How Developer Communication Frequency Relates to Bug Introducing Changes

Roberto Abreu
rdz300@few.vu.nl

Rahul Premraj
rpremraj@cs.vu.nl
(contact author)

Department of Computer Science
VU University Amsterdam
The Netherlands

ABSTRACT

Communication between developers plays a very central role in team-based software development for a variety of tasks such as coordinating development and maintenance activities, discussing requirements for better comprehension, assessing alternative solutions to complex problems, and like. However, the frequency of communication varies from time to time—sometimes developers exchange more messages with each other than at other times.

In this paper, we investigate whether developer communication has any bearing with software quality by examining the relationship between communication frequency and number of bugs injected into the software. The data used for this study is drawn from the bug database, version archive, and mailing lists of the JDT subproject in ECLIPSE.

Our results show a statistically significant positive correlation between communication frequency and number of injected bugs in the software. We also noted that communication levels of key developers in the project do no correlate with number of injected bugs. Moreover, we show that defect prediction models can accommodate social aspects of the development process and potentially deliver more reliable results.

Categories and Subject Descriptors:
D.2.8 [*Software Engineering*]: Testing and Debugging; D.2.8 [*Software Engineering*]: Metrics; D.2.9 [*Software Engineering*]: Management

General Terms: Human Factors, Management, Measurement

1. INTRODUCTION

Sound team communication is vital to the success of any software project involving multiple developers. Developers talk amongst themselves to coordinate development activities such as prioritize and assign tasks, comprehend requirements better, discuss candidate solutions to complex problems, and like. Healthy communication is likely to result in good quality software [7]; some benefits include decisions being made as a team, requirements and pending issues are well discussed before implementation, information needs are quickly addressed, and developers are in sync with each others activities. However, in the real world, communication amongst developers may be sporadic, i.e., sometimes they talk intensely between themselves; at others, seldom.

In this paper, we empirically investigate whether the frequency of communication has any impact on the quality on contemporarily written code. Our expectation is that code written when developers frequently talk to each other will be of higher quality as compared to code written when developers communicate infrequently.

We conduct our investigation on data from the JDT project of ECLIPSE (see Section 3) by examining it's communication data and quality of code. Communication data is extracted from the publicly archived mailing lists dedicated to JDT developers. We measure quality of code as the number of commits to the version control repository that have introduced bugs (bug introducing changes) that were later identified and fixed. Thus, fewer number of bug introducing changes reflect higher quality of code and vice versa.

If there is any relationship between communication frequency and bug introducing changes, we expect that social aspects of software development (such as communication) can be leveraged and incorporated into defect prediction models (Section 2) that identify candidate files or packages that require thorough testing to weed out defects.

Our analysis is focussed on investigating the following three hypotheses:

H1 There is a relationship between communication frequency amongst *all developers* and contemporarily developed code (Section 4.1).

H2 There is a relationship between communication frequency amongst *key developers* and contemporarily developed code (Section 4.2).

Key developers are identified using social network analysis techniques.

H3 Communication frequency can be potentially incorporated into defect prediction models to augment their accuracy (Section 4.3).

By investigating the above hypotheses, this paper makes the following contributions:

- provides a first empirical study that directly compares communication frequency of developers with quality of developed code.

IWPSE-Evol'09, August 24–25, 2009, Amsterdam, The Netherlands.

- provides evidence in support of the use of communication data in defect prediction models.

We conclude our paper with reflections on our results, threats to validity (Section 5), and directions for future work (Section 6).

2. RELATED WORK

Communication related metrics have been previously used in the software engineering domain to extrapolate the latent social structures underlying the OSS development with the application of social network analysis methods [2, 5, 6, 9].

Additionally, the social network analysis methods have been applied to predict post-release defects. Zimmermann and Nagappan [18] used network analysis methods on low-level dependency graphs extracted from Windows Sever 2003 to predict post-release defects. Meneely et al. [12] constructed developer collaboration structures by examining code churn information to predict file-level defects. Pinzger et al. [13] also focussed on predicting post-release defects by examining the relationship between number and structure of developer contributions to software modules. Hossain and Zhou [8] examined whether network measures have any bearing on the quality of software projects. They conceptualized quality as (a) number of defects fixed per software promotion, (b) the number of defects reported at different severity levels, and (c) the average number of days for a defect to be fixed for different project teams. More recently, Wolf et al. [17] used a combination of several network metrics to estimate quality of products developed in geographically distributed teams that demand high coordination efforts. Similar to Hossain and Zhou [8], they found that a combination of network measures correlate with fluctuations in software integration quality. Additionally, they found no single measure as a good estimate for the success or failure of a software integration.

