2016 IEEE International Conference on Software Maintenance and Evolution

Using Temporal and Semantic Developer-Level Information to Predict Maintenance Activity Profiles

Stanislav Levin
The Blavatnik School of Computer Science
Tel Aviv University
Tel-Aviv, Israel
stanisl@post.tau.ac.il

Amiram Yehudai
The Blavatnik School of Computer Science
Tel Aviv University
Tel-Aviv, Israel
amiramy@tau.ac.il

***Abstract*—Predictive models for software projects' characteristics have been traditionally based on project-level metrics, employing only little developer-level information, or none at all. In this work we suggest novel metrics that capture temporal and semantic developer-level information collected on a per developer basis. To address the scalability challenges involved in computing these metrics for each and every developer for a large number of source code repositories, we have built a designated repository mining platform. This platform was used to create a metrics dataset based on processing nearly 1000 highly popular open source GitHub repositories, consisting of 147 million LOC, and maintained by 30,000 developers. The computed metrics were then employed to predict the corrective, perfective, and adaptive maintenance activity profiles identified in previous works. Our results show both strong correlation and promising predictive power with R^2 values of 0.83, 0.64, and 0.75. We also show how these results may help project managers to detect anomalies in the development process and to build better development teams. In addition, the platform we built has the potential to yield further predictive models leveraging developer-level metrics at scale.**

***Index Terms*—Software Maintenance; Software Metrics; Mining Software Repositories; Predictive Models; Human Factors;**

I. Introduction

Forecasting maintenance activities performed in a source code repository could help practitioners reduce uncertainty and improve cost-effectiveness [1] by planning ahead and pre-allocating resources towards source code maintenance. Maintenance activity profiles of software projects have been therefore a subject of research in numerous works [1–4]. In this work we adopt the maintenance activity categories **corrective**, **perfective**, and **adaptive** as defined by Mockus et al. [2]:

- Corrective: fault fixing.
- Perfective: code structure / system design improvements.
- Adaptive: new feature introduction.

and put forth previously unexplored **temporal and semantic developer-level metrics**, which we then utilize to study the corrective, perfective and adaptive maintenance activity profiles on a developer-level granularity.

We seek to gain better understating of how personal characteristics such as commit patterns, commit frequencies, project join and departures dates, etc., impact the maintenance activities of individual developers, as well as the projects they work on. Moreover, given a software project, we consider its maintenance activity profiles as an aggregation of the maintenance activity profiles of all developers' that have taken part in its development. Therefore, predictions made on a developer-level (i.e., for all individual developers in the project), can be used to reason about, and derive project-level predictions.

Studying developer-level impact requires sufficient and sufficiently fine-grained data, as well as the computational power to process it. This work is therefore driven by two main factors that have been trending up for the past decade:

- Big Code [5]: the availability of large source code corpora via open source.
- Big Data [6] ecosystem: the availability of tools capable of processing extremely large data volumes.

The combination of the two has created unprecedented opportunities to collect and process an enormous volume of source code, and provide insights that were previously exponentially harder, or even impossible to obtain [7].

II. Related Work

Maintenance activity profiles and the application thereof have been the subject of numerous works dealing with fault prediction models, commit classification, software change recommendations, and more [2,4,8–11]. While the precise distribution of maintenance activities is inconclusive [8], their classification into the corrective, perfective and adaptive categories has been a common practice. Our work therefore seeks to explore how these maintenance activity categories relate to the semantic and temporal developer-level metrics we define, and whether such metrics can be used to build effective predictive models for the corrective, perfective and adaptive maintenance activity profiles.

III. Research Question

RQ: How do temporal and semantic developer-level metrics relate to developers maintenance activity profiles?

To the best of our knowledge, this work is the first to explore temporal and semantic *developer-level* metrics (see Table I), such as the number of distinct semantic changes performed by each developer, the mean number of distinct semantic changes in a given developer's commits, mean time between commits and others. Moreover, we explore these metrics at large scale,

978-1-5090-3806-0/16 $31.00 © 2016 IEEE
DOI 10.1109/ICSME.2016.21

and analyze nearly 1000 different repositories that consist of dozens of millions of commits in total. Our large scale study was conducted using a VCS mining platform we have built to enable large scale analysis of version control systems (VCS). Our platform leverages Spark [12], an industrial cutting edge data processing framework.

We used the GQM approach [13] to derive the questions, and then the metrics that are needed in order to answer the research question in a measurable way. The developer-level metrics we measured are listed in Table I.

