

**Monitoring the Health of an Open Source Project:
A Case Study**

by
Jessica T. McKellar

Submitted to the Department of Electrical Engineering and Computer
Science

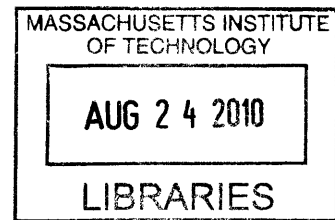
in partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2010

Copyright 2010 Jessica T. McKellar



The author hereby grants to M.I.T. permission to reproduce and
distribute publicly paper and electronic copies of this thesis document
in whole or in part.

ARCHIVES

This work is licensed under the Creative Commons Attribution
License. To view a copy of this license, visit
<http://creativecommons.org/licenses/by/3.0/> or send a letter to
Creative Commons, 559 Nathan Abbott Way, Stanford, CA 94305.

Author
Department of Electrical Engineering and Computer Science
May 21, 2010
Certified by
Eric Grimson
Professor
Thesis Supervisor
Accepted by
Terry Orlando
Chairman, Department Committee on Graduate Theses

Monitoring the Health of an Open Source Project: A Case Study

by

Jessica T. McKellar

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 2010, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis describes MOSHPIT, an open source project health monitoring tool that can be used to assess project health over time. The free and open source project Twisted is used as a case study in analyzing project health using MOSHPIT. This thesis also summarizes important metrics that are often observed informally or irregularly in open source projects but have never been published, and introduces novel metrics cultivated over the course of the author's experiences contributing to Twisted.

Thesis Supervisor: Eric Grimson
Title: Professor

Acknowledgments

Professor Eric Grimson, thank you for supporting me during my unconventional path through MIT and for supervising this thesis. Thank you Adam Fletcher, my best friend, favorite pair programming buddy, proofreader, and constant victim of my debugging questions. Thank you Keith Winstein and Waseem Daher for convincing me to stick with it.

Thank you to the Twisted community for encouraging me as a new contributor, forcing test-driven development on me, and making me a better software engineer and team player because of it, with particular thanks to J.P. Calderone and Glyph Lefkowitz.

Thank you to my family for the trust and freedom you have given me with my academic career. 20 years after sitting on the corner in my 80s-style acid-wash jeans, holding a lunch box and waiting for the bus to round the corner, I look back and would not have changed anything. I love you all.

Contents

1	Introduction	15
1.1	Background	15
1.2	Open source challenges	17
1.3	Introducing MOSHPIT	18
1.4	Introducing Twisted	19
1.4.1	Life cycle of a Twisted ticket	19
2	Metrics	23
2.1	User base	23
2.1.1	Distinct ticket openers	24
2.2	Project activity	26
2.2.1	Release rate	26
2.2.2	Ticket resolution rate	26
2.2.3	Packages for supported platforms	28
2.2.4	Number of open tickets	29
2.2.5	Help channels	30
2.2.6	Conferences and sprints	30
2.3	Durability	31
2.3.1	Distinct ticket closers and distinct ticket reviewers	31
2.3.2	Out-of-review to into-review ratios per contributor	32
2.3.3	Distinct build slave maintainers	35
2.4	Usability	36
2.4.1	Documentation coverage	36

2.5	Reliability	37
2.5.1	Code coverage	37
2.5.2	Open defect tickets	38
2.5.3	Open regressions	38
2.6	Responsiveness	38
2.6.1	Time until first comment	38
2.6.2	Open ticket ages	40
2.6.3	Time between ticket creation and ticket resolution	41
2.6.4	Time spent waiting in review	42
2.7	Fostering growth	42
2.8	Conclusions on key metrics	45
3	MOSH PIT	49
3.1	MOSH PIT input and output	49
3.2	Trac	50
3.3	Other dependencies	50
3.4	Non-sensitive output	51
4	Conclusions and future work	53
A	Trac Schemata	55
A.1	ticket schema	55
A.2	ticket_change schema	56

List of Figures

1-1	Life cycle of a Twisted ticket	22
2-1	Distinct ticket openers over time	25
2-2	Tickets closed over time	27
2-3	Number of open tickets over time	29
2-4	Distinct ticket reviewers over time	34
2-5	Distinct ticket closers over time	35
2-6	Distinct ticket openers, reviewers, and closers over time	36
2-7	Ratios of times taken out of review to times put into review	37
2-8	Time until first comment	40
2-9	Time until resolution	43
2-10	Time spent in review	46
2-11	Long tail contributors: time until first comment	47
2-12	Long tail contributors: time spent in review	48
3-1	MOSHPIIT output	52

List of Tables

2.1	Ticket priority percentages by ticket type for all tickets	30
2.2	Ticket priority percentages by ticket type for open tickets	30
2.3	Number of tickets that took N reviews until resolution as fixed, by ticket type	33
2.4	Average number of reviews before resolution as fixed, by ticket type .	33
2.5	Oldest open tickets by type	41
2.6	Comparisons with long tail contributor statistics, in weeks	44

Chapter 1

Introduction

Open source is in its heyday. Free and open source software has influential advocates [4], modern revision control software options make distributed contribution easy, and open source project aggregator websites make finding projects easy. Free platforms exist to manage and communicate about projects, for example the freenode IRC network [5] or the SourceForge development platform [23].

However, open source project management has not had time to mature into the art that commercial software management has, and academic research on how to effectively structure, maintain, and assess the health of an open source project is limited. This thesis strives to lay a data-driven foundation for open source project management, with a focus on monitoring project health.

1.1 Background

An early exploration of the organization and success of open source projects was the landmark essay *The Cathedral and the Bazaar* by Eric S. Raymond. Written in 1997, Raymond compares the closed development or “cathedral” style of older open source projects like GNU Emacs with a new, open, distributed, “bazaar” style facilitated by the Internet and credited to Linus Torvalds and his management of the Linux project. The essay convinced most existing projects to switch to the bazaar model

that is commonplace today. It also pushed Netscape Communications Corporation to release the source code for Netscape Communicator, which would soon become Mozilla [20].

Large, long-lived software projects like Mozilla and Apache have been analyzed in an effort to understand how open source management principles from these projects could be applied to commercial software management [13], although these projects are not the common case because they are exceptionally large, centrally organized, and well funded.

Attention has been given to analyzing how collaboration works in the extremely distributed environment in which most open source projects are developed [29]. The composition of open source projects and the motivation for contributors has been explored in [10] and [3].

The FLOSSmole [8] project aggregates high-level open source project data from free and open source directory websites like SourceForge and freshmeat [6]. However, project-specific information at the level of individual commits or ticket changes is not available and would require access to svn logs or the ticketing system database. There would be privacy concerns regarding aggregating ticketing system data and hosting it on the web for researchers through a project like FLOSSmole, as it often contains sensitive contributor information like IP addresses and e-mail addresses.

Most of the work exploring open source projects is top-down and analyzes many projects at once or focuses on high-level information about a project's organization. This thesis uses a bottom-up approach. It focuses on a specific project and data generated by contributors as they develop for it, extrapolating useful metrics from observations as a real contributor that can be generalized to and used in other open source projects in future work.

1.2 Open source challenges

One way to summarize the challenges faced by maintainers of and contributors to open source projects is to contrast them with commercial software projects.

For example, most contributors to an open source project are volunteers. An open source project might be sponsored by for-profit companies that use the software and want to promote its continued existence or are giving money in exchange for fast-tracking a particular enhancement request. An open source project might also have funding through a not-for-profit organization dedicated to promoting a particular interest; for example, the Python Software Foundation [19] funds special software projects that are implemented in Python. That said, making money is not typically a goal for an open source project, and instead a large user base and active developer community are the end goals. Meanwhile, employees at companies working on commercial projects are paid, and the company has the goal of profiting from the project.

Most open source projects have a very flat organizational hierarchy that is not well-expressed as a tree. Having power to complete a certain operation is often colloquially termed as having “bits” on that operation. A contributor might have bits to commit to the code repository, to edit the website, to run a build slave, or to expense something on behalf of the project. There are no bits that are analogous to being a manager at a commercial software company, and typically contributors do not have defined roles within the project’s community (an individual is far less likely to be solely an open source project’s “accountant” than a long-term developer who finds bugs and creates and commits patches for them and who has stepped into the role because nobody else is doing it). The implication is that rarely does a contributor to an open source project have the well-defined role of supervisor or health monitor. Conversely, managers exist at all levels of the organizational chart in a traditional commercial software company with the purpose of maintaining a holistic understanding of the health and success of their components and keeping them on

track to meet goals.

1.3 Introducing MOSHPIT

This thesis takes initial steps towards understanding how to maintain an open source software project by focusing on how to monitor a project's health. Health is a broad term related to a number of ideas, including activity, durability, reliability, and responsiveness, and is elaborated upon below. Health statistics can be gathered from a variety of sources, including the ticketing system database, website traffic logs, and repository logs. This thesis summarizes metrics from many sources but focuses its software deliverable, MOSHPIT, exclusively on generating health statistics from the ticketing database.

MOSHPIT (Monitoring Open Source Health and Project Involvement Trends) is a tool that takes a dump of the ticketing system database used by an open source project and generates a comprehensive health report for the project, with statistics and plots grouped by health category. The report is generated as a series of HTML pages linked from a central index, ready to be displayed on a project's website.

