Ph.D. research proposal

Method diffusion in large open source projects

Martin F. Krafft <martin.krafft@lero.ie>

http://martin-krafft.net/phd/

11 Aug 2006

The Irish Software Engineering Research Centre
CSIS, University of Limerick, Ireland
Under supervision of Prof. Brian Fitzgerald

Abstract

The proposed research aims to investigate the challenge of how to diffuse methods designed to improve the efficiency of existing processes in large open-source volunteer projects. The primary focus is on the Debian project, possibly the largest open-source project to day. The processes in operation within this project often no longer meet its tremendous growth in size and popularity, mostly failing due to bottlenecks and accountability issues. As a result, the project suffers from delays and stagnation, and a noticeable drop in the efficiency of its thousands of contributors. Some of these issues can already be addressed with existing tools, but their deployment within a project of the size as Debian requires careful planning. In addition, the Debian project consists almost entirely of volunteers with vastly different motivations, an aspect which renders authoritarian deployments ineffective.

The project has seen a number of successful (and unsuccessful) tool adoptions, and their study in combination with surveys, process and work habit analysis, mock designs, and experimental rollouts is expected to reveal the necessary conditions under which a large-scale diffusion of collaborative tools will be successful. The definition of such conditions would help to improve the efficiency of open source projects, as well as contribute to general management strategy a framework that can help treat personnel as volunteers.

1 Introduction

1.1 Overview of the research domain

The Debian project [deb] is perhaps the largest, globally distributed computer project [GBROP+04]. It's flagship product, the Debian GNU/Linux operating system combines the Linux kernel with more than 10 000 software packages, adhering to a common policy aimed to bring tight integration. While its development over the past decade tells an astonishing success story, the project is beginning to groan under its sheer size [Mic04]. Large parts of the Debian development process are accomplished with tools which have iteratively improved to meet new needs as they arose. The immense speed at which new developments take place, the size of the project, and the ever increasing popularity of the Debian operating system frequently render these tools inadequate and cause severe delays. Figure 1 illustrates how the time span between subsequent releases has been increasing for the past seven years [Mic05a].

With scaling problems surfacing on a regular basis, Debian developers started to explore new techniques and approaches to cope with the challenge of growth. In a project where maintainers enjoy near absolute control over their software package or components of the infrastructure [Col05, ch. 6], issues frequently arise over bottleneck situations, when a single developer is unreachable, overloaded, or not willing to consider suggestions by others. For this reason, the Debian project has been identified as a "bazaar of cathedrals" [Kra05a] (*c.f.* [Ray99a]). One of the most notable trends in recent years is for developers to team up to co-maintain important software packages [RGBM05]. Going a step further, Ubuntu, an immensely successful derivative of Debian [ubu], practises completely open collaborative package maintenance. Instead of Debian's notion of package ownership by maintainers, any of the Ubuntu developers are authorised to make changes to a given package of the core distribution.

However, the adoption of a team maintenance model has been considered an "unrealistically simplistic solution" to bottlenecks [MH03]. A move from single to multiple maintainers brings with it a sudden increase in complexity of communication and coordination [Bro95]. The classic response to such an increase in coordination complexity is the deployment of methods from the wide domain of computer-supported cooperative work [Gre88]; for example, apart from the extensive and multifarious communication media in active use [Col05, ch. 6], the open source community relies heavily on version control systems (VCS)[1] [AB02, Mic05b, Joe97], such as `CVS` [FB03], which are part of the large set of CSCW tools. Nevertheless, as I will argue further down, the available tools, that are applicable to Debian package maintenance, suffer from usability problems themselves, generally adding complexity for the individual and making it harder to contribute, or require a considerable amount of time to setup,

[1] Even though technically a subset of software configuration management (SCM), VCS is often used interchangeably with SCM in the F/OSS world. Other synonyms include source code management, source control, and revision control systems (RCS) [Whe05].

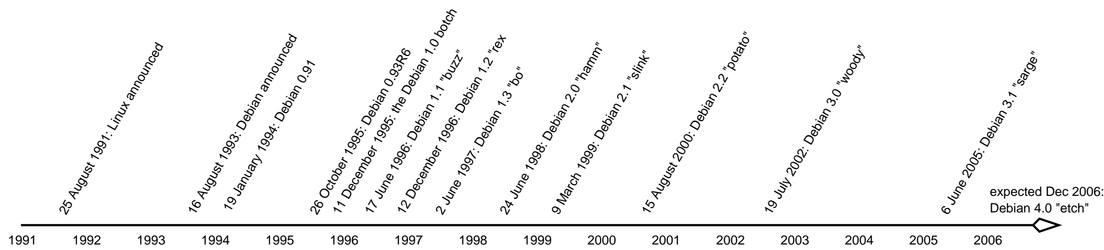

Figure 1: Debian has had difficulty dealing with its own growth, a problem most apparent in the increasingly long cycles between stable releases.

which is often not available at the moment such tools would be needed.

A core focus of this research is the volunteer nature of the Debian project. While commercial deployments are often based on authoritarian decisions, no such authority exists in the Debian project, nor could it be established. A successful rollout must therefore use other vehicles to achieve the critical mass, and those have to be empirically determined.

1.2 Terminology

In this paper, a number of domain-specific terms are used according to the following definitions:

Method
: a strategy to accomplish a given goal, often involving the use of tools.

Technique
: a specific way to cause a change. This may be through the use of a tool, but could also refer to a more abstract means, such as an algorithm.

Methodology
: the comparative and critical study of methods in general.

Process
: a predefined sequence of operations designed to result in a particular change.

Tool
: a means to facilitate a given operation.

An in-depth analysis of the terminology will constitute part of my Ph.D. thesis.

1.3 Statement of the problem

The tendency for team development in Debian calls for methods that help eliminate bottlenecks, rather than create new ones. Additionally, reducing the complexity and standardising a new, flexible approach to package management should make it easier for one-time/infrequent contributors to donate bits of their time here and there, without having to first acquaint themselves with a different set of tools for each package. The playful motto of this research is to support the random Debian user, who has two hours to spend on a rainy Saturday afternoon before heading off to a concert in the evening, and would much rather use that time

to make a valuable contribution to the project than to spend most of it trying to figure out how to do that.

From a technical point of view, modern VCS are capable of coordinating even the largest F/OSS projects [Ian05, MFH02]. In fact, several developers are already maintaining their packages under source control, a practise backed and facilitated by tools such as `cvs-buildpackage`, which strive to automate most aspects of building a package maintained in a repository. Because the choice of VCS can be religious to some users, every other major source control system has a corresponding package with tools to aid the building of packages maintained in the system (*e.g.* `svn-buildpackage`).

The usage of the aforementioned building tools requires the developer to strictly adhere to a structure laid on top of the VCS mechanisms and generally leaves the maintainers exposed to the intricacies of each system, rather than allowing them to stay clear of their intentions to maintain a Debian package. Furthermore, the systems' requirements often call for the migration or restructuring of existing repositories or development structures, which can be a time-consuming task. Lastly, with four or five of these build tools available, which readily diverge in terms of usage paradigm, a developer familiar with one may not be able to easily contribute or help out with maintenance of a package using another one of these tools.

As I will illustrate below, the act of building packages is only a small component of Debian package management. Other pertinent tasks include patch management, change log keeping, correlating changes with entries in the bug tracking system, bug triaging, unit and regression testing, checking for policy compliance, and uploading the package. At present, these tasks are separate processes with little or no integration. Each process may further be governed by preferred practises by the primary package maintainer. As a result, package maintenance leaves a lot of room for human error, and requires a strict set of guidelines to be followed by all contributors in the case of team-managed packages.

As Debian continues to grow, the project will have to deal with the problem of bottlenecks and rely more on distributed contribution [Daf01]. At present time, several approaches to Debian package management in teams exist. However, none can do without a set of guidelines or best practises, and none integrate the separate sub-processes in a way to eliminate redundancy or reduce the manual (and thus error-prone) work required from the maintainers in a meaningful way. Moreover, several different, incompatible approaches limit the domain of potential contributions for each volunteer, as I assume that few will go through the trouble of learning different methods to accomplish the same task.

1.4 Statement of the challenge

As previously mentioned, the Ubuntu derivative follows a radically different strategy of package maintenance than Debian. Moreover, the company behind Ubuntu, Canonical Ltd., is pushing the standardisation of a number of collaborative

methods within the community, including a single VCS as well as one coordination platform to be used by all contributors for almost all aspects of the project. Even though still in its infancy, the approach looks promising.

A similar approach within the Debian project would fail, however, due to its inevitably revolutionary nature. Ubuntu is able to successfully standardise its development practises due to a number of traits that distinguish it from the Debian project: it's a young project profiting from early adoption, and it concentrates mainly on its core distribution, which consists of only about 2 000 packages. Almost half of the core development team members are on a Canonical payroll and hence in a favourable position to act as "opinion leaders" [Rog95, p. 27] regarding any rollout, while simultaneously working closely with the company to improve the methods.

Debian, on the other hand, is more than a decade older than Ubuntu, and the need for collaborative maintenance has surfaced long before Ubuntu came to life. Over the course of the years, package maintainers have often developed ways to facilitate package management, but as a result, a plethora of methods exist, some overlapping others in terms of functionality, and many not flexible enough to allow for their incorporation into existing processes. It thus seems highly unlikely that any of these methods will enjoy project-wide adoption, nor can the project as a whole push any such adoption as it is not possible to force volunteer contributors to adopt new methods [Mic04]. Many developers treat Debian development as a hobby and would discontinue their contributions if they were told what to do.

The core challenge of the proposed research is to determine the conditions under which voluntary adoption of methods can occur among a diverse, globally-spaced group of individuals. My strategy will be the diffusion of a tool designed to improve the Debian workflow, following guidelines established from past experiences with successful and unsuccessful tool adoptions within the project.

2 Debian package management and source control

2.1 A brief overview of Debian package management

The nature of the work performed by Debian developers is different from conventional software engineering. The task of a Debian developer is that of integration, rather than the development of software *per se*. Debian developers take public releases by software projects, such as Apache or KDE, compile them, and assemble them into packages according to a strict set of rules to ensure the coexistence of thousands of software packages on a single system [Deb05].

When a software has been successfully packaged, it is uploaded to the Debian archive and globally distributed to hundreds of mirror servers. The upload marks the beginning of the regular package life-cycle. The package will first spend some time in the `unstable` archive, and when it meets a set of basic consistency crite-

ria, it propagates to the `testing` archive, from where the collection of packages included in a `stable` release is sampled. Users are able to install software from any of these three archives, and usually identify bugs in the software, which they report to the Debian bug tracking system [bts]. The package maintainer then coordinates with the upstream author to fix the reported problem(s) and uploads a new package to the `unstable` archive, which eventually replaces the buggy version. This process is detailed in [Kra05a, ch. 4].

Debian developers usually depend on the software they package and may need to add features to the software for their own use, or integrate improvements, enhancements, and bug fixes from other contributors. Largely to lessen their own maintenance load, the Debian developers generally try to feed all such changes to the upstream author, in the hope that they might be integrated with the original source code and thus need not be maintained specifically for Debian any longer. Regardless, it is not uncommon for upstream to be slow in accepting patches, to refuse them, or for patches to be specific to Debian. Therefore, package maintenance often includes the management of differences between the upstream version of a software and the version provided in the Debian package, and to port these differences to new upstream releases.

2.2 Applications of VCS in Debian package management

An ever more popular trend is the maintenance of Debian packages in VCS. For simple packages, it usually suffices to commit the Debian-specific control files to a versioned repository, from where they can be checked out and combined with the pristine upstream source code to produce a Debian package. However, as soon as changes to the upstream source are required to fix bugs, or new features are to be implemented, it is favourable to have access to the original source code from within the VCS. Since upstream authors and Debian package maintainers do not usually share the same repository, the maintainers are faced with the challenge to track new upstream releases while maintaining changes made to previous versions.

One approach to meet this challenge is to reapply changes made during the process of "debianisation" of the predecessor version to the source code belonging to the recent release. This approach gives priority to upstream changes and requires the maintainer to reassess the necessity and plausibility of each change with each new upstream release, therefore favouring those who keep the set of changes small (and have cooperative upstreams). In fact, this approach is inherent in the structure of a Debian source package, which consists of a tarball of the original source tree (`.orig.tar.gz`), as well as a patch that encapsulates the changes needed to turn the source tree into the source tree of a Debian package (`.diff.gz`).

Another approach uses what is commonly referred to as "vendor branches" (for lack of a better reference: [CSFP06]). With this approach, the upstream source is stored in VCS alongside the Debian control files. On release of a new upstream version, the differences between the predecessor and the current release are merged into the VCS and conflicts resolved. This is subtly different from reapplying changes made by the Debian maintainer to the new release, in that it gives priority to the Debian-local changes. Over time, vendor branches may unwillingly lead to the divergence from upstream.

Orthogonal to the representation of differences is their separation into smaller chunks: it often makes sense to keep changes related to a specific feature or bug fix isolated from other changes to the package. The benefits of such compartmentalisation extend from facilitated upstream submission of features to bug triaging and authoring package changelogs, to name but a few. The Debian archive is home to tools that aid in managing related patches in bundles (`dbs`, `dpatch`, and `quilt`), and VCS with proper branch support make it easy to track sets of changes in multiple parallel "functional branches" [ABCO98] (which are often called "feature branches").

2.3 Status of VCS use for Debian package management

Despite the aforementioned trend towards collaborative maintenance, VCS approaches to package maintenance are having a slow start. For a long time, only a small number of projects used the official Debian `CVS` server, which can only be used by Debian developers. In early 2003, the Debian project introduced `Alioth` [Her03], a service similar to SourceForge [sou]. Among communication tools, such as web space, mailing lists, and bug and request trackers, `Alioth` hosts `CVS`, `Subversion`, and `Arch` repositories. After three years, 670 projects were registered with the `Alioth` server, and around 250 of them used one of the source control mechanisms available, with the majority on the side of `Subversion`.

With around 10 000 source packages in the Debian archive, this number may seem strikingly low, but large part of the Debian archive consists of trivial packages with little or no upstream activity, and few Debian-specific modifications, if any. For those packages, as well as ones maintained by single individuals, where a VCS is not essential, no motivation exists to migrate the source package to a VCS repository. Nevertheless, even packages with two or more maintainers often do not use VCS. Common practises in such a situation include turn-taking and sending patches back and forth. While this procedure tends to work fine most of the time, (temporary) disappearance of a developer, which is quite common in volunteer projects [Mic04], can cause maintenance of packages to stall. The lack of revision history or a proper patch management system make it difficult for new maintainers to pick up where others left off, work is often done more than once due to lack of coordination, and one-time/infrequent volunteers frequently do not even know where to start.

Several packages aim to facilitate package

management with VCS, among them `cvs-`, `svn-`, `tla-`, `arch-`, `darcs-`, and `bzr-buildpackage`[2]. All of these seem to descend — in spirit at least — from the first, but their usage paradigms readily diverge when it comes to actual use. Moreover, all require specifically configured repositories and are thus not (always) applicable to existing repositories, hence their use calls for a migration. Such migrations mostly cannot happen in-place, which means that a new repository is usually required, and if only temporarily. From experience, this burden already discourages most developers.

The slow acceptance of VCS methods in Debian can be attributed to several reasons. First and foremost, package maintainers seem unable to decide on which VCS to converge, even for a single package. The three officially supported ones by Debian listed in this section's first paragraph have well-known flaws and maintainers are thus reluctant towards investing time into them. Over the past years, a considerable number of alternative VCS have become popular, but those are generally considered to be too novel and young for a developer be able to judge one system over another. A small number of developers have adopted these new methods and are each going their own ways without much exchange of ideas or efforts to unify approaches. The other package maintainers seem to prefer to let them sort out the intricacies first and stick to the conventional methods for the time being.

Another widely accepted reason for the slow adoption of source control for Debian package management appears to be the inertia on the side of the maintainers, who have maintained packages the traditional way for a longer time and would rather employ quirks than to invest time in learning a new system. Several developers cling to standard Unix tools they can expect to find on any compatible system, rather than to embrace helper utilities designed to automate repetitive tasks in Debian package management (*c.f.* [Sri05]). One of the most important motivations driving Debian developers is that they are allowed to do what they want, as long as they observe basic rules of conduct [dmu05], try to follow best practises [BCHS05], and their packages adhere to the Debian policy [Deb05].

2.4 Usability problems: bottom-up vs. top-down

By far the largest reason that Debian package management and VCS are still unwed, however, seems to be the low usability and steep learning curves of VCS tools[3]. Perhaps these reasons explain why `CVS` and `Subversion` continue to be widely used systems, despite their technical limitations: they are moderately easy to learn, and it is not hard to find assistance. When the Debian installer team decided on a VCS to manage the new installer, one of the reasons they chose `Subversion` was because a large part of the hundred contributors was already challenged enough by the usage paradigm of `CVS`, so an even more complex system would have been counter-productive [Hes05].

[2] `bzr-buildpackage` is in development and does not yet exist as a package. See `http://bugs.debian.org/380198`

[3] A formal usability/acceptance study in the spirit of [Dav93] does not seem to exist.

This is in line with Raymond's claim: "the number of contributors [...] is strongly and inversely correlated with the number of hoops each project makes a contributing user go through" [Ray99b]. Specific to the domain of VCS, Hudson adds: "In many environments, a shallow learning curve is the most important feature of a VCS. [...] [A] steep learning curve [is one of] fundamental and insurmountable obstacles." [Hud04]. Translators, for instance, are motivated to contribute to a project such as the Debian installer to help people from their cultural background use Debian in their native language; they are not (necessarily) keen on learning how to interact with a VCS, and then to apply for commit access to the project's repository, only to be able to do their work (*c.f.* [Stu05]).

Even with a thorough understanding of package management procedures and VCS concepts, a developer may often have to jump through hoops to make proper use of VCS when maintaining Debian packages. The involved processes are far from integrated.

The low usability of VCS tools is a result of several factors. First and foremost, VCS itself is subtle and non-intuitive [Lor03], and requires perhaps more of an understanding of the underlying model from the users than other tools [Poo05]. Therefore, it is necessary to distinguish between usability as experienced by the uninitiated user, and usability experienced by the acquainted developer.

The two centralised methods, `CVS` and `Subversion`, seem to form the basis for both groups. Their underlying model for basic usage is accessible to most, the user interface is consistent (if awkward at times), the tools are vastly documented, and support is readily available as most every acquainted developer will have had to use either of these tools at one point in time. `Subversion` addresses and fixes some of the deficiencies of`CVS` and is thus the first choice for projects looking to deploy a VCS.

When it comes to distributed systems, however, the experiences of new users and experienced developers readily diverge. First, the underlying concepts are harder to learn [Hud04]. And second, the tools are generally younger and thus not too well documented, known, or supported[4].

Furthermore, the distinguishing features of distributed VCS systems exceed the basic tasks like updating and committing. In distributed source control, other topics are at the focal point, such as conflict resolution, three-way merging, repository aggregation, and complex branching strategies. Advanced concepts such as the aforementioned are primarily of benefit to experienced developers taking part in projects of greater complexity [Ask02]. Usability tends to play a less important role for advanced users, as they are usually familiar with their methods and have learnt their usage paradigms, or are generally more capable of abstraction in this domain [NT03]. Therefore, the demand for technical correctness and flexible features is much larger than the desire for usability. Second, advanced users are also often

[4] Documentation does exist for the major tools, but it is not always to be found easily. Also, it is usually targeted at the experienced user or just serves as a scratch pad for the developers to dump their thoughts.

contributors to the tools they employ, or develop tools that meet their specific needs (*c.f.* Torvalds' `git`), which results in "top-down tool design": the development is driven by the theoretical model underlying the VC model, and the designer moulds the code until it fits the model [BA02, Ber91].

Tools designed to meet a certain purpose require users to learn to bridge the gap between their intuition and the user interface or model of operation. Quite often, this gap decides between success and failure of a tool. A good example for the lack of usability leading to the failure of a tool would be the `tla` implementation of `GNU Arch` (and all its ancestors), which quite possibly has the worst user interface across all VCS tools. This claim is backed by Canonical's `tla` fork `bazaar`, which does not add features but merely aims to remove the awkwardness from `tla`, and to make it more usable; the fork has since been abandoned, probably due to design deficiencies in `GNU Arch` itself. The development of `GNU Arch 2.0` seems to have been discontinued.

If usability is the goal, simple top-down approaches will not succeed, and a design strategy must include a bottom-up, user-centric component [Tha05]. The balance between approaches decides whether the result is "usable but not useful" [Dav93, Bev95], or *vice versa*. As previously noted, however, an understanding of the conceptual model of VCS is indispensable for users of VCS at any level. Therefore, traditional bottom-up approaches, such as user process analysis [Tha05] (*c.f.* the Multiview methodology [KPH99]), are not going to succeed without the user acquiring knowledge about the theory of VCS. To be able to understand this theory requires users to be acquainted with software development processes, at which point they are not beginners anymore. This Catch-22 situation makes it unsurprising that VCS has been identified as a "wicked problem" [RW73], that is a problem which is not understood until after the formation of the solution, a problem of which stakeholders have radically different views, a problem with constraints and resources changing over time, and a problem that is never solved [Con03].

This does not intend to suggest that usability in source control tools is hopeless. On the contrary, the development teams behind modern tools, such as `Bazaar-NG` (`bzr`), are making an effort not to forget about usability. A motto such as "Bazaar-NG should be a joy to use." [Poo04] may be sound like a vacuous slogan from the marketing department, the current state of development seems to suggest otherwise, however. `Bazaar-NG` aims to address usability problems by reducing the number of concepts for a simplified mental model, and by providing a consistent, modular user interface with only a handful of commands. At the same time, a plugin infrastructure gives hope to advanced users who don't want to sacrifice functionality. Furthermore, the developers tried to stay true to the conventions of existing VCS to facilitate migration, and to speed up essential processes for instant gratification, which increases acceptance.

2.5 The role of Canonical and Ubuntu

Canonical, the company behind the Ubuntu derivative [ubu] has been investing much effort into *Launchpad*, an infrastructure designed to unify all aspects of packaging (bugs, translations, enhancements) in a single interface. As part of the endeavour, Canonical is also sponsoring the so-called *super mirror*, a server designed to host `bzr` repositories that track *all* active F/OSS projects, on which basis the Ubuntu operating system will then be built [Rem06]. As part of this endeavour, Canonical is sponsoring the development of the aforementioned `Bazaar-NG` VCS (`bzr` for short).

All in all, Canonical's strategy for Ubuntu overlaps closely with the ideas I put forth in my original Ph.D. proposal submitted a year ago, which were concerned with the large-scale deployment of VCS in Debian. The core focus of the research I propose has since changed, but Canonical will continue to play a role in it, even though Canonical's work is unlikely to be of direct benefit for Debian, as it is geared towards Launchpad. Launchpad is provided for cost-free use, but Canonical restricts access to the source code as well as the raw data[5], which does not align with Debian's ideology of freedom [DFS04] and could result in the type of vendor lock-in which Debian has meticulously avoided. Furthermore, Launchpad is at the core of Canonical's business plan, and will thus probably not take into account the interests of Debian or its developers [Tow05].

Finally, frictions between Debian and Ubuntu as perceived from the Debian angle [Kra06] are likely going to keep some Debian developers from adopting Ubuntu's methods, even though those methods may well be technically superior to the ones in use by Debian developers. The influence of these frictions on the use of the super mirror as hosting platform for `bzr` repositories, which will be possible independent of Launchpad, will have to show.

As said, the Ubuntu project and Canonical as its sponsor, however, continue to play a role in the proposed research, mainly because of the lessons to be learnt from the diffusion of Launchpad into the Ubuntu development process, but also because of the continuing challenge for Debian to profit more from its derivatives. Any surveys and studies of habits will include Ubuntu developers (as well as developers from other derivatives) in the test groups.

3 Proposed research

3.1 Overview

The final goal of this research is the documentation of conditions under which volunteers, such as the developers of the Debian system, adopt new methods to accomplish their tasks.

Due to my involvement with the Debian project, the choice of participatory action research [WGL91] as fundamental research model seems logical. In particular, the client-system infrastructure is defined to be the domain of development of the Debian GNU/Linux operating system, and my position as official developer of

[5] This also applies to SourceForge [sou].

the system automatically makes me a member of both, the researching as well as the practitioner group. Moreover, other developers have already announced their interest to perform as "client researchers." Apart from voluntary contribution and cooperation from select individuals from the developer collective, however, I do not expect the Debian project as a whole to condone or support my research. Specifically, I shall not abuse my privileges as a Debian developer in any way that could harm the project.

According to the action research approach as formulated in [Bas99], the research endeavour consist of five phases, for which I list my intended steps in the following:

Diagnosing
: based on an analysis of the alleged stagnation and bottlenecks within the Debian project, using mailing list posts as well as a number of selected interviews, it is my hope to clearly identify the need for change.

Action planning
: given the previous analysis, I plan to identify a set of potential changes in such a way that the adoption of new methods into the developer workflow could cause the needed change. Based on a study of previous diffusions, I will then formulate a set of guidelines to follow for the introduction of new methods.

Action taking
: together with the community, I plan to develop a tool designed to address one specific change, following as close as possible the previously established guidelines.

Evaluating
: with the diffusion underway, I want to analyse the success, both in terms of adherence to the guidelines previously established, as well as from the perspective taken during the analysis of previous diffusion during the "action planning" phase.

Specifying learning
: the final goal is the specification of conditions that lead to the successful diffusion of a tool into the Debian developer workflow. This specification summarises the lessons learnt and hopefully can serve as the basis for future rollouts.

For the purpose of the following discussion, I would like to highlight two aspects of the research:

1. The analysis of previous diffusions, both those that have led to successful adoptions, as well as those that have failed.
2. The design of a new tool, planning of its diffusion strategy, and subsequent release as well as the analysis of its adoption.

3.2 Previous diffusions

In the 13 years of existence of the Debian project, a large number of tools aimed to change the development process have come and gone,

and some have stayed. For this part of my research, I would like to select a number of tools that were not widely adopted (where the diffusion process has failed), as well as some tools that have changed the way in which Debian developers work (and which can thus be said to have been successfully diffused). For each tool, I intend to study the circumstances of its adoption (or failure) in an attempt to determine the critical factors of influence.

3.2.1 Analysing diffusions

Rogers' framework [Rog95] can serve as a comprehensive model to descriptively analyse past diffusion/adoption processes. Rogers identified four main aspects of diffusion: innovation, the communication process, the adoption process, and the social system. While little change in the social system can be perceived over the years, the previous diffusions differ substantially in the other three elements.

The element of innovation actually consists of a number of constituents (adapted from [RC89]):

Relative advantage
: the degree to which an innovation is perceived as an improvement upon its precursor;

Visibility
: the degree to which the results of an innovation are visible to others;

Compatibility
: the degree to which an innovation is perceived as being consistent with the existing values, needs, and past experiences of potential adopters;

Complexity
: the degree to which an innovation is perceived as being difficult to grasp and use;

Assessibility
: the degree to which an innovation may be tried before adoption;

Moore and Benbasat have expanded on the model to create the "Perceived Characteristic of Innovating" (PCI), which essentially builds on Rogers' framework but assesses perceptions of use by adopters rather than perceptions of the innovation itself [MB91], which is an interesting alternative, though not necessarily a replacement: hackers — as Debian developers can undoubtedly be classified — are not necessarily always pragmatic and only perceive the actual benefit of using a tool, they are often also influenced by the perception of the tool itself, whether it is elegant or ugly, for instance. Nevertheless, especially in the light of the strategy of participatory action research, where "the emphasis is more on what practitioners do than on what they say they do" [ALMN99], the PCI gains importance as a qualitative measure of diffusion success.

With the help of interviews and possibly even a large-scale survey, it should be possible to come up with a side-by-side comparison of previous diffusions, based on Rogers' framework, from which I can find answers to questions such as "why did this diffusion fail," or "what made this tool be so widely adopted."

Another interesting point of focus could be acceptance measurements, including but not limited to Davis' Technology Acceptance Model (TAM) [Dav93], though it is difficult to judge their usefulness at this stage, having no concrete data available. TAM specifically, despite being controversial, is one of the few validated models in the information systems research domain, and thus certainly deserves careful consideration.

In assessing the success of tool adoptions, it may be relevant not only to look at the acceptance on the side of the developer, but also at the impact on the quality of the produce of development. Debian is a very quality-oriented distribution, and tools that are perceived to improve the quality generally seem to gain more acceptance than those with unclear or negative effects on the overall quality of the output.

3.2.2 Possibly interesting previous diffusions

Maybe the most interesting and important pair of tools to investigate are `debhelper` and `cdbs`, which are both tools designed to facilitate the creation of Debian packages. While `debhelper` is in wide use, the younger `cdbs` is struggling for acceptance, with contenders in both camps, those who love and those who loathe it. Along the lines of `cdbs`, `yada` also never got off the ground.

Another interesting package to inspect would be the collection of small helpers in `devscripts`, simply because of the way in which these are distributed. It is my impression that many of the available tools are not known to many, but readily adopted if discovered.

Yet another package to consider is `pbuilder`, a tool that helps maintain a base level of quality when building package, and which has found many friends despite its technical limitations and resource-intensive way of operation. Interestingly, a number of seasoned developers seem to object to its use and often criticise others for using it.

Debian's collaboration server `Alioth` will make an interesting case, because it is made up of components, some of which have been readily adopted, but others have not.

Patch management systems like `dpatch`, `dbs`, and `quilt`, as well as the VCS helpers like `svn-buildpackage` are certainly to be included in the list. Of particular interest may be `tla-` and `arch-buildpackage`, which is associated by many with a general failure to use `GNU arch` for Debian package maintenance due to complexity.

Package checker tools like `lintian` and `linda` have enjoyed wide adoption, in part because many developers refuse to work with packages that do not pass these automated checks, and also because they have been integrated with other tools, automating the process. It is this aspect that makes them specifically interesting. Also worth a look are `piuparts`, which seems very complex/cumbersome to work yet is still being adopted, and `debian-test`, which utterly failed.

The above are some examples. In the inter-

[6] `http://wiki.debian.org/madduck/adoptions`

est of gathering further input, I've set up a Wiki page[6], which has seen many contributions already.

3.3 Diffusion of a new tool

3.3.1 Idea and design

The `Bazaar-NG` team is producing a VCS that is technically sound, as well as engineered with usability in mind. However, it still requires a general understanding of the model of the VCS to be useful [Poo05], and just like any other VCS, it depends on the users to make decisions about the organisation of files in the tree under version control, their discipline to meticulously commit changes, and their understanding of the VCS to allow them to extract information. The reason for all these points stems from the genericity of a VCS, which is designed to bookkeep essentially any type of file data.

Conceptually, a layer on top of the VCS, which hides the actual VCS is possible; a user would not tell the VCS to commit a file, or to create a branch, but rather issue commands such as "put this project tree under version control," "I want to add the feature *foo*," or "prepare everything for a release." However, VCS are already at a fairly high conceptual level, and tools that implement even higher concepts are bound to make decisions or assumptions about the data they control, thus losing genericity. In Debian package management, however, the genericity of VCS is not needed and assumptions can be made for the highly specific domain, as certain operations would always follow more or less the same patterns. Nevertheless, the line has to be drawn carefully as it is unlikely for a method to be adopted out of free will, if the method imposes a strict *modus operandi* on the user. Put differently, a method's adoption rate seems to be inversely proportional to the learning effort required from new users.

Compatibility

In fact, a major point of focus will have to be the compatibility of any new tools and approaches with existing methods, for a number of reasons: First, acceptance of novel techniques in Debian is slow, so a transitional period of several years is to be expected, and some users will never switch away from their own way of packaging.

Second, Unix users, and Debian developers especially, like to stay in control and frown upon monolithic solutions with a lot of obscure, inaccessible magic; the entire Debian operating system is built in accordance with the Unix philosophy of modularity, and much of Debian's robustness and professional reputation comes from its numerous small utilities that do exactly what they should (and nothing more), and which are purely optional in that the steps they take to accomplish a certain task are standardised and/or well-documented, such that no developer is forced to use them.

The third point calling for new tools to maintain compatibility is the tight interdependence between the Debian infrastructure and the processes of Debian package management. A "revolutionary" approach would require parts of the infrastructure to adapt, which, given the size and complexity of the project and its infrastructure, would simply not be possible.

Fourth, because the goal to produce a standardised approach to Debian package management is somewhat idealistic, it is important to produce compatible alternatives and gradually attract users through functionality rather than to set out writing "the standard;" processes cannot be made standard, they become standard, and this is especially true in a volunteer project such as Debian.

Finally, using existing and well-known building blocks will facilitate contributions by others, and hence make development of these tools possible in true F/OSS fashion.

Obviously it will be quite a challenge to provide a tool that can seamlessly integrate with existing packages and maintain compatibility to the maintenance practises in use. I expect policy-based design [Ale01] to go a long way towards the solution of this problem: export assumptions about the structure of the source package and its management practises to a configuration file that is to be managed as part of the package.

Specific ideas on integration

Debian package management is a very broad field, and one without any "right answers." It is thus difficult to isolate specific areas that need to be improved. The lack of VCS is certainly a huge gap because it puts the burden to coordinate between multiple developers on those developers, as well as the task of tracking changes to a package across several versions. However, VCS is not all there is to it, several sub-projects in Debian have been using VCS for years, and yet there is still room for improvement, mainly through better integration. The following paragraphs illustrate some ideas.

Every Debian package is required by policy to contain a change log identifying modifications between Debian revisions of a package, but not including changes made by upstream. Good packaging practise includes keeping this log meticulously up to date. Version control systems can attach log messages to changes in the repository, so it would only be logical if those log messages could be trivially reused for the Debian change log. No tool currently exists to automate this process in a meaningful way.

The Debian change log is also used to interact with the Debian bug tracking system [bts]. For instance, the recommended procedure to mark a bug as resolved is to include the bug number in the change log next to the relevant entry, which will automatically cause the bug to be closed when the package has been successfully updated.

At the same time, the bug tracking system is often used as a request tracker and coordination platform for single developers or maintainer teams. Currently, the bug tracking system provides for no integration with VCS mechanisms. For instance, patches attached to reports filed with the bug tracker need to be manually imported into source control, and vice versa, changes committed to the repository addressing a certain bug are not made available as part of the bug report, which is often a helpful resource for users encountering problems.

Many packages use additional tools like `dpatch` or `quilt` to manage patches within the package, even when VCS is already in use

and could be used to manage those patches in a more advanced manner. Even though the approach may seem backward, it seems that it's preferred because it's conceptually more accessible.

Furthermore, the Debian archive infrastructure is unaware of VCS. Thus, the upload of a new version of a package currently requires a developer to check out the source from the repository, build the package locally, and then upload it to the Debian archive. This process can potentially introduce security problems [Kra04], and developers with slow or unreliable Internet connections find it difficult to upload larger packages. In [Kra05b] exist some preliminary thoughts on how to improve the situation through the integration with VCS, while paying particular attention to quality assurance. Canonical is already taking Ubuntu in a direction to circumvent these issues [Rem06].

Lastly, Debian's infrastructure encompasses several dozens of servers, as well as the most extensive mirror network in the F/OSS world. Most of the infrastructure (and the mirrors especially) are designed in a decentralised fashion to reduce bottlenecks. An integrated package management system designed for team maintenance should harness the power and availability of this infrastructure. One solution is to enhance the Debian source package format in such a way that it can integrate VCS, rather than rely on the extra step to generate the conventional source package from information in the VCS. This is related to the direction in which Ubuntu seems to be going [Rem06], although the idea was developed separately and long before Ubuntu went public with it.

3.3.2 Plan of research and research methodology

Rather than drafting an abstract usage model and expecting developers to accept and learn the model — an approach that would almost certainly fail — I want to study the habits and processes of developers, them identify points of integration in existing processes to hide those repetitive processes that come with but do not constitute a conceptual part of the Debian packaging endeavour, and finally create tools that start at the developer's habits and work their way up from here. Thus, the proposed approach can be said to be bottom-up towards the conceptually high goal of an integrated workflow.

The research plan consists of 3 stages:

Stage 1: surveying and studying habits

During the first stage of the research, I intend to review the fields of version control, Debian package management practises, as well as approaches to process integration found in other, comparable projects (*e.g.* the BSD operating system family, in which version control is used extensively). The results will be used to devise strategy that maximise reuse of existing tools in true Unix spirit, and to gather ideas for possible integration and consolidation steps, as well as experience with pitfalls and successful migration strategies. Looking at comparable projects, it is my hope to learn from their approaches, but also to identify problems and pitfalls their

use has uncovered to try to guard against them in later stages of the proposed research.

Also part of this stage is the studying of developer habits. With the help of a survey, I would like to find answers to questions such as the following (which are only general ideas):

- where do you see the strengths/weaknesses of the Debian packaging process?
- what steps do you consider redundant and where do you see potential for improvements?
- what level of automation in the packaging process would you find acceptable?
- can you think of parallel processes, possibly from real life, which resemble the packaging process?

In addition, I intend to conduct interviews with several developers who are either very progressive about new methods, or very conservative. I expect to gain more insights into the motivation of volunteers, specifically with respect to their preferences when it comes to getting work done. Even though it is unlikely that any diffusion will reach every single volunteer, that should be the goal.

In a further step, I then want to study how developers would like to approach certain tasks. A part of this study will include "live commandline sessions," in which I pretend to be the "system," while the interviewed developer pretends to steer the system with high-level commands. Figure 2 is an example interview held over IRC (Internet Relay Chat) as part of the writing of the original proposal. This transcript describes a typical procedure in Debian package management, but it does not mention VCS at all; all interaction with the VCS mechanisms is hidden behind the high-level wrapper, which will have the interface of a tool designed according to the human language requests formulated by the developers. This form of real-time research over IRC allows for very specific, adaptive studies into the habits of individual developers.

With the data from surveys, interviews, and such live sessions, I expect to be able to identify patterns in management procedures. Furthermore, screening the results will give a good indication of what approaches developers consider intuitive.

Stage 2: planning the diffusion and mock-up implementation

Based on the data gathered during the first phase of the proposed research (chapter 3.2: "Previous diffusions"), I intend to make use of Rogers' framework [Rog95] in a prescriptive way and identify requirements for a successful diffusion of any tools developed in this phase of the research. The outcome should be a flow diagram, possibly a Gantt chart or PERT diagram, which can be used to coordinate the group effort of implementing and diffusing the tool in a way that allows for its success to be monitored.

In this stage, I will also begin with the implementation of mock-ups to be able to determine and possibly improve acceptance from the side of the developers. These mock-ups are going to be "empty hulls" in that they provide a user interface, but no internal functionality (other than

```
<oracle> i want to work on the mutt package
<martin> downloading mutt (1.5.9-2) to ./mutt-1.5.9 ... done.
<oracle> i want to fix bug #654321
<martin> preparing isolated environment for fixing of bug #654321 ... done.
 * martin records intent ...
 * oracle edits a file, fixes the bug
<oracle> i have fixed the bug
<martin> thank you. please give a brief description of the bug fix.
 * martin spawns editor-of-choice
 * oracle describes the bug fix and closes editor
<martin> fix recorded in change log, what else can i do for you?
<oracle> please prepare a new release
<martin> processing release of mutt, version 1.5.9-3 ...
 * martin runs any unit tests ... success.
 * martin integrates fix with distribution source ... success.
 * martin prepares a new package file in ../mutt_1.5.9-3_i386.deb ... done.
 * martin runs policy checkers and other tests against the package ... success.
<martin> new package ready for release in ../mutt_1.5.9-3_i386.deb .
<oracle> please upload the package to unstable
 * martin verifies authenticity and authorisation of requester ... success.
<martin> sealing revision 1.5.9-3 ...
<martin> upload request sent to archive daemon.
```

Figure 2: An imaginary transcript between a package maintainer and a high-level tool

fake feedback and status messages to the user). Feedback from the users will help improve the intuitive feel of the user interface in the context of Debian package management.

Stage 3: deployment and analysis

The third stage starts with the implementation of functionality into the tools, after having settled on a user interface in the second stage. Here, the intention is not to implement tools and release finished versions, but rather to take the "release early, release often" approach [Ray99a] and draw from extreme programming [Bec99] to implement and iteratively improve the tools in cooperation with select developers and in the context of some packaging projects within Debian.

A *modus operandi* has to be found which allows for the collaborative development of the tools, but which also does not preclude the analysis of the tool's success. One possible approach is the inclusion of feedback generators into the tool, so that a central machine can be used to gather statistics about the use of the tool, provided, of course, that the user gave his/her approval.

After some time, the rollout of the tool should be assessed in a fashion similar to the way discussed in chapter 3.2. In addition, a survey gathering feedback would most likely return interesting results.

In a final step, the diffusion process should be documented (*c.f.* [ALMN99]) and lessons learnt clearly identified. The desired output of this re-

search should be a document of guidelines that can serve as the basis for future diffusions.

3.4 Expected impact

Since the proposed diffusion follows a user-centric approach and builds upon the concepts of participatory action research, the probability of acceptance among developers is high. By providing a standardised Debian maintenance process compatible with existing processes, it will become easier to coordinate teams of developers to work around the bottleneck problems the Debian project is experiencing due to overloaded or inactive developers holding a monopoly on packages that use peculiar packaging approaches and are thus not accessible to others.

With an intuitive interface on the side of the developer, it will be easier to adjust processes to new policies or guidelines, and it will be possible to consolidate and integrate processes further for improved workflows. This should improve Debian's scalability. Furthermore, based on the lessons learnt from this research, future diffusions can be staged accordingly to yield higher success rates.

Furthermore, a standardised method will make it easier for one-time/infrequent contributors or non-technical contributors (such as translators) to donate their expertise without having to invest time into learning package-specific techniques or guidelines.

This research is relevant because it will help Debian scale and retain high technical standards. This will benefit not only the countless users, who rely on the technical strengths of the Debian operating system, but also to the more than a hundred derived distributions, which help to diversify the Linux distribution market and thus drive innovation.

Finally, results from this research will be useful to other projects facing the task of coordinating small teams on sub-projects characterised by contributor fluctuations typical to volunteer efforts. Quite possibly, the results could also influence business processes in companies with flat hierarchies.

4 Acknowledgements

I would like to thank Biella Coleman for her assistance in writing this proposal, and my supervisor, Prof. Brian Fitzgerald, for his feedback. Further thanks go to Martin Pool, Aaron Bentley, Nathaniel Smith, James Blackwell, and several others on the `#revctrl` and `#bzr` IRC channels (`irc.freenode.org`) for their input and interesting discussions. And, of course, all of Debian…

References

[AB02] Ulf Asklund and Lars Bendix. A study of configuration management in open source software. *IEEE Proceedings - Software*, 149(1):40–6, February 2002.

[ABCO98] Brad Appleton, Stephen P. Berczuk, Ralph Cabrera, and Robert Orenstein. Streamed lines: Branching patterns for parallel software development. In *Proceedings of the 5th Annual Conference on Pattern Languages of Program Design*, 1998.

[Ale01] Andrei Alexandrescu. *Modern C++ Design*. Addison-Wesley Professional, 1 edition, February 2001.

[ALMN99] David E. Avison, Francis Lau, Michael D. Myers, and Peter Axel Nielsen. Action research. *Commununications of the ACM*, 42(1):94–97, 1999.

[Ask02] Ulf Asklund. *Configuration Management for Distributed Development in an Integrated Environment*. PhD thesis, Department of Computer Science, Lund Institute of Technology, Lund University, Lund, Sweden, 2002.

[BA02] Stephen P. Berczuk and Brad Appleton. *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*. Addison-Wesley Professional, Boston, MA, USA, November 2002.

[Bas99] Richard L. Baskerville. Investigating information systems with action research. *Communications of the AIS*, 2(3es):4, 1999.

[BCHS05] Andreas Barth, Adam Di Carlo, Raphaël Hertzog, and Christian Schwarz. Debian developer's reference. http://www.debian.org/doc/developers-reference, August 2005. Last accessed: 26 July 2006.

[Bec99] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, Cambridge, MA, USA, 1st edition, October 1999.

[Ber91] H. Ronald Berlack. *Software Configuration Management*. Wiley Series in Software Engineering Practice. John Wiley & Sons, Hoboken, NJ, USA, September 1991.

[Bev95] Nigel Bevan. Measuring usability as quality of use. Technical report, National Physics Laboratory Usability Services, Teddington, England, 1995.

[Bro95] Frederick P. Brooks. *The Mythical Man Month: Essays on Software Engineering*. Addison-Wesley Professional, Boston, MA, USA, anniversary edition, February 1995.

[bts] Debian Bug Tracking System. http://bugs.debian.org. Last accessed: 31 July 2006.

[Col05] E. Gabriella Coleman. *The Social Construction of Freedom in Free and Open Source Software: Hackers, Ethics, and the Liberal Tradition*. PhD thesis, University of Chicago, Chicago, IL, USA, August 2005.

[Con03] Jeff Conklin. *Dialog Mapping: An Approach for Wicked Problems*. PhD thesis, CogNexus Institute, Napa, CA, USA, 2003.

[CSFP06] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. Version control with subversion. http://svnbook.red-bean.com/, 2006.

[Daf01] Georg N. Dafermos. Management and virtual decentralised networks: The linux project. *First Monday*, 6(11), November 2001. Last accessed: 30 August 2005.

[Dav93] Fred D. Davis. User acceptance of information technology: System characteristics, user perceptions and behavioral impacts. *International Journal of Man-Machine Studies*, 38:475–87, 1993.

[deb] The Debian project. http://debian.org. Last accessed: 31 July 2006.

[Deb05] The Debian Policy Manual. http://www.debian.org/doc/debian-policy, June 2005. Last accessed: 31 July 2006.

[DFS04] The Debian Free Software Guidelines. http://www.debian.org/social_contract#guidelines, April 2004.

[dmu05] Debian machine usage policies. http://www.debian.org/devel/dmup, August 2005. Last accessed: 31 July 2006.

[FB03] Karl F. Fogel and Moshe Bar. *Open Source Development with CVS*. Paraglyph Press, Phoenix, AZ, USA, July 2003.

[GBROP+04] Jesús M. González-Barahona, Gregorio Robles, Miguel Ortuño-Pérez, Luis Rodero-Merino, José Centeno-González, Vicente Matellón-Olivera, Eva Castro-Barbero, and Pedro de-las Heras-Quirós. Analyzing the anatomy of GNU/Linux distributions: Methodology and case studies (Red Hat and Debian). In Stefan Koch, editor, *Free/Open Source Software Development*, pages 27–58. Idea Group Publishing, Hershey, PA, USA, 2004.

[Gre88] Irene Greif. *Computer Support for Cooperative Work*. Morgan Kaufmann Publishers Inc., San Mateo, CA, USA, 1988.

[Her03] Raphaël Hertzog. Introducing Alioth: SourceForge for Debian. http://lists.debian.org/debian-devel-announce/2003/03/msg00024.html, March 2003. Mailing list post.

[Hes05] Joey Hess, September 2005. Personal communication.

[Hud04] Greg Hudson. Undiagnosing subversion. http://web.mit.edu/ghudson/thoughts/undiagnosing, January 2004. Last accessed: 2 September 2005.

[Ian05] Federico Iannacci. Coordination prrocesses in open source software development: The linux case study. Technical report, Department of Information Systems, London School of Economics, London, England, April 2005.

[Joe97] Gregor Joeris. Change management needs integrated process and configuration management. In Mehdi Jazayeri and Helmut Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pages 125–41, Heidelberg, Germany, 1997. Springer Verlag.

[KPH99] Karlheinz Kautz and Jan Pries-Heje. Systems development education and methodology adoption. *ACM SIGCPR Computer Personnel*, 20(3):6–26, July 1999.

[Kra04] Martin F. Krafft. Rebuilding packages on all architectures. http://lists.debian.org/debian-security/2004/09/msg00014.html, September 2004. Mailing list post.

[Kra05a] Martin F. Krafft. *The Debian System — Concepts and Techniques*. Open Source Press, Munich, Germany, June 2005.

[Kra05b] Martin F. Krafft. Thoughts on a new upload process. http://blog.madduck.net/debian/2005-08-11-rcs-uploads.html, August 2005. Last accessed: 31 July 2006.

[Kra06] Martin F. Krafft. Ubuntu and Debian. http://blog.madduck.net/debian/2006.05.24-ubuntu-and-debian, July 2006. Last accessed: 28 July 2006.

[Lor03] Tom Lord. Diagnosing Subversion. http://web.mit.edu/ghudson/thoughts/diagnosing, February 2003. Last accessed: 31 August 2005.

[MB91] Gary C. Moore and Izak Benbasat. Development of an instrument to measure the perceptions of adopting an information technology innovation. *Information Systems Research*, 2(3):192–222, September 1991.

[MFH02] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. Two case studies of open source software development: Apache and Mozilla. Technical report, Avaya Labs Research, eBuilt, Bell Laboratories, Lucent Technologies, Basking Ridge, NJ, USA, 2002.

[MH03] Martin Michlmayr and Benjamin Mako Hill. Quality and the reliance on individuals in free software projects. In *Proceedings of the 3rd Workshop on Open Source Software Engineering*, pages 105–9, Portland, OR, USA, 2003.

[Mic04] Martin Michlmayr. Managing volunteer activity in free software projects. In *Proceedings of the 2004 USENIX Annual Technical Conference, FREENIX Track*, pages 93–102, Boston, MA, USA, 2004.

[Mic05a] Martin Michlmayr. Quality improvement in volunteer free software projects: Exploring the impact of release management. In Marco Scotto and Giancarlo Succi, editors, *Proceedings of the First International Conference on Open Source Systems*, pages 309–10, Genova, Italy, July 2005.

[Mic05b] Martin Michlmayr. Software process maturity and the success of free software projects. In *Proceedings of the 7th Conference on Software Engineering, KKIO 2005*, Krakow, Poland, 2005. accepted.

[NT03] David M. Nichols and Michael B. Twidale. The usability of open source software. *First Monday*, 8(1), 2003. Last accessed: 31 August 2005.

[Poo04] Martin Pool. Bazaar-NG design. http://bazaar-ng.org/obsolete-docs/design.html, December 2004. Last accessed: 30 August 2005.

[Poo05] Martin Pool. Personal communication, September 2005.

[Ray99a] Eric S. Raymond. *The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, October 1999.

[Ray99b] Eric S. Raymond. The magic cauldron. In *The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*, chapter 3. O'Reilly & Associates, Inc., Sebastopol, CA, USA, October 1999.

[RC89] Sridhar A. Raghavan and Donald R. Chand. Diffusing software-engineering methods. *IEEE Software*, 6(4):81–90, 1989.

[Rem06] Scott James Remnant. No more source packages. h., June 2006. Last accessed: 28 July 2006.

[RGBM05] Gregorio Robles, Jesús M. González-Barahona, and Martin Michlmayr. Evolution of volunteer participation in libre software projects: Evidence from Debian. In Marco Scotto and Giancarlo Succi, editors, *Proceedings of the First International Conference on Open Source Systems*, pages 100–7, Genova, Italy, July 2005.

[Rog95] Everett M. Rogers. *Diffusion of Innovations*. Free Press, London, UK, 4 edition, 1995.

[RW73] Horst Rittel and Melvin Webber. Dilemmas in a general theory of planning. *Policy Sciences*, 4:155–69, 1973.

[sou] Sourceforge. http://www.sourceforge.net. Last accessed: 1 September 2005.

[Sri05] Manoj Srivastava. Debian package management with GNU Arch. http://arch.debian.org/arch/private/srivasta/, August 2005. Last accessed: 1 September 2005.

[Stu05] Matthias Stuermer. Open source community building. Master's thesis, University of Bern, Bern, Switzerland, March 2005.

[Tha05] John Thackara. *In the Bubble: Designing in a Complex World.* MIT Press, Cambridge, MA, USA, April 2005.

[Tow05] Anthony Towns. Launchpad. http://azure.humbug.org.au/ aj/blog/2005/09/04#2005-09-04-launchpad-freeness, September 2005. Last accessed: 4 September 2005.

[ubu] The Ubuntu project. http://www.ubuntu.com. Last accessed: 31 July 2006.

[WGL91] W. F. Whyte, D. J. Greenwood, and P. Lazes. Participatory action research: through practice to science in social research. In W. F. Whyte, editor, *Participatory Action Research*, pages 19–55. Sage, Newbury Park, CA, USA, 1991.

[Whe05] David A. Wheeler. Comments on open source software / free software (OSS/FS) software configuration management (SCM) systems. http://www.dwheeler.com/essays/scm.html, May 2005. last visited: 30 August 2005.