

Why and How Should Open Source Projects Adopt Time-Based Releases?

Martin Michlmayr, Hewlett-Packard

Brian Fitzgerald and Klaas-Jan Stol, Lero—The Irish Software Engineering Research Centre

// Traditional release strategies have associated problems that can be overcome by time-based release management. Findings from interviews with key members of seven prominent volunteer-based open source projects reveal several benefits of adopting a time-based release strategy. //



FREQUENTLY RELEASING software helps improve its quality. Open source evangelist Eric Raymond phrased this succinctly as “Release early. Release often.”¹ However, the software engineering community has devoted little attention to

release engineering.² Although this is changing,³ research on the topic tends to focus on proprietary software development;^{4,5} few studies have focused on open source software.⁶ (For more details, see the sidebar.)

Studying open source projects is useful for several reasons. The software industry depends increasingly on open source and is investing in and sponsoring successful open source projects.^{7–9} So, an understanding of how such volunteer communities work is important because companies must be able to interact with them.⁸ Also, companies can draw lessons from open source communities by studying best practices and applying them internally.^{10–12}

Regarding releases, volunteer-driven open source projects usually employ one of two strategies. Many projects issue a new release after implementing a certain set of features,¹³ which involves numerous challenges. Alternatively, projects might adopt a time-based strategy, in which releases are planned for a specific date.¹⁴ Previous research has suggested that community activity increases close to release dates¹⁵ and that new releases result in spikes of downloads.¹⁶ Such a regular “heartbeat” signals to both users and potential contributors that the project is progressing in a healthy way. For instance, the date of the latest release may impact a potential contributor’s first impression of the project;¹⁷ attracting new developers is important for a project’s sustainability. It also allows vendors and distributors to rely on a consistent stream of new releases and to include the latest features and bug fixes.

Here, we report on an interview study of release management that included seven prominent open source projects that have moved from feature-based to time-based releases. On the basis of our findings, we discuss why projects should adopt time-based releases and how they can adopt such a strategy.

RELATED WORK IN OPEN SOURCE RELEASE MANAGEMENT

Few research studies have focused on release management in open source development. In one of the first studies, Justin Erenkrantz presented a taxonomy of open source release management that helps in understanding and comparing release strategies and applied the taxonomy to compare three projects: the Linux kernel, Subversion, and the Apache HTTP server.¹ (Hyrum Wright observed that all projects have changed their release process since then.²)

Hyrum Wright and Dewayne Perry reported a qualitative case study on release mismanagement in the Subversion project.³ They described how the new version at the time (ver. 1.5) was delayed owing to numerous setbacks, such as high complexity and many bugs. Later, Wright identified several challenges related to release engineering, pertaining to software architecture, release failure's social causes, and challenges related to the software domain and tools.^{2,4}

Foutse Khomh and his colleagues investigated whether faster releases improve software quality.⁵ Their study of Firefox focused on the relationship between release cycle

length and postrelease defects, how quickly bugs were fixed, and the adoption rate by users. Our study, in contrast, focuses on why and how to adopt a time-based release strategy (see the main article).

References

1. J.R. Erenkrantz, "Release Management within Open Source Projects," *Proc. 3rd Workshop Open Source Software Eng.*, 2003, pp. 51–55.
2. H.K. Wright, "Release Engineering Processes, Their Faults and Failures," PhD dissertation, Univ. of Texas at Austin, 2012.
3. H.K. Wright and D.E. Perry, "Subversion 1.5: A Case Study in Open Source Release Mismanagement," *Proc. 2nd FLOSS Workshop*, 2009, pp. 13–18.
4. H.K. Wright and D.E. Perry, "Release Engineering Practices and Pitfalls," *Proc. 2012 Int'l Conf. Software Eng. (ICSE 12)*, 2012, pp. 1281–1284.
5. F. Khomh et al., "Understanding the Impact of Rapid Releases on Software Quality: The Case of Firefox," *Empirical Software Eng.*, May 2014, doi:10.1007/s10664-014-9308-x.