This thesis has three primary contributions: 1) A demonstration of the MOSHPIT software on an active, long-lived open source project, 2) a summary of important health metrics for open source projects that are often used informally by project maintainers to assess health but which have not before been classified or aggregated in a publication, and 3) a summary of novel metrics cultivated and observed by the author while she contributed to the open source project Twisted over a nine month period prior to the composition of this thesis.

1.4 Introducing Twisted

Twisted [27] is an event-driven networking engine written in Python. It has been under development since 2001 and currently has just over 175,000 lines of code and documentation kept under version control. The software is maintained for Linux, Windows, and Mac OS X under a range of Python versions (currently 2.4 - 2.6) and has a release cycle of 6-12 months. Twisted receives some non-recurring funding from sponsors who are typically companies that use the software. It uses Trac [26] as its project management and bug/issue tracking system and has a website containing documentation and software downloads. Code and documentation are kept under Subversion (svn) [25] and a Bazaar (bzd) mirror.

Twisted has a strict and well-documented test-driven development policy; all patches are required to include tests for the code added or changed, and the full test suite must pass on a set of Buildbot [2] instances covering the supported operating systems and Python versions before changes are committed to trunk. Tickets must also be reviewed by someone other than the code author before the code can be committed to trunk. The project also has strong coding and documentation standards, which are kept under version control and displayed on Twisted's website along with the testing standards. Trac, svn, bzr, and Buildbot are all themselves free and open source projects. Project contributors interact with each other and provide help to users through Trac's ticketing system, mailing lists, and on IRC.

1.4.1 Life cycle of a Twisted ticket

Twisted's ticket life cycle, while possessing some unique properties like the use of the "review" keyword described below, is fairly representative of the way an issue for an open source project goes from reporting to resolution.

When a user or developer identifies a Twisted issue to be addressed, a Trac ticket is opened with a summary of the issue. A ticket can be one of 4 types: defect, en-

hancement, task, or regression. Anyone can open a ticket. Along with the summary, the reporter might include an attached “short, self-contained correct example” [24] if the issue is a defect or regression, and may have assigned the ticket to a developer who is typically responsible for the affected code.

A ticket can be closed at any point with a resolution of `INVALID`, `WONTFIX`, or `DUPLICATE`, in which case code associated with the ticket is not applied to trunk. The rest of this life cycle description is for a ticket that will eventually be closed as `FIXED` with a patch being merged addressing the ticket issue.

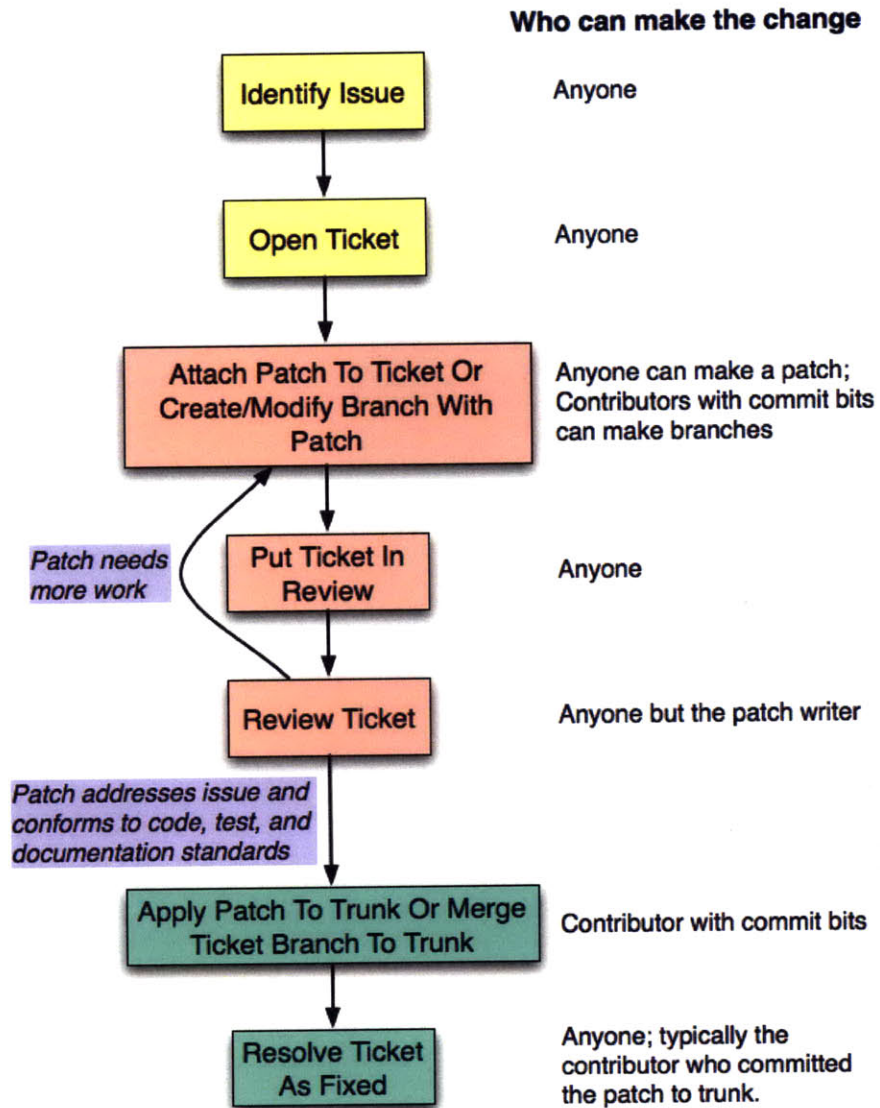
Once a ticket is open, someone submits a patch for the ticket or makes an svn branch containing the proposed patch. Anyone can submit a patch, which is a text file with a unified diff against trunk that is simply attached to the ticket. Only contributors with commit bits can create branches. Frequently, a contributor with commit bits will apply a patch in a branch on behalf of a contributor without commit bits. The person assigned to a ticket is typically the person who created a patch for it.

A patch that contains both a solution for the issue and unit tests for the affected code is submitted for review by the author. A ticket is put into review by adding the “review” keyword to the ticket. Anyone who is not the author of the patch can review a ticket. A reviewer is checking the patch for several properties:

- Does the patch actually address or fix the reported issue?
- Does the patch conform to Twisted’s coding standards? These standards include formatting and naming conventions.
- Does the patch conform to Twisted’s documentation standards? These standards require module-, class-, and function-level document strings with parameter and return value markup.
- Does the patch conform to Twisted’s testing standards?

- Does the full Twisted test suite pass with this patch?

A reviewer puts feedback on the patch in a comment on the Trac ticket and then removes the “review” keyword to indicate that the ticket has been reviewed. If the review feedback indicates that the patch needs more work to meet the properties described above, the author will make changes and resubmit it for review. Otherwise, it is ready to be merged to trunk. Merging can be done only by individuals with commit bits. Once a ticket’s changes have been merged, the ticket is resolved as FIXED. The Twisted life cycle is summarized in Figure 1-1(a).



(a)

Figure 1-1: Life cycle of a Twisted ticket

Chapter 2

Metrics

This chapter summarizes key metrics for measuring and monitoring the health of an open source project in 7 categories: user base, project activity, durability, usability, reliability, responsiveness, and fostering growth. All graphs in this chapter come from a real MOSHPIT report generated against Twisted's Trac database as described in Chapter 3.

2.1 User base

An important class of health metrics is that concerning the user base. An open source project wants to see growth in its user base and sustained use by other projects for a number of reasons:

1. Users can convert to developers. This often starts with a user contributing a ticket and/or patch for a defect that is affecting his or her software and grows into general contributions. Additionally, if a project starts stagnating, companies that rely on the project may step in and contribute because they do not want to have to rewrite their software to not use an unmaintained dependency.
2. Having many users increases the chance of defects being found and reported so they can be fixed.
3. Companies that use the software may provide funding.

Measuring the number of users accurately is challenging. Unlike in a commercial software project that has a subscription service or sells a product, there is no monthly fee or one-time payment to use the software. Using a tool like Google Analytics [7] or Site Meter [22], the number of unique website visitors and unique software downloads can be tracked over time, but that information is not the same as knowing how many projects use your software. An incomplete list of users can be garnered from lists of sponsors and testimonials and by observing who answers questions on mailing lists (individuals asking questions on behalf of a company often have the company in an e-mail signature or volunteer that information while giving background information about themselves).

Another resource to track voluntary acknowledgment of use of a project is through a website like Ohloh, which is a free public directory of open source software projects and their contributors [15]. Individuals with Ohloh accounts can share on their profiles what software projects they use, and that information is aggregated on a project's Ohloh page.

Metrics that can be obtained from the Trac database that give further insight into the growth of a project's user base include the following:

2.1.1 Distinct ticket openers

Ticket openers may be users reporting defects or requesting enhancements, or contributors creating tickets to track tasks or reporting issues discovered while working on existing tickets. While measurements on the number of distinct ticket openers do not distinguish between strict users and contributors, the line is often fuzzy anyway, and the metric is a good general indicator for how many people are using software in a way that is serious enough to merit taking the time to open tickets.

The number of distinct ticket openers can be further broken down by type, into distinct ticket openers of defect, enhancement, task, and regression tickets. See Fig-

ures 2-1(a), 2-1(b), 2-1(c), and 2-1(d) for these MOSHPIT metrics for Twisted.

