Seminar “Open Source Software Engineering”

Wintersemester 2004

Quality Assurance of Open Source Projects

István Bartkowiak

bartkowi@inf.fu-berlin.de

Advisor: Christopher Oezbek

Berlin, 22nd April 2005

Special thanks to Florian Pötter for translating this paper into English.

Abstract

From the outset on, the development of Open Source Software (OSS) was characterized by the difficulty of evaluating the quality of the product. Due to the massive parallel development and the absence of the approved control structures of conventional software development, new strategies for ensuring quality had to be conceived. This paper gives a review of the established processes of the Open Source environment and outlines the remaining problems.

Contents

1 Introduction **3**
1.1 The notion of Quality 3
1.2 Requirements for quality assessment 4
1.3 The creation of a quality consciousness 4
1.4 The creation of quality standards 5
1.5 Definitions 5
1.6 The management of software defects 5
1.7 Management of error reports 7

2 Software Errors **7**
2.1 Possibilities of a user inquiry 7
2.2 Possibilities of user help 8

3 Software Defects **9**
3.1 Defect classification 9
3.2 Defect prevention 10
3.3 Defect compensation 11

4 Software Faults **11**
4.1 Unit tests und test suites 12
4.2 Mass tests 13

5 Software Failures **13**
5.1 Plug-in-Architektur 13
5.2 Scenario-specific adaptations 14

6 Product responsibility **14**
6.1 The team model 16
6.2 The remuneration model 16
6.3 The hybrid model 17

7 Product certification **18**

8 Further development of OSS **19**
8.1 Versioning 20
8.1.1 Release authority 21
8.1.2 Development stages 22
8.1.3 Distribution of releases 22
8.1.4 Version number assignment using the example of the Linux kernel project 23
8.2 Security aspects 23

9 Summary and outlook **24**

1 Introduction

One of the main advantages of the Open Source concept is the possibility to have access to the source code at any time. Especially for high-risk products designed for delicate sectors (see chapter 8.2), this is an indispensable precondition for their application. The person using the software, as well as a third party charged with the security check, can make sure that the source code is *safe* [uoD00].

A disadvantage resulting from the freedoms of development is the lack of any sort of documentation, which can only be provided by a few projects. However, this documentation is necessary for the analysis of a product for its architectural or algorithmic defects (see chapter 2 ff.).

Change and adaptation of program source codes facilitate the development of data protection-friendly products and technologies in a broader sense and thus are an important prerequisite for using digital data in a responsible way (see chapter 6). In the past, it has become apparent that software has always to be considered as potentially defective, and OSS is not likely to change this fact. That is why formalized procedures for the elimination of defects are necessary to create incentives for the development of the Open Source idea.

Note to the reader This study is based on the most recent sources available to outline actual trends. In a first step, the main arguments of the different sources are reproduced, followed by the source references, to be commented afterwards. This procedure aims at presenting the different points of view in an unbiased manner.

1.1 The notion of Quality

To be able to agree on quality issues in software development, the criteria to establish have to be exactly defined. A commonly held notion is the **stage of maturation** of a product [Kad00]. But this means replacing one comprehension problem by another, as it is as hard to define the term “maturation”. It is well possible to quantify it, according to detailed specifications. It is also possible to decide if a product fulfils the expectations in qualitative terms – provided that the evaluation criteria have been preliminarily defined. But as a result, both the quantitative and the qualitative approach will merely deliver a snapshot.

Another approach consists therefore in integrating the **development process** of the software into the quality evaluation. This process-centered approach provides, under the condition of a complete and consistent product specification, even the means of provability of quality (see also chapter7).

Moreover, the economy has developed yet another approach to evaluate the quality of a product. It is certainly of advantage to be able to prove that a product was developed in conformity with given specifications, but this not helpful if nobody is willing or able to work with it. In fact, the decision to adopt a product is based on its price and its

capacity of integration into day-to-day procedures. Both criteria can be calculated and are commonly referred to as *cost-benefit calculation*.

1.2 Requirements for quality assessment

In the early days of the Open Source movement, it was often observed that projects were simply lacking a system plan [Koc01]. However, the existence of such a plan is indispensable because it serves, in the context of a quality assessment, as a foundation for the genesis of the product. This is also the reason why the application of established quality assurance procedures to OSS projects has failed in the past. Subsequent corrections of architectural defects proved to be particularly complex. As it was not known how far-reaching the consequences of a modification would be, any correction could generate entirely new defaults or even ruin the project as a whole. As a result of those mistakes, it has become clear by now that quality assurance shall also accompany OSS projects from the beginning on. It can be seen as a benevolent consultant *(consulting function)*.

The prevention of mistakes is the most effective way of their correction. Therefore it is important to make sure that during the development process the occurrence of mistakes does not exceed a certain level [Hen01] (see also chapter 8.2).

1.3 The creation of a quality consciousness

Of course there are also other approaches in the area of quality assessment. For example, [Bac95] claims that software only has to be as good as its user expects it to be *(discipline of utilitarian approach)*. This approach may cause astonishment, as it can be argued that expectations are always subject to change. But after all, it has always been necessary to progress in stages. It is counterproductive to want it all at once. Rather, it makes sense to progress incrementally towards the achievement of clearly defined goals *(milestones)*. If quality is defined as the solution set and the milestones as the problem set, a gradual progression is possible, without running the risk of losing the ground already gained.

This philosophy was turned upside down by the company *Microsoft*. As it exclusively produces Closed Source Software (subsequently referred to as *CSS*) and is determined to defend its own monopoly at any price, Microsoft acts according to the motto: *"Sell the right product at the right time"*. Therefore the company fixes deadlines and is bent on keeping them, to be always a step ahead of its competitors. However, it is obvious that Microsoft is generally not able to stick to these *deadlines*. As a consequence, a number of features are deliberately omitted, while those features judged indispensable are implemented in a way that they seem to work normally. The necessary corrections are then released little by little, which has led to the creation of the term *"banana software"* – that means software that becomes mature only after its purchase and consequently attains its initially projected quality only with considerable delay.

1.4 The creation of quality standards

Recent certification and processing standards accept the application of nearly every type of development process. As a result of this interchangeability these standards become so-called *meta-processes*. For example, the norm ISO 9126 with its 21 sub-characteristics arranged in 6 focus areas permits an application on OSS projects. As this standard originates from the professional sector, any OSS project compliant to these standards can acquire the same status as a conventional project (see chapter 7). If an OSS project offers a high quality, the developers usually do not get direct feedback. They rather expect error reports and improvement suggestions. That is why the absence of criticism can indeed be interpreted as a positive judgment, according to the proverb: "No news is good news".

1.5 Definitions

An overview of the emergence of undesirable product features – product deficiencies, in judicial terms – necessitates the definition of a number of terms (see [Som04].