The Study

Our study focused on large, distributed open source projects controlled by volunteers. Although some contributors may receive payments to work on a project, they cannot be controlled by a company. Therefore, we didn't include projects controlled by companies, such as the Ubuntu Linux distribution produced by Canonical. We interviewed 20 key informants, who all were knowledgeable about the respective release process through their roles as a core developer, release manager, quality assurance (QA) team member, project foundation board member, or steering committee member. We analyzed the transcribed interviews for patterns relating to release management, time frames, deadlines, and delays. Although all the studied

projects intended to follow a time-based release schedule, some projects (for example, the Linux kernel) achieved this better than others (for example, Debian).

Table 1 describes these projects and the interviewees. The online supplement to this article (at <http://doi.ieeecomputersociety.org/10.1109/MS.2015.34>) provides additional information on the projects and interview guide.

Why Adopt Time-Based Releases?

Traditional release strategies deliver a new version of the software based on a set of new features or defect fixes. In theory, such feature-based releases can work well because developers have identified the work to do before the next release. Such

feature-based releases could follow a short cycle such as sprint-driven development—Scrum, for example, works this way.

In practice, however, such feature-based strategies can cause a variety of interlinked problems, especially in volunteer-run open source projects. This can result in a long, unpredictable release cycle because certain features may never be finished, and consequently the release may not happen. Thus, feature-based releases are associated with a number of problems.

Unpredictable Release Schedules Cause Rushed Code

When using a feature-based strategy, an open source project might make a release even if not all planned features have been implemented. This can

TABLE 1

Open source projects investigated for this study.

Project	Type of software	When was the time-based strategy introduced?	Intended interval	Approximate size, mid-2014 (LOC)	Interviewees
Debian	Linux distribution	After v3.1, mid-2005	2 yrs.	324 M	Two release managers
GNU Compiler Collection (GCC)	Compiler	2001	6 mo.	6.5 M	Steering-committee member Distributor
GNOME	Desktop environment	Early 2003	6 mo.	8.2 M	Release manager for Java-GNOME Core developer Two release managers
Linux kernel	OS kernel	Mid-2005	2–3 mo.	17 M	Two maintainers Core developer
OpenOffice.org	Office productivity suite	Early 2005	6 mo.	9 M	Quality assurance team member Community manager Distributor
Plone	Content management system	Early 2006	6 mo.	550 K	Plone release manager Archetypes release manager Release manager
X.org	GUI window manager	Late 2005	6 mo.	2.3 M	Release manager Member of X.org Foundation board of directors Distributor / release manager

happen because work on many open source projects is voluntary, so nobody may have worked on these features. One core developer explained,

In an open source environment, the feature-based strategy is just basically impossible unless you want to wait forever ..., which is what happens to a lot of projects. Some haven't had a release in five years. You cannot tell anybody to do anything.

So, a release might be announced suddenly and unexpectedly. This results in “rushing in” code—what one Linux kernel developer called a “thundering herd of patches.” This is because developers want to integrate

their features and bug fixes in the latest release and have no idea when the next release will occur.

Having time as the criterion (rather than features that may or may not be completed) enables a project to plan and establish a predictable schedule. One release manager stated that, by making the schedule public, the project sends a message that

certain things will happen at certain times [and] you take away some of the mystery.

A regular release also helps developers refrain from rushing code contributions. Developers wish to see their contributions included in

a stable release as soon as possible as this improves their reputation and provides the satisfaction of seeing their code used by others. In a time-based strategy, they don't need to rush in code as the timing of the next release is known in advance. One participant explained,

For developers, regular releases are like trains: if you miss one, you know that there will be another one in the not-too-distant future.

No Release Plan, No Adoption Plan

An unpredictable release schedule makes it difficult for adopters to plan. This is a particular concern for those users or corporations that act as consultants or component

integrators because they're unable to decide whether to use the latest stable (but possibly old) version or a more recent (but possibly less stable) development version. Vendors shipping an open source project as part of a distribution or larger product need to know when a release will be stable to ensure stability of the whole distribution or product.