In contrast to previous research, we specifically investigate the relationship between team communication and the number of bugs introduced into the source code and provide an initial assessment of the possibility of using communication frequency as an indicator of defect prone files.

3. DATA COLLECTION

We focussed on the JDT project of ECLIPSE for this study. The JDT project is a set of core plug-ins that comprise the ECLIPSE development environment for JAVA development. We considered JDT be well-suited for this study because it is sufficiently large in size, has several active developers, and a vigorous mailing list (approx. 12 messages per week).

To conduct our analysis, we need to extract (a) the count of bugs introduced into the source code and when they were introduced and (b) the frequency of messages exchanged between developers in the JDT mailing list. The following two sub-sections elaborate upon our method for collecting these data.

3.1 Count of bug introducing changes

Bug introducing changes are those made to a file that were later identified as defects and then fixed. Our analysis requires the count of such changes in the JDT project and when their respective commit timestamps. The latter is needed to sum the counts of bug introducing changes by week to compare it with communication frequency.

Śliwerski et al. [15] proposed an approach that leverages bug databases and the version control system of the project (in their case, BUGZILLA and CVS) to extrapolate the location and time when a bug was introduced into the code. Their approach can be summarised as follows:

Link the bug database and version control system. First, examine the log messages from CVS to search for patterns such as *Fixed Bug #* or *PR #*. This enables establishing links between a bug report and the CVS transaction that fixes the bug, thus revealing the author, time, and location (file names and line numbers) of the fix in the source code.

Find the bug introducing change. For each fixed bug report that can be matched to a corresponding CVS transaction, use the ANNOTATE command to identify the last modification made (previous to the date the bug report was filed) to the same location (file name line numbers) in the source code. This CVS transaction is considered to be the bug introducing change, i.e., the change in the source code that introduced the defect, which was later fixed.

The authors [15] made their data set available to us for the purpose of this study. The data set comprised bug introducing changes in the ECLIPSE project from November 4th, 2001 to November 24th, 2006. We filtered the data set to only include bugs introduced in the JDT project and grouped the counts by week using the timestamps of the commits.

Note that we are aware of the risks involved with the approach used by Śliwerski et al. [15] to identify bug introducing changes and have discussed them in Section 5.

3.2 Extraction of Communication Data

Like most ECLIPSE projects, JDT archives the messages exchanged on its mailing lists[1] each month and makes them publicly available. These messages can be easily downloaded using free tools such as `wget` and processed to extract information from them. We downloaded all JDT message archives, processed them, and imported them into a database.

We extracted the following fields from the messages using regular expressions: *message-id* (a unique identifier for a message), *reference-id* (a response's identifier which is same as the message-id of the parent message), *subject*, *sender name*, *sender email*, and *date*.

Number of messages exchanged per week.

This was achieved by using the *Date* field of the messages and aggregating the number of messages exchanged each week.

Number of message threads started per week.

A message thread is a collection of a parent message and its responses. Extracting this information could have been simple if each message contained a *reference-id*. However, this was not the case, plausibly because people replied to messages by copying the text into a new email rather than using the reply button. We circumvented this problem by focussing on the subject of the messages: nearly all messages belonging to a single thread contained the same subject as the original message with "Re:" as a prefix. Messages with missing *reference-ids* and "Re:" in their subjects were interpolated with *message-ids* of the most recent email bearing the same subject. A manual inspection of the resulting data confirmed that this approach was reliable to reconstruct email threads.

Identifying distinct developers in the mailing list.

Developers often use multiple email aliases to participate in the mailing lists. We borrowed prescribed heuristics by Bird et al. [1,2] and Sowe [16] to infer distinct developers such as matching email prefixes (e.g., similarity of usernames, inferring usernames from

[1]Projects often have more than one mailing list.