- **Bug** – This term is usually employed for any kind of software deficiencies. However, this is by far insufficient in the context of quality assurance. But it may be adequate to indicate a product deficiency in a more general context.
- **Error** – Describes any problem that occurs while somebody is working with the software. For example, an incorrectly programmed user interface may accept invalid or contradictory values that the underlying program logic can't process.
- **Defect** – Any mistake in the program logic is called a defect. Such logical mistakes are caused by semantic mistakes. Another reason is the absence of any program logic.
- **Fault** – A *fault* terms an unwanted program status. It can be caused by a defect, but it can also occur through external influence, e.g. the access to external resources. If the software is able to detect this kind of situation and to intercept it autonomously, it is usually qualified as robust. In a qualitative sense, it is called fault-tolerant.
- **Failure** – This is the most serious deficiency that can occur with a software. It describes the collapse of the software, leading to a system crash.

Fig. 1 illustrates the term hierarchy; fig. 2 correlates these terms.

1.6 The management of software defects

The management of software defects depends heavily on the individual project. By way of example, *Debian GNU/Linux* has introduced a group classification, with each group

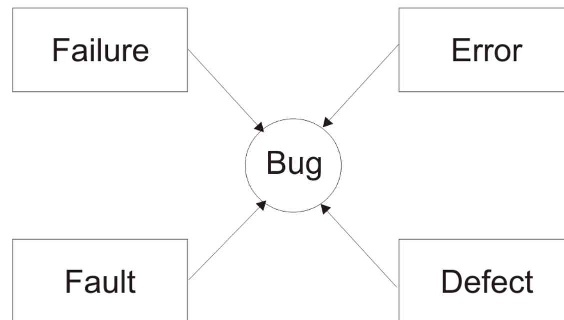

Figure 1: Term hierarchy

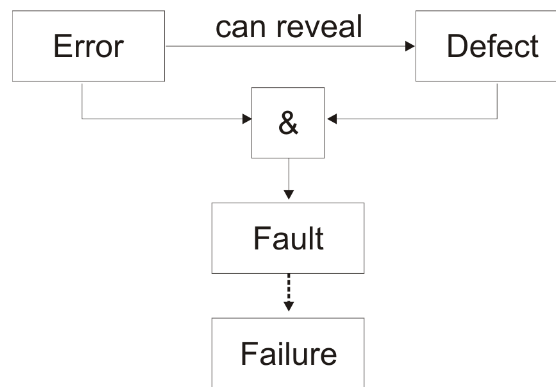

Figure 2: Correlation of the terms

having an exactly defined field of responsibility [Mie02]. To assure an uncomplicated availability and to conduct a pre-filtering of incoming reports, each group has a separate e-mail address. This decentralized structure of the administrative tasks enables the different groups to work independently from each other. In order that this method functions effectively, and to avoid dependencies as far as possible, the software structure is also conceived in a modular way. This makes it possible to conduct changes in one module without being bound to wait for the completion of another group. Debian consists of over 8,000 so-called packages, each of them constituting a proper project. A *maintainer* is associated to each of these sub-projects, who is responsible for his work. The granularity of the pre-filtering increases further within the Debian project, so that each sub-project has its proper e-mail address composed of the name of the sub-project. In order that the operations can be coordinated, the maintainer observes the progress and classifies the incoming reports according to their priority. In addition, he is in charge of the motivation of the project participants and the allocation of all the resources they require. In the case of groups that are not separated geographically, this can comprise the supply of rooms and technical equipment, as well as the ordering of food [Koc01].

1.7 Management of error reports

For OSS projects, the e-mail has established itself as the favored method of communication. In the course of the further development of this method, the projects have set up *mailing lists*. They receive the different error reports in the form of an e-mail and send them as a copy to all the registered people concerned. By this means, anybody who has registered on such a list disposes of the same information [Mie02]. A positive side effect of this form of communication is its informality. A casual conversation facilitates free thinking and encourages people to submit unorthodox solutions. As e-mails usually contain only text as information, their storage and the search for keywords are indeed easy. In addition, it is possible to implement, through standardized samples and *tags*, an automatic processing, which can be followed by a fault tree analysis.

Ambitions in this direction are pursued by the *Mozilla Project* [Org04], which employs a fault database *(bug tracking system)*. The project calls this database *Bugzilla*, and besides a mere error report it also features its processing status. The *posting* of new errors, their management and the automatic search for conflicts and dependencies represent everything that is feasible for the time being and allows the channelization and target orientation of communication [Wie04].

Like commercial software firms, the Debian projects classify error reports according to their priority, thereby distinguishing *critical*, *serious*, *important*, *normal*, *resolved* and *wish list*. The last category also includes ideas and improvement suggestions related to a specific feature.

2 Software Errors

A major weak point of OSS in the past was the absence of a user-friendly program interface and satisfying product documentation. This task was assumed by so-called distributors (e.g. Suse/Novell, RedHat, etc.), who in addition to the proper compilation of different OSS packages increasingly used to develop interfaces or suggested their programming. As a consequence of the latter point, a change is slowly taking place in the OSS scene – besides the sheer function of a software, their ergonomic handling comes to the fore. However, the principal task of the distributors will remain the integration of different OSS packages into a product, the development of installation routines, the compilation of good documentations and last but not least the supply of maintenance and product assistance *against payment* [uoD00].

2.1 Possibilities of a user inquiry

Besides theory, practice has led to the development of alternative methods to help the users with their problems. Because it is not possible to earn money directly with OSS, but only with the services linked to it, new, unconventional ways had to be found to communicate with the users. Here the Mozilla Project may again serve as an example,

as it uses the well-known principle of *newsgroups* for this purpose. Everybody using the software can exchange ideas with others, describe his problems and search for appropriate solutions. In most cases, he will find an experienced user or even a developer who will deal with his problems [Org04].

To design these inquiries in a way that they are convenient for the program user, Bugzilla offers *user-friendly wizards*, which facilitate the composition of error reports. In contrast, at Debian it is common practice to write e-mail reports which, besides the mere description of the fault, also contain the name of the package. The responsible supervisor always receives a copy of the e-mail, like all the developers registered on the mailing list `debian-bugs-dist` [Gra02]. Especially such important projects like Debian increasingly demonstrate their willingness to register error reports automatically and classified by categories [uSE03]. One of the great challenges in the near future will be the transmission of a corporate feeling to the program user and his integration into the development process.