Due to the uncertainty surrounding releases, many vendors avoid official development versions and work on their own versions. This leads to fragmentation between vendors (such as Linux distributions), which diverts significant resources from the official development releases. Not only is this inefficient duplication of

a significant period of time. Some projects, such as Debian and the Linux kernel, had long development phases. Many projects experienced additional delays resulting in numerous changes to be tested at the end of the development phase before a release. Consequently, getting the development tree stable in preparation for a release was difficult. As one QA team member explained,

It's a problem if you start packing too many features into a release. The features are there, but they are simply not stable.

As the development version increasingly diverges from the latest

If you can keep the development more tightly linked to the actual usage, you will get much better feedback in terms of bugs and development direction.

Furthermore, more regular releases improve project members' proficiency in managing releases, which can help optimize the release process. Developers will become more used to the process, and release managers will become familiar with preparing a release, thus making the process more straightforward and standardized. One release manager illustrated this as follows:

With something like cooking, we're just used to doing it every day so you know how you do it and it doesn't fail.

More regular releases improve project members' proficiency in managing releases.

work, it also has a substantial negative impact on these "sponsored" projects that benefit from company contributions.

A time-based strategy offers a predictable release schedule, which is of great benefit to vendors who distribute the software, as one participant explained:

You know the schedule in advance, and you can decide which version to ship. That's a really good point from the distributor's point of view.

Infrequent Releases, Workload Accumulation

An infrequent, unpredictable release cycle causes developers to add features as they see fit, resulting in the accumulation of many changes over

stable release, fewer users are willing to test these releases. Most users don't start testing until a stable release is imminent, which would require an explicit plan or announcement. A long interval between stable releases therefore means that there's little feedback coming from the user community. Owing to the long time spent on development, some projects rushed out releases without sufficient testing or making test releases available for the development versions. This had a negative impact on the quality of the released software.

A time-based strategy ensures a more regular release cycle and creates a much tighter feedback loop with users. A core developer of GNOME commented,

Outdated Software, Outdated Bug Reports

Before the projects adopted time-based releases, stable releases used by end users were often quite old and outdated compared to the latest development version. Some projects, such as the Linux kernel, spend substantial effort to update their latest stable releases, particularly for urgent features such as new hardware support that would have to be backported from a development release to a stable release. While this provides benefits for end users, it limits the resources that could otherwise have been used to prepare a new stable release.

As a project's latest stable release becomes outdated, many bug reports often become of less value as they may already have been fixed in the latest development version. Even if a bug was highly critical, which would warrant a back-port of the fix to the stable release, developers

are unlikely to be motivated to invest time in doing so, given that their work is voluntary.

A time-based strategy implies a more regular stream of new versions and thus a more regular synchronization, not only among developers but also across the user base. This is because users become more used to updates to the latest release. For developers, both those within the project and others who integrate the software, it's important to synchronize regularly so that they work from the same code base to prevent merge conflicts. This also helps in keeping the issue database relevant as fewer bugs relating to older versions will be reported.

Delays Leading to Further Delays

Delays may manifest themselves in a vicious cycle leading to further delays. Once it becomes clear to developers that a release target can't be met or that deadlines aren't enforced, developers may try to push in more changes, thus destabilizing the code base and causing further delays. As intervals between releases increase, developers will realize that their new contributions won't be made available for a long time, causing them to push even harder to get the new code in.

Furthermore, these delays could lead to developer and end-user frustration. Developers who spend significant time and effort to complete their contributions could be left disillusioned if expected releases are constantly delayed, leaving their work unavailable to the general public. Given that many volunteers are driven by such things as fun and a sense of satisfaction, they may choose to spend their time on a project in which they can make an immediate impact. This will further

reduce the most critical resource of an open source project, namely contributing developers.