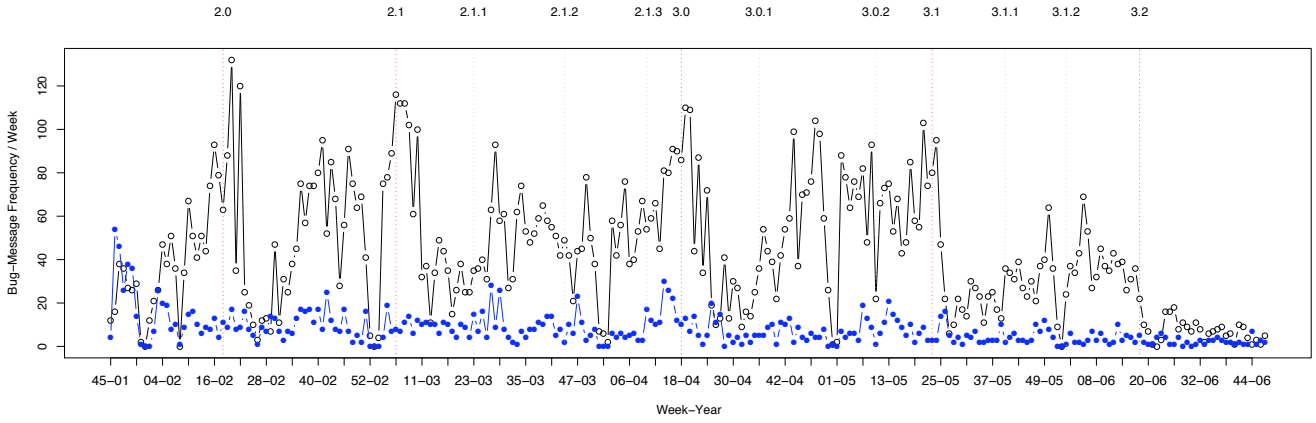

Figure 1: A plot of weekly number of messages exchanged on the JDT mailing list (solid circles) and number of bug introducing changes (hollow circles) in the JDT code.

names and measuring username similarity) and name-email matching (e.g., similarity of names). Using the heuristics, we identified 380 distinct individuals from the 439 name and email address pairs extracted from the messages. Again, we manually inspected the results and found them very reliable.

Note that all 380 individuals identified from the mailing lists were not necessarily developers. Rather than applying further heuristics to identify developers, we used the list of 35 JDT developers provided on the project's web page [10] for our analysis. Furthermore, recall that the bug introducing changes data only ranged from November 4th, 2001 to November 24th, 2006. Hence, we truncated the data extracted from the mailing lists to match this time period. During this time period, only 18 of the 35 listed developers were active, so we only considered messages exchanged by them.

4. EXPERIMENTS AND RESULTS

We present the details of our experimental setup and results from our analysis in the following sections. Our analysis was conducted using the R statistical software [14].

4.1 Bug introducing changes vs. Communication frequency

We begin our analysis by investigating if there is any relationship between communication frequency and bug introducing changes (hypothesis **H1**). Figure 1 plots the weekly count of messages exchanged and bugs introduced in the JDT project starting November 2001 to November 2006. At the top of the plot, we indicate when in time which ECLIPSE version has been released. A quick glance at the plot shows that peaks in number of bug introducing changes occur around scheduled release dates. Hence, concentrating test efforts on files changed during these times should simply suffice. However, only five major releases (2.0, 2.1, 3.0, 3.1, and 3.2) have occurred in the time window of our data set; in-between these releases, there are many further peaks in the number of bug introducing changes.

To verify if such peaks coincide with any pattern in communication, we apply a statistical method called *cross-correlation* [4] on the weekly count of bug introducing changes and communication frequency. Cross-correlation verifies if two entities are related to each other with a systematic time lag. This is achieved by repeatedly computing correlation values by introducing a series of time lags to one of the entities. This allows us to verify if communication between developers systematically coincides, precedes, or follows bug introducing changes. We computed correlation values between the number of bug introducing changes and lagged frequency of communication. We chose to compute correlations using a time lag of up to ten weeks (equal to a just over two months) to account for the possibility that developers discuss important changes way ahead of implementing them or experience the consequences of bug introducing changes after a long delay.