2.2 Possibilities of user help

It is commonly held that one of the major disadvantages of *free* software is the absence of a claim for support on the part of the producer, i.e. the author of the software [uoD00]. But the fact that such a claim is indeed missing is misleading, because even the commercial suppliers of CSS generally offer their help only against payment. If the user is obliged to help himself, he relies upon detailed product documentation. It may not be sufficient for everybody to take a short look into the program source code and to derive from this how to proceed further. The already mentioned lack of documentation of OSS points to the heart of the problem: if the only help available is the source code, it can easily lead to helplessness. This is exactly the point where the discussion forums come into play again. As a general rule, the program users help each other. So the Mozilla project provides FAQ *(frequently asked questions)* dealing with the users' most burning problems [Org04]. The Bug FAQ can be seen as the counterpart – it lists the most reported software errors for which a direct solution has not yet been found. Like e-mails, these forums can be searched for *key words*.

The product-related Web sites function as a complement to the OSS projects. This is the place where the project and the product are presented and marketed, its features are described and other news are announced [Kru04]. Another important function is the indication of the download servers *(mirrors)*, as they are the place from where the software and the patches can be received and where the user manuals are supplied – as it is the case with *FreeBSD* or Debian. By this means, both projects provide multilingual installation and operating instructions. A paper dealing with the categorization of OSS projects from this and other perspectives can be found in [Mat03]. Another study [uSE03] has found out that the detailedness of documentation increases with the growing size of a project. Especially data base projects have an elaborate documentation.

3 Software Defects

Software programming is a creative process. Usually, only the features of the subsequent program are defined, but not the way to put them into practice. Many developers work together on an OSS project to find adequate solutions, with each of them pursuing his own philosophy. That is why compliance with pre-defined proceedings is very difficult to achieve [uoD00]. Some guidelines for the developers are therefore indispensable.

The danger, especially with iterative development processes, is that only new system components are tested, while the compatibility with the system as a whole is taken for granted. This implies the risk that incompatibilities are detected very late and possibly cannot be corrected anymore.

The *Open Source Definition* (OSD) [www05] does not contain any formal obligation for documentation or specification, as this would constitute an obstacle for many small projects. Nevertheless, the existence of at least a specification is inevitable for the evaluation of a product.

In order that the Open Source concept is not infringed, the OSD contains a passage prohibiting the deliberate *obfuscation* of the source code. It makes no sense if the code is available, but unintelligible through futile identifiers. The direct access to the source code shall induce people to solve problems by changing the program logic. This is by no means disadvantageous – but it is important to make sure that these patches are not integrated too hastily. The programmers or associated inspectors should always check for their system compatibility, to ensure that a patch does not create more problems than it resolves. An advantage is that there is no tight timeframe and that the developers work on a voluntary basis. The absence of deadline constraints and the individual motivation of the software engineers considerably reduce the risk of defects.

The inspection of the source code *(code review)* is a well-tried method of conventional quality assurance and is often employed in the commercial sector. As for CSS, only members of the development section concerned are authorized to do this. If a defect is overlooked by these specialists, it cannot be detected by anybody else [Wie04]. This was probably the starting point and the motivation for the development of the Open Source philosophy. However, it is important to keep in mind that the perceived quality of a software product does not depend on the number of defects, but on the severity of each defect when the product is used [Bac95].

3.1 Defect classification

The simplest and most effective method to classify defects is the function-oriented classification. This means that each defect is attributed to the function most affected by it [Org04]. The Mozilla project, for example, constitutes teams responsible for the six different functions, charged with debugging (see fig. 3).

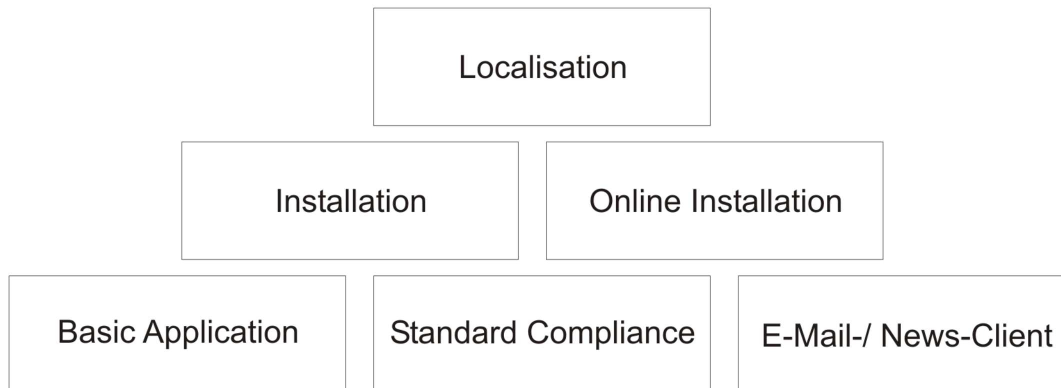

Figure 3: Quality assurance teams of the Mozilla project

3.2 Defect prevention

A golden rule of software development is the conclusion that software always contains defects. This perception results from the fact that these defects are often discovered much later. In many cases, only practice reveals defects causing a malfunctioning of the software. If the goal was the release of absolutely defect-free software, it would be unlikely that it could ever be finished [Mie02]. Therefore it has become accepted, as a pragmatic solution, to deliberately release *unfinished* products. It is expected that defects can be detected and attributed to a function through the practical application of the software. If no more errors are reported, this means just that there remains no function of the software that is affected by a defect. This approach implies that quality assurance has to be applied at an early stage, ideally as early as the specification phase. As already outlined above, the variability of the source code implies the risk of an infiltration with new defects, because anybody can carry out changes [KK00]. For this reason, the patches should always be obtained from the same source, as it unfortunately cannot be taken for granted that all sources providing the software are on the same level. In addition it might be advisable to remain skeptical and to test the new version extensively before its application.

The greatest advantage of OSS compared to CSS is the *"many eyes"* principle. Independent programmers conduct source code reviews and thus can accumulate a competence that no commercial software provider can attain. It stands to reason that not every developer can overlook from the outset the project in which he wants to get involved. Therefore it is necessary for OSS projects to initiate the newcomers into the project work, instead of letting them immediately conduct changes in critical sectors [Hen01].

During the conception phase it is likewise possible to prevent effectively the emergence of defects. To this end, implicit requirements have to be formulated explicitly. Not everyone involved in the project has the same notion of the final product. Especially in a heterogeneous developer community, planning for example to develop several portings of a software for hardware platforms incompatible to each other, the special conditions

accompanying this process have to be taken into account. Furthermore, basic guidelines for the programming style lead to positive results. For example, it is common practice that *coding conventions* and regular source code reviews are already determined at the beginning of a project.