When a project fails to meet release targets several times, the project as a whole, and its release manager in particular, loses credibility. The realization by developers, end users, and vendors that a project's release schedule is unpredictable could result in dropping adoption and support of the project altogether.

Regular, predictable releases help increase motivation for three main reasons. First, developers who contribute features or bug fixes know their contributions will be available to users within a fairly short time. Second, positive user feedback can reinvigorate developers. Finally, a more organized and predictable process may also prompt potential contributors to engage with the project as a regular release heartbeat is an indication of a project that delivers.

How to Adopt Time-Based Releases

On the basis of insights from our study, we identified the following steps for implementing time-based releases (see Figure 1).

Assess the Project's Suitability

Despite the various benefits, a time-based strategy may not be suitable for every project. Thus, the first step is to decide whether or not a project is suitable for a time-based release strategy.

Project characteristics. A project's characteristics help determine whether a time-based release approach is appropriate. It doesn't make much sense to release regularly if there has been little work done that would warrant a new release. Furthermore, changing the

Step 1. Assess the project's suitability

- Project characteristics
- Developer buy-in

Step 2. Determine the release interval

- Regularity and predictability
- The nature of the project and its users
- Commercial factors
- Cost and effort
- Network effects

Step 3. Implement the release schedule

- Identify dependencies
- Plan the schedule
- Avoid specific periods

FIGURE 1. Guidelines for implementing time-based releases.

release strategy can cause significant disruption. Therefore, projects that don't have problems with release management shouldn't risk changing unless they see clear benefits. The need for change was clear in the GNOME project, as one participant described:

I guess GNOME was close to being a disaster. Everyone agreed that it had to change, and they were looking for a [solution].

Developer buy-in. While a release manager has the formal authority to make decisions regarding releases, it's important that he or she doesn't alienate voluntary contributors by failing to consult with them on major changes, which could lead to them leaving a project. One release manager suggested that

releasing only works if all involved developers just trust [the release managers] to do it. If people think it won't happen anyway, they will just upload broken changes and then it won't happen.

If a project has failed to meet targets for several years, introducing a time-based strategy won't convince developers straightaway that deadlines are now real. To build trust, contributors must gradually gain positive experience with the new process. A successful implementation of time-based releases also relies on introducing appropriate control structures, as one participant illustrated:

Simply declaring that you'd release every six months is not good enough. It's actually a whole set of rules that were implemented at the time that made it go again.

However, establishing such control structures and getting them accepted by developers takes time. Furthermore, without enforcing these control mechanisms, a project might regress into its previous, unstructured process.

Determine the Release Interval

The next step is to choose an appropriate release interval. We identified five factors that affect the choice of interval.

release cycles. As one release manager explained,

When you speak about release cycles of 1.5 years, it's much harder to make a forecast about what will happen, what it will look like.

The nature of the project and its users.

Different users have varying requirements, and the type of users also depends on the nature of the project. For example, the GNU Compiler Collection (GCC) is a development tool whose users are (necessarily) developers. Developers may often be interested in frequent releases that deliver "cutting edge" software. They'll also be able to cope with technical difficulties, whereas less tech-savvy users might prefer more rigorously tested software. Such users might lean toward a slower release cycle that doesn't require them to upgrade too frequently.

The nature of the project may influence the release interval as well. For example, both the Linux kernel and X.org include hardware drivers. Since new hardware appears on the market regularly, projects trying

Commercial factors. Commercial interests could also affect the release interval. For example, many open source projects rely on commercial Linux vendors for wide distribution of their software. Although open source projects are freely accessible through the Internet, being included in a commercial distribution (such as Red Hat) can benefit open source projects. This will make them more widely distributed and help them gain a larger user base—a goal of any open source project.

A member of the OpenOffice.org community gave another example. The OpenOffice.org project is served by community members who write books on the software. A too-short release interval would negatively impact book sales:

It would be the death if you would really bring out a new release every three months with new features. They would not sell any books at all.

