Analysis and description of an open source janitor project

by

Håkon Løvdal

Master Thesis in
Information and Communication Technology

Agder University College

Grimstad, June 13, 2006

Abstract

The objective of this study is to describe the inside and impact of the Linux Kernel Janitor Project. To describe and discuss how such janitor activity can be useful for others is also an objective. The Linux Kernel Janitor Project is a project defined to perform maintenance of the Linux kernel source, often taking on tasks that nobody else will be doing. The patches produced by the janitors have been analysed and some of the effects and properties of the work the project has carried out are described. Analysis show that janitor activity reduces the amount of code while still keeping the same functionality or improving it. The patches that are produced are kept in a backlog where typically 10-15% of them are replaced from one release to the next release. The process and rules/guidelines that the project uses for participation are described. Some of the participants of the Linux Kernel Janitor Project have been interviewed. Comparison with other open source projects that have some janitor activity has been performed.

Keywords: *Linux, kernel, janitor, software maintenance*

Preface

This thesis was done as part of the degree Master of Science at Agder University College, Faculty of Engineering and Science. It was supervised by Mikael Snaprud at Agder University College, Parastoo Mohagheghi at Norwegian University of Science and Technology, NTNU and Bruce Perens.

I would like to express my gratefulness for the help I have received from my supervisors, without whom I would have been completely lost. Thanks to all of the janitors that answered my questions, both on the mailing list and in the interview.

Grimstad, May 2006

Håkon Løvdal

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

During software evolution preventive maintenance is required to prevent declining quality and to compensate for increasing complexity according to Lehman's software evolution laws (see chapter 2.1.1 on page 14).

One project that in some way does this is the Linux Kernel Janitor Project [1] which states as its mission statement *"We go through the Linux kernel sources, doing code reviews, fixing up unmaintained code and doing other cleanups and API conversion. It is a good start to kernel hacking"*. So the Linux Kernel Janitor Project has both a quality improving purpose and a mentor purpose. This thesis has looked into both the mentor and quality aspects, although mainly at the quality aspects.

The Linux Kernel Janitor Project was started in 2001 and today, 5 years after the start it has become a strong and mature project. The concept of organizing janitor projects has not however spread significantly to other open source projects which is a bit surprising since I agree with Linux Weekly News's conclusion[2] that many other projects would likely benefit from it. Perhaps this thesis could inspire to a little increase.

A few other projects have however defined some tasks as Janitor Tasks. Although these projects do not have janitor activity as a separate sub-project, this thesis tries to compare them to the Linux Kernel Janitor Project when this is relevant. When searching for such projects at the start of writing this thesis two such was found, Wine [3] and Asterisk [4].

[1] Also sometimes referred to just as Kernel Janitors and abbreviated KJ.

[2] See chapter 2.3.2 on page 27.

[3] Wine is an open source implementation of the Windows API on top of X11 and Unix. `http://www.winehq.org/`

[4] Asterisk is an open source PBX (Private Branch eXchange). `http://www.asterisk.org/`

1.2 Research questions

The motivation for choosing to describe the Linux Kernel Janitor Project was partly that this is not done before as far as I can tell, and thus is a undiscovered white spot on the knowledge map. But also because I believe that this type of activity and organising could be useful for others, both for other open source project as well as for commercial development of proprietary products.

This leads to two main objectives, one of generally describing janitor activity and another of describing how this can be used by others. The research questions where thus defined to be:

How can a janitor project be characterised in terms of :

- participants
  - Who is participating and why?
- process (model)
  - How can the process be described?
- impact
  - What is the result of the janitor work? (product metrics)
- performance
  - What is the effect of the janitor work? (process/project metrics)
- tools used
  - What tools are used?
  - How important are they for the project?
- frequency of patches
  - How often are patches produced?
  - How many patches are produced?
- type of patches
  - What type of change does the patches represent?

What elements of the janitor project can be reused in other projects :

- Scalability issues
  - How large does a project have to be before starting a separate janitor subproject? Is it suitable for your hello world project?
- Determine aspects of an other project to evaluate the usefullness of a Janitor project
  - What generalisations can be made?
  - Which criteria are there for starting a janitor project?
- Cost benefit analysis
  - What are the benefits of janitor projects?
  - What are the costs of janitor projects?
- Quality requirements
  - What criteria are set for accepting patches in a janitor project?

1.3 Research methods

This study have used a combination of both qualitative and quantitative methods to gather data about the Linux Kernel Janitor Project. The outline is shown in Figure 1.1.

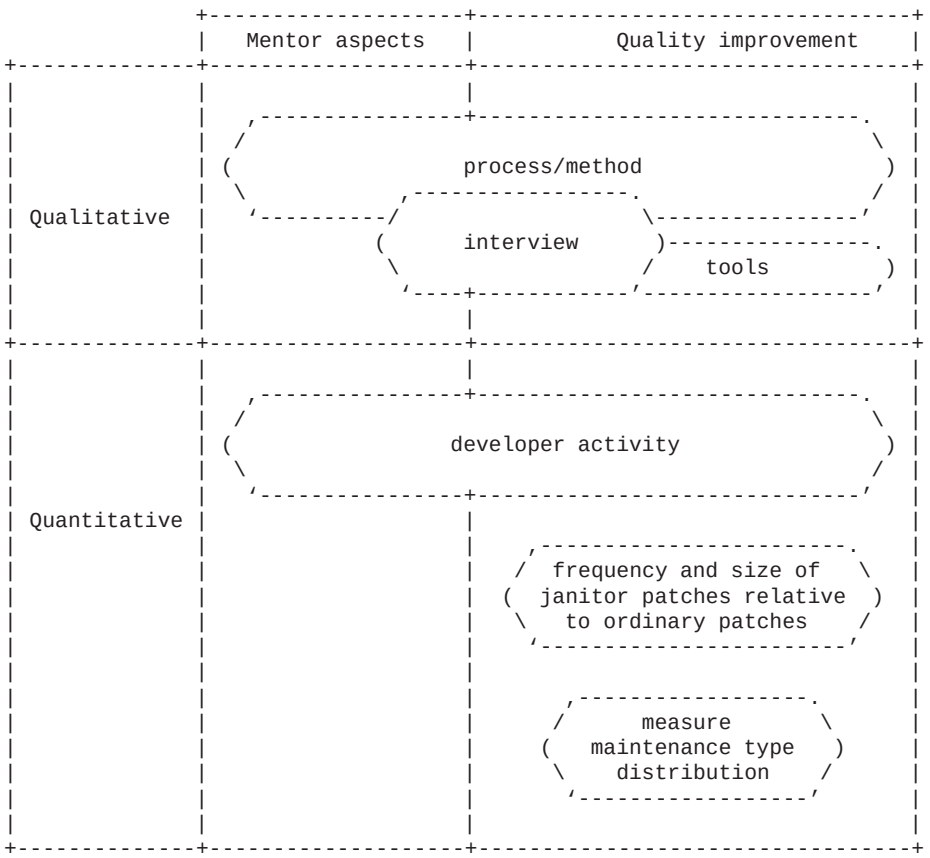


Figure 1.1: Research methods used and areas covered

The qualitative methods used included an examination of the recent history of the mailing list as well as the patches produced by the project in order to make description of the process and methods that make up a janitor project. This was done with respect to both mentor aspects and quality improving aspects. An inspection of the tools that the janitors used as sources for generating janitor task or to solve them was done. An interview was performed, asking both current janitor project participants and some participants that had participated earlier but no longer did. This interview was done to investigate the janitor process, mentor aspects and tools usage.

The mailing list was also analysed quantitative to get characteristics that could describe the janitor project. The patches that have been made was examined for properties like how often they are made and of the size. Changes can be classified in different types as described in chapter 2.1.6 on page 19, and such a classification of the patches was done on some of the patches.

1.4 Sources of information

The two main sources of information has been the archives of the Linux Kernel Janitor Project mailing list kernel-janitors@lists.osdl.org at `http://lists.osdl.org/pipermail/kernel-janitors/` as well as the ftp server storing the kernel janitor patchsets, `ftp://coderock.org/kj/`. Interviewing the janitor participants was also an important source of information.

Mailing list archives for Wine and Asterisk was examined. In addition information was fetched from the home page to the different projects

- `http://janitor.kernelnewbies.org/` for the Linux Kernel Janitor Project.
- `http://www.winehq.org/` for Wine.
- `http://www.asterisk.org/` for Asterisk.

The Kernel Traffic website, `http://www.kerneltraffic.org/`, contains summaries of the discusion on the main Linux kernel mailing list, LKML, and has been a very valuable source of general information about the Linux kernel development. Linux Weekly News has a section about the progress and status of the Linux kernel development in the weekly editions which has been used as an information source.

1.5 Limitations of scope of this thesis

While both Wine and Asterisk have some janitor activity it has not been very as easy to extract information about it, at least compared to the Linux Kernel Janitor Project. When information about Wine or Asterisk janitor activity hard to find, the effort has been put on the Linux Kernel Janitor Project since the main focus has been on that.

No economical evaluations of in what amount this is applicable for commercial environments has been done.

Only the current state has been examined. No attempt has been made in looking into how things were before compared to now.

For some aspects it would clearly have been intersting to compare the janitor activity with the ordinary development. However I choose to consentrate on only the janitor activity.

1.6 Report outline

Chapter 2 gives an technical and historical background for the rest of the thesis. Chapter 3 describes software development process in the context of Linux kernel develop-

ment and for the Linux Kernel Janitor, Wine and Asterisk projects.

Requirements and criteria used for the interview as well as summary of the responses are contained int chapter 4. In chapter 5 the results of the quantitative analysis of the patches produced by the Linux Kernel Janitor Project is presented. Discussion of the findings of this thesis and possible further work is done in chapter 6. Chapter 7 contains the conclusion.

Chapter 2

Theoretical Background and History

2.1 Programming and software engineering

2.1.1 Lehman's software evolution laws

The following two laws, which are selected from "the laws of software evolution" as specified by [1], are those that most directly relates to software maintenance.

No.	Brief Name	Law
II	Increasing Complexity	An E-type[1] system evolves its complexity increases unless work is done to maintain or reduce it.
VII	Declining Quality	The quality of E-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.

While some studies show that the development of the Linux kernel is quite optimal, [2]

> We have examined the growth of Linux over its six year lifespan using several metrics, and we have found that at the system level its growth has been superlinear. This strong growth rate seems surprising given ...

Other studies show that the problems described by the laws above also are affecting Linux. In [3] an architectural discovery and repair was performed on the Linux kernel where the authors achieved the following

> The repair actions narrowed the gap between the conceptual architecture and the concrete architecture from 502 anomalies down to 40.

[1] E-type software is software that is Embedded in a real-world environment.

2.1.2 Orders of ignorance

One particular interesting article about software development is The Laws of Software Process by Phillip G. Armour, [4] which has a telling observation

> In some circles, software process is considered to be *the* issue that needs to be resolved to fix "the software crisis." Improving process has become an article of faith in some corners, while avoiding it has assumed the status of guerrilla warfare in others.

The author states that

> Perhaps our problem isn't process, it's what we are asking process to do, and when and where we apply it.

He then formulates three laws of software process based on what he defines as "The Five Orders of Ignorance"

0OI - Lack of Ignorance. You know something.
1OI - Lack of Knowledge. You know that you do not know something.
2OI - Lack of Awareness. You do not know that you do not know something.
3OI - Lack of Process. You have no method of converting 2OI into either 1OI or 0OI.
4OI - Meta Ignorance. You do not know about the Five Orders of Ignorance.

0th Order Ignorance, 0OI, represents when you have the answer and 1OI represents that you have a well defined question that can be answered. For these two levels detailed, well defined processes work well. On the other hand, for 2OI a detailed process, based on some pre-existing knowledge which might or might not be relevant, does not make sense. The different levels of ignorance must therefore be handled differently.

> The challenge is all projects have different quantities of 0OI, 1OI, 2OI, and even 3OI, and therefore require different types of processes.

2.1.3 People more important than process

Some people argue that people are more important than process. "Peopleware: Productive Projects and Teams" by Tom DeMarco and Timothy Lister is a book about software management, often characterised as a classsic, which has a strong emphasis on people:

> The major problems of our work are not so much technological as sociological in nature.

DeMarco has also written other books about software development where he expresses a skepticism over a too strong focus on process:

> The danger of standard process is that people will miss chances to take important shortcuts[2]. – "Deadline"
>
> Process obsession is the problem. Process obsession is not just an anomaly that occurs now and again. It is an epidemic. – "Slack"

The Manifesto for Agile Software Development, which is a set of principles signed by the members of The Agile Alliance, states that *"we have come to value . . . Individuals and interactions over processes and tools"*. A literature review made for "Integration of human factors for user interfaces into the software development life cycle", [5] contains a separate chapter "8. People more important than process" which has several references to relevant literature.

The outermost variant of this view is that only people are important and that the process is neglectible.

> "Hell, there are no rules here – we're trying to accomplish something."
> – Thomas A. Edison

2.1.4 The Linux developers take at software development process

The Linux kernel developers are typically opposed to using industry standards for software development. This is not to say that the developers are working without guidelines or rules, but these are then rather made up by them selves and only when they feel that there is a need to.

> No major software project that has been successful in a general marketplace (as opposed to niches) has ever gone through those nice lifecycles they tell you about in CompSci classes. – Linus Torvalds

When the suggestion of using CMM came up on the main Linux kernel developer mailing list, the following was one of the replies:

> With SEI CMM level 3 for the kernel, complete testing and documentation, we'd be able to release a new kernel every 5 months, with new

[2] Refer to *Armour's observation on software process* (in [4]),

> What all software developers really want is a rigorous, ironclad, concrete, hidebound, absolute, total, definitive, and complete set of process rules they can break.

> drivers 2 years after release of the device, and support for new platforms 2-3 years after their availability, as opposed to 1-2 years before (IA-64, for instance...)
>
> We'd also kill off all the advantages that the bazaar-style development style actually has, while gaining nothing in particular, except for a slow machinery of paper-work. No thanks. – David Weinehall

I think this can be viewed as that the kernel developers are taking a bottom-up approach to defining the software development process while the industry standards are most certainly using a top-down approach.

2.1.5 Industry standards for software development

A process is a description what to do and how to do it. The IEEE standard 610, IEEE Standard Computer Dictionary, defines *process* and *software development process* as

process A sequence of steps performed for a given purpose.

software development process The process by which user needs are translated into a software product. The process involves translating user needs into software requirements, transforming the software requirements into design, implementing the design in code, testing the code, and sometimes, installing and checking out the software for operational use.

Software development process is alternatively also called *software engineering process*, *software life cycle* or just *software process*. The IEEE and ISO standards for this are "IEEE/EIA 12207.0-1997 Standard for Information Technology – Software Life Cycle Processes" and "ISO/IEC 12207 Information Technology – Software Life-Cycle Processes". These standards are quite high level framework and has to be accompanied with more detailed process models or methodologies.

Exactly what constitutes a process/model/methodology/approach/framework/mechanism/recommendation/discipline/practice/method/etc is not a clear cut[3] and I will not try to draw any borders. The following is a brief list of some — well, methodologies or whatever they are. Many of these are heavily tailored to fit projects whose developers are full time workers at the same location. This does not coincide very well with the great diversity in geography, time and stake in the open source development of Linux. See [6] for an excellent overview and comparison of some of the differences between various processes and methods.

[3] For instance the Wikipedia article about Extreme Programming, [7] says that *"Extreme Programming (XP) is a software engineering methodology for the development of software projects."* while Ron Jeffries (one of the creators of XP) does not call it a methodlogy:

agile development
http://agilealliance.org/,
http://en.wikipedia.org/wiki/Agile_software_development

CMM- The Capability Maturity Model
http://www.sei.cmu.edu/cmmi/,
http://en.wikipedia.org/wiki/Capability_Maturity_Model

Crystal family
http://www.arches.uga.edu/~cjupin/,
http://agile.csc.ncsu.edu/crystal.html

DSDM - Dynamic Systems Development Method.
http://www.dsdm.org/,
http://en.wikipedia.org/wiki/Dynamic_Systems_Development_Method

iterative development
http://en.wikipedia.org/wiki/Iterative_and_incremental_development

PSP - Personal Software Process
http://www.sei.cmu.edu/tsp/,
http://en.wikipedia.org/wiki/Personal_Software_Process

RUP- Rational Unified Process
http://www-106.ibm.com/developerworks/rational/library/content/RationalEdge/jan01/WhatIstheRationalUnifiedProcessJan01.pdf
http://en.wikipedia.org/wiki/Rational_Unified_Process

TSP - Team Software Process
http://www.sei.cmu.edu/tsp/

waterfall[4]
http://en.wikipedia.org/wiki/Waterfall_model

XP- eXtreme Programming
http://en.wikipedia.org/wiki/Extreme_Programming

SCRUM
http://en.wikipedia.org/wiki/Scrum_%28management%29,
http://en.wikipedia.org/wiki/Scrum_%28development%29,

Yes, except that I wouldn't even call XP a method or methodology.

XP is a community of software development practice which has grown up around a particular combination of ideas (values, practices, principles) which Kent Beck named Extreme Programming.

XP is a way of approaching doing software development. It is not a recipe, nor a formula. It is not a methodology as I understand the word, and frankly I'm not sure what other things WTH would call methodologies. Neither CMM nor RUP are methodologies. What things can we name that are methodologies?

http://www.controlchaos.com/about/

2.1.6 Classification of maintenance types

The ISO standard for software maintenance, *ISO/IEC 12207, Information technology - Software life cycle processes*, has the following classification of maintenance types.

- Adaptive
- Corrective
- Perfective
- Preventive

Adaptive maintenance is maintenance done as a response to environment changes. This might for instance be a new compiler, a new version of an external library or updates to the operating system. *Corrective maintenance* means correcting some imperfect behavior (also known as fixing bugs). When functionality is added, changed or removed this is called *perfective maintenance*. *Preventive maintenance* is non-functional changes with the purpose of improving later maintenance. For instance a large function is split into smaller pieces without changing the overall functionality or a self-made sort routine might be replaced with a standard library sort function.

The IEEE Standard for Software Maintenance, IEEE Std 1219, uses the following classification of maintenance types:

- Corrective
- Adaptive
- Perfective
- Emergency

The three maintenance types corrective, perfective and adaptive was first described in [8] by Swanson in 1976. The last type, preventive maintenance was termed in [9] by Pressman in 1987 [5].

Critics of these classifications includes [11] in that the classification *"depends on the reason for the change, and not on an objective characteristic of the change"*. One proposed alternative to the traditional Swanson + Pressman classification is presented in [12] where the authors keeps corrective and adaptive but then proposes a enhancement category with several sub-categories.

- corrective

[4] It is worth noting that the waterfall model in the original paper was presented as a poor model, one that "is risky and invites failure". See http://tarmo.fi/blog/2005/09/09/dont-draw-diagrams-of-wrong-practices-or-why-people-still-believe-in-the-waterfall-model/ for more discussion of why the model nevertheless gained popularity.

[5] Pressman notes that the preventive maintenance approach was first reported as "structured retrofit" by Miller in 1981. See chapter "2.2.5 An explorative definition of software maintenance" in [10] for more details on the history.

- adaptive
- enhancement
  - Data Handling
  - Control flow
  - User Interface
  - Computation
  - Module Interface
  - Initialization

2.1.7 Existing studies of distribution of maintenance types

A study of the distribution of maintenance types in the Linux kernel has been done before in [13]. However this study did not include preventive maintenance, only the tree other types.

This study included kernel versions from version 1.0 through version 2.3.51, which is from a time when the developed was done on two parallel branches. Today the development model for the 2.6 kernel is different and uses only one branch (see chapter 2.2.4 on page 23). This was also done before Linus started using source code control tools which both has increased productivity and partly changed the way of working. Undoubtfully, the development process (as well as many of the developers) is today much more mature than it was around version 1.0.

So while not certain, these are factors that might change the results if the same study had been redone on todays code, and I think that would have been very interesting. However this will probably be a considerable amount of work, the authors of [13] used over 9 months classifying 391 versions of the Linux kernel.

2.1.8 Mentoring

The Wikipedia article, [14] has the following description of mentoring: *a trusted friend, counselor or teacher, usually a more experienced person. Some professions have "mentoring programs" in which newcomers are paired with more experienced people in order to obtain good examples and advice as they advance, and schools sometimes have mentoring programs for new students or students who are having difficulties*.

The origin to the modern use of the word is from Greek mythology. Mentor was a person who was left in charge of the son of a friend that left for war. When the goddess Athena visited the son, she took form of Mentor and helped the son and gave him advise.

Mentoring is generally strongly reccomended both in technical and non-technical professions, see for instance [15] and [16]. Drawbacks of mentoring is that is can be time

consuming for the mentor, [17].

2.2 Linux kernel development

2.2.1 What is Linux?

Since the Linux Kernel Janitor Project is a part of development of the Linux kernel some information about Linux is is appropriate. However the purpose is not to give a complete listing of all the benefits or features of the Linux kernel, only a brief description is given.

Linux is an unix-like operating system that Linus Torvalds started writing for his own amusement and hobby in 1991. Linux is free for anyone to use, modify and distribute. The development of Linux is open so that anyone can contribute[6]. The development is lead by Linus Torvalds.

2.2.2 Who is developing the kernel?

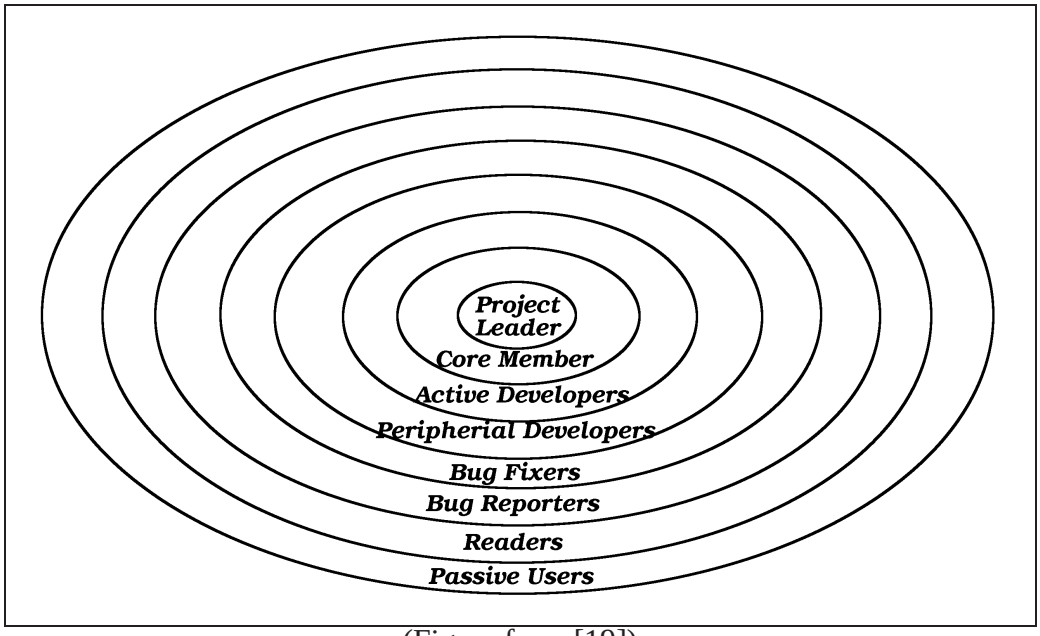
In a recent interview, [18], Linus Torvalds said the following about the people he communicate with.

> I actually only work with a few handfuls so I tend to directly interact with maybe 10 - 20 people and they in turn interact with other people. So depending on how you count, if you count just the core people, 20 -50 people. If you count everybody who's involved; five thousand people – and you can really put the number anywhere in between... Almost, pretty much all, real work is done over e-mail so it doesn't matter where people are.

One of the studies that have been performed on open source projects is [19], which provides a model for how the community is divided into different hierarchical groups as shown in Figure 2.1 on the next page. For Linux it is Linus Torvalds that is the Project leader, but the division into the other groups are theoretical and there are no clear borders, illustrated quite clearly by the fact that the project leader himself cannot give an exact number of how many developers there are.

An similar model is presented in [20] which is shown in Figure 2.2 on page 23. This model has split tasks and users, but overall it is quite similar to the one in Figure 2.1.

[6] Although there is no guarantee that your changes will be accepted.



(Figure from [19])

Figure 2.1: General structure of an open source community

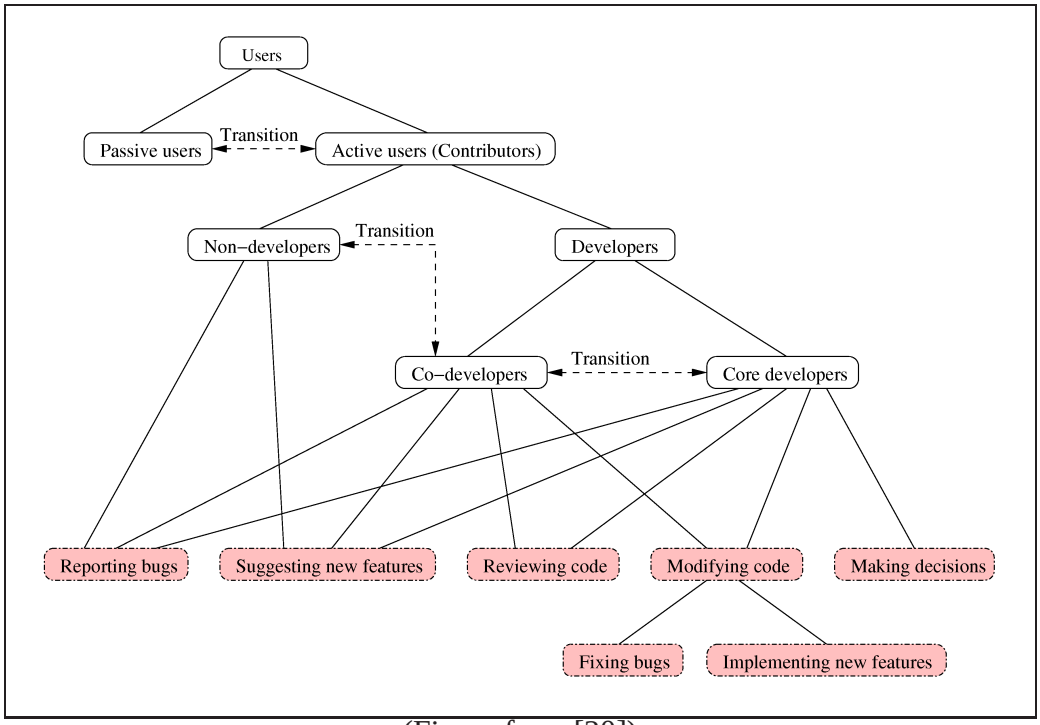
2.2.3 Patches - the heartbeat of the Linux kernel development

Since the beginning of the collaborative development of the Linux kernel, private email/mailinglists/news groups has been the main communication channel between the developers. Distributing complete source code trees with modifications to share development effort is not practical, so the way the developers share their work is to distribute *changes* to a given base of the source code. Such a change is expressed as a patch. A patch is typically generated with the `diff` program, using the unified format.

Figure 2.3 on the next page shows an example of a patch where a hardcoded constant is replaced with a preprocessor token. Such a patch is easily distributed via email/news. As an extra feature, if the patch is inserted in-line into the body of the email/post (i.e. not as an attachment) it is easy for other developers to discuss the patch and insert their comments in between the quoted lines or making an extract when replying. See Figure 2.4 on page 24 for an example of this.

Actually just calling this an extra feature is too weak because this is a very essential property of a patch. Patches might be rejected on the sole basis of being sent as an attachment instead of inserted in-line.

Without patches being sent between the Linux developers the development would stop completely (and here git (see chapter 2.2.5 on page 26) is just considered to be a special packaging of patches). So patches are the heartbeat of the Linux kernel development.



(Figure from [20])

Figure 2.2: The classification of open source users and developers

```
--- linux-2.6.15-git9/drivers/atm/lanai.c  2006-01-13 18:21:22.000000000 +0100
+++ linux-2.6.15-git9_patched/drivers/atm/lanai.c  2006-01-13 18:24:36.000000000 +0100
@@ -1972,7 +1972,7 @@
                    "(itf %d): No suitable DMA available.\n", lanai->number);
                return -EBUSY;
        }
-       if (pci_set_consistent_dma_mask(pci, 0xFFFFFFFF) != 0) {
+       if (pci_set_consistent_dma_mask(pci, DMA_32BIT_MASK) != 0) {
                printk(KERN_WARNING DEV_LABEL
                    "(itf %d): No suitable DMA available.\n", lanai->number);
                return -EBUSY;
```

In the unified format the lines prefixed with minus are removed compared to the original, while lines prefixed with plus are added.

Figure 2.3: Example of a patch

2.2.4 Linux development branches

After release 1.0 of the Linux kernel the development was split onto two tracks, or branches, *stable* and *development*. The development releases was numbered 1.1.x[7]

[7] where x is a sequence number

```
On 2006-01-04 at 18:18:19 +0100, Matthew Wilcox <matthew@wil.cx> wrote:
> On Wed, Jan 04, 2006 at 06:01:19PM +0100, Tobias Klauser wrote:
> > diff -urpN -X dontdiff linux-2.6.15/mm/slab.c linux-2.6.15~tk/mm/slab.c
> > --- linux-2.6.15/mm/slab.c  2006-01-03 14:41:57.000000000 +0100
> > +++ linux-2.6.15~tk/mm/slab.c       2006-01-04 15:52:41.000000000 +0100
> > @@ -921,7 +921,6 @@ static int __devinit cpuup_callback(stru
> >             down(&cache_chain_sem);
> >
> >             list_for_each_entry(cachep, &cache_chain, next) {
> > -                   struct array_cache *nc;
> >                     cpumask_t mask;
> >
> >                     mask = node_to_cpumask(node);
>
> While this does work, it's quite bad style.  Much better to move the
> upper level declaration (line 856) into the block it's used in (line
> 893).  BTW, that function is too big at 133 lines and should be split.

Sure. That makes more sense. I'll send a new patch.

Thanks, Tobias
```

Figure 2.4: Example of discussion of a patch

and these was intended to be the playground where the developers could go wild and work on all their new and exciting features, in support of future world domination[8]. The stable releases had release numbers 1.0.x and was meant to be for general use by "normal" users, where features were frosen and only bugfixes should be done.

A split between a stable and a development branch is a very common strategy in open source projects, although some use more[9]. The release numbers used is then also often using a odd/even scheme on the form `a.b.c` where `a` is a major release number, `b` is even for stable and odd for development and `c` is a sequence release number (sometimes an additional `d` is also used).

The end of the 1.1 development of the Linux kernel resulted in a stable 2.0 branch. This was since followed by 2.1, 2.2, 2.3, 2.4, 2.5 branches up till 2.6. Linus Torvalds decided that he did not want to have a 2.7 development branch, and 2.6 is now working as a combination of both stable and development. The version number has been extended to support -*stable releases* numbered 2.6.x.y. From the Documentation/HOWTO file in the Linux source:

> 2.6.x.y -stable kernel tree
>
> Kernels with 4 digit versions are -stable kernels. They contain relatively small and critical fixes for security problems or significant regressions dis-

[8]The phrase "world domination" is a self-ironic phrase used in the Linux community. See [21] for a discussion of the origin.

[9]For instance, the XEmacs development is split between three branches: stable, gamma, and beta. Debian uses experimental, unstable, testing and stable.

> covered in a given 2.6.x kernel.
>
> This is the recommended branch for users who want the most recent stable kernel and are not interested in helping test development/experimental versions.
>
> If no 2.6.x.y kernel is available, then the highest numbered 2.6.x kernel is the current stable kernel.
>
> 2.6.x.y are maintained by the "stable" team <stable@kernel.org>, are released almost every week.
>
> The file Documentation/stable_kernel_rules.txt in the kernel tree documents what kinds of changes are acceptable for the -stable tree, and how the release process works.

In addition to the official source tree produced by Linus Torvalds, some of the other kernel developers provide their own variations of the kernel source. These are often given a suffix with the initials of the developer[10], for instance `"-ac"` from Alan Cox [11]. The currently most important tree of those is the `"-mm tree"` which now has taken over for the previously development branch.

> But there needed to be a mechanism for testing new technologies, a place where they could be revised, updated and even removed before actually getting into the mainline kernel. As such, it was decided that the -mm tree would be the place where things were tested before they got into the mainline kernel. – Greg Kroah-Hartman

> The -mm patches are a set of patches, released by Andrew Morton, against the official kernel series. They are frequently more experimental in nature than the official series. – `http://kernel.org/patchtypes/mm.html`

After Linus Torvalds has made a 2.6.x release, there is a two week period where Linus is open for accepting development patches. After that period he creates the first release candidate for the next kernel release, 2.6.(x+1)-rc1. After rc1 only bugfixes will be accepted. A number of following release candidate releases are made until the kernel is perceived good enough to make the 2.6.(x+1) release. See Figure 2.5 on the following page for connection between the kernel branches relevant for this thesis.

[10] `http://www.kernel.org/git/` lists 149 different git repositories with the path linux/kernel/git/....

[11] For the 2.4 development the -ac tree had a somewhat similar function to the current -mm tree, being slightly more experimental than the development tree. The Linux Kernel Janitor Project started submitting some of the patches through the -ac tree.

2.2.5 Version Control System

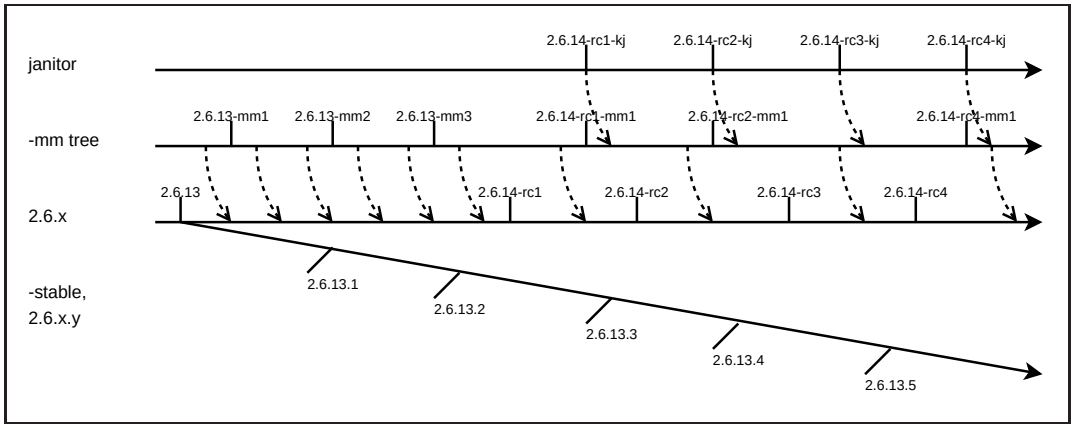
Up till 2002 patches was the only way to distribute changes. In January 2002 Linus Torvalds decided to start using a version control system called BitKeeper [22]. The move to start using a version control system was not controversial, but the choice of BitKeeper was since this was a commercial product (which was made freely available for Linux kernel developers).

Despite the opposition Linus decided that he wanted to try. This was after all an optional addition/alternative to the existing development process; patches in mail would still be acceptable and used. Linus was satisfied with that BitKeeper made him more effective and kept using it.

BitMover, Inc., the company producing BitKeeper, issued a press release in 2004[12] claiming that Linus Torvalds had more than doubled his productivity. This claim was supported with quantitative measurements of the activity before and after Linus Torvalds started using BitKeeper. A more in depth coverage of this is present in [23].

In 2005 the usage terms for BitKeeper changed and Linus decided that he wanted to write his own version control system[24]. He found none of the existing free version control systems suitable, but after using BitKeeper for three years he knew quite well

[12] http://www.bitkeeper.com/press/2004-03-17.html



The dotted arrows indicate merge from one branch to another, but the exact placement and number are guesses. The merges are not full merges, only selected parts. So when Linus Torvalds merges from -mm between 2.6.13 and 2.6.14-rc1 he is likely to accept janitor patches. After the rc1 release they are not accepted (unless they fix bugs which in case they might be considered).
(Pay little attention to the time scale in the figure)

Figure 2.5: Different Linux kernel branches and releases

how he wanted a version control system to be. This was the start of the tool called `git`. Since version 2.6.12-rc3 the kernel has been developed and distributed using git.

2.3 The Linux Kernel Janitor Project

2.3.1 Origins of the term janitor

The Wikipedia article about janitor, [25] has the following description

> A janitor is a person who takes care of a building, such as a school, office building, or apartment block. They are responsible primarily for cleaning, and often (though not always) some maintenance and security.

and notes that the origin *is derived from the Latin word ianitor meaning "doorkeeper". A female janitor is called a janitrix, although this term is rarely used.*

2.3.2 History

The Linux Kernel Janitor Project was started in 2001 by Arnaldo Carvalho de Melo. He was maintaining a TODO list for things to fix or clean up. The list was available via HTTP and he noticed in the webserver logs that many people accessed it. He therefore decided to organize the janitor activity into its own project.

This form of janitor organization appears to be relative new. In the March 29, 2001 edition of Linux Weekly News, the newly started Linux Kernel Janitor Project was the main coverage with the following closing words:

> The kernel, meanwhile, is far from the only large development project in the free software community. No doubt, many other projects should look at the kernel janitors organization and consider setting up something similar. The benefits, in terms of improved code and a better supply of new hackers, could be both large and immediate.

Given the high quality of the content of Linux Weekly News and their effort of providing a complete and full picture in their news coverage this is a strong indication that no other projects similar to Linux Kernel Janitor Project existed at that time.

The janitor concept was partly inspired from FreeBSD (although this was not organized as an separate activity or called janitor). In an old irc log, archived at the Linux Kernel Janitor Project website, Dave Jones said the following:

> An interesting parallel to the kernel janitor project is what happens with the FreeBSD folks. Regularly, you'll see on their mailing list "junior kernel hacker tasks". Simple (but often tedious) cleanups. More experienced hackers will mentor newcomers to kernel hacking, often pointing them in the correct direction.

Linux Weekly News also drew parallels to OpenBSD:

> And the janitors have noted an important point: an error pattern that is found in one section of code has a high likelihood of recurring in other places. Once a particular type of mistake has been found, it makes great sense to go looking for instances of the same mistake elsewhere. This is essentially the same approach as that used by the OpenBSD team to root out security problems before they are exploited.

2.3.3 Janitor Patchsets

The patches from the Linux Kernel Janitor Project are released in patch sets, also called *patchsets*. A patchset is a collection of files that each contains a single patch. These patches have first been posted to the mailing list where they are possibly discussed and modified. The project leader reviews mailing list patches from time to time and add those that are accepted to the patchset. This process is described in chapter 3.1 and also shown in Figure 3.1 on page 30.

The first Linux Kernel Janitor Project patchset was 2.5.70-bk13. Before that the janitor patches was handled individually by each author. Some of them were sent to and handled by the Trivial Patch Monkey (see chapter 2.3.4 on the following page).

Released by	From	Date	To	Date
Randy Dunlap	2.5.70-bk13	2003-06-11	2.6.6-rc2-kj1	2004-04-23
Maximilian Attems	2.6.7-rc1-kjt1	2004-05-24	2.6.10-rc2-kjt1	2004-11-20
Domen Puncer	2.6.10-kj	2004-12-24	2.6.13-rc4-kj	2005-07-29
Alexey Dobriyan	2.6.13-git4-kj1	2005-09-03	...	

Figure 2.6: Earlier and current patchsets

[13] It seems that `http://www.osdl.org/archive/rddunlap/kj-patches/...` and `http://developer.osdl.org/rddunlap/kj-patches/...` was identical (they are no longer available). The first patchset release was announced as using www.osdl.org while all the following releases was using developer.osdl.org.

The internet archive, `http://www.archive.org/` has nothing stored of `http://developer.osdl.org/rddunlap/` while the last entry for `http://www.osdl.org/archive/rddunlap/` contains patchsets up till including 2.6.6-rc2-kj1 which was announced as `http://developer.osdl.org/rddunlap/kj-patches/2.6.6-rc2/2.6.6-rc2-kj1.patch.bz2`.

2.3.4 The Trivial Patch Monkey

Although the Trivial Patch Monkey has no direct connection to Linux Kernel Janitor Project there might be some overlap in the work done, so a brief description is included. The Trivial Patch Monkey collects and submits patches that are trivial[14], taking care of re-submitting and following up so that the patch will not be lost. It was started by Rusty Russell which in February 2002 announced

> Hi all, trivial@rustcorp.com.au is set up to take trivial patches, ie. one-liner, documentation, spelling fix, etc. I will acknowledge your patch, and take care of the retransmissions until the patch is either applied, or does not apply any more.
>
> The aim is to encourage people to submit minor tweaks without fear of them getting lost. Do not expect real time behavior: I am not a bot.

Later in May 2002 he wrote the following.

> With the recent flurry of inclusions, the trivial@rustcorp.com.au Trivial Patch Monkey has passed 100 patches which have filtered into the various kernels . . . With this surprising success (I thought the damn thing would die after a few days), I will be continuing to provide the service, which only takes me about an hour a week.

Late 2005 the Trivial Patch Monkey occupation was handed over to another Linux kernel developer, Adrian Bunk, and the email address is now trivial@kernel.org.

[14] "If you aren't sure whether a patch is trivial, it most likely isn't..."

Repository	From	To
http://www.osdl.org/archive/rddunlap/kj-patches/[13]	2.5.70-bk13	2.6.6-rc2-kj1
http://debian.stro.at/kjt/	2.6.7-rc1-kjt1	2.6.10-rc2-kjt1
ftp://coderock.org/kj/	2.6.10-kj	. . .

Figure 2.7: Earlier and current patchset repositories

Chapter 3

Software development process

3.1 Kernel Janitor Process

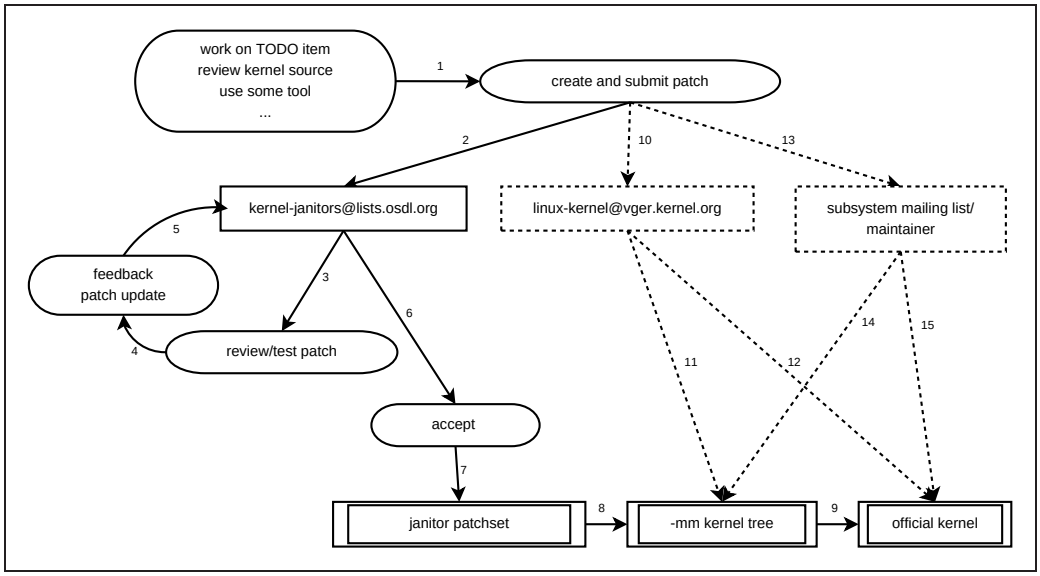


Figure 3.1: Process for submitting patches in the Linux Kernel Janitor Project

As with the normal Linux kernel development, the way to contribute is by creating patches. A patch is created either based on one of the TODO items or it is something the developer creates by his or her own initiative (arrow 1 in Figure 3.1). After creating the patch it is sent to the Linux Kernel Janitor Project mailing list for review (arrow 2). If the patch modifies a part of the kernel that has an active maintainer and/or separate mailing list it is polite to send a copy to them (arrow 13). In that case the maintainer will often accept the patch, “stealing” it out of the janitor process (arrow 14 or 15).

If the patch is special (say improves performance with 200%, has security implications

or something like that) it might be appropriate to send a copy to LKML[1], which is the main mailing list for the Linux kernel developers. You might try to send to Linus Torvalds directly as well, although this will have low chance of success.

> "Don't send to Linus" is pretty much the first non-obvious thing I've learned about process. – Alexey Dobriyan
>
> In fact Linus is fairly random at patches at the best of times. Generally, Linus will cc: it to me because he knows I'll pick it up. – Andrew Morton

After posted to the mailing list, the other janitors read and possibly test it. If they have any feedback (like in Figure 2.4 on page 24) they will post that and the patch creator will make an updated patch (arrow 5).

After some time the Linux Kernel Janitor Project leader will review the mailing list for patches posted (i.e. after the possible discussions have settled), and accept those that are found acceptable (arrow 6). Those patches are then included in the janitor patchset (arrow 7). Over time parts of the Linux Kernel Janitor Project patchset will be included in the -mm tree (arrow 8) and will most likely finally end up in the official kernel (arrow 9).

3.1.1 Linux Kernel Janitor Project mailing list activity

Figure 3.2 on the next page shows a plot of how persons come and go as participants on the Linux Kernel Janitor Project mailing list. For each month two lists of emails were made, one containing all the email addresses that occurred for the first time that month, and one list that contained the email addresses that posted for the last time that month. Counting these list gives quantitative measurements of how the number of participants on the mailing list change over time.

One evident characteristic of the plot in Figure 3.2 on the following page is the spikes where in one month there is a steep increase in the number of new mailing list participants followed by a correspondingly steep decrease the following month or couple of months. The same pattern is mirrored by the last time posted curve with one month delay compared to first time posted.

This is caused by mailing list threads that either at some point included other mailing list or maintainers or that threads elsewhere from started including the Linux Kernel Janitor Project mailing list. For July 2004 the spike is mostly triggered by threads about "replace schedule_timeout() with msleep()", about IO-APIC debug and min/max macros.

[1] linux-kernel@vger.kernel.org

3.1.2 Tools used by the Linux Kernel Janitor Project

In the beginning when defining the thesis and figuring out what to write about I wanted to include tools since they are used by the janitors and I think they are important. However when working with the thesis I found that the effect of the different tools used, in the context of the process of the Linux Kernel Janitor Project, was only to contribute to the list of possible work items.

This is not to downplay the importance of tools, but rather an expression of that the janitor process is rather independent of the tools used.

Why are tools important?

If a developer reads a source code file from start to end scanning for bugs/improvements this is a one time effort that is valid for that given version of the file only. Manually reading source code is very time consuming and, depending on what the objective for the examination is, might require a high degree of technical knowledge and understanding of the source code.

Tools will of course never be able to replace the overall all-aspects competence of humans, but tools have the benefit of taking a very short time to run (at least compared to human effort). So using a tool on all version is normally quite feasible.

A specialised tool might additionally find errors that go undetected by humans. And they do. Coverity recently ran a project analysing several open source projects, among others the Linux kernel. The results are available at `http://scan.coverity.`

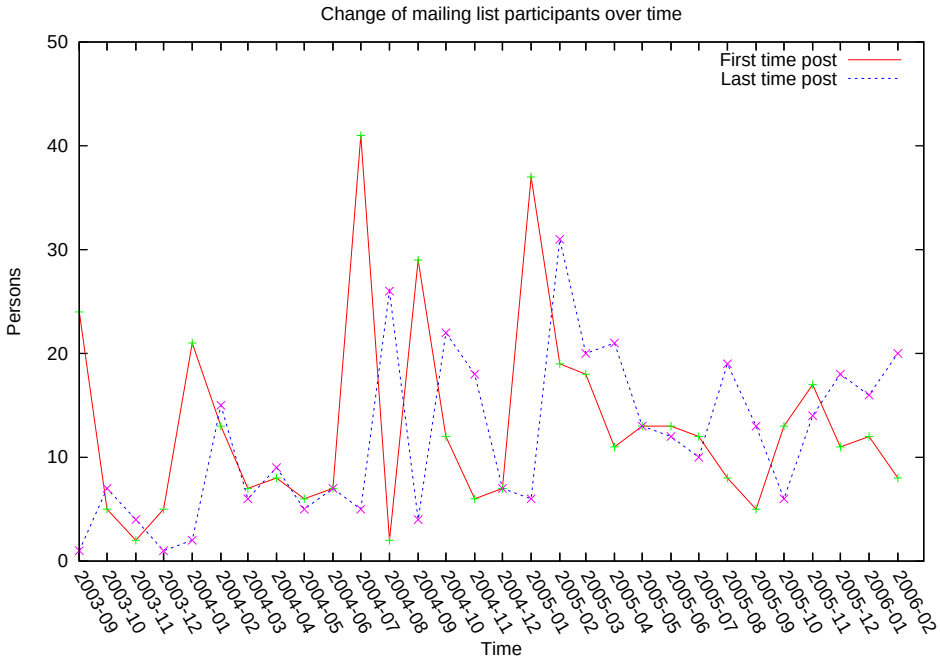


Figure 3.2: Change of mailing list participants over time

com/ and the report from the project is discussed on the front page of Linux Weekly News, March 9, 2006 edition [2].

Linus Torvalds has also written specific a tool for source code analysis of the kernel, sparse, which is used to find bugs in the handling of user-space pointers.

3.2 Wine Development Process

The process for Wine is in several ways similar to the one for the Linux Kernel Janitor Project. A detailed drawing of the different elements are shown in Figure 3.3 on the next page. The main differences are the following.

- There is no separate handling of janitor patches, they are handled together with the patches from the normal development.
- There are no different kernel trees, just one common git repository[3].
- Wine has one separate mailing list for submitting patches and only that, wine-patches@winehq.org. If there are any discussion this will take place on the wine-devel@winehq.org list.
- After a patch is committed in the git repository a message is sent to the read-only wine-cvs@winehq.org list.

After working on either a janitor task or one of the normal Wine tasks the developer creates a patch and sends it to the wine-patches mailing list (arrows 1 and 2 in Figure 3.3 on the following page). This list is read by the Wine developers which review the patches submitted (arrow 3), and if they have any feedback this will be posted on the wine-devel mailing list (arrow 4 and 5). The patch creator will then make an update of the patch (arrow 6) and resend it to the wine-patches list (arrow 7). If the patch is accepted it will be committed to the git repository (arrow 8 and 9). Accepted patches will appear in the wine-cvs mailing list (arrow 10).

3.3 Asterisk Development Process

Asterisk turned out to be quite different from both the Linux Kernel Janitor Project and Wine. Asterisk was created by Mark Spencer which is still the primary maintainer. He later founded the company Digium which is an telecommunications supplier and is the main sponsor of the development of Asterisk [4].

[2] http://lwn.net/Articles/174125/

[3] Wine started last year using git for source code management, but the code is also available through CVS, Subversion and SVK repositories which are synchronised with the git repository. Alexandre Julliard (who is Wine's project leader) is the only person with commit access to the git repository.

[4] Not everyone is happy with the strong influence that Digium has over the development of Asterisk or with the monolithic architecture. In 2005 a fork of the Asterisk code was made and a new project OpenPBX was started.

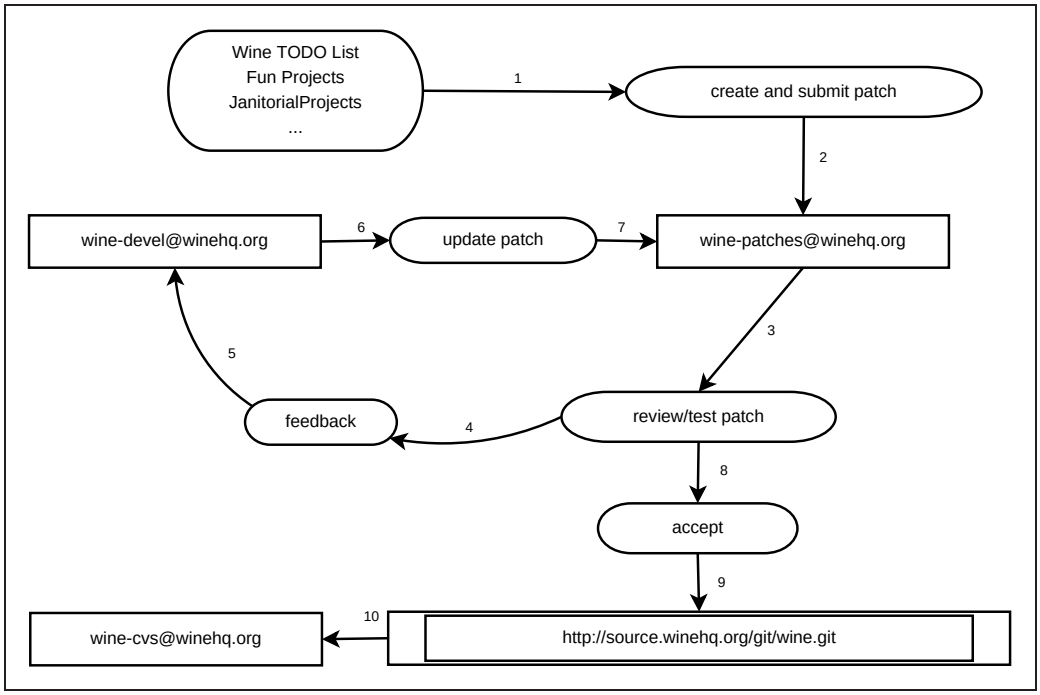


Figure 3.3: Process for submitting patches in Wine

The source code of Asterisk is stored in subversion and the servers are hosted by Digium. In order to contribute to Asterisk you have to fill out a copyright disclaimer and send or fax it to Digium. Asterisk is released under dual license scheme using both GPL [5] and a commercial license which Digium uses in products it sell.

Figure 3.4 on the next page shows the process for how to contribute to Asterisk. As mentioned in the previous paragraph, a copyright disclaimer is required to start contributing to the development of Asterisk (arrows 1, 2, 3 and 4). After finishing working with a task the change is committed with subversion (arrow 5) and placed directly in the subversion repository (arrow 6). The developers will monitor the repository for new functionality (arrow 7).

3.4 Summary

The largest difference between the Linux Kernel Janitor Project on one side and Wine and Asterisk on the other side is that the the janitor activity in the Linux Kernel Janitor Project is active and selfsupporting while for Wine and Asterisk it is more passive and a lesser part of the normal development. This is not to say that there is no janitor work done in Wine or Asterisk however.

[5] GNU General Public License. See `http://www.gnu.org/licenses/gpl.html` and `http://en.wikipedia.org/wiki/GPL` for more information.

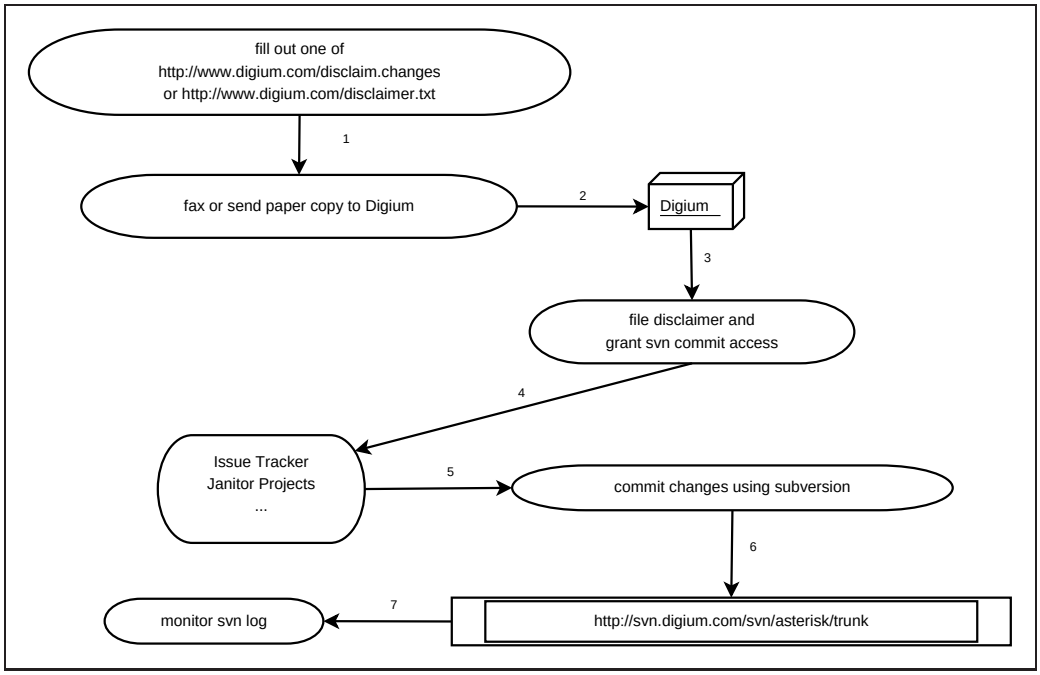


Figure 3.4: Process for contributing code in Asterisk

The process for submitting patches is quite similar for Wine and the Linux Kernel Janitor Project where in both projects patches are first sent to a mailing list for review. A committer will later merge the patches that are accepted into the projects repository. This is different from Asterisk which has a steeper entry point for starting contributing but where it is easier to contribute features or fixes when started.

Chapter 4

Interviews

4.1 Selection of interview participants

When examining a population it is usually not feasibly to ask every member. Therefore a selection is made and this subset of the population members is then examined. To obtain valid results this selection should be representative for the whole population. The representativeness is the key factor here; if this this is weak then the validity of whole outcome is also weak.

There are several ways of making a selection. Either by using *probability sampling* or *non-probability sampling*. In probability sampling there is a probability associated with the selection of each participant. In non-probability sampling there is a subjective decision involved in selecting each participant.

One particular, important non-probability factor is *self-selection*. If the sample is made out of population members that themselves decides to participate, this will introduce a bias (for instance are people with strong opinions more likely to participate). Self-selection is best avoided.

But even if the participants are carefully chosen one way or another, the response rate will normally be lower than 100%. This introduces a self-selection factor in that some people chose not answer.

4.1.1 Probability sampling

Simple random sampling which is a method where each member in the population has an equal chance of being selected. Basically this corresponds to making a list of the population and randomly select a number of members from that list. One of the problems with simple random sampling is that there is a risk of missing out participants of smaller population groups. Therefore it typically requires a relatively large sample to be valid.

Cluster sampling is when selecting a whole group instead of just several individuals like in simple random sampling. An example could be examining one class in a school. This method is usually used for practical reasons since it might be less expensive or difficult than using a sample of individuals.

In *stratified sampling* the population is divided into groups called strata. The proportion between the different strata groups is then decided. For instance you might decide that the sample should be consist of 50% males and 50% females. This might be done in order to ensure that sub-groups of the population is included in the sample or included in the right proportion. It might also be used if certain groups are to be given more or less weight.

4.1.2 Non-probability sampling

In *non-probability sampling* the selection made out of some other criteria. This will normally make the sample likely to be in some degree different from average, so it might be not possible to use non-probability samples to generalize conclusions for the whole population.

Convenience sampling is when the sample is made out of participants that are easy to get. For instance you might only select people you already know.

When the researchers selects participants based on their own judgment of who they think might be appropriate to include, this is called *Purposive sampling* or *Judgment Sampling*. For instance if you are planning a new payroll system and seeking requirements, it would probably be more useful to select one of the secretaries than 10 random workers.

Quota Sampling is similar to stratified sampling in that the population is divided into groups. But in quota sampling the members are chosen freely (and not with a certain probability) as long as the quota is met (say that 25% are 50 years or older).

4.1.3 Criteria for selection

I could have just sent an open post to the mailing lists asking for anyone to participate. But that would have been self-selection, so that was not an option.

Only persons which have posted on the Linux Kernel Janitor Project mailing list have been asked. This is sort of a cluster sampling, however since neither wine nor Asterisk have a separate janitor organization the effect of including them would probably be marginal.

In this study there is no point in finding an exact average of the janitors' viewpoints. So I will not use simple random sampling. Instead I will selecting some groups/strata of people that I think will give perhaps different but important answers.

Arguably the most active developers are the most important members of the janitor project, so I will give most attention to them. However, recruitment is also very important for the long term stability of the project. Just as some new people are joining in others will have left, so I will also ask some of those who no longer are actively participating in the janitor project. Table 4.1 shows the details.

Strata	Sample Size	Selection Criteria
The most active developers	50%	Top posters between 2005-03-01 and 2006-02-28
Newcomers/just started	25%	First time posted between 2005-11-01 and 2006-02-28
No longer active	25%	Last time posted between 2004-07-01 and 2005-06-30

Table 4.1: Selection criteria and strata distribution

After deciding what groups to ask and their internal distribution, I had to decide on how many persons to ask. When doing quantitative studies it is very important that the sample is not to small, in order to be reasonable representative. It is possible to calculate the required size of the sample given a *confidence interval*[26] and *confidence level*.

confidence interval gives a range of values for the measured variable. For instance when flipping a coin, you might state that the confidence interval for getting heads is between 45% and 55%[1].

confidence level of say 95% means that if the study is repeated many times, the true value of the variable measured will lie within the confidence interval 95% of the times.

These factors are not so critical for a qualitative study like this, but they should at least to some be degree relevant. Therefore some on-line sample size calculators ([28], [29] and [30]) were used to estimate a sample size. A population of 100, confidence interval of $\pm$10% and confidence level of 90% was used as input. They gave 36, 40 and 41 as answers. 40 persons was chosen as sample size, which then gives 20 Top posters, 10 First time posters and 10 Last time posters.

To find the top posters the mboxstats tool[31] was used. This tool produces various statistics when fed with a mailing list saved in mbox format [2]. So this was quite simple and straight forward.

[1]The probability is in fact not exactly 50%. It depends on which side that starts lying up[27]

[2]It was primarily used for analyzing the main Linux kernel mailing list (LKML) by Kernel Traffic, `http://www.kerneltraffic.org/` (Kernel Traffic has provided summaries of the discussions

To find persons that posted for the first time to the Linux Kernel Janitor Project mailing list between 2005-11-01 and 2006-02-28 I wrote some perl scripts to find all emails before a given time and then compare with this emails in posts after that. Finding last time posted between 2004-07-01 and 2005-06-30 was more or less the reverse of this.

The result was then two lists from which 10 names were randomly extracted from each. However the lists was manually "washed" afterwords, replacing a few entries with new ones in order to make sure that the person had not just changed mail address as well as discarding persons that I determined had posted to threads that started out on a different mailing list and that one some point had been cc'ed to the janitor mailing list if that person did not appear to othervise be an active janitor.

This was however quite difficult because the mailing list archive had stripped out almost all headers from the original posts (notably both `To:` and `Cc:` [3]). and a couple of "why did you ask me" responses were received from people that received the interview questions that ideally should have been washed out.

One of the selected persons was both part of the top posters group as well as newly started. I consider this OK since I assume he will represent both groups (quota sampling). 39 persons was therefore asked to participate. The questions asked are listed in appendix A on page 61.

4.2 Interview responses

The response rate was much lower than I had expected. I sent an email with the questions Saturday 8th April 2006. The intention was to give one week response time, however I noticed later (on Sunday 16th April) that I had made an error and written Saturday 9th April. Since some of the receivers then probably interpreted that to be only one day response time and because of the very low response I therefore resent the questions with a new response period to those that had not answered. Only 5 persons answered in the first round and 2 persons answered in the second round.

See appendix A on page 61 for the full sett of complete questions.

on LKML for many years, but is currently on a break). A modification of mboxstats was necessary because out of the box it crashed when processing the janitor mailing list. The janitor mailing list archive is processed by pipermail which strips out most of the mail headers and mboxstats was not happy with that. It was however not a big problem to comment out the code that crashed (report of most busy day of the week).

[3] As of writing this today, exactly one week before the finish deadline, I noticed that the headers are not stripped out in the complete all-time archive, only in the monthly archives which I happened to use excursively…

Q1, using Linux	less than one year, 3, 3.5, 4, 10, 11, 12 years
Q2, programming	3, 3.5, 4-5, 9, 14, 15, 18 years
Q3, when involved	2 persons 3 months ago, 1 person 2 years ago 3 persons 3 years ago and one 3-4 years ago
Q4a, more/less involved	3 less, 2 more and 1 unchanged
Q5, other open source projects	2 no, the others varied from most effort in KJ to most effort elsewhere
Q7, manual code reviews	1 seldom, 3 sometimes, 2 often, 1 always
Q8, post corrections	everyone had done this
Q9, coding style knowledge	2 basic, 2 between basic and solid, 3 solid
Q10, start time estimate	estimates from 1-2 weeks up till 1-2 months
Q11, Detailed knowledge about submitted patch handling	2 no, 2 mostly, 3 yes
Q12, number of submitted patches (through KJ)	0-10: 3 20-30: 1 40-50: 1 ca 100: 1 ca 200: 1
Q13a, accepted notification	1 always, 2 sometimes, 1 often, 1 don't know
Q13b, rejected notification	2 always, 2 sometimes, 1 seldom, 1 don't know
Q15, age	20 years: 1 23 years: 2 28 years: 1 29 years: 1 32 years: 1
Q16, male/female	all male
Q17, nationality	mostly from Europe, everyone from different countries
Q18, educational degree	3 of approximately high school 1 of Bachelor 3 of Master (finished or studying)
Q19 working as a programmer	3 months, 9 and 10 years

Q3, why involved in KJ	Not everyone answered on why they got involved, but the answers were a combination of personal interest in learning (programming, operating system etc) and that it was a way to contribute back to the community.
Q4b, why more/less involved	For those less involved than before lack of time was common. For those more active this was triggered by being more confortable with contributing.
Q6, tools used	It was very common to use the normal build system and look into warning from the compiler or use specific make targets (like `randconfig` or `namespacecheck`). Some used various scripts and unix utilities (find, sed, awk, etc). Also looking into the result produced by other external projects (like ICC[4] and Coverity) was done.
Q14, improve KJ	Common here was a wish for clearer TODO list (perhaps with some more details on the work items) and better feedback on patches.

When working with the responses one person from the top 20 group was noticed to also be in the list of persons newly started. The email entries had been slightly different and therefore not detected before. The 7 replies fell into the following strata: 1 no longer active, 3 newly started and 5 from top 20.

4.3 Summary

With just 7 answers it is difficult to draw any statistical significant conclusions but some characteristics can be made. The results from the interviews are discussed more in chapter 6 on page 51.

- The age ranged from 20 to 32. Some of them was working with programming as a profession (for as long as 9-10 years) but others were not. The education was spread from high school equivalent up till Master.
- The persons was mainly from Europe.
- The experience with Linux was widely spread, ranging from under one year up till twelve years.
- Everyone started programming before they started using Linux, or at the same time.
- The participation in other open source projects was spread, ranging from not participating in other projects to being a core developer in another project (larger

[4]Intel C Compiler

than the janitor project).

- The number of patches submitted through the Linux Kernel Janitor Project varied as well, ranging from a one digit number to a couple of hundreds. One noteworthy fact however was that some the participants had submitted patches to the kernel in other ways (for instance via the -mm tree) and had mainly contributed that way (with estimates of up till around 1000 patches).

Chapter 5

Analysis of janitor patches

As mentioned in chapter 2.2.3 on page 22, patches are an essential part of the development. In this chapter I will look into some quantitative properties of the patches that are produced by the Linux Kernel Janitor Project.

5.1 Possible quantitative aspects that could be analysed

The following is a list of quantitative properties that could be investigated. With the limited time I have available I will of course not look into all of those.

- Frequency of patches, added/removed/total over time.
- Average size of patches in terms of lines and of number of affected files.
- Classify type of change and show distribution of corrective/adaptive/perfective/-preventive change.
- Time from a patch is
  - posted to it is included in the kernel janitor patchset.
  - included in the kernel janitor patchset to it is included in the -mm tree.
  - included in the -mm tree to it is included in the official Linus' tree.
    (these points corresponds to arrows 6 + 7, 8 and 9 in Figure 3.1 on page 30)
- How many patches are reworked and posted in an updated version.
- How many patches are rejected.
- Which parts of the kernel the janitor patches modify.
- Frequency of releases.
- Dependencies between patches,
  - Sequential dependency.
  - Functional dependency.

I will focus the analysis in this chapter to only look at the evolution of the janitor patchset, and not the interaction with the main kernel. This is because I want to focus on the work done by the janitors.

For some of the properties it would probably be insteresting to not just look at the janitor patches in isolation but to also compare with the corresponding numbers from the ordinary kernel development as well. Unfortunately I will not have time for that.

5.2 Frequency and size

Each patchset release is stored in a correspondingly subdirectory at the ftp server. The patches are available as both one combined file as well as all the individual patches in a separate file. These were downloaded and different analysis was made.

5.2.1 Frequency

Table 5.1 on page 48 gives a full list of the number of added and removed patches, the total number of patches as well as the number patches not changed in each release. These numbers are also drawn graphically over time in Figure 5.1 which shows total and unchanged, Figure 5.2 on the following page which shows added and Figure 5.3 on page 46 which shows the number of removed patches.

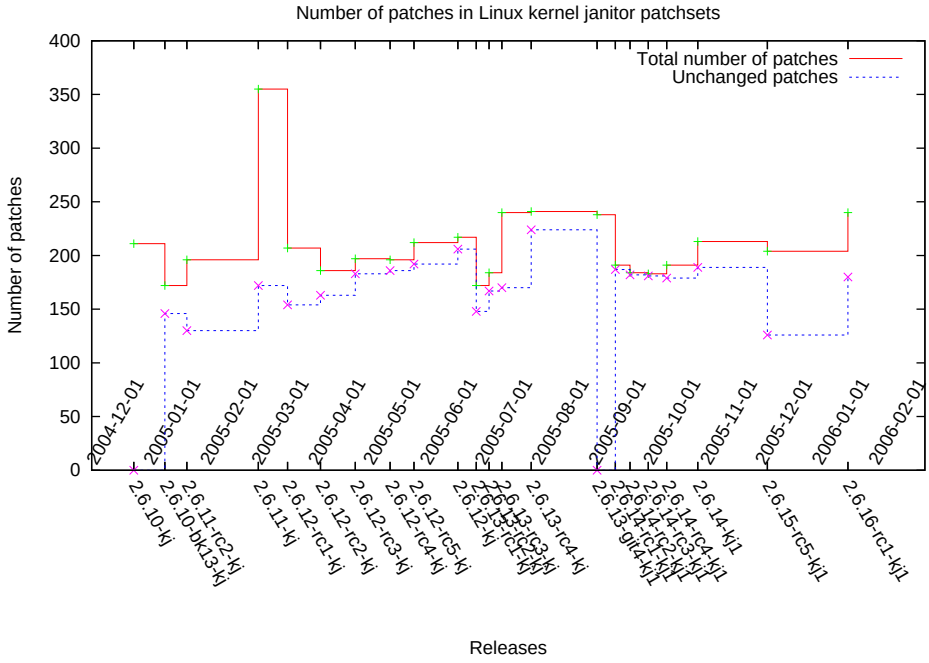


Figure 5.1: Total number of patches and number of unchanged patches

There are two spikes in the graphs. The first one at release 2.6.11-kj is caused by the fact that this release just had a large amount of patches added. The second one, at release 2.6.13-git4-kj1, is triggered by a renaming of all the patch files. From the announcement to that release: *I cleaned up changelogs and subject lines. Minor tweaking*

and collapsing of patches into bigger ones was also made. Therefore all the previous existing patches appears to be removed and the new total appears as all just added in this release.

Ignoring the two releases 2.6.10-kj and 2.6.13-git4-kj1 in Table 5.1 on page 48 gives arithmetic mean average of 35.8, 34.2 and 209.05 for added, removed and total number of patches. The median is 21.5, 15 and 196.5 respectively. So a typical Linux Kernel Janitor Project patchset release consists of around 200 patches and typically 10-15% of the patches are replaced from release to release.

When patches are removed from one patchset release to the next this might be due to different reasons. The most important reasons are given in the following list. No effort has been spent on determining the exact reason for any of the individual patches.

- integrated into the -mm tree.
- rejected for some reason.
- the target that a patch modifies has changed so that the patch does not
  - apply cleanly (possibly sent back to the creator for an update).
  - make sense any longer (for instance if the function modified is removed).
- the same modification is already done elsewhere independently.

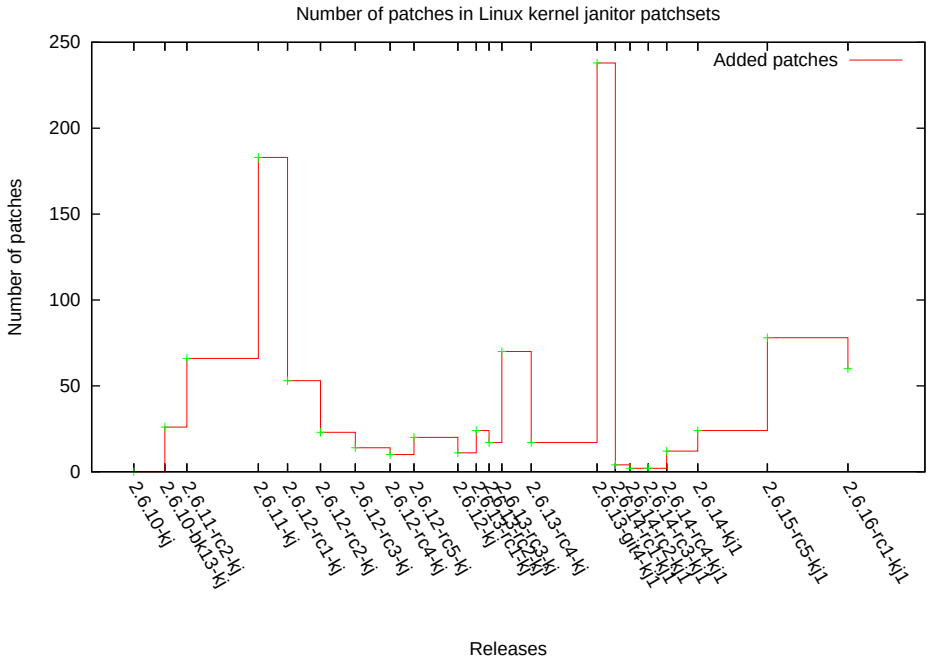


Figure 5.2: Added number of patches

5.2.2 Average size

Summing all the added patches numbers from Table 5.1 on page 48 plus the initial 211 gives a total of 1165 unique patches. However when counting the unique number of filenames from the combined series[1] files[2] I only got 1162. These files were also the basis for generating Table 5.1 on page 48 so I suppose that this means that a very few filenames have been reused.

The vast majority (90%) of the patches did only modify one single file. A complete overview is given in Table 5.2 on page 49. The most intrusive patch which modifies 255 files is `space_before_n_removal.patch`, but this is quite exceptional. The second most intrusive patch is down to "just" 43 files. The 0 files entry in the table is `2.6.11-rc2-kj/split/msleep-drivers_telephony_ixj.patch` which is corrupt and does not contain a patch.

A particular interesting finding was the following. The accumulated number of lines added in the unique set of patches is 29200 lines. The corresponding number of deleted lines is 50561. So this means that the net contribution from the Linux Kernel Janitor Project to the kernel quite clearly is a reduction of code.

Looking at the individual distribution of lines added/removed reveals that 405 (35%)

[1] A *series* file is a file containing a list of all the patches in a patchset which is created and needed when using the tool quilt, `http://savannah.nongnu.org/projects/quilt`.

[2] I had to manually correct a few of the series files. In some cases quilt had missed including some files while for one release the series file listed non-existing files.

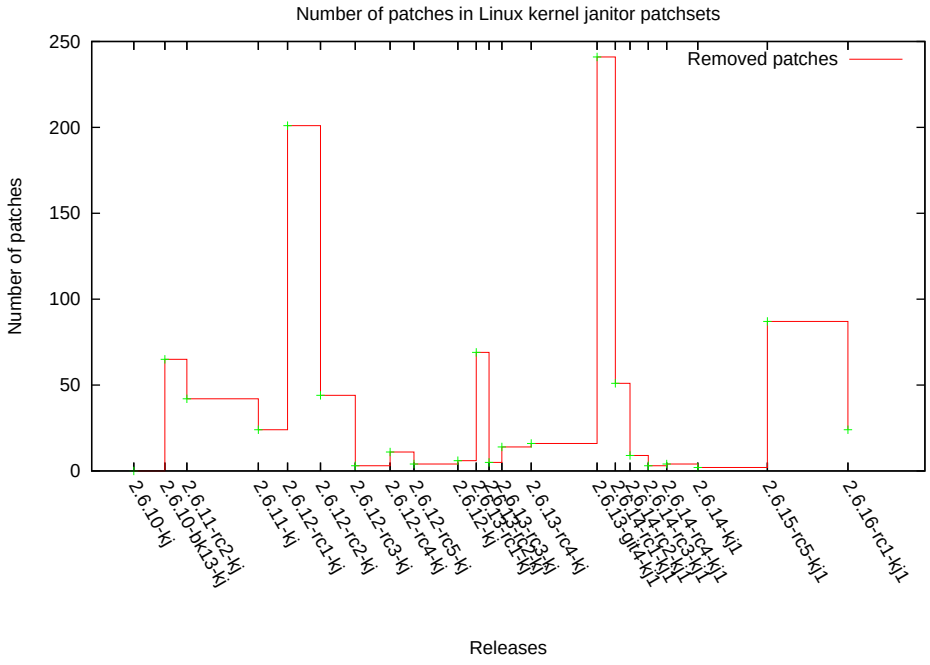


Figure 5.3: Removed number of patches

of the patches removes more lines than they add while 432 (37%) of the patches does not change the number of lines.

5.3 Maintenance types

5.3.1 Distribution of maintenance types for janitor patches

Due to time constraints I did not have time to classify all the 1162 Linux Kernel Janitor Project patches. I selected one of the patchset releases, 2.6.12-rc3-kj, which contained 197 patches and analysed those. There was no specific reason for choosing this particular release other than when looking at Figure 5.1 on page 44 this release looks rather average and is not near the edges.

The results of the analysis of release 2.6.12-rc3-kj are given in Table 5.3 on page 50. This is a little weak result since it is not that many patches that was examined, but still I think the trends are representative. Since the purpose of the Linux Kernel Janitor Project is *fixing up unmaintained code and doing other cleanups* a high degree of corrective and preventive maintenance is to be expected.

The original Lientz, Swanson and Tompkins study in 1978 found that 17.4% of maintenance effort was categorized as corrective, 18.2% as adaptive, 60.3% as perfective while 4.1% was categorized as other. In [13] these figures are shown to be quite inaccurate and the numbers are found to be in the following range for the Linux kernel:

perfective	20-55%
adaptive	<1%
corrective	40-80%

Since this study does not include perfective maintenance the results are not directly comparable. Still, comparing the result from the Linux Kernel Janitor Project patches reveals that the project has much less perfective maintenance, but this should not come as a surprise since the main focus of the Linux Kernel Janitor Project is not development of new functionality.

Note that my definition of correction does not necessarily match that study. See appendix B on page 64 for which criteria I used.

Release	Date	Added Patches	Removed Patches	Unchanged Patches	Total Patches
2.6.10-kj	2004-12-24	0	0	0	211
2.6.10-bk13-kj	2005-01-10	26	65	146	172
2.6.11-rc2-kj	2005-01-22	66	42	130	196
2.6.11-kj	2005-03-02	183	24	172	355
2.6.12-rc1-kj	2005-03-18	53	201	154	207
2.6.12-rc2-kj	2005-04-05	23	44	163	186
2.6.12-rc3-kj	2005-04-24	14	3	183	197
2.6.12-rc4-kj	2005-05-13	10	11	186	196
2.6.12-rc5-kj	2005-05-26	20	4	192	212
2.6.12-kj	2005-06-19	11	6	206	217
2.6.13-rc1-kj	2005-06-29	24	69	148	172
2.6.13-rc2-kj	2005-07-06	17	5	167	184
2.6.13-rc3-kj	2005-07-13	70	14	170	240
2.6.13-rc4-kj	2005-07-29	17	16	224	241
2.6.13-git4-kj1	2005-09-03	238	241	0	238
2.6.14-rc1-kj1	2005-09-13	4	51	187	191
2.6.14-rc2-kj1	2005-09-21	2	9	182	184
2.6.14-rc3-kj1	2005-10-01	2	3	181	183
2.6.14-rc4-kj1	2005-10-11	12	4	179	191
2.6.14-kj1	2005-10-28	24	2	189	213
2.6.15-rc5-kj1	2005-12-05	78	87	126	204
2.6.16-rc1-kj1	2006-01-18	60	24	180	240

Table 5.1: Number of patches in Linux Kernel Janitor Project patchset releases

Table explanation: Release 2.6.10-kj contains 211 patches. Release 2.6.10-bk13-kj has a total of 172 patches, where 26 of those are new compared to 2.6.10-kj. The difference are those removed, i.e. $211 + 26 - 65 = 172$

Number of files modified by a given patch	Number of patches with this property
0	1
1	1042
2	49
3	18
4	12
5	7
6	3
7	5
8	1
10	1
12	1
13	6
15	1
16	1
18	4
21	1
24	1
25	1
31	1
33	1
34	1
35	1
40	1
43	1
255	1

Table 5.2: Number of files changed in patches

Type	Number of times
invalid	1 (0.5%)
preventive	91 (46%)
perfective	1 (0.5%)
adaptive	1 (0.5%)
corrective	102 (52%)

See appendix B on page 64 for criterias used for classification.

Table 5.3: Distribution of maintenance types in patchset release 2.6.12-rc3-kj

Chapter 6

Discussion

6.1 Weaknesses and uncertainties in the results

With so few answers to the interview the information extracted from the answers is statistically weak.

The result that 90% of the patches only modify one file might be somewhat skewed in that in that a large change affecting many files might have been split into several patches. In fact if the janitors perform correction of error patterns, which they do, this will have a tendency to create patches that modify many files. When so many files are single file patches this is most likely because such many-files-patches are split up.

This makes it difficult to compare patches without knowing if a patch is

- a stand alone patch
- part of a series of patches but can be considered individually (typically API conversion)
- part of a set of patches which are dependent on each other

No effort in determining dependencies between patches has been made for this thesis.

Since there is much “guest” traffic on the Linux Kernel Janitor Project mailing list it was very difficult to try to guess which persons that were janitors. The best solution had been if historical mailing list subscriber information had been available.

6.2 How is janitor work different from normal development?

In contrast to normal development which is about adding features (to adapt to Lehman's first law of software evolution, *"E-type systems must be continually adapted else they become progressively less satisfactory"*), maintenance is about preserving features. While traditional maintenance perhaps has been focused on fixing thing that has broken and "if it aint broke, don't fix it", janitor maintenance is more about pro-active actions with a "fix it before it breaks" attitude.

6.3 Starting your own janitor project?

A project that works in parallel with normal development reducing the code size[1] while still keeping the same functionality or even improving it, that ought to sound attractive from a management perspective. The attraction will be a little for open source and proprietary software, so these will be covered separately in the following sections.

If the developers have time to do the janitorial cleanup themselves there will be no need for a janitor project. This is hardly the case normally, so almost any project would benefit from the assistance of some janitors. The size of a project will probably have to be above small in order to maintain a longterm janitor activity.

If a project have say just one or two developers, then if an additional person joins the project offering to do janitor tasks he or she is likely to end up as a co-developer more than as a janitor pretty fast. The janitors will need mentorship and partial participation from at least some of the developers, and if the project is small this will then probably include most of the developers.

On the janitor side I do not think that there is a lower limit. I.e. a medium sized project would probably work well and benefit even with just one person working as a janitor.

6.3.1 Open source projects

For open source projects the cost involved in adding janitors to the project will probably only be a small management overhead. The main project will most likely already have a mailing list and website so adding that for the janitors should be simple, or

[1] There is of course a theoretical possibility that the reduced code is more complex and thus harder to maintain, but this is not very likely since

- The janitors are doing only maintenance and they would do themselves a disfavor by changing code to be less maintainable.
- The developers would be reluctant to accepting patches from the janitors that they would consider reduced the maintainability.

in any case the janitors could create their own project on sourceforge or similar. The initial investment should therefore be neglectible. The continuously running costs will be the following

- maintain[2] or help maintaining a list of janitor tasks.
- Answer questions from the janitors.
- Integrate patches.
- Provide feedback for patches integrated or rejected.

but the benefits from the janitor work will most likely be valueable enough that this will be a good investment. I think the following list will be a sufficient checklist for starting a janitor sub-project in open source projects:

- ☐ The size of the project is medium or large.
- ☐ Someone is willing to be janitors.

For small projects it will very likely be beneficial to set up a list with janitor tasks to attract new developers. But it will not make sense to run the janitor activity separately.

6.3.2 Projects developing proprietary software

This study has been of the Linux kernel, a large open source project for which janitor activity obviously is beneficial, the more the better. While janitor activity obviously will be good for the quality for development of closed, proprietary source code as well, it is not clear in what amount this will be most profitable. Will allocating 1% of the resources to do janitor work be the optimal effort? Or 5%? Or 10%? This will be highly dependent on both the current quality status and the given quality requirements.

Open source projects typically have a *"It's ready when it's ready"* philosophy or more specifically, given the old saying *"Time, Quality, Cost – Pick any two"* the developers might choose to pick only one – quality[3] – and let both time and cost [4] slide.

This is in contrast with most proprietary projects where the quality should be just "good enough" (this is discussed in [32]) and time to market often is essential. The cost of being delayed can easily exceed the cost of development at the end of the project.

So for a commercially developed project any activity like source code janitor activity is an upfront investment that must have an expected return of value in increased sales and/or reduced maintenance cost.

[2] The janitors could very well maintain the list themselves but will at least require input and co-operation from the main project.

[3] *"Nobody knows when a kernel will be released, because it's released according to perceived bug status, not according to a preconceived timeline."* – Andrew Morton

[4] Even though many open source projects are done for free by volunteers in their spare time, the effort invested is certainly a cost.

The initial investment will be neglectible, the main cost in this case will be the work effort of the janitors. The question will be more about the degree of janitor work than yes/no.

- □ The expected return value of such and such amount of janitor work will be greater than the cost.

6.4 How do janitor participants compare to other open source developers?

The developers that answered the interview questions in this study had among others the following properties:

- age between 20 and 32 years
- all male
- none were from the same country
- 3 of 7 working as programmers (0.25, 9 and 10 years)

Other studies of open source developers have been made, one of them is the Boston Consulting Group/OSTG Hacker Survey, [33] which surveyed 684 hackers participating on software projects on SourceForge.net. The results from this thesis are almost an exact match with the results from that survey:

- 70% between 22 and 37 years
- 98% male
- "open source is a global enterprise"
- 45% working as programmers, with 11 years average experience

So this means that the kernel janitor participants are not very different from other open source developers. While this discovery might be described as non-shocking, it implies that there is nothing particular about source code janitor activity that is limited to some subset of developers and that janitor activity should be possible to practice universally by people working with software development.

6.5 What quality mechanisms are used?

How does Linux developers decide on what patches to accept and what to reject? As usual for open source projects, this is highly informal. In the article ”Release criteria for the Linux kernel“, [34] the author states that

> Analysis of the release criteria for the Linux kernels shows a process which is dynamic and whose nature depends on the developer in charge of the administration of a particular version.

One of the criteria that are used for accepting patches is that the patch should only do one thing. Patches which contains several different changes will almost automatically be rejected. Not everyone finds it easy to work this way but the kernel developers are unlikely to loosen this requirement. See [35] for a summary of the discussion that arose around attempts to submit large IrDA changes.

Another mechanism is to pass all changes through one committer. In Wine, all patches will pass through Alexandre Julliard. In the case of Linux everything passes through Linus Torvalds but most of the patches will even have passed through one or several of the other kernel developers on their way to the official kernel.

The article ”Teaching the Old Dog, a lesson in code review from the open source community“, ([36]) tells a story of a team that started working with Wine and had to deal with adjusting into doing things like they were done in Wine.

> But we hadn’t worked on any Open Source projects before. I hadn’t expected an Open Source project to be quite so rigidly controlled: I’d imagined the exact opposite. Unadulterated chaos. I’d pictured myself having to wade through reams of crap code trying to identify useful features, like separating a bowl of party mix into its core components.
>
> Instead, I – and my team – had encountered a concrete system as mature as any I’d ever seen.

There is nowhere to hide when all patches are submitted publicly and everyone has a chance of reviewing and make comments. If your patch has weaknesses others will point them out. Maybe even in a non-diplomatic way; ”crap“ is a term that sometimes is used to describe other peoples code on the LKML (but this is not used on the mailing list of the Linux Kernel Janitor Project).

For janitor patches the patches will first be posted for public review on the mailing list, then the Linux Kernel Janitor Project leader acts like a single committer when accepting patches into the janitor patchset. The patches then must pass Andrew Morton’s acceptance in order to slip into the -mm tree. And finally the patch must be accepted by Linus Torvalds in order to get into the official kernel.

6.6 Suggestions for improvements

While the Linux Kernel Janitor Project is working quite well, there is always room for improvement.

6.6.1 Better feedback on patches for new janitors

Several of the janitors interviewed expressed that the feedback on patches submitted could be improved. To improve this I would suggest that new janitors always receive an accept/reject email for patches until X patches have been accepted. This could be implemented by adding a new tag to the patch

JanitorPatchID: Some.Janitor@example.com 001 v0

where the patch numbers are specific to each author which keeps track his or her own numbers. This will then also have a positive bonus effect in that most newcomers then would have as a goal to reach at least X patches accepted.

6.6.2 New logo

To create more publicity about the Linux Kernel Janitor Project, perhaps a contest for creating a logo could be made? Currently the janitor website has the following picture

which is just a standard tux[5] with a small broom added. A specific logo could perhaps strengthen the identity and increase the publicity of the Linux Kernel Janitor Project.

[5] Tux is the “official” Linux mascot, picture created by Larry Ewing, `http://www.isc.tamu.edu/~lewing/linux/`.

Chapter 7

Conclusion

7.1 Results

The most important result from this study was perhaps the discovery that janitor activity reduces the amount of code while keeping or improving the functionality.

A summary of the most important items are that janitor activity

- has a net contribution of reduction of code
- improves quality
- corrects bugs
- recruits developers to normal development
- has a process that is independent of tools

The product produced by the Linux Kernel Janitor Project is a set of patches which are maintained as a patchset release consists of around 200 patches where $\pm$ 10-15% are replaced from one release to the next release.

The high level of guest traffic on the Linux Kernel Janitor Project mailing list (as seen in Figure 3.2 on page 32) indicates that the Linux Kernel Janitor Project is very cooperative and not just some outside part of the development.

7.2 Recommendation

To make the initial period a better experience for newcomers I suggest that they are always given feedback on patches they submit until a certain level of skills is achieved.

The findings in this report indicate that additional projects can benefit from including a Janitor activity, both for maintaining the software and for maintaining the programmers skills in the organisation.

Bibliography

[1] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski, "Metrics and laws of software evolution - the nineties view," *metrics*, vol. 00, p. 20, 1997.

[2] M. W. Godfrey and Q. Tu, "Evolution in open source software: A case study," in *ICSM '00: Proceedings of the International Conference on Software Maintenance (ICSM'00)*. Washington, DC, USA: IEEE Computer Society, 2000, p. 131.

[3] J. Tran, M. Godfrey, E. Lee, and R. Holt, "Architectural repair of open source software," 2000. [Online]. Available: citeseer.ist.psu.edu/tran00architectural.html

[4] P. G. Armour, "The business of software: the laws of software process," *Communications of the ACM*, vol. 44, no. 1, pp. 15–17, 2001.

[5] C. Hayne, "Software engineering for usability," prepared for General DataComm's Multimedia R&D Centre, Sept. 1996. [Online]. Available: http://hayne.net/HCI/Seu/SE_for_usability.html

[6] S. W. Ambler. (2006, Apr.) Choose the right software method for the job. [Online]. Available: http://www.agiledata.org/essays/differentStrategies.html

[7] Wikipedia, "Extreme programming – wikipedia, the free encyclopedia," 2006, [Online; accessed 21-May-2006]. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Extreme_Programming&oldid=54064678

[8] E. B. Swanson, "The dimensions of maintenance," in *ICSE '76: Proceedings of the 2nd international conference on Software engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1976, pp. 492–497.

[9] R. S. Pressman, *Software Engineering - A Practitioner's Approach*, ser. Computer Science Series. McGraw-Hill International Editions, 1987.

[10] E. Tryggeseth, "Support for understanding in software maintenance," 1996. [Online]. Available: citeseer.ist.psu.edu/tryggeseth97support.html

[11] B. A. Kitchenham, G. H. Travassos, A. von Mayrhauser, F. Niessink, N. F. Schneidewind, J. Singer, S. Takada, R. Vehvilainen, and H. Yang, "Towards an ontology of software maintenance," *Journal of Software Maintenance*, vol. 11, no. 6, pp. 365–389, 1999.

[12] E. J. Barry, C. F. Kemerer, and S. A. Slaughter, "Toward a detailed classification scheme for software maintenance activities," in *Proceedings Of the 5th Americas Conference on Information Systems*, Aug. 1999.

[13] S. R. Schach, B. Jin, L. Yu, G. Z. Heller, and J. Offutt, "Determining the distribution of maintenance categories: Survey versus measurement," *Empirical Softw. Engg.*, vol. 8, no. 4, pp. 351–365, 2003.

[14] Wikipedia, "Mentor — wikipedia, the free encyclopedia," 2006, [Online; accessed 10-May-2006]. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Mentor&oldid=51291039

[15] C. King. Mentoring and being mentored on the technology track. [Online]. Available: http://developers.sun.com/toolkits/articles/mentor.html

[16] J. H. Holloway. The benefits of mentoring. [Online]. Available: http://www.nea.org/mentoring/resbene050603.html

[17] S. E. Sim and R. C. Holt, "The ramp-up problem in software projects: A case study of how software immigrants naturalize," in *International Conference on Software Engineering*, 1998, pp. 361–370. [Online]. Available: citeseer.ist.psu.edu/5440.html

[18] K. L. Stout, "Reclusive linux founder opens up," *Cable News Network*, may 2006. [Online]. Available: http://edition.cnn.com/2006/BUSINESS/05/18/global.office.linustorvalds/

[19] K. Nakakoji, Y. Yamamoto, Y. Nishinaka, K. Kishida, and Y. Ye, "Evolution patterns of open-source software systems and communities," in *Proceedings of the International Workshop on Principles of Software Evolution*, ser. session 4. New York, NY, USA: ACM Press, May 2002, pp. 76–85. [Online]. Available: http://portal.acm.org/citation.cfm?id=512055&jmp=cit&dl=portal&dl=ACM

[20] C. Gacek and B. Arief, "The many meanings of open source," *IEEE Software*, vol. 21, no. 1, pp. 34–40, 2004.

[21] J. Barr, "Perceptions of world domination," http://www.linuxworld.com/linuxworld/lw-2000-03/lw-03-vcontrol_4.html, Mar. 2000, also avaliable at http://xent.com/pipermail/fork/2002-January/008429.html. [Online]. Available: http://web.archive.org/web/*/http://www.linuxworld.com/linuxworld/lw-2000-03/lw-03-vcontrol_4.html

[22] Linus gives bitkeeper a test run. [Online]. Available: http://www.kerneltraffic.org/kernel-traffic/kt20020211_153_print.html#9

[23] (2004, May) Bitkeeper after the storm - part 1. [Online]. Available: http://software.newsforge.com/software/04/05/10/1235236.shtml?tid=151&tid=2&tid=82&tid=94

[24] Linus no longer using bitkeeper; creates 'git' replacement. [Online]. Available: http://www.kerneltraffic.org/kernel-traffic/kt20050426_307.html#5

[25] Wikipedia, "Janitor – wikipedia, the free encyclopedia," 2006, [Online; accessed 19-May-2006]. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Janitor&oldid=53078581

[26] ——, "Confidence interval – wikipedia, the free encyclopedia," 2006, [accessed 18-April-2006]. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Confidence_interval&oldid=48268729

[27] E. Klarreich, "Toss out the toss-up: Bias in heads-or-tails," *Science News*, vol. 165, no. 9, p. 131, 2004. [Online]. Available: http://www.sciencenews.org/articles/20040228/fob2.asp

[28] Sample size calculator. [Online]. Available: http://www.dxresearch.net/index.cfm?fa=resources.sample

[29] Sample size calculator. [Online]. Available: http://www.gensurvey.com/resources05.asp

[30] Sample size calculator. [Online]. Available: http://www.macorr.com/ss_calculator.htm

[31] mboxstats. [Online]. Available: http://www.vanheusden.com/mboxstats/

[32] J. Bach, "The challenge of "good enough" software," 2003, http://citeseer.ist.psu.edu/639234.html. [Online]. Available: http://www.satisfice.com/articles/gooden2.pdf

[33] The boston consulting group/ostg hacker survey. Http://www.ostg.com/bcg/. [Online]. Available: http://www.ostg.com/bcg/BCGHACKERSURVEY-0.73.pdf

[34] D. G. Glance, "Release criteria for the linux kernel," *First Monday*, vol. volume 9, no. number 4, Apr. 2004. [Online]. Available: http://firstmonday.org/issues/issue9_4/glance/index.html

[35] Big irda changes accepted into 2.4; linus on patch submissions. [Online]. Available: http://www.kerneltraffic.org/kernel-traffic/kt20001120_94.html#9

[36] S. Lussier, "Teaching the old dog, a lesson in code review from the open source community," *Software Quality*, vol. Volume 1, no. No. 1, pp. 14–18, 2004. [Online]. Available: http://www.osqa.org/documents/sq_1_1.pdf

[37] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, June 1999, iSBN: 0201485672.

[38] P. Weaver, *Success in Your Project: A Guide to Student System Development Projects*. FT Prentice Hall, Dec. FT Prentice Hall, iSBN: 0273678094.

[39] The linux kernel janitor project home page. [Online]. Available: http://janitor.kernelnewbies.org/

[40] The research process. [Online]. Available: http://www.ryerson.ca/~mjoppe/ResearchProcess/

[41] (2005) The boston consulting group/ostg hacker survey. Http://www.ostg.com/bcg/BCGHACKERSURVEY-0.73.pdf. [Online]. Available: http://www.ostg.com/bcg/

[42] Wikipedia, "Andrew morton – wikipedia, the free encyclopedia," 2006, [Online; accessed 4-May-2006]. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Andrew_Morton_%28computer_p%rogrammer%29&oldid=48607334

[43] ——, "Linus torvalds – wikipedia, the free encyclopedia," 2006, [Online; accessed 4-May-2006]. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Linus_Torvalds&oldid=51135601

[44] ——, "List of tools for static code analysis – wikipedia, the free encyclopedia," 2006, [Online; accessed 17-May-2006]. [Online]. Available: http://en.wikipedia.org/w/index.php?title=List_of_tools_for_static_code_analysis&oldid=53704047

[45] H. Løvdal, "Analysis and description of open source janitor projects," Master's thesis, Agder University College, May 2006. [Online]. Available: http://student.grm.hia.no/master/ikt06/ikt590/g33/

Appendix A

Interview Questions

A.1 Connection Between Interview Questions and Reseach Questions

The mapping between the questions asked in the interview and the reseach questions are shown in Figure A.1 on page 63.

A.2 Questions

1. How long have you been using Linux?
2. How long have you been programming?
3. Why and when did you get involved in the Linux kernel janitor project?
4. (a) Are you more or less involved in kernel janitor activity now than before?
   (b) If so, what are the cause(s) of the change?
5. Are you contributing to any other open source projects?
6. What tools are you using when doing janitor work?

   I am not thinking of basic tools like shell/editor/distribution etc, but are you looking into warnings from the normal build system, compiling with a special compiler, using a source code analyser like lint, have you written your own scripts for searching for certain things, etc.
7. Are you doing manual code reviews by just reading code?

   never/seldom/sometimes/often/always

8. Do you post corrections to incomplete or improper patches from other developers?

9. How confident are you with coding style, how to properly submit a patch and how the patch should be formated?

   not at all/a little/basic knowledge/solid knowledge

10. How long time did you use or do you estimate the ramp-up time for beginners to be educated about the process of participating in the janitor project?

11. Do you know in detail about what happens to the patch after it is sent to the mailing list?

12. Take a guess (a range is fine) about how many patches you have submitted.

13. (a) When patches was accepted did you receive a notification about it?

    (b) When patches was rejected did you receive a notification about it?

    not applicable/don't know/never/seldom/sometimes/often/always

14. How do you think the janitor project can be improved?

15. What is your age?

16. Are you male/female?

17. What is your nationality?

18. What is your highest educational degree?

19. (a) Are you working as a programmer (i.e. employed or running your own firm)?

    (b) If so for how long?

20. Anything else?

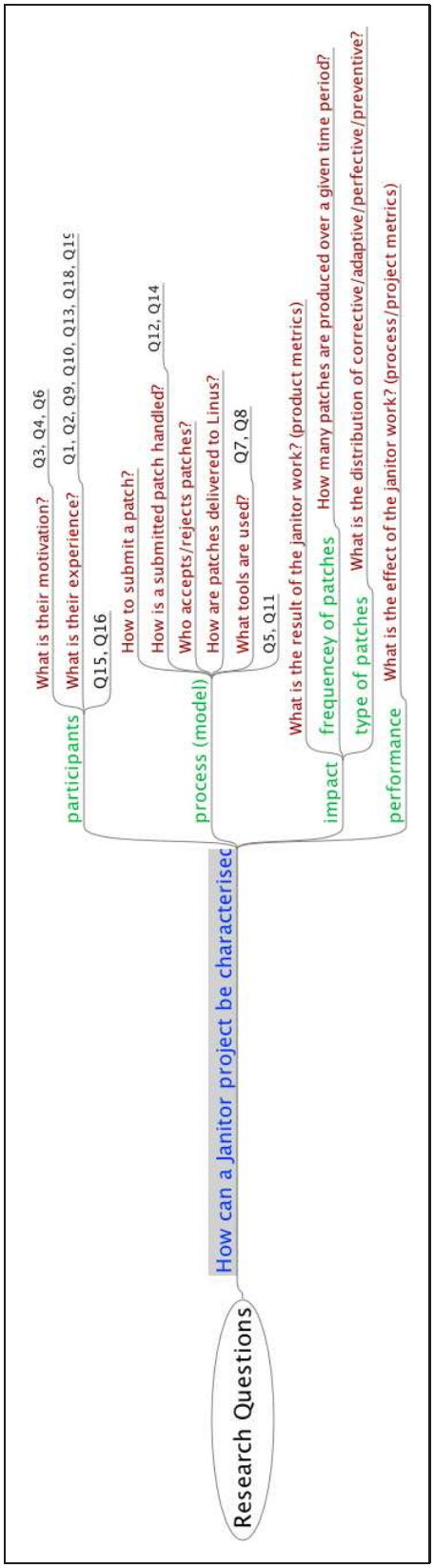


Figure A.1: Connection between interview questions and reseach questions

Appendix B

Criteria used for determination of maintenance types

A quite strict interpretation of the classification preventive was used, only when there was no change in functionality. If the change did something that could be interpreted as a correction of any kind, including spell corrections in comments, it was classified as corrective.

B.1 Example of a corrective patch

The description of the patch `2.6.12-rc3-kj/all-patches/int_sleep_on-drivers_net_tokenring_ibmtr.patch` contains "The patch also fixes a potentially racy conditional in tok_open" and this patch was therefore classified as corrective.

B.2 Example of a perfective patch

The patch `2.6.12-rc3-kj/all-patches/add_module_version-drivers_net_8139cp.patch` adds a line `MODULE_VERSION(DRV_VERSION);` which is a "new feature" and was therefore classified as perfective.

B.3 Example of a preventive patch

The patch `2.6.12-rc3-kj/all-patches/lib-parser-fs_devpts_inode.patch` replaces a `while` loop test with a `for_each_pci_dev` macro instead. No

change in functionality, but the code has become easier to understand and maintain. Classified as preventive.

B.4 Example of an adaptive patch

The patch `2.6.12-rc3-kj/all-patches/function-string-arch-mips.patch` changes to that the `__FUNCTION__` macro is not string concatenated. This had to do with the gcc compiler.

B.5 Example of an invalid patch

`2.6.12-rc3-kj/all-patches/kj_tag` does not contain any patch.

Appendix C

BibTeX entry for this thesis

If you want to cite this this thesis you can use the BibTeX entry in appendix C in [45].

```
@mastersthesis{ masterthesis:janitor-project,
  author = {H\aa{}kon L\o{}vdal},
  title = {Analysis and description of an open source
           janitor project},
  school = {Agder University College},
  year = {2006},
  month = may,
  url = "http://student.grm.hia.no/master/ikt06/ikt590/g33/",
  keywords = {linux, kernel, janitor, software maintenance},
}
```

Note that the college is in the process of becomming a university. The current domain is `.hia.no` (Høgskolen i Agder) but as university it will be `.au.no` (as of writing this, www.au.no is the same IP address as www.hia.no).

Index