Measuring the Maintainability of Open-Source Software

Liguo Yu
Computer Science and Informatics
Indiana University South Bend
South Bend, IN, USA
ligyu@iusb.edu

Stephen R. Schach, Kai Chen
EECS, Vanderbilt University
Nashville, TN, USA
srs@vuse.vanderbilt.edu
kai.chen@vanderbilt.edu

Abstract

An editorial in Empirical Software Engineering *suggested that open-source software projects offer a great deal of data that can be used for experimentation. These data include artifacts such as source code and defect reports. In this paper we show that sources of open-source maintenance data, such as defect-tracking systems, change logs, and source code, cannot, in general, be used for measuring maintainability. We further show that approaches such as using defect distributions and the average lag time to fix a defect can be equally unusable. We conclude that, despite the plethora of open-source maintenance data, it is extremely hard to find data for determining the maintainability of open-source software.*

1. Introduction

The explosion in open-source software projects (some 140,000 in just two sites [1], [2]) has been a boon for empirical software engineers. A wide variety of software artifacts are freely available to any researcher who cares to download and analyze them. Most useful of all, source code of all kinds can be obtained.

The unrestricted availability of open-source software has radically changed the practice of experimental software engineering. An editorial in *Empirical Software Engineering* stated [3]:

> "As empirical software engineers, we should embrace this development [open-source software]. Suddenly one of the greatest obstacles in the way of empirical software engineering has been cleared! Not only is source code available, but also defect reports, update logs, etc. For a change, we can now focus on the analysis rather than the data collection."

Notwithstanding this enthusiastic paean, we recently showed [4] that update logs have omission percentages ranging from 3.7 to 78.6 percent. We concluded that, before using change logs as a basis for research into the development and maintenance of open-source software, experimenters should carefully check for omissions and inaccuracies.

The majority of open-source data we have encountered is maintenance data, because it relates to releases other than the first release of a given open-source product. In this paper, we show that, despite the cornucopia of open-source software maintenance data, it is extremely hard to measure maintainability using open-source maintenance data.

For the sake of clarity, in this paper we use the standard 1990 IEEE terminology, as reaffirmed in 2002: When a programmer makes a *mistake*, the consequence of that mistake is a *fault* in the code; executing the software product then results in a *failure*, that is, the observed incorrect behavior of the product as a consequence of the fault; and an *error* is the amount by which a result is incorrect [5], [6]. In addition, we use the word *defect* as a generic term that refers to a fault, failure, or error.

In Section 2 of this paper we define maintainability. In Section 3 we discuss the unreliability of open-source maintenance data. Other approaches to measuring maintainability are discussed in Section 4. In Section 5, we discuss the general issues in measuring the maintainability of open-source projects. Our conclusions appear in Section 6.

2. Maintainability

Maintainability may be defined as the ease with which maintenance can be performed. The three main maintenance categories are corrective maintenance, perfective maintenance, and adaptive maintenance [7]. *Corrective maintenance* is performed to correct any

0-7803-9508-5/05/$20.00 (c)2005 IEEE

residual faults in the analysis, design, implementation, or documentation, or any other type of fault. A measure of corrective maintenance effort is the number of faults. That is, a way to measure *corrective maintainability* indirectly is to measure fault proneness.

Perfective maintenance is performed to improve product effectiveness by adding additional functionality, making the product run faster, or improving maintainability. According to Boehm et al. [8], we can use modifiability to represent *perfective maintainability*. For example, to improve the performance of an existing module, we must modify that module. A measure of this modifiability is the time spent on changing the original module.

Adaptive maintenance is performed in response to changes in the environment in which the product operates, such as when the product is ported to a new compiler, operating system, and/or hardware. Again according to Boehm et al. [8], we can use understandability to represent *adaptive maintainability*. For example, to adapt a module to a new operating system, we must understand this module first. A measure of this understandability is the time spent on understanding the design and the functionality of the module.

In this paper, we consider all three types of maintainability: corrective, perfective, and adaptive.

3. Direct sources of open-source maintenance data

Three resources that could possibly provide the maintenance data needed for measuring maintainability are described below.

3.1. Defect-tracking systems

Open-source software development is loosely managed. Contributions are purely voluntary and the quality of the source code is not guaranteed. In the software industry, management frequently enforces the keeping of fault records, but defect tracking is selectively performed within open-source projects.

