to the preponderance of problems that arise in the course of constructing a software system. Disagreement between software engineers over which techniques, solutions and methodologies are more efficient in achieving certain objectives, are likely to be

continuous.[19] These disagreements on many occasions are compounded by a lack of consensus about the objectives.

Being constituents of an epistemic community, members of a particular open source project share a set of values, beliefs and heuristics which may attenuate the severity and scope of disagreement. However, in many cases these values and rules will be too abstract to be directly applicable to conflict resolution. Even when the objective is clear (i.e. creation of a high quality software program), the best way to achieve this (i.e. implementing solution 'a' or solution 'b') may be difficult to assess. The distance between values as abstract aims and rules embedded in the open source culture, and their actual manifestation in the technical innards of a software program is long and the way between the two may not be always be very clear.[20]

Open source projects, of course, afford relative ease of entrance. Despite the processes of socialisation, which communicates the values of the community, alignment between the different members regarding objectives will not necessarily be very good in the initial stages of project development.[21] We have already discussed the different motivations of individuals contributing to open source development, and that conflict between individuals with, for example, different political agendas (see Open Software vs. Free Software), would occur in the struggle for the authority necessary to steer a project in a particular direction. In the socialisation processes as described by Edwards (2001), there may be shortcomings in transfusing the values of the community. The lack of explicitness in this process about the values, norms and visions may create an atmosphere of apparent stability and agreement that is shattered as soon as there is any hint of disagreement in the development of the project.

In open source communities, we find a set of individuals with different knowledge and skills (which are associated with a preference for different implementation methodologies, coding styles, etc.) and beliefs and aims (which produce different 'visions' of the nature and evolution of a project). Any group of individuals, even with a shared vision, will inevitably encounter problems of design choice. In this context, the authority of the project integrator is the sword that can cut the Gordian knot of endless deliberation. The greater knowledge about the project that makes the integrator a 'leader' in the community could be expected to reduce the number and intensity of these debates. In the environment of open source software, the strength of technical arguments and knowledge about the inner workings of the program being developed are the main weapons in any discussions, and we could expect the program leader to have the monopoly on both.

Growth in complexity during the life the project. This issue introduces a temporal dimension to the analysis of open source communities and their characteristics. In order to illustrate our argument and clarify some of the points made earlier, we consider the evolution of a project, as it grows in complexity, showing the sources of conflict and challenge to authority in the framework of the heterogeneity of skills and objectives described above.

---

19 Manifestations of this situated at completely different levels of the computer science discipline include the controversy surrounding the issue of Monolithic Kernels Vs Micro-kernels that has persisted for years at a theoretical level: the problems of deciding whether a reduced or complex instruction set is most appropriate or the diversity of preferences over coding style and approaches that exist in the programming community.

20 So, even if values are shared at an abstract level, disagreement is bound to emerge at a more concrete one.

21 These objectives, related to the tasks that the software program should accomplish, the niche it should fill, etc. are termed by Edwards the 'vision' of a collaborator in the project.

At the beginning of the project there is little redundancy of effort, which reduces the need to decide about implementation of alternative options that address the same problem, and the project's relative lack of complexity at this early stage will make any decisions that are necessary an easier matter. We are not taking a 'rosy' view of the initial stages of a project, however, and acknowledge that even at this stage conflicts may arise. Nonetheless, the substantial differential in knowledge about the project between the leader and the other contributors, as well as a wide space for development in which contributors can work independently on different parts of the program, reduce disagreements and the need for resolution through the direct use of authority.

However, when (and if) a project starts growing in size (both in terms of the number of contributors and lines of code, extra modules and components), the following dynamics are likely to occur:

a) Once the basic elements of the software have been identified, different individuals are bound to start work on the same pieces of the program. If the options they provide for the solution of a specific problem are conflicting, the issue of having to decide which of them is more efficient or suitable will start becoming more important. The 'release early, release often' policy adopted by many Open Source projects, and usually justified on efficiency grounds (that is, to avoid 'everyone reinventing the wheel') may attenuate this process, but still we could expect it to be present and to progressively become more serious.[22]
b) As the project grows in size and complexity, the contributions made by the community will become more numerous, exerting increasing pressure over the capabilities of the maintainer responsible for assessing and incorporating them into new releases.
c) The increasing complexity and size of the project will necessarily reduce the knowledge differential between the leader and the other contributors. As other individuals invest more and more effort in development there will be a process of catching up or perhaps even overtaking the progenitor in terms of knowledge about specific modules in the project. This opens the way to serious challenges to the authority of the progenitor in resolving technical disagreement. If the progenitor keeps investing major effort in the development, and if the knowledge she or he already has is difficult for other contributors to acquire (perhaps due to lack of documentation), the progenitor will be able to maintain the knowledge differential for a longer period. However, even in these circumstances we would predict that decreasing marginal returns in the progenitor's effort and capabilities will eventually lead to convergence between his or her knowledge and that of the leading participants. The progenitor's authority then becomes less obvious and less conclusive in both decision-making and conflict resolution.[23]

---

[22] Favouring modularity in an open source project is, as Linus Torvalds explains it, another way of avoiding this problem: 'without modularity, I would have to check every file that changed, which would be a lot, to make sure nothing was changed that would affect anything else. With modularity, when someone sends me patches to do a new filesystem and I don't necessarily trust the patches per se, I can still trust the fact that if nobody's using this filesystem, it's not going to impact anything else' (Torvalds 1999)

[23] For example, providing insufficient documentation about code may be a strategy used by the leader to retain superior knowledge about the project and retain authority. However, we have noted that 'documentation' is one of the ways of facilitating participation in a project and, more specifically, of signalling an individual's technical proficiency and skill. Even though poor documentation would

d) In the case of complex pieces of software, in the initial stages of development the main objective of contributors will be to make the program achieve a certain degree of functionality: as already mentioned there is ample room for development, and there is also ample room for growth. Following Edwards (2000), we could say that the progenitor's 'vision' of the program (in technical, social and teleological terms) is implicit in the initial code she or he provides. The visions of other contributors will not, however, necessarily coincide with that of the progenitor, as they are created independently, as interpretations of the available text (code).

In replying to any of the above circumstances, the progenitor may issues explicit directives regarding future directions of development, thereby foregoing the potentially innovative inputs of others and reducing their incentives to participate. Alternatively, by accepting such inputs the progenitor will avoid most disputes over conflicting visions of the program. In the earliest stages of the project, the software will still be a rudimentary and amorphous artefact, and these inputs are needed to make it more functional. However, once a project is well advanced in its development, explicit decisions about the more specific aims and how to achieve them will be necessary.[24] It is at this stage that the visions of individual developers and that of the leader will start to conflict. For example, in the case of the Linux desktop vs the Linux server, factions that were not initially obvious became visible as the need for decisions on these issues became more urgent.

There are some instances of forks in open source projects that can be explained by the previous framework. For example, in 1993, GNU (Gnu's Not Unix) Emacs (Editor MACroS) project forked into two development branches, one continuing to be supported by the Free Software Foundation (FSF) and the other by a group of developers led by Jamie Zawinski. Zawinski and his collaborators, who were frustrated at the slow pace of FSF decision-making as the number of group participants grew, were sufficiently knowledgeable about Emacs that they were able to implement their own patches. In doing so they created a 'new version' of the program and demanded that this should be accepted as the official version. This amounted to a 'hijacking' of the program by Zawinski, as he was challenging the FSF's authority to control the program's technical progress. Richard Stallman and the FSF refused to acknowledge the new versions, resulting in a development fork from which two Emacs products emerged.

The GNU C Compiler (GCC) project was a similar case. Cygnus Enterprises introduced a fork in the original FSF program because, as Michael Tienman, a member of the firm at the time said 'more and more of our ports of the compiler and enhancements to the compiler got backed up in the patch reviewing process […] the FSF [the original integrator] process became a limiter of our business' (Moody 2002). In this case, the challenge to authority was conceded and the FSF decided to adopt the Cygnus version as the basis for future official development, thus 'healing' the fork.

---

render the participants more dependent on the progenitor, it would make it more difficult for members of the group to perceive and appreciate his or her skills and knowledge, thereby diminishing his or her authority.

[24] One example of this would be the integrator becoming more exigent concerning submissions. Edwards (2001) pointed out that the leaders of successful projects eventually turn into choosers, as their visibility and attractiveness grow.

