Open Source Engineering Processes

Open Source-Entwicklungsprozesse

Wolfgang Mauerer*, Siemens AG, Corporate Technology, Erlangen,
Michael C. Jaeger, Siemens AG, Corporate Technology, München

* Correspondence author: wolfgang.mauerer@siemens.com

Summary Software engineering in open source projects faces similar challenges as in traditional software development (coordination of and cooperation between contributors, change and release management, quality assurance, …), but often uses different means of solving them. This leads to some salient distinctions between both worlds, especially with respect to communication and how technical issues are addressed. The variations within open source software (OSS) communities are considerable, and many different approaches are currently in use, ranging from informal conventions to highly systematic, formally specified and rigidly applied processes. We discuss the archetypal best practises in the field, illustrate them by presenting example projects, and provide a comparison to traditional approaches. ►►► **Zusammenfassung** Software-Engineering in Open Source-Projekten begegnet ähnlichen Herausforderungen, die auch in der klassischen Softwareentwicklung auftreten – angefangen bei der Koordination beteiligter Parteien über Veränderungs- und Versionierungsverwaltung bis hin zur Qualitätssicherung. Allerdings verwenden Open-Source-Softwareprojekte häufig andere Lösungsmethoden. Daraus entstehen substantielle Unterschiede zwischen beiden Welten, insbesondere bezüglich der Art und Weise, wie Kommunikationsfragen und technische Probleme gelöst werden. Innerhalb der Open-Source-Welt variieren die Problemlösungen von Projekt zu Projekt; es existieren viele unterschiedliche Ansätze, die den Bogen von informellen Konventionen hin zu systematischen, formal-definierten und stringent angewendeten Prozessen spannen. Dieser Artikel beschreibt archetypische Erfolgsmethoden, illustriert sie anhand von Beispielprojekten, und vergleicht die Methoden mit traditionellen Ansätzen.

Keywords ACM CSS → Software and its engineering → Software creation and management → Collaboration in software development → Open source model; open source software engineering; software engineering processes, software releases, Linux
►►► **Schlagwörter** Open-Source-Softwaretechnik, Softwareentwicklungsprozesse, Softwarereleases, Linux

1 Introduction

While the core ideas of open source software (OSS) have been around for several decades (albeit under different names), extended interest in the differences between OSS engineering processes and traditional development techniques has only been sparked a few years ago, when the first projects that were developed in an open, collaborative manner started to compete with traditional solutions, match their capabilities, and finally even exceed their market shares. Consequently, it is interesting to study in which aspects OSS engineering differs from traditional techniques.

There is no single OSS engineering process, and it is obvious that there cannot be one. However, OSS projects necessarily share the commonality that their source code is available to the general public – although even the terms under which the code may be modified and used differ wildly. Every project follows its own set of rules that may or may not overlap with the approaches of other efforts. However, there is a distinct "flavour" attached to OSS engineering that we will try to distil in this paper: Partly based on existing literature, partly based on our own experience. Specific example cases are used to provide facts that underline or exemplify the statements.

Note upfront that the license of a project does not provide any indication of what engineering processes are used. Two factors guide the engineering process: On the one hand, the contributors to a project; on the other hand, the project's development history and experiences. Naturally, OSS projects have to face the same challenges

that occur in general software development. We will particularly focus on *collaboration* (how developers exchange information beyond code, how decisions are reached, and how new developers are introduced to the mechanics and styles of a project), *technical aspects* (how code is written, what stylistic and technical requirements it has to fulfil, and how new code is integrated and merged into a project), *documentation* (how information for users, but also for new developers is provided, and how code is documented), *testing and release management* (how it is ascertained if and when a piece of software is sufficiently free of bugs to be released), and *maintenance* (how the communities deal with the need to fix issues in already released software, and how the longer-term availability of code is secured).

After discussing generic possible approaches of the OSS communities to these problems, some examples are given for specific, well-known projects. Finally, we illustrate the differences between traditional and open source engineering in more detail.

2 Best Practices

2.1 Collaboration Issues

Communication between stakeholders is known to be one of the important building blocks of software engineering, particularly when developers work in dispersed global teams with intercultural issues. For many open source projects, development is far from happening behind any walls that could concentrate communication. Globality is the rule, not the exception.