Figure 2 plots the results from the cross-correlation analysis. We computed Spearman's correlation (ρ) which is a non-parametric variant of correlation suited to non-normally distributed data (as was our case). In the figure, the *x-axis* marks the number of weeks by which the communication frequency data was shifted. Thus, at lag 0, correlation was computed between the number bug introducing changes and communication frequency in the same week; correlations for positive valued lags were computed by shifting the communication frequency ahead, while for negative valued lags, correlations were computed by shifting communication frequency behind.

We see that the correlation peaks at lag 0 standing at $\rho = 0.47$ and statistically significant with $p < .0001$; thus communication frequency and number of bug introducing changes are most strongly in sync in the same week. Additionally, we examined if the number of message threads started in a week bear any correlation with the number of introduced bugs. A Spearman's correlation analysis resulted in $\rho = 0.45$ and statistically significant at $p < 0.001$. Such a correlation value is fairly high in the software engineering domain where several confounding forces are operational at the same time.

It is more noteworthy that both correlation values are positive suggesting that increase in communication coincides with a high number of bug introducing changes and vice versa, which is quite the contrary to our expectations. We looked further into why the correlation values are positive by examining some of the messages exchanged when bug introduced peaked. Many messages seem to be sent out after a commit to notify forthcoming changes, caution others from using some parts of the code, or report a new failure. Some excerpts from the messages are below (messages' subjects are in bold):

Version v428 a posted as early preview. *"[. . .] Please note that further changes may be committed until we integrate, but unless notified these should not affect the overall behavior of this preview [. . .]"*

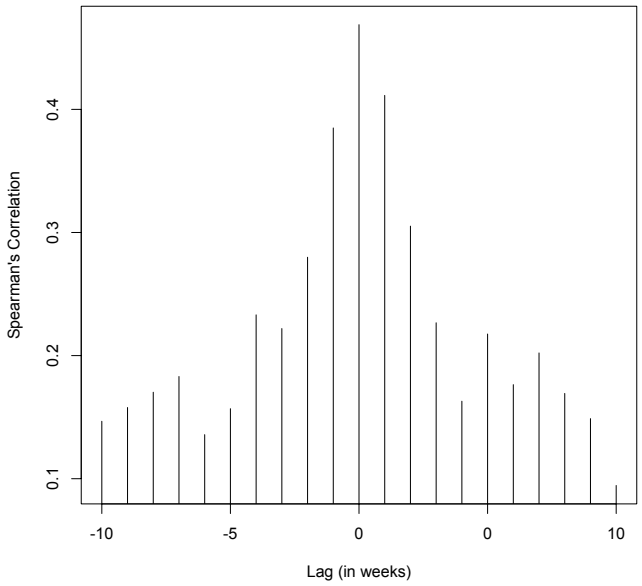

Figure 2: Cross-correlation between bug introducing changes and communication frequency.

New for loop. *"[. . .] We still need to do more testing, but the implementation is in place [. . .]"*

Released to HEAD. *"7985 extract method: cannot extract (entire) try/catch block"*

Peaks in communication may coincide with peaks in bug introducing changes, even between the product's release dates.

4.2 Centrally Communicating Developers

In Section 4.1, we investigated the relationship between bug introducing changes and communication between *all* 18 JDT developers. But teams often comprise some *key* developers who specialize in different parts of the system and are a source of knowledge for the remaining team. In this section, we investigate whether fluctuations in communication frequency of key developers may have any influence on the number of bug introducing change (hypothesis **H2**)—when key developers are active, it is likely that knowledge is dispensed more often, but when they become dormant, other developers may be less guided and introduce relatively more bugs.

To identify key developers, we used a social network analysis metric called *degree of centrality*. This metric measures importance for each developer by summing the number of messages sent and received: more the number of messages, higher the degree of centrality. Degree of centrality can also be computed as *in-degree* (number of messages received) and *out-degree* (number of sent messages). We considered all three measures of centrality for our analysis; *in-degree* because identifies developers whose opinions are most sought after, *out-degree* because it identifies developers who share more knowledge with others [5], and *degree of centrality* because it incorporates both previous metrics.