4.2 Why is Forking Rare?

In summarising the above discussion, we maintain that, as a project grows, conflict will necessarily emerge. Divergences in the objectives and methodologies of the members of the community will become more frequent and more explicit. The leader may have to solve these disagreements through the exercise of 'ownership' authority as authority based upon knowledge will inevitably diminish as the project progresses. The progenitor's decisions may start to be challenged, and new would-be leaders proposing alternative solutions will emerge. This is the root cause of fragmentation of a community into factions supporting different technical directions that could eventually lead to the creation of divergent, forked, development branches, such as those we have described.

It would seem, therefore, that a project's successful development contains the seeds of its eventual rupture, the processes described above being general enough as to occur in most moderately complex open source software development efforts. While their prevention is made impossible by the wide rights granted by open source licensing schemes, we still find that forking is a relatively rare and dramatic phenomenon within open source communities. As noted earlier, Raymond sustains his property right argument by positing the existence of taboos and rules in the hacker culture, mainly an internalised respect for the project founder, that is, its legitimate owner, that turns forking into an extremely undesirable option. This option is only adopted as the final solution to unresolvable conflicts inside the community.

Raymond argues that stability is preferred as evidenced by the existence of rules and norms inside open source communities and the presence of hierarchies, which are another conflict prevention and minimising factor. While for Raymond these hierarchies are linked to the 'property rights' of the progenitor, our contention is that it is the growth of competing expertise that produces the contests for authority and demands specific qualities from the project leader. The distinction is important because it is dubious to assume that all members of an open source community will recognise and respect the progenitor's property right. Even more important, focussing on the constitution of authority and its distribution offers a much more specific set of guidelines for empirical study, and design of strategy and policy. For example, Raymond's formulation leads to the conclusion that forging a common ideological perspective among members regarding the relation between authority and property rights while the distributed authority formulation indicates that the mechanisms of conflict resolution that a community discovers, adapts, or chooses are of greatest importance.

In the 'distributed authority' explanation, the progenitor's choices regarding contributions and the people she or he will associate with will, over the longer term, either sustain or erode her or his leadership position. As a project grows in size, the burden on its integrator increases, in terms of assessing and incorporating into new versions of the program more and more patches and enhancements in modules that almost inevitably she or he is not intimately familiar with. Given the limits to any individual's capacity to process information, eventually the progenitor will be unable to cope with the submissions of collaborators. An obvious way to deal with this problem is to delegate the task of filtering patches and submissions to others individuals. In the case of Linux, these individuals are referred to as 'lieutenants'.

The delegation process is exemplified by the conflict that arose in the Linux Operating System project in Autumn 1998. Linus Torvalds, the project's progenitor, was unable to process all of the patches submitted to him by contributors. This nearly resulted in a fork of

the project into two different branches, Torvald's version, and the Vger version (maintained by individuals whose contributions to Linux had not been accepted or even perceived by Torvalds). The solution to the problem in this case was to adopt a star-shaped structure, with Torvalds at the centre and a set of 'trusted lieutenants' at each point. These lieutenants became responsible for particular modules of Linux in which they were most knowledgeable, and undertook the task of processing contributions submitted to them by 'lower-level' developers, thereby reducing the amount of information that Torvalds had to assess when taking decisions.[25]

The growth in complexity impels the creation of hierarchical layers in open source projects. To understand the resulting reconstitution of authority, it is necessary to focus on the process through which the actual members of these layers are selected, and more specifically, to identify their characteristics.

First, we would expect them to have a deep knowledge of the portion of code/module (sub-project) for which they are responsible, this knowledge having being acquired through high-quality, continuous contributions. This knowledge, in our framework, is the source of their authority over the particular module.