In stark contrast to classical software engineering, the best amount of communication for OSS projects is done in writing – dominantly via mailing lists, but also via chats, message boards, and other forms of network-based communication. The amount of traffic captured on central archiving sites like http://gmane.org, together with the geographically dispersed nature of open source development, is a strong indicator for this behavioural observation. Periodic status phone calls to bring spatially distributed developers together are established for some larger projects, but unlike, say, daily scrum meetings, they are only an extension to textual communication, not a crucial component of the development process. This has two important implications:

- Discussions are intended to be as technically focused as possible, but the arising sterile and especially impersonal nature of communication (often, contributors to a thread of discussion do only know the names of the discussion partners, but need not have any information about their personal habits, attitudes or idiosyncrasies that are easier to obtain in other communication scenarios – in fact, even the gender might be unknown) lowers the barrier to drift into inappropriate, rough, sometimes even explicitly insulting language in the sake of technical clarity and directness (missing non-verbal communication [6] may be one explanation for the phenomenon). While most projects deem such behaviour inappropriate and undesired (some communities even formulate explicit rules of communication to prevent such behaviour [13]), it cannot be prohibited by any "official" authority, and can increase the barrier towards new contributions by project outsiders.
- Information and the content of discussions can be distributed to a large number of developers without involved technical measures – a simple, publicly accessible archive is sufficient. However, mistakes that are pointed out on a public mailing list remain public forever, as do technically flawed or inaccurate statements. Depending on the cultural background and personal attitude of individual developers, this can easily act as an inhibition for contribution.

To address the negative aspects of partly anonymous communication, many projects do organise regular meetings (conventions, hackfests, summits, …) to bring developers together face-to-face. Typical examples include ApacheCon, PyCon or LinuxCon. It is established practise to organise development sprints as part of larger conferences to work on currently open problems in close spatial proximity. The KDE community, for example, participates in 15–20 such sprints per year, see http://community.kde.org/KDE_e.V./Sprints. Consequently, active open source developers tend to invest some time in travel.

Not only substance, but also the form of communication is a factor that receives consideration especially in larger projects: When hundreds of messages are exchanged per day and need to be at least briefly glimpsed over by many developers, it becomes increasingly important to employ common conventions on how to describe problems or how to report bugs. Even the question where an answer to a previous email is placed – on top of the old message, or on bottom of it – can be deemed as a crucial ingredient of efficient communication [7].

Open and public communication lowers the barrier between developers and users: Non-programmers can not only observe, but directly participate in discussions about the project, and can gain immediate influence on the future directions. This strengthens the bonds between users and developers, and is rarely possible with closed source software.

2.2 Technical Aspects

The core idea of OSS is that shared development on common code produces advantages that range from concerns about freedom (as in speech [24]) to commercial benefits. This implies global cooperation involving multiple companies, and poses numerous technical problems. The very technical aspects of producing common code form therefore one of the distinguishing factors of open source engineering.

Regardless of the specific cooperation and management style of a project, there are three main contributor

roles: *Developers* who create code and submit it to a project; *maintainers* who decide whether the code is accepted, needs to be revised, or is rejected; and *reviewers*, who evaluate the quality of new code (contributors usually act in multiple roles). This structure poses several challenges:

- Developers need to be able to work independently of other developers, and the independence must particularly be maintained in the presence of large numbers of other contributors. With OSS development being inherently less synchronised than traditional methods, the ability to change overlapping portions of code and re-integrate the changes into the main code basis requires more support on the technical level, in particular efficient branching/merging capabilities, and the possibility to re-base ongoing development work onto the latest changes in the mainline distribution.
- Likewise, maintainers must be able to deal with submissions that address overlapping portions of the code, and need to be able to solve the arising conflicts efficiently without time-consuming manual work.
- Reviewers must be able to evaluate contributions with as little reference to external information sources (i. e., the patches are ideally self-contained), and review results need to be accessible to other reviewers and developers for discussion. In the absence of design documentation and upfront planning, code itself is usually the basis for reviews, which happen on a project's main communication channels, typically mailing lists.

While many older revision control systems claim to satisfy these requirements, practical experience gathered in large projects has falsified many of these claims, and led to the creation of massively distributed VCSes like git, mercurial or bazaar.