A directed graph of the communication network of the 18 developers is plotted in Figure 3. The graph has been plotted using the *sna* package [3] available for the R statistical software. Each node in the graph represents a developer (names obfuscate to maintain anonymity). Arrows indicate the direction of flow of the messages from one developer to the other.

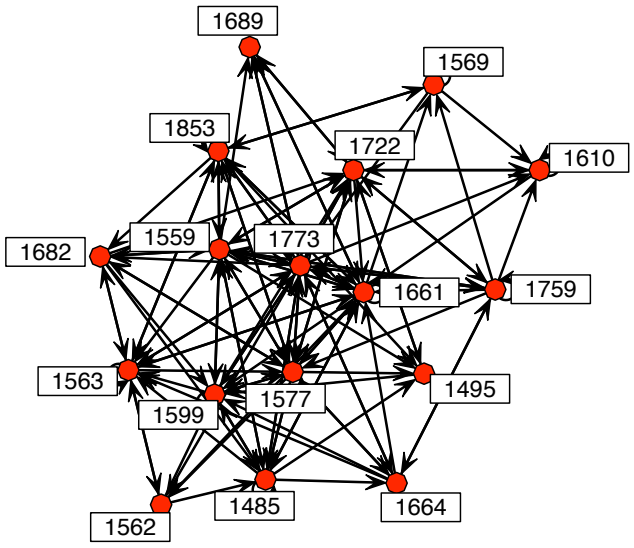

Figure 3: A directed graph connecting JDT developers who exchanged messages with each other.

We identified four developers with the highest scores for all three centrality measures, namely developers *1773*, *1661*, *1577*, and *1559*. These developers are also positioned in the centre of Figure 3 because the *sna* package uses the Kamada and Kawai algorithm [11] to plot directed graphs which places central nodes in the centre of the graph.

Next, we computed Spearman's correlation between the number of weekly messages received / sent by each of these four central developers and the weekly number of bug introducing changes committed to the JDT repository. All correlation values were low ranging from 0.06 to 0.16, yet statistically significant at $p < .05$. These values suggest that there is no relation between the activity of key developers on the mailing lists and the number of introduced bugs. Our results support the recommendation by Crowston and Howison [5] to focus on network measures at the project level and not individual level.

The communication levels of the project's entire development community, and not strictly key developers, are indicative of the number of bug introducing changes.

4.3 Potential Predictive Power

In Section 4.1, we showed that peaks in communication and bug introducing changes are not restricted to around the project's release times. Software testers can potentially leverage this information help narrow down parts that require more testing. In this section, we provide evidence to support that knowledge of communication frequency is valuable to identify files with bugs (hypothesis **H3**).

We checked whether the files that were committed during weeks experiencing peaks in communication frequency have high number of bugs. To do so, we first located files with more than 50 fixed bugs in the time window of our data set. Next, we identified weeks in which both, weekly communication frequency and bugs introduced exceeded their respective means by one standard deviation. Hence, these are weeks when both, communication frequency and number of bug introducing changes peaked; more than thirty such weeks were found.

We sampled ten of these thirty weeks while ensuring that they

were at least 4 weeks apart from any ECLIPSE release date and inspected the files changed during these weeks. In each week from this sample, at least 10, and up to 17 of the 39 previously identified files were found to be changed. This suggests that communication frequency, perhaps when coupled with other metrics such as static code metrics, may be valuable in identifying files that require thorough testing.

Communication frequency may potentially serve as a good indicator of files that should be thoroughly tested for bugs.

5. THREATS TO VALIDITY

Like other studies of this nature, ours too faces some threats to validity. First, our analysis is restricted to measuring communication frequency over mailing lists that do not necessarily capture all communication between developers. However, our results show that the data from the lists can be potentially put to use to deliver higher quality software.

Second, bug introducing changes were identified using commit logs from the CVS archive. This approach relies upon developers' discipline in authoring commit messages that indicate whether the change is a bug fix. Thus, there is a chance that some mappings between bug reports and their respective fixes may have been missed. Additionally, it is not necessary that a bug may have been introduced in the most recent CVS transaction that changed the relevant lines in the file.

Lastly, our results have been drawn from a single ECLIPSE subproject, that too open-source. Further investigations on more open- and close-source projects are required to arrive at more general conclusions and consequences of this research.