There are a number of open-source defect-tracking systems, of which Bugzilla [9] is the most successful. Bugzilla has been used in numerous open-source projects, including Mozilla [10], Apache [11], and Gnome [12]. For Linux, defects have been recorded using Kernel Bug Tracker [13] and Gentoo Linux Bugzilla [14].

We have encountered three limitations in the process of obtaining useful maintenance data from defect-tracking systems. They are:

(1) The information provided by defect-tracking system may not be complete. For example, we have studied various sets of Linux defect-tracking records and have found that the information available for analysis is incomplete, primarily because the first open-source defect-tracking product was released about 10 years after the first version of Linux, and no-one has gone back and recorded all the earlier defects using a defect-tracking system.

(2) The process used by Bugzilla and all other defect-tracking tools is called "forward tracking" because it provides the information from the time that a defect is reported until it is resolved. However, such tools do not provide any information regarding the origination of a fault. For example, suppose that a fault was introduced in version V10, detected for the first time in version V20, and corrected in versions V23 and V25. From the viewpoint of studying corrective maintenance, we need to know that the fault originated in version V10; depending on the research project, we may or may not be concerned as to when the fault was detected or when it was corrected. Unfortunately, no defect-tracking tools record the version in which the fault originated; Bugzilla data would associate this fault with versions V20, V23, and V25, but there would be no mention whatsoever of version V10. This is not surprising; it is rare for any organization (let alone an open-source development group) to track backwards through version after version to try to determine precisely where a fault was first introduced. This lack of backward tracking information precludes the use of defect-tracking data for studying corrective maintenance of open-source software.

(3) Neither Bugzilla nor any other defect-tracking tool we have encountered provides time or effort data for maintenance. Accordingly, it is not possible to obtain data such as the time spent on understanding a previous version of a module or the effort spent in making changes. That is, we cannot determine from defect-tracking tools how hard it is to perform adaptive or perfective maintenance.

3.2. Change logs

Another resource that could provide maintainability data is a change log, that is, a description of changes that have been made in a given version, together with an explanation of why they were made. Not all open-source projects maintain change logs, and some have only partial change logs; for example, Linux change-log data are available only after version 2.4.

Change-log information concerns all three types of maintenance. As with defect-tracking tools, the infor-

mation provided in change logs cannot be tracked back to the version containing the original fault and can therefore not be used to extract corrective maintainability data. None of the change logs we have seen provide time or effort information either, so they cannot be used for measuring ease in adaptive or perfective maintenance. A severe problem with using change logs is that, as a consequence of the loose management of open-source projects, the data provided in change logs are usually incomplete [4]. Omissions and inaccuracies in change-log data prevent their use for validation purposes. In other words, not only do change logs not contain the data we need but, even if they did, omissions and inaccuracies would bar their use as sources of empirical data.

3.3. Source code

In principle, information regarding the version in which each fault was introduced can be determined by examining the source code itself. That is, we can determine what changes have been made from one version to the next, and hence can determine precisely when a fault first entered the source code. However, two reasons make this method impractical:

(1) Determining precisely when a fault was introduced is not always as simple as finding when one specific statement first appears in the code. All too frequently, the cause of a fault is an interaction between a number of different statements in a variety of modules. Determining precisely when such a fault is introduced is then a deductive process. It is hard to determine when a fault has been fixed (as evidenced by the numerous Bugzilla defect reports that have been reopened after an apparently successful fault repair). It is equally hard to determine when a fault first appears in the code. Accordingly, data regarding the claimed original source of a fault must be treated with circumspection.

A complicating factor is the size of some of the data sets with which we are dealing. For example, Linux now has over 500 released versions, and the size of the latest version is over 4,000,000 lines of code. Because of the deductive nature of the process of determining the origin of a fault, no CASE tool can assist us in this task. Accordingly, we cannot use source code to obtain corrective maintainability data on Linux.

(2) Source code does not provide time and effort information for any kind of maintenance. Therefore, we cannot use it to extract perfective or adaptive maintainability data.

In summary, at present, we know of no complete and accurate data available for measuring the maintainability of an open-source product. This holds for all three maintenance types: corrective, perfective, and adaptive.

In the future, more powerful CASE tools may appear. However, it is extremely unlikely that, in the foreseeable future, backtracking of faults could be automated or that time and effort information could be made available.