Instead of integrating large chunks of development work at a time into the VCS, changes in OSS projects are usually presented in the form of *patch stacks*: Collections of modifications that address a single issue (implement a new feature, fix a deficiency, etc.), but are structured into separate steps that perform one small, self-contained modification at a time. Each of these modifications is ideally accompanied by a short explanation of what the change is good for, why it is required, how it works, etc. This structure simplifies not only patch review, but eases the partial merging of features (pick useful components, and omit undesirable contributions). It also implicitly documents design decisions and modifications done on the project. Depending on the rigour with which these means are applied, it may also be required that the patches in the stack are *orthogonal*: A software system that at least builds properly must arise when only the first n patches of the N patches of the series are applied in correct order. This is similar to the product increments of scrum, but is more fine-grained, and serves a wider purpose: When a bug is known to be *not* present at revision r_j, and present at revision $r_k (j < k)$, it is possible to build all intermediate revisions of the software and determine which change introduced the bug in an automated manner – see Ref. [16].

A common requirement is that contributions and patches adhere strictly to a given coding style, which eases not only merging and discussing new contributions, but also makes it easier to modify existing code. Many projects include tools to determine if a given patch set fulfils all required criteria.

2.3 Documentation

OSS projects typically don't provide a comprehensive collection of documents that describe all aspects of a project from low-level details to high-level design documents, as is traditionally suggested [8]. Design decisions and other choices that shape a project are, however, implicitly documented on the mailing lists. Alternate design choices that were rejected can be found there. This makes more information available than when decisions are based upon personal or group discussions that are rarely recorded in their entirety. However, it is more difficult to find the appropriate pieces of information from an unstructured data source.[1]

A considerable amount of information is implicitly contained in the revision control logs, which is a second source for design rationales. Another popular method is to employ integrated documentation tools like doxygen to provide information at the API level; the raw material is directly contained in the source code, and can be kept up-to-date with comparatively little effort.

2.4 Release Management

Managing releases is a highly project-specific endeavour, with approaches ranging from fixed, periodic release dates to releasing code only when it seems finished to specific deciders that may or may not follow formal criteria to arrive at the conclusion. Many projects employ a multi-stage strategy before releases are cut: After a merge- or development period in which completely new code and far-reaching changes to existing code can be added to the source base, a testing phase (typically termed *release candidate* or, more like in traditional terminology, *beta* phase) is started, and only error-correcting changes and uncontroversial small, focused improvements may be merged into the source base. After handling integration problems and settling a satisfactory fraction of open issues, a release is cut.[2]

Operating system distributions tend to follow more formalised release schedules: The Linux distribution

[1] Some developers have established the habit of providing a link to mailing list discussions that took place during the development of a feature in the commit logs, but this is by no means a universally accepted practice.

[2] Since most open source projects are not concerned about shipping physical media or printing physical documentation, this usually means nothing more than providing a tag in the revision control system, and sending an announcement to the appropriate mailing lists.

Ubuntu provides two releases per year, one at the end of October, another at the end of April. OpenBSD also features two releases per year. Enterprise distributions (for instance Red Hat Enterprise Linux) typically also offer a post-release management that guarantees updates and back-ports over an extended period of time, typically spanning several years.

2.5 Testing

The reasons for testing in open source projects are not much different than for testing in traditional software engineering. Many of the widely-deployed unit test frameworks are released under open licenses, and most OSS projects include unit tests nowadays. There is, however, usually little system or integration testing on the level of individual projects; distributions are the exception: Making sure that the components of a distribution interact in the desired manner, and that changes in individual components do not introduce faults at the system level is an obvious desideratum for them.

This lack of comprehensive explicit testing is partially compensated by implicit testing: Users and especially developers can operate with arbitrary snapshots and intermediate versions of the software, which implicitly leads to a continuous testing of the components in the individual use-cases and scenarios the intermediate stages are deployed in.

2.6 Maintenance and Stable Versions

Maintaining stable versions of software is required for essentially two reasons: When the software is deployed in scenarios with high stability requirements that can only tolerate a small amount of changes (for instance, when safety/security certifications are required, when system robustness is more important than the benefits of the latest features, or when interface stability is needed). The first requirement is usually satisfied by maintenance branches that, from a given revision onward, ports only important bug fixes from the development branch into the old tree, but omits new features. Maintaining such branches is a time-consuming venture, and only larger projects or projects with explicit commercial funding can sustain such measures – smaller projects typically mandate users to update to fresh releases when critical bugs are fixed.