6. CONCLUSIONS AND CONSEQUENCES

Communication between software developers is vital for them to organise their development activities. However, communication is not consistently frequent; sometimes developers talk more amongst each other and at others, seldom.

In our examination of communication frequency of developers from the JDT project, we have shown that the level of communication has a strong and positive relation with number of bugs introduced in contemporarily authored code. This suggests that developers tend to talk more often at times when a higher number of bugs are introduced in the software. This knowledge can be used by test managers working with constrained resources by examining the development teams communication archives and focus on testing files changed in time periods that experienced high communication.

This new insight also supports the use of social aspects of development in defect prediction models to gain better prediction accuracy at identifying defect prone files. We have further bolstered our support by showing that often during peaks of communication, many bug introducing changes were made to some of the most defect prone files in the JDT project.

Additionally, we have also shown that communication levels of the entire development community, and not strictly key developers, bears a relationship with the number of bug introducing changes. Thus, defect prediction models must focus on project level network metrics rather than at the individual level.

In the future, we plan to extend our study by analysing communication archives of multiple projects to verify if our results are generalizable. In addition, we plan to build defect prediction models that incorporate both, code and social metrics, to identify files with potentially high number of bugs.

Acknowledgements. The authors thank Thomas Zimmermann for making the data on bug introducing changes available to us. Also, many thanks to the anonymous reviewers for their valuable and helpful suggestions on earlier revisions of this paper. Lastly, thanks to Nicolas Bettenburg for providing us the template to plot Figure 1.

7. REFERENCES

[1] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan. Mining email social networks in postgres. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 185–186, New York, NY, USA, 2006. ACM.

[2] C. Bird, D. Pattison, R. D'Souza, V. Filkov, and P. Devanbu. Latent social structure in open source projects. In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 24–35, New York, NY, USA, 2008. ACM.

[3] C. T. Butts. sna: Tools for social network analysis, 2007. R package version 1.5.

[4] J. Y. Campbell, A. W. Lo, A. C. MacKinlay, and A. Y. Lo. *The Econometrics of Financial Markets*. Princeton University Press, 1996.

[5] K. Crowston and J. Howison. The social structure of free and open source software development. *First Monday*, 10(2), 2005.

[6] K. Crowston and J. Howison. Hierarchy and centralization in free and open source software team communications. *Knowledge, Technology and Policy*, 18(4):65–85, December 2006.

[7] J. H. Hayes. Do you like piña coladas? How improved communication can improve software quality. *IEEE Software*, 20(1):90–92, 2003.

[8] L. Hossain and D. Zhou. Measuring OSS quality through centrality. In *CHASE '08: Proceedings of the 2008 international workshop on Cooperative and human aspects of software engineering*, pages 65–68, New York, NY, USA, 2008. ACM.

[9] J. Howison, K. Inoue, and K. Crowston. Social dynamics of free and open source team communications. In *Proceedings of the IFIP 2nd International Conference on Open Source Software*, volume 203/2006, pages 319–330, Boston, USA, 2006. Springer.

[10] JDT Development Team. Project summary - eclipse.jdt. Last accessed 2009-05-27.

[11] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7–15, 1989.

[12] A. Meneely, L. Williams, W. Snipes, and J. Osborne. Predicting failures with developer networks and social network analysis. In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 13–23, New York, NY, USA, 2008. ACM.

[13] M. Pinzger, N. Nagappan, and B. Murphy. Can developer-module networks predict failures? In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 2–12, New York, NY, USA, 2008. ACM.

[14] R Development Core Team. R: A language and environment for statistical computing, 2009. ISBN 3-900051-07-0.

[15] J. Sliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *MSR '05: Proceedings of the 2nd International Workshop on Mining Software Repositories*, pages 24–28, St. Loius, USA, May 2005.

[16] S. Sowe, I. Stamelos, and L. Angelis. Identifying knowledge brokers that yield software engineering knowledge in OSS projects. *Information and Software Technology*, 48(11):1025–1033, November 2006.

[17] T. Wolf, A. Schroter, D. Damian, and T. Nguyen. Predicting build failures using social network analysis on developer communication. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, pages 1–11. ACM, May 2009.

[18] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *ICSE '08: Proceedings of the 30th International Conference on Software engineering*, pages 531–540. ACM, May 2008.