4. Other approaches to measuring maintainability

In this section, we describe other approaches to measuring maintainability.

4.1. Using the average lag time to fix a defect

The time between reporting a defect and fixing that defect is called the *lag time* to fix that defect. Using the data stored in a defect-tracking system, it is easy to compute the average lag time over any time period. An increase in average lag time indicates a decrease in maintainability, and vice versa (but see below).

Unfortunately, there are several reasons why the average lag time may not be meaningful for open-source software products:

(1) The lag time is calculated as the time that has elapsed from the instant at which a defect is reported until the instant at which the defect report is closed. Unfortunately, this lag time is not necessarily the time to diagnose and fix the defect. A defect may begin to be handled one day or ten weeks after it was assigned to someone. We have no way of knowing when that person began working on it. Similarly, we have no way of telling how long after the defect was fixed the person working on it reported that the defect report was closed.

(2) In closed-source software development, the number of maintainers generally stays relatively constant. In open-source software development, where participation is purely voluntary, the size of the work force continually changes with personal interest and schedule. An increase in average lag time may simply mean that, during that time interval, the number of individuals choosing to work on the product has decreased.

(3) In closed-source software development, the employees work roughly the same number of hours per week. In open-source software development, where participation is purely voluntary, each participant decides how many hours to work each week. The lag time largely depends on the schedule of the individuals involved, and may not reflect the maintainability of the

product. We would expect an increase in lag time during a holiday period and during summer. Thus, an increase in average lag time may simply mean that, during that time interval, participants were spending more time with their families or on the beach.

(4) When new participants volunteer to join an open-source product, they are usually assigned a maintenance task. It frequently happens that a new participant does not have the skills needed to perform the task. It is embarrassing to have to admit this for everyone to read (all e-mail relating to a defect is kept, permanently, in the relevant defect report of the defect-tracking software, with unrestricted worldwide access), so the new participant simply does nothing. Open-source software development is loosely managed, so what usually happens is that, after a few months, someone realizes that the maintenance task has not been done, and does it himself or herself. The lag time for that task includes the many months that the task lay dormant. Accordingly, an increase in average lag time may simply mean an increase in the number of new volunteers during that time period. (In other words, adding new personnel slows down the project, in accordance with Brooks's Law [7].)

(5) Many organizations use open-source software for mission-critical software. For example, an online ticket agency may use an open-source database. If there is a serious failure in the database software, all the programmers working for the ticket agency will be told to work on detecting and correcting the fault until the database once again comes online. That is, the ticket agency will ensure that the lag time for this fault is as short as possible. The resulting decrease in average lag time will be caused by the sudden increase in the number of individuals working on the open-source project (many of whom may not even be participants in the project, but rather ticket-agency employees told to work on the fault until it is repaired.)

In summary, we believe that average lag time data for open-source software projects is unlikely to be a meaningful way to represent maintainability, and cannot be used for empirical research into maintainability.

4.2 Using defect distributions

Another empirical approach is to utilize defect distributions computed from data obtained from open-source defect-tracking systems. Such an experiment would be performed as follows:

We can count the number of defects reported for each version. As the number of lines of code increases, we would expect the number of defects to increase [7]. The number of defects reported for each version, adjusted for lines of code, could represent the maintainability of that version.

However, our investigation of the Tomcat and Apache Ant open-source projects shows that this approach is as undesirable as using the average lag times. Figures 1 and 2 show the reporting rate of new defects in Tomcat 4.01 through 4.04. The data are summarized in Table 1.

For Tomcat 4.01, 354 defects were reported, and the next release (version 4.02) occurred 18 weeks after the release of version 4.01. For Tomcat 4.02, 116 defects were reported; the next release occurred only 4 weeks after the release of version 4.02. For Tomcat 4.03, 236 defects were reported, and the next release occurred during week 16. For Tomcat 4.04, only 165 defects were reported, and the next release occurred during week 15.