Second, an individual's authority is likely to be based not only on their technical expertise and quality of contributions, but also on the extent to which they share the progenitor's 'vision' of the project and its aims and methods of implementation. Even in the technology-focused cultural context of open source communities, it would be difficult to conceive that an integrator would delegate responsibility for an important module in the program to anyone expressing divergent vision about its future (in terms of both technical and strategic aspects). As Linus Torvalds puts it when describing the policy he follows for the integration of patches in his development tree of the Linux Operating System,

> I take stuff that I feel is good. Often that feeling of goodness comes from trusting the person who sends it to me, simply by past performance. At other times, it is because I think the feature is cool, or well done, or whatever. Hint: if you want stuff in my tree, make me trust you. Or work on things that I feel are innately interesting. Don't bother dragging me into your flame-wars and trying to convince me that I 'must' apply your patches.[26]

Therefore, we find that as the capabilities of integrator for keeping pace with a project's development start to diminish, a structure incorporating layers of trusted individuals will emerge as a way of helping her or him cope with the increased complexity and size of the project. These layers will be composed of individuals proficient in the project. Their vision of the project will also concur with that of the leader in some essential points (such as preferred programming languages and techniques, basic aims of the project, etc.).

The second, and simultaneous, process that we would suggest takes place in open source projects as they grow is also implicit in Torvald's remark, and can be summarised by the phrase that the 'trusted submissions will have preference'. In a sense, as an inner circle of 'trusted individuals' emerges, those in the outer circle will be unable to reach the inner circle and will hence be denied influence over the direction of the project. An example of where

---

[25] Moody (2002)

[26] Fragment of a Linus Torvalds posting on the Linux kernel traffic forum http://www.lib.uaa.alaska.edu/linux-kernel/archive/2002-Week-44/0094.html Accessed 8/11/2002.

this kind of structure would become apparent is when a project integrator has to choose between two contributions addressed to the same problem - one from a 'trusted lieutenant', and the other from an outsider to the inner core of developers. In economic terms, the opportunity cost of having to assess both in technical terms may be higher than the heuristic of adopting the contribution from the 'trusted' individual, a heuristic clearly implied by Torvalds in the above quote.

We would suggest, therefore, that projects gain stability as (more or less formal) hierarchies, composed of the integrator and groups of trusted individuals, emerge. These individuals reduce demands on the integrator by acting as filters for submissions coming from lower (less trusted) ranks in the community and by providing contributions that the integrator can assess and accept more easily (because of his trust in the senders). This results in a relative rise in the barriers to entry for individuals outside the inner circle, as participation, the main avenue through which knowledge about a project is acquired, becomes more difficult. Or, in Lave and Wenger's terms, the space for peripheral legitimate participation diminishes. Less desirable tasks such as 'documentation' or 'helping users' may remain open to these outsiders, and may be the only means by which they can hope eventually to gain entry into the inner sanctum of the core developers.

The most obvious consequences of such hierarchies, inside our framework, are the impossibility for outsiders to acquire authority inside the project (as previously discussed, authority is obtained through participation), and a reduction in the scope for potential challenges to the decisions of the integrator.[27] Their lack of authority denies them the opportunity to fork the project even when they completely disagree with the integrator's decisions, as their chances of attracting a significant fraction of the community to their alternative branch of development will necessarily be very low. Therefore, we find a stabilisation of the project under the control of the inner circle of developers. Evidence based on the importance of contributions, shows the existence of a high concentration of most of the work in a few individuals. This is usually attributed to the existence of differentials in effort and quality of work between core developers and outsiders, which, in this new framework, can be seen to be the result of discrimination against the latter.[28]

[27] The possibilities for conflict remain latent in the inner circle, where authority is concentrated in a few individuals, as disagreements may emerge, but given the existing trust and mutual understanding between its member, we would expect them to be resolved more easily.

[28] See Koch, S. and Schneider, G. (2000).

5 Synthesis and Critique

In the previous sections we have presented a theoretical framework that describes the sources of authority, conflict and hierarchy in open source projects, as well as empirical evidence that supports some of the processes and dynamics implicit in it. In this section, we compare this framework with those offered by von Hippel and Raymond with the aim of summarising our main claims and setting them in the broader context of existing research on the open source phenomenon. The two main issues we have dealt with in this paper are the 'structure' and 'motivations' for participation and the relation of participation to authority.

