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

Adrian Mackenzie[1], Phillipe Rouchy[2] and Mark Rouncefield[1]

[1]Department of Computing, Lancaster University
[2]Blekinge Institute of Technology

Abstract:
This paper is concerned with understanding the character of open source (OSS) project work. 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 'hacker' projects. 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 to the project as a whole. The contrast we draw is not with idealised views on software projects but with ethnographic and ethnomethodological studies of software production that emphasise the project as a practical, ongoing achievement. The paper moves beyond idealised versions of the open source project towards understanding OSS as a practical sociological phenomenon.

· 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 Software Development (OSSD) 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 OSS 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 (2) for example, 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 OSS "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.

Background

1.Software engineering vs hacking

How different is open source software produced through hacking from the software produced by software engineers? Answering this question decisively is difficult. Therefore, most accounts actually stress differences in engineering process rather than differences in the software itself as a outcome of the process. 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".

Vixie bases his conclusions on a comparison between the formal textbook methods of software engineering (e.g. [Sommerville, 2001]) 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" [Raymond, 1999]

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 almost too novel to be 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.

2.Making things orderly

Following Button & Sharrock (1996), we suggest that the continuities and some important differences between OSS 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". (1996: 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." (Button & Sharrock, 1996, 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.

1.Method - studying Cocoon as geographically dispersed, fluctuating and open-ended project

Our research focuses on an OSS project, 'Cocoon,' which itself is related to the much larger and more well-known Apache OSS 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."(Cocoon website)

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 observations are 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 Wieder (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' (Zimmerman, 1997; Siebel 1998) 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' (Casey, 1995: 45). 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, 1996) - with OSS the position is more complicated. While the OSS 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.

2.Achieving the orderly character of OSS 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.

1.The website as an ordering device.

Quite clearly the Cocoon 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. 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.

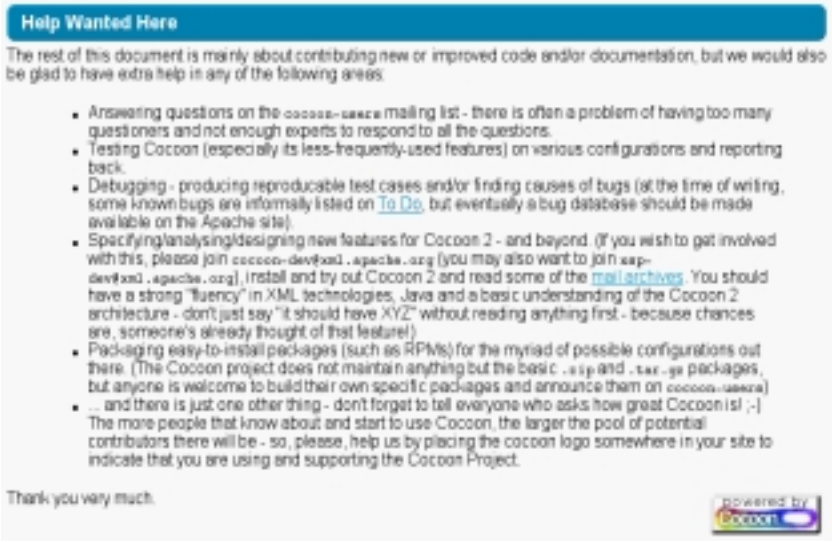

Fig 2.

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 (Anderson and Sharrock (1993). 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.

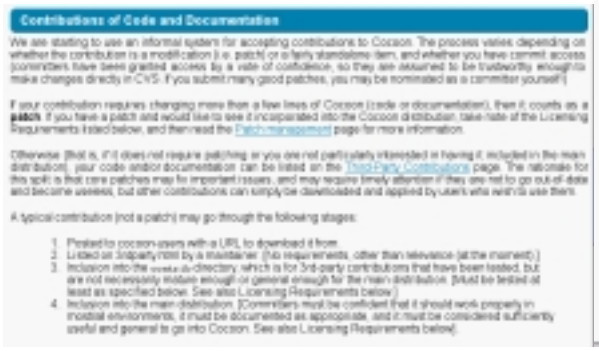

Fig 3.