Another approach that proved to be effective especially for OSS projects is the problem-oriented perspective. If a defect is detected, the first step is to identify its consequences [Bac95]. Though this is a very laborious process, it offers the certainty that the defect elimination is permanent. Even if, on the one hand, the elimination of the defect implies the removal of the product deficiency, this should by confirmed by a respective validation. On the other hand, a reasoned weighting of the consequences also implicates a practice-oriented defect classification. In doing so, grave defects can be identified more easily, and those which seem to be serious but in practice have no relevance can be rejected. Finally, the removal of a deficiency can in no way be compared to the elimination of a defect. The practical relevance clearly plays the superior role.

3.3 Defect compensation

Conventional proceedings in the case of a malfunctioning were always aimed at removing the product deficiency. The goal was exclusively the correction of the program functioning [Gra02]. This kind of defect removal may be compared to the treatment of the symptoms of an illness. In contrast, OSS projects are always able to identify the cause of a product deficiency, because normally there is always a specialist working just at the right place – as if a general practitioner was at the same time a specialist for every possible field of medicine. As CSS must generally be purchased, there may be a product liability on the part of the producer, derived from the sales contract. Though the majority of the producers tries to avoid this obligation by exclusion clauses, they do not always succeed. So the producer always considers a deficiency report as an expertise, proving that the product is not free of defects in judicial terms. If the producer then has the obligation to repair his product, this entails additional costs. Deficiency reports submitted by the users are therefore looked upon badly by the producer and concealed.

4 Software Faults

Software faults are not only disagreeable and annoying for those using the software, but also constitute a serious risk. In terms of security, such faults cannot be tolerated and should at any price be prevented from occurring. But this problem is not restricted to OSS; it is rather an indicator for the thoroughness of the system design and the loading capacity of the software architecture. The second aspect can be controlled by a so-called load test and requires a rigorous quality management.

4.1 Unit tests und test suites

Again, a test run begins with a source code review. Based on this analysis, test cases are deduced and compared to the specification. It is particularly important that the tested part of the system does also tolerate values declared invalid according to the specification – as it can't be taken for granted that the implemented part of the system is totally compliant with the specifications. Furthermore, the problem with OSS projects is that they sometimes lack a specification, and that the source code review is therefore the only base for the formulation of test cases [Wie04]. If enough test cases, as well as appropriate data, are available, an automated test run can be conducted, providing reproducible results. In addition, there is the possibility of a productivity analysis, which has identified, via prototypes or previous versions, input patterns on the part of the user, and whose results can be incorporated into the test runs.

If a project and its product increase in size, its testing gets more complex as well. Unsurprisingly, the willingness to conduct detailed tests decreases with the size of the project [uSE03]. Therefore it makes sense to outsource the system test as an independent project and to develop it autonomously. As the tests are then uncoupled from the software development as such, this task can also be assumed by the people using the software. Working with the latest version of the product, they develop appropriate test cases amending the testing project. If possible, *test suites* of the Open Source scene are also employed in the implementation of testing procedures, with their availability also determining their areas of application. For example, there are much more test suites for Java-based projects than for C/C++ projects. A lot of things are going to happen in this area in the near future, especially because C/C++ is the most widely used programming language, therefore generating enormous demand.

From the beginning on, Linux felt the necessity of wide-spread and detailed testing. This does not come as a surprise, because every software running on this operating system depends on it. However, it is very rare that the testing conditions for the new functions and patches are elucidated [Tho03]. In consequence, it is hard to verify if the new program source code is indeed correct. Another example may show how testing procedures can be followed closely: the *GNU C Compiler Project* specifies the test cases conducted for each source code change, in order that the correctness can be proved and retraced. In contrast to the Linux kernel project, it benefits from the fact that the programming language C is specified according to industrial standards, thus offering a reliable reference. As for the Linux kernel project, new functions tend to be inadequately specified, which makes an authoritative validation nearly impossible. Likewise, new functions can rarely be found in written guidelines, or even specifications. The logical reaction of some developers was to focus exclusively on testing.

This has generated a new methodology of testing procedures, namely the development of *micro benchmarks*, running perpetually and testing permanently the functions of the actual source codes. Due to this "smart" procedure, requests from developers can be answered in a couple of hours, which would take considerably more time in commercial processes. The *Open Source Development Lab* (OSDL) even goes one step fur-

ther, by developing the *Scalable Test Platform* (STP, http://www.osdl.org/stp/). By this means, future kernel versions can undergo specified load and performance tests. As Linux meanwhile constitutes an established factor on the commercial server market, an increasing professionalization of the testing procedures is taking place, e.g. *IBM Linux Technology Center*, *Linux Stabilization Project*. To completely automate the whole testing process, OSDL is developing the so-called *Patch Lifecycle Manager* (PLM) and combining it with the STP mentioned above. This makes it possible to gather comparative data, in order to retrace the appearance of new deficiencies in the kernel.

4.2 Mass tests

The basic idea behind mass tests is the notion that the people working with the product constitute an excellent complement to the testing procedures carried out earlier. In this way, the whole sector of product tests gains additional practical relevance. Error reports submitted by the users allow modifications of the program source code, improvement suggestions lead to ergonomic optimization, i.e. improvement of the usability of the product [uSvE04]. A survey [uSE03] showed that more than half of the OSS projects contained justified error reports and that the promptness of the user feedback increased with the size of a project. Apparently the willingness to give a feedback correlates with the number of users of a product. Especially internet applications benefit from such mass tests, as the field test can be carried out on many different platforms. Such kind of tests would be much too expensive for the developers and would in many cases exceed the project resources.

An interesting point of view is discussed in [Gra02]. The developer and user community of OSS is referred to as *"collective intelligence"*, which is not only able to effectively discover product deficiencies, but also to eliminate them with the same success. Thereby the aptitude of this intelligence increases with the number of participants.

5 Software Failures

The collapse of programs logically generates the greatest damages, as rescue interventions can no longer be carried out. In most cases, a collapse prevents people from working with the software. But often enough they lose important data or suffer other kinds of economic damages. Especially for professionals the losses can be considerable. To mitigate this danger, it is necessary to employ the kind of software architecture that takes these risks into account and reduces as much as possible the loss of data or the emergence of damages.

5.1 Plug-in-Architektur

The OSS project *Eclipse* may serve as an example, as it constitutes the basis for many applications. The architecture of Eclipse starts from the so-called *"chameleon principle"*,

which adapts itself to the circumstances [Dau03]. Without any complement, Eclipse serves almost no purpose. It reveals its potential only through its complements *(plug-ins)*. By means of these modules, Eclipse can be exactly adapted to the purpose of an application and represents the ideal solution. The rigorous capsulation of the functional parts makes them interchangeable, and Eclipse develops an almost complete resistance in the case of a malfunction in one of the modules. From a more general perspective, it becomes clear that the OSS principle is reflected in this open architectural concept. Likewise, Eclipse is decentrally conceived and organized, and the functional parts can be developed independently from each other and combined at random [Sau04]. Especially in software development, this is a popular feature, as a whole application "zoo" is necessary to implement all stages of the development process. If a project uses exclusively CSS it becomes dependant on many providers, which leads to an excess of administration effort and investment costs. The ideal solution, in contrast, is again OSS: the solution to the problem is composed of many partial solutions *(top-down)* and combined accordingly. So Eclipse provides a perfect development platform for OSS. The continuous development of new extensions is very likely to amplify this effect in the future.