TABLE I
DEVELOPER-LEVEL METRICS

Commits_{repo}	See Table II
Muse_{repo}	
$\text{DeveloperVersatility}_{repo}$	
$\text{MeanCommitV}_{repo}$	
$\text{VersatilityLevel}_{repo}$	
$\text{ContribStartRel}_{repo}$	Developer's join date (first recorded commit) expressed as the number of days since project's first observed commit
$\text{ContribDuration}_{repo}$	The duration in days, of the period between the developer's first and last commit dates.
MTBC_{repo}	**M**ean **t**ime **b**etween developer's **c**ommits, in days.

* Developers who have committed changes (contributed) to multiple source code repositories are considered as if they were different individuals.

IV. BACKGROUND

Fluri et al. [14] put forth a taxonomy of semantic source code changes for object-oriented programming languages (OOPLs), and Java in particular. This taxonomy consists of 47 different change types, such as *statement_delete*, *statement_insert*, *statement_update*, *removed_class*, *additional_class*, *return_type_change* and so on. The computation of these change types from source code files was later implemented by a tool named "ChangeDistiller" [15].

We embrace this taxonomy and define the notion of versatility based measures for developers and commits as defined in Table II.

For example, if developer $Alice$ performed 2 commits:

$commit_1$	2 x *stmt_insert*, 1 x *stmt_update*
$commit_2$	3 x *stmt_delete*

$\text{CommitVSet}_{repo}(commit_1) = \{\text{stmt\_insert}, \text{stmt\_update}\}$

$\text{CommitVSet}_{repo}(commit_2) = \{\text{stmt\_delete}\}$

$\text{DeveloperVSet}_{repo}(Alice) = \{\text{stmt\_insert}, \text{stmt\_update}, \text{stmt\_delete}\}$

$\text{CommitVersatility}_{repo}(commit_1) = 2$

$\text{CommitVersatility}_{repo}(commit_2) = 1$

$\overline{\text{CommitVersatility}}_{repo}(Alice) = \frac{2+1}{2} = 1.5$

$\text{DeveloperVersatility}_{repo}(Alice) =\mid \text{DeveloperVSet}_{repo}(Alice) \mid = 3$

$\text{Muse}_{repo}(Alice) = \max\{2, 1\} = 2$

$\text{VersatilityLevel}_{repo}(dev) =\mid \{\{\text{stmt\_insert}, \text{stmt\_update}\}, \{\text{stmt\_delete}\}\} \mid = 2$

The intuition behind these versatility based measures is capturing the number of different fine-grained semantic change types present in commits made by individual developers.

V. METRICS COMPUTATION

To provide reproducible results, we sought to base our empirical study on publicly (and freely) available data. We chose GitHub as the data source for this work due to its prevalence among the repository hosting services. Candidate repositories were selected according to the following criteria:

1) used the Java programming language
2) had more than 100 stars (i.e. more than 100 users had "liked" these repositories)
3) had more than 60 forks (i.e., more than 60 users had "copied" these repositories for their own use)
4) had their code updated since 2016-01-01 (i.e., these repositories were active)
5) were created before 2015-01-01 (i.e., these repositories had been around for at least $\sim$1.5 years)
6) had size over 2MB (i.e. these repositories were beefy)

The result set consists of 1000 repositories, the maximum limit as stated by GitHub's documentation [16]. Due to various technical reasons our final dataset consisted of 979 unique Git repositories, where the average number of developers per project was 32, and the average project age was 4.2 years. These repositories were then cloned and processed by our VCS mining platform

First, to distill semantic source code changes as per the taxonomy defined by Fluri et al. (see also section IV), our VCS mining platform repeatedly applied the ChangeDistiller ([15,17,18]) on every two consecutive revisions of every Java file in every repository in our result set. This stage yielded 30 million semantic source code change instances. Then, all the semantic source code changes were aggregated using the key (developer-id, repository-id), forming data bins from which the temporal and semantic developer-level metrics (see Table I) were computed. To classify developer's commits into the corrective, perfective, and adaptive categories, our VCS mining platform used methods similar to [2,19,20], and searched for indicative keywords in the commit's comment field. Keyword matching was boosted by using common techniques such as stemming and case-folding, the keywords are listed in table III.

VI. RESULTS

We use generalized regression modeling (GLM) [21] in the R statistical environment [22] to explore our dataset and build predictive models. The predictive models were trained on randomly chosen 90% of the repositories in the dataset, while the remaining 10% were used for validation and measuring goodness of fit.