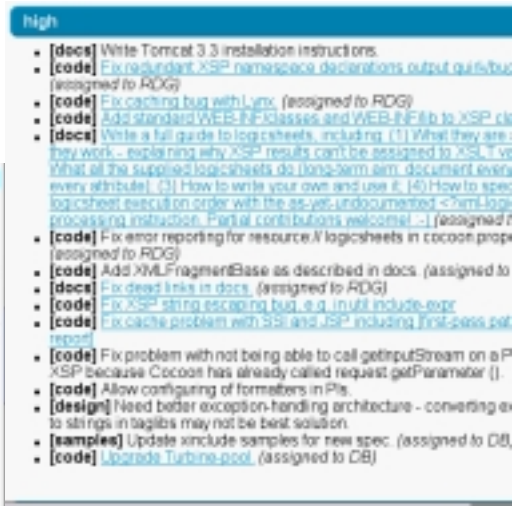

Fig 4.

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' (Suchman 1994; Bowers, Button and Sharrock 1995) 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.

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 bugfixes 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 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 alittle issue here. I voted -1 on the engine synch. patch since I thoughtthat 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.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 (2000) 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' open source. This is depicted in even more detail in accounts of the 'hacker ethic' and is used to provide some kind of explanatory account - 'why hackers do it'. In these approaches compliance to the 'code' explains behaviour. The open source community seen as governed by set of rules. Our argument is rather different since we are not interested in offering explanatory or motivational accounts of open source but of understanding how these projects 'get done'. We are interested in examining how the OSS community both construct and make use of the code 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 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 Wieder comments (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.' Wieder: 175.

3.Working the 'code'

4.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, yet in some ways it is also the most difficult kind of observational data that an ethnographic study has to deal with. The reasons for this are complex. While as ethnographers, we can read the source code for open source projects (something that is more difficult to negotiate in conventional industry software projects), by virtue of its formality and rule-governed nature, source code tends to hide the traces of its own development. Similarly, because open source projects are de-centralised and nearly always involve multiple sites, tracking the interactions between different kinds of reading and writing the code can be difficult.