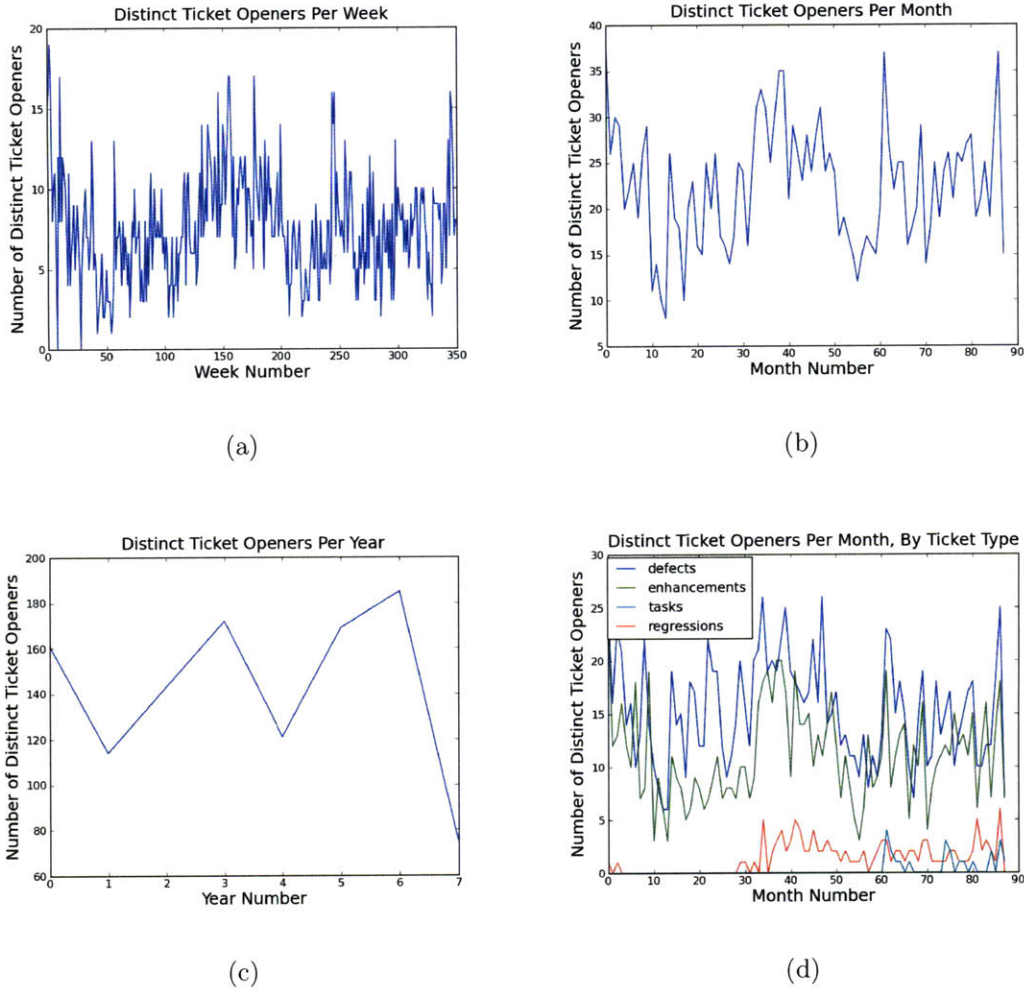


Figure 2-1: Distinct ticket openers over time

Twisted oscillates around 9 distinct ticket openers per week, 23 per month, and 140 per year with a very small positive slope, indicating that the number of distinct ticket openers has changed little over the last 7 years.

Monitoring user base statistics is important because a drop in user base numbers can be indicative of an image problem, not being active enough about promoting the project and an awareness of the project as a solution to some set of problems in the community, or having insufficient help channels. All of these issues are addressable

but can be severe enough to kill off an open source project if a downward trend is persistent and nothing is done.

2.2 Project activity

Activity in this sense means community-visible changes to the project, including commits to the repository, software releases, and activity in help channels. An open source project usually needs steady activity because dependencies change and because inactivity will scare away potential users who do not want to invest their time on unmaintained software. Additionally, the platforms in demand to be supported by your user base change over time. An example of a dependency change is the deprecation and removal of APIs used in your software. Examples of platform changes include Microsoft end-of-lifing Windows XP [28], and Python forking the 2.X release series to create 3.X [18].

Key metrics in the project activity category include the following:

2.2.1 Release rate

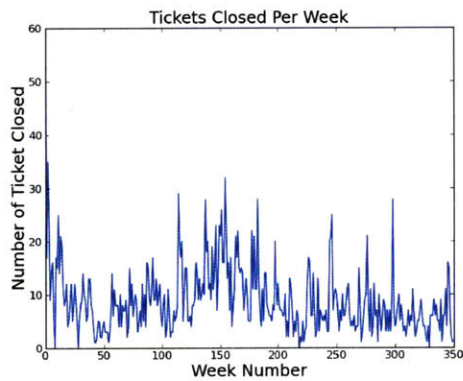
Short, regular release cycles mean that defect fixes, regression fixes, and enhancements get propagated to users quickly. They also suggest liveness and that the project is well-maintained to potential users.

2.2.2 Ticket resolution rate

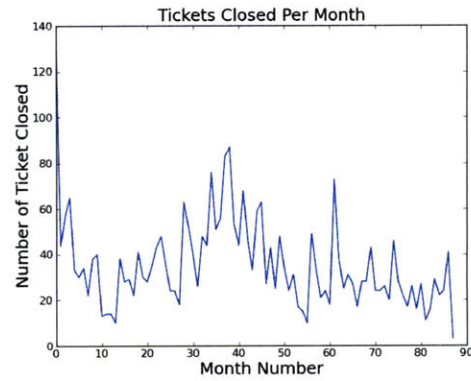
For most projects, there are always open tickets for defects or enhancement requests, and if that is the case regular commits mean the developer community is making regular efforts to improve the quality of the software by committing changes for and then closing these tickets. Tracking particular types of tickets over time provides insight into where resources are focused. Is this project only under maintenance and thus dominated by resolution of defects, or is it in a period of expansion and closing

a lot of enhancement requests?

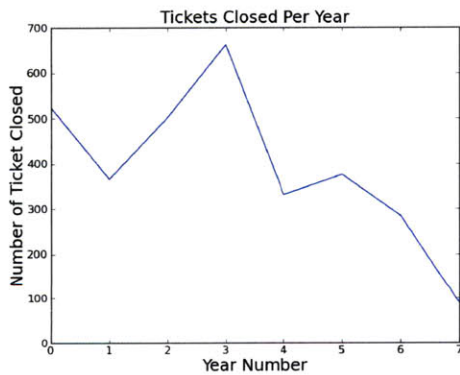
This metric can be broken down into distinct defect, enhancement, task, and regression tickets. See Figures 2-2(a), 2-2(b), 2-2(c), and 2-2(d) for these MOSHPIT metrics for Twisted.



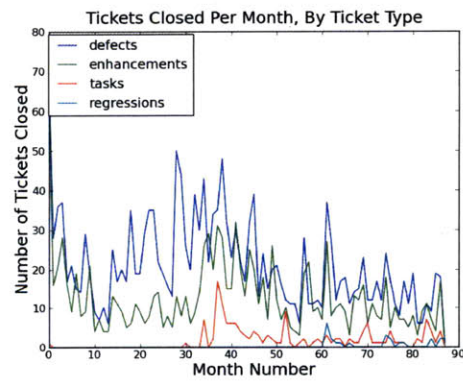
(a)



(b)



(c)



(d)

Figure 2-2: Tickets closed over time

Twisted has seen a gradual decline in ticket closing rates starting around 4 years ago. Figure 2-2(d) indicates that defect fixes dominated ticket closing in the first few years of the project but now enhancements are closed at an almost equal pace. Tasks started getting assigned and then closed 5 years ago, when a focus on a standardized

release process developed. Regressions started appearing with some regularity 2.5 years ago, which could be an indication that recently there has not been a thorough enough review process such that regressions are more likely to develop, that more regressions are getting caught, or simply that the regression ticket type was not used regularly until then. Inspecting tickets in the Trac database that have the regression ticket type or “regression” in comment bodies would clarify the root of this emergence.

2.2.3 Packages for supported platforms

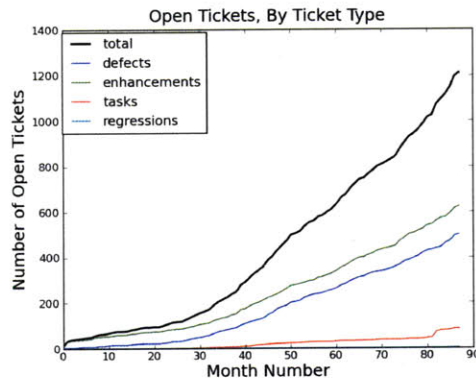
Having access to a source code repository or release tarball on an open source project’s website is reasonable for some users. However, for some Linux users, in particular companies, the only reasonable method of acquisition is through the officially supported software repository for a particular distribution (the analogous situation for OS X or Windows is being a default package in the system). Being in a supported repository means the software package has been vetted by a package maintainer associated with the distribution and that the package maintainer will keep the package up to date, in particular with regard to security updates. (For Ubuntu, this is “main”. Another practical location would be “universe”, which is less restrictive about what open source projects are included in the repository but also does not come with a maintenance guarantee).

Getting an open source project release into the major Linux distributions requires substantial effort. Someone must package the software, and for each target distribution someone with the appropriate bits must be convinced to include the package in a supported repository. Sometimes there just are not people in the community with the technical background to handle packaging, or authority within a distribution to get a project included. Thus, getting into an official repository does not happen for some projects, or the packages lag behind the releases available on the project website.