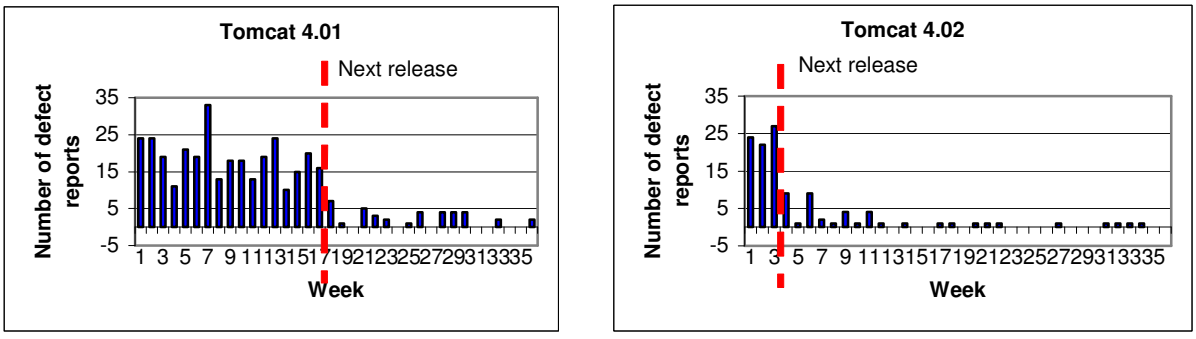

Figure 1. Reporting of new defects in Tomcat 4.01 and 4.0.2

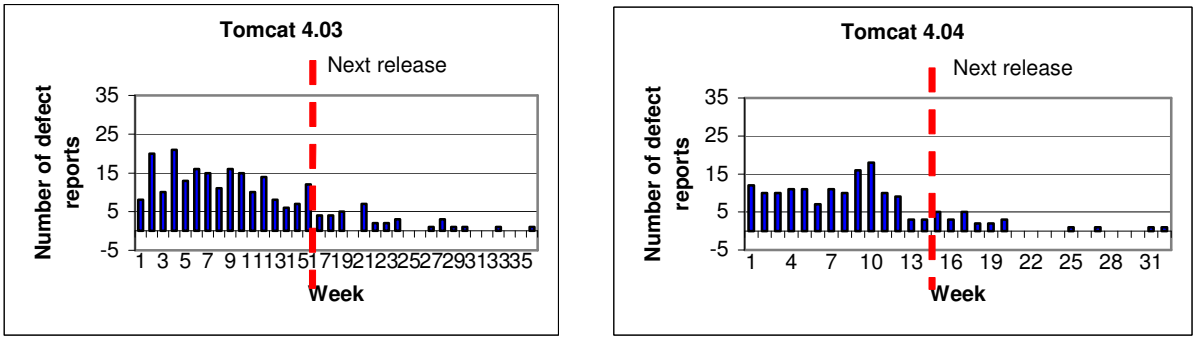

Figure 2. Reporting of new defects in Tomcat 4.03 and 4.0.4

Table 1. Defect reporting data for Tomcat 4.0

Release number	Number of defects reported	Date of next release
4.01	354	Week 18
4.02	116	Week 4
4.03	236	Week 16
4.04	165	Week 15

The graphs in Figures 1 and 2 show that most defects are reported before the release of the next version. After a new release, fewer people pay attention to the previous version, so fewer defects will be reported after that. The data show that the number of defects reported for each version depends not only on the number of defects that are present in that version, but is also affected by the release time of the next version. Closed-source products tend to be released at roughly regular intervals. However, it can be seen from the figures that, for an open-source product, the release time of a new version keeps changing, and this has an effect on the number of defects reported for each version. Similar results for Apache Ant are shown in Figures 3 and 4.

A wide variety of different defect distributions are found in open-source projects, as shown for Tomcat 4.01 through 4.04 (Figures 1 and 2) and for Apache Ant 1.3 through 1.5 (Figures 3 and 4). The pattern could have one "hump" (increase, then decrease), such as for Tomcat 4.0.3. It could also be a pattern with several humps, such as for Ant 1.4.1. The number of defect reports may reach its highest value during the first week after the release, and then begin to decrease, such as for Ant 1.5. On the other hand, the number of defects reported each week may be roughly constant, such as for Ant 1.3. It could also follow other patterns. Due to the large differences among defect distributions, it is not possible to adjust the number of defects for the release time. For example, we cannot say that Tomcat 4.0.1 had more defects than Tomcat 4.0.2, because we do not know how many defects would have been found in Tomcat 4.0.2 if Tomcat 4.0.3 had been released during week 17 instead of week 4.