API stability (which explicitly *excludes* ABI stability) is less of an issue for OSS: Components that build on top of an existing framework are required to be available in source form when the basis license is restrictive, which enables communities to continuously update components to new APIs, rendering stable interfaces mostly pointless. For permissive licenses, one notable scenario is when proprietary extensions or plugins are built on top of an open framework (e. g., SugarCRM). Interface users then need to take care of providing a stable API in the base platform when they don't intend to continuously update their closed components, or ensure that binary-only releases continue to work with new versions of the base platform.

3 Project Examples

3.1 Linux Kernel

As the largest and most active open source projects by the usual descriptive standards, the Linux kernel serves as an engineering role model of high influence for many other projects, especially in the low-level and middleware regime. This makes it a particularly interesting target for closer evaluation. During the last two decades, the project underwent several considerable changes how the development process was organised, usually caused by a growing number of contributors that brought the previous process to its scalability limits. While we won't provide a detailed historical record of the changes, it is worth noting that the distributed revision control system git was specifically invented to deal with these problems.

The current process is semi-formally documented (see Ref. [2], and Ref. [7] for a more detailed description), and comprises of the following stages to merge modifications into the mainline kernel.[3]

- The initial requirement collection and design discussions are performed on the main kernel mailing list (LKML). However, it is also an established pattern that these steps are done without community involvement, driven by needs of a developer or vendor.
- The patch stack is posted to subsystem-specific mailing lists for discussion, is reviewed, and iteratively improved and re-posted. It is not unusual that five or even more iterations are required until all review comments have been addressed.
- The subsystem maintainer merges the patch into a subsystem-specific tree, where it is subjected to review from a wider range of stakeholders, for instance maintainers of other subsystems. This may again raise various issues that need to be addressed.
- Once all issues have been addressed, the subsystem maintainer sends a pull request for the new features, which are eventually merged into the vanilla tree, that is, the tree tended by the Linux core maintainer. However, it can also be decided that the feature is rejected, which requires a complete re-engineering or re-writing of the approach.

Note that the project features a hierarchical maintainer structure, and depending on the area of interest, patches can flow through multiple maintainers, extending the previously described process accordingly.

The Linux kernel employs a rigid release schedule: New code is only integrated during a two-week *merge window*, and the approximately six following weeks are used for testing and debugging. Interim releases cut during this

[3] We intentionally omit the staging mechanism that was established to bring device drivers of sub-optimal quality into the main kernel tree as early as possible.

phase are termed *release candidates*. The fixed release schedule must not be confused with upfront planning: The set of features integrated during one release cycle is only determined by the state of the code during the merge phase, and is not influenced by any (public) road-map.

Besides the vanilla tree, other important trees exist:

- The *stable* trees collect security- and otherwise critical fixes and apply them to a series of stable kernel revisions. By means of special patch annotations, it is possible to simplify the process of ensuring that the change ends up correctly in all relevant trees.[4]
- The *-next* tree is an integration tree that collects (at short intervals) all patch sets intended to be merged during the next release cycle. The goal is not to produce a stable kernel, but to detect any interference between otherwise unrelated upcoming patches early.
- Individual developers typically maintain a number of *topic trees* dedicated to their particular interests. Once a feature has reached a stable state, the changes are submitted to the appropriate maintainer.

3.2 Android

The Android project serves in two roles: As a distribution of various components that may be equipped with slight modifications, and at the same time as a replacement for classical embedded Linux distributions, accompanied by considerable and far-reaching changes to the traditional Posix programming model that require own developments all across the stack. Consequently, the project has to cover two different engineering aspects: On the one hand, base components are taken from the OSS ecosystem and augmented with patches to fix critical issues or provide desired enhancements, while avoiding to create a privately maintained fork. For several large and important external projects (the Linux kernel, OpenSSL, WebKit, …), the preferred policy is to fix issues upstream, and benefit from the changes by updating the component to a new release. On the other hand, the development of custom, Android-specific components needs to be handled.