2.2.4 Number of open tickets

A project might have the number of open tickets trending up, trending down, or staying roughly the same over time. If the trend is down or horizontal, that suggests that there are enough contributors to handle the ticket load generated by users. If the trend is up, the project may need to focus on getting more regular contributors.

See Figures 2-3(a) for Twisted’s open tickets over time, broken down by ticket type.



(a)

Figure 2-3: Number of open tickets over time

Twisted sees a change in slope for defect and enhancement tickets at the 30 month mark, which is interestingly at the same time that task tickets start appearing. The regression line is almost at the x axis, indicating that regressions do not stay open. Looking at the 26 regressions that have been opened in Twisted to date, the most common priority is “highest”, and 96% of all regressions have a priority greater than or equal to “normal”. No “highest” or “high” priority regressions remain open. See Tables 2.1 and 2.2 for priority breakdowns for all ticket types.

Table 2.1: Ticket priority percentages by ticket type for all tickets

	Highest	High	Normal	Low	Lowest	Total opened
defects	17.7	36.8	39.3	4.8	1.4	2351
enhancements	11.6	4.2	63.1	18.6	2.5	1733
tasks	19.6	8.1	66.2	5.1	.9	234
regressions	38.5	23.1	34.6	3.8	0.0	26

Table 2.2: Ticket priority percentages by ticket type for open tickets

	Highest	High	Normal	Low	Lowest	Total opened
defects	2.6	15.0	69.2	11.2	2.0	500
enhancements	2.1	4.2	70.0	21.1	2.9	662
tasks	2.4	6.0	84.5	6.0	.2	84
regressions	0.0	0.0	1.0	0.0	0.0	2

2.2.5 Help channels

Qualitative information about project health can be garnered by monitoring a project’s help channels. Does the mailing list have engaging discussions between users and contributors about how to implement ideas, or does a help request go unanswered? Are developers routinely in active discussion about possible enhancement APIs on IRC? An excellent way to prove to users who use IRC as a help channel that a project is active is to give them a glimpse of project progress without making them do any extra work. For example, Twisted has an IRC bot send a message to the Twisted IRC channel on branch creation, commits, ticket creation or closing, and when a ticket is reviewed.

2.2.6 Conferences and sprints

Another qualitative source of information more pertinent to contributors and potential contributors is project involvement in conferences and “sprints”, which are regional gatherings of contributors to work on a particular set of project issues. For example, Twisted contributors give talks at local Python meetups and at PyCon [16], and sprints have been held all over the world where there is some developer density,

including Spain and Australia. (The project began in Cambridge, MA). Conferences and sprints not only promote visibility to potential users, but to potential developers who get an opportunity to learn the technology alongside other new people and with the assistance of veteran developers.

2.3 Durability

Commercial software projects usually die when they do not have enough customers to turn a profit. Open source projects die when they do not have enough developers to adequately maintain the software. It is unwise for an open source project hoping for longevity to rest all its proverbial eggs in one basket by relying on a small number of regular contributors to do all of the work. People get busy, lose interest, or may change jobs, making the project no longer relevant to their work, and a project should not die due to the disinterest of a single contributor. A large and diverse number of contributors means durability for a project, and maintaining project durability is critical to keeping up the probability of a project's long-term success.

Durability means more than just having many contributors who close defects. It means having many people willing to work on enhancement requests, run build slaves, run a website, maintain documentation, secure funding, answer user questions on help channels, and handle build and release management.

2.3.1 Distinct ticket closers and distinct ticket reviewers

These metrics indicate how many people are putting energy into the system to get tickets resolved. A project wants these numbers to be large and growing as they indicate that the responsibility for seeing tickets through to resolution is distributed amongst the community.

See Figures 2-4(a), 2-4(b), and 2-4(c) for Twisted's distinct ticket reviewers over time and Figure 2-4(d) for a reviewer breakdown by ticket type. Twisted indicates

that a ticket is ready for review by adding the “review” keyword to the ticket. This keyword is removed once the ticket has been reviewed. This keyword is what enables MOSHPIT to track reviews over time, but it was not used until 30 months into the projects, hence the complete inactivity until that time in the graphs.

Patches for tickets often require multiple reviews before being considered ready to commit to trunk and close the ticket. Statistics on the number of reviews until resolution are broken down by ticket type in Tables 2.3 and 2.4. As shown in Table 2.3, the number of reviews a Twisted ticket has gone through before resolution ranges from 1 to 14, although task and regression tickets have never required more than 5 reviews, and only 1 enhancement took 14 reviews. Regardless of ticket type, it is most common for a ticket to have 1 review. As shown in Table 2.4, enhancement tickets have a higher review average than other ticket types, intuitively because enhancements often require substantial new code and the other ticket types frequently do not.

See Figures 2-5(a), 2-5(b), and 2-5(c) for Twisted’s distinct ticket closers over time and Figure 2-5(d) for a closer breakdown by type.

Note that with Twisted one needs commit bits to commit to trunk before closing a ticket. Thus, if the primary patch contributor for a ticket is someone without commit bits, a regular contributor who does have bits is doing the committing. Because getting commit bits is a barrier to closing tickets, unlike for opening or reviewing tickets, the number of distinct closers is substantially smaller. Distinct ticket openers, reviewers, and closers are compared in Figure 2-6(a).

2.3.2 Out-of-review to into-review ratios per contributor

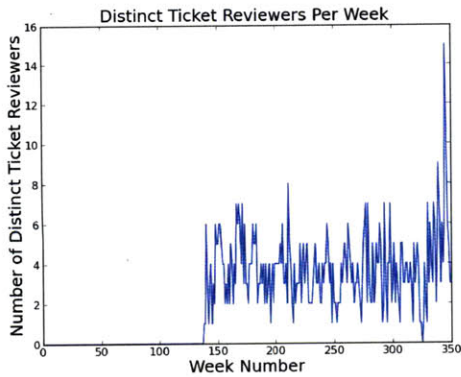
Writing up patches for defects is a necessary first step towards removing defects from software, but someone has to review those patches before they can be committed to trunk. The out-of-review to into-review ratio is an indicator of which contributors

Table 2.3: Number of tickets that took N reviews until resolution as fixed, by ticket type

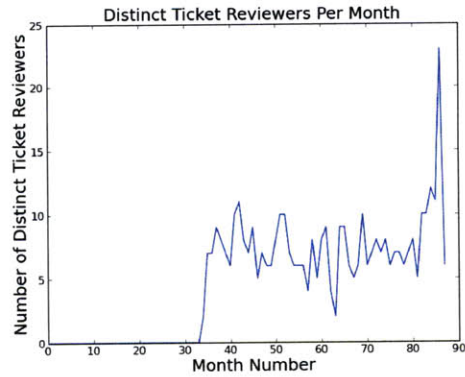
Number of reviews	Defect	Enhancement	Task	Regression
14	0	1	0	0
13	1	1	0	0
12	1	0	0	0
11	1	1	0	0
10	1	2	0	0
9	2	3	0	0
8	1	4	0	0
7	4	7	0	0
6	8	4	0	0
5	14	15	1	1
4	37	28	6	0
3	63	51	8	1
2	139	113	15	9
1	358	172	48	10

Table 2.4: Average number of reviews before resolution as fixed, by ticket type

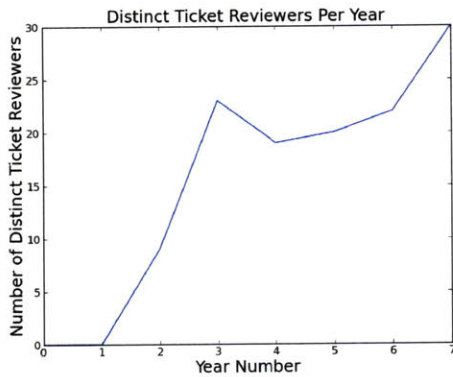
Defect	Enhancement	Task	Regression
1.9	2.3	1.7	1.7



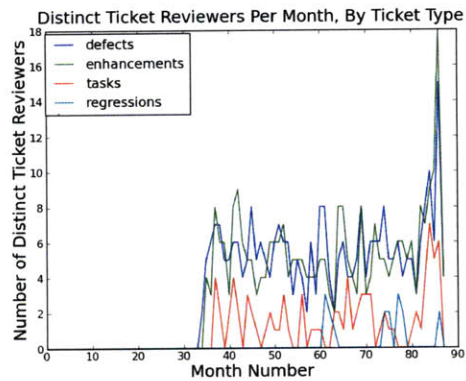
(a)



(b)



(c)

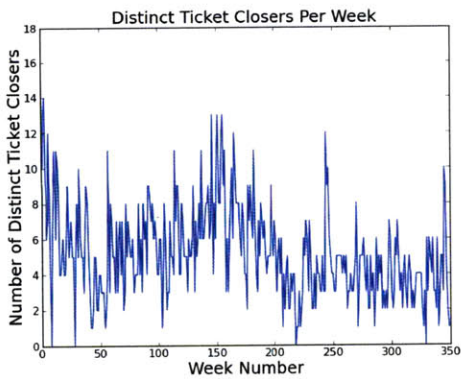


(d)

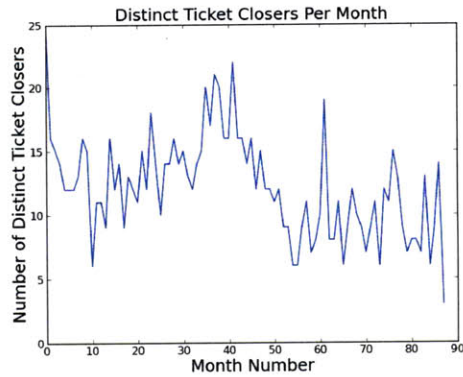
Figure 2-4: Distinct ticket reviewers over time

take review energy out of the system and which contributors put energy in. A contributor who only ever puts tickets into review is an energy sink and has a ratio less than 1, while a contributor who reviews more tickets than he or she puts into review is an energy source and has a ratio greater than 1. In an ideal world, every contributor would put in as much energy as he or she takes out and have a ratio of 1.