5.2 Scenario-specific adaptations

As already mentioned, OSS offers the opportunity to tailor application software. Even if an OSS product is already in use and the conditions change slightly, it can easily be adapted, either by doing it alone or by asking the community for help [uoD00]. There is a broad and growing choice of OSS products. As the application context induces their adaptation and selection, the real challenge is to know which products are needed.

6 Product responsibility

The responsibility for a product of the OSS sector is not to be confused with product liability in national jurisdiction. The participants in an OSS project *feel* responsible for their product. In contrast, they cannot be held liable. However, given the possibility of a detailed analysis by independent third parties, the absence of a legal claim does not constitute a problem for the user [uoD00]. If he chose a trusted partner or company, he can act on the assumption that the source code is *clean*. Admittedly, there is no guaranty that the elimination of defect will be carried out promptly. Moreover, the publication of the source code also implicates security risks: the time span between the detection of a defect and its elimination might be cleverly used to inflict damages or conduct espionage. But pursuing this idea, it quickly becomes apparent that CSS, with its significantly higher latency, is rather inappropriate for the application in critical sectors. According to the OSD, the character of an OSS project is well defined, but the geographical dispersion of the participants makes sanctioning of infringements practically impossible. A similar case is the observance of national legislation by OSS

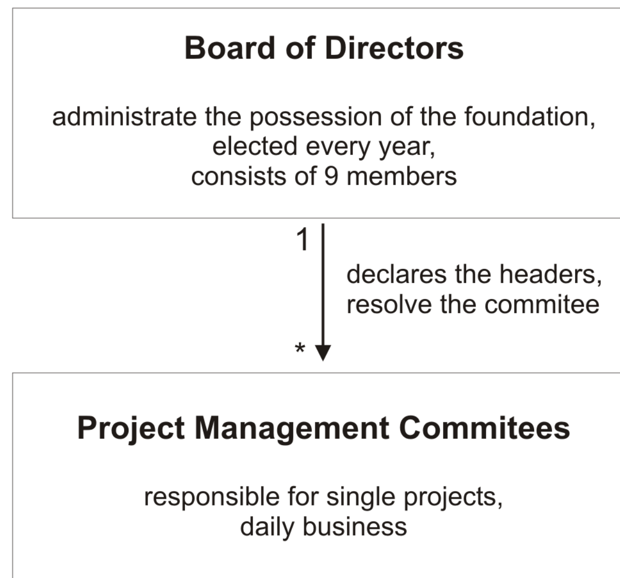

Figure 4: Management of the Apache Software Foundation

projects. These freedoms entail a special responsibility of the user to protect himself. Visiting regularly the websites of projects that are security relevant to the user should therefore become a matter of course. Self-responsibility in using digital data technology will become more important in the future, anyway.

Project teams of the products used should always be the privileged contact persons if software problems occur. Often the responsibility derives from the project structure itself *(generic product responsibility)*, as the respective authors consider themselves responsible for *their* source code [uSvE04]. They offer voluntarily and often promptly solution descriptions *(work-around, patch-work)*, which can be retrieved from the specific project data bases. However, the problem of task assignment increases with the complexity of an OSS project. In most cases the attempt is to subdivide the project still further, in order to develop the different parts of the product independently from each other. But this is risky, as it cannot be guaranteed that there are enough volunteers to develop each part of the product and correct its deficiencies. Furthermore the friction losses increase due to communication, and so does the risk of team fragmentation.

In order that OSS can meet professional requirements, continuous product assistance and the product's further development have to be guaranteed. It is rare that this can be accomplished directly by the project. So specialized companies, so-called *distributors*, offer the first service against payment. Big projects can fulfill the second condition by building well-known hierarchical structures [Kru04], e.g. the *Apache Software Foundation* (ASF), see fig. 4.

In [Gan03], the author argues that bugs are more likely to occur in OSS than in commercial products. But this allegation has to be rejected. It is indeed correct that every-

body can carry out changes in the program source code, but these changes take place under the permanent surveillance of the community (see p. 10). CSS, in contrast, does not have such supervision, because the CSS principle prevents an examination by independent third parties. Moreover, it should be taken into account that the number of programmers in charge of a CSS project is relatively low when compared to OSS. Controls fixed in advance cannot solve this problem. In the case of staff shortage, they can be avoided more easily. Like in CSS projects, changes in OSS projects are recorded and can be attributed to an author at any time. This could even go as far as an author being excluded from a project if he apparently caused damage to the project. Released versions of CSS products, on the contrary, are locked against all modifications and cannot be reviewed by anybody without authorized access to the source codes. This turned out to be an illusory safety, as it was proved last but not least by the cracking of undisclosed cryptographic algorithms. Especially in the security sector, the pursuit of the CSS principle is grossly negligent and will still cause many damages in the future.

6.1 The team model

If somebody wants to launch an OSS project, the question of task allocation comes up. Two fundamentally different concepts have become accepted in the Open Source scene, which nevertheless can be combined. What characterizes the *team model* is that closed developer teams work on a project according to formal proceedings. External developers assist them with sporadic contributions [uSvE04]. The submission of such contributions is carried out according to a procedure verifying their quality, consistency and correctness. Each team works on a clearly defined part of the product – this makes sure that at any time a contributor or the team as a whole can be held responsible. The popularity of this community model increases. It has become common practice for recent projects to integrate the name of an external author in the protocols, so that his contributions to the project may be reconstructed at any time [Gra02].

6.2 The remuneration model

The remuneration or reputation model is based on the voluntary participation of developers in the further development and improvement of a product. Their only motivation is to gain reputation in the developer community. That is why a project is also used as a platform by professional developers to display their own competence and to improve their chances on the job market [uSvE04]. Due to the almost complete absence of an organizational structure, the users also can become part of the developer community. Based on their own experiences in using the product, they prepare appropriated product documentations and actively participate in test runs of new product versions. Thus, not only the authors of the program source codes can feel as part of the product, but the users, too.