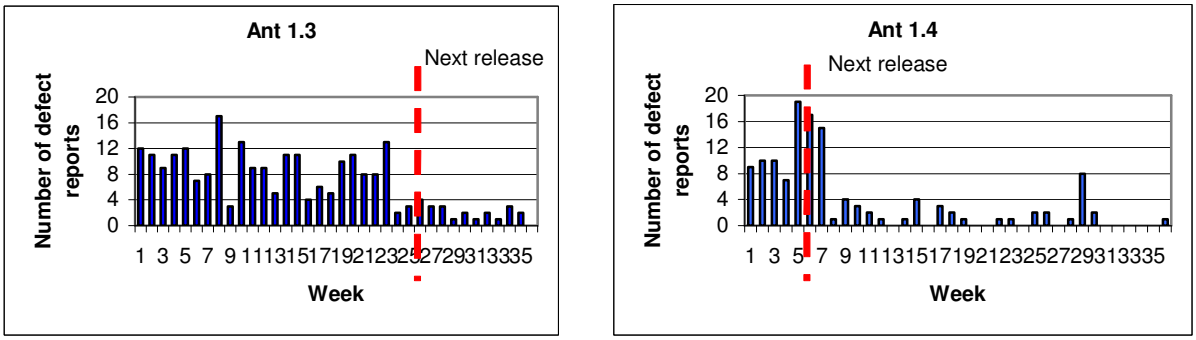

Figure 3. Reporting of new defects in Apache Ant 1.3 and 1.4

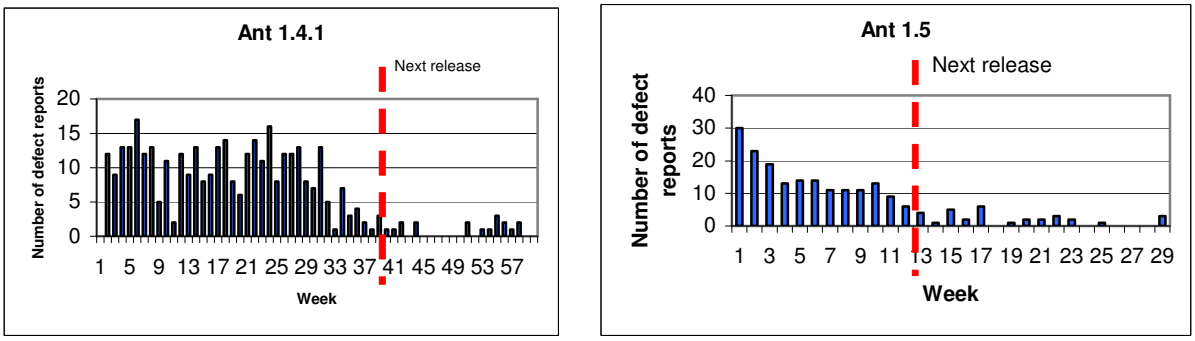

Figure 4. Reporting of new defects in Apache Ant 1.4.1 and 1.5

Table 2. Defect reporting data for Apache Ant

Release number	Number of defects reported	Date of next release
1.3	240	Week 26
1.4	126	Week 6
1.4.1	341	Week 40
1.5	310	Week 13

Even when the time to the next release is nearly the same, the number of defects reported can vary drastically. Referring to Table 1, we see about 17 weeks elapsed before Tomcat versions 4.01, 4.03, and 4.04 were each superseded by a new version. However, the number of defects reported was 354, 236, and 165, respectively.

Similar drastic differences are reflected in Table 2, which summarizes defect-reporting data for Apache Ant. A total of 341 defects were reported during the 40 weeks between the release of Ant 1.4.1 and Ant 1.5, and 310 in the only 13 weeks between the release of Ant 1.5 and Ant 1.6.

The reason for these huge differences in both Tomcat and Apache Ant is that the number of defects reported is largely due to the nature of the maintenance activity carried out prior to each release. If major changes have been made since the last release, we would expect a large number of defects to be reported, and conversely.

Another reason is the nature of open-source defect reporting. The use of a new version and the reporting of defects are completely voluntary. For example, suppose that version 2.0 of a product is a relatively stable version with few fatal defects, and that many users are familiar with version 2.0. Accordingly, when version 2.1 is released, only a few users may choose to update, and it would be unlikely that many defects would be reported for version 2.1. Suppose further that version 2.2, with many attractive features, is released soon after version 2.1. Users would soon switch to version 2.2, version 2.1 would be largely ignored, and most of the defects in that version would not be reported.