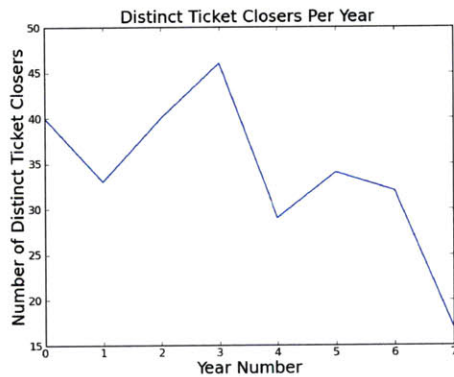
As seen in Figures 2-7(a) and 2-7(b), Twisted is rather far away from this ideal situation. Figure 2-7(a) shows out-of-review to into-review ratios for the contributors with the greatest numbers of review changes. The greatest contributor, with over 2000 review changes, is at 1.06, but no one else in the top 10 has a ratio above 1.



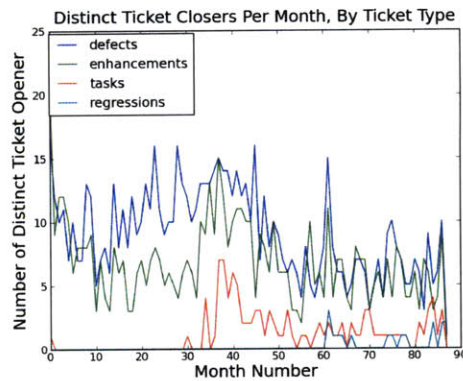
(a)



(b)



(c)



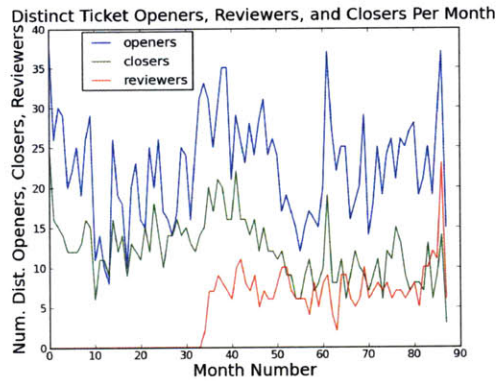
(d)

Figure 2-5: Distinct ticket closers over time

Figure 2-7(b) orders the contributors by ratio, and from this plot we see that a) only 10 contributors have ratios at or above 1, and b) most of those contributors are not regular contributors at all.

2.3.3 Distinct build slave maintainers

Distributing build slave responsibility reduces the likelihood that a developer becoming inactive hamstrings a project's ability to test its code. It also increases the likelihood that the servers are geographically diverse and thus not all susceptible to the same network problems while people are trying to run tests.



(a)

Figure 2-6: Distinct ticket openers, reviewers, and closers over time

Similar durability metrics can be garnered informally for the number of people able to administer the website or handle finances for the project.

2.4 Usability

An open source project is not doing much good if it has no users, and users are only going to use software if it has good usability properties. These include qualitative observations like “are the tutorials helpful” and “is the website easy to navigate”. Usability can also be monitored more formally.

2.4.1 Documentation coverage

What percentage of classes and functions have API documentation? Twisted’s end users are developers themselves, so API documentation is the primary way users will understand how to use the system besides the tutorials. Forcing users to source-dive does not encourage them to use your software. Twisted ensures that documentation coverage always goes up by enforcing its coding standard: all new code must be accompanied by module-, class-, and function-level documentation, including Javadoc-like API markup. Additionally, existing code being modified as part of a patch that did not previously have documentation strings is generally required to get documentation

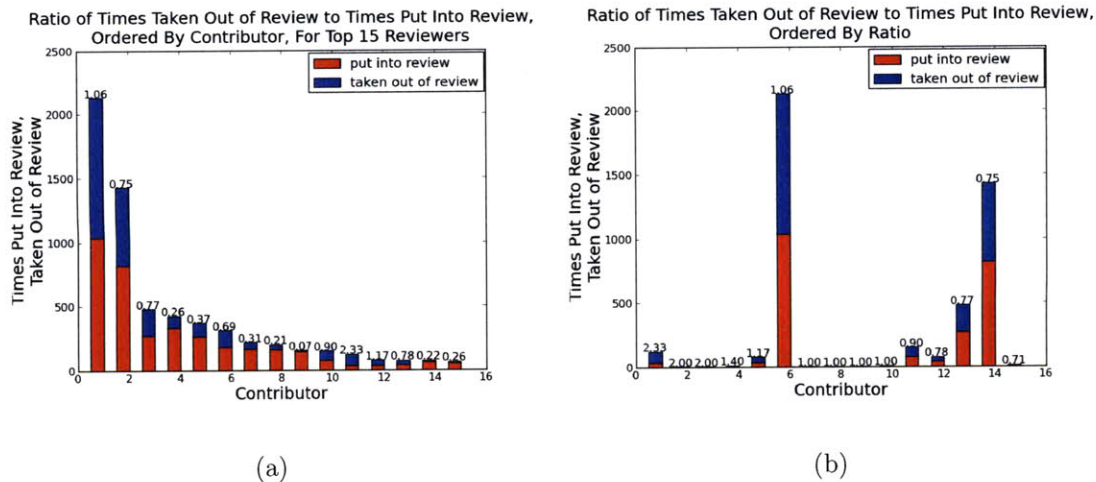


Figure 2-7: Ratios of times taken out of review to times put into review

as part of that patch.

2.5 Reliability

Projects aspire to be featureful and defect-free; they want to grow without sacrificing reliability. Efforts put into reducing known defects by fixing them, reducing the likelihood of future defects by having unit tests, and metrics on how often defects and regressions are discovered are all related to reliability.

2.5.1 Code coverage

Code coverage tools exist in most popular programming languages. They will run a test suite and generate statistics on what percentage of a project is exercised by unit tests. A sign of project health is a high percentage of code coverage, or a percentage that continues to go up over time. Twisted ensures that code coverage always goes up by enforcing its test-driven development policy: all new code must be accompanied by unit tests. Additionally, existing code being modified as part of a patch that did not previously have unit tests is generally required to get unit tests as part of that patch.

2.5.2 Open defect tickets

Does the project have many known defects that remain unresolved? As seen in Figure 2-3(a) and Tables 2.1 and 2.2, defects make up over half of the tickets created for Twisted, and 500 are still open, including a number of “high” and “highest” priority tickets.

2.5.3 Open regressions

Does the project encounter many regressions? This might be a sign that more rigorous auditing needs to be done during code reviews. Regressions are more of a problem for users wedded to packaged versions of a project, who might have to work around regressions until the next release where they are hopefully fixed, than users willing to pull fixes from the source code repository.

As seen in Figure 2-3(a) and Tables 2.1 and 2.2, regressions have been quite rare over Twisted’s history, making up just .6% of tickets. Only 2 are still open, both with a priority of “normal”.

2.6 Responsiveness

Responsiveness in the ticketing system is important both to users and developers. Users who report defects or enhancement requests feel ignored when tickets go untouched for weeks, discouraging them from continuing to submit bug reports and patches that would benefit the project. Developers are similarly disincentivized from contributing if their branches are never reviewed or tickets to which they have applied energy stall on the path to resolution.

2.6.1 Time until first comment

The first response to a ticket might be the contribution of a unit test, confirmation that a defect exists, review feedback, API suggestions, or any number of other com-

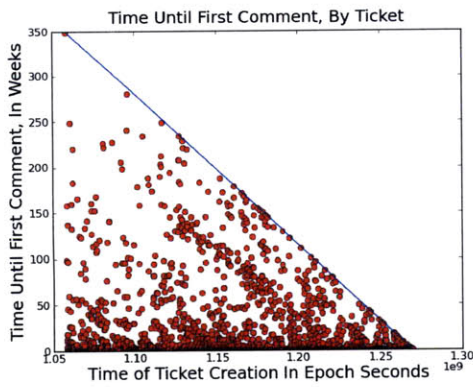
ments. Regardless of the content, the first response is an ACK from the project that lets the user or developer know that he or she is being heard.

See Figures 2-8(a), 2-8(b), and 2-8(c) for MOSHPIT figures for Twisted's time until first comment.

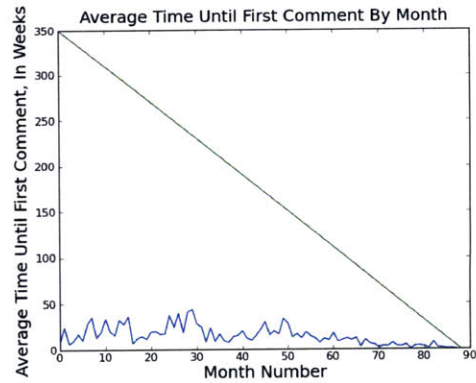
The nearly perfectly outlined triangle in Figure 2-8(a) may seem implausible at first but is explained by the line creating the hypotenuse. This line was drawn between points (time of first ticket creation, time between first ticket creation and present time) and (present time, 0). The time between ticket creation and first comment is bounded by the time between ticket creation and present time; thus, as you travel to the right along the x-axis, the possible time until first comment shrinks to 0.