The ASF also works according to this principle of an achievement-oriented commu-

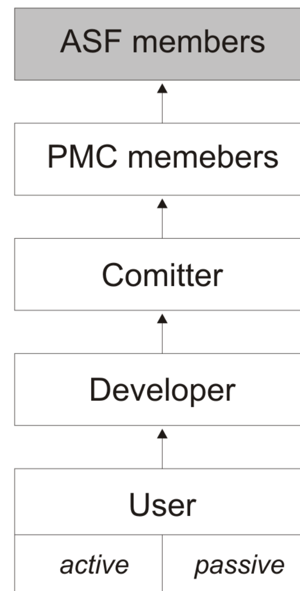

Figure 5: Privilegious hierarchie of the ASF

nity [Kru04]. Particularly ambitious developers gain prestige by submitting constructive suggestions and patches for the product. It has to be pointed out that this prestige is linked to the granting of privileges: the greater the contribution of a developer to a project, the more rights he obtains. From a certain point on, he can himself influence the direction of the product's development. In consequence, a hierarchy on the basis of privileges has emerged (see fig. 5).

Controversial topics are decided by democratic majority rule, with veto rights for the higher privileged. Thanks to the flat hierarchies, such controversies can be resolved quickly. Everybody entitled to vote may choose between *rejection* (−1), *don't know* (0) or *approval* (+1).

In general, it can be stated that projects are often led by veteran or particularly active members [Gra02]. As the product grows with the size of the OSS project, so-called *core teams* emerge, which are responsible for the source code integration. This is already necessitated by the large number of contributions from the external developers.

6.3 The hybrid model

The combination of the two organizational models is termed a hybrid. This integration can be observed for particularly big OSS projects [uSvE04], where whole teams provide sporadic contributions to a sub-project. In this context, the OSS distributors play a decisive role, for they can incorporate their experiences in the fields of distribution and product support. Linux, however, constitutes an exception: there is no official developer team, but six confidants of LINUS TORVALDS [Gra02], who collect all the patches

and suggestions submitted by voluntary developers and filter them. As the founder of Linux, Linus Torvalds has the final say on their utilization and integration.

7 Product certification

If OSS is to be used in a professional context, there has to be the possibility of certification and verification. As this cannot always be guaranteed for OSS, the search for appropriate products turns out to be rather laborious at times [uoD00]. Nevertheless, the advantages of OSS are obvious. The user can see for himself if the program source code is correct and can detect a virulent code or *backdoors*. On principle, the OSD does however not offer the guaranty that the source code corresponds to the executable binary code – a weak point that can be maliciously exploited *(replacement attacks)*. To obtain a *genuine* certification in terms of inward *security* and outward *protection*, all elements of the system have to be certified. This begins with the application software to use and ends with the choice of the hardware on which the software is run. This recursive certification was hitherto only possible in the military sector, where customer and program user are part of one and the same institution. For the first time, OSS offers this absolute security for the civil sector, too. The OSS philosophy can thereby be seen as the consequent continuation of the cryptographic perspective, whose concept is that every undisclosed codification algorithm is unsafe. If a security-critical system has to be kept up to date, this necessitates providing every patch with a digital signature proving the origin of the patch. Consequentially, the history of the product's development can be documented completely, too.

Previous methods of certification were very expensive and complex (concerning the complexity, see [Gru02] about the grant of the internationally recognized *German IT Security Certificate*). Therefore, only *stable* or *released* revisions of a product became certified, which had conducted a *major release*. Today, it is still necessary to synchronize the version changes with their certification. For private users, certified software is certainly desirable, but in the past this was simply unaffordable. To close this gap, the OSS distributors increasingly assume this task. They authenticate the different program source codes and certify their compilations. By this means the user can trust the guaranty of origin of the OSS, so long as he trusts the distributor.

The increasing abstraction of software development processes can also be useful for the certification. For example, an appropriate process could be employed to certify OSS, regardless of its genesis. Such processes are e.g. the *ISO 9000* for producers, *ITSEC* (European) or *CC* (international) (see fig. 6).

The conditions for certification are quality assurance on the part of the producer and the provision of the source code. In principle, neither of them constitutes a problem if the project has been geared to certification right from the start. The certification office examines the specified functionality, the quality of its implementation, the trustworthiness of the project participants and the compliance with security standards. The last criterion might depend on the assigned certification office and on national legislation.

Figure 6: Hierarchy of standard certificates

Assuring the absolute reliability of OSS products still entails a considerable amount of work. Not only the product itself has to be controlled, but all the tools that were employed for its development [KK00]. In theory, a certified program source code might have been converted into machine code by means of a manipulated *compiler*. In such a case, the software can no longer be considered as reliable, and the certificate is denied. For cryptographic algorithms, there has always been an obligation to prove their correctness. No banking company would run the risk to employ uncertified algorithms. Therefore even online banking, though very popular with the private sector, constitutes a security risk, as only very few users pay attention to security certificates. To maintain the reliability of OSS, the certification has to be cross-checked at every stage of the certification process, so that any form of manipulation can be excluded. Even the idea of *Open Hardware* has been forwarded, which applies the principles of OSS to the development and production of technical components.

Robust and safe products have to be based on a reliable foundation, which simply cannot be provided by CSS. The attempt to develop reliable products on an unstable foundation is obviously complicated and costly. Of course, there is always a remaining risk [Neu00]. Nevertheless, different approaches aim at insulating the safe – or the unsafe – part of the product. This can be done by the so-called *wrapper technology* – which, however, is rather an emergency solution than a fully-fledged concept. In June 2001, the certification of Linux by the *Free Software Group* took place, accompanied by the launch of a corresponding test suite (*Linux base Certification Suite*, http://sourceforge.net/projects/lsb). This made it possible to certify Linux versions easily and quickly, which in April 2003 had already led to the issuing of 18 certificates [Ere03].

8 Further development of OSS

One of the greatest risks of CSS is its dependency on special formats of data storage. Only very rarely the competing CSS producers can agree on a common stan-

dard [Gra02]. It may happen that a producer vanishes from the market and that his product is no longer developed. A change of the technical equipment then poses the problem that modification and reading of archived data become impossible. Open standards are the only solution to this problem. So the golden rule for the storage of digital data is to use wide-spread and open standards. Especially for OSS it can be observed that open standards are indeed widely used (e.g. *XML, PostScript, OpenPGP, MP3* etc.) [uSvE04]. This positive feature is particularly helpful if system parts have to be exchanged. As a general rule, an OSS product can be replaced by another, but this should already be confirmed at the outset.

The fact that OSS products are used in many sectors entails a special responsibility of the authors. They should make sure that their products are continuously developed, or at least provide for alternatives. With the professionalization of OSS projects, quality and ergonomics of the products come to the fore; this is especially true for large projects. If a product wants to stand out from the "crowd", sophisticated testing procedures are an appropriate method and a helpful quality indicator. If the number of detected deficiencies or their gravity increase, the further development should be put on ice, or perhaps the spin-off of a new version should be considered [Wie04].