The project code is distributed in a large number of git repositories; a special-purpose tool (`repo`) was designed to ease simultaneous updates of all components. The structure keeps contributions from outside sufficiently disentangled from the custom development, yet integrates all components into a coherent whole.

Submitting patches is done via a fully documented, tool-based process (see Ref. [1]) that is roughly summarised as follows:

- Patches are prepared in the author's environment of choice.
- A patch is submitted for review to the *Gerrit* system, a web-based tool to handle the review process.
- Approvers are notified about the patch, and can provide "verified" (after ensuring that the code builds and functions properly) or "reviewed" (following a code review) tags.
- If the patch is accepted, it is automatically merged into the current state of the trees by gerrit. When the merge fails, the code can be either fixed by the approver, or be sent back to the author to adapt it to the current state of the code.

The project does not formally distinguish between Google-internal and external contributors, although most development on the custom components is done internally form Google, which sees itself as responsible for engineering, marketing, and core platform development.

In contrast to the Linux kernel, there are no explicit stylistic requirements on a patch that are checked with automated tools. However, review questions that have to be addressed when patches are considered for integration are formally documented [1] and range from potential design flaws over good use of best practices to security or instability risks.

3.3 Apache Software Foundation

The Apache Software Foundation is an organisation to host and support various open source projects. Currently, about 200 projects are hosted by the foundation, with the most popular one being the Apache HTTP server. Apache projects share commonalities not only for the open source license, but also for the software development process that has evolved over the years.

The foundation provides a description of the engineering process [9] that defines the basics of collaboration. The project model does not define roles with assigned responsibilities (e. g., architects, testers, …), but uses a voting model to take decisions which are discussed using the Apache hosting infrastructure (e. g., mailing lists, issue trackers, project wikis, …).

Voting represents an essential instrument for decision making, and the following cases are distinguished: voting for procedural activities in a project, voting on code modifications or voting for releasing a package [10]. This requires different voting rights for individual contributors. The particular rules for voting can be defined by each project. One important alternative to voting is the concept of *lazy consensus*, which implies acceptance if involved persons remain silent. Given that an OSS project represents a distributed effort, the lazy consensus avoids blocking situations when involved persons cannot contribute to decisions in a timely manner.

Two general modes for source code votes are distinguished: In the lazy case, the contributor can commit source code, which is reviewed only afterwards (*commit-then-review*). This procedure appears more suitable for starting projects. Alternatively, the contributed source code is first reviewed, then a voting is carried out and if a positive result is obtained, the source code is committed (*review-then-commit*).

For working on source code, the foundation provides suggested work-flows. For example, it describes the re-

[4] See www.kernel.org/doc/Documentation/stable_kernel_rules.txt for the details.

lease management for incubator projects [11]. However, individual projects can define their own release management or their own way of handling contributions. Detailed characteristics about the applied processes are provided by the individual projects.

4 OSS Engineering versus Traditional Processes

Software engineering processes are an old and well established topic. While the waterfall model, one of the earliest approaches in the field, tried to apply a classic engineering approach to the construction of software, the deficiencies of the approach became quickly apparent: The created asset, software, can be modified, changed, and replaced to a much greater extent than physical goods, which rendered the process too inflexible. Adapted with iterations and feedback loops in subsequent efforts, the waterfall model still represents a classic idea of developing software.

We consider two other examples of software engineering approaches from the 90ies: The V-Model, which is intended for general IT projects in the governmental area including the engineering of software [14], and the Rational Unified Process (RUP) [15], which was considered a trend due to the adoption of the Unified Modeling Language (UML) on which the RUP built upon [17].

Both approaches differ from each other: The V-Model separates engineering into two halves: the first half defines requirements and specifications steps with increasing level of detail and the second half describes the according verification of these steps with decreasing level of detail. Furthermore, the V-Model provides precisely expected assets and responsibilities of the involved roles. With checkpoints for comparing the outcome with previous set definitions, the project cannot deviate easily from initial definitions.

The V-Model ensures a high degree of security when it comes to delivering the originally intended assets. However, a project cannot be easily adapted in the presence of problems, or when outside settings change. Unexpected challenges are not optimally dealt with. The model fits the administrative and economical interests of a project: A customer (or funding party) is provided with a clear view on what to expect as early as possible. When many sub-contractors are involved, the unambiguous definition of deliveries is an important asset.