Note the decreasing plot density as y increases in Figure 2-8(a). Points near the x-axis received comments almost immediately upon ticket creation, while points with high y values, near the hypotenuse, did not receive first comments until nearly present time. The greater the density decrease as y increases, the faster tickets are receiving comments.

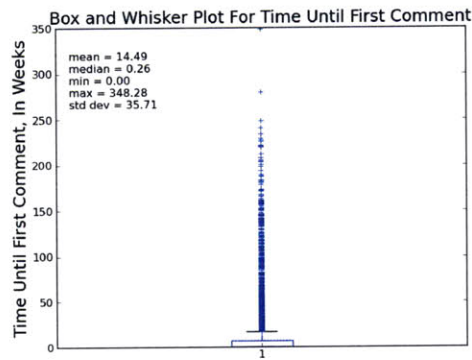
Figure 2-8(b) gives another view into how far below the maximum time most tickets have received comments. It plots the average time until first comment and the same hypotenuse from Figure 2-8(a); because the average line is well under the hypotenuse, most tickets are receiving comments well under the maximum time. However, the box and whisker plot in Figure 2-8(c) shows a long tail beyond the upper quartile; time until first comment has a median of only .26 weeks but a mean of 14.49 weeks dragged upwards by this long tail.



(a)



(b)



(c)

Figure 2-8: Time until first comment

2.6.2 Open ticket ages

Having ancient tickets open suggests that:

1. There are some thorny tickets that are valid but no developers are willing to attempt to close because they are not a high enough priority.
2. The ticketing system is not routinely flushed of tickets that should really be INVALID, WONTFIX, or DUPLICATE instead of remaining open.
3. There are not enough developers to handle the ticket load, and old tickets go by the wayside. The number of open tickets over time (Section 2.2.4) also informs this problem.

Table 2.5: Oldest open tickets by type

	Ticket number	Ticket creation date
Defect	78	July 15th, 2003
Enhancement	50	July 12th, 2003
Task	278	September 24th, 2003
Regression	3998	September 3rd, 2009

See Table 2.5 for Twisted’s oldest open tickets by type. The oldest open ticket, an enhancement, is almost 7 years old; note that Twisted has existed for roughly 10 years. The oldest regression is only 6 months old.

Old tickets make a good sprint focus. A sprint goal addressing ticket age could be to close every ticket below some threshold number.

2.6.3 Time between ticket creation and ticket resolution

Tickets can languish at many steps along the path to resolution, including initial patch submission, the addition of unit tests, review, and responding to review feedback. Time between ticket creation and ticket resolution summarizes the ticket life cycle.

See Figures 2-9(a), 2-9(b), and 2-9(c) for MOSHPIT figures for Twisted’s times between ticket creation and resolution.

Perhaps even more implausible than the near-perfect triangle in Figure 2-8(a) is the near-perfect trapezoid in 2-9(a). The top line is the same as that forming the hypotenuse in 2-8(a). Determining the equation for the bottom line required investigating the tickets with creation times before the first point on the x axis as well as a little Twisted history. As it turns out, at one point Twisted imported its tickets into Trac from Launchpad [11]. Trac inherited all of the Launchpad ticket properties, including creation time, ticket status, and resolution, but not the individual changes

that resulted in a ticket becoming closed. Thus, the only tickets that can get a point in Figure 2-9(a) by becoming resolved are those created after that import, and the lower line bounding the trapezoid is the line between (time of first ticket creation, time of first ticket creation after the import - first ticket creation) and (time of first ticket creation after the import, 0).

As with the time until first comment metrics in Section 2.6.1, Figure 2-9(b) shows how far below the maximum time most tickets have been resolved. This curve is much closer to the hypotenuse than the first comment line, implying that a lot of the time elapsing before resolution is after the first comment has happened. The box and whisker plot in Figure 2-9(c) again shows a long tail beyond the upper quartile, with a median of 6.15 weeks and a mean of a whopping 41.53 weeks.

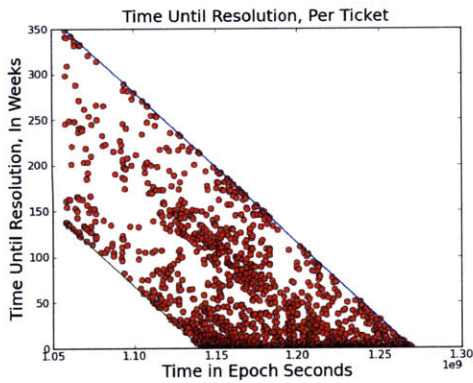
2.6.4 Time spent waiting in review

For projects with more developers taking energy out of the system than putting energy in by submitting tickets for review but never reviewing other peoples' tickets, review can be a major roadblock for a ticket getting resolved.

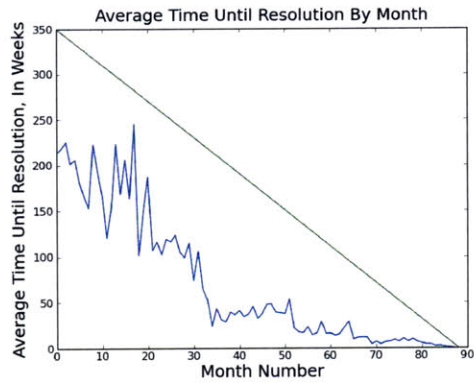
Figures 2-10(a), 2-10(b), and 2-10(c) show positive review statistics for Twisted. With a median time in review of just 1.40 days and a mean time of 5.84, this part of the ticket life cycle is much shorter than the average time until first comment discussed in Section 2.6.1.

2.7 Fostering growth

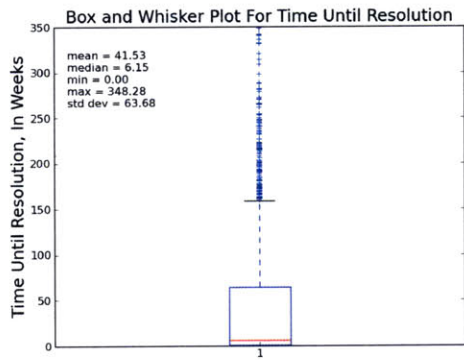
A particular subset of the developer population for a project deserves special attention related to responsiveness: these are the irregular contributors, those that are in the long tail outside the 95% of ticket changes caused by the most active contributors. Because they represent an opportunity to convert to regular contributors, it may make sense for a project to make particular efforts to shorten the ticket cycle for



(a)



(b)



(c)

Figure 2-9: Time until resolution

them with prompt commentary on open tickets and rapid review cycles. Inattentiveness may drive them to a project with better responsiveness.

How the long tail is handled is summarized by adopting the time until first comment metrics from Section 2.6.1 and the time waiting in review metrics from Section 2.6.4. These metrics only include data points from contributors with the smallest number of ticket changes whose changes together make up 5% of all changes. See Figures 2-11(a), 2-11(b), and 2-11(c) for Twisted’s time until first comment statistics for its long tail contributors, and Figures 2-12(a), 2-12(b), and 2-12(c) for Twisted’s time spent in review statistics for its long tail contributors.

Table 2.6: Comparisons with long tail contributor statistics, in weeks

	Time until first comment		Time spent in review	
	Mean	Median	Mean	Median
Long tail	11.00	.09	8.02	1.54
Main	15.23	.32	5.81	1.40
All	14.49	.26	5.84	1.40

Twisted has had 755 distinct contributors since it started using a ticketing system. 95% of ticket change are concentrated in the top 194 contributors, leaving 561 contributors, or 74.3% of all Twisted contributors, in the long tail who are represented in these figures.

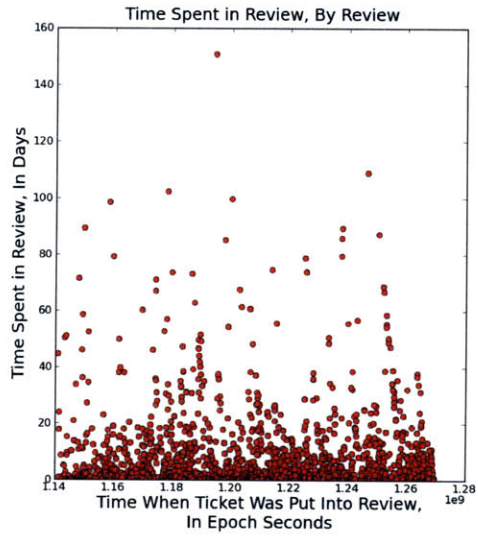
Interestingly, time until first comment statistics are better for the long tail, but statistics on the time spent in review are worse. These statistics are summarized and compared against “main” (not long-tail) contributors and all contributors in Table 2.6. Long tail contributors wait a median .09 weeks (15 hours) for a first comment while main contributors wait .32 weeks (over 2 days). Tickets wait a median 1.54 weeks in review for long tail contributors, one day more than the median of 1.40 weeks for main contributors.

The longer review times could be because new contributors are less familiar with the review policies and have thus require more feedback which takes more time to generate. Another reason could be that people are more hesitant to review work from contributors who do not have a demonstrated history of high-quality patches, because that places more of a burden for catching correctness issues and evaluating design decision choices on the reviewer.