The development of new features of an OSS product normally results from increased demand. The conclusion may be that the further development of a product is induced by – and consequently depends on – its practical application [Jek04]. So the actual situation could be described as a *"war of projects"*, which try to surpass each other. In the context of a "natural selection", only the most appropriate product "survives" and will be developed further. As the appropriateness of an OSS product results from its four-fold quality (see p. 3), quality assurance constitutes a vital factor. An unusual characteristic of OSS projects, which nevertheless can be found from time to time, is the possibility to resume their development after it was abandoned. This is always possible and has to be seen in the context of the take-over by a new project manager [Koc01].

Due to intensive and further development and user feedback, it takes OSS projects much less time to reach the *quality optimum*. This goal is attained when no more discernable corrections have to be carried out [Bac95]. However, it is important to remember that this does not mean that the product is free of defects. In contrast to CSS, the status *"good enough"* does not exist, as OSS is improved as long as there are defects in the program source code – even if they do not have any practical relevance. Besides the classification of defects according to purely qualitative aspects, the risk potential of the defects should also be evaluated *(risk management)*. As OSS is frequently subject to modifications and its fields of application might also change, this risk evaluation is a kind of forecast, aimed at bringing down the probability of malfunction of the software.

8.1 Versioning

It may well be a great advantage that many developers are working simultaneously on OSS projects, but at the same time this implies some additional effort for the ver-

Figure 7: The basics of an OSS project

sion management. The solution to this problem consists in using *repositories*, which are operated automatically by Open Source tools, e.g. CVS, Subversion and BitKeeper (Linux) [KK00]. Here again, mailing lists are employed to keep the community up-to-date about local modifications (see fig. 7).

In addition, each modification has to be accompanied by a comment describing the modification and its cause. As mailing lists inform all registered users with a single incoming e-mail, it is sufficient – after the modification has been deposited in the source code archive – to send an e-mail with the modification comment to the list. If new developers want to join a project, they first have to prove themselves. Therefore, they submit improvement suggestions or patches to the responsible developers. These proposals shall prove their professional competence; and if this competence is judged to be sufficient, the newcomer gets access to source code archive, including write permission.

As OSS projects do without deadlines, the product is released only after the projected degree of maturation has been reached. The version numbers imply the level of modifications. OSS should not be seen as a product in the traditional sense of the word, but rather as a continuous process with open ending [Gra02]. Futurists even perceive an OSS product as a form of artificial life, developing autonomously and continuously producing descendants. The collective intelligence mentioned above (see p. 13) is seen as the creator or breeder. If the product meets the requirements of a milestone, the development of the actual version is stopped and the final tests begin. If they have been completed successfully, the product is officially *released*.

8.1.1 Release authority

So-called *release managers* are responsible for the releases and coordinate them. In case of questions about the integration of source codes into the final product, they are the ultimate instance, deciding whether or not further corrections shall be carried out [Ere03] (see fig. 8).

Of course, every project has to tackle the question of who will assume the responsible task of a release manager. The procedure for his appointment differs from project to project; current methods are nomination, election or a rotation system.

Figure 8: Hierarchy of the release authority

8.1.2 Development stages

As already mentioned above, the product version numbers reflect the development status of the product. Another possibility is a definite identification of the product, or obtaining information about product attributes like stability, range of features, date of creation etc. [Ere03]. The massive parallel development of OSS products often necessitates the simultaneous development of different *version branches*, thereby making it difficult to deduce dependencies and relations between these versions only from the version numbers. In order to better manage the tree-like structure of the version branches, stability "milestones" are fixed. They are called *Alpha releases*, *Beta releases*, or *release candidates*, the criteria for the different milestones depending on the individual project. The purpose of release candidates is to integrate the user into final tests, in order to increase the quality of the definitive releases. Product users are then in charge of validating the releases, while teams provided by the developer community are responsible for the verification. Other measures to assure the quality of the releases, like control boards or release committees, have proved to be helpful for big and medium-sized projects.

8.1.3 Distribution of releases

It is important to promote the release of a new product version – the *user community* has to be informed about its availability and the advantages of the new release [Ere03]. The better people are informed, the more they will be interested in future releases. The integrity of a release must not get lot lost, under any circumstances, on the way to its practical application. Its authenticity has to be ensured on the whole *delivery path*, e.g. by using checksums and appropriate certificates. The distribution of the releases can also be assumed by third parties, or they can be packaged to form meta products, as it is the case with Linux distributions or *MikTex*. The task of the distributor is the *packaging* of OSS in different formats, e.g. RPM packages for Linux or installation programs for Windows.

8.1.4 Version number assignment using the example of the Linux kernel project

The Linux kernel project uses the format *x.y.z* for its version numbers, with the variables having the following functions:

- x stands for releases containing many and radical changes,
- y stands for releases with few and small modifications,
- z describes the patch level of the release.

The particularity of steadily running releases is an even y, while it is uneven for unstable releases. Unofficial releases contain, additionally, the initials of their author, and release candidates can be distinguished by the suffix `-pre`. The unstable releases are not destined for practical application, but serve software producers as testing platform and technology preview.

8.2 Security aspects

In comparison to CSS, the transparency of the source code of OSS facilitates highly accelerated revision cycles and provision of patches. In the ideal case, a patch is available at the same time as the corresponding error report [KK00]. Especially the similarity of the concepts of OSS and cryptography concerning the visibility of the algorithms leads to the assumption that cryptography served as a model for OSD. Confidence is mostly based on conviction / persuasion. Reutilizing a (certified) program code can build confidence, but in the end, the goal should always be the certification of the product as a whole (e.g. certification of the *SuSE Linux Enterprise Server V8* by the BSI [fSidI03]), as transmission or heredity of certificates are not possible (see chapter 7). OSD creates ideal conditions for a safe program source code, but does not provide guaranteed security. Instead of a quality optimum in the context of security, there is rather a dualism of the security-relevant qualitative features of an OSS product – it is either safe or unsafe [Bac95].

For security-relevant products, a risk weighting function has to be developed empirically, allowing conclusions about the *tolerable degree* of maturity of a product (see fig. 9), i.e. from which revision on a product can be used. The evaluated risk can oscillate between two extreme values: 0% corresponds to the formally proved correctness of the product, 100% stands for the certainty that the tested part of the product is deficient. Thus, the certification of the whole product should take place only after all critical parts have reached the level of acceptance. This level can be defined as the willingness of the people using or buying the product to accept risks up to a certain point – which can be measured quantitatively by user or client surveys.