The RUP, having evolved from the efforts of the Unified Modeling Language (UML) approach, represents a framework that can be tailored to create an individual process. The engineering process addresses iterative and feedback-loop based software creation to improve adaptability in case of unexpected challenges. Notably, the approach demands high discipline to use models not only for documentation or as a means of collaboration, but also as the specification of a software system.

Notwithstanding other efforts that are not discussed here, the previous decade unearthed an apparent problem of the software industry: Surveys report a project failure rate between 10–50%, depending on the definition of project failure. While hard numbers are difficult to determine [19], estimates exist that project failures cause a waste of billions of dollars [3], which is a huge potential for cost savings. Improper requirements engineering and the lack of software engineering know-how training for developers were among identified main causes.

More importantly, though, it was claimed that the traditional way of engineering software was apt for standard projects that mostly consist of adapting existing pieces for the needs of a particular customer. When it comes to the development of new approaches, a more flexible engineering process was supposed to be necessary.

One effort originating in the early 2000s is extreme programming, notably the approach by Beck [18]. Subsequently, the agile software development movement evolved (see http://agilemanifesto.org/), which influenced the recent years of software development. In a nutshell, agile software development places the development of features in the centre of development planning and the engineering process. The Scrum process framework is among the most popular agile engineering approaches [20].

It could be argued that "classic" software development efforts traded the certainty of achieving exactly what was demanded for efficiency. However, software projects demand a high degree of liveness in terms of coping with unexpected challenges, and the original problem was to reduce the number of project failures. The term *liveness* roots from the research about reliable systems and describes in general the ability of a system to result in "something good" even if unforeseen events happen during execution [21].

4.1 Differences

As we have outlined in the previous section, more recent software engineering processes improve liveness of a project by better adapting the engineering techniques to a changing and dynamic environment or a shifting project focus. OSS engineering processes need to cater for the desires of a variety of projects and an even larger variety of contributors. Since there is no coupling to default processes of organisations, a wide variety of diverse processes resulted in consequence. These processes are optimised for the immediate goals of a project.

However, among this diversity, a generic open source engineering attitude exists, which is described by the adage "release early, release often"; another adage says "start fast, fail fast". This means that starting software engineering very quickly with less upfront planning will lead to more transparent progress and therefore result in a smaller extent of failure. In addition, the public submission of defects or issues involves externals from the project adding transparency to the software quality. Of course, open source projects cannot provide a guarantee that all reported issues will be solved at a defined time. But the unfortunate project case of a hidden serious

"show stopper" revealed at late phases of the development becomes less likely with that additional layer of public quality feedback. If there are defined schedules for releases of decisions to release only after a set of significantly blocking issues are resolved, defined goals with respect to software quality are implicitly set.

With a reduction of the rate of project failures as most important goal of software engineering in mind, this transparency represents a considerable asset. The earlier a project failure is recognised, the less resources are spent towards the wrong direction. This is one of the goals of agile development, but also marks the main strength of OSS development: efficiency. At every stage of the project, the involved persons are as well-informed as possible about technical risks and therefore about the entire risk of failure.

However, there are also drawbacks of OSS engineering as compared to traditional approaches: There is no guarantee what is going to be built before the actual creation of the software. Open source projects can be called incident driven instead of implementing upfront requirements engineering, as in the traditional software industry. Contractual safety about the outcome does not exist.

Two major trade-off goals for projects need to be considered when comparing OSS engineering to traditional methods: Firstly, safety, namely a guarantee that software is exactly built as specified (using requirement specifications or contracts). This covers functionality as well as trade-offs to functionality, time and budget. Of course, the more innovative a software project is, full safety regarding functionality, budget and time can never be achieved. Secondly, a degree of project liveness, which refers to the the level of adaptability. When a project is being worked on, it is desired that it can cope with unplanned situations and adapt in a flexible manner to unforeseen changes.

Different studies discuss the nature of the OSS engineering process and their differences to other engineering processes (see, for instance, Ref. [5]). For example, OSS engineering involves public feedback, an element not found in most engineering processes applied to proprietary software. Public issue trackers, open code, the use of mailing lists, etc. are measures to ensure a high degree of liveness. The safety of the process is rooted in tracking issues, instead of doing general planning upfront. Therefore, OSS engineering implements less safety than traditional efforts for proprietary software in our opinion.