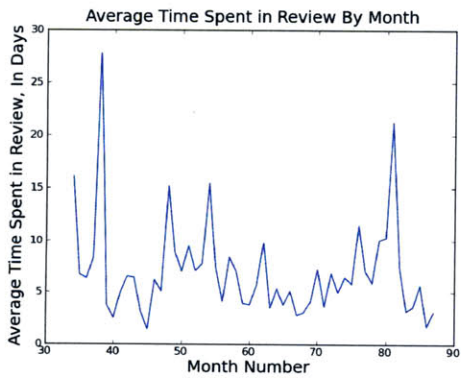
2.8 Conclusions on key metrics

An analysis of the metrics presented above for an open source project provides a good overview of the project's health over time. These metrics can be used to pinpoint problem areas in the ticket life cycle, allocate human resources more effectively to address the problems, and then to monitor improvements in the metrics based on the changes made.

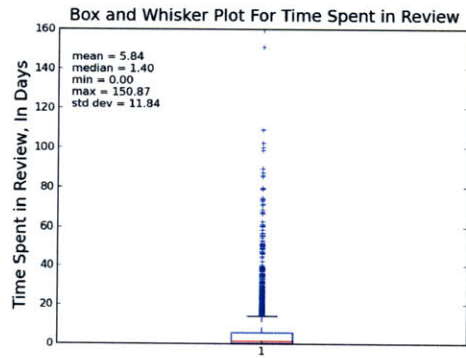
If particular types of tickets are problem areas for a project, the metrics above can be customized for these areas. An example would be documentation tickets. These can be tagged with the “documentation” keyword so that statistics on distinct reviewers (Section 2.3.1) and time until first comment (Section 2.6.1) for documentation tickets can be compared against the all-ticket averages. If the statistics are in fact worse for documentation tickets and documentation is a project priority, resources can be re-allocated to focus on them.



(a)

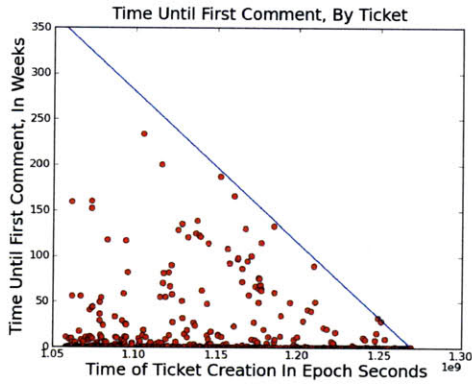


(b)

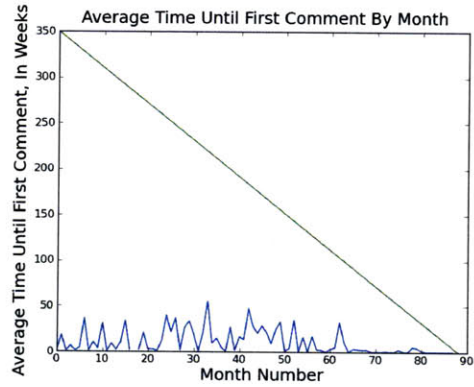


(c)

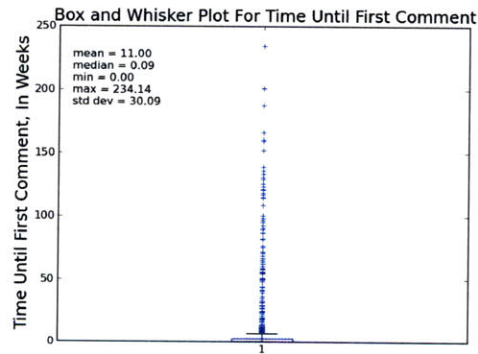
Figure 2-10: Time spent in review



(a)

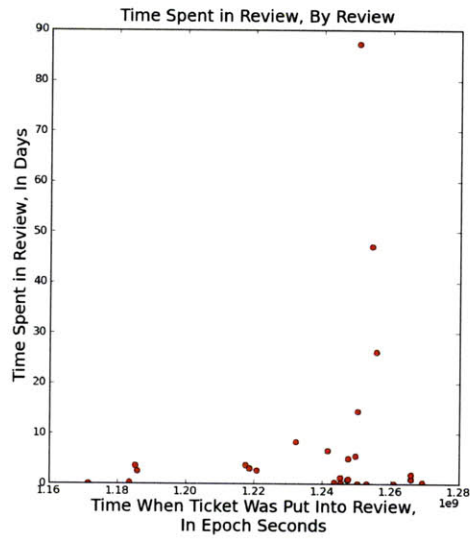


(b)

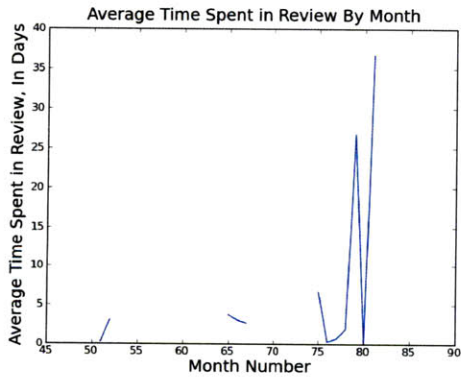


(c)

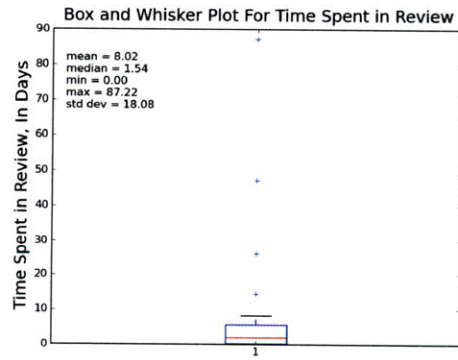
Figure 2-11: Long tail contributors: time until first comment



(a)



(b)



(c)

Figure 2-12: Long tail contributors: time spent in review

Chapter 3

MOSHPIPIT

MOSHPIPIT is a health report generator written in Python. All figures in this thesis came from a MOSHPIT report generated from the Twisted Trac database.

3.1 MOSHPIT input and output

MOSHPIPIT takes as input the name of a Postgres database backup file for a Trac instance. A backup can be generated with the `pg_dump` command, which writes to a file all of the database operations necessary to reconstruct a database. The database is restored from this backup file, and then a series of scripts run database queries and plot the results. MOSHPIT queries a restored backup file instead of a live Trac database so MOSHPIT queries do not degrade database performance for a project's Trac users.

After generating the content of the health report, MOSHPIT assembles the report itself by creating a directory and populating it with an HTML file for each of the 7 metric classes described in Chapter 2: user base, project activity, durability, usability, reliability, responsiveness, and fostering growth. Each file has the graphs, statistics, and annotations analyzed for Twisted in Chapter 2 for its metric class. These class files are linked from an `index.html` file and ready to be served as web content. See Figures 3-1(a) and 3-1(b) for an example index and responsiveness page.

3.2 Trac

MOSHYPIT expects particular schemata which are compatible with Trac versions 0.7 through its latest release, 0.11.7 at the time of this writing.

Only two tables in the Trac database are used to produce the health report: `ticket` and `ticket_change`. `ticket` has a one-to-one relationship between tickets and ticket properties and summarizes the current state of the ticket, including reporter, time of creation, status, and resolution. `ticket_change` has a one-to-many relationship between tickets and changes and records all modifications made to the ticket over time, including ownership changes, comments, and keyword changes. See Appendix A for the full schemata.

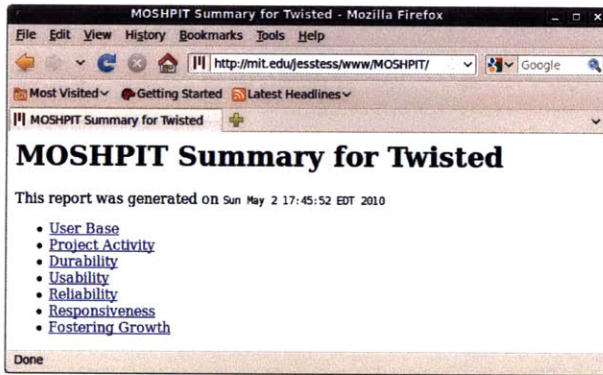
While MOSHPIT currently expects a set of field names for these tables that is particular to Trac, Bugzilla [1], RT [21], and JIRA [9], three other popular ticket and bug tracking systems, have analogous `ticket` and `ticket_change` tables with the fields used in MOSHPIT's SQL queries. Thus, dumps from any of these other systems could be trivially converted to the expected namespace so MOSHPIT could generate health reports from them.