In the rest of this section we present the predictive models (see Table IV) for the developer-level corrective, perfective and adaptive maintenance activity profiles. To predict the profile of a maintenance activity category $\text{MA}_c \in \{\text{Corrective}, \text{Perfective}, \text{Adaptive}\}$

TABLE II
VERSATILITY BASED MEASURES FOR DEVELOPERS AND COMMITS

Given a source code repository $repo$ and a developer dev that has committed code to $repo$:

$\mathbf{Commits_{repo}(dev)} = \{\text{all commits by developer } dev \text{ to repository } repo\}$

$\mathbf{CommitVSet_{repo}(commit)} := \{\text{all semantic source code changes in } commit\}$

$\mathbf{DeveloperVSet_{repo}(dev)} := \bigcup_{commit \in \text{Commits}_{repo}(dev)} \text{CommitVSet}_{repo}(commit)$

$\mathbf{CommitVersatility_{repo}(commit)} :=| \text{CommitVSet}_{repo}(commit) |$

$\mathbf{DeveloperVersatility_{repo}(dev)} :=| \text{DeveloperVSet}_{repo}(dev) |$

$\mathbf{Muse_{repo}(dev)} := \max_{commit \in \text{Commits}_{repo}(dev)} \text{CommitVersatility}_{repo}(commit)$

$\overline{\mathbf{CommitVersatility_{repo}(dev)}} := (\sum_{commit \in \text{Commits}_{repo}(dev)} \text{CommitVersatilisy}_{repo}(commit)) * \frac{1}{|\text{Commits}_{repo}(dev)|}$

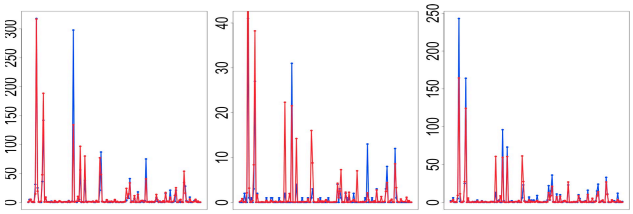
$\mathbf{VersatilityLevel_{repo}(dev)} :=| \bigcup_{commit \in \text{Commits}_{repo}(dev)} \{S | S = \text{CommitVSet}_{repo}(commit)\} |$

Note that: $\text{DeveloperVersatility}_{repo}(dev) \geq \text{Muse}_{repo}(dev)$

TABLE III
KEYWORDS FOR CLASSIFYING MAINTENANCE ACTIVITIES

Corrective	*fix, esolv, clos, handl, issue, defect, bug, problem, ticket*
Perfective	*refactor, re-factor, reimplement, re-implement, design, re-plac, modify, updat, upgrad, cleanup, clean-up*
Adaptive	*add, new, introduc, implement, implemented, extend, feature, support*

Fig. 1. Prediction graphs for corrective, perfective and adaptive (left to right) maintenance activity profiles for 150 developers randomly selected form the test dataset. Actual values in blue, predicted ones are in red.



for a developer dev the following formula was used:

$$\text{Profile}_{\text{MA}_c}(dev) = \text{Const}_{\text{MA}_c} + \sum_{P_i \in model_{\text{MA}_C}} \text{coeff}^{P_i}_{\text{MA}_c} * P_i(dev)$$

where $model_{\text{MA}_c}$ is the regression model for MA_c, $\text{Const}_{\text{MA}_c}$ is the constant in $model_{\text{MA}_c}$, and $\text{coeff}^{P_i}_{\text{MA}_c}$ is the coefficient of predictor P_i in $model_{\text{MA}_c}$ as specified in table IV. $P_i(dev)$ is the value of predictor P_i for a given developer dev.

All predictors were log transformed to alleviate skewed data, a common practise when dealing with software metrics [23,24]. The standard error is specified in parenthesis underneath the estimate for each predictor. Figure 1 presents predictions results for 150 developers randomly selected from the test dataset.

For each maintenance activity category MA_c, the X axis is a running developer-id and the Y axis is the number of the developer's commits of category MA_c. For each developer, we plot in blue the number of her commits classified as MA_c by the commit message classification algorithm (see section V), and overlay it in red with the number of commits predicted to be of category MA_c by the GLM (see Table IV).

For all three maintenance activity profiles, the vast majority of both temporal and versatility based developer metrics were statistically significant with *p-value* < 0.01. Metrics that were not statistically significant were excluded from the predictive models (represented by the empty cells in table IV) to improve prediction quality.

TABLE IV
GLM FOR DEVELOPER-LEVEL MAINTENANCE ACTIVITY

	Predictor	*Predicted profile:* Corrective (1)	Perfective (2)	Adaptive (3)
(P_1)	$log(\text{Commits}_{repo})$	0.797