5 Related Work

The project discussions of Sect. 3 gave examples of the diversity faced among OSS projects, while previous sections have pointed out common characteristics. These commonalities pose differences to software development in proprietary environments. The overview provided by Boehm has served as an orientation in the area of software development in the proprietary world [17].

This article provides a qualitative analysis of these characteristics as they are observed today, and brings them into context with two examples of proprietary software development. In addition, Sect. 3 provides examples from three major forms of open source development process implementations. It is important to note that we have studied the current state of affairs; while similar analyses have been published a decade ago (e. g., [26; 30]), the open source project landscape is dynamical and evolves constantly as new ideas are brought forward by contributors, which makes it important to periodically re-consider the state of the art.

A large number of publications cover the phenomenon of open source projects as such: They discuss the motivation for stakeholders to contribute to such effort [25; 27; 29; 31]. While common perception understood contributions to open source projects as being without monetary compensation, recent research indicates that a relevant part of contributions appear to be paid work by participating companies [28]. In both cases, contributions by private persons and as part of paid work, a self-organising nature of project governance and inter-operation is required [28]. Understanding this requirement as important influence represents a relevant consideration for understanding the best practices as presented in Sect. 2.

One of the main characteristics of open source projects is transparency on what and how things happen in a project, and how decision are made. These aspects influence the progress of a project, which can be quantitatively measured. Various researchers have performed general analyses (e. g., [25]), or have examined specific aspects of open source development in detail. For example, Kuro and Tian provide an analysis based on how many days it takes to open, cover and resolve (or deny) reported issues, bugs or defects [22]. Another example is Ref. [23], which examines the size distributions of committed source code [23]. Contrary to these approaches this article does not focus on one specific aspect of an open source project, but instead describes the more general characteristics.

6 Conclusion

Open source engineering processes are similar to all traditional engineering processes in some aspects, but exhibit crucial differences in other areas – in particular with respect to communication, and even more so as to how technical issues are handled and resolved. While a considerable focus is placed on which and how technologies are used, little emphasis is placed on legacy issues. Actually implementing code has a much higher priority than discussing possible alternatives or planning ahead. We have illustrated these differences to traditional software engineering by discussing generic best practices of OSS development, by means of several project example discussions, and by comparatively outlining the approaches applied in traditional software engineering.

References

[1] Google patch submission and review guidelines, http://source.android.com/source/life-of-a-patch.html and http://source.android.com/source/submit-patches.html.

[2] Linux Kernel Development process, http://www.kernel.org/doc/Documentation/development-process.

[3] R. N. Charette, *Why Software Fails*, IEEE Spektrum, September 2005.

[4] K. Schwaber, J. Sutherland, *The Scrum Guide*, available at http://www.scrum.org/Portals/0/Documents/Scrum%20Guides/Scrum_Guide.pdf.

[5] J. Robbins, *Adopting Open Source Software Engineering (OSSE) Practices by Adopting OSSE Tools* in Making Sense of the Bazaar: Perspectives on Open Source and Free Software, J. Feller, B. Fitzgerald, S. Hissam and K. Lakham (Eds.) Sebastopol, CA: O'Reilly & Associates. 2003.

[6] F. Schulz von Thun, *Miteinander reden: Störungen und Klärungen. Psychologie der zwischenmenschlichen Kommunikation*. Rowohlt, Reinbek, 1981.

[7] W. Mauerer, *Professional Linux Kernel Architecture*, Wiley & Sons, 2008.

[8] P. Clements et al., *Documenting Software Architectures*, Addison-Wesley Professional, 2010.

[9] The Apache Software Foundation, *How the ASF Works*, available at: http://www.apache.org/foundation/how-it-works.html.

[10] The Apache Software Foundation, *Apache Voting Process*, available at: http://www.apache.org/foundation/voting.html.

[11] The Apache Software Foundation, *A Guide To Release Management During Incubation*, available at: http://incubator.apache.org/guides/releasemanagement.html.

[12] The Apache Software Foundation, *The HTTP Server Proejct*, available at: http://httpd.apache.org/dev/release.html.

[13] *Debian Code of Conduct*, electronically available at: http://www.debian.org/MailingLists/#codeofconduct.

[14] iBAG, Industrieanlagen-Betriebsgesellschaft mbH (Hosting), Development Standard for IT Systems of the Federal Republic of Germany (V-Model 97), available at: http://www.v-modell.iabg.de/, June 1997.

[15] I. Jacobson, G. Booch, and J. Rumbaugh, *The unified software development process.* Addison-Wesley, Reading (MA), USA, 1999.

[16] Ch. Couder, *Fully automated bisecting with git bisect run*, Linux Weekly News, available at: http://lwn.net/Articles/317154/.

[17] B. Boehm, *A view of 20th and 21st century software engineering*, In Proceedings of the 28th international conference on Software engineering. ACM, 2006.

[18] K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change*, Boston, MA: Addison-Wesley, 2005.

[19] R. L. Glass. *The Standish report: does it really describe a software crisis?* Communications of the ACM 49, Issue 8, August 2006.

[20] K. Schwaber and J. Sutherland, *The Definitive Guide to Scrum: The Rules of the Game*, Scrum.org, available at: http://www.scrum.org/Portals/0/Documents/Scrum%20Guides/Scrum_Guide.pdf.

[21] B. Alpern and F. B. Schneider, *Defining Liveness*, Information Processing Letters, Vol. 21, Issue 4, October 1985, Elsevier.

[22] A. G. Koru and J. Tian, *Defect handling in medium and large open source projects*, Software, IEEE Software 21.4 (2004): 54–61.

[23] O. Arafat and D. Riehle, *The comment density of open source software code*, 31st International Conference on Software Engineering – Companion Volume, 2009. ICSE-Companion 2009, May 2009.

[24] The Free Software Manifesto, electronically available at: http://www.gnu.org/philosophy/free-sw.html.

[25] S. Krishnamurthy, *Cave or community?: An empirical examination of 100 mature open source projects*, First Monday, 2002, available at: http://ifipwg213.org/system/files/krishnamurthy.pdf.

[26] G. v. Krogh, S. Spaeth, and K. R. Lakhani, *Community, joining, and specialization in open source software innovation: a case study*, Research Policy, Volume 32, Issue 7, July 2003.

[27] D. Riehle, *The Economic Motivation of Open Source Software: Stakeholder Perspectives*, IEEE Computer, vol. 40, no. 4, April 2007.

[28] D. Riehle, *Best of (Our) Empirical Open Source Research*, invited talk at the 39th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2013), available at: http://www.sofsem.cz/sofsem13/files/presentations/Invited/Riehle.pdf, January 2013.

[29] S. Shah, *Motivation, Governance, and the Viability of Hybrid Forms in Open Source Software Development*, Managemenet Science 52, 7, 1000–1014, July 2006.

[30] A. Mockus, R. T. Fielding, and J. D. Herbsleb, *Two case studies of open source software development: Apache and Mozilla.* ACM Transactions on Software Engineering and Methodology (TOSEM) 11.3 (2002): 309–346.

[31] K. R. Lakhan and R. G. Wolf, *Why Hackers Do What They Do: Understanding Motivation and Effort in Free/Open Source Software Projects* MIT Sloan Working Paper, September 2003.

Received: March 9, 2013

Dr. Wolfgang Mauerer is a software architect for Siemens Corporate Technology, where he works on topics including low-level system architecture, real-time and virtualisation with a focus on open source technologies, and also manages internal research efforts on software development platforms. He holds a PhD in theoretical physics from the Max Planck Institute for the science of light.

Address: Siemens AG, Corporate Technology, Wladimirstraße 3, 91058 Erlangen, Germany
Tel.: +49-9131-721815,
e-mail: wolfgang.mauerer@siemens.com

Dr. Michael C. Jaeger, at Siemens CT, Michael works in different roles as project manager, software architect, trainer and consultant for distributed systems, server applications and their development with an emphasis on open source software. He has more than 10 years of experience in professional software development. Michael received a diploma degree in computer engineering and a Dr.-Ing. degree in informatics, both from TU Berlin.

Address: Siemens AG, Corporate Technology, Otto-Hahn-Ring 6, 80200 München, Germany
Tel.: +49-89-636-48362,
e-mail: michael.c.jaeger@siemens.com