3.3 Other dependencies

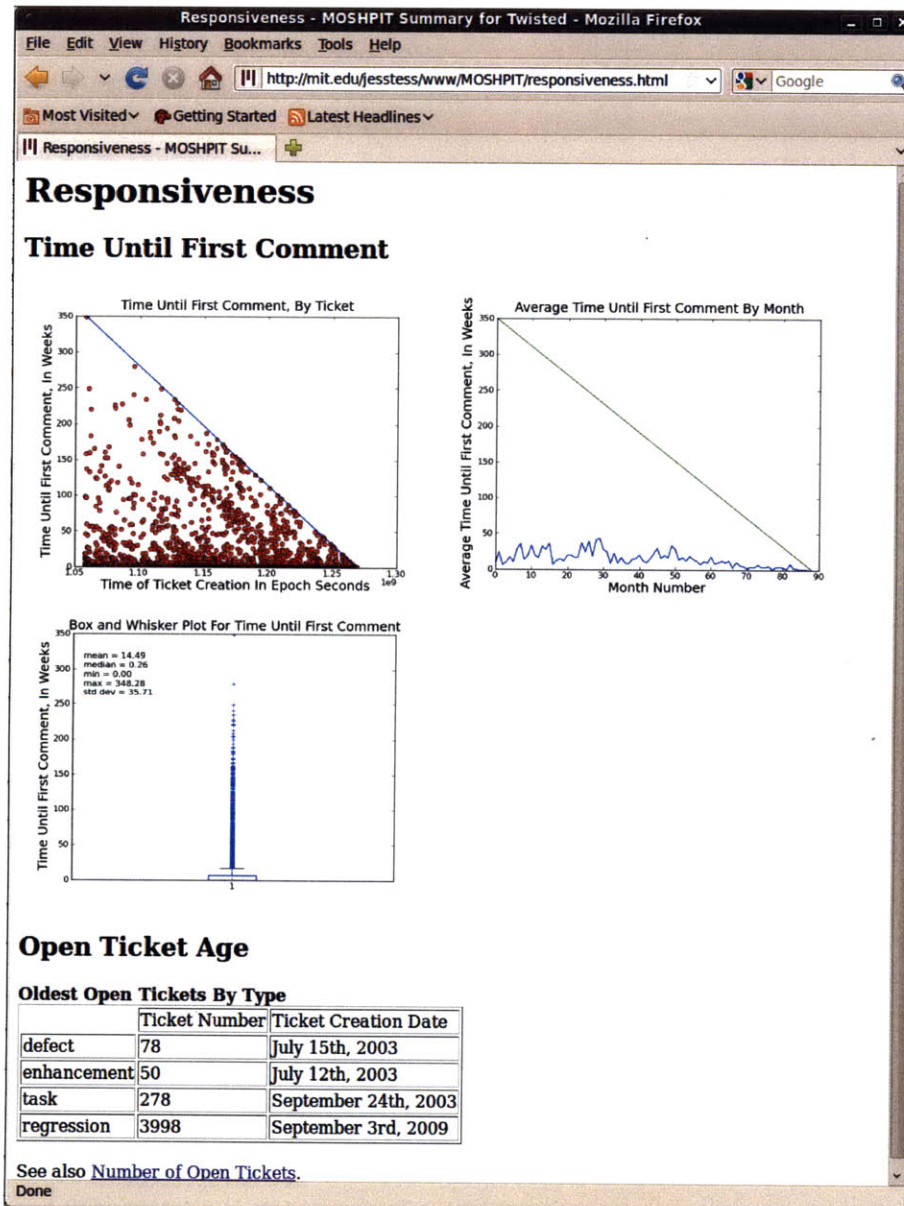
MOSHYPIT is written in Python and is compatible with Python version greater than 2.4 in the 2.X branch. It uses the DB-API 2-compliant `pyPgSQL` PostgreSQL Python bindings [17] to query the database reconstructed from a `pg_dump`. Graphs are generated using the `matplotlib` Python plotting library [12], and some of the statistics are generated using the `NumPy` scientific computing package [14].

3.4 Non-sensitive output

While it would be possible to produce contributor-specific statistics with attribution from the Trac schemata, for example rankings for who closes the most tickets or whose ticket contributions lead to regressions, the focus of MOSHPIT is not individual contributors but rather a holistic view of the community. An advantage of this focus is that no personal information is revealed in a health report, so the report can be placed on a project's website without causing discomfort for contributors.



(a)



(b)

Figure 3-1: MOSHPIT output.

Chapter 4

Conclusions and future work

This thesis summarizes key health metrics in the categories of user base, project activity, durability, usability, reliability, responsiveness, and fostering growth. Real-world applicability of these metrics is demonstrated by using the MOSHPIT health report generated on an actual open source project, Twisted, and discussing the results.

The goal of MOSHPIT is to provide project maintainers with enough health information to be able to identify problem areas in a project's workflow, select metrics to target for improvement, and watch how community changes or procedural changes impact that metric. Thus, a follow-up to this project would be to identify a problem area in Twisted using MOSHPIT, select metrics related to that problem to optimize, initiate changes that are believed to affect the metrics, and monitor the metrics over a period of 6 to 12 months. An example problem area for Twisted is cultivating new contributors. Metrics to optimize while addressing this problem could include time until first comment and time spent waiting for review as described in Sections 2.6.1 and 2.6.4.

The key take-home message for project maintainers is that the more information you log in your ticketing system, the better you can track health and problem areas over time. For example, tracking out-of-review to into-review ratios per contributor (Section 2.3.2) or time spent waiting on review (Section 2.6.4) would not be possi-

ble if the “review” keyword were not a part of the Twisted workflow. If Windows tickets or documentation tickets are ongoing problem areas for a project, they should have “windows” and “documentation” keywords applied consistently so the number of people taking on those tickets and where they spend languishing in the ticket life cycle can be measured and improved.

MOSHPIPIT is a first step in describing and demonstrating how open source health can be measured. Open source projects have the ability to produce much more than just ticketing data for analysis, however. Website traffic, the version control system, and financial records are other rich sources of user and contributor information that help paint a picture of project health and could be analyzed with similar rigor.

Future goals for MOSHPIT include:

- Hosting the source code online so others can use it and contribute to it.
- Generalizing the software so it can create useful reports for projects with workflows that are substantially different from that of Twisted.
- Writing converters that will take database dumps from RT, Jira, and other popular ticketing system and convert relevant tables to schema compatible with MOSHPIT.
- running MOSHPIT against a variety of project databases that have different workflows and analyzing the results to help assess which kinds of workflows produce and maintain healthy projects.

Appendix A

Trac Schemata

A.1 ticket schema

Table "public.ticket"

Column	Type	Modifiers
id	integer	not null default nextval('ticket_id_seq'::regclass)
type	text	
time	integer	
changetime	integer	
component	text	
severity	text	
priority	text	
owner	text	
reporter	text	
cc	text	
version	text	
milestone	text	
status	text	
resolution	text	
summary	text	

```
description | text    |
keywords    | text    |
```

Indexes:

```
"ticket_pkey" PRIMARY KEY, btree (id)
"ticket_status_idx" btree (status)
"ticket_time_idx" btree ("time")
```

A.2 ticket_change schema

Table "public.ticket_change"

Column	Type	Modifiers
ticket	integer	not null
time	integer	not null
author	text	
field	text	not null
oldvalue	text	
newvalue	text	

Indexes:

```
"ticket_change_pk" PRIMARY KEY, btree (ticket, "time", field)
"ticket_change_ticket_idx" btree (ticket)
"ticket_change_time_idx" btree ("time")
```

Bibliography

- [1] Bugzilla. <http://www.bugzilla.org/>.
- [2] Buildbot. <http://buildbot.net/>.
- [3] Scott Christley and Greg Madey. Analysis of activity in the open source software development community. In *The 40th Hawaii International Conference on Systems Science*, Hawaii, January 2007.
- [4] Free software foundation. <http://www.fsf.org/>.
- [5] freenode. <http://freenode.net/>.
- [6] freshmeat.net. <http://freshmeat.net/>.
- [7] Google analytics. <http://www.google.com/analytics/>.
- [8] J. Howison, M. Conklin, and K. Crowston. Flossmole: A collaborative repository for floss research data and analyses. *International Journal of Information Technology and Web Engineering*, 1(3), 2006.
- [9] Bug, issue and project tracking for software development - jira. <http://www.atlassian.com/software/jira/>.
- [10] Sandeep Krishnamurthy. Cave or community?: An empirical examination of 100 mature open source projects. *SSRN eLibrary*.
- [11] Launchpad. <http://launchpad.net/>.
- [12] matplotlib: python plotting. <http://matplotlib.sourceforge.net/>.
- [13] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11(3):309–346, 2002.
- [14] Scientific computing tools for python: Numpy. <http://numpy.scipy.org/>.
- [15] Ohloh, the open source network. <http://www.ohloh.net/>.
- [16] Pycon: Connecting the python community. <http://www.pycon.org/>.
- [17] pypgsql. <http://pypgsql.sourceforge.net/>.

- [18] Python 3.0 release. <http://python.org/download/releases/3.0/>.
- [19] Python software foundation. <http://www.python.org/psf/>.
- [20] Eric S. Raymond. *The Cathedral and the Bazaar*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1999.
- [21] Rt: Request tracker. <http://bestpractical.com/rt/>.
- [22] Sitemeter - counter and statistics tracker. <http://sitemeter.com/>.
- [23] Sourceforge.net: Find and develop open source software. <http://sourceforge.net/>.
- [24] The short self-contained correct example. <http://sscce.org/>.
- [25] Apache subversion. <http://subversion.apache.org/>.
- [26] The trac project. <http://trac.edgewall.org/>.
- [27] Twisted. <http://twistedmatrix.com/>.
- [28] Windows life-cycle policy. <http://www.microsoft.com/windows/lifecycle/default.mspx>.
- [29] Yutaka Yamauchi, Makoto Yokozawa, Takeshi Shinohara, and Toru Ishida. Collaboration with lean media: how open-source software succeeds. In *CSCW '00: Proceedings of the 2000 ACM conference on Computer supported cooperative work*, pages 329–338, New York, NY, USA, 2000. ACM.