In conclusion, we believe that open-source defect distribution data, like average lag time data, cannot be used for research on maintainability.

5. Validity issues

A fundamental question that should be considered early in the design of an experiment is the validity of the experiment. Of the four classes of validity criteria, namely, conclusion validity, construct validity, internal validity, and external validity [15], we need to pay particular attention to construct validity before designing the experiment. Construct validity is the extent to which the variable successfully measures the theoretical construct in a hypothesis. In the approaches suggested above, the theoretical construct is maintainability, and the variables we would measure are lag time in Section 4.1, defect distribution in Section 4.2, and so on. In other words, a question that should be asked in general is: "Does the variable we measure represent maintainability?"

A major threat to the construct validity of the above experiments is the lack of theoretical proof and empirical evidence that the variables we measure represent maintainability. For example, lag time may well be a valid maintainability metric for a closed-source project, where the time taken to diagnose and fix a defect is frequently accurately recorded. In addition to time elapsed, effort (in person-days) may also be recorded. However, for open-source projects, as discussed in Section 3.1, the lag time does not directly represent effort and there is generally no effort information, so we can not use the

lag time until it has been theoretically or empirically proved to be an accurate measure of maintainability.

In summary, we must be sure of the construct validity for any empirical method. Furthermore, we cannot apply a closed-source measurement construct to an open-source project without prior careful examination. Unfortunately, the nature of open-source software development precludes our showing that the factors we can measure indeed represent maintainability. On the contrary, we have given numerous reasons why the approaches we have proposed would not have construct validity.

6. Conclusions

We have examined various sources of open-source maintenance data, such as change logs, source code, and defect tracking systems. We have also examined indirect means of measuring maintainability, such as defect distributions and the lag time to fix a defect. None of the data we have found can be used to measure maintainability of software. The obstacles we have encountered include missing data, incomplete data, inaccurate data, and lack of construct validity.

It is conceivable that open-source data might be found that could be used to measure maintainability. However, none of the artifacts we have encountered to date leads us to believe that such data currently exist.

It is vital for software to be maintainable. Maintainability is particularly critical for open-source software, because maintenance (the production of new versions) is generally the primary activity in open-source projects. However, there does not seem to be a way of measuring the maintainability of an open-source project using the data currently available.

7. Acknowledgments

This work was sponsored in part by the National Science Foundation under grant number CCR–0097056.

8. References

[1] "SourceForge.net: Software Map," 2005, sourceforge.net/softwaremap/trove_list.php.

[2] "Freshmeat.net: Browse project tree – Topic," 2005, freshmeat.net/browse/18/.

[3] W. Harrison, "Editorial: Open source and empirical software engineering," *Empirical Software Engineering* 6 (September 2001), pp. 193–94.

[4] K. Chen, S. R. Schach, L. Yu, G. Z. Heller, and J. Offutt, "Open-Source Change Logs," *Empirical Software Engineering* 9 (September 2004), pp. 197–210.

[5] *A Glossary of Software Engineering Terminology*, IEEE 610.12-1990, Institute of Electrical and Electronic Engineers, Inc., 1990.

[6] "Products and Projects Status Report," June 3, 2003.standards.ieee.org/db/status/status.txt,

[7] S. R. Schach, *Objected-Oriented and Classical Software Engineering*, Sixth Edition, McGraw-Hill, New York, 2005.

[8] B. W. Boehm, J. R. Brown, H. Kaspar, M. Lipow, G. J. MacLeod, and M. J. Merrit, "Characteristics of Software Quality," *TRW series on Software Technology,* North-Holland, Amsterdam, 1978.

[9] "Home :: Bugzilla :: bugzilla.org," 2005, www.bugzilla.org.

[10] "Find a Specific Bug," 2005, bugzilla.mozilla.org/query.cgi.

[11] "ASF Bugzilla Main Page," 2005, issues.apache.org/bugzilla/.

[12] "GNOME Bug Tracking System," 2005, bugzilla.gnome.org/.

[13] "Linux Kernel Tracker," 2005, bugme.osdl.org/.

[14] "Gentoo Linux Bugzilla Home," 2005, bugs.gentoo.org/.

[15] C. Judd, E. Smith, and L. Kidder, *Research Methods in Social Relations*, Sixth Edition, Harcourt Brace Jovanovich, Orlando, 1991.