The publicity factor is another related, important factor. Projects with a long release cycle may have a "big bang" release that may create a splash of publicity, but more frequent releases could lead to constant press coverage of the project's progress. This is important information for potential users because it suggests an actively maintained project.

Finally, a short release interval might allow open source projects to compete more effectively with proprietary software products—for example, Mozilla Firefox versus Microsoft Internet Explorer—that typically have much slower release cycles. As one participant said,

A fast release schedule gives substantial advantages over a competitor with a long cycle; for

A short release interval might allow open source projects to compete more effectively with proprietary software products.

Regularity and predictability. If the interval is too short, contributors will have insufficient time to integrate their work and test adequately. On the other hand, long release cycles lead to many changes, which could result in some of the problems we mentioned earlier. Furthermore, planning is much more difficult with long

to support new hardware are dealing with a moving target. A project may choose a short release cycle to cope with this continuous need to deliver additional functionality. Or, it may split such functionality (such as hardware drivers) from the main software package. The X.org project adopted the latter solution.

example, during their beta cycle, several new releases of your product will hit the market.

Cost and effort. Several cost factors are important for deciding on a release interval. First, old releases typically must be supported for a certain amount of time. A short release cycle will increase the maintenance burden, as one participant explained:

If you have to maintain lots of older releases, it creates a huge burden.

Furthermore, the preparation of releases may require significant effort in large projects. One member of the OpenOffice.org QA team commented,

There is lots of work associated with a release. I don't want to do a full release test every two months.

Interestingly, this comment was made several months before OpenOffice.org moved to a three-month release interval. After about a year, the project announced its intention to move to a six-month release cycle, for two reasons. First, users expressed a preference for less frequent delivery of new features. Second, the QA team found it difficult to thoroughly test releases within the relatively short cycle of three months.

Too-frequent releases also fragment a project's user base, making it harder to track outstanding issues. Feedback could become less relevant if many users remain with older versions that differ from the current development version. Besides additional effort for developers, end users may be adversely affected by additional effort related to upgrades (installation and configuration, the

learning curve for a new version, and so on).

A final issue is the potential difficulty of making large (or radical)

distributors who were also main contributors to the project stated they had no intention to deploy that particular version.

The QA team found it difficult to thoroughly test releases within the relatively short cycle of three months.

changes to the code base. However, these need not occur in the main development line. Significant features may comprise several releases in a separate branch, which can be merged once they're finished.

Network effects. A project can gain tremendous advantages if it synchronizes its release schedule with the schedules of other projects from which it can leverage benefits. For example, a key reason why the Plone project moved to a six-month time-based strategy was to align its development more closely with that of Zope, as Plone is built on top of the latter. A Plone release manager outlined how the implementation of a similar release strategy enabled the project to use Zope's newest features:

In order to stay up to date, we need to tie ourselves to a specific Zope release, and the way to do that is to have our release schedules synchronized.

Network effects can also be observed in the influence of vendors who contribute to open source projects and vendors who ship the project in distributions (such as Linux distributions). For example, at some point, there was some debate on the GCC mailing list as to whether to skip a certain version, since two

Implement the Schedule

Finally, the third step defines activities to implement the release schedule.

Identifying dependencies. In large projects consisting of many components, much of the work can start only when other tasks have been completed. It's important to identify such dependencies when creating a schedule. Offering general advice on how to identify such dependencies is difficult as it relies on experience with, and the context of, each unique project. Nevertheless, a good starting point is to investigate a project's past problems. Then, institute clear deadlines so that developers working on a specific activity know in advance when a dependent task is due.

Although being overly strict in a volunteer-based project can be difficult, it's important to make clear that deadlines are firm. If a release manager doesn't enforce these deadlines, it's likely that volunteers will increasingly ignore them, leading to further delays. A release manager should aim to reduce dependencies where possible. One way to do this is by identifying fallback options. For example, if the new interface of a library isn't finished by a certain time, it should be possible to go back to the previous version of the library.