5.1 The Role of the Progenitor in von Hippel and Raymond

Most of the analyses that try to explain the advantages mentioned in the previous subsection rely on a very strong assumption about the conditions in which open source development takes place, namely, the existence of a community of individuals willing to contribute by their efforts to the sustained development of a particular program. Issues related to the creation, evolution and/or decay of these communities are, therefore, not dealt with in detail. They also tend to ignore the actual organisation structures adopted by open source communities. The use by von Hippel of the term 'horizontal network' is in a sense misleading, given that it fails to acknowledge the fact that there are hierarchies in open source projects, even if they are much more flexible than those that can be found in traditional software development organisations. The institutional dynamics inside open source communities are influential in their evolution and their success or failure. A more detailed examination of these processes challenges optimistic assumptions about the spontaneity of the appearance and progress of open source communities.

In his work, von Hippel (2002) proposes that one of the defining characteristics of 'user innovation networks' is the absence of intermediate agents (such as specialised innovation manufacturers) providing users with the innovations they require (this being associated to potential information-transfer and principal-agent problems). Instead, he describes a world of 'users/self-manufacturers' able and willing to develop innovations better suited to addressing their own needs. These innovations, under the conditions described above, end up being left in some vaguely defined 'public pool' (that is, made 'freely available') from which they can subsequently be extracted by other users.

We contend that this constitutes a very imperfect picture of the open source environment, or at least one seen from too far away to be accurate and precise. It should be remembered that in the case of open source development, most of the 'innovations' contributed by collaborators are simple 'bugs', 'patches' and 'enhancements' to existing programs. These 'minor innovations' will generally only be useful in the context of the project to which they are submitted. It is here that the figure of the 'project administrator' or 'project leader', whose existence tends to be ignored in von Hippel's analysis of the open source model, becomes relevant. The main task of this administrator is to integrate the contributions of developers in functioning programs ('innovations'), which are afterward released, tested, improved and re-enhanced by the supporting community in what we could consider to be the 'open source cycle'. Therefore, it is useful to ask whether there is an intermediate agent between users and the innovations produced by open source communities.[29]

[29] Unless, of course, we consider users developing complete functioning programs by themselves and then releasing them as 'open source'. Even then, from a dynamic perspective, these programs will be

The presence of individuals fulfilling this role and operating by 'folding back innovations' contributed by users into a functioning program, puts the structure of open source communities closer to the second 'vertical' model proposed by von Hippel and mentioned in Section 2.[30] In other words, each open source community may be seen as a network of 'users/innovators' bound together and co-ordinated by a central figure or cadre serving as an 'integrator'. The rationale for users developing their own innovations is obvious; their capacities to implement these innovations *and* maintain the integrity and continuity of a particular development effort are, however, dubious. In the free dissemination of those innovations, gaining reputation, selling complementary assets (i.e. skills and knowledge about the software program) in the labour market or influencing a standard appear to be potentially powerful incentives that would make contributing to the project a rational strategy.

In departing from von Hippel's analysis, we are proposing that the costs of turning the different innovations into useful software programs would be too high in a 'horizontal network', that is, a completely decentralised structure. For example, individual's would face considerable costs from constantly searching for, assessing and integrating innovations contributed by the rest of the community - we could expect the results to be a large collection of highly personal and idiosyncratic, potentially incompatible programs. In the case of open source software, the integrator acts as a filter for the innovations generated inside the community of users. The integrator may have little or no role in co-ordinating or specifying contributors' efforts (which are of a voluntary nature). Instead, the integrator co-ordinates the results of these efforts – the actual innovations developed and made freely available in a useful software program.

Raymond's (1999) analysis assumes that an individual gains authority over a project by creating a first working version, by gaining the trust of the previous owner and the right to inherit it, or by greatly improving it in the case of its abandonment by the previous owner. The justification for ownership follows the classical philosophical scheme that asserts that the result of an individual's work, in this case the project, rightfully belongs to her.[31] We acknowledge that the logic of Raymond's argument regarding the legitimisation of authority is essentially correct, but we contend that it is incomplete in the context of the widespread collaboration required for open source projects and the meritocratic and technical orientation of the community of practice of software designers or 'hackers.' In our opinion, it is useful and clarifying to introduce an intermediate variable between 'contribution' and 'authority' over a project, which we call 'distributed authority.'[32]

We argue that by contributing to a project, an individual gains knowledge about its inner workings (technical knowledge) and also about the rules governing the community engaged in its development (social knowledge). Contribution, in addition to being a way of learning, is also in this context a direct method for individuals to signal that they are technically proficient and skilled. Given the great value hacker communities give to those traits, we can

---

reviewed and improved by their users, the intermediate figure responsible for the integration of patches and enhancements submitted appearing on stage again.

30 Although the redistribution will in most cases be devoid of 'commercial' motive.

31 Raymond acknowledges his debt to the classical 'property rights' tradition by setting his own work inside the Lockean 'Theory of Property'.

32 Our definition of 'contribution' is broad, encompassing not only coding, testing and debugging, but also documenting, helping new users, etc.

see contribution, with its epistemological connotations (as a way of acquiring and signalling knowledge) as a source of authority over the project that is independent of the property right interpretation offered by Raymond.[33] In addition, there is a less obvious way in which 'contributing' to a project may become a source of authority. Following Lessig, we could say that if there are rules embedded in software code (technical assumptions that determine the permitted behaviours of both users and future developers), those who code become legislators and obvious figures of authority. Therefore, an individual who contributes to a project determines to an extent the possibilities for future development, by embedding his assumptions, rules and decisions into the program.[34]

5.2 The Role of the 'User' in the Open Source System of Distributed Authority

In our framework we have described the organisational structure of an open source project as a set of hierarchical layers the topmost of which encompasses the project leader and a core of trusted elite developers. The latter have earned this trust and the ensuing rank through continued and reliable participation in the project. Their contributions have signalled their knowledge and skills, as well as an alignment with the objectives and 'vision' of the leader. In this structure of inner and outer layers, the status of users, who by definition lie outside the 'development sphere', is problematic. In the pragmatic environment of 'code first and talk later' present in open source projects, users, who are less technically knowledgeable, are often regarded with indifference, derision or outright hostility. In the context of legitimate peripheral participation and distributed authority described above, the space for participation of individuals lacking a minimum of technical skills may be severely limited or even non-existent.

This ostracising of users and the lack of attention to their feedback might be seen as one of the main reasons for the usability problems that are commonly associated with open source projects. Given the lack of alignment between the vision and intentions of developers regarding a particular program, and user's needs, and given that users lack the required skills to obtain the authority necessary to introduce their preferences in the program (by participating in its development), users' needs are neglected (or insufficiently addressed).

These problems offer an opportunity for the commercial software firms that have adopted open source business models and therefore are concerned with the success and acceptance of open source software. These firms may become mediators between large markets of users (with a set of defined usability needs) and the technically 'savvy' developer community. By participating in development (and therefore obtaining authority in a project), creating usability labs and sponsoring the introduction of usability features into the open source development process, these firms may favour the alignment of the seemingly divergent interests of developers with those of the potential users. The effort that IBM has devoted to

---

[33] For example, the founder of a project will be the individual with the highest authority over it, as, having created the first working version, she/he will undoubtedly be more knowledgeable about it (at least during its initial stages as we have already seen).

[34] An obvious example of this is the way developers with highly idiosyncratic and tacit programming techniques may become essential in the development of a program because no one else can understand their contributions, this dependence on them becoming an important source of authority. In this respect, the vision referred to at several points in this paper has much to do with Alan Kay's 1971 observation 'Don't worry about what anybody else is going to do… The best way to predict the future is to invent it. Really smart people with reasonable funding can do just about anything that doesn't violate too many of Newton's Laws!'

Linux accessibility through the creation of an IBM Linux Accessibility Team is one example. Similarly the support given by firms such as Red Hat, Eazel, Helix or Sun to the GNOME project, aimed at creating an Open Source Graphic User Interface and boosting the usability of the Linux Operating System, might also be seen from this perspective (German 2002).

5.3 Implications of Distributed Authority for Research and Policy

This paper has explored the potential of the open source movement to serve as a new paradigm for the division of labour in software development. We concur with the existing literature with regard to the sources of technological advantage, but we highlight the limited nature of these advantages. Open source is able to establish a closer connection with users that have a direct interest in the design of the product by enrolling them in the software's creation. It is also able to improve the processes of error identification and correction when it is successful in recruiting large numbers of individuals to become involved in such activities. This paper has, however, emphasised that participant-users are a relatively small share of all users and may not more accurately represent the larger population of users needs than software designers organised in other ways, including proprietary software production. In particular, when non-conflicting user needs are widely shared the closer connection achieved by enrolling users in software production does not seem beyond the capabilities of proprietary software producers or antithetical to existing divisions of labour. We have argued that the enrolment of relatively large numbers of individuals is fairly rare, citing evidence from previous work as well as inspection of SourceForge. Thus, commercial software producers, who may choose to share, on a restricted basis, source code with 'lead' users and others believed to have the technical capacities to make useful contributions may imitate such processes. Our conclusions about the technical advantage of open source software development, therefore, stop short of concluding that it represents a new paradigm; such advantages as it has may be re-absorbed within the existing division of labour.

The remaining advantage of the open source development process is organisational, and it is here that we find the strongest evidence for the possibility that open source software represents a new paradigm for the division of labour. By coupling a flexible recruitment process based upon voluntary participation with a means for individuals to achieve higher levels of authority as the consequence of their contributions, open source development efforts may evolve into voluntary virtual communities with shared ethos, practices and aims. This method of accumulating talent and building a development team may have unique advantages compared with other methods of accomplishing the division of labour. We have found it useful to describe the process of accumulating reputation leading to the exercise of authority as the creation of a system of 'distributed authority.'

There is an important distinction between Raymond's (1999) property rights model, which confers authority on the progenitor and project leader, and the model we offer of distributed authority. Raymond's explanation relies upon cultural norms to explain the infrequency of forking and upholds the principle that ultimate authority devolves from property rights. Our explanation emphasises the process of challenging authority and winning a negotiated truce in which participants may or may not share ideological convictions regarding the progenitor's property rights over the project. This distinction may be seen as particularly important in efforts to promote or 'spark' the development of open source-type developments in new areas of software development or in the production of other information goods. While Raymond's model would suggest the importance of cultural homogeneity in terms of norms and values concerning the progenitor's authority, our interpretation stresses the vital importance of

adopting effective rules both for making and for resolving challenges to the progenitor's authority over the project.

The first important implication of this work for further research is the importance of developing comparative histories of the development of 'distributed authority' in different open source communities. It will be possible to use these histories to map out the mechanisms for stimulating initiative (and hence challenge to authority) and for resolving the conflicts over ends and means in these communities. Even though without detailed history, a variety of quantitative evidence will soon become available from the FLOSS study regarding the motives and practices of open source developers that will allow a more precise specification of the evolutionary processes governing open source community development.

A second implication of this paper for further research is the need for a more complete model of the industrial dynamics of such communities. A better understanding of the origins or 'birth processes' by which such communities are formed is needed. This would be useful in explaining whether the high incidence of single developer communities to be found in SourceForge is the consequence of the idiosyncratic ideas of aspiring progenitors or some shortcomings in the process of recruitment. An example of the latter, would be that joining an ongoing community is seen as conveying higher opportunities for a developer than helping 'ignite' the efforts of a fledgling community. The growth to maturity of open source communities involves competition with other projects for the human resources of participants, and an understanding of this competitive process is another feature of the research that should be undertaken on the industrial dynamics of open source communities. For example, when the more talented and innovative software developers are unable to satisfy their needs or interests within a community because their efforts do not admit them to positions of authority and hence allow them the possibility to shape the outcomes of the project, they are likely to join another community. Finally, more knowledge is needed about the demise of open source communities. It seems likely that one source of demise is a continuous churn of initiatives and ideas that create candidate communities which experience partisan divides before they are able to 'ignite' into a sustained development effort. Identifying the sources of demise at other stages in open source community lives is also an urgent and important task. As the open source movement matures, it will become more important to investigate what happens to projects that become technologically obsolete or that, despite widespread use, lose a viable community of developers to maintain and upgrade them as computer hardware systems evolve. The latter case clearly offers the opportunity, under some form of open source licence, for proprietary developers to take up the development effort.

A third area of research suggested by this paper is the extension of open source community models to other types of public information goods production. The capabilities of the Internet for supporting collaborative creation processes and distributing their results are relevant for a wide range of other collaborative activities including scientific research, the development of educational materials, the production of cultural expressions, and the creation of texts, archives, and libraries. Many of these activities may also require public funding support to achieve a critical mass but will achieve considerable benefits relative to these costs if the open source-type collaborative processes can be effectively applied to them. The new paradigm of open source-type collaboration may, indeed, be only in its infancy as a source of public information goods.

References

Bezroukov, N. (1999). 'Open Source Software Development as a Special Type of Academic Research (Critique of Vulgar Raymondism),' *First Monday* (http://www.firstmonday.dk/issues/issue4_10/bezroukov/index.html): Accessed 1 Feb. 03.

Cowan, R., P. A. David and D. Foray (2000). 'The Explicit Economics of Knowledge Codification and Tacitness,' *Industrial and Corporate Change* **9** (2): 211-254.

David, P. A. (1993). 'Path Dependence and Predictability in Dynamic Systems with Local Network Externalities: A Paradigm for Historical Economics' in D. Foray and C. Freeman, *Technology and the Wealth of Nations*, London, Pinter Publishers**:** 209-31.

Edwards, K. (2000). 'When Beggars Become Choosers,' *First Monday* (http://www.firstmonday.dk/issues/issue5_10/edwards/index.html): Last Accessed 1 Feb 03.

Edwards, K. (2001) 'Epistemic Communities, Situated Learning and Open Source Software' http://opensource.mit.edu/papers/kasperedwards-ec.pdf. (Last accessed 01/02/03).

German, D. M. (2002). *The Evolution of the GNOME Project*. Workshop Proceeding presented at 'Meeting Challenges and Surviving Success: The 2nd Workshop on Open Source Software Engineering' (May 19-25), Available at http://opensource.ucc.ie/icse2002/German.pdf.

Haas, P. M. (1992). 'Introduction: Epistemic Communities and International Policy Coordination,' *International Organization* **46** (1): 1-36.

Himanen, P., L. Torvalds and M. Castells (2001). *The Hacker Ethic*. New York, Vintage.

Hirschman, A. O. (1970). *Exit, Voice, and Loyalty: Responses to Decline in Firms, Organizations, and States*. Cambridge MA, Harvard University Press.

Koch, S. and Schneider, G. (2000). 'Results From Software Engineering Research Into Open Source Development Projects Using Public Data', Vienna University of Economics and Business Administration http://opensource.mit.edu/papers/koch-ossoftwareengineering.pdf.

Kogut, B. and A. Metiu (2001). 'Open-Source Software Development And Distributed Innovation,' *Oxford Review of Economic Policy* **17** (2): 248-64.

Krishnamurthy, S. (2002) 'Cave or Community? An Empirical Examination of 100 Mature Open Source Projects' University of Washington, Bothell. May http://opensource.mit.edu/papers/krishnamurthy.pdf,.

Lave, J. and E. Wenger (1991). *Situated Learning: Legitimate Peripheral Participation*, Cambridge University Press.

Lerner, J. and J. Tirole (2000) 'The Simple Economics of Open Source' Cambridge MA, National Bureau of Economic Research, Working Paper 7600.

Levy, S. (2001). *Hackers*. New York, Penguin Books.

Mateos-Garcia, J. and W. E. Steinmueller (2003a) 'Dynamic Features of Open Source Development Communities and Community Processes', Brighton: SPRU -- Science and Technology Policy Studies, Open Source Movement Research INK Working Paper No. 3. February.

Mateos-Garcia, J. and W. E. Steinmueller (2003b) 'Applications of Open Source Software Development Methods to Other Information Goods,' Brighton: SPRU -- Science and Technology Policy Studies, Open Source Movement Research INK Working Paper No. 2. January.

Moody, G. (2002). *Rebel Code: Linux and the Open Source Revolution*. New York, Penguin Books.

Nichols, D. and Twidale, M. (2003). "The Usability of Open Source Software", *First Monday* (http://firstmonday.org/issues/issue8_1/nichols/index.html): Last Accessed 5 February 03.

Raymond, E. S. (1999). *The Cathedral and the Bazaar: Musings on Linux and Open Souce by an Accidental Revolutionary*. Sebastopol, CA, O'Reilly and Associates, Inc.

Rosenberg, N. (1976). 'Factors Affectign the Diffusion of Technology', *Perspectives on Technology*, Cambridge University Press**:** 189-210.

von Hippel, E. (2002) 'Horizontal innovation networks - by and for users' Cambridge, MA, MIT MIT Sloan School of Management, Working Paper No. 4366-02. June.