So, as to the reliability of software, two things should be kept in mind: confidence in a software product requires the accessibility of the source code; substantial and effective security necessitates its adaptability to special conditions. As CSS can fulfill neither

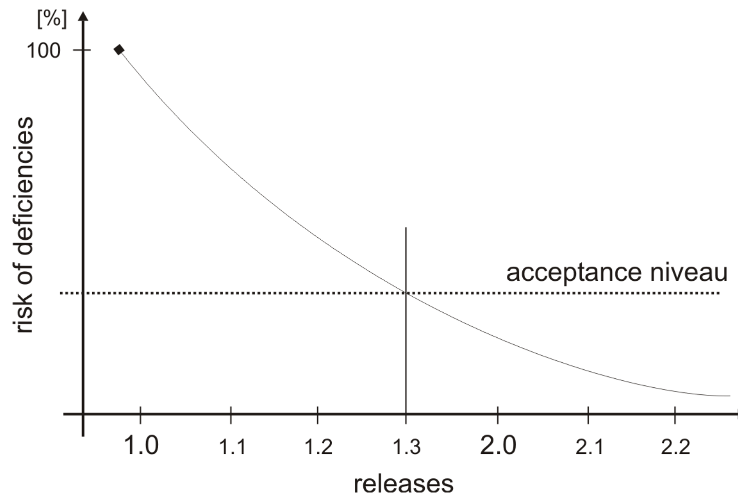

Figure 9: Deficiencies risk estimation

of the two conditions, it should be categorically rejected as untrustworthy [Bac95]. The problem OSS has to deal with is of quite a different kind: OSS is new and therefore unfamiliar to many users. Little by little, confidence will be built up during a familiarization period. In contrast, the established CSS enjoys unjustified confidence, only due to the fact that it is well-known and widely accepted. This attitude is not only risky, but might cause serious damages.

9 Summary and outlook

The goal of this paper was to emphasize the difficulty of conceptualizing and evaluating the term "quality". OSS projects also have to face the problem of quality assurance, particularly as nowadays, the professional IT sector cannot be imagined without OSS. Effective quality assurance requires the rational definition of priorities [Bac95]. Thus quality assurance can also be conceived as the art of making compromises. This can be compared to politics, where it is also essential to set priorities.

Absolute quality does not exist. The only way to achieve it would be the formal verification of software, which reaches unfeasible proportions, even for small OSS projects. So the only solution is approach high quality as much as possible. In most cases, experience will show if a product has already reached an acceptable quality – this should be kept in mind for every OSS project.

The trend indicates an increasing professionalization of Open Source. Due to its huge cost-saving potential, this method of software development is economically alluring. In the medium and long term, the Open Source idea is expected to be applied to hardware development, so that a continuous certification of complete *Open Systems* becomes possible (first approaches to a *FreeBIOS*, cf. [Sta05]). Current efforts focus on

improving documentation and ergonomics of OSS.

References

[Bac95] James Bach. The challenge of good enough software. *American Programmer magazine*, 1995.

[Dau03] Berthold Daum. *Java-Entwicklung mit Eclipse 2*. dpunkt.verlag GmbH, Heidelberg, 2003.

[Ere03] Justin R. Erenkrantz. Release management within open source projects. In *3rd Workshop on Open Source Software Egineering*, pages 51–55. International Conference on Software Egineering, 2003.

[fSidI03] Bundesamt für Sicherheit in der Informationstechnik. *Certification Report BSI-DSZ-CC-0216-2003*. Bundesamt für Sicherheit in der Informationstechnik, July 2003.

[Gan03] David Ganster. Sicherheit up to date. *IT-Security-Special 4/2003*, 4:39–40, 2003.

[Gra02] Volker Grassmuck. *Freie Software - Zwischen Privat- und Gemeineigentum*. Bundeszentrale für politische Bildung (bpb), 2002.

[Gru02] Dr. Ernst-Hermann Gruschwitz. *Zertifitierungstag 2002: Aktuelles aus der TÜViT-Zertifizierungsstelle*. TÜV Informationstechnik GmbH, June 2002.

[Hen01] Elisabeth Hendrickson. *Better Testing – Worse Quality?* Aveo Inc., 2001.

[Jek04] Sebastian Jekutsch. *Diverses zum Thema Open-Source-Entwicklungsmodell*, December 2004.

[Kad00] Prof. Dr.-Ing. Firoz Kaderali. *Ansätze zur Qualitätssicherung und Rückkopplung der Campus-Source-Nutzer*. FernUniversität Hagen - Fachbereich Elektrotechnik, June 2000.

[KK00] Marit Köhntopp und Andreas Pfitzmann Kristian Köhntopp. *Sicherheit durch Open Source? - Chancen und Grenzen*. Secure Electronic Marketplace for Europe (SEMPER), July 2000.

[Koc01] Dr. Stefan Koch. *Entwicklung von Open Source und kommerzieller Software: Unterschiede und Gemeinsamkeiten*. Wirtschaftsuniversität Wien, 2001.

[Kru04] Stefan Kruber. *Die Rolle des Internets für Open Source Projekte*. FH Regensburg, November 2004.

[Mat03] Martin Matuska. *Kategorisierung von Open Source Projekten*. Institut für Informationsverarbeitung und -wirtschaft, Wirtschaftsuniversität Wien, 2003.

[Mie02] Caspar Clemens Mierau. *Motivation und Organisation von Open Source Projekten*. Bauhaus-Universität Weimar, 2002.

[Neu00] Peter G. Neumann. Robust nonproprietary software. May 2000.

[Org04] The Mozilla Organization. *Mozilla Quality Assurance*, November 2004.

[Sau04] Heinz Sauerburger. *Open-Source-Software*. dpunkt.verlag GmbH, Heidelberg, August 2004.

[Som04] Ian Sommerville. *Software Egineering*. Addison-Wesley, Longham, Amsterdam, May 2004.

[Sta05] Richard Stallman. *The Free Software Foundation's Campaign for Free BIOS*. Free Software Foundation, February 2005.

[Tho03] Craig Thomas. Improving verification, validation, and test of the linux kernel: the linux stabilization project. In *3rd Workshop on Open Source Software Egineering*, pages 133–136. International Conference on Software Egineering, 2003.

[uoD00] Arbeitskreis Technische und organisatorische Datenschutzfragen. *Transparente Software - eine Voraussetzung für datenschutzfreundliche Technologien*. Der Bayerische Landesbeauftragte für den Datenschutz, November 2000.

[uSE03] Luyin Zhao und Sebastian Elbaum. Quality assurance under the open source development model. *The Journal of Systems and Software*, 66:65–75, 2003.

[uSvE04] Markus Pasche und Sebastian von Engelhardt. *Volkswirtschaftliche Aspekte der Open-Source-Softwareentwicklung*. Friedrich-Schiller-Universität Jena, 2004.

[Wie04] Stephan Wiesner. *Qualitätssicherung von J2EE Anwendungen V1.0*, November 2004.

[www05] www.opensource.org. *The Open Source Definition*, 2005.