Planning the schedule. Developers may work on new features outside the main development branch and integrate the work once it's finished. In fact, some projects, including OpenOffice.org, develop most of their features in such a “branched” way. One participant clarified that

if you insist on doing all development work on [the main development branch], then yes, of course you have a problem because you need a certain amount of time to get anything done.

This suggests that the structure of the development process has an important impact on the schedule. A regular release strategy requires the development tree to be fairly stable over time. Some practices we observed to keep the development tree stable are the use of branches for features that take significant effort, peer review, testing discipline, and reverting and postponing features.

Schedule planning should also take into account activities such as testing and translations (for multilingual projects).

Projects should establish and enforce a clear policy to ensure that milestones are achieved and deadlines are respected. By actively omitting or postponing features that aren't ready, a release manager asserts control and shows developers that deadlines are enforced. One practice to enforce this is the use of freezes (such as code freezes), which set clear deadlines for certain changes.


Avoiding specific periods. When deciding on a release schedule, projects should consider the availability of the voluntary contributors. While proprietary software vendors have defined holidays, many open source

projects operate throughout the year. Although development might take place at any time, there are nevertheless restrictions as to when a release can be prepared. For example, the projects we studied actively avoided the Christmas period, during which many volunteers might not be available. One participant explained,

One shouldn't have the freeze of all packages while people are away on Christmas vacation. We cannot release exactly on Christmas or Easter where important people won't be available.

Most projects in our study also have their own events, such as conferences, during which participants are unavailable, and hence a release at such moments should also be avoided. Other periods are less critical—evidence as to whether to avoid the summer break is much less conclusive. Although many volunteers might be unavailable during that period, the absence of different contributors is spread over several months, so this “balances out.” In fact, some contributors, such as students, might even increase their involvement as they have more time available.

While a time-based release strategy offers several benefits, it's important to realize that a time-based release strategy doesn't necessarily benefit all projects. Our study has a number of limitations that could be addressed by further research. First, we can't draw any conclusions as to whether our findings apply to proprietary software development because different considerations, such as economic and business factors,

affect decisions regarding a release strategy. Second, the increasing involvement of the software industry in open source projects over the last decade will introduce additional constraints and factors that may affect the suitability of a time-based release strategy for those projects, especially if these projects have a significant role in the release process. Finally, we should emphasize that the steps we identified for adopting a time-based strategy don't guarantee success. Adopting a time-based release approach is challenging, which we also found to be true for the projects in our study. 

Acknowledgments

We thank the anonymous reviewers whose comments helped improve this article. This research was supported partly by Science Foundation Ireland under grant 10/CE/I1855 to Lero—The Irish Software Engineering Research Centre (www.lero.ie), Enterprise Ireland under grant IR/2013/0021 to ITEA2-SCALARE, and the Irish Research Council.

References

1. E.S. Raymond, *The Cathedral and the Bazaar*, O'Reilly, 2001.
2. H.K. Wright and D.E. Perry, “Release Engineering Practices and Pitfalls,” *Proc. 34th Int'l Conf. Software Eng.*, 2012, pp. 1281–1284; doi:10.1109/ICSE.2012.6227099.
3. *Proc. 1st Int'l Workshop Release Eng. (RELENG 13)*, B. Adams et al., eds., 2013; doi:10.1109/ICSE.2013.6606779.
4. N. Kerzazi and F. Khomh, “Factors Impacting Software Release Engineering: A Longitudinal Study,” *Proc. 2nd Workshop Release Eng.*, 2014.
5. S. Phillips, G. Ruhe, and J. Sillito, “Information Needs for Integrating Decisions in the Release Process of Large-Scale Parallel Development,” *Proc. 2012 ACM Computer-Supported Collaborative Work (CSCW 12)*, 2012, pp. 1371–1380; doi:10.1145/2145204.2145408.
6. M. Steff, B. Russo, and G. Ruhe, “Evolution of Features and Their Dependencies—an Explorative Study in OSS,” *Proc. 2012 ACM Int'l Symp. Empirical Software Eng. and Measurement*, 2012, pp. 111–114; doi:10.1145/2372251.2372270.