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;
```

Some of the ordering devices present in this code are common in software engineering today. Some are somewhat specific to open source style projects. A combination rderings are present in this example. This particular code shows evidence of being ordered for at least two distinct kinds of readers.

Firstly, it affords reading by programmers and developers. Their ‘reading’ is associated with modifying the program. What is known to programmers as ‘code style’ - 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 – allows people reading the program on screen to 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 other people or 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 code affords processing by other programs such as compilers, document generators (e.g. javadoc will read certain lines), and configuration management systems (e.g. Concurrent Versioning System, CVS). Programming languages constitute large scale and complex ordering device for people to working with information systems. This is both a trivial and significant point. Programmers assume that 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. Programmers compile source code so frequently that it becomes just a routine habit. Yet source code is a formal representation (governed by rules operating on a 'vocabulary' of written characters) that can be directly processed by other programs *and* by programmers. The trivial point that programs are read/written by programmers and read by compilers covers the crucial point that the programmer and the compiler must share the same set of rules. (By contrast, other types of computer files such as an MPEG video file will only ever be 'read' by programs.) OSSD, as a collective activity, only makes sense because of this shared set of rules.

Do these ways of ordering source code indicate anything specific about OSSD? The formatting of the code, the use of Java, and the method of documenting the source (using javadoc) are all textbook academic 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.

Some minor differences show that this code belongs to an open source project, although even these are somewhat ambiguous data. 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 enroll contributors who are prepared to read and amend the source code and other documents.

In any case, the real significance of the source code lies elsewhere. The theme running through our study has been the clustering of ordering devices around OSS projects. Our argument has been that as ordering devices, none of them are entirely novel. It would very strange if the source code was an exception to this. On the contrary, perhaps we could regard the very familiarity and readability of the source code as an important affordance in OSS.

5.Where are the differences between OSS and professional software development?

One final ordering device confirms this point. Almost every open source software development project currently active, including Cocoon, 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 the product of an open source project (http://www.cvshome.org), the CVS program makes it possible forl arge number of people to access, 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." It is so crucial that certain institutions in the open source arena, such as SourceForge (www.sourceforge.net), a large website that hosts thousands of open source projects, can basically be interpreted as a public interface or 'frontend' to a vast CVS repository.

In the source code quoted above, revision numbers show on how many occasions a source code file has been modified. For instance, revision 1.12 implies that this file has been edited at least 12 times, although it may have been read many times before. The symbols shown in a line we have already cited are relevant here: * @version $Revision: 1.12 $ $Date: 2000/05/16 21:11:51 $. The $ characters are put there by another reading and writing device, the versioning system, in this case CVS.

Talk about the status of the CVS repository 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:

> On Wed, 6 Jun 2001 22:23:06 +0200 (CEST), giacomo <giacomo@apache.org> wrote:
>
> > > *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 reliably synchronize 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 about how to name the place where changed code is to be stored, the project would start to fragment. Rather than look for some OSS specificity in the code itself, we can regard the practices of open source as taking existing coding practices and configuration management techniques into a different form of organization.

Open Source and 'Epistemic Communities' - " IF ANYONE THINKS MY CODE SUCKS...TELL ME IT SUCKS...BUT TELL ME WHY".

This paper uses a study of the Cocoon Open Source project to explicate some of the ways in which Open Source projects are accomplished and how participants observably, reportably, accountably orient to the project as a whole. Of particular interest is an examination of how some sense of the Open Source 'community' manifests itself in and through the project. This requires moving beyond idealised versions of Open Source - as exemplified in the "Hacker Ethic" (Himanen 2001) or "Rebel Code" (Moody 2001) - towards understanding open source development as a sociological phenomenon. Our analysis suggests we transcend the simplistic motivational or economic approaches that characterise much of the debate and move toward a praxiological understanding of open source. Such an approach focuses on some of the practical ways in which a project is developed and sustained and 'codes of practice' are routinely displayed, achieved and maintained as features of everyday work.

In particular we are interested in the notion of the Open Source community as an 'epistemic community' (Edwards 2001). Edwards (2001) 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 their working. It is evident that the development and

orientation to some philosophy or notion of an open source 'code' regularly appears in Open Source discussions on 'good' or 'elegant' code, design philosophy and the principles of open source. This is depicted in detail in sociological accounts of the 'hacker ethic' where compliance to the 'code' is often used to provide some kind of explanatory account of behaviour. The Open Source community is seen as being governed by a publicly available and documented set of norms.

Edwards (2001) suggests that OSS can be characterised as an 'epistemic community', consisting of participants from various disciplines with various previous experience, but having "the following four characteristics:

- A shared set of normative and principled beliefs, providing a value based rationale for the social action of community members;
- Shared causal beliefs, which are derived from their analysis of practices leading or contributing to a central set of problems in their domain and serving as the basis for elucidating the multiple linkages between possible policy actions and desired outcomes;
- Shared notions of validity – that is, intersubjective, internally defined criteria for weighing and validating knowledge in the domain of their expertise;
- A common policy enterprise – that is, a set of common practices associated with a set of problems to which their competence is directed, presumably out of the conviction that human welfare will be enhanced as a consequence."

Outline of the 'code' that 'governs' OSS - depicted in even more detail in accounts of the 'hacker ethic' etc are used to provide some kind of explanatory account in that behavious is explained by compliance to the code. The OSS community seen as governed by set of normative rules and values. Analysis of the email archive facilitates an examination of some aspects of the OS 'epistemic community'. From numerous examples one will suffice - , a series of emails outline some aspects of the Open Source 'code' relating to what is 'good' code and 'ownership' of code within open source projects. As Edwards suggests this is a central value within the OSS community: " The open source software community in general (not just speaking for a single project) shares a strong disbelief in software patents and closed source software as mechanisms, which restrict the freedom of the users. Closed source software is perceived as barriers that hinder people's free choice and ability to improve software. It is very explicit in the community that sharing code is positive and that contributing code to a project is a way of getting status within the community."

The first email is a comment from a commiter relating what changes have been implemented:

Simplified Extract 1.

Modified: src/org/apache/cocoon Tag: xml-cocoon2 Notification.java
Notifier.java

Log:

removed that unportable (and useless) ASCII art along with (slow) system.out (logs \ are there for a reason) and cleanup the messy code

This evoked the following, hurt, (with appropriate emoticons) response:

Simplified Extract 2.
I'm sorry that you think my coding is messy, and I would prefer that you tell me first, being it my code, if it's not too much of a fuss >:-(
Anyway, why is it messy? Would you like it if I go round saying _your_ code is messy? Don't think you are perfect, we _all_ have to learn.
It is important that I get feedback on my code.
I think these remarks are _unfair_. :-(
The " unportable (and useless) ASCII art " was there to make errors more evident and not pass unnoticed. No problem if it annoys you, I don't care if you take it away, but it was not put there just for fun.
The "(slow) system.out" is rather important to me and to many other programmers IMO, and if it's slow who cares, it should not fire if all is ok. "(logs are there for a reason)"... yes, I KNOW, I already said I knew logging had to be kicked in, and the "(slow) system.out" is there just till C2 gets finished. Alpha 2a? Get real.

I would appreciate you all refer to the author of the code first before spreading _bullshit_ because:
1. nobody is perfect, this is for the users and the coders.
2. if you didn't write the code maybe you don't understand it. Some things that seem stupid to you have been thought of.
3. if the author doesn't get notified he cannot improve.
4. it's not nice to refer to other's people code as sloppy, you are not perfect.
5. if the code is not ok, who wrote it is the first one who has to make it ok.

I am astonished by this lack of sensibility.
I hope it will not happen again.
Nobody would like to contribute if this is done behind their backs.
If you don't trust me, tell me.
If you don't like my code tell me.
If you don't think I'm good enough, don't accept my work.

But please, don't make fun of me.

The response that followed contains a number of pointers as to the nature of the 'code' that governed the OS community:

Simplified Extract 3.
I was rather surprised to see a response like this to a simple code change.

>I'm sorry that you think my coding is messy, and I would prefer that you tell me first, being it '>my code, >if it's not too much of a fuss >:-(
I think it is important to recognize that we are working on an open-source project. I know 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 all are committed to making that code as good as possible. It's one of the strengths of open source.
>I would appreciate you all refer to the author of the code first before spreading _bullshit_ >because:
>1. nobody is perfect, this is for the users and the coders.
The users want high-quality code. The coders want to be able to change it. A "code ownership" attitude prevents both.
>3. if the author doesn't get notified he cannot improve.

> You have been notified. You got CVS commit mailings just like everyone else, and instead of taking the opportunity to improve or make inquiries into how to improve or why improvement was necessary, you took offense to it.
> >4. it's not nice to refer to other's people code as sloppy, you are not perfect.
> We could play a touchy-feely dance where we try and get the words right so we don't offend anyone at all, or we can call it like we see it. Personally, I'd rather know that someone thought my code was sloppy instead of never finding out because they don't know how to tell me without offending me. Since the issue has been brought up, I'd like to state the following to all Cocoon developers:
> IF ANYONE THINKS MY CODE SUCKS...TELL ME IT SUCKS...BUT TELL ME WHY.
> >5. if the code is not ok, who wrote it is the first one who has to make it
> ok.
> I completely disagree with this statement. Whoever wrote it, released it to an open-source project. Anyone can (and will) make it ok. I can't make this point strongly enough...there is NO PLACE for code-ownership on an open-source project.

The final email in the series attempted to summarize the lessons learned for the project as a whole. In so doing it indicates and reinforces some of the central working norms of the OS community.

> **Simplified Extract 6.**
> Now that testosteron storms are cleared, I would like you to know that Nicola and I talked a lot on the phone and he apologized for what he now considers a misbehavior.
> I would like to repeat to all of you what I told him:
> 1) nothing to apologize for: everyone of us makes mistakes, sometimes technical, sometimes human. I like people that make mistakes more than people that don't: the first have something to learn, they have a much more interesting life to live.
> 2) development is done by committing first and asking later. This is about release early and often since code is a thousands time more expressive than words. CVS is there *exactly* to know who did what, why and to be able to rollback at any time.
> 4) open source is about working under the eyes of hundreds of the best developers in the world. There is no place where the quality of you work is judged more strictly than in an open source environment, but code is never important, the community is much more.

In Edwards' view the notion of epistemic community and the normative values that bind them together are assumed to explain the whys and wherefores of the actions of the community. The norms, the 'code' is both a descriptive and an explanatory device, providing simple ways of interpreting actions. 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'. Drawing on Wieder's (1974) account of the 'convict code', we are interested in examining how the OS community construct and make use of 'the code' as a resource in the course of their mundane interactions. Here 'the code' is used as displays, or accounts of actions. There is, as Wieder (1974) suggests, another way of understanding the 'code' - not as somehow external to the setting but constitutive of some of the ways in which people act within the setting - ", part of the seamless fabric of action which make up the activities". In this view 'telling the code' - as outlined in the extracts above, is not simply reciting rules but sharing joint actions. This approach is not concerned with producing sociological explanation of behaviour in terms of compliance with code. Instead it is more interested in examining how OSS community both construct and make use of the code in the course of their interaction. The idea of some OSS code that is invoked in the various e-mail discussions is one that enables the OSS community to define and perform actions by making reference to the code. This includes

accounting for their own actions in terms of conformity to the code, and as such it is used by parties to the interaction as displays, accounts of what those actions are. The code can thereby be used for stopping a conversation, defending or defeating a proposed course of action and for accounting for one's actions in an acceptable way. As Wieder writes of the 'convict 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.' Wieder: (1974: 175). Thus the phrase " Once the code has been committed, it is no longer *your*code...it is *our* code" formulates what has just happened; provides an account of and a motive for what is said and attempts to direct the email 'conversation' along familiar lines - that there is no place for code-ownership issues in OSS.. As Heritage (1984) argues, this analysis, "vividly demonstrates that where sociological research encounters institutional domains in which values, rules or maxims of conduct are overtly invoked, the identification of these latter will not provide an explanatory terminus for the investigation. Rather their identification will constitute the first step of a study directed at discovering how they are perceivedly exemplified, used, appealed to and contested."

4.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". (Raymond, 2000)

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 centred on source code, but also circulating through emails, websites, configuration management systems and release/configuration documents.

In this paper we moved beyond idealised versions of the OSS project - as exemplified in the "Hacker Ethic' or 'Rebel Code' - towards understanding OSS 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 understanding of OSS in terms of the everyday practicalities of software projects. Standing outside 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.

5.Acknowledgement

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

6.References

- Anderson, R and Sharrock, W. (1993) 'Can Organisations Afford Knowledge?' in Computer Supposted Cooperative Work. Vol1. No. 3. Pp 143-162.
- 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,
- Edwards, K. (2001) 'Epistemic Communities, Situated Learning and Open Source Software Development' paper prepared for the 'Epistemic Cultures and the Practice of Interdisciplinarity' Trondheim. Avaliable at http://www.opencontent.org/openpub/
- Himanen, P. (2001) The Hacker Ethic and the Spirit of the Information Age. Random House.
- 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.
- Moody, G. (2001) Rebel Code: Linux and the open source revolution. London. Penguin.
- 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
- Raymond, E. S. (2000) "The Cathedral and the Bazaar迫 http://tuxedo.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/
- 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 January 1999
- Wieder, D.L. (1974) Language and Social Reality, The Hague, Mouton.,