The Linux Managing Model

By Federico Iannacci

Department of Information Systems, London School of Economics

Houghton Street, London WC2A 2AE

Tel. +44 (0)20 7955 7655; Fax +44 (0)20 7955 7385

Email: F.Iannacci@lse.ac.uk

Abstract:

This study focuses on the distinguishing traits of the Linux managing model. It introduces the concept of process to capture the idea of impermanence, dissolvability and change. Far from being a predictable flow of programming, assembling and releasing activities, it is suggested that the Linux development process displays a stream of activities that keep feeding back into each other, thus creating a complex and unpredictable outcome. The paper further introduces the concept of contingent response patterns to investigate the interaction flows occurring on the Linux mailing lists and subsume patch postings, bug reports and the associated reviewing and debugging activities under its umbrella. The enactment-selection-retention (ESR) model is subsequently brought forward to conceptualize this process as enactment of programming skills subject to selection activities conducted by Torvalds who retains the selected features and feeds them back to the developers' pool to undergo further enactment activities. Key managerial decisions concerning portability and modularity are, subsequently, analyzed through the lenses of the ESR model to show that Linux features an unconventional decision-making process whereby decisions follow rather than precede actions. Finally, Torvalds' beliefs are investigated in the Bitkeeper context to argue that the Linux managing model leans toward adaptability rather than adaptation.

Key words: process, change, retrospective sensemaking, strategy, adaptability;

Type of submission: Social.

1 Introduction: overview and research objectives

The use of metaphors in the open source landscape is predominant. Linux, to date one of the most successful open source projects, has been depicted as a babbling bazaar of different agendas and approaches to be contrasted to cathedral-like styles of software development (Raymond 2001).

This paper analyzes the Linux development process by conflating such apparently antithetical views on software development. It makes use of the evolutionary metaphor (Kuwabara 2000) to propose an alternative managing model: a model that thrives on intuition rather than forecasting, on opportunities rather than constraints and serendipity rather than plans. Time and again, the data collected will show that the Linux development process is not the outcome of planned design but of retrospective sensemaking, not the result of ordered change but of random almost chaotic enactments.

The reminder of this paper unfolds as follows: section two introduces the open source development process in general and the Linux development model in particular; section three highlights the Enactment-Selection-Retention (ESR) framework in use; section four attempts to suggest that adaptability is a better way of tackling random enactments than adaptation. Finally, section five draws some preliminary conclusions concerning this alternative managing model.

2 Open source development and Linux development

Traditional software development processes are based upon a four-stage cycle involving: a) Planning; b) Analysis; c) Design; d) Implementation. The open source development process, on the contrary, does not incorporate the first three stages being entirely premised on implementation (Feller and Fitzgerald 2002).

Although several scholars have investigated the open source development process, Feller and Fitzgerald (2002) claim that Jørgensen's (2001) model appears to be the most detailed. By analyzing the FreeBSD project, Jørgensen (2001) has identified six fundamental steps, namely: a) Code; b) Review; c) Pre-commit test; d) Development release; e) Parallel debugging; f) Production release.

According to Jørgensen (2001), the most straightforward aspect of the code stage is represented by the astonishing rate of software developers contributing high quality code. Parallel review of such code constitutes the next stage. Paradoxically, even though more complex code would benefit from peer reviewing, it turns out that the simpler the code is, the more feedback contributors obtain; this is an outcome that is consistent with our findings considering, for instance, the following comments on the Linux kernel mailing list:

> [] Start small, because for small patches people will have the few minutes needed to teach you. The bigger a patch, the harder it is to review it, and the less likely it happens []

[1].

Pre-commit test follows next. The FreeBSD project differs from Linux insofar as commit privileges are concerned. Whereas Linux exemplifies a case where just one individual is in control of the main repository, the FreeBSD project epitomizes a case where as many as 200 developers are delegated the right to commit (i.e. check in) changes into the main tree (i.e. the repository). This, in turn, is due to the fact that Linus Torvalds does not make use of the Concurrent Versions System (CVS) to maintain the official kernel tree. What is CVS? CVS is a version control tool that allows developers to work concurrently on the same repository (Fogel and Bar 2001). Basically, it grants developers the ability to download or check out code from the main tree and, subsequently, modify it. When the developer is satisfied with the changes, he can commit (i.e. check in) the changes back into the main tree. This model of software development is defined as copy-modify-merge because developers download their

[1] Source: http://www.uwsg.iu.edu/hypermail/linux/kernel/0201.3/1146.html

working copy from the CVS tree, modify it and commit it back into the CVS tree. Since CVS as opposed to other tools, namely BitKeeper (BK)[2], does not allow developers to remove committed patches unless a reversed-patch operation is performed, the FreeBSD project requires developers to perform pre-commit testing before committing their changes to the main tree; unfortunately, such reversed operations cannot be performed when it comes to extracting lines of code out of the middle of the source code which have already been built upon for more code[3].

Development release follows next. The FreeBSD project as much as Linux features two distinct releases, namely development and production release. The former refers to experimental releases, the latter to more stable releases. Experimental releases are, normally, odd-numbered releases; stable releases are even-numbered ones. In the FreeBSD project, developers committing changes to the main tree are indeed performing the act of releasing new code, since a new release is the effect of a commit to the development branch (Jørgensen 2001). In the Linux environment, otherwise, developers are not allowed to release experimental versions since Torvalds is in charge of the development branch. Linux, moreover, features the following release policy: the first integer represents a major release; the second one denotes a minor release, whereas the third indicates the patch level of the release. Unstable releases are indicated by odd minor release numbers, while stable releases are denoted by even minor release numbers (Erenkrantz 2003).

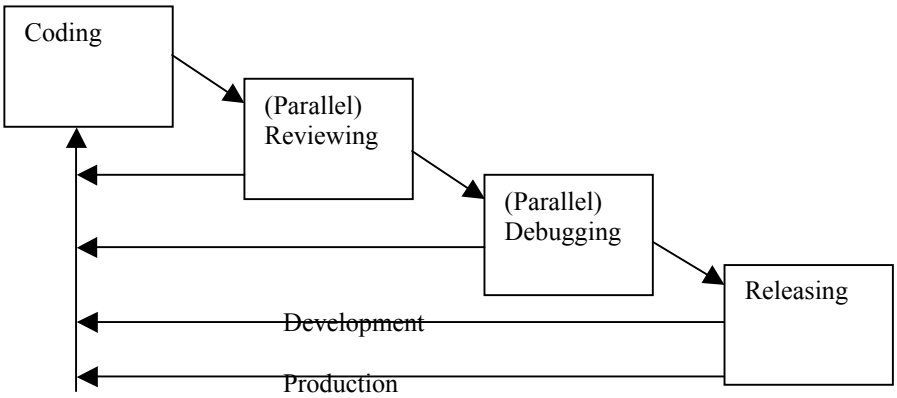
Parallel debugging constitutes the next stage. Whereas parallel development refers to the fact that developers are working in parallel on different versions of the source code and/or different reusable software components (i.e. modules), parallel debugging assumes that the larger the number of developers working on the same release, the more bugs (i.e. software misbehaviors) turn shallow (Raymond 2001).

[2] Source: http://www.ussg.iu.edu/hypermail/linux/kernel/0201.3/1987.html
[3] Source: http://www.ussg.iu.edu/hypermail/linux/kernel/0112.3/0474.html

The final stage, production release, is achieved whenever bug fixes previously submitted to the development tree are merged into the latest stable production branch. This step is performed in different ways in different projects. The FreeBSD project, for instance, makes use of a core team's release engineer who announces code freezes both in the production and the development stages during which all changes are rejected other than severe bug fixes. Linux, otherwise, sports feature and code freezes preceding forthcoming stable releases. A feature freeze announces that all changes will be rejected and that only bug fixes will be accepted; a code freeze is stricter because it only accepts severe bug fixes. Additionally, production releases encompass coding, reviewing and debugging operations as much as development releases, thus requiring managing activities in their own right.

Based on Jørgensen's (2001) open source development process, we have adapted his model to suit the Linux kernel development process, thus identifying the following stages:

Figure 1: The Linux Development Process



Notice that the use of gerunds instead of nouns is not accidental; we intend to pay as much attention as possible to the word 'process' because we want to convey an idea of impermanence, dissolvability and (disordered) change. The process imagery evokes the idea of flow, flux and stream. It displays a flow of random programming and nonrandom assembling and releasing activities that keep feeding back into each other, thus creating a complex and unpredictable outcome. These ideas, however, require a new organizational

metaphor; a model where individuals act on intuition instead of following orders from above; a model that values reactive strategies rather than proactive plans and admits novel actions rather than pre-defined uses.

Ultimately, the above remarks raise a penetrating question: how does Torvalds cope with random enactments that trigger disordered change? The next sections will draw on Weick's (1979) work to argue that the Linux managing model resorts to adaptability rather than adaptation to cope with random enactments.

3 Open source, mutual equivalence structure and the ESR model

The open source model portrays a new way of developing software products whereby the source code is made available on-line to anyone to view, modify and distribute. In his much celebrated work, Raymond (2001) argues that the open source model epitomizes a gift economy where developers engage in gift exchanges to maximize their own reputation.

Such exchanges have been further labeled as network, generalized exchanges where the contributor is not necessarily reciprocated by the recipient of his contribution but by some other actor(s) in the network (Yamagishi and Cook 1993). To further clarify this point, consider a situation where a programmer volunteers an answer to a tricky programming question posted on the mailing list. He may have no expectation of being helped in return by the recipient of his contribution. However, he may expect to receive help from some other member(s) of the network who, in turn, might be reciprocated by yet some other actor(s) (Kollock 1999).

Network, generalized exchanges can be viewed as contingent response patterns encompassing cycles consisting of double interacts. Consider, for instance, a bug report. Developer A posts a bug report on the kernel mailing list in order to get the bug fixed. Developer B provides the

bug fix and is, in turn, reciprocated by developer C when he posts his own bug report. Posting the bug report is instrumental to obtaining a bug fix and, subsequently, enjoying a non-misbehaving program. Thus, we can distinguish two different types of activities: instrumental and consummatory activities. The former always precede the latter by providing the conditions under which the pleasurable consummatory activities can occur. We define each cycle consisting of instrumental and consummatory activities as a double interact to highlight the fact that there are two interlocked behaviors, namely an instrumental activity (i.e. the bug posting activity) being interlocked into an original activity (i.e. the scrutinizing activity) and a consummatory activity (i.e. enjoying a non-misbehaving program) being contingent onto the bug fixing activity. Throughout this work, these contingent response patterns will represent our unit of analysis (Weick 1979).

Besides enacting their programming skills by posting bug reports, developers can also post new features (i.e. patches) on-line to obtain feedback from other developers. Whereas in such circumstances developers are not bracketing areas of the source code to further scrutinize them, they are, in a way, producing ecological changes that translate into new programming features. Moreover, the aforementioned logic of double interacts holds once again to the extent that posting a certain feature is an instrumental activity to actually obtaining a patch review.

In sum, enactment activities consist of cycles of double interacts and may imply novel uses of the software program. In addition, such activities create a mutual equivalence structure based upon mutual prediction not mutual sharing. This, in turn, means that each developer's ability to engage in consummatory activities (i.e. obtaining a patch review and/or a bug fix) depends on the behavior of others (i.e. someone posting a patch review and/or a bug fix) which, in turn, is contingent on the former developer's willingness to actually enact a preliminary coding activity and post its outcome on a mailing list. Furthermore, the developer's performance of his patch reviewing and/or bug fixing acts has the function of eliciting the

other(s)' patch reviewing and/or bug fixing activities. And to preserve this structure, developers need to repeatedly perform such patch reviewing and/or bug fixing acts.

Enactment is followed by selection. Influential open source members need, at this stage, to select a subset of cycles out of the pool of all possible cycles. Selection always works backward and is behind because the only thing which can be selected is an enacted environment that is already there. Selection can be viewed as applying a certain set of rules to assemble a subset of cycles out of the poll of all possible cycles. Such rules, to be labeled as assembly rules, are procedures, instructions or guides being used by influential organizational members to create the process. Torvalds and his army of trusted lieutenants, for instance, seem to apply criteria of relevance, enhancement and success.

Relevance refers to the fact that Torvalds and his lieutenants are used to selecting patches (i.e. feedback loops) against a specific development and/or production tree. Enhancement, otherwise, refers to the fact that Torvalds and the other official tree maintainers automatically accept bug fixes (i.e. interacts) that enhance their latest releases. While relevance and enhancement seem to apply both to production and development releases, success only refers to development trees and points to the fact that patches need to be used by a great number of users not developers as the LKCD [Linux Kernel Crash Dump] example shows:

> "[] I will merge it [i.e. the LKCD patch set] when there are real users who want it - usually as a result of having gotten used to it through a vendor who supports it. (And by "support" I do not mean "maintain the patches", but "actively uses it" to work out the users problems or whatever). Horse before the cart and all that thing.
>
> People have to realize that my kernel is not for random new features. The stuff I consider important are things that people use on their own, or stuff that is the base for other work. Quite often I want vendors to merge

> patches _they_ care about long long before I will merge them (examples of this are quite common, things like reiserfs and ext3 etc).
>
> THAT is what I mean by vendor-driven. If vendors decide they really want the patches, and I actually start seeing noises on linux-kernel or getting requests for it being merged from _users_ rather than developers, then that means that the vendor is on to something"[4].

It is worth noting that, broadly speaking, selection is not the outcome of planned design but of retrospective sensemaking whereby Torvalds is attempting to discover a world on which he has already imposed what he believes. This, in turn, requires an organizing model that corroborates the idea of acting before thinking where strategy comes after development not before it. Linux kernel developers, in other words, develop whatever feature suits their interests and needs and Torvalds, subsequently, decides whether to select such features.

Retention represents the last stage of the ESR model. The selected feedback loops and the associated enhancements are stored together with the memory rules used to assemble these cycles. To be sure, patches may always be removed from the official trees but removal rarely happens:

> "[] If you really believed the stuff you say you'd put it [i.e. the LKCD patch set] in and promise to take it out if people didn't find it useful or there were inherent limitations []"[5].

Bill Davidsen commented on Linus Torvalds' refusal to apply the LKCD patch set into his tree. And Torvalds voiced back:

[4] Source: http://www.ussg.iu.edu/hypermail/linux/kernel/0210.3/2062.html
[5] Source: http://www.ussg.iu.edu/hypermail/linux/kernel/0211.0/0011.html

> "This never works. Be honest. Nobody takes out features, they are stuck once they get in. Which is exactly why my job is to say "no", and why there is no "accepted unless proven bad []"[6].

Once retained, development and production branches are fed back to the developers' pool to undergo further enactment activities. The assembly rules, otherwise, affect the selection process by framing the criteria whereby cycles are to be chosen and, subsequently, inputted to retention. Thus, each stage of the ESR model consists of a flow of inputs and outputs (i.e. a process); moreover, the assembly rules are to be intended as the criteria by which the development and production processes are constructed.

Even though conceptually simple, the ESR model portrays the pivotal role that selection plays in accommodating the antithetical pressures stemming from enactment and retention. Whereas enactment spurs ecological changes that make development and production releases in tune with fleeting environments, retention lags behind these changes because it hinges on stored memory rules that suit past environments. This, in turn, implies that although enactment stimulates adaptability, retention propels adaptation. The purpose of the following section is to show that the Linux managing model in general and its selection process in particular lean toward adaptability rather than adaptation.

4 The Linux managing model

This section attempts to apply the ESR model to the Linux case study in order to highlight the dynamics of its managing style. By using a qualitative content analysis, this section focuses on analyzing data extrapolated from the Linux mailing lists. It is worth reminding the reader that such a case study was selected because, even though it is vastly studied in the literature, we felt that its managing issues have not yet been modeled after a dynamic framework. To be

[6] Source: http://www.ussg.iu.edu/hypermail/linux/kernel/0211.0/0014.html

sure, Kuwabara (2000) has already resorted to the evolutionary metaphor to account for the dynamics of the Linux operating system as a whole. The present work, on the contrary, elaborates on Torvalds' beliefs to argue that they epitomize a case where random enactments are retrospectively being dealt with adaptability rather than adaptation. Moreover, Kuwabara's (2000) study is focused on mimetics rather than contingent response patterns, on incremental changes rather than deviation-amplifying feedback loops, on discarding the military metaphor rather than complementing it and, subsequently, on self-organization rather than top-down strategizing. Thus, in our view, Kuwabara (2000) fails to appreciate the essence of our argument, namely that strategy in Linux comes after development not before it and consists of adaptability not adaptation. After all, it is as though the Linux managing style reifies a model of acting before thinking where meaning is imposed after the fact, not before it.

In several occasions Torvalds has engaged in discussions concerning the Linux development philosophy. Let us look at the following excerpt concerning the design of Linux:

> [] Let's just be honest, and admit that it wasn't designed. Sure, there's design too - the design of UNIX made a scaffolding for the system, and more importantly it made it easier for people to communicate because people had a mental _model_ for what the system was like, which means that it's much easier to discuss changes. But that's like saying that you know that you're going to build a car with four wheels and headlights - it's true, but the real bitch is in the details. And I know better than most that what I envisioned 10 years ago has _nothing_ in common with what Linux is today. There was certainly no premeditated design there. And I will claim that nobody else "designed" Linux any more than I did, and I doubt I'll have many people disagreeing. It

> grew. It grew with a lot of mutations - and because the mutations were less than random, they were faster and more directed than alpha-particles in DNA []7.

Here Torvalds is arguing that Linux did not grow out of a pre-defined design. This argument is further corroborated by the fact that modularity itself was not on Torvalds' mind at the beginning. The Linux kernel is a monolithic kernel. With a monolithic kernel, memory is divided into user space and kernel space. Whereas kernel space refers to the space where kernel code is loaded and memory is allocated for kernel level operations, user space includes those processes that are separated from the kernel and need to communicate with the kernel through message passing. Kernel operations include scheduling, process management, signaling, device Input/Output, paging and swapping. These are the core operations that other programs need to rely on to execute. Because such core operations are not abstracted from kernel space, monolithic kernels are specific to the particular hardware architecture they are interacting with. The alternative is a microkernel-based system where most of the operating system runs as separate processes, mostly outside of the kernel. In other words, a microkernel performs a much smaller set of operations in more limited form in kernel space. Such operations include: interprocess communication, limited process management, scheduling and low level input/output. Microkernels appear to be less hardware specific because basic operations are pushed into user space in order to abstract the details of process control, memory allocation and resource allocation and, therefore, allow for ports to other hardware architectures. At the time Torvalds started work on Linux, researchers believed that microkernels were superior to monolithic kernels because they could be ported and modularized more easily. Their argument was that, due to their intrinsic design, microkernels' basic functions could be transformed into modules and, thus, loaded and unloaded as needed. Monolithic kernels, otherwise, could not be modularized as easily because, according to their theory, it was more difficult to remove functions from kernel space and, subsequently,

[7] Source: http://kt.zork.net/kernel-traffic/kt20011217_146.html#1

transform them into loadable modules. Yet in 1995 Torvalds released version 1.2 and announced loadable kernel modules:

> "Helsinki, Finalnds—March 8, 1995 –Linus Torvalds announced the release of version 1.2 of the popular Linux operating system. [] Linux is a full-featured Unix operating system for PCs (386 or higher) which supports true multitasking, 32-bit virtual memory, shared libraries and executables, demand paging, advanced memory management, dynamically linked libraries and TCP/IP networking. Linux v1.2 adds to this list: performance enhancements, loadable kernel modules, increased portability support for many more peripherals, PCI support, PCMCIA support, EIDE support, an increasing variety of network protocols, support for ELF-format binaries, and more []"[8]

To be sure, this decision did not happen in a vacuum. A year before Peter MacDonald announced loadable kernel module patches against Linux 99p14 f:

> "This is to announce patches against Linux 99p14f that convert nearly all Linux devices and facilities into loadable modules []"[9]

Other kernel developers, however, started this conversion to modularization in cloned Linux trees including Bas Laarhoven, who is currently listed in the credits file for contributing loadable modules and ftape drivers, Jon Tombs, Jacques Gelinas, Jeremy Fitzhardinge and Björn Ekwall (Welsh 1995). Bas Laarhoven, in particular, while working on the ftape driver releases, developed a system call that allows the root to load/unload code[10]. Thus, Torvalds decided to introduce loadable kernel modules in his tree to make sure that developers from around the world could work in parallel and add changes and enhancements at run time without frequent kernel builds and reboots (deGoyeneche and deSousa 1999). This, in turn,

---

[8] Source: http://groups.google.com/groups?q =Linus+Torvalds+Loadable+Kernel +Modules&hl=en&lr=&ie=UTF-8&selm=3kkdi5%243m0%40kruuna.helsinki.fi&rnum=4

[9] Source: http://groups.google.com/groups?q=Linux+loadable+kernel+modules&hl=en&lr=&ie=UTF-8&selm=2hhgek%243rm%40klaava.Helsinki.FI&rnum=1

[10] Source: http://groups.google.com/groups?q=modules+Jon+Tombs&hl=en&lr=&ie=UTF-8&selm=1993Dec2.180207.8705%40lucrece.robots.ox.ac.uk&rnum=1

shows that Linux exemplifies an unconventional decision-making process where the implementation of an explicit structure for creating loadable modules occurred after the development of a set of features that allows to load/unload code.

Even porting the kernel to other platforms was not on Torvalds' mind at the beginning. The kernel underwent an unplanned major re-write in its early stages because of the developers' effort to port it to other platforms other than the x86 architecture. Thus, Torvalds decided to re-write the kernel code to make sure that it could be ported to other platforms in a common code base (Torvalds 1999). To reiterate, this proves that the Linux kernel is the outcome of an unconventional decision-making process whereby managing consists of making sense of enactments that are already there. Torvalds, in other words, lets developers follow their personal interests and needs. Once their enactments are in place, he looks back to impose his own beliefs.

This reasoning, in turn, begs the question: what are Torvalds' beliefs? The absence of design and up-front planning implies no goal setting and, therefore, lack of general direction at a macro level. Hence, the kernel development boils down to the developers' personal interests and needs and, to the extent that it ensures readiness for change, favors adaptability over adaptation:

> [] The impressive part is that Linux development could _look_ to anybody like it is that organized. Yes, people read literature too, but that tends to be quite spotty. t's done mainly for details like TCP congestion control timeouts etc - they are _important_ details, but at the same time we're talking about a few hundred lines out of 20 _million_. And no, I'm no tclaiming that the rest is "random". But I _am_ claiming that there is no common goal, and that most development ends up being done for fairly random reasons - one persons particular interest or similar.

It's "directed mutation" on a microscopic level, but there is very little macroscopic direction. There are lots of individuals with some generic feeling about where they want to take the system (and I'm obviously one of them), but in the end we're all a bunch of people with not very good vision. And that is GOOD. A strong vision and a sure hand sound like good things on paper. It's just that I have never _ever_ met a technical person (including me) whom I would trust to know what is really the right thing to do in the long run. Too strong a strong vision can kill you - you'll walk right over the edge, firm in the knowledge of the path in front of you. I'd much rather have "brownian motion", where a lot of microscopic directed improvements end up pushing the system slowly in a direction that none of the individual developers really had the vision to see on their own. And I'm a firm believer that in order for this to work _well_, you have to have a development group that is fairly strange and random.

To get back to the original claim - where Larry idolizes the Sun engineering team for their singlemindedness and strict control - and the claim that Linux seems ot get better "by luck": I really believe this is important. The problem with "single-mindedness and strict control" (or"design") is that it sure gets you from point A to point B in a much straighter line, and with less expenditure of energy, but how the HELL are you going to consistently know where you actually want to end up? It's not like we know that B is our final destination.

In fact, most developers don't know even what the right _intermediate_ destinations are, much less the final one. And having somebody who shows you the "one true path" may be very nice for getting a project done, but I have this strong belief that while the "one true path" sometimes ends up being the right one (and with an intelligent leader it may _mostly_ be the right one), every once in a while it's definitely the wrong thing to do. And if you only walk in single file, and in the same direction, you only need to make one mistake to die.

> In contrast, if you walk in all directions at once, and kind of feel your way around, you may not get to the point you _thought_ you wanted, but you never make really bad mistakes, because you always ended up having to satisfy a lot of _different_ opinions. You get a more balanced system []11.

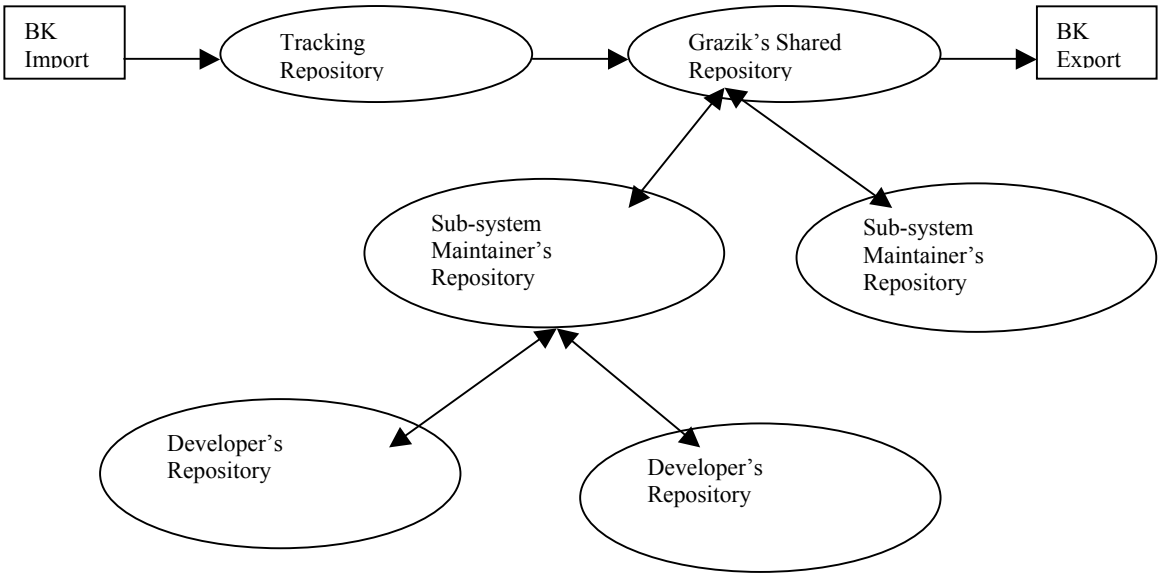
Adaptability, in turn, calls for an organizing system that rotates around minimal disruption of the underlying enactment processes as the BK vs. CVS example shows. On January 28th, 2002, Rob Landley started a very intriguing thread entitled "A modest proposal: we need a patch penguin"[12]. Notice the self-deprecation norm at work, if anyone needs be reminded of the role norms play in the open source development. Since Torvalds was dropping way too many patches, Landley's proposed to introduce a further layer of authority between Torvalds and his closest lieutenants to collect such patches. Torvalds' reply, though, was rather negative suggesting a "network of people approach" by using BK rather than CVS. What is BK and why does it stimulate a "network of people approach"? Most importantly, why, in this context, is a network of interpersonal bonds a better organizing strategy than introducing a further layer of authority?

BK is a distributed, replicated source code management system that Landley has defined as "a technical tool attempting to deal with a social problem"[13]. BK unlike CVS grants developers write access to their own repository not the main repository, thus avoiding a situation where the official tree becomes a chaotic mess of stuff that has not yet been filtered (Shaikh and Cornford 2003). In other words, instead of allowing automatic acceptance of patches, BK encourages developers not to dump their patches on other people's trees and act as real maintainers of their own trees[14]. This, in turn, creates a network of maintainers, each of whom knows other maintainers[15], that is characterized by a push/pull workflow from/to the shared repository. BK, in other words, features a tracking repository and a shared repository:

[11] Source: http://www.uwsg.iu.edu/hypermail/linux/kernel/0112.0/0004.html
[12] Source: http://www.uwsg.iu.edu/hypermail/linux/kernel/0201.3/1000.html
[13] Source: http://www.ussg.iu.edu/hypermail/linux/kernel/0201.3/1072.html
[14] Source: http://www.uwsg.iu.edu/hypermail/linux/kernel/0112.3/0474.html
[15] Source: http://www.ussg.iu.edu/hypermail/linux/kernel/0201.3/1087.html

whereas the former keeps track of external projects such as, for instance, an open source project, the latter combines both external and localized work[16]. Suppose a maintainer, say Jeff Garzik, imports open source patches into the tracking repository and pulls them into the shared repository. At this stage, he can give commit privileges to a handful of local sub-system maintainers he trusts. These, in turn, will clone the shared repository into personal repositories, do their work and, subsequently, push their changes back into the shared repository[17]. Local changes submitted to the shared repository can, subsequently, be pushed to external projects as traditional patches by using the BK export command. In addition, these sub-system maintainers may trust another handful of local developers to commit changes into their own repositories. These developers, once again, will clone the sub-system maintainers' shared repositories to do their work in their personal trees and will subsequently submit their changes to them who, in turn, will submit them to Garzik provided that such changes have been discussed on the mailing list(s) and have met their judgment criteria. This push/pull workflow is depicted below:

Figure 2: The Tracking Workflow



[16] Source: http://www.ussg.iu.edu/hypermail/linux/kernel/0201.3/1657.html
[17] Such changes are called change sets (csets). Each cset is one or more changed files, bundled together. On this point see the Linux documentation archive: http://lxr.linux.no/source/Documentation/BK-usage/bk-kernel-howto.txt?v=2.5.56

This shows that BK spurs "a network of people approach" where developers need to rely on each other to push their patches forward. BK, in addition, implies the acknowledgment that, as the organizational size grows, it is a network of interpersonal bonds not single individuals that guarantees minimal disruption of the underlying system of causal cycles. Ultimately, an organizing model that leans toward adaptability admits that it is minimal disruption of the enactment processes at work that ensures prompt reaction to changing contingencies.

5 Conclusions

"How can I know what I think 'till I see what I say?" is the poignant epistemology of Weick's (1979) masterpiece. This paper, otherwise, is dominated by a slightly different epistemology: how can I know what I think 'till I see what they say? The use of the pronoun "they" as opposed to "I" aims at emphasizing the fact that decision-making in Linux does occur but it happens after development, not before it. In other words, contributors are free to develop whatever features they like and Torvalds, subsequently, decides whether to implement them or not. In our view, this subtle point has passed unnoticed for too long in the literature on Linux: researchers and practitioners alike focus too much on the evolutionary nature of Linux, thus missing its unconventional decision-making process. This paper attempts to fill this theoretical gap by showing that the Linux managing model challenges conventional rational wisdom by epitomizing a reversed managerial metaphor that corroborates the idea of acting before thinking. Once we acknowledge a reactive strategy in Linux, the next conceptual challenge is asking: what does this strategy consist of? This paper attempts to suggest a reactive strategy that leans toward adaptability rather than adaptation, enactment rather than retention, action rather than coordination (Yamaguchi, Yokozawa et al. 2000) and exploration rather than exploitation (March 1991). After all, in the Linux case study, portability,

modularity and versioning tools (BK) all grant the same property, namely the ability to fit changing environmental contingencies. Portability and modularity, in particular, grant the ability to create a minimal kernel that is independent of the hardware setup whereas Bitkeeper allows for scaling up and down as the organizational size changes.

Even though it is premised on the assumption that developers are driven by anticipated reciprocation and overlooks the more stringent design requirements stemming from vertical architectures, this study shows that organizing models a-la Linux are better suited to highly volatile and unstable environments.

References[18]

deGoyeneche, J. M. and E. A. F. deSousa (1999). "Loadable Kernel Modules." IEEE Software **15**(1): 65-71.

Erenkrantz, J. R. (2003). Release Management within Open Source Projects. Taking Stock of the Bazaar: Third Workshop on Open Source Software Engineering.

Feller, J. and B. Fitzgerald (2002). Understanding Open Source Software Development. London, Addison-Wesley, Pearson Education Ltd.

Fogel, K. and M. Bar (2001). Open source development with CVS. Scottsdale, AZ : Coriolis Group Books.

JØrgensen, N. (2001). "Putting it all in the trunk: incremental software development in the FreeBSD Open Source project." Information Systems Journal **11**(4).

Kollock, P. (1999). The Economies of Online Cooperation: Gifts and Public Goods in Cyberspace. Communities in Cyberspace. M. Smith and P. Kollock. London, Routledge.

Kuwabara, k. (2000). Linux: A Bazaar at the Edge of Chaos, First Monday: 5/3. **http://www.firstmonday.org/issues/issue5_3/kuwabara/, Accessed on 22/02/02**.

March, J. G. (1991). "Exploration and Exploitation in Organizational Learning." Organization Science **2**(1): 71-87.

Raymond, E. S. (2001). The Cathedral and the Bazaar. Sebastopol, CA, O'Reilly.

[18] Do not include data from the Linux mailing lists.

Shaikh, M. and T. Cornford (2003). Version Management Tools: CVS to BK in the Linux Kernel. Taking Stock of the Bazaar: Third Workshop on Open Source Software Engineering.

Torvalds, L. (1999). The Linux Edge. Open Sources: Voices from the Open Source Revolution, O'Reilly and Associates. **http://www.oreilly.com/catalog/opensources/book/linus.html; Accessed 31/0/3/02**.

Weick, K. E. (1979). The social psychology of organizing. Menlo Park, California, Addison-Wesley Publishing Company.

Welsh, M. (1995). Implementing Loadable Kernel Modules for Linux. **http://www.ddj.com/documents/s=991/ddj9505a/9505a.htm#01c1_0087, Accessed on 15/03/03**.

Yamagishi, T. and K. S. Cook (1993). "Generalized Exchange and Social Dilemmas." Social Psychology Quarterly **56**(4): 235-248.

Yamaguchi, Y., M. Yokozawa, et al. (2000). Collaboration with lean media: how open-source software succeeds. Proceedings of the Conference on Computer Supported Cooperative Work (CSCW '00), Philadelphia, PA.

Acknowledgments:
The author is indebted to Jannis Kallinikos, Maha Shaikh and Edgar Whitley for their precious comments and support.