MARTIN MICHLMAYR is an open source community expert at Hewlett-Packard. Previously he was the project leader for the Debian project. His research interests include open source, quality, and release management. Michlmayr received a PhD in technology management from the University of Cambridge. Contact him at tbm@cyrius.com.



BRIAN FITZGERALD is chief scientist at Lero—The Irish Software Engineering Research Centre and holds the Frederick Krehbiel Chair in Innovation in Business and Technology at the University of Limerick. His research interests include open source software, inner source, crowdsourcing, and lean and agile methods. Fitzgerald received a PhD in computer science from the University of London. Contact him at bf@lero.ie.



KLAAS-JAN STOL is a research fellow at Lero—The Irish Software Engineering Research Centre. His research interests include open source software, inner source, and agile and lean methods. Stol received a PhD in software engineering from the University of Limerick. Contact him at klaas-jan.stol@lero.ie.

7. B. Fitzgerald, "The Transformation of Open Source Software," *MIS Q.*, vol. 30, no. 3, 2006, pp. 587–598.
8. J.M. Gonzalez-Barahona et al., "Understanding How Companies Interact with Free Software Communities," *IEEE Software*, vol. 30, no. 5, 2013, pp. 38–45; doi:10.1109/MS.2013.95.
9. J.M. Gonzalez-Barahona and G. Robles, "Trends in Free, Libre, Open Source Software Communities: From Volunteers to Companies," *it - Information Technology*, vol. 55, no. 5, 2013, pp. 173–180; doi:10.1515/itit.2013.1012.
10. K. Stol and B. Fitzgerald, "Inner Source—Adopting Open Source Development Practices within Organizations: A Tutorial," *IEEE Software*, preprint, 2 May 2014; doi:10.1109/MS.2014.77.
11. T. O'Reilly, "Lessons from Open Source Software Development," *Comm. ACM*, vol. 42, no. 4, 1999, pp. 33–37; doi:10.1145/299157.299164.
12. P.C. Rigby et al., "Contemporary Peer Review in Action: Lessons from Open Source Development," *IEEE Software*, vol. 29, no. 6, 2012, pp. 56–61; doi:10.1109/MS.2012.24.
13. K. Fogel, *Producing Open Source Software: How to Run a Successful Free Software Project*, O'Reilly, 2005.
14. M. Michlmayr and B. Fitzgerald, "Time-Based Release Management in Free and Open Source (FOSS) Projects," *Int'l J. Open Source Software and Processes*, vol. 4, no. 1, 2012, pp. 1–19; doi:10.4018/jossp.2012010101.
15. B. Rossi, B. Russo, and G. Succi, "Analysis of Open Source Software Development Iterations by Means of Burst Detection Techniques," *Proc. 2009 Int'l Conf. Open Source Systems*, 2009, pp. 83–93; doi:10.1007/978-3-642-02032-2_9.
16. A. Wiggins, J. Howison, and K. Crowston, "Heartbeat: Measuring Active User Base and Potential User Interest in FLOSS Projects," *Proc. 2009 Int'l Conf. Open Source Systems*, 2009, pp. 94–104; doi:10.1007/978-3-642-02032-2_10.
17. N. Choi, I. Chengular-Smith, and A. Whitmore, "Managing First Impressions of New Open Source Software Projects," *IEEE Software*, vol. 27, no. 6, 2010, pp. 73–77; doi:10.1109/MS.2010.26.

IEEE Annals of the History of Computing

From the analytical engine to the supercomputer, from Pascal to von Neumann—the *IEEE Annals of the History of Computing* covers the breadth of computer history. The quarterly publication is an active center for the collection and dissemination of information on historical projects and organizations, oral history activities, and international conferences.

www.computer.org/annals



Selected CS articles and columns are also available for free at <http://ComputingNow.computer.org>.