Technical Report CS-TR-760 (February 2002)
Department of Computing Science
University of Newcastle upon Tyne

Dependability Issues in Open Source Software
DIRC Project Activity 5
Final Report

Budi Arief[1], Diana Bosio[2], Cristina Gacek[1], and Mark Rouncefield[3]

[1] Department of Computing Science
University of Newcastle upon Tyne
Newcastle upon Tyne NE1 7RU

[2] Centre for Software Reliability
City University
London EC1V 0HB

[3] Department of Computing
Lancaster University
Lancaster LA1 4YR

1. Introduction

This project activity was worked on by 3 DIRC sites, City University, Lancaster University and the University of Newcastle. The main aim of this project was to determine whether a full project activity in the area of development of systems by open source approaches would be worthwhile in the DIRC project. In order to be able to make such a recommendation, different objectives were set in the project description [12]: establishing links with groups involved in open source; gathering data from one small project; reviewing the role of people and their interaction; analysing opinions and facts; establishing how the effectiveness of this method of software development would be measured; and looking at open source in a broader context.

Some of the more specialized objectives were worked upon by separate sites, whereas all sites looked into the more general issues.

The activities performed by Newcastle within Project Activity 5 included reading various literature sources on the open source subject in general and on individual projects, interviewing several people involved with open source while playing different roles and/or in different projects, giving presentations to and getting feedback from external audiences, and holding some discussions with one of DIRC's Senior Visiting Fellows, Michael Jackson. There was also a participation in the first Workshop on Open Source Software Engineering as part of ICSE 2001, with a position paper about software architecture issues in open source [5].

Lancaster University has been mainly involved in the investigation of psychology and sociology of groups; in particular a study of the Cocoon open source development project was conducted. Cocoon is a Java publishing framework for the creation of web content using latest technologies such as XML.

City University's effort was spent in: clarifying issues of quantitative description and evaluation of dependability; researching examples that would support or serve as counter-examples for claims generally made about open source products which may be intuitively plausible but not generally true; generating a list of research questions that appear worth investigating and would contribute towards identifying "openness" process factors which affect product dependability. Towards this last goal, City built a speculative model of the reliability growth of a product, which takes into account the multiplicity and diversity of usage profiles involved in fault finding by execution of the software.

All of the above activities will contribute to the proposed follow-on Project Activity.

This report presents the findings of this investigation by reporting on the main activities that have been undertaken and presenting our informed final recommendation on a follow-on project activity. It is structured in the following way. Section 2 explains the obstacles encountered while trying to understand the term "open source", contacts pursued and projects observed with respect to open source. Section 3 presents insights into the sociology of open source software development, whereas section 4 describes observations drawn and main issues identified for open source software development and dependable systems engineering. Finally, section 5 explains our recommendation together with the reasons behind our decision. Further insights on the activities described in this report, as well as various papers that have been written in relation to this activity can be found in the appendices A - E.

2. Understanding Open Source

The term *open source* is widely applied to describe software development projects. The lack of a precise definition was an obstacle towards an immediate in-depth investigation into dependability issues in open source. By taking a multi-disciplinary point of view, we proposed a collection of characteristics that are common, as well as some that differ among open source projects. The summary of these characteristics can be found in section 2.1.

This collection of characteristics resulted from our effort to understand open source, which was the outcome of a number of activities. We interviewed several individuals, some of whom were involved with open source projects in their free time, and others as part of their paid job. We also did an extensive literature review and observed web sites supporting individual open source projects. The main findings of these activities are outlined in section 2.2. Additionally, as a side effect of these activities, we have established contacts with academic and industrial researchers also investigating open source. These contacts are briefly mentioned in section 2.2.3.

2.1. What really is Open Source?

Given the task of looking into claims of increased dependability in open source projects, we read several information sources on the subject. A fact that became clear very early on was that the term open source is widely applied to describe several software development approaches, rather than providing a precise definition of a single approach used to support software development projects. A definition for the term open source by the *Open Source Initiative* (OSI) exists [2], it does address legal issues extensively and encompasses some economic aspects, but it hardly touches on computing science related aspects and completely ignores the areas of management, psychology, social and organisational sciences. These other aspects must be considered if one is to support claims about software dependability. As a result of this observation we spent considerable effort looking into several large projects conforming to the definition put forth by the OSI. This led us to propose a collection of characteristics that are common, as well as some that differ among open source projects. The set of the relevant characteristics that we found is summarised below.

The common characteristics are:

- *Open Source Definition*: all projects that are considered to be open source follow this definition, which covers licensing issues and is provided by the OSI[1].
- *Community*: active open source projects have a well-defined community of users and contributors, in the sense that they share a common interest.

[1] This is clearly a consequence of the fact that our criteria for observing projects were their conformance to the OSI definition.

- *Motivation*: work evolves because the contributors have some kind of motivation, even though the drivers might vary.
- *Developers are users*
- *Process of accepting submissions*: open source projects have some form of process for accepting submissions of different artefacts (e.g. bug reports, source code, etc.).
- *Development improvement cycles*: product improvement is manifested in both breakthrough and continuous improvement modes.
- *Modularity of code*: this supports effective remote collaboration and concurrent work.

There are also some characteristics that vary from project to project:

- *Choice of work area*: some open source projects only accept solicited contributions, whereas others also accept spontaneous contributions.
- *Balance of centralisation and decentralisation*: open source communities are organised differently, some employ a strict developer hierarchy, while others have loose organisational structures.
- *Meritocratic culture*: knowledge shown by means of contributions increases the perception of merit, which in turn leads to power. This transition occurs in different ways from project to project.
- *Business model*: this could be for own use, packaging and selling, or as a platform for commercial or research software development.
- *Decision making process*: variations can be observed based on the quality goals, the acceptance criteria enacted, the cognitive abilities of the decision group, and the social structure within the project.
- *Submission information dissemination process*: this can be done passively (through newsgroup or commented code), actively (using emails and mailing lists) or by using some dedicated web space.
- *Project starting points*: open source projects may start from scratch or from older, extant systems.
- *Visibility of software architecture*: open source projects may have explicit architecture and design documents available or not.
- *Documentation and testing*: the effort spent on these areas varies widely, some projects have extensive documentation and testing resources whereas others put little effort into these.
- *Licensing*: different licensing terms are used, with different levels of flexibility.
- *Operational support*: open source projects use various supporting tools (web pages, mailing lists, CVS, etc.) to varying degrees.
- *Size*: the differences are both in terms of involved-community sizes and code base sizes.

This work culminated with the writing of a paper on the topic [17], one of the papers that are appended to this report, see appendix A.

2.2. Contacts established and projects observed

While working on this project activity, we observed several existing open source/free software projects and efforts, interviewed several individuals involved with open source projects, as well as made contact with several people doing research on open source. In this section, we discuss our findings from this undertaking.

2.2.1. Contacts pursued

We were able to gather very interesting information with respect to various open source projects using sources such as open literature, the world wide web, individuals contributing to open source projects either on their free time or as part of their day time job, as well as individuals in contact with open source software during their daily jobs but not as contributors.

By looking at the Apache HTTP Server, Cocoon and NetBSD projects we were able to learn things about open source software projects in general, as well as understand some of the community and social aspects around them. While interacting with people from the HP-Arjuna Lab we were able to gain some insight on the aspects that play a role on getting corporations involved in open source software projects. Further insight with respect to security aspects and perceived reaction time upon the discovery of a flaw in software systems were obtained via an interview with an IT Security Coordinator.

We also had some conversations with people involved in the GENESIS project. We were unable to further our knowledge on open source software from that exchange, but it became clear that there may be some mutual benefits from future contacts with that project. Maintaining contact with the people from the HP-Arjuna lab may also prove mutually fruitful in the future.

Details about the specific information relating to each of the above can be found in appendix B. Some of the conclusions drawn from this effort can be found on section 4.

2.2.2. Contacts dropped

Not all of our attempts proved fruitful. We did contact the group at the University of Newcastle involved in the SSETI (Student Space Exploration & Technology Initiative) project. SSETI is a student project aimed at obtaining the distributed design, construction and launch of micro-satellites. It involves multiple sites and multiple disciplines collaborating towards a common goal and sharing the results obtained in the various fronts of endeavour, but it is not open source. Its results will not be made available nor accept contributions of individuals outside its closed community.

We also had some discussion with people involved in the PRODIGY system. PRODIGY is a computer-based decision support system for general practitioners (GPs). Its focus is on assisting GPs on the process of generating diagnosis and providing the appropriate treatment prescription. PRODIGY incorporates information from literature on diagnosis and treatments, but does not allow people to actively submit information to be added to the system. Its source code is not developed using an open source approach and its knowledge base uses open source knowledge but is not evolved using an open source approach. At some point in time there were discussions towards making PRODIGY open source, but for fear of losing their competitive advantage, this idea was abandoned.

No further contacts were made with either of these groups because they were not really working on open source. Future exchanges might be fruitful, depending on the openness factors that we decide to pursue further, if any.

2.2.3. Academic/industrial contacts

As a result of submitting a position paper on software architectural issues and open source (see appendix D [5]) for the 1st Workshop on Open Source Software Development at ICSE 2001 Toronto, a number of contacts were made possible.

Firstly, the workshop organizers, Brian Fitzgerald and Joseph Feller, from University College Cork in Ireland, asked us to review their forthcoming *Open Source Software Development* book [15] and are presently considering future visits to Newcastle to discuss a collaboration with DIRC.

Secondly, another member present at that ICSE workshop event, Francis Hunt attended our 3rd open source software project meeting in July. He is a member of the Centre of Information Technology Management at Cambridge University (Institute for Manufacturing). He and his colleagues are particularly interested in how open source software processes can be successfully introduced into traditional organisations [25].

The HP-Arjuna Lab (see section 4 of appendix B) and the GENESIS project (see section 6 of appendix B) also provide us with some potential future collaboration opportunities.

Depending on the focus and direction of future DIRC project activities some of these contacts might prove to be worthwhile to pursue.

3. The sociology of Open Source development

Sociology as an analytic discipline concerns itself with descriptions of how the social world is. The sociological examination of open source used a variety of data (the website, the code, the email archive and interview material) from the Cocoon open source project to explicate the ways in which open source projects are accomplished and the ways in which the various participants observably, reportably, accountably orient their actions, their contributions to the project as a whole. In other words, we did not set out to 'discover' new facts about open source, so much as to acquaint (or re-aquaint) ourselves with what any competent member of an open source project already knows. One of our main interests was to use our studies to compare open source development with more traditional software approaches as revealed through ethnographic and ethnomethodological studies of software production. Our research provides an interesting and important contrast with idealised versions of the open source project, for when seen as a practical (as opposed to philosophical) project, open source development can be seen to be little different to traditional software production.

Our second interest was in exploring the notion and existence of an open source 'community' as a possible source of dependability. While the data suggests an open source community can clearly be seen to have an 'ethic' or 'code of practice', this appears less a determinant of behaviour within the community than a resource that is drawn on in discussion within and without the community. Nevertheless, when viewed in terms of 'dependability', open source can be seen to have some advantages in terms of the observed and documented multiple scrutiny of code (Linus' Law). The data also suggests (at least in the case of Cocoon) other factors within an open source project that might impinge on 'dependability'. So, for example, within Cocoon there are clear lines of responsibility, without any obvious 'blame culture' and communication, both through email and the documentation attached to code releases, appears to be regularly maintained.

3.1. Studying the Cocoon Open Source Development Project

There is a large and burgeoning literature on the challenges faced by researchers when investigating 'virtual' or online communities and the appropriate methods for study [23]. The challenge arises from the nature of these communities - in our case an open source community - the fact that they transcend conventional notions of geographical, physical or temporal space, and therefore lack many of the traditional characteristics of 'real' communities poses

methodological problems. The approach traditionally followed at Lancaster has been that of ethnomethodologically informed ethnography [24]. A distinguishing characteristic of ethnographic approaches is the researcher's immersion in the field of study and the provision of a 'rich picture' through naturalistic descriptions of domains. The general advantage claimed for such an approach lies in the 'sensitising' it promotes to the real world character and practical context of activities. Ethnomethodologically informed ethnography [24] involves a focus on the study of *doing the work,* and in the ways in which it is done in *actual practice*, as opposed to work in *idealised* form. Clearly this can prove problematic in the study of online communities that are not easily open to observational techniques.

The ethnomethodological programme of inquiry [18] focuses on description of the local, in situ organisation of activities and the ways in which any sphere of practical action is produced and managed `from within' as a recognisably and reliably orderly social environment. Ethnomethodological studies investigate social settings – in our case an open source software project - for their `accomplished orderliness' as socially organised environments of practical conduct. Ethnomethodological analysis proceeds through the faithful description of the social practices through which the witnessed activity was observably produced and achieved which in this case are largely through the medium of email. As [35] suggests:

> "Ethnographies have always taken advantage of written materials from a culture, but that has usually formed only a part of the evidence for analysis. Online communities present the researcher with nothing but text. The ethnographer cannot observe people, other than through their textual contributions to a forum. All behaviour is verbal in the form of text. There are no other artifacts to analyze other than text."

Without going into the debate about whether there is anything 'beyond the text', whether ethnography is simply text and the task of ethnography simply 'decoding', it seems obvious that careful analysis of online discourse and text - the examination of the email archive, the website and the code - can produce an appropriately 'thick description' of the Cocoon open source community.

In a similar fashion there is a growing social science interest in and varying approaches to understanding the open source software phenomenon. While Economics appears frankly baffled by it - reverting to revisiting outdated notions of 'public good' or 'gift exchanges' - there appears to be no shortage of sociological approaches. French and Thrift for example - in 'The machine in the Ghost' [16] - begins by outlining the lack of sociological interest in software:

> "First, software takes up little in the way of visible physical space. It generally occupies micro-spaces. Second, software is deferred. It expresses the co-presence of different times, the time of its production and its subsequent dictation of future moments. So the practical politics of the decisions about production are built into the software and rarely recur at a later date. Third, software, is therefore a space that is constantly in-between, a mass-produced series of instructions that lie in the interstices of everyday life, pocket dictators that are constantly expressing themselves. Fourth, we are schooled in ignoring software, just as we are schooled in ignoring standards and classifications. Software very rapidly takes on the status of background and therefore is rarely considered anew."

But concludes by stressing (and exaggerating) the role of software in modern life:

> "... as a more practical extension of human spaces, consisting of three different processes. The first is a simple extension of textuality. Modern western cities are effectively intertextual - from the myriad forms issued by bureaucracies, through the book, the newspaper and the web page, through the checkout till roll and the credit card slip to the letter and the e-mail, the city is one vast intertext. Cities are quite literally written and software is the latest expression of this writing passion. Second, software is a part of the paraphernalia of everyday urban life revealed by the turn to the noncognitive. It is one of those little but

> large technologies that are crucial to the bonding of urban time and space, technologies like the pencil ... and the screw ... which in their very ubiquity go largely unnoticed. We can think of software in this way - as a holding together accomplished through the medium of performative writing. Third, we can see software as a means of transport, as an intermediary passing information from one place to another so efficiently that the journey appears effortless, movement without friction [27]".

Unfortunately the growing number of sociological studies often tell us more about Sociology and its internal disputes than they do about online communities or open source development. That it is a difficult and complex subject is beyond dispute. However, the problems the social sciences attempt to tackle when considering these kinds of issues primarily evolve around an overwhelming concern with *internal* debate - about who has the 'best' theory. The dilemma thus posed is a product of the methodological choice to attempt to give *explanatory* accounts of social life. They set themselves up to settle explanatory questions and in so doing they are not so much involved with actually explaining anything connected to 'real world, real time' activity but instead are more concerned with questions about *the form of explanation*. Instead of examining what it is about human activity and human interaction that make open source development the recognisably distinct phenomena it is understood to be by those involved, the phenomena becomes yet another incidental area in which to observe theorised social forces and processes at work, just another tool for illuminating topical sociological theories. Whenever sociological theories and methods are brought into play to produce social science accounts of phenomena the phenomena itself becomes hard to recognise - that is, "It's life Jim, but not as we know it". Our interest however, is in explicating, through ethnomethodologically informed ethnography, what might be characterised as the 'missing what' in all these studies.

> "As a non-ironic sociology, ethnomethodology takes seriously the great questions of sociology: How do actions recur and reproduce themselves? How is it that interaction displays properties of patterning, stability, orderliness?How does social life get organised? For ethnomethodology this orderliness must be seen as arising from within activities due to the work done by parties to those activities" [6].

Our analysis then was drawn from interview, source code and email archive data. We were fortunate in having access to one of the 'committers' on the Cocoon project and a number of informal interviews were conducted. Similarly we were fortunate in being able to contact and correspond with the originator of the Cocoon project. We use this data to explicate the ways in which open source software projects are accomplished and the ways in which the various participants observably, reportably, accountably orient their actions, their contributions, their emails to the project and to notions concerning 'good' or 'elegant' code, ideas about 'ownership' and so on. We thus adopt an ethnomethodological approach, which, with its emphasis on the achievement or accomplishment of orderliness appears to have a particular resonance with open source software development, and has already featured in an understanding of the process of reading and writing computer programs, and in software project work [11].

Button and Sharrock [11] particularly highlight the importance of 'the project' in software engineering suggesting that:

> " the project is a prominent way in which engineering work is socially organised so as to confront the sorts of contingencies that face software engineering that we have alluded to such as the threatened curtailment because of, for example, drastic slippage, or such as the pressures to abandon good practice." (p 372)

We suggest that the notion of 'the project' has strong relevance to open source software development. Our interest was in documenting the *ordering practices* commonly found in the Cocoon open source software project.

".. devices which will provide 'orderliness' in the conduct of work and in ensuring that such devices can be implemented and enforced. These devices are meant to enable the achievement of orderly work where it requires the collaborative participation of many individuals ". ([11] p 373)

Since our interest is primarily in understanding the practical character of the open source software project as a 'project' - how it actually 'gets done', the contrast we draw is not with romanticised or idealised views of open source development [29]. Instead our comparison is with ethnographic and ethnomethodological studies of 'realworld, real time' software production [11]. These emphasise the project as a practical, ongoing achievement and concentrate on the everyday, mundane aspects of keeping a project going. We place particular emphasis on various kinds of 'ordering work' that occurs at a number of levels throughout the project and draw attention to the set-up of the website, coding, email correspondence and the project archive.

3.2. Achieving the orderly character of open source project work

In their paper on the organisation of collaborative design and development in software engineering, Button and Sharrock document how engineers achieve the formatted arrangements of the project and how they display an orientation to these arrangements in the way they order and accomplish their work [11]. In project work the organisation of the work itself can be a source of troubles that is accommodated through the organisation and re-organisation of work. Ordering work as a project does not in itself ensure the orderliness of work or provide remedies for all contingencies, instead the project structure and plan is an achievement of everyday work and a response to and recognition of the contingent nature of such work. In these circumstances a number of devices are noticeable for ensuring the orderly character of work. 'Phasing' ensures that necessary tasks are adequately completed and provides for the interdependence of activities and the recognition of uncompleted stages. The 'methodic handling of tasks' provides for some kind of system in the confrontation and elimination of problems. 'Orienting to the project as a totality' provides a method for project teams to keep each other's progress in view and make it visible to others. 'Measured progression' refers to procedures and devices - organisational metrics - for documenting how much of the project has been done and what remains; checking work against schedules and so on. Finally they note how 'making sure the documentation gets done' is regarded as 'dirty work' not an integral part of job and superfluous to engineers practical needs.

3.3. The Cocoon website as an ordering device.

The Cocoon website [1] can be viewed as an ordering device orienting both 'newbies' and established project members to features of the project through devices such as the menu-bar, the 'to do' list, requests for help, advice for contributors and so on. The advice on making a contribution for example describes a number of stages through which a 'typical contribution' may go and how any contribution is treated once submitted. The 'to do' list prioritises requirements for code, documentation, samples and design and assigns particular tasks to named individuals.

The website thus 'affords knowledge' [3], providing for project members knowledge of the state of the project, where they are up to what needs to be done, etc. - and it was evidently designed with this possibility in mind. The website is both the public focus for work and a visible, a publicly available, record of work that has been done or remains to be done. Members of the Cocoon project are able to use website and associated email system, to see unproblematically what needs to be done urgently, what is less important, what the next phase of the project is and what progress they are making. The website provides the project team with the means to see at a glance, and recognise immediately what is going on in the project. The website thereby also acts

as a 'technology of accountability' [34] enabling members to see the status of the project and calculate whereabouts they might be in the organisational and temporal cycle of events.

3.4. Examining the Email Archive: Order by email.

Examination of the email archive is also instructive of the various ways by which order is accomplished in an open source project. What is evident is the way in which email communication provides for the administration, voting and scheduling of the project as well as orienting to the project as a whole. A lot of email communication is about the scheduling of activity that in turn is influenced by the 'lazy consensus' system of voting (whereby anyone who does not vote is assumed to concur). What also comes over in the email correspondence is an orientation to the Cocoon project as a whole both in terms of the management and administration of the project as well as some notion of a 'code' or orientation to the ethos of open source in general. The email system has become a way of keeping each other's progress in view and making their own progress visible to others through activities such as involving themselves in others activities and tasks by talking them through; and knowing where their work impacted on others and informing them.

3.5. Rebel Code? The open source 'code' of work.

The development and orientation to some philosophy or notion of an open source 'code' regularly appears in the email discussions on 'good' or 'elegant' code, design philosophy and the principles of open source:

> "I think it is important to recognise that we are working on an open source project. I know that there are "code ownership" political issues in many companies, but I would sincerely hope that those attitudes would not bleed into this project. Once the code has been committed, it is no longer 'your code' it is 'our code', and we are all committed to making that code as good as possible. It's one of the strengths of open source."

The email discussions clearly document the existence of some idea of a 'code' that 'governs' or shapes open source. This is depicted in even more detail in sociological accounts of the 'hacker ethic' and is often used to provide some kind of explanatory account - 'why hackers do it'. In these approaches compliance to the 'code' is used as an explanation of behaviour. The open source community is simply seen as governed by set of normative rules. Our argument is rather different and subtler since we are not interested in offering explanatory or motivational accounts of open source but instead of understanding how these projects 'get done'. We are interested in examining how the open source community both constructs and makes use of the code as a resource in the course of their mundane interactions where the code is used by parties to the interaction as displays, or accounts of what those actions are. Orienting to the code in an email, for example, can be used for changing topic, defending or defeating a proposed course of action and for accounting for one's actions in an acceptable way. As Weider [36] in his study of 'the convict code' suggests:

> "The code then, is much more a method of moral persuasion and justification than it is a substantive account of an organized way of life.' (p 175).

3.6. Ordering devices at work in the source code

The availability of the source code is one of the most salient attributes of the open source phenomena. We therefore spent some time considering the code and examining whether it could be argued that there are ordering devices at work in the code itself. Presumably the code itself should bear the marks of the ordering devices since they afford orderliness in the conduct of work. They allow a *project* to take place, even if its documents, its 'deliverables' and the

relations between developers and users looks quite different to that envisaged by accounts of fully-equipped software engineering projects. The practices and ordering devices surrounding open source code are concerned with regulating how it is read, and channeling how it is written and re-written. Open source programmers are often encouraged to read the code, as well as reading the documentation that accompanies the code. In this domain, we expected to find ordering devices concerned with reading and writing code.

At a fairly coarse-grained level, the formatting of segments of the Cocoon Java code is obvious from the layout on the page. Clearly text formatting is a kind of ordering to do with *reading*. The formatting is a contrivance that allows the code to be read more easily by different kinds of reader. This particular code shows evidence of being ordered for at least three distinct kinds of readers.

Firstly, it affords reading by humans. By virtue of the restricted line lengths, the use of nested indentation to represent something about the flow of execution of the program, and the use of blank space to show separations between different components of the code, people reading the program can begin to interpret the code as a set of operations and structures. For such readers, particular zones of the text are marked out for different modes of reading. Any line beginning with an asterisk will attract attention as a comment, something programmers particularly addresses to human readers, including themselves. This may be an explanation, an apology or a request (e.g. *"So, if you find any better way to implement this class (clever data models, smart update algorithms,etc...), please, consider patching this implementation or sending a note about a method to do it.*"). By contrast, any line that begins with a keyword like 'class', 'public' or 'private' will stand out to a programmer since it signals an important boundary in the organisation of the program. Reading these lines involves separating out keywords, operators and syntax marks from the proper names that the programmer(s) have used to designate elements of the program. Words such as 'freememory' or 'getStatus' describe designate places where an important value is stored, or places where significant operations will be specified. On these lines, the reader is alerted that they must read the code as naming something specific to this program.

Secondly the source affords reading by the compiler. By virtue of such things as the termination of lines by semicolons, the use of brackets of various kinds - {}, [], and (), - and the presence of keywords such as 'public' and 'class', the compiler will be able to parse the source code file into an executable file containing instructions that can be used by the Java Virtual Machine. Thirdly, this code allows reading by another specialized program, javadoc. Javadoc is a program that will take these source files and generate html-formatted documents from them. These documents, the "API (Application Programmer Interface) docs" will be read by other programmers who want to use the operations furnished by this piece of code, without looking at the actual code itself.

Like the source code, the html-formatted documents will be browsed extensively by programmers involved in either using or extending the Cocoon framework. Developers involved in the open source project and technically sophisticated users (such as web-site architects and developers) will both refer to these documents. However their presentation as a web-page implies important differences. These documents are not editable, whereas the source code is. Secondly, the use of headings, hyperlinks, tables, different font sizes and types for the text is clearly directed towards quicker movement around the text.

Finally, the link between reading and writing code - through a text editor and a repository for source code - is important since almost every open source software development project

currently active makes use of a single important ordering device, a program called 'cvs', Concurrent Versioning System. This device is profoundly enabling for open source development in several respects. Itself an open source project, CVS makes it possible for almost any number of people to read and write copies of the same source code files, and amalgamate the results. CVS's developers claim both that *"its client-server access method lets developers access the latest code from anywhere there's an Internet connection"* and that *"its unreserved check-out model to version control avoids artificial conflicts common with the exclusive check-out model."* Revision numbers in the source code indicate how many times a source code file has been modified. Talk about CVS is a major feature of the email communication amongst developers. Many email messages describe events in the CVS repository. For instance, in describing a milestone release of Cocoon, the developer responsible writes to the developer list:

> > > *Now the CVS stuff:*
> > > *- I tagged the beta with cocoon_20_b1*
> > > *- I checked in the build.xml with the new version 2.1-dev*
> > > *- I made a branch of cocoon_20_b1 with the name cocoon_20*
> > > *- I checked in the build.xml with the new version 2.0b1-dev under*
> > > *the branch cocoon_20_branch.*
> > > *So the HEAD is the 2.1 version and the 2.0 is a branch.*

The developer describes in detail the operations that had to be carried out so that the software source was named in an orderly way, and accessible to other readers and writers of the code. The developer's descriptions of their actions within CVS render the contents of the archive manageable for other developers. If for instance, a particular 'build' or version of the project does not have a commonly agreed upon name, then the team of developers cannot synchronise their editing of the source code. Agreeing on what the name of the version will be is sometimes not enough. It may still leave open the question of where in the CVS repository further changes will take place. Another developer replies to the preceding message:

> > *Yes, that good. I assume all the new development will happen only on*
> > *the HEAD and bug fixes will be applied to both HEAD and 2.0b1-dev*
> > *branch. Is this the common understanding?*
>

Again, negotiations around how source code will be named, stored and retrieved are taking place here, but in this case about future changes to the code. Without these negotiations, the project would start to fall apart.

The formatting of the code, the use of Java, and the method of documenting the source (using javadoc) are all textbook or industry standard. Almost identically formatted and commented code can be found on any Java-related industry web-site, or in any Java programming textbook. At the level of the reading and writing practices carried out by programmers using text editors or integrated development environments, the ruling conventions in this open source project come from well beyond the domain of open source software projects. There is no evidence of a specific style of coding.

However, there are differences that show that this code belongs to an open source project. Firstly, there is a request to anonymous readers to contribute a better algorithm or data structure for part of the system that is said to be *'_CRITICAL for a fast performance of the whole system. ... please consider patching this implementation.'* It is unlikely that a critical component of a professional software system would publicly acknowledge that it is *'_HIGHLY un-optimized.'* The source code itself, as well as the API documents, solicit contributions and involvement in

developing the software. Secondly, the authors' email addresses are provided suggesting the possibility of responding to the source code itself. Again, making source code available for reading is linked to providing an address for responses arising from that reading. The Cocoon project keeps going only so long as it manages to recruit contributors who are prepared to read and amend the source code and other documents.

Our research at Lancaster has emphasised moving beyond idealised versions of the open source project - as exemplified in the "Hacker Ethic" [22] or "Rebel Code" [29] - towards understanding open source development as a sociological phenomenon. Our analysis suggests we transcend the simplistic motivational or incredible economic approaches that characterise much of the debate and move toward a praxiological understanding of open source - an understanding of the everyday practicalities of software projects. Standing outside of, such debates about motivation, allows us to concentrate on understanding exactly how and in what ways an open source project is accomplished as practical work. Such an approach focuses on some of the practical ways in which a project is developed and sustained and 'codes of practice' are displayed, achieved and maintained as features of everyday work. Of course much remains to be done. In particular we are exploring the notion of 'epistemic communities' [13, 14, 20, 21] or 'communities of practice' when applied to open source development, primarily because it meshes with much that we have already described. Edwards [13, 14] argues that the notion of epistemic communities provides an understanding of how open source software develops under difficult circumstances and that the four characteristics of an epistemic community; shared normative and causal beliefs, notions of validity and common policy enterprise; provide some insight into the workings of open source communities. Our interest would be in exactly how and in what ways these characteristics are manifested, made accountable and inculcated into the community.

3.7. Summary

1. The work described in this section has used a variety of data from the Cocoon open source project to explicate the ways in which open source projects are accomplished and the ways in which the various participants observably, reportably, accountably orient their actions, their contributions to the project as a whole.
2. We contrast open source development with more traditional software approaches through ethnographic and ethnomethodological studies of software production.
3. Our research provides an interesting and important contrast with idealised versions of the open source project, encouraging a move towards a praxiological understanding of software development.
4. When seen as a practical project, open source development can be seen to be little different to traditional software production - the project is dominated largely by practical (often commercial) considerations rather than philosophical idealism.
5. While the open source community can clearly be seen to have an 'ethic' or code of practice this is less a *determinant* of behaviour within the community than a *resource* that is drawn on in discussion within and without the community. In considering Open Source as an epistemic community less emphasis needs to be placed on such exhoratory norms and more on the constitution of the 'epistemics' of open source.
6. When viewed in terms of 'dependability', open source can be seen to have some advantages in terms of multiple scrutiny of code (Linus' Law). Similarly (at least in the case of Cocoon) there are clear lines of responsibility without any obvious 'blame culture'. Communication, both through email and the various documentation attached to code releases appears to be regularly maintained.

4. Open Source development and dependable systems engineering

During the work on this project activity, we formulated some conjectures based on observations, which might serve as a basis for future research. These conjectures are laid out in this section. This section also contains a model to investigate informal claims that can be made regarding products developed using open source methodologies (see section 4.4).

4.1. Software architecture

Software Architecture can be defined as the structure(s) of a system, which comprise software components, the externally visible properties of those components and the relationships among them [7]. It typically acts as a bridge between software requirements and lower level design leading to an implementation. The architectural design of a software system can represent the most vital artefact for a software project, as it directly impacts upon the important management and technical processes of production and integration [33]. Hence, a sound software architecture is desirable in order to build a solid software system.

Some of the reasons why software architectures are believed to be important are that they: facilitate the communication among stakeholders, represent the manifestation of the earliest design decisions, and constitute a relatively small and understandable model of how a system is structured [7]. Garlan further elaborates six aspects of software development within which software architecture can play an important role: facilitating understanding by using high-level abstractions, supporting reuse at multiple levels of granularity, providing a partial blueprint for development by indicating the major components and dependencies among them, exposing the dimensions among which a system is expected to evolve, providing analysis opportunities at early stages of development, and for basic management support [19].

In order to fulfil their expected roles, software architectures should be modularised. This modularisation plays a triple role:

- It facilitates understanding by using high-level abstractions and reducing the complexity of the task at hand,
- It highlights areas where work can occur in a concurrent and distributed fashion, and
- It can also be used to determine the organizational structure that should be in place for developing the system being considered.

Based on its intrinsic characteristics, software architecture design becomes essential while developing a large complex software system. It should be the responsibility of a main architect (group) to keep the vision of the overall system [10]. Additionally, in order to support system evolution, while avoiding architecture erosion and drift [31], the software architecture must also be evolved accordingly, at times requiring some major restructuring.

Architecture Description Languages (ADLs) were created in order to facilitate communications among stakeholders and support various forms of analysis. ADLs facilitate communication by providing a common vocabulary, hopefully without ambiguities, that can be shared by the people involved. The level of formalism and kind of analysis supported varies considerably from one ADL to the next, depending on the intent of their original designers.

One of the goals we had while studying the open source approach was to determine the impact that applying this approach would have on a specific system's software architecture. The intention being to establish whether more or less architectural decay is seen in open source software projects. A possible assumption would be that having many people to review it would

help keep a cleaner architecture. On the other hand, having too many people involved could trigger faster decay. We were unable to conduct a detailed study on software architectures of various open source software systems, yet we were able to derive several observations on the topic, as well as determine several research issues on software architectures that can be further studied by observing open source software systems. The latter has resulted on a paper that can be found on appendix D and was published in the 1st Workshop on Open Source Software Engineering, "Making Sense of the Bazaar", at ICSE 2001 [5].

A small minority of open source software projects has explicit system documentation, including explicit architectural models. We are unaware of any open source software project that uses an ADL to describe its architecture. Software architectures of open source software systems tend to be highly modularised, thus allowing for concurrent and highly decentralised software development. It is also the case that most open source software projects are in well-understood domains. Consequently, they tend to have fairly well understood standard architectures.

The original interest and vision in open source software projects usually emanates from the initiating projects' owners, such individuals often assume complete authority. Even in "shared-leadership" situations, such as the Apache HTTP Server, investigations have established that the core-developers still exercise major influence over the design and direction of open source software development [28]. Consequently, in contrast to traditional software approaches, open source software project managers seem to possess greater power to determine the architectural direction of the software product. In this respect, even in the (supposedly) decentralised open source software process, the traditional architect role still appears to be a prerequisite for preserving the conceptual integrity of software [10]. However there are views expressed that open source software project leaders may abuse this power to protect their own position by concealing the software architecture [8]. In doing so, they risk removing the blueprint that is vital for detailed understanding. Nevertheless, even in the absence of an explicit architectural blueprint, it may still be possible that the open source software development process can overcome the traditional software development barrier by narrowing the conceptual gap between requirements and implementation. The reasons being:

- Many of the users of open source software are also contributing developers [26],
- Creating programs for oneself has long been considered less demanding than developing software for others [37],
- The rapid releases and early feedback allow a greater level of incremental development in the open source software process [30, 32].

Finally, initial open source software releases may be lacking in code refinement and contain many residual faults [32]. Nevertheless, it has been recognised that for open source software projects to be successfully initiated, evolved, and maintained, the architecture must be modularised to promote code comprehension and concurrent collaboration [9].

It is uncertain as to whether open source software projects undergo major restructuring of the architecture or not. While many of the open source software project domains are stable (e.g. program compilers and operating systems) other researchers have claimed that open source software projects may undergo major restructuring of the architecture throughout its life cycle [4]. In this regard, the necessity for restructuring may be due to the particular application domain. Irrespective, however, this continues to be a particularly interesting area for further investigation and insight and should be carefully considered in any future follow-on project.

In the rare case of architectural restructuring being undertaken, it is usually a result of some technological evolution or some *far better* architectural option being detected. Architectural restructuring and its consequent code changes are only implemented after being thoroughly discussed by the developer's community involved and some form of agreement being reached, be it by consensus, voting, or as a decision of some form of core group after consulting with various developer groups within their community.

It should be noted that specifications for open source software systems tend to be *achieved* by discussions among the developer teams rather than having them formally defined. This can also be used to explain why the lack of a formal architectural description does not seem to hinder the development effort, the people involved just *understand* the concept behind the system.

4.2. Application domains

There are claims that all software should be built in an open source fashion but in our opinion there are domains in which this may prove to be very difficult. It is questionable whether open source communities will develop software outside their own field of interest and expertise. This is because motivation plays a major role in making an open source software project successful. Additionally, for an open source project to be successful, the developers need to be expert users of the system as well. The latter observation also probably explains the lack of explicit design/specification in open source: the developers more or less know the structure of the system already, since they are using it themselves.

There are also doubts raised whether software developed in an open source manner can meet the requirements to attain certain assurance levels. For example, it is not commonplace in open source software projects to find traceability information or the level of documentation required by certain regulatory agencies. Open source does not preclude this information to be included, but one does not frequently find people interested in voluntarily contributing towards these assets.

Unstable domains may also prove to be more challenging for open source software development. The term "unstable domains" here relates to specific software areas that are still being investigated or their status is still on progress. This kind of software is still evolving a lot, and it is unclear whether the "loose" approach (no explicit design, no time-plan) employed in the open source method will make the software more fragile (susceptible to breaking apart), or foster innovation. Additionally, the motivation for people voluntarily involved here is impacted. Peer recognition is no longer a major driver but the challenge of learning something new plays a bigger role.

We had several discussions regarding open source and legal responsibility. There is no apparent legal responsibility assigned to individuals and/or organisations with respect to open source software systems. This seems to imply that the open source approach is unsuitable for environments where legal responsibility is an issue. Yet it should also be noted that most non-open source software systems also carry no legal responsibility. Our lack of expertise in legal issues prevents us from trying to detect if and where there are real differences here. Although there is no legal responsibility assigned to individuals, the open source community seems to take on "moral" responsibility and does react promptly when major problems are discovered. Quite often reacting faster than vendors of proprietary software (see section 5 in appendix B).

4.3. Software processes

We have observed that there exist open source projects that have started from scratch, yet some big successes in open source come from projects that utilize extant, older systems (e.g. Linux from Minix, Apache from NCSA's HTTPd 1.3 web server). It would be interesting to study what conditions are required for an open source (and non-open source) software project to be successful if it was to start from scratch.

The fact that open source developers are users as well might result in a simpler requirements engineering process in open source projects, since they know first hand what is actually needed or desired. If this is really the case, the complexity of project development could be reduced using the open source approach.

The open source approach can often be regarded as a "massive human scrutiny" process for finding and fixing bugs. Finding bugs here means locating and reporting the bugs to the mailing list or project owner(s), whereas fixing bugs involves coding and testing, as well as providing patches and updating the repository. One observation made is that for efficiency, it is better to have more people involved in finding than fixing bugs. This will reduce the communication overhead among the developers. Furthermore, having many different contributors towards the code encourages diversity. It is expected that by having submissions coming from diverse sources one can have an improvement in fault tolerance levels and may also introduce novel approaches.

There have been claims that by having many reviewers the quality of a software system is improved [32]. An interesting issue to pursue is whether having more reviews could replace the role of formal analysis in software development. And if so, where does the threshold lie?

The "participative" nature of open source, where everyone is encouraged to contribute towards the project, brings the consequence of potentially having numerous submissions with varying degrees of relevance and quality. Therefore, there is a need to have a process for determining what to accept or reject. The details of this process are different from one open source project to another, but the main concern remains the same: how to pick and incorporate good submissions from the available ones. We have not explored the process of handling submissions extensively, this might be interesting to investigate further.

4.4. Quantitative issues on Open Source dependability

Most dependability claims about open source software seem to be highly anecdotal, and the same evidence can be used to support opposite, though equally plausible claims, for instance the availability of code may improve security by allowing quicker fixes or make it worse through increased exposure to malicious attacks. Our work has been aimed at clarifying the kinds of arguments that can be made about open source's dependability advantages or disadvantages and what might cause them.

Even for statements about specific open source projects, e.g. "Open source product X is more dependable than its commercial counterpart Y", it is necessary to clarify which of the many facets of dependability the statement is about. When generalising to open source processes and their effects on dependability, we want to focus on statements of the kinds: "we should expect open source projects to deliver better dependability from viewpoint X (e.g. a certain measure of security, or availability) because of factor Y in which open source processes tend to differ from non-open source processes" (i.e., a prediction about dependability) or "we know that open

source projects tend to deliver better X because of factor Y" (i.e., an explanation of observed dependability).

We give below examples of such conjectured causal factors. Furthermore, we have developed a first exploratory model to help clarify the plausible effects of some of these factors.

4.4.1. Identification of different dependability measures

A general point to be made is the importance of giving a context to each measure. A bald statement of the kind "X is more dependable than Y" makes little sense. A measure is always "with respect to" a particular type of service or failure, and "given a" particular usage profile or condition. A product can be better than another according to one measure but worse according to another. For instance, one could be interested in confidentiality, measured in terms of how often attackers gain access to confidential information, or in availability, measured in terms of how long a service is not usable. If one knows that product X exhibits better availability than product Y, one cannot draw any conclusion about which one protects information better from unauthorised access.

Any statement about software dependability needs to specify the aspects (and thus measures) of dependability it refers to. Apart from distinguishing e.g. between availability and confidentiality (as in the examples above), it is important to clarify the time span and the population (of installations) considered. For instance, dependability statements might concern:

- a release of a product at a specific point in time and for a specific user,
- the reliability evolution during the lifetime of a product, due to:
    - the process of fault fixing and of new releases; and
    - changes of usage in time, for instance in the case of a web server, dependability could depend on the rate and type distribution of accesses, or for an operating system, on the types and number of programs running;
- time evolution is important as, e.g., there are application domains where teething problems can be tolerated, especially if the software is expected to improve rapidly, whereas there are situations in which it is important to have achieved a certain reliability with the first release;
- averages over time of the measure of interest;
- averages over a population of users. This will not reflect the fact that different users experience different reliability, and in general give only a general estimate, not being able to give a characterisation of all terms giving the average and thus could be misleading for a particular individual user. For instance, one might be interested in how many users experience a reliability worse than a particular threshold, or what is the worst reliability that one can experience.

An attempt in this direction has been made by D. Wheeler [38]. He examines various claims about Open Source Software, discriminating between the various qualities of interest and stressing the importance of the fact that claims about dependability should always be linked to the usage profile, and that with different profiles opposite claims can be verified.

4.4.2. Conjectures on causal mechanisms affecting dependability

There are many different factors and causal mechanisms which can plausibly explain superior reliability of open source products. These are useful to study if we have enough evidence of often-superior dependability to make us wish to learn which factors account for it, or, if evidence of measured dependability is too difficult to collect, to seek indirect ways of predicting the effects of specific "openness" factors in software development.

Amongst these, we list some which we think worth investigating further (a few of these are explored elsewhere in this report):

- the source code is exposed to many different inspections by different users. The diversity between inspectors increases the chances of finding faults;
- users' bug reports come from usage under different profiles. This is equivalent to testing under a set of different realistic profiles, exploiting the diversity between them. Under certain conditions, this is more efficient in terms of reliability growth (i.e. in achieving reliability) than testing with a more systematic approach;
- there is a "democratic" approach to all bug reports, so that even users experiencing failures due to bugs that do not affect other users will be heard; hence users with very unusual profiles will see a better reliability improvement with open source software;
- users can individually correct the code of their own installations and thus improve their own experienced dependability;
- the environment promotes the improvement of the code by newsgroup or mailing lists discussion. Hence, people contributing to a project share a common "language";
- changes to the code are discussed at length. The discussions help to think or to notice details or possible improvements that were previously overlooked;
- maintainers may choose among different solutions to the same problem thanks to what is commonly referred to as "multiple submissions";
- the developers are also users of the software, hence they understand better the issues involved. A common problem in design is that the designers are usually not the end users of a product. Having the users developing the software seems to overcome this difficulty;
- contributors are usually very good programmers, so that behind the quality of the product lies the high coding skills of the individuals involved in the particular project. There is no "new" factor contributing to the high dependability of open source software products;
- open source software projects tend to be built upon a sound underlying architecture; and/or have better modularised code, allowing more people to work independently and concurrently;
- there is more total effort involved, so better quality is to be expected (an important conjecture, which may mean that there is no project management technique to learn from open source software projects except that of motivating many developers to contribute);
- more effort is put into the inspection of the code, or some other particular stages of the development where it is more productive.

For some of these conjectures, a common doubt applies: for each famously successful open source software project, there are many that fail or whose results are unknown. Perhaps keeping a project running with an open source "team" is more difficult than in a conventional "closed" team, so that projects that do not enjoy skilled participants or sound architectures (for instance) are less likely to survive in open source conditions. Additionally, perhaps the success of an open source project depends on having a stable group of core developers, so that they thoroughly understand the project (its architecture, history, etc.), and hence are able to facilitate evolution. Consequently, a greater "infant mortality rate" could be the main factor assuring that the surviving open source software projects are "better" than the average comparable non-open source software project.

Some of these conjectures lend themselves well to investigation via probabilistic modelling, and we have started some exploratory effort in this direction.

4.4.3. A speculative model

Amongst the statements cited above, we chose initially to concentrate our attention on the process of fault finding via execution (testing or normal use) and subsequent corrections. We built a model of the conjectured effects of three "openness" factors on the reliability growth of a product. The three factors in question are more bug reporting and associated fault fixing, and the diversity in users' usage profiles. To each factor corresponds a set of parameters of the model. A first goal of this style of modelling is to clarify whether it is at least plausible that certain factors and mechanisms actually improve the dependability aspects claimed, under which conditions, and which empirical observations could check whether these are actually driving factors in reality.

It is interesting to note that one of the conclusions drawn, which has also been observed in other work exploring the use of diversity in software development, is that diversity does not necessarily always imply (causes) an improvement in (some or all) of the dependability measures for a product. Diversity can bring both advantages and some counter-intuitive disadvantages. This model is described in a working paper included in appendix E.

5. Recommendation

This project activity was undertaken with the expectation of deciding whether further effort should be spent towards looking into open source software for producing a positive impact on software systems' dependability. During our investigations we have realized that there is much variation between open source software projects just as there is between more "conventional" software projects.

Our conclusion is that we shouldn't recommend that open source be put out of the picture completely, nor that we have further project activities to look into open source in general. We believe that there are various aspects of openness that may impact the dependability of software systems, aspects that may manifest themselves both in open source software and in non-open source software projects.

Our recommendation is that DIRC should have a further project activity to study the impact of various factors on the design process of software systems and their resulting dependability. This recommendation will be further refined in an upcoming project activity proposal on "Effective collaboration in design (of dependable software)".

6. Acknowledgements

The work reported here was undertaken mainly by Budi Arief, Diana Bosio, Cristina Gacek, Cliff Jones, Tony Lawrie, Mark Rouncefield, and Lorenzo Strigini. It was also the result of several discussions with many of our DIRC colleagues, such as Denis Besnard, John Dobson, David Greathead, Bev Littlewood, Adrian MacKenzie, Carles Sala-Oliveras, Brian Randell, Perdita Stevens, Robert Stroud, as well as one of DIRC's senior visiting fellow Michael Jackson. We are very grateful to Julian Coleman, Mike Ellison, Stuart Weather and several other people outside DIRC that graciously accepted to discuss with us matters related to open source, thus helping us to deepen our understanding on the matter.

7. References

[1] "Cocoon Website", online at *http://xml.apache.org/cocoon/* .

[2] “The Open Source Initiative: Open Source Definition”, online at *http://www.opensource.org/docs/definition.html* .

[3] Anderson, R. and W. Sharrock, “Can Organisations Afford Knowledge?”, *Computer Supported Cooperative Work*, Vol. 1, No. 3, pp. 143-162 (1993).

[4] Aoki, A., K. Hayashi, K. Kishida, K. Nakakoji, Y. Nishinaka, B. Reeves, A.Takashima, and Y. Yamamoto, “A Case Study of the Evolution of Jun: an Object-Oriented Open-Source 3D Multimedia Library”, *Proc. 23rd ICSE Conference*, Toronto, Canada, pp. 524-532 (12-19th May 2001).

[5] Arief, B., C. Gacek, and T. Lawrie, “Software Architectures and Open Source Software - Where can Research Leverage the Most?”, *Proc. 1st Workshop on Open Source Software Engineering*, Toronto, Canada, pp. 3-5 (15 May 2001).

[6] Armour, D., “Socio-Logic and the Big Idea,” in *School of Management*: Unpublished MSc Dissertation, Lancaster University (1998).

[7] Bass, L., P. Clements, and R. Kazman, *Software Architecture in Practice*, Addison-Wesley (1998).

[8] Bezroukov, N., "A Second Look at the Cathedral and the Bazaar", in *First Monday*, online at *http://www.firstmonday.dk/issues/issue4_12/bezroukov/* (9 December 1999).

[9] Bollinger, T., R. Nelson, K. M. Self, and S. J. Turnbull, “Open-Source Methods: Peering Through the Clutter”, *IEEE Software*, No. July/August, pp. 8-11 (1999).

[10] Brooks, F. P., *The Mythical Man Month: Essays on Software Engineering*, Addison-Wesley (1995).

[11] Button, G. and W. Sharrock, “Project Work: The Organisation of Collaborative Design and Development in Software Engineering”, *Computer Supported Cooperative Work: The Journal of Collaborative Computing*, Vol. 5, No. 369-386 (1996).

[12] DIRC, “Interdisciplinary Research Collaboration on Dependability of Computer-Based Systems -- First Workshop”, Storrs Hall, Windermere (24-26 July 2000).

[13] Edwards, K., “Towards a Theory for Understanding the Open Source Software Phenomenon”, online at *http://www.its.dtu.dk/ansat/ke/towards.pdf* (2000).

[14] Edwards, K., "When Beggars Become Choosers", in *First Monday, volume 5, number 10*, online at *http://firstmonday.org/issues/issue5_10/edwards/index.html* (October 2000).

[15] Feller, J. and B. Fitzgerald, *Open Source Software Development* (to be published).

[16] French, S. and N. Thrift, “Machine in the Ghost: Software Writing Cities”, paper given at Conference on Information and the Urban Future, New York University (2000).

[17] Gacek, C., T. Lawrie, and B. Arief, “The Many Meanings Of Open Source”, Department of Computing Science, University of Newcastle upon Tyne, Technical Report CS-TR-737 (August 2001).

[18] Garfinkel, H., *Ethnomethodological Studies of Work*, Routledge (1986).

[19] Garlan, D., “Software Architecture: a Roadmap,” in *The Future of Software Engineering*, A. Finkelstein, Ed.: ACM Press, pp. 93-101 (2000).

[20] Hass, P. M., “Do regimes matter? Epistemic communities and the Mediterranean Pollution Control”, *International Organisation*, Vol. 43, No. 3, pp. 376-403 (1989).

[21] Hass, P. M., “Introduction: Epistemic Communities and International Policy Coordination”, *International Organisation*, Vol. 46, No. 1 (1992).

[22] Himanen, P., *The Hacker Ethic and the Spirit of the Information Age*, Random House (2001).

[23] Hine, C., *Virtual Ethnography*. London, Sage (2000).

[24] Hughes, J. A., V. King, T. Rodden, and H. Andersen, “Moving out from the control room: Ethnography in system design”, *Proc. CSCW ’94*, Chapel Hill, North Carolina (1994).

[25] Hunt, F., "Developer Motivations in Open Source Development" (in preparation).

[26] Johnson, K., "Towards a Descriptive Process for Open-Source Software Development", Submission for the 2nd Workshop on Software Engineering over the Internet, online at *http://www.cpsc.ucalgary.ca/~johnsonk/SENG/SENG691/towards.htm* .

[27] Latour, B., "Trains of thought: Piaget, Formalism and the Fifth Dimension", *Common Knowledge*, Vol. 6, pp. 170-191 (1997).

[28] Mockus, A., R. T. Fielding, and J. Herbsleb, "A Case Study of Open Source Software Development: The Apache Server", *ACM Proc. 22nd International Conference on Software Engineering*, Limerick, Ireland, pp. 263-272 (2000).

[29] Moody, G., *Rebel Code: Linux and the Open Source Revolution*, The Penguin Press (2001).

[30] Pavlicek, R. C., *Embracing Insanity: Open Source Software Development*, SAMS Publishing (2000).

[31] Perry, D. E. and A. L. Wolf, "Foundations for the Study of Software Architecture", *ACM Software Engineering Notes*, Vol. 17, No. 4, pp. 40-52 (October 1992).

[32] Raymond, E. S., *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*, O'Reilly & Associates (1999).

[33] Royce, W., *Software Project Management: A Unified Framework*, Addison Wesley (1998).

[34] Suchman, L., "Working Relations of Technology Production and Use", *Computer Supported Cooperative Work*, Vol. 2, pp. 21-39 (1994).

[35] Thomsen, S., J. Straubhaar, and D. Bolyard, "Ethnomethodology and the Study of Online Communities: Exploring the Cyber Streets.", *Proc. IRISS'98*, online at *http://www.sosig.ac.uk/iriss/papers/paper32.htm* (1998).

[36] Weider, D., *Language and Social Reality*. The Hague, Mouton (1974).

[37] Weinberg, G. M., *The Psychology of Computer Programming*, Dorset House (1971).

[38] Wheeler, D. A., "Why Open Source Software / Free Software (OSS/FS)? Look at the Numbers!", online at *http://www.dwheeler.com/oss_fs_why.html* .

Appendix A: The many meanings of Open Source

Cristina Gacek, Tony Lawrie, and Budi Arief
Centre for Software Reliability
Department of Computing Science
University of Newcastle
Newcastle upon Tyne NE1 7RU
United Kingdom
{Cristina.Gacek, A.T.Lawrie, L.B.Arief}@ncl.ac.uk

Abstract

The term *Open Source* is widely applied to describe some software development methodologies. This paper does not provide a judgment on the open source approach, but exposes the fact that simply stating that a project is open source does not provide a precise description of the approach used to support the project. By taking a multi-disciplinary point of view, we propose a collection of characteristics that are common, as well as some that vary among open source projects. The set of open source characteristics we found can be used as a tick-list both for analysing and for setting up open source projects. Our tick-list also provides a starting point for understanding the many meanings of the term open source.

1 Introduction

The term *Open Source* has been widely used to describe a software development process that relies on the contribution of its geographically dispersed developers by the means of the Internet. Amongst other criteria, one basic requirement of open source projects is the availability of its source code [1], without which the development or evolution of the software is very difficult if not impossible. But apart from these characteristics, there seems to be some confusion on what actually makes a project an open source project.

The aim of this paper is therefore to provide a clearer description on what is meant by "open source". To achieve this aim, we investigated several well-known open source projects such as Linux [2], Apache [3] and Mozilla [4]. We also did literature studies on published materials about open source, notably *The Cathedral and the Bazaar* [5], *Rebel Code* [6], *Open Sources* [7] as well as work by other people interested on open source (for example, [8-12]). We have also used several on-line resources dedicated to various open source projects [13, 14] and interviewed both individuals working on open source projects at their free time and individuals involved with open source as part of their job in large corporations. From there, we tried to dissect open source further by determining the characteristics that open source projects should or usually have. We determined a set of characteristics that are almost always present and others that vary among open source projects, and this serves as the core of this work.

The rest of this paper is structured as follows: Section 2 presents a brief history of open source, which is important for understanding its motives and directions; Section 3 describes some open source characteristics that can be used in determining whether a project is or not open source; Section 4 provides some initial conclusions of our work; and Section 5 outlines areas that can be researched further.

2 A Brief History of Open Source

2.1 How it started

The idea of building software within a cooperating community, where the source code was made available so that everyone could modify and redistribute it began with the GNU project at MIT in the early 1980s. The intention was to provide *freedom* relating to software systems. In 1985 the *Free Software Foundation (FSF)* was pioneered by Richard Stallman to generate some income for the free software movement, not restricting itself to GNU.

Free software, as defined by the FSF, is a program that grants various freedoms to its users. A free software program provides its users with [15]:

- Freedom to run the program for any purpose
- Freedom to study and adapt the code for personal use
- Freedom to redistribute copies of the program, either gratis or for a fee
- Freedom to distribute improved or modified versions of the program to the public

The discourse used by the FSF tends to be confrontational and against proprietary (closed) software, since they view anyone producing this kind of software as big obstacles to the four basic freedoms mentioned above. This is reflected in the restrictive viral nature of some of their licenses (see section 3.3).

2.2 Free Software and Open Source Movements

In the early 1998, the term *Open Source* was coined as a response to the announcement made by Netscape on its plan to give away the source code of its web browser. The new term came out of a strategy meeting in which people present realised that:

> *"…it was time to dump the confrontational attitude that has been associated with 'free software' in the past and sell the idea strictly on the same pragmatic, business-case grounds that motivated Netscape." [16]*

Immediately afterwards, the *Open Source Initiative (OSI)* was set up to manage and promote the *Open Source Definition (OSD)*. The OSD was composed as a guideline to determine whether a particular software distribution can be called open source or not. OSD asserts nine criteria that open source software must follow; the main three are:

- The ability to distribute the software freely
- The availability of the source code, and
- The right to create derived works through modification.

The rest of the criteria deals with the licensing issues and spell out the "no discrimination" stance that must be followed [1]. They are:

- The integrity of the author's source code must be preserved, making the source of changes clear to the community
- No discrimination against persons or groups both for providing contributions and for using the software
- No restriction on the purpose of usage of the software, providing no discrimination against fields of endeavour

- The rights attached to the software apply to all recipients of its (re)distribution
- The license must not be specific to a product, but apply to all sub-parts within the licensed product
- The license must not contaminate other software, permitting the distribution of other non-open source software along with open source one

The Open Source and Free Software movements can be compared to two political parties within a community. While two political parties agree on the basic principles but disagree on practical issues, the Open Source and Free Software do exactly the opposite. They disagree on the basic principles (commercialism, licensing, etc.), but agree on (most of) practical recommendations (availability of source code, ability to modify the code, etc.). They even work together on many specific projects to achieve the same goal: to provide software that is free (in terms of liberty) for all [17].

2.3 Commercialisation of Open Source

Open source is often seen as a marketing ploy to make Free Software more attractive to business users since it allows greater liberties with its licenses (see section 3.3). This means that the open source licenses do not prevent people or companies from making profit from the software, as long as the source code remains available and can be modified freely.

The most prominent way of commercialising open source is by providing service and distribution packages for software developed in an open source fashion. This is due to the fact that open source software is usually more difficult to install since it was originally aimed for the hacker community. Another way of making money out of open source is by using the relevant open source as a platform, upon which commercial (often proprietary) application software can be built.

More and more computing corporations turn their attention to open source as a business opportunity. What they are looking for in this new development method is *innovation*, and sharing source code is perceived to be a good way for facilitating creativity. Commercial organizations are also attracted to contributing to open source projects as they see a strategic opportunity to undermine (more powerful/dominating) competitors. On the down side, they are afraid that maintaining control of an active open source project can be difficult. They are particularly concerned with the risk of code forking – the evolution of two (or more) separate strands of work from the original code base, which threatens compatibility. This fear prevents some individuals and many companies from active participation in open source developments [7].

Although this code forking risk is always present, it is usually overcome by the novel attitude that the open source community has. Instead of basing their reputation on “what they have”, they measure it against “what they give”. This “gift-culture” encourages people to contribute more and binds people together in the same strand of work. More information on the “gift-culture” is available from Eric Raymond’s paper, *Homesteading the Noosphere* [18].

2.4 The Open Source Approach compared with Others

To provide a clearer picture on where open source (free) software stands in relation to other software, we provide some comparisons (mostly in licensing and distribution terms) among several categories of software. For simplicity, we could say that the two main categories are the “free” software (meaning open source as well) and the “proprietary” software.

There are two kinds of software within the "free" category: *non-copylefted free software* and *copylefted software*. Non-copylefted free software comes from the author with permission to modify and redistribute, and in a legal term it means "not copyrighted". On top of that, it is allowed to add more restrictions to the modified version, which means that some copies (modified versions) may not be free at all. Anyone can compile the program and redistribute the binary as proprietary software. *Public domain software* is a special case of non-copylefted free software. On the other hand, with copylefted software, it is not allowed to have additional restrictions to be added when someone redistributes or modifies the software. As a consequence, every copy of copylefted software, even after modification, must be a free software. The most prominent distribution terms for copylefted software are covered in the *GNU GPL (General Public License).*

Proprietary software is *closed* software in that the source code is not available to the public. It has very restrictive terms on its condition of use, and its redistribution or modification is prohibited. There are two special cases within this group of software: *shareware* and *freeware*. Both allow people to download, use and redistribute the software for free, but modification is (almost) impossible because they are usually released in executable (binary) format only. The difference is on the limit of usage, if someone wants to keep using a shareware, he/she must pay a license fee. One important note is that freeware must not be confused with free software, especially because modification of a freeware is not possible (since the source code is not available).

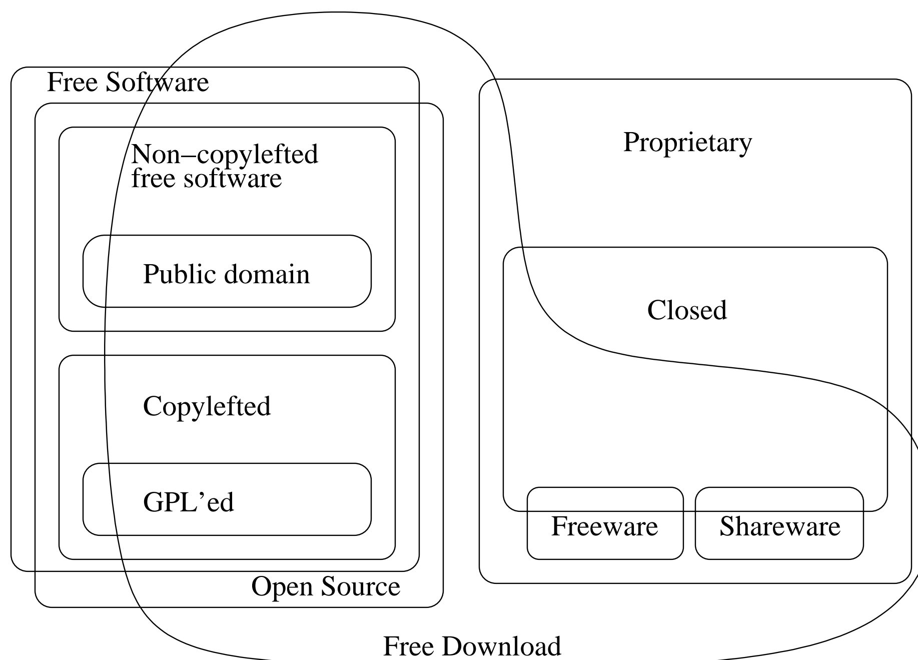


Figure 1: Categories of software

The classification of software in the manner above can be seen diagrammatically as Figure 1, which was adapted from the software categories based on the Free Software Foundation view [19]. Table 1 below summarises the main comparisons between the characteristics of those software categories.

There are subtle differences between open source and free software, in particular around licensing issues. For example, open source software may use proprietary

library (e.g. the KDE project [20] was using a proprietary library called Qt until September 2000), which is unacceptable in free software. Further investigation surrounding these differences could provide better understanding, as highlighted in section 5.

Table 1: Comparisons of different kinds of software

	Open Source (Free) Software		**Proprietary Software**		
	Non-copylefted	*Copylefted*	*Closed*	*Shareware*	*Freeware*
Availability of source code	Y	Y	N	N	N
Permission to					
• redistribute	Y	Y	N	Y	Y
• modify	Y	Y	-	-	-
• add restriction	Y	N	N	N	N
Modified version always free	N	Y	-	-	-
Free Download	Y	Y	N	Y	Y
Time Limit in usage	N	N	N	Y	N
Possibility of making money	Y	Y	Y	Y	N

3 Characteristics of Open Source

By exposing the characteristics that open source projects usually have, we hope to be able to develop a clearer picture on what it really means for a particular project or software development to be an open source project[1] or not. The idea is to have a "tick-list" of open source characteristics, against which the characteristics of the project in question can be compared. Additionally, these characteristics highlight the fact that just stating that a project is open source does not necessarily provide a precise definition.

3.1 Disciplines to consider

In the spirit of DIRC[2], the sponsor of this paper, it is important to highlight that software development is a very complex process that draws upon knowledge/expertise from many scientific disciplines. Therefore, to understand it better, it is necessary to emphasise its interdisciplinary nature. It appears that open source software development is no exception, and in order to determine the relevant open source characteristics, there are several disciplines that we would like to consider:

- Computing Science
  Covering the technical aspects that need to be considered to engage in an open source project.
- Management Issues
  Dealing with managerial issues and how they relate to open source projects.
- Social Sciences
  Addressing areas related to the communities involved in open source projects and their behaviour.

[1] The term 'project' is used loosely in this paper, as it is doubtful whether OSS projects fulfil the more generic management definition of a unique/novel activity with explicit/finite timescales. Should the use of this term create conflicts of definition, for readers, they can interpret the term 'project' as 'undertakings' or 'initiatives'.

[2] DIRC is a UK EPSRC project based on a Dependable Interdisciplinary Research Collaboration (DIRC) on computer-based systems (see http://www.dirc.org.uk/).

- Psychology
  Accounting for the characteristics of the individuals involved in open source projects.
- Organisational Aspects
  Dealing with aspects such as organisational structures.
- Economics
  Looking into economic models that underlie open source projects and/or corporations with respect to their involvement in open source projects.
- Law
  Focusing on legal issues.

Clearly, the OSI definition for the term open source does address legal issues extensively, and encompasses some economic aspects. On the other hand, it hardly touches on computing science areas; it also completely ignores the areas of management, psychology, social sciences and organizational aspects. Furthermore, there is no guarantee that a given project, by simply adhering to the OSI definition of the term open source, benefits from the positive effects that are usually related to the term open source (e.g. being reviewed by many people). The open source software characteristics proposed by Wang and Wang [11] address some technical aspects, and in less depth, legal and managerial aspects.

In our attempt to understand open source, we determined a set of characteristics that occur under that umbrella term, while considering the various disciplines mentioned above. Some characteristics are common to all efforts we were able to investigate, whereas others vary between projects. The set of characteristics we deem relevant for discussing open source are described below, section 3.2 covering those that are common throughout open source projects and section 3.3 addressing those that vary between projects.

3.2 Common characteristics

Open source projects have many common characteristics. All items listed under the OSI definition of open source, OSD (see section 2.2), are the basic requirements for projects to qualify as open source. Moreover, *living* open source projects rely upon several other characteristics. We have identified six characteristics that are present in successful open source projects, these are addressed below.

Community

All active open source projects have a well-defined community with common interests that are either involved in continuously evolving its related products and/or in using its results. Anecdotally, the community, in its vast majority, is composed by men. Communications tend to be constructive, at times becoming confrontational.

Motivation

The biggest question surrounding the open source phenomena is *why do people do it?* What is the explanation behind having people providing contributions for free? The answer to these questions is not as straightforward as one might have thought. There are *different types of contributors*, individuals and corporations. Individuals usually contribute for personal satisfaction; some have really strong philosophical beliefs others do not care as much about such issues. Corporations usually get involved with

the aim to gain market share, undermine their competitors, or simply rely on products generated by open source without having to build a fully equivalent product from scratch.

Peer recognition also plays a role on motivating contributions. By having their contributions recognized as appropriate and of good quality by the community involved, both individuals and corporations have their status raised within the given project. Consequently, their opinions are considered more carefully with respect to project related decisions and their reputation may even improve outside the project boundaries.

Developers are always users

The set of people that contribute code to specific open source projects is always composed of those that are also users of the code produced. This means that open source developers are a subset of the open source user community, i.e. all open source developers are users, but not all users are developers (Figure 3).

This characteristic explains the fact that there are normally no precise specifications or requirements documents clarifying what is to be achieved in the project. It also highlights that it is quite unrealistic to expect the open source community to start developing arbitrary kinds of software. Software developers are usually not expert users of medical systems, nuclear plant control systems, or air traffic control systems.

The process of accepting submissions

An open source project evolves by receiving submissions from various sources to address various aspects of the project. The most common submissions are those of bug reports and source code, others include documentation and test cases. Furthermore, open source projects often post the areas in which they are interested in receiving submissions. As a consequence, multiple concurrent submissions may be received addressing the exact same area. Therefore, open source projects have in place processes for accepting various types of submissions, also making it clear on how to handle multiple concurrent submissions.

The process of accepting submissions is composed of two main parts: the *decision making process* and the *process of disseminating information on submissions*. How these two parts get implemented varies from one open source project to another (see section 3.3).

Development improvement cycles

Product improvement in the open source software development process can manifest in both breakthrough and continuous improvement modes. Breakthrough improvement involves dramatic and relatively impromptu changes [21]. Evidence of this form of product improvement in open source development was provided by Raymond [5] in the development of Fetchmail. He notes that:

> *"The real turning point in the project was when Harry Hochheiser sent me his scratch code for forwarding mail to the client machine's SMTP port…this SMTP-forwarding concept was the biggest single payoff I got…The Cruftiest parts of the driver vanished. Configuration got radically simpler…the only way to lose mail vanished...and performance improved." (p. 47-50)*

Continuous improvement involves an increased frequency of change but in smaller and more incrementally consolidating stages [21]. This philosophy of product development recognises that small improvements build up to larger improvement overtime, but with the added advantage of being far easier to implement. Incremental product improvement through bug finding and fixing is a development hallmark of the open source paradigm and is embodied in Eric Raymond's original characterisation "release early, release often" [5] The idea is to get quick feedback, which can then be incorporated back into the product.

More recently such anecdotal claims have been further reinforced by the research findings of Aoki et al. with the open source *Jun* project [22]. They tracked the evolution of the software over 360 versions and identified both incremental improvements within single version updates followed by significant functionality increases requiring major modification to the existing architecture. Both of these forms of product improvement are generically shown in Figure 2 below.

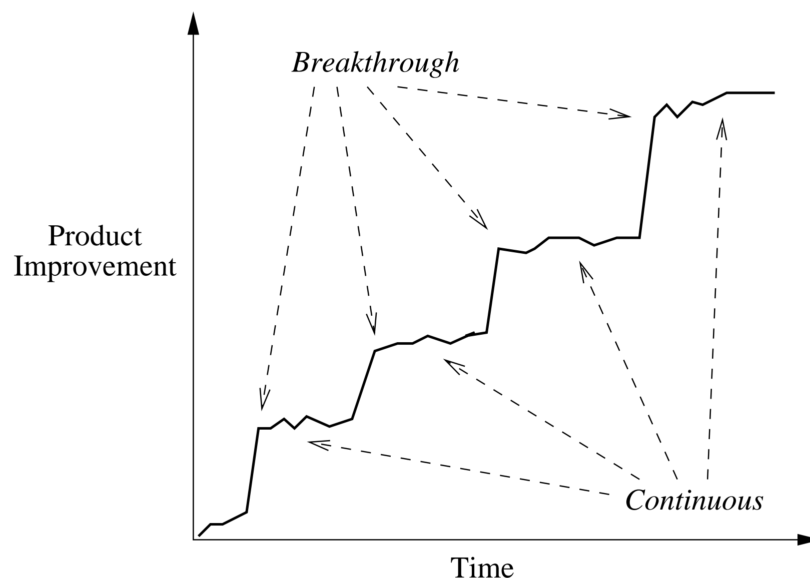


Figure 2: Open source product improvement over time.

Modularity

The benefits of modular design are well established in all engineering disciplines, as it supports increased understanding during design and concurrent allocation of work during implementation [23]. However, due to the globally distributed nature of open source development, well-defined interfaces and modularised source-code are a prerequisite for effective remote collaboration [24].

3.3 Variable characteristics

The areas in which open source projects vary are much more numerous than those that they have in common. Below is a discussion of some of those.

Choice of work area

As previously mentioned, open source projects often request contributions to the areas in which they are interested in receiving submissions. Some open source projects will

process both solicited and spontaneous contributions, whereas other open source projects may be prone to ignoring spontaneous contributions.

Balance of centralisation and decentralisation

The communities within various open source projects are organised differently. Some have a very strict hierarchy differentiating among various levels of developers (see Figure 3), whereas others have a much looser structure. The strict hierarchies bring with them a more centralised power structure, for example, the *core* developers have more power than ordinary (co-) developers in making executive decisions. In some open source projects (e.g. Apache), it is even possible to have more than two levels of developers. But not all open source projects have multi-level developer groups. Looser organisational structures have all their developers on the same level, which implies decentralisation of decisions, at times being based on full consensus for approving decisions.

Meritocratic culture

The basic model underlying open source projects is that knowledge shown by means of contributions increases the perception of merit, which in turn leads to power. Exactly how this transition takes place varies from project to project in terms of timing and the obstacles that must be overcome, and depends on the actual organisational structure of the project. For example, Figure 3 shows the possible transition from passive to active users when they start contributing to the project. If they could then show their ability (or they could gain respect from the community), they might be invited into the developer group, where they would have greater rights over the code (e.g. to incorporate their own modifications into the code base). In some projects, there is also a possibility of promotion from the co-developer to the core developer group. The transitions can also go the other way, e.g. a core developer might wish to resign and become a co-developer instead (or even leave the project completely) due to other commitments or personality clash.

Business model

Depending on the domain that an open source project addresses, different business models may motivate the involvement of commercial corporations, researchers, individual developers and end-users. The business models we have identified so far are: own use, packaging and selling, and platform/foundation for commercial or research software development.

Decision making process

The decision making process relies on four dimensions that vary from open source project to project. These are the *quality goals*, the *acceptance criteria* enacted, the *cognitive abilities* of the decision group, and the *social structure* within the project. Quality goals vary widely from one open source project to another; this can be observed even in the same application area (e.g. one focusing on performance and another on portability). The acceptance criteria used also vary among open source projects. It can be the best solution out of the first *n* submissions, some form of aggregation of multiple submissions (even by requesting that someone changes their solution to add some other aspect seen elsewhere), some memory of previous submissions by the same person, the first submission received, etc. Additionally, the

ability to recognise better solutions is highly dependent on the cognitive abilities of the decision group. This implies that the decision making process on accepting submissions varies among projects and potentially within projects as well, unless the same people are involved in all decisions.

The social structure inherent to an open source project may be a defined hierarchy where different groups of people get to evaluate different submissions (e.g. by focus area) and/or some people exercise greater power, or a monolithic group composed of all developers. The social structure impacts directly on the decision making process. If the group is monolithic then the acceptance of submissions may be achieved by consensus or majority voting. If there is some other form of social structure, the same consensus or majority voting may apply, at times with the votes of some of the members counting more than others.

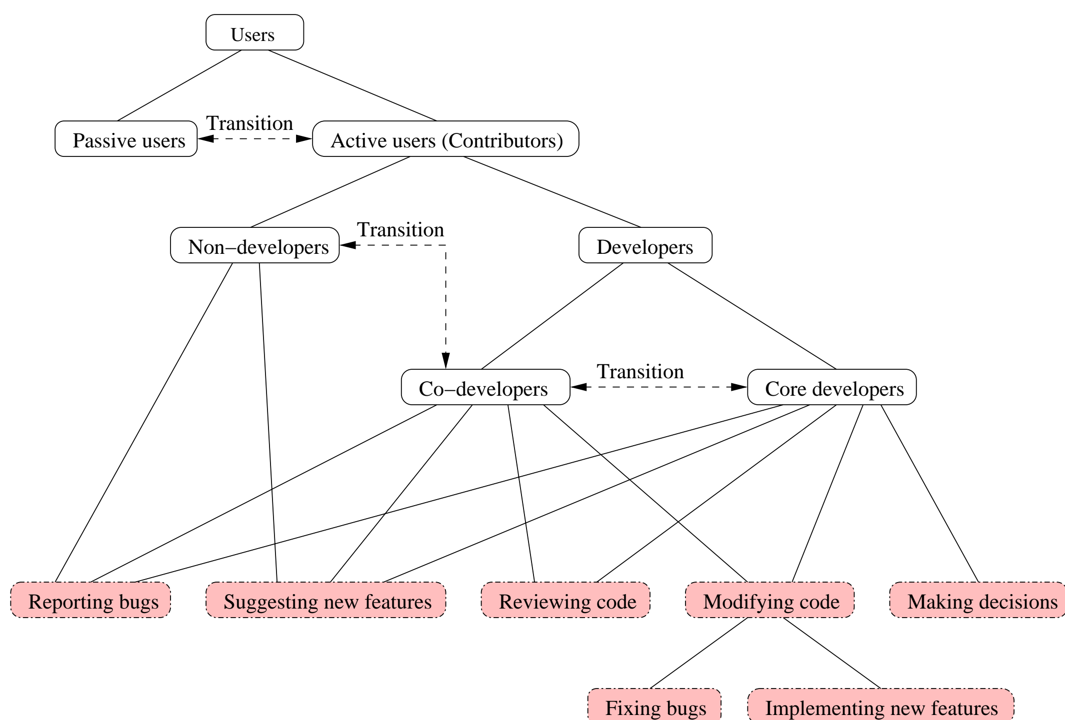


Figure 3: The classification of open source users and developers

Submission information dissemination process

The information on submissions and their acceptance may be passively disseminated by the means of newsgroups or comments in the code itself, it may be actively disseminated by using emails and mailing lists, or there may be some dedicated web space for statistical information.

Project starting points

Open source software projects may start from scratch or from existing closed source software systems, either commercial or research. From the various projects that we studied we could only find examples of projects that transitioned the full package from closed to open source at once. Nevertheless, one can envision some closed source software making a gradual transition to open source, one part (e.g. a subsystem) at a time.

Visibility of software architecture

The software architecture of a computing system depicts its structure(s) and comprises its software components, the externally visible properties of those components, and the relationships among them [25]. The architecture of an open source software system may be itself open or closed. The "closedness" may occur intentionally or accidentally. Having an intentionally closed software architecture means that the core group will consciously not reveal the structure to the general public. An unintentionally closed software architecture suggests that the structure exists in some people's minds only.

Documentation and testing

Documentation and testing are important aspects of the software development process. Good documentation allows people to use – and more specifically in open source projects, to understand and modify – the software. Thorough testing enables the users (and the developers) to have confidence that the software they are using (or developing) is going to function as expected.

These two areas are often overlooked or vary widely in the open source development process. Open source contributors tend to be more interested in coding than documenting or testing. This is probably due to the nature of open source that tries to replace the formal testing process with "many eyeballs" effect in eliminating the bugs. Also, adding comments in the source code is often perceived as sufficient for documentation. There has been some effort in addressing the problem of lack of documentation (e.g. the *Linux Documentation Project* [26] and *Mozilla Developer Documentation* web page [27]), but this is still a rarity for smaller open source projects. We have yet to find some sort of testing strategies for open source projects. They might exist, but implicitly and not open to the outside the project.

Licensing

The basic freedoms of open source software and how they differ from other software distributions were discussed in section 2.1 and 2.4 earlier. Here we consider the main varying features of OSD and FSF qualifying licenses[3]. Whether the software is viral or can become closed (proprietary) reflects the two main varying features of free and open source software.

Table 2 illustrates this with some of the more popular public licenses conforming to the OSD/FSF definitions. Viral licenses ensure that if any of the software code is used in other software developments then this will cause all of the software to come under the terms of that original license. The other varying feature

Table 2: Varying characteristics of open source licenses

Licenses	Is it viral?	Can it be closed?
GPL	Yes	No
LGPL	No	No
BSD	No	Yes
Q Public	No	No
IBM	No	Yes
Netscape (i.e. Mozilla)	No	Yes

[3] The term 'qualifying' refers to the four fundamental freedoms that both the OSD and FSF agree on.

concerns whether the license allows any of the original source code to be distributed in binary form only in future derived software products.

Operational support

In order to facilitate concurrent software development and fast controlled evolution, all open source projects implement some form of configuration management. This is enacted by using CVS, other tools, or even an ad-hoc solution using some web-based support.

The communication within communities related to specific open source projects is done almost exclusively by electronic means, which are also used to organise their work. The electronic means most commonly used are dedicated mailing lists, newsgroups, and web site. The exact structure and usage of web sites, mailing lists and newsgroups vary among open source projects.

Size

Size is not a distinctive measure in open source projects. Both involved-community and code base sizes vary widely from project to project.

4 Conclusion

The term open source is being used within the computing science community at large in a vague manner, consequently creating confusion and misunderstandings. In our efforts to understand open source we have done an extensive literature review, explored several web sites related to the topic, and interviewed some individuals and

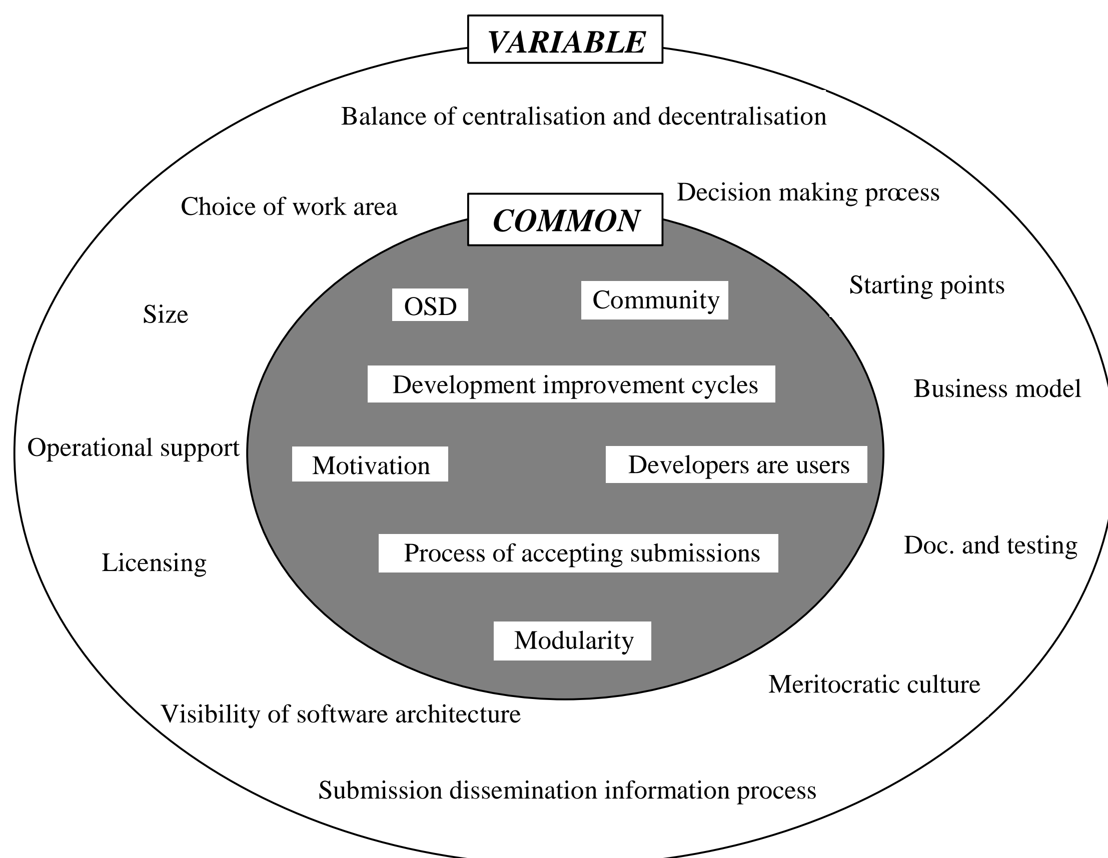


Figure 4: Open source characteristics – common and variable

corporations involved with open source. Our work was performed bearing multiple disciplines in mind.

We have determined many project characteristics that are relevant for open source. Some of these characteristics are common to all efforts, whereas others vary among open source projects (Figure 4).

How the various characteristics relate to the disciplines discussed in section 3.1 is highlighted in Table 3.

The set of open source characteristics we found can be used as a tick-list both for analysing and for setting up open source projects. We understand that there is no way that an absolute tick-list can ever be generated due to the variations that exist from one open source project to another, so additional variable characteristics may exist. Our proposed tick-list provides a starting point for understanding open source and its many meanings.

Table 3: Open source characteristics and disciplines considered

	Computing Science	Management Issues	Social Sciences	Psychology	Organizational Aspects	Economics	Law
OSD	√					√	√
Community			√	√			
Motivation		√	√	√		√	
Developers are user	√		√		√		
Process of accepting submissions	√	√			√		
Development improvement cycles	√	√	√				
Modularity	√	√		√			
Choice of work area	√	√		√			
Balance of centralisation and decentralisation		√			√		
Meritocratic culture			√		√		
Business model						√	
Decision making process	√	√	√	√			
Submission information dissemination process		√	√				
Project starting points	√	√				√	
Visibility of software architecture	√	√	√	√	√		
Documentation and testing	√	√	√				
Licensing						√	√
Operational support	√	√	√		√		
Size	√		√		√	√	

5 Future Work

There are many issues still left to be investigated with respect to understanding and exploiting the open source approach. Future work should further clarify the exact differences between open source and free software, as well as generate a table relating various existing open source and free software projects to the characteristics we set forth, while describing how each of these projects implement the variable parts. Some of the open questions that we would like to investigate include:

- How do open source and free software foster more dependable software development?
- One of the claims about the quality of software products developed as open source is the benefit experienced by having an very large number of reviewers examining the code. Consequently, the question arises on whether more reviews can replace formal analysis as a guarantee of dependability.
- What are the mutual influences between software architecture and group structure in open source or free software development?
- Does architecture decay occur faster in open source and free software?
- Is there responsibility attached to software developed as open source or free?

We shall also be looking into statistical information regarding open source and free software, as well as run controlled experiments to isolate and validate various assumptions from the community at large and ours.

6 Acknowledgements

This paper has been funded by the UK EPSRC project on Dependable Interdisciplinary Research Collaboration (DIRC – http://www.dirc.org.uk/). We would like to thank the volunteers that spent their time while sharing their experiences with us, as well as our colleagues from the DIRC project involved in the Open Source activity for various fruitful discussions contributing towards this paper.

7 References

[1] "The Open Source Initiative: Open Source Definition", http://www.opensource.org/docs/definition.html.

[2] "The Linux Home Page at Linux Online", http://www.linux.org/.

[3] "The Apache Software Foundation", http://www.apache.org.

[4] "mozilla.org", http://www.mozilla.org/.

[5] E. S. Raymond, *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*, O'Reilly & Associates, 1999.

[6] G. Moody, *Rebel Code: Linux and the Open Source Revolution*, The Penguin Press, 2001.

[7] C. Dibona, M. Stone, and S. Ockman, *Open Sources: Voices from the Open Source Revolution*, O'Reilly & Associates, 1999.

[8] A. Mockus, R. T. Fielding, and J. Herbsleb, "A Case Study of Open Source Software Development: The Apache Server," Proceedings of ICSE 2000, pp. 263-272, 2000.

[9] M. W. Godfrey and Q. Tu, "Evolution in Open Source Software: A Case Study," Proceedings of International Conference on Software Maintenance (ICSM'00), 2000.

[10] B. J. Dempsey, D. Weiss, P. Jones, and J. Greenberg, "A Quantitative Profile of a Community of Open Source Linux Developers", SILS TR-1999-05, 1999.

[11] H. Wang and C. Wang, “Open Source Software Adoption: A Status Report,” *IEEE Software*, March/April, pp. 90-95, 2001.

[12] J. Feller and B. Fitzgerald, “A framework analysis of the open source software development paradigm,” Proceedings of 21st International Conference on Information Systems, pp. 58-69, 2000.

[13] “SourceForge”, http://sourceforge.net/.

[14] “Geocrawler”, http://www.geocrawler.org/.

[15] “The Free Software Definition - GNU Project - Free Software Foundation (FSF)”, http://www.fsf.org/philosophy/free-sw.html.

[16] “The Open Source Initiatiative: History of the OSI”, http://opensource.org/docs/history.html.

[17] “Why Free Software is better than Open Source”, http://gnu.metagensoft.com/philosophy/free-software-for-freedom.html.

[18] E. S. Raymond, “Homesteading the Noosphere”, http://tuxedo.org/~esr/writings/homesteading/homesteading/.

[19] “Categories of Free and Non-Free Software”, http://www.gnu.org/philosophy/categories.html.

[20] “K Desktop Environment Home”, http://www.kde.org/.

[21] N. Slack, S. Chambers, C. Harland, A. Harrison, and R. Johnston, *Operations Management*, 2nd ed, Financial Times Pitman Publishing Series, 1998.

[22] A. Aoki, K. Hayashi, K. Kishida, K. Nakakoji, Y. Nishinaka, B. Reeves, A.Takashima, and Y. Yamamoto, “A Case Study of the Evolution of Jun: an Object-Oriented Open-Source 3D Multimedia Library,” Proceedings of 23rd ICSE Conference, Toronto, Canada, pp. 524-532, 2001.

[23] R. N. Britcher, *The Limits of Software: People, Projects, and Perspectives*, Addison Wesley, 1999.

[24] T. Bollinger, R. Nelson, K. M. Self, and S. J. Turnbull, “Open-Source Methods: Peering Through the Clutter,” *IEEE Software*, July/August, pp. 8-11, 1999.

[25] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, Addison Wesley, 1998.

[26] “Linux Documentation Project”, http://www.linuxdoc.org.

[27] “Mozilla Developer Documentation”, http://www.mozilla.org/docs/.

Appendix B: Contacts Pursued

Introduction

This appendix contains information relating to various contacts pursued during the course of our work in DIRC Project Activity 5.

1. Apache HTTP Server

The information on the *Apache HTTP Server* project was gathered by observing their web-sites, containing relevant documents [2]. Apache itself is an umbrella project composed of several projects and supported by the *Apache Software Foundation*. The Apache HTTP Server is the seminal project within Apache. Its aim is to provide "a robust, commercial-grade, featureful, and freely-available source code implementation of an HTTP (Web) server" [1]. The *Netcraft* survey found that the Apache HTTP Server is the most widely used web server on the internet, gaining almost 60% of the market share as of September 2001 [7].

There is a strong sense of community among the people involved in the Apache HTTP Server project. Within this project, there are five levels of participation, and starting with the group with the highest level of privileges, they are:

- Core development team, which is composed of the developers most closely involved in the project and are the decision makers. This team is often called the *Apache Group*, a term that is specific to the Apache HTTP Server project.
- People with direct access (CVS) to all source code
- People with CVS access to some subset of source code
- People participating in Mailing Lists and have earned enough respect for their votes to be counted
- Everyone else

Anyone can check out a copy of the source code, but only those in the top three levels can have changes incorporated back to the repository directly. A meritocratic culture governs the community, where people with proven track records are given more privileges ("the more you do, the more you are allowed to do"). This way, people can climb up to higher levels of participation.

There are two methods employed for controlling the source code (i.e. on how to commit the changes) [9]:

- *Review-Then-Commit (RTC)*: the proposed changes are discussed by the development team first before some sort of agreement is reached on whether to include it into the source code or not.
- *Commit-Then-Review (CTR)*: the core developers are allowed to change the source straight away, which is then examined and might be discarded later if it was deemed inappropriate.

Each method has its own advantages and disadvantages. The good thing of RTC is that only those changes that are really supported and approved by the development group will get applied. On the down side, RTC generally takes quite a long time (to get the consensus) and it does not encourage innovation very well. On the other hand, CTR speeds up the development process but it needs an increased care from the development team to make sure that the changes being made are

appropriate and will not cause any problem later on. The current Apache guidelines state that new ideas must be RTC, whereas patches can be CTR [3].

Communication among the developers is done through several mailing lists. The primary mailing list is *dev@httpd.apache.org*, which used to be called *new-httpd@apache.org*. This mailing list is open to all, although only subscribers can post directly to the list. There are also several private mailing lists for discussion among the Apache Group members, in which issues that are not appropriate for the general public (such as legal, personal and security issues) are discussed.

Decision making within the Apache community is mostly done through consensus drawn from email discussions. When that is not possible, the voting method is applied. The voting arrangement is a bit peculiar, the details of which can be found in [3].

2. Cocoon

At Lancaster our research focused on an OS project 'Cocoon' - which is related to the Apache OS project. Apache Cocoon 2 is an XML-based web publishing framework which runs on top of a webserver and a servlet engine. Cocoon is described by its originator as:

> " .. a software project that I started to "ease" the task of writing documentation creating tools that allowed publishing to be easier and more specific for their needs." (email from Stefano Mazzocchi)

but describes itself as:

> ".. a 100% pure Java publishing framework that relies on new W3C technologies (such as XML and XSL) to provide web content. The Cocoon project aims to change the way web information is created, rendered and delivered. This new paradigm is based on the fact that document content, style and logic are often created by different individuals or working groups. Cocoon aims to a complete separation of the three layers, allowing the three layers to be independently designed, created and managed, reducing management overhead, increasing work reuse and reducing time to market." [4]

The Cocoon project aims to change the way web information is created, rendered and delivered. This new paradigm is based on fact that document content, style and logic are often created by different individuals or working groups. Cocoon aims to a complete separation of the three layers, allowing the layers to be independently designed, created and managed, reducing management overhead, increasing work reuse and reducing time to market."

The Cocoon project is based on the idea of 'separation of concerns' and thereby changing the way that documents are 'created, rendered and delivered.' Examples of 'concerns would include:

- Presentation concern - layout, decoration, marketing
- Usability concern - navigating, purchasing etc. should be reasonably intuitive
- Internationalisation (i18n) concern - available in multiple languages, etc.

These concerns often overlap but the notion of 'separation of concerns' (SoC) enables a project to split into smaller, easier to manage pieces, ensuring that it is easier to understand the project, and the individual pieces, during initial development and especially during maintenance. Such an approach is also claimed to promote greater reusability and facilitate independent and parallel working. SoC also offers reductions in redundancy and limits the impact of change and Cocoon aims to provide support for collaboration between the different specialists involved in web page development.

Cocoon aims to separate functions that are normally mixed up into different layers. The content, formatting and logical organisation of documents will be disentangled from each other so that the work of creating or altering them can also be separated. In fact, different people are already given responsibility for constructing and maintaining different parts of a web page. But at the moment, on large, complicated websites, the different responsibilities often collide. Cocoon's hope is that the web designers should be able to work independently of web authors, for instance, but cannot because changing the layout of a page may also affect the content of page.

More generally, the advantages of separating content from style stem from the fact that each can be changed *independently* - provided that the relevant content-style contract is adhered to. These benefits are special cases of the benefits of SoC in general:

- **Parallel working** - both during creation and maintenance
- **Reduction of redundancy** - no need to "implement" a mockup
- **Easier to understand** (in theory!!) - With SoC, a web designer whose job focuses on presentation/graphics does not have to deal with e.g. complex HTML generators in servlets
- **Reusability** - Reuse a consistent style across many pages
- **Limited impact of change** - Changing the presentation of a (sub)site might not require editing every page / page generator!

The advantages of separation of concerns are outlined by one contributor to the Cocoon email discussion lists:

```
> I believe the most important Cocoon feature is SoC-based design.
>
> SoC is something that you've always been aware of: not everybody is
> equal, not everybody performs the same job with the same ability.
>
> It can be observed that separating people with common skills in
> different working groups increases productivity and reduces management
> costs, but only if the groups do not overlap and have clear "contracts"
> that define their operativity and their concerns.
>
> For a web publishing system, the Cocoon project uses what we call the
> "pyramid of contacts" which outlines four major concern areas and five
> contracts between them. Here is the picture:
>
> management
> /   |   \
> /     |     \
> style - content - logic
>
> Cocoon is "engineered" to provide you a way to isolate these four
> concern areas using just those 5 contracts, removing the contract
> between style and logic that has been bugging web site development since
> the beginning of the web.
>
> Why? because programmers and graphic people have very different skills
> and work habits... so, instead of creating GUIs to hide the things that
> can be harmful (like graphic to programmers or logic to designers),
> Cocoon allows you to separate the things into different files, allowing
```

> you to "seal" your working groups into separate virtual rooms connected
> with the other rooms only by those "pipes" (the contracts), that you give
>them from the management area.

Cocoon can run without modification on a number of different computing platforms, ranging from small handheld computers to IBM mainframes. Cocoon is ‘a publishing framework’ meant to support the derivation and implementation of other systems, software that perhaps provide detailed features that Cocoon itself does not.

Cocoon relies on ‘interoperable technologies' - its existence is a mixture of other technologies, existing at different levels and meant to work together. It relies on XML and XSL. Both languages are formal representations of the structure of a document. In contrast to HTML, which is a fixed representation of the way a text document should look when displayed on a computer screen, XML and XSL are, as their names suggest, mutable. Like a software framework, they are designed to be refined and extended. They work at different levels of abstraction. XML is concerned with content of a document. It breaks a document into a conceptually distinct set of chunks. Usually these chunks are hierarchically nested. So a refinement or extension of XML works by breaking a flow of text into a set of discrete components, each of which has a specific meaning within a given domain. XSL does the same thing, but the focus is instead on the formatting of the document, the way it is displayed on screen or page. This separation between the conceptual structure and the visual formatting of a document is important to what Cocoon is offering. The goal that Cocoon is setting itself is to refine the management of web pages at every stage ranging from creation to maintenance. It is a device that seeks to co-ordinate the collaborative work involved in web-sites. Cocoon as a device is designed to create and support teamwork in the domain of the creation of web pages. It focuses on ordering the conduct of work so that 'management overhead' is reduced.

3. NetBSD

NetBSD is one of the flavours of the *BSD (Berkeley System Development)* Unix operating system that is freely available and redistributable [6]. It is derived from 4.4BSD and 386BSD, but it has a strong emphasis on:

- High portability
  NetBSD is portable on a wide range of platforms. This is achieved by keeping everything cleanly split into Machine Dependent and Machine Independent areas.
- Clean design
  The primary goal is to have correct design and well-written code, which also helps to ensure the high portability aspect.

Investigation into the development of NetBSD was performed by conducting an interview with one of its developers who happens to live in Newcastle.

His contribution towards the NetBSD development started around 1997, concentrating on the Atari port of the NetBSD and the *curses* library. The main reason for his involvement is simply because he enjoys doing it. He spends around three hours per week on programming and 20 minutes per day on processing emails (from mailing lists etc.), mostly at nights and weekends. He tends to work longer hours when there is something new to work on, as well as during winter. He is not involved

in any other open source project and he is not interested to get into ideological discussions regarding open source. He usually works in small groups, mostly online (he has met only three other NetBSD people in person). He is careful about what he submits. This is due to the peer pressure (he would not like to submit something that creates problems to the rest of the project) and also because there is no time constraint imposed (hence he can take more time to work on his submission). He believes that corporations will not go open source if there is no threat of competition, and the timing for launching an open source software is important in determining its success.

There were several points made concerning the characteristics of the NetBSD project in general:

- Regarding people:
    - There are around 250 developers, but among those, only 50-60 commit code regularly, and an even smaller number (around 20 people) contributing a lot towards the project. There is a core group of about 5 developers, who act as technical managers. They set the direction and goals of the NetBSD project as a whole, promote people's interest in the NetBSD project and the system it produces, and consider the serious architectural questions that need to be addressed if the NetBSD Project is going to keep producing a viable system [5].
    - Newcomers are sponsored by existing developers before being assimilated.
    - There is a very friendly and collaborative environment, although at times, it might become confrontational.
- Regarding code submission:
    - Anyone can check out the source code, but only the developers can check the changes back in.
    - It is possible for a change to be undone.
    - Technical issues are dealt with by consensus only, i.e. there is no voting involved. Conflicts are either resolved or die out, and no conflicting code is put in.
    - Everything is peer reviewed by many people.
- Regarding communication:
    - One main mailing list and several side ones for specific purposes, e.g. port specific, particular areas of code.
- Regarding other BSDs:
    - Forking in BSD was caused by personality clashes.
    - Other BSDs have different emphasis (NetBSD emphasises on clean and portable code).
    - The branches learn from each other on ideas, but they do not copy each other's code.

Note: The NetBSD project does not claim to be open source. The Berkeley license is generally used as the template for the NetBSD license terms, but it does not apply to all works in the source tree. Parts of it are covered by the GNU General Public License (GPL) of the Free Software Foundation.

4. Interactions with HP-Arjuna Lab

The HP-Arjuna Lab is located in Newcastle and was originated from the research conducted by the distributed system group of the University of Newcastle's Computing Science department. They are currently contributing towards the development of some open source software to be used with their

developed platform. Given that context we gave a presentation about open source and the work on DIRC's open source Project Activity at their site, as well as held several meetings with them trying to define potential modes of cooperation in the topic. The feedback we received there to our presentation was very interesting. The people from the lab raised several questions from a corporation's perspective.

One of the main concerns the people at the HP-Arjuna Lab have is the lack of systematic testing and of system's documentation relating to the open source projects that they have been looking into. This has lead to discussions on how to remedy this problem and on trying to understand the reason why there are much fewer volunteer technical writers as opposed to code writers.

The topic of having corporations embracing open source was also discussed. There are many risks involved in such action. These include but are not limited to:

- Adopting the appropriate business model to ensure that a corporation can benefit from/with open source software.
- An open source software project may die out at any point in time. The corporation may then need to maintain the full package.
- There are no guarantees regarding the quality of open source software.
- It may be hard to motivate employees to contribute to open source software projects. The environment is not always friendly and individuals may not feel comfortable about exposing themselves to having their work criticised in a less than constructive manner.
- There is no legal responsibility associated with open source software. This means that a corporation using open source software to run their business or within the solutions it provides may become fully liable for problems inherited from such software.
- There are no deadlines in the open source community. A corporation cannot rely on having certain parts of the open source software developed within a given time frame. Volunteer contributors work towards parts of the software that trigger their interest, at their own pace using their free time, and only submit their contributions when they believe their code is in a stable and acceptable shape. Contributions must then be evaluated by other volunteers (from the core group in cases like the Apache HTTP Server) also working at their own pace on their free time.

Note that some of the risks mentioned above also exist with respect to Commercial Off The Shelf (COTS) packages in varying degrees.

An interesting question that came out of our discussions at the HP-Arjuna Lab was about what would happen if open source software became endorsed by the authorities. Would that deter the usual contributors?

The HP-Arjuna Lab is now building an infrastructure around an existing open source software project. Within that given project's community, they are mainly focusing on building proper test suites and documentation. The intention at the lab is to be able to sell their infrastructure using this OSS package, while reducing some of the concerns that their clients may have regarding the quality of the resulting product. This effort is to officially start in October 2001. We have agreed that they will share with us monthly updates on the matter.

5. Interviewing IT Security Coordinator

We held a meeting with Mike Ellison, the IT Security Coordinator of the University Computing Services (UCS) at the University of Newcastle to discuss his views and observations with respect to open source software in the university environment. This was done by drawing comparisons between open source software and commercial software. His observations on open source were based mainly on Linux and the Apache HTTP Server, and not on less known open source software packages.

The general problem experienced by UCS while trying to keep systems secure is that of obtaining patches for vulnerabilities in time. Maintaining systems secure once a vulnerability is discovered by hackers is a three stage process:

1. Hacker security probes initiated – tracking probes into the systems IT security officers determine suspicious patterns coming in.
2. Recognising resulting vulnerabilities – the suspicious patterns detected in the probes are further investigated, trying to recognize existing known vulnerabilities.
3. Attaining patches – patches for known vulnerabilities must be found and installed in a timely fashion to prevent malicious attacks.

The effort behind maintaining systems secure is made worse by IT decentralization, because it becomes much more difficult to co-ordinate who has what version of various software packages and to enforce the installation of patches accordingly. Mike Ellison's observation is that with open source packages it is much easier for individual users to download their own version of a software package unbeknownst to their central organization, making it extremely hard for the support staff to ensure the timely installation of patches throughout the organization.

On the other hand, closed software vendors are slower at responding to recognized vulnerabilities on existing packages because they have heavily bureaucratic software processes in place, and there is no profit to be made in providing patches, developers are more productively used in developing new versions for their software packages. Open source software communities are much faster at responding to recognized vulnerabilities because there are more "eyes" available to detect the problem and more "hands" available to fix them. Additionally, they have a less bureaucratic process in place, making patch code more easily inserted.

6. Exchange with the GENESIS project

GEneralised eNvironment for procEsS management In cooperative Software engineering (GENESIS) is a European project that intends to develop an open source environment that supports the co-operation and communication between software engineers belonging to distributed development teams involved in modelling, controlling, and measuring software development and maintenance processes. We had some exchange of ideas with one of the partners of this project, Dr. Cornelia Boldyreff from the University of Durham. Within GENESIS, it is not yet clear as of when their results will be made open source: from the very beginning, start closed and move to open source later on, or do all the development in a closed source manner and move the results to open source upon completion. It has also not yet been decided which flavour of open source will be adopted. We have agreed that we might be able to help them down the line, once they are ready to define these aspects, but it is far too early in the GENESIS project for DIRC's Project Activity 5 to be able to benefit. Future contacts here may prove to be fruitful.

7. References

[1] "About Apache", online at *http://httpd.apache.org/ABOUT_APACHE.html* .

[2] "Apache Project Development Site", online at *http://dev.apache.org/* .

[3] "Apache Project Guidelines and Voting Rules", online at *http://dev.apache.org/guidelines.html* .

[4] "Cocoon Website", online at *http://xml.apache.org/cocoon/* .

[5] "NetBSD People", online at *http://www.netbsd.org/People/* .

[6] "The NetBSD Project", online at *http://www.netbsd.org/* .

[7] "Netcraft Web Server Survey", online at *http://www.netcraft.com/survey/* .

[8] Arief, B., C. Gacek, and T. Lawrie, "Software Architectures and Open Source Software - Where can Research Leverage the Most?", *Proc. 1st Workshop on Open Source Software Engineering*, Toronto, Canada, pp. 3-5 (15 May 2001).

[9] Coar, K., "Apache and Open-Source Development, online at *http://golux.com/coar/slides/Apache-development.pdf*" (1999).

[10] Feller, J. and B. Fitzgerald, *Open Source Software Development* (to be published).

[11] Hunt, F., "Developer Motivations in Open Source Development" (in preparation).

Appendix C: How 'hacking' hides a project: from software engineering to open source and back again

Adrian Mackenzie and Mark Rouncefield
Department of Computing
Lancaster University
Lancaster LA1 4YR

Abstract:
This paper is concerned with understanding the character of open source (OS) project work as part of a project investigating 'dependability' in computer systems. Using data from interviews, email communications and the code itself we describe how the orderliness of such projects is achieved in contrast, perhaps, to stereotypical views of open source as mere 'hacker' projects. We use our data to explicate the ways in which OS projects are accomplished and the ways in which the various participants observably, reportably, accountably orient their actions, their contributions to the project as a whole. The contrast we draw is not with idealised or theoretical views of software development but with ethnographic and ethnomethodological studies of software production that emphasise the project as a practical, ongoing achievement. In this way we move beyond idealised or stereotypical versions of the open source project towards a praxiological understanding of open source development.

• Introduction:

As society's dependence on computer-based systems increases, the systems themselves become ever more complex and achieving dependability in these systems, and demonstrating this achievement in a rigorous and convincing manner, is of crucial importance. One of the attractions of Open Source development approaches, at least as suggested by its advocates, comes in terms of the improvements in reliability dependability, and flexibility for the process of software development and the quality of the end product:

> "Open source promotes software reliability and quality by supporting independent peer review and rapid evolution of source code …Mature open-source code is as bulletproof as software ever gets." (1).

The OS approach, characterised as 'massively diverse human scrutiny', or peer-reviewed software, extends the idea of review and introduces a way of confirming final decisions about the inclusion of changes to a system. Examples of open source (e.g. operating systems, development tools, web and mail servers) indicate that a community can be built which can create software that is highly reliable.

However, even studies that might be regarded as broadly supportive of OSS development have pointed to the scarcity of what might be regarded as conventional attributes of orderly software development. Mockus et al (2000) for example (2), use email archives to develop quantitative measures of dependability attributes such as defect density and problem resolution but suggest that "there is no project plan, schedule, or list of deliverables" and that OS "lacks many of the traditional mechanisms used to coordinate software development, such as plans, system level design, schedules, and defined processes". This links with popular, if fanciful, conceptions of open source software development as the product of 'hacking.' Even when 'hacking' is distinguished from 'cracking' (attempting to breach the security of computer systems), it still often implies

unplanned, improvised work. The great attention open source software has attracted in technical and mainstream press fosters that view. It has focused on relatively small bands of highly motivated, even visionary, programmers working in geographical isolation. They seem to be engaged in a brilliantly productive yet free-wheeling production of new software artifacts. Elaborate operating systems and major pieces of software infrastructure (e.g. Apache) seem to flow from their fingertips.

1.1. Software engineering vs hacking

How different is open source software produced through hacking from the software produced by software engineers? Both software engineers and open source programmers tend to stress the differences. From the perspective of software engineering, [Vixie, 1999] argues:

> "Open Source developers often succeed for years before the difference between programming and software engineering finally catches up to them, simply because Open Source projects take longer to suffer from the lack of engineering rigor". (3)

Vixie bases his conclusions on a comparison between the formal textbook methods of software engineering (e.g. [Sommerville, 2001](4)) and what he labels, somewhat derogatively, 'programming'. If professional software engineers are eager to point out the 'lack of rigor' of open source programming, open source programmers have been even quicker to distance themselves from conventional software engineering. The famous hacker, Eric Raymond, writes:

> "What I saw around me was a community which had evolved the most effective software-development method ever *and didn't know it*! That is, an effective practice had evolved as a set of customs, transmitted by imitation and example, without the theory or language to explain why the practice worked" [5]

The contrast with Vixie's position could not be greater. Instead of an absence of method, Raymond is suggesting that the development practices involved in open source software was so radically new that there was no way to explain it.

We propose that much shared ground runs between these two diametrically opposed positions. However, this shared ground is not highly visible. Instead of a deficiency in methodical rigor (Vixie) or an effectiveness too novel to be even explained (Raymond), we would like to describe a habitually ignored middle-ground between the highly formalized vision of software engineering and the myth of collective improvisation. There are important continuities between the two types of activity which neither account recognise. A scarcely visible infrastructure of practices and contrivances is woven through both open source and professional software engineering. The analysis of a significant case study, the Cocoon project (http://xml.apache.org/cocoon), will allow us to show how open source projects are grafted onto practices developed in software engineering.

1.2. Making things orderly

Following Button & Sharrock (6), we suggest that the continuities and some important differences between OS and conventional software projects consist in the *ordering practices* commonly found in open source software projects.

> ".. much effort is expanded on contriving devices which will provide 'orderliness' in the conduct of work and in ensuring that such devices can be implemented and enforced. These devices are meant

> to enable the achievement of orderly work where it requires the collaborative participation of many individuals, and may, crudely, be characterised as devices which are designed to create and support teamwork". (p 373)

These practices are intricate and fine-grained, and, as we will show in the case of the Cocoon project, criss-cross every level of project work, ranging from end-user documents down to source coding. Button and Sharrock also highlight the importance of 'the project':

> It is commonplace to refer to engineering projects and the easy way in which this term is used can detract from the recognition that the project is a prominent way in which engineering work is socially organised so as to confront the sorts of contingencies that face software engineering that we have alluded to such as the threatened curtailment because of, for example, drastic slippage, or such as the pressures to abandon good practice." (p 372)

While the contingencies may be different, the notion of the project retains strong relevance to open source software. They too involve social and technical organisation, albeit now directed towards the contingencies of geographical dispersion, fluctuating teams of participants, and open-ended timelines.

2. Method - studying Cocoon

Our research focuses on an OS project 'Cocoon' - which itself is related to the Apache OS project. Cocoon is described by its originator as:

> " .. a software project that I started to "ease" the task of writing documentation creating tools that allowed publishing to be easier and more specific for their needs." (email from Steffano Mazzochi)

but describes itself as:

> ".. a 100% pure Java publishing framework that relies on new W3C technologies (such as XML and XSL) to provide web content. The Cocoon project aims to change the way web information is created, rendered and delivered. This new paradigm is based on the fact that document content, style and logic are often created by different individuals or working groups. Cocoon aims to a complete separation of the three layers, allowing the three layers to be independently designed, created and managed, reducing management overhead, increasing work reuse and reducing time to market."(7)

The goal that Cocoon is setting itself is to refine the management of web pages at every stage ranging from creation to maintenance. It is a device that seeks to co-ordinate the collaborative work involved in web-sites. As a mode of ordering a certain kind of documentary work, Cocoon conforms to what Button and Sharrock describe as an ordering device. Cocoon as a device is designed to create and support teamwork in the domain of the creation of web pages. It focuses on ordering the conduct of work so that 'management overhead' is reduced.

Our analysis is drawn from interview, source code and email archive data. We use this data to explicate the ways in which OSS projects are accomplished and the ways in which the various participants observably, reportably, accountably orient their actions, their contributions, their emails to the project and to notions concerning 'good' or 'elegant' code, ideas about 'ownership' and so on. Much in the way that Weider (8) describes the 'convict code' - as a resource that is drawn upon to account for and understand action rather than a simple normative stipulation and explanation for behaviour - so the OSS

Cocoon community uses sets of ideas about coding, about participating in an OSS project and so on as resources for their accounting practices in the course of contributing to the project itself.

Our interest is primarily in understanding the character of the OSS project as a 'project' - how it actually 'gets done'. The contrast we draw is not with idealised views on software projects or open source development but with ethnographic and ethnomethodological studies of 'realworld, real time' software production (Button and Sharrock 1996). These emphasise the project as a practical, ongoing achievement and concentrate on the everyday, mundane aspects of keeping a project going. We place particular emphasis on various kinds of 'ordering work' that occurs at a number of levels throughout the project and draw attention to the set-up of the website, coding, email correspondence and the project archive. As an instantiation of 'virtual teamwork' Cocoon as a project necessarily needs to attach considerable importance to issues of distributed coordination; plans and procedures; and developing an 'awareness of work'. The concept of the 'virtual team' (9) is intended to denote an organisational form consisting of networks of workers and organisational units, linked by information and communication technologies, that flexibly co-ordinates activities, skills and resources to achieve common goals without traditional hierarchical modes of central direction or supervision. Such teamwork, 'less fettered by the constraints of traditional hierarchies and spheres of responsibility, engenders a heightened sense of empowerment, commitment and collective responsibility' (10). Whilst with conventional software projects understanding the organisational context is vital - "software engineering is often carried out within an organisational environment which threatens to overwhelm the project"(Button and Sharrock) - with OS the position is more complicated. While the OS may be likened to a 'virtual organisation' there are manifestly real problems both connected to the organisation within which the code contributor ordinarily works (for example in time constraints), and within the virtual team itself to do with communication and awareness.

3. Achieving the orderly character of OS project work

In their salutary paper on the organisation of collaborative design and development in software engineering, Button and Sharrock (1996) point to 'the project' as a formatted organisational arrangement within which software engineers typically coordinate their design and development work and make their work mutually and organisationally accountable. They carefully document how engineers achieve the formatted arrangements of the project and how they display an orientation to these arrangements in the way they order and accomplish their work. In project work the organisation of the work itself can be a source of troubles that is accommodated through the organisation and re-organisation of work. Ordering work as a project does not in itself ensure the orderliness of work or provide remedies for all contingencies, instead the project structure and plan is an achievement of everyday work and a response to and recognition of the contingent nature of such work. In these circumstances a number of devices are noticeable for ensuring the orderly character of work. 'Phasing' ensures that necessary tasks are adequately completed and provides for the interdependence of activities and the

recognition of uncompleted stages. The ’methodic handling of tasks’ provides for some kind of system in the confrontation and elimination of problems. ’Orienting to the project as a totality’ provides a method for project teams to keep each other’s progress in view and make it visible to others. ’Measured progression’ refers to procedures and devices - organisational metrics - for documenting how much of the project has been done and what remains; checking work against schedules and so on. Finally they note how ’making sure the documentation gets done’ is regarded as ’dirty work’ not an integral part of job and superfluous to engineers practical needs.

3.1. The website as an ordering device.

Quite clearly the website can be viewed as an ordering device orienting both ’newbies’ and established project members to features of the project through devices such as the menu-bar(Fig.1), the ’to do list, requests for help (Fig 2.), advice for contributors (Fig 3) and so on (11). The advice on making a contribution for example describes a number of stages through which a ’typical contribution’ may go and how any contribution is treated once submitted. The ’to do’ list prioritises requirements for code, documentation, samples and design from ’high’ (Fig 4) - "upgrade Turbine-pool" - to low and a ’wish’ list. The list also assigns particular tasks to named individuals.

Fig 1.

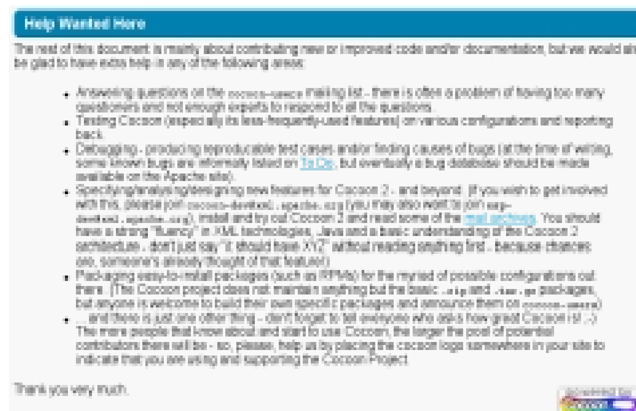
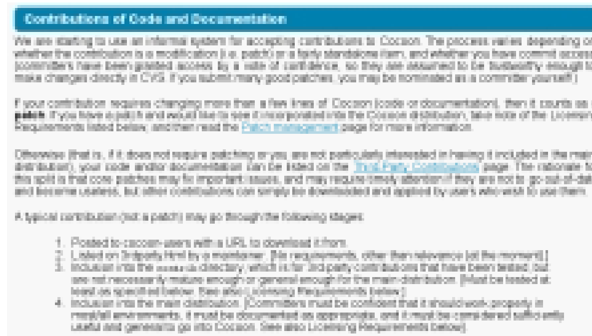


Fig 2.

One way of understanding the working of the website, as effectively the ’desktop’ or ’front-office’ of the project, is in terms of ’affordances’ of knowledge (12). The website provides for project members knowledge of the state of the project, where they are up to what needs to be done etc - and it was evidently designed with this possibility in mind. The website is both the public focus for work and a visible, a publicly available, record of work that has been done or remains to be done. In other words, what these representations do, among other things, is make the work ‘visible’ so that it can be ‘taken note of’, ‘reviewed’, ‘queried’, and so on, by others involved. They put the work on display so that others may be aware of it.

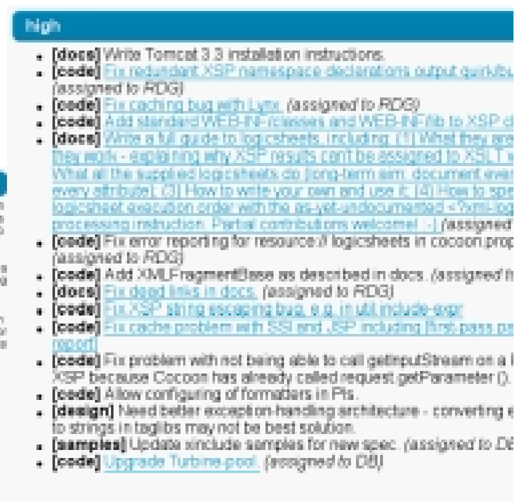


Contributions of Code and Documentation

A typical contribution (not a patch) may go through the following stages:

1. Posted to cocoon-users with a URL to download it from.

Fig 3.



high

- [docs] Write Tomcat 3.3 installation instructions.
- [code] Fix redundant XSP namespace declarations output (assigned to RDG)
- [code] Fix caching bug with Lynx. (assigned to RDG)
- [code] Add standard WEB-INF/classes and WEB-INF/lib to XSP c
- [docs] Write a full guide to logicsheets, including: (1) What they are ... they work ... (2) What all the supplied logicsheets do ... (3) How to write your own and use it. (4) How to spe ... logicsheet execution order ... processing instruction. Partial contributions welcome! :-) (assigned
- [code] Fix error reporting for resource:// logicsheets in cocoon.prop (assigned to RDG)
- [code] Add XMLFragmentBase as described in docs. (assigned t
- [docs] Fix dead links in docs. (assigned to RDG)
- [code] Fix XSP string escaping bug
- [code] Fix cache problem with SSI and JSP including first-pass ... report
- [code] Fix problem with not being able to call getInputStream on a XSP because Cocoon has already called request.getParameter ().
- [code] Allow configuring of formatters in PIs.
- [design] Need better exception-handling architecture - converting to strings in taglibs may not be best solution.
- [samples] Update xinclude samples for new spec. (assigned to DB
- [code] Upgrade Turbine-pool. (assigned to DB)

Fig 4.

Our interest is how the different features of the website are constructed so as to 'afford knowledge' as to the working division of labour by which the various tasks on Cocoon are performed. The notion of affordance used here treats perception as resolutely embedded in particular cultural practices. Just being fully enculturated members of the Cocoon project means being able to use website and associated email system, to see unproblematically what needs to be done urgently, what is less important, what the next phase of the project is and what progress they are making. The website (and the email system) provides the project team with the means to see at a glance, and recognise immediately what is going on in the project. The website thereby also acts as a 'technology of accountability' (13; 14) enabling members to see, at a glance, the status of the project and calculate whereabouts they might be in the organisational and temporal cycle of events.

3.2. Order by email: Finding order in the archive

Examination of the email archive is also instructive of the various ways by which order is accomplished in an open source project. Although the richness (and occasional vehemence) of the exchanges is difficult to adequately captured here what is evident is the way in which email communication provides for the administration, voting and scheduling of the project as well as orienting to the project as a whole. Despite the opinion that: *""scheduling" is ultimately impossible: we are talking about volunteers that spend their free time. How can you tell when you'll finish planting your garden? or when you'll finish your WW2 tank model? When you do it. Period"* (email correspondence) it is clear that a lot of communication through email is about the scheduling of activity. Thus:

> " Okay, how about this for a schedule: (too formal, I know!) If anyone wants to change it, better make it quick!
>
> * I'll commit what I've done so far on the FAQ tomorrow (Saturday), plus some other minor changes. ..
>
> * Feature freeze 00:00 GMT (not BST) Monday - i.e. no new features, only minor bug fixes and doc improvements

> * Around the same time I'll send emails to cocoon-users and cocoon-dev asking for testers to download from CVS and test, and report back what configuration they have and whether there were any problems."

Scheduling is also affected by the 'lazy consensus' system of voting (whereby anyone who does not vote is assumed to concur) as the following email makes clear:

"Are we all agreed to implement content aggregation in the way specified by Stefano in his RT? I've been pondering it for the last few weeks and playing some thought experiments, and I'm definitely +1. How about the rest of you?"

What also comes over is an orientation to the Cocoon project as a whole both in terms of the management and administration of the project as well as some notion of a 'code' or orientation to the ethos of open source in general. For example the following email:

> "I got a little issue here. I voted -1 on the engine synch. patch since I thought that we shouldn't put in a patch that messes around with cocoon that deeply shortly before a new major release. according to the terms of the asf project constitution: my veto is binding unless you convince me otherwise".

Brought this response:

> "So, for the sake of the "dignity" of the Cocoon project overall, including myself, I'd like to get it in a more shipshape condition. Now you may say that's about image and PR and marketing etc. which we are things we should stop getting hung up about - but it's not just image, it's about code quality (and quality of the docs).
>
> While I can see the sense in being cautious just before a release, as a general principle - if I were a complete outsider I expect I'd _still_ think that leaving these known simple bugs in was... odd, for an open source project".

What becomes apparent in these discussions is a clear orientation to the project as a whole rather than a collection of tasks. The email system has become a way of keeping each other's progress in view and making their own progress visible to others through activities such as involving themselves in others activities and tasks through talking them through; and knowing where their worked impacted on others and informing them.

3.3. The open source 'code' of work.

Finally, the development and orientation to some notion of an open source code regularly appears in the email discussions on 'good' or 'elegant' code, design philosophy and the principles of open source. Perhaps the best example came in the various responses to the following upset contributor:

> "> removed that unportable (and useless) ASCII art along with (slow) system out (logs >/are there for a reason) and clean up messy code..
>
> I'm sorry that you think my coding is messy, and I would prefer that you tell me first, being its my code..
>
> I would appreciate you all to refer to the author of the code first before spreading bullshit.."

That brought the following reply (amongst many):

> "I think it is important to recognise that we are working on an open source project. I know that there are "code ownership" political issues in many companies, but I would sincerely hope that those attitudes would not bleed into this project. Once the code has been committed, it is no longer 'your code' it is 'our code', and we are all committed to making that code as good as possible. It's one of the strengths of open source."

Without necessarily following Edwards' (15) suggestion of 'epistemic communities' what comes over in this email exchange - too lengthy to fully document here - is the outline of

some idea of a ’code’ that ’governs’ or shapes open source. This is depicted in even more detail in sociological accounts of the ’hacker ethic’ and is often used to provide some kind of explanatory account - ’why hackers do it’. In these approaches compliance to the ’code’ is used as an explanation of behaviour. The open source community is simply seen as governed by set of normative rules. Our argument is rather different and subtler since we are not interested in offering explanatory or motivational accounts of open source but instead of understanding how these projects ’get done’. We are interested in examining how the OS community both construct and make use of the code as a resource in the course of their mundane interactions where the code is used by parties to the interaction as displays, or accounts of what those actions are. Orienting to the code in an email, for example, can be used for changing topic, defending or defeating a proposed course of action and for accounting for one’s actions in an acceptable way. As Weider (8) comments (though on a very different kind of code):

> "The code then, is much more a method of moral persuasion and justification than it is a substantive account of an organized way of life.’ (p 175).

4. Working the ’code’

4.1. Ordering devices at work in the source code

Can we find ordering devices embedded in the source code itself? The availability of the source code is one of the most salient attributes of the open source phenomena. Presumably the code itself should bear the marks of the ordering devices since they afford orderliness in the conduct of work. They allow a *project* to take place, even if its documents, its ‘deliverables’ and the relations between developers and users looks quite different to that envisaged by accounts of fully-equipped software engineering projects. The practices and ordering devices surrounding open source code are concerned with regulating how it is read, and channeling how it is written and re-written. Open source programmers are often encouraged to read the code, as well as reading the documentation that accompanies the code. In this domain, we expect to find ordering devices concerned with reading and writing code.

Some source code for a part of the Cocoon system is shown below. This code defines a part of the system that manages the caching of web-pages processed by the Cocoon framework. It helps the system decide whether a particular item (such as a web page, or an xml file) should be kept in system memory ready for another page request, or shunted back onto secondary storage, such as a hard disk, because it is not being frequently requested.

```
package org.apache.cocoon.store;

import java.io.*;
import java.util.*;
import org.apache.cocoon.framework.*;

/**
 * This class implements a memory-managed hashtable wrapper that uses
 * a weighted mix of LRU and LFU to keep track of object importance.
 *
 * NOTE: this class is _HIGHLY_ un-optimized and this class is _CRITICAL_
 * for a fast performance of the whole system. So, if you find any better
 * way to implement this class (clever data models, smart update algorithms,
 * etc...), please, consider patching this implementation or
 * sending a note about a method to do it.
 *
 * @author <a href="mailto:stefano@apache.org">Stefano Mazzocchi</a>
 * @author <a href="mailto:michel.lehon@outwares.com">Michel Lehon</a>
 * @version $Revision: 1.12 $ $Date: 2000/05/16 21:11:51$
 */

public class MemoryStore implements Store, Status, Configurable, Runnable {
    /**
     * Indicates how much memory should be left free in the JVM for
     * normal operation.
     */
    private int freememory;

    /**
     * Indicates how big the heap size can grow to before the cleanup thread kicks in.
     * The default value is based on the default maximum heap size of 64Mb.
     */
    private int heapsize;

...

    class Container {
        public Object object;
        public long time = 0;
        public int count = 0;

        public Container(Object object) {
            this.object = object;
        }
    }

    /**
     * Initialize the MemoryStore.
     * A few options can be used :
     * <UL>
     *  <LI>freememory = How much memory to keep free for normal jvm operation. (Default: 1 Mb)</LI>
     *  <LI>heapsize = The size of the heap before cleanup starts. (Default: 60 Mb)</LI>
     *  <LI>usethread = use a cleanup daemon thread. (Default: true)</LI>
     *  <LI>threadpriority = priority to run cleanup thread (1-10). (Default: 10)</LI>
     *  <LI>interval = time in seconds to sleep between memory checks (Default: 10 seconds)</LI>
     * </UL>
     */
```

Some of the ordering devices present in this code are common in software engineering today. Some are specific to open source style projects. A combination of generic and specific orderings are present in this example.

At a fairly coarse-grained level, the formatting of this Java code is obvious from the layout on the page. Clearly text formatting is a kind of ordering to do with *reading*. The

formatting is a contrivance that allows the code to be read more easily by different kinds of reader. This particular code shows evidence of being ordered for at least three distinct kinds of readers.

Firstly, it affords reading by humans. By virtue of the restricted line lengths, the use of nested indentation to represent something about the flow of execution of the program, and the use of blank space to show separations between different components of the code, people reading the program can begin to interpret the code as a set of operations and structures. For such readers, particular zones of the text are marked out for different modes of reading. Any line beginning with an asterisk will attract attention as a comment, something programmers particularly addresses to human readers, including themselves. This may be an explanation, an apology or a request (e.g. *"So, if you find any better way to implement this class (clever data models, smart update algorithms,etc...), please, consider patching this implementation or sending a note about a method to do it.*"). More than half of the source code text in this example consists of comments. By contrast, any line that begins with a keyword like 'class', 'public' or 'private' will stand out to a programmer since it signals an important boundary in the organisation of the program. Reading these lines involves separating out keywords, operators and syntax marks from the proper names that the programmer(s) have used to designate elements of the program. Words such as 'freememory' or 'getStatus' describe designate places where an important values is stored, or places where significant operations will be specified. On these lines, the reader is alerted that they must read the code as naming something specific to this program.

Secondly the source affords reading by the compiler. By virtue of such things as the termination of lines by semicolons, the use of brackets of various kinds - {}, [], and (), - and the presence of keywords such as 'public' and 'class', the compiler will be able to parse the source code file into an executable file containing instructions that can for the Java Virtual Machine. Thirdly, this code allows reading by another specialized program, javadoc. Javadoc is a program that will take these source files and generate html-formatted documents from them. These documents, the "API [Application Programmer Interface] docs" will be read by other programmers who want to use the operations furnished by this piece of code, without looking at the actual code itself. A brief extract of the html API document that the program javadoc produces when it processes the source code quote above shows again a complicated and fairly fined-grained ordering of information.

Method Summary	
boolean	**containsKey**(java.lang.Object key) Indicates if the given key is associated to a contained object.
void	**free**() Frees some of the fast memory used by this store.
java.lang.Object	**get**(java.lang.Object key) Get the object associated to the given unique key.
java.lang.String	**getStatus**() Returns the signature of this store implementation
void	**hold**(java.lang.Object key, java.lang.Object object) Holds the given object in a volatile state.
void	**init**(Configurations conf) Initialize the MemoryStore.
java.util.Enumeration	**list**() Returns the list of used keys.
void	**remove**(java.lang.Object key) Remove the object associated to the given key and returns the object associated to the given key or null if not found.
void	**run**() Background memory check.
void	**store**(java.lang.Object key, java.lang.Object value) Store the given object in a persistent state.

Like the source code, the html-formatted documents will be browsed extensively by programmers involved in either using or extending the Cocoon framework. Developers involved in the open source project and technically sophisticated users (such as web-site architects and developers) will both refer to these documents. However their presentation as a web-page implies important differences. These documents are not editable, whereas the source code is. Secondly, the use of headings, hyperlinks, tables, different font sizes and types for the text is clearly directed towards quicker movement around in the text.

Finally, the link between reading and writing code - through a text editor and a repository for source code - is important since almost every open source software development project currently active makes use of a single important ordering device, a program called ‘cvs’, Concurrent Versioning System. This device is profoundly enabling for open source development in several respects. Itself an open source project, CVS makes it possible for almost any number of people to read and write copies of the same source code files, and amalgamate the results. CVS’s developers claim both that *“its client-server access method lets developers access the latest code from anywhere there's an Internet connection”* and that *“its unreserved check-out model to version control avoids artificial conflicts common with the exclusive check-out model.”* Revision numbers in the source code indicate how many times a source code file has been modified. In our example, revision 1.12 implies that this file has been edited at least 12 times, although it may have been read many times before. Talk about CVS is a major feature of the email communication amongst developers. Many email messages describe events in the CVS repository. For instance, in describing a milestone release of Cocoon, the developer responsible writes to the developer list:

> > > *Now the CVS stuff:*
> > > *- I tagged the beta with cocoon_20_b1*
> > > *- I checked in the build.xml with the new version 2.1-dev*
> > > *- I made a branch of cocoon_20_b1 with the name cocoon_20*

> > > - I checked in the build.xml with the new version 2.0b1-dev under
> > >   the branch cocoon_20_branch.
> > > So the HEAD is the 2.1 version and the 2.0 is a branch.

The developer describes in detail the operations that had to be carried out so that the software source was named in an orderly way, and accessible to other readers and writers of the code. The developer's descriptions of their actions within CVS render the contents of the archive manageable for other developers. If for instance, a particular 'build' or version of the project does not have a commonly agreed upon name, then the team of developers cannot synchronise their editing of the source code. Agreeing on what the name of the version will be is sometimes not enough. It may still leave open the question of where in the CVS repository further changes will take place. Another developer replies to the preceding message:

> > Yes, that good. I assume all the new development will happen only on
> > the HEAD and bug fixes will be applied to both HEAD and 2.0b1-dev
➢ *> branch. Is this the common understanding?*
➢

Again, negotiations around how source code will be named, stored and retrieved are taking place here, but in this case about future changes to the code. Without these negotiations, the project would start to fall apart.

4.2. Differences between professional and open source software development

We can now shift the focus of analysis to address the question of whether these reading contrivances show anything specific about open source software development. The formatting of the code, the use of Java, and the method of documenting the source (using javadoc) are all textbook or industry standard. Almost identically formatted and commented code can be found on any Java-related industry web-site, or in any Java programming textbook. At the level of the reading and writing practices carried out by programmers using text editors or integrated development environments, the ruling conventions in this open source project come from well beyond the domain of open source software projects. There is no evidence of a specific style of coding.

However, there are differences that show that this code belongs to an open source project. Firstly, there is a request to anonymous readers to contribute a better algorithm or data structure for part of the system that is said to be *'_CRITICAL for a fast performance of the whole system. ... please consider patching this implementation.'* It is unlikely that a critical component of a professional software system would publicly acknowledge that it is *'_HIGHLY un-optimized.'* The source code itself, as well as the API documents, solicits contributions and involvement in developing the software. Secondly, the authors' email addresses are provided suggesting the possibility of responding to the source code itself. Again, making source code available for reading is linked to providing an address for responses arising from that reading. The Cocoon project keeps going only so long as it manages to enroll contributors who are prepared to read and amend the source code and other documents.

5. Conclusion: Many-eyed bugs: co-ordination amongst the team of readers and writers

Eric Raymond's notorious 'The Cathedral and the Bazaar' argues passionately that open source development substitutes a potentially huge crowd of people for the small number of expert debugging engineers found in conventional modes of software development:

> "No quiet, reverent cathedral-building here -- rather, the Linux community seemed to resemble a great babbling bazaar of differing agendas and approaches (aptly symbolized by the Linux archive sites, who'd take submissions from *anyone*) out of which a coherent and stable system could seemingly emerge only by a succession of miracles". (16)

This is a very commonly cited difference between professional software engineering and open source development. This contrast is framed in terms of the difference between the monumental 'cathedral' style of software construction associated with traditional software engineering and the buzzing hive of 'bazaar' style of activity out of which open source software emerges. What is less often emphasized is how this contrast could actually work. How has a great babbling bazaar of potential readers and writers of the source code been drawn together? This paper suggests that the 'bazaar' takes place along very specific lines and highly organized lines. It is an achievement that requires a good deal of communication, and the implementation of contrivances that afford certain kinds of reading and writing of source code.

In this paper we have shown that we are interested in moving beyond idealised versions of the OS project - as exemplified in the "Hacker Ethic' or 'Rebel Code' - towards understanding OS as a sociological phenomenon. Our analysis suggests we transcend the simplistic motivational or incredible economic approaches that characterise much of the debate on OSS. Instead we offer a praxiological understanding of open source - an understanding of the everyday practicalities of software projects. Standing outside of, such debates about motivation, allows us to concentrate on understanding exactly how and in what ways an open source project is accomplished as practical work in which participants pressures are usually egological - "what do I do next" - rather than motivational - "what's my motivation here". Such an approach sees OSS less in terms of a lifestyle choice, though this may well be individually important, but instead focuses on some of the practical ways in which a project is developed and sustained and 'codes of practice' are displayed, achieved and maintained as features of everyday work.

6. Acknowledgement

This work is funded by the EPSRC/ESRC Dependability Interdisciplinary Research Collaboration (DIRC).

7. References

- 1. The Open Source Initiative: Frequently Asked Questions http://www.opensource.org/advocacy/faq.html
- 2. Mockus, A., Fielding, R, and Herbsleb, J. (2000) 'A case Study of Open Source Software Development: The Apache Server.' in Proceedings of ICSE 2000, Limerick, Ireland.

- 3. Vixie, P. (1999) “Software Engineering” eds. Chris DiBona, Sam Ockman & Mark Stone, *Open Sources: Voices from the Open Source Revolution*, O’Reilly Books, 1st Edition
- 4. Sommerville, I. (2001) Software Engineering. London. Addison-Wesley.
- 5. Raymond, E. S. (1999), “The Revenge of the Hackers”, ” eds. Chris DiBona, Sam Ockman & Mark Stone, *Open Sources: Voices from the Open Source Revolution*, O’Reilly Books, 1st Edition January 1999
- 6. Button, G and Sharrock, W. (1996) “Project Work: The Organisation of Collaborative Design and Development in Software Engineering” *Computer Supported Cooperative Work: The Journal of Collaborative Computing* 5: pp 369-386.
- 7. Cocoon website http://xml.apache.org/cocoon/
- 8. Weider, D. (1974) Language and Social Reality. The Hague. Mouton.
- 9. Zimmerman, F-O. (1997) ‘Structural and managerial aspects of virtual enterprises’ from *Proceedings of the European Conference on Virtual Enterprises and Networked Solutions - New Perspectives on Management, Communication and Information Technology,* April 7-10 1997, Paderborn, Germany
- 10. Casey, C. (1995) Work, Self and Society After Industrialism. London. Routledge.
- 11. Yamauchi, Y., Yokozawa, M., Shinohara, T and Ishida, T. (2000) Collaboration with Lean Media: How Open Source Software Succeeds. In Proceedings of CSCW 2000. ACM Press
- 12. Anderson, R and Sharrock, W. (1993) 'Can Organisations Afford Knowledge?' in Computer Supported Cooperative Work. Vol. 1. No. 3. Pp 143-162.
- 13 Suchman, L. (1994) 'Working relations of technology production and use', in Computer Supported Cooperative Work. Vol, 2: pp 21-39.
- 14. Bowers, J., Button, G. and Sharrock, W. (1995) 'Workflow from within and without', in *Proceeings of ECSCW ’95*. Stockholm, Sweden, Kluwer Academic Publishers
- 15. Edwards, K. (2000) Towards a Theory for Understanding the Open Source Software Phenomenon. http://www.its.dtu.dk/ansat/ke/towards.pdf
- 16. Raymond, E. S. (2000) “The Cathedral and the Bazaar” http://tuxedo.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/

Appendix D: Software Architectures and Open Source Software – Where can Research Leverage the Most?

Budi Arief, Cristina Gacek, and Tony Lawrie
Centre for Software Reliability
Department of Computing Science
University of Newcastle
Newcastle upon Tyne NE1 7RU
United Kingdom
{L.B.Arief, Cristina.Gacek, A.T.Lawrie}@ncl.ac.uk

1 INTRODUCTION

Software architectures have been playing a central role in software engineering research for some years now. They are considered of pivotal importance in the success of complex software systems development. However, with the emergence of Open Source Software (OSS) development, a new opportunity for studying architectural issues arises. In this paper, we introduce accepted notions of software architectures (Section 2), discuss some of the known issues in OSS (Section 3), resulting in a set of aspects we consider to be relevant for future research (Section 4).

2 SOFTWARE ARCHITECTURES

Software Architecture can be defined as the structure(s) of a system, which comprise software components, the externally visible properties of those components and the relationships among them [1]. It typically acts as a bridge between software requirements and implementation. The architectural design of a software system can represent the most vital artefact for a software project, as it directly impacts upon the important management and technical processes of production and integration [2]. Hence, a sound software architecture is desirable in order to build a solid software system.

Some of the reasons why software architectures are believed to be important are that they: facilitate the communication among stakeholders, represent the manifestation of the earliest design decisions, and constitute a relatively small and understandable model of how a system is structured [1]. Garlan further elaborates six aspects of software development within which software architecture can play an important role: facilitating understanding by using high-level abstractions, supporting reuse at multiple levels of granularity, providing a partial blueprint for development by indicating the major components and dependencies among them, exposing the dimensions among which a system is expected to evolve, providing analysis opportunities at early stages of development, and for basic management support [3].

In order to fulfil their expected roles, software architectures should be modularised. This modularisation plays a triple role:

- It facilitates understanding by using high-level abstractions and reducing the complexity of the task at hand,
- It highlights areas where work can occur in a concurrent and distributed fashion, and
- It can also be used to determine the organizational structure that should be in place for developing the system being considered[1].

Based on its intrinsic characteristics, software architecture design becomes essential while developing a large complex software system. It should be the responsibility of a main architect (group) responsible for keeping *the vision* of the overall system [4]. Additionally, in order to support system evolution, while avoiding architecture erosion and drift [5], the software architecture must also be evolved accordingly, at times requiring some major restructuring.

The issues addressed in this section have been recognised within proprietary software development. In the next section, we explore how software architectures relate to OSS development.

[1] Conversely, it is important to note here that organizational structures have also been known to influence the creation of software architectures, by having the latter reflect the areas of expertise and availability of people in the former.

3 SOFTWARE ARCHITECTURES IN OPEN SOURCE SOFTWARE VS. PROPRIETARY SOFTWARE

With the emergence of Open Source Software (OSS) development [6, 7] as an alternative approach in building software systems, it is interesting to investigate whether software architecture still plays as prominent a role in the OSS development as in the traditional or proprietary software development.

At its root, the popular OSS definition makes the distinction between proprietary software development and OSS as being a centralised vs. decentralised software development argument, where the process of carefully controlling the construction of software is replaced by a rapid evolutionary process of voluntary submissions from all over the world [7]. Although a rapid, decentralised, and participative approach is not unique to OSS development, it does pose fundamental architectural considerations for the development of software systems.

However, OSS also presents a different attitude towards software development. By giving away the source code, OSS lets anyone inspect and modify the code as they please. There is also no explicit planning or project management in the open source approach, which puts a lot of strain on the architecture of the system.

Unlike most traditional software development, the original interest and vision in OSS projects usually emanates from the initiating projects owners [7]. Such individuals often assume complete authority [8]. Even in "shared-leadership" situations, such as the Apache web server, investigations have established that the core-developers still exercise the major influence over the design and direction of OSS development [9]. Consequently, in contrast to traditional software approaches, OSS project managers seem to possess greater power to determine the architectural direction of the software product. In this respect, even in the (supposedly) decentralised OSS process, the traditional architect role still appears to be a prerequisite for preserving the conceptual integrity of software [4]. However there are views expressed that OSS leaders may abuse this power to protect their own position by concealing the software architecture [10]. In doing so, they risk removing the blueprint that is vital for detailed understanding. Nevertheless, even in the absence of an explicit architectural blueprint, it may still be possible that the OSS development process can overcome the traditional software development barrier (c.f. Section 2) by narrowing the conceptual gap between requirements and implementation. The reasons being:

- Many of the users of OSS software are also contributing developers [11],
- Creating programs for oneself has long been considered less demanding than developing software for others [12],
- The rapid releases and early feedback allow a greater level of incremental development in the OSS process [7, 13].

Finally, initial OSS releases may be lacking in code refinement and contain many residual faults [7]. Nevertheless, it has been recognised that for OSS projects to be successfully initiated, evolved, and maintained, the architecture must be modularised to promote code comprehension and concurrent collaboration [14].

An indication of the importance of architectural coherence in OSS was provided by Eric Raymond's interpretation of why the controversial release of Netscape's Mozilla source-code did not fulfil initial expectations [7:p 77]:

> *"…going 'open' will not necessarily save an existing project that suffers from ill-defined goals or spaghetti code, or any of the software engineering's other chronic ills."*

It should also be noted that most OSS endeavours are undertaken in fairly stable domains. This characteristic can help explain why major architectural restructuring is hardly ever witnessed.

4 FUTURE DIRECTIONS

In recognition of both traditional and OSS architectural issues (see Sections 2 and 3), a number of points can be tentatively stated for future discussion and research:

1. Analyse how the OSS organisational structures affect their software architectures.
2. Investigate how OSS approach may harmonise the two extremes of the centralised and decentralised software development.
3. Compare how the gap between requirements and implementation is handled within OSS vs. proprietary software development.
4. Explore the role of software architectures in OSS development.
5. On the issue of software architecture decay:
   a) Assess whether architectural decay happens in OSS. If so, how quickly does it occur, and how is it dealt with?
   b) Try to get some insight on architectural drift and erosion by studying OSS projects that failed.
6. Leveraging the benefits offered by interdisciplinary research in order to determine:
   a) which OSS characteristics are suitable for adoption within other software development processes,
   b) what implications the OSS characteristics may have, for example on "time-to-market",
   c) the communication patterns in OSS, both in terms of communication mode and quality of content.

ACKNOWLEDGEMENT

This paper has been funded by the UK EPSRC project on Dependable Interdisciplinary Research Collaboration (DIRC – `http://www.dirc.org.uk/`). We would like to thank Denis Besnard, Michael Jackson, and Cliff Jones for various fruitful discussions contributing towards this paper.

REFERENCES

[1] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*: Addison-Wesley, 1998.

[2] W. Royce, *Software Project Management: A Unified Framework*: Addison Wesley, 1998.

[3] D. Garlan, *Software Architecture: a Roadmap*, in *The Future of Software Engineering*, A. Finkelstein, Ed.: ACM Press, pp. 93-101, 2000.

[4] F. P. Brooks, *The Mythical Man Month: Essays on Software Engineering*: Addison-Wesley, 1995.

[5] D. E. Perry and A. L. Wolf, “Foundations for the Study of Software Architecture,” *ACM Software Engineering Notes*, 4, vol. 17, pp. 40-52, October 1992.

[6] The Open Source Initiative online at *http://www.opensource.org*.

[7] E. S. Raymond, *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*: O’Reilly & Associates, 1999.

[8] M. Maclachlan, “Panelists Describe Open Source Dictatorships,” in *TechWeb News*, 12 August 1999, online at *http://www.techweb.com/*.

[9] A. Mockus, R. T. Fielding, and J. Herbsleb, “A Case Study of Open Source Software Development: The Apache Server,” presented at 22nd International Conference on Software Engineering, Limerick, Ireland, pp. 263-272, 2000.

[10] N. Bezroukov, “A Second Look at the Cathedral and the Bazaar,” in *First Monday*, 9 December 1999, online at *http://www.firstmonday.dk/issues/issue4_12/bezroukov/*.

[11] K. Johnson, “Towards a Descriptive Process for Open-Source Software Development,” Submission for the 2nd Workshop on software Engineering over the Internet, online at *http://www.cpsc.ucalgary.ca/~johnsonk/SENG/SENG691/towards.htm*.

[12] G. M. Weinberg, *The Psychology of Computer Programming*: Dorset House, 1971.

[13] R. C. Pavlicek, *Embracing Insanity: Open Source Software Development*: SAMS Publishing, 2000.

[14] T. Bollinger, R. Nelson, K. M. Self, and S. J. Turnbull, “Open-Source Methods: Peering Through the Clutter,” *IEEE Software*, July/August, pp. 8-11, 1999.

A probability model to aid the comparision of Open Source Software and Closed Source Software DRAFT

D. Bosio, B. Littlewood, M. J. Newby, L.Strigini
Centre for Software Reliability,
City University, London, England

December 5, 2001

Abstract

In this paper we discuss a theoretical model of reliability growth in relation to Open Source Software (OSS). We examine some of the conjectures/claims which have been made about OSS process with respect to their Closed Source Software (CSS) counterparts and translate them into statements about differences between the parameters of the model. In particular we study conjectures about the alleged fast reliability growth of OSS software after release. We present interesting, non intuitive initial results and discuss further possible developments.

1 Introduction

There seem to be a lot of different claims about the dependability of Open Source Software (OSS), some of them contradicting each other (OSS is generally better than Closed Source Software (CSS), or vice-versa), some of them presenting a challenge to intuition (OSS is more secure because of the accessibility of its source code to all, including would-be intruders). Far from proposing a universal way to verify all these claims we wish to show a way of studying statements about the dependability of OSS products and its supposed causes. We will give an example of how probabilistic models can be used for understanding the effects on software dependability of factors in the software production process. In this paper, we only use -as examples- models of a class which we developed earlier on to weigh claims about the merits of different testing methods [1, 2, 3].

Our aim is to shed some light on the possible contributing factors to the claimed greater reliability of OSS. The questions of practical interest we want to answer are of the form “ Does factor X in the development process tend to improve dependability measure Y” ?

Such modelling is a first step in giving some substance to imprecise statements like "the diversity between reviewers enjoyed by OSS processes (in greater degree than others) causes better reliability in OSS products", often used either to argue that OSS processes favour dependability or to explain the good dependability observed in some products of OSS processes.

Modeling forces us to explain what we mean by "diversity" and "greater degree of diversity", which results we wish to compare, etc. After being so specific, we can often check whether it is plausible that the invoked factor actually increases dependability, and under which additional conditions we should observe this effect [1].

The only alternative to modelling for supporting a claimed causal effect between aspects of the software production process and its achieved results would be appealing to bare statistical evidence of correlation between the two. This could prove to be prohibitively difficult. Checking empirically even a simpler statement like "OSS products are more reliable than the others" is difficult in practice, for various reasons: paucity of products with documented reliability, difficulty of choosing terms of comparison, difficulty — in the end — of claiming anything at all since we would expect all processes to exhibit great variability in their results.

To study the effects of software processes on dependability we have to specify which aspects of the achievement of dependability we wish to discuss. Here we concentrate on the detection of faults via execution of the software, in testing or in normal operation. We are initially interested in the often claimed fast improvement of OSS code after release.

2 The model

2.1 Description and basic assumptions

In this section we state explicitely all the assumptions and definitions we will use in the model.

An intuitive description of the fault-finding process goes as follows: there are well-identifiable defects ("bugs", "faults") in the code, which may cause the program to fail. When one observes a failure, the fault that caused it is identified and an attempt is made to remove it (which may succed or fail). More formally we use the following model, as previously described in [1, 2, 3].

First, for simplicity, we restrict ourselves to a demand-based model of programs execution. A program is given a demand, computes a result and

[1]Of course, it does not matter whether the factor that we choose to study (here, diversity of reviewers) is conjectured to *improve* dependability, or to make it worse: the modelling itself shows which effects that factor should produce, given the assumptions of the model, through which we describe the intuitive basis of the conjectured cause-effect relationship.

terminates. In other words, we characterise the "extent of exposure" of the program to failing as a discrete variable represented by the number T of executions of (i.e., demands applied to) the software. A demand is characterised by the values of all the input parameters and machine state (e.g. files) that determine the behaviour of the program in one execution. This model is very general applying e.g. even to interactive programs if we consider the sequence of all user inputs during a session.

A continuously operating program can be treated similarly, but using a continuous time variable.

The collection of all the possible demands is called the demand space. The demand space is partitioned in two sets, the failure set and the success set corresponding respectively to demands that will cause the system (software) to fail (failure points) or not to. We describe the failure set as composed of multiple, non-overlapping failure regions, each a collection of failure points corresponding to a specific defect in the code. If a failure point is found (through observing a failure), then either the failure region to which it belongs is completely eliminated (successful fix) or not at all. Thus, informally, the number of bugs in the code corresponds to the number of failure regions in the failure set.

There are many users with different profiles using the program and we want to establish the reliability growth they observe as faults are identified and possibly removed.

Users are characterised by the way they use the software, i.e., by their *usage profiles* [2], and by the probabilities of their reporting a bug when they observe it. This also determines the different probabilities of that user observing failures due to each bug in the software. The reliability observed by each user is strongly affected by that user's usage profile.

We also assume that the demands chosen on different executions (by the same or different users) are statistically independent.

Different users may use the software more or less frequently. This is modeled by the fact that, at any point in time at which we choose to analyse the software's reliability, each user may have applied a different number of demands (T_j for user j).

When an execution results in a failure, the user may or may not notice it and may or may not report it to someone who can fix it (the official maintainers for a closed-source product, the project community for an OSS product). Then someone may fix the bug that caused the failure, with a certain probability smaller than 1.

For this discussion, it does not matter wheter a user is using the program in his/her usual fashion, or is intentionally probing for faults ("directed testing"): the difference is modelled by different usage profiles and different

[2] A user's usage profile is the set of the probabilities of each possible demand being chosen by that user.

rates of bug reporting.

2.2 Description of the model

The parameters in the model are

- $q_{i,j}$ the probability of the fault i causing a failure for the user j on a randomly selected demand,
- $r_{i,j}$ the probability of a failure caused by the fault i being reported by user j when this user sees the fault,
- f_i the conditional probability of the fault i being fixed given it has been reported[3]
- T_j the number of demands applied by user j by the moment in time at which we study the achieved reliability of the software, and total number of demands by all users $T = \sum_k T_k$.

All the probabilities just described $q_{i,j}, r_{i,j}$ and f_i are considered constant over successive demands[4]. In other words, we are assuming that the number of users, and their behaviour in terms of software execution and bug reporting, are constant over time. This does not constrain the generality of the model. Indeed, we can describe users recruited later as users who were always present but perform no executions until a certain time. We can descride a change in a user's usage profile in terms of two virtual users, one of which stops executing the software when the other starts.

The meaning of the parameters can be translated into a real-life scenario by the following observations:

- we model better developers with smaller $\sum q_{i,j}$s (i.e., faults introduced by better developers have a smaller reliability impact or fewer bugs).
- better reporting corresponds to a higher value of the $r_{i,j}s$
- finally better fixing translates into higher f_is.
- larger $q_{i,j}$s correspond to users with higher probabilities of being affected by failures caused by fault i, thus users who are better at finding that fault: if these users are intentionally testing the software, they are better testers, and if they are using the software, they are the less lucky users.

[3]Note that we do not consider the consequence of a failure: the severity of a bug is only given in terms of the probability of selection in operation of a point from the failure region associated to it.

[4]Note the underlying simplifying assumption that the probability of a user reporting a fault does not depend on how many times that user has observed failures caused by that fault before.

This model represents the fact that the reliability improvement process is a stochastic process. The fixing of faults depends on when (and whether) they are found during execution, and their being reported, and the report prompting an action to fix the bug and the fix being effective. By effective we mean that the fault is removed. For simplicity we assume that an ineffective fix leaves things exactly as they were with the fault still present, i.e., we are excluding the cases of partial and of deleterious fixes. The model describes statistically how this process will evolve.

For instance, the initial reliability of the program as seen by user j corresponds to $T = 0$, i.e. after no executions of the software, and is described by its probability of failure on demand (pfd) pfd_j

$$pfd_j = \sum_{i \in \{failure\ regions\}} q_{i,j} \ . \tag{1}$$

We now consider the case where multiple users have executed the software, each user k having executed T_k demands. The probability of a fault having being removed is the probability of the fault having been reported (at least once) and fixed. The probability of the fault being reported at least once is $1 - P(\text{fault } i \text{ not reported})$. The probability of the fault not being reported by any user is given by

$$U_i = P(\text{fault } i \text{ not reported}) = \prod_{k \in users} (1 - r_{i,k} q_{i,k})^{T_k} \ , \tag{2}$$

hence, recalling that f_i is the probability of fixing bug i once it has been reported, we have that the probability of the fault i being removed is

$$P(\text{fault } i \text{ removed}) = f_i(1 - U_i) \ .$$

Each user will experience an improvement in reliability as a result of the faults fixed when revealed in this multi-user "testing" activity, but this growth will be different for different users (as, indeed, would be their initial perceived reliabilities before testing). The expected reliability as seen by user j as a result of the multi-user testing, after a total number of executions $T = \sum_k T_k$, depends on the usage profiles of the other users, in the following way

$$pfd_j = \sum_{i \in \{failure\ regions\}} q_{i,j} \left(1 - f_i(1 - U_i)\right) \ . \tag{3}$$

The associated expected increase in reliability as observed by the user j will be

$$I_j = \sum_{i \in \{failure\ regions\}} q_{i,j} f_i (1 - U_i) =$$

$$= \sum_{i \in \{failure\ regions\}} q_{i,j} f_i (1 - \prod_{k \in \{users\}} (1 - r_{i,k} q_{i,k})^{T_k})) \ . \tag{4}$$

All the following factors will decrease pfd_j (and hence increase I_j, the increase in reliability of the software for user j): adding more users, increasing the value of any of the $r_{i,k}$ or $q_{i,k}$, or f_i parameters (a mathematical justification of these assertions is given in the appendix). Let us discuss the real world significance of this in brief. A higher k corresponds to the intuitive notion that the bigger the community of users executing (and thus testing) the software, the greater the benefit that the community would have. An increase in $r_{i,k}$ also corresponds to an intuitive notion: if a person is more likely to report a bug, more bugs will be exposed, allowing for more bugs to be corrected. An increase in f_i represents the intuition that the more likely the bug is to be removed, the higher the resulting reliability of the software.

We note that if $r_{i,j} = 0$ then user j is a "free rider", that is to say a user for whom the reliability improves without reporting any bugs, and she/he benefits from bugs being reported by and fixed for others.

All these are somewhat unsurprising properties simply confirming that the model captures "common-sense" understanding of how people interact in fixing bugs. It is interesting to see how competing claims about OSS processes can be represented in this model. In a CSS process, one would expect a small community of *special* users, the in-house testers, who put in a big "lump" of executions early on in the life-cycle and after major changes. These users have very high $r_{i,j}$ and aim to have high $q_{i,j}$ as they often *try* to cause failures. Yet theory shows [1, 2, 3] that if their $q_{i,j}$ for certain faults are lower than for "ordinary" users, the latter will see much worse reliability than the testers. There is anecdotal evidence that this happens with many products. In an OSS process, this nucleus of heavy duty testers may well be smaller. On the other hand, it is plausible that the $r_{i,j}$ are often much higher than with CSS software, due to perceived higher chance of obtaining a fix; visibility of the source code and higher number of potential "fixers" should give higher $f_i s$ than for many commercial products (at least after some time from release).

2.3 Discussion of assumptions. (to be done)

- independence of executions with respect to different users
- independence of executions with respect to different executions by the same user.
- parameters constant in time i.e., no change over successive demands of f_i, $q_{i,k}$, $r_{i,k}$
- disjoint failure regions
- bijective mapping failure region $<->$ fix

3 Diversity is useful

In this section we will illustrate how models of this kind can be used to investigate the plausibility of quite general hypotheses, by trying to express them in the formalism of the model. The hypothesis we shall investigate here is the following: "Diversity is a good thing: all things being equal, it is better for testers to have diverse demand profiles than for them to have the same profile."

We want to compare two situations, one in which all the users have the same profile (we will see below which one) with a situation in which there are many diverse profiles, assuming that the total outlay of effort is the same. We take this to mean that the total number of executions of the software is the same in both cases: $\sum T_k = T$.

Let us now define our "ideal average equivalent user", where "average" refers to the effectiveness in fault reporting (per execution), measured by its effect on the potential reliability improvements if all faults reported are fixed. For each bug i, it is a user who has the average of all the considered profiles, weighted with their respective number of executions. Mathematically this corresponds to a user with associated parameters $r'_{i,j}$ and $q'_{i,j}$ such that

$$r'_{i,j}q'_{i,j} = \frac{\sum_{k\in\{users\}} T_k r_{i,k} q_{i,k}}{\sum_k T_k} \ . \tag{5}$$

Let us then consider a situation in which all the users have parameters $r'_{i,j}$ and $q'_{i,j}$ satisfying equation (5) for all js. The theorem of arithmetic and geometric means [4] now tells us that for each bug i we have that the probability of not reporting bug i is smaller in the case of diverse users than in the case of all users having this ideal average profile

$$\prod_{k\in\{users\}} (1 - r_{i,k}q_{i,k})^{T_k} < \left(1 - r'_{i,j}q'_{i,j}\right)^{\sum_k T_k} \ , \tag{6}$$

unless all the products $r_{i,k}q_{i,k}$ are equal, in which case equality occurs. Combining the effect over all the failure regions i, we obtain

$$\sum_i q_{i,j} f_i \left(1 - \prod_{k\in\{users\}} (1 - r_{i,k}q_{i,k})^{T_k}\right) > \sum_i q_{i,j} f_i \left(1 - \left(1 - r'_{i,j}q'_{i,j}\right)^{\sum_k T_k}\right) \ . \tag{7}$$

What does this tell us? Any user (with profile $r_{i,j}$, $q_{i,j}$) would prefer the previous exposure of the software to have been diverse rather than uniform, because diversity gives him higher reliability. There are two conditions here to represent "all things being equal" in our comparison of diverse-profile testing with uniform-profile testing. They are (i) that the same number of demands are executed in each case and, (ii) that the uniform profile is, in

an intuitively appealing way, the mean of the different profiles used in the diverse-profile testing, from the viewpoint of the expected number of reports per demand for each fault. Subject to all things being equal in this natural way, we have thus shown that diverse-profile testing is superior to uniform-profile testing, in the sense it can be expected to deliver greater reliability improvement.

4 Discussion

4.1 Further implications of the model — conjectures

We consider here a few questions of clear interest, and speculate about which answers the model would give to these questions. I.e., we formulate mathematical conjectures; if in the future we manage to prove them, they will gain the status of conjectures about the actual evolution of software reliability. We rely heavily on our understanding of similar models described in [1, 2, 3].

4.1.1 The individual user's viewpoint

In practice many studies in software reliability often refer to the *average* reliability over all users. In other words, it refers to predictions of the total number of failures observed by the whole population of users. Of course, if users have very diverse profiles, as is the case for many products, this still allows for some users to observe very poor reliability while the average is very good. In other words, just because a product is known to be very reliable on average, I cannot trust that it will be very reliable for me. The "diversity is a good thing" theorem is a first step: it refers to the reliability for a specific user, and thus it allows one to talk about distributions rather than averages.

In a practical situation in which no equivalence relation like (5) can be stated between alternative scenarios, what would the model predict for an individual user, or set of users, who benefit from the collective fault reporting and fixing effort ?

A simple (approximate) analogy with the previous work cited is that it is "as though" we had two users, user 1 whose viewpoint we are taking, and user 2 representing all the other users, and executing many more demands, $T_2 >> T_1$. We can compare two scenarios with equal total fault-detecting and reporting efficacies, i.e., two scenarios in which, if we tried to estimate the reliability of the program by looking at the rate of generation of fault reports, we would have identical estimates of the reliability of the program (averaged among all users).

Assume that user 1's profile is very different from all others. We can expect that after any amount of time pfd_1 will be better than if user 1 were

alone to report failures; yet much worse than if all users had the same profile as user 1. Yet the situation is probably more complex. The optimal profile from user 1's viewpoint, in terms of reliability after a certain amount of use, with the same number of bug reports being generated by the two users, is defined in [2] (eq. 28) and is a complex function of the total number of demands to date and the $q_{i,1}$s.

It will be interesting to characterise better the conditions under which a user could be aware of being too "special" to benefit greatly from diversity in the user community.

4.1.2 Evolution over time

All the results so far have been discussed in terms of reliability at a certain, arbitrary moment in the history of use of the program. All results contain parameters T_k, or their sum $T = \sum_k T_k$.

We are really interested in how the program's reliability evolves over time. We showed in [3] a phenomenon whereby testing with a profile similar to the usage profile yields better reliability growth in the short term, but in the long term different profiles, with some emphasis on the "less important" bugs, are more beneficial (i.e., in the long run, the "important" bugs will have been found and fixed no matter what; but the other ones are difficult to get rid of).

In our model we can therefore conjecture two main contributing factors to the reliability growth as observed by user j. In the short term, corresponding to initial rapid reliability growth, it is affected mostly by those users with similar profiles to j's own. In the long run, the profiles which differ from user j's will contribute more to improving reliability as seen by j.

Notice that [2] shows that the best profile is asymptotically, for T tending to infinity, one in which all faults have identical q_is, no matter how different from the q_ijs defined by user j's profile.

4.1.3 Predictability of results, dependability of process

We have so far referred to the average, or expected value of reliability measures. This acknowledges that reliability growth is a stochastic process: for instance, a fault with high q is likely to be discovered early on, but it may well (with low probability) go undetected for a long time. The probabilities of different histories of reliability growth are determined by our model parameters. The averages that we have been discussing are defined over all the possible histories of reliability growth. So, they are useful indicators, but they may be misleading as they hide the potential variation between different histories.

In reality, what matters in a project is the reliability growth history that

actually takes places. When I am stuck with an unreliable product, it does not matter much that, if I consider all other possible histories, the average of all the reliability levels that I *could* have obtained would be much better than the one I actually see. So, in project decisions the probability of "bad" reliability growth histories - thus, the probability of histories that are "much worse" than average - matters.

In [3] we studied probability distributions of reliability growth histories, accounting also for the fact that the initial set of faults is unknown. We could show some counterintuitive examples of how comparatively bad failure reporting rates, from usage with a user's own usage profile, would be better defences against the risk of very poor reliability growth than even much higher reporting rates based on someone else's, different profile.

We would like to explore how this translates into predictions, for a new prospective user, of the risk of very bad reliability growth after adopting a new product, and how the degree of "openness" of the process and diversity of the user base should affect them.

4.2 Further insights from the model

If diversity is "A Good Thing", it seems plausible that more diversity is better than less. We need to understand what "more" means before we can ask how OS and other approaches differ in this respect. In terms of the model, we need to understand the interplay between qs and rs (and these with fs). Some tentative conjectures:

- r_{ij} increases with the number of failures observed by user i due to fault i), so that the r_{ij}s and q_{ij}s are positively correlated - if you think the fault is a frequent one you will be more likely to report it;
- we have shown that diversity of rq is desirable - if we can show that "the more the better" here, it seems to imply that positive correlation between r and q is good, (negative correlation would smooth out variation in the product?). This is not mathematically obvious and the situation might be more complex.

4.3 Limits and possible extensions

This model does not want to be exhaustive: there are many aspects of OSS processes that we have not described. Some of these could be studied through extensions to the model, some require different methods. To give some examples:

- in OSS an individual user might make available to the other users a fix which would work only in his particular profile, which is not allowed

in CSS as this would be commercially unviable[5];

- this model does not directly describe conjectured factors like a "community effect" increasing the efficacy of fixes in a OSS environment, nor the modularity of the code which is usual in OSS and allegedly greater than in CSS;

- in general, having additional users reporting bugs is useful as it will increase the chances to improve the reliability in any case. However, in this model there is no explicit representation of any resource bottleneck, like scarcity of bug fixing staff, or simply delays due to the need to coordinate many fixes. As an example, consider the situation when bug fixing is a competing activity,i.e., bug fixers may need to abandon one bug for another, so that increasing the likelihood of user k reporting bug i implies a higher probability of fixing bug i, but at the expense of fixing bug l. Mathematically this can be described by saying that increasing $r_{i,k}$ increases f_i, which in turn decreases f_l, or assuming that the sum $f_i + f_l$ is constant.

- the model lacks any notion of consequence of a failure, or failure "severity", yet it seems likely that users will be influenced by this (as well as their perception of its frequency) in deciding whether or not to report it. It may well be that attitudes to failure severity are different between OSS and commercial developments.

5 Conclusions

This paper intends to illustrate the potential of simple probabilistic modelling for shedding some light on the plausibility and consistency of beliefs about the mechanisms that affect dependability, in the context of disputes about the merits of OSS processes. The simple model described has already yielded an interesting theorem about the advantages of a diverse user community, but there are many other implications we intend to study.

We aim to obtain both clarifications of what "one should believe" given some plausible assumptions, and thus about the consistency of various claims about the differences between OSS and CSS processes, and predictions that can be checked empirically to decide whether these models, however stylised, are useful approximations for important, dependability-related aspects of real, complex processes.

[5]This may sometimes happen in CSS when computer vendors issue a patch specific to their products, although this happens mainly for support of new hardware, for instance printers or video cards.

6 Appendix: derivatives

Let us consider the expression giving the increase in reliability

$$I_j = \sum_{i \in \{failure\ regions\}} q_{i,j} f_i (1 - \prod_{k \in \{profiles\}} (1 - r_{i,k} q_{i,k})^{T_k}) \ . \tag{8}$$

Its derivative with respect to f_i is

$$\frac{\partial I_i}{\partial f_i} = q_{i,j} (1 - \prod_{k \in \{users\}} (1 - r_{i,k} q_{i,k})^{T_k}) \ . \tag{9}$$

Its derivative with respect to $q_{i,m}$ with $m \neq j$ is given by

$$\frac{\partial I_i}{\partial q_{i,m}} = -q_{i,m} f_i \frac{\partial U_i}{\partial q_{i,m}} = q_{i,m} f_i \left(r_{i,m} T_m r_{i,k} q_{i,k})^{T_k} \frac{U_i}{(1 - r_{i,m} q_{i,m})} \right) \ . \tag{10}$$

The derivative with respect to $r_{i,m}$ is analogous to the derivative with respect to $q_{i,m}$ and is given by

$$\frac{\partial I_i}{\partial r_{i,m}} = -q_{i,m} f_i \frac{\partial U_i}{\partial r_{i,m}} = q_{i,m} f_i \left(q_{i,m} T_m \frac{U_i}{(1 - r_{i,m} q_{i,m})} \right) \ , \tag{11}$$

note that the following relation holds

$$\frac{\partial I_i}{\partial r_{i,m}} = \frac{q_{i,m}}{r_{i,m}} \frac{\partial I_i}{\partial q_{i,m}} \tag{12}$$

The derivative with respect to $q_{i,j}$ is given by

$$\frac{\partial I_i}{\partial q_{i,j}} = f_i (1 - U_i) + q_{i,j} f_i \left(r_{i,j} T_j \frac{U_i}{(1 - r_{i,j} q_{i,j})} \right) \ . \tag{13}$$

The derivatives are all positives for values of the variables between 0 and 1.

7 Appendix: theorem

In this appendix we describe the theorem of arithmetic and geometric means. Let us consider two set of non-negative numbers $a_1, a_2, \ldots, a_n$ and $p_1, p_2, \ldots, p_n$. We will call the ps weights. The weighted arithmetic means of the numbers as is defined as

$$A(a,p) = \frac{\sum_{k=1}^{n} p_k a_k}{\sum_{k=1}^{n} p_k} \ ,$$

whereas the weighted geometric means is defined as

$$G(a,p) = \left(\prod_{k=1}^{n} a_k^{p_k} \right)^{1/\sum_k p_k} \ .$$

The theorem ([4], p.17) says that $G(a,p) < A(a,p)$ unless all the as are equal. Raising both sides to the power $\sum_k p_k$ the theorem gives the equation

$$\prod_{k=1}^{n} a_k^{p_k} < \left(\frac{\sum_{k=1}^{n} p_k a_k}{\sum_{k=1}^{n} p_k} \right)^{\sum_k p_k} . \qquad (14)$$

Replacing a_k with $(1 - r_{i,k} q_{i,k})$ and the weights p_k with the number of executions T_k in equation (14) yields equation (6). Indeed, we find

$$\prod_{k=1}^{n} (1 - r_{i,k} q_{i,k})^{T_k} < \left(\frac{\sum_{k=1}^{n} T_k \, (1 - r_{i,k} q_{i,k})}{\sum_{k=1}^{n} T_k} \right)^{\sum_k T_k} .$$

In the right hand side term we can recognise our definition of the "ideal average user". Indeed, by the definition of "ideal average user"(5), we have

$$\left(\frac{\sum_{k=1}^{n} T_k \, (1 - r_{i,k} q_{i,k})}{\sum_{k=1}^{n} T_k} \right) = 1 - \frac{\sum_{k=1}^{n} T_k r_{i,k} q_{i,k}}{\sum_{k=1}^{n} T_k} = 1 - r_i q_i \ .$$

References

[1] P. Frankl, D. Hamlet, B. Littlewood, and L. Strigini. Choosing a testing method to deliver reliability. In *Proceedings 19th ICSE'97*, 1997.

[2] P. Frankl, D. Hamlet, B. Littlewood, and L. Strigini. Evaluating testing methods by delivered reliability. *IEEE TSE*, 1999.

[3] M. Pizza and L. Strigini. Comparing the effectiveness of testing methods in improving programs: the effect of variations in program quality. *ISSRE*, 1998.

[4] G. Hardy, J.E. Littlewood, and G. Polya. *Inequalities*. Cambridge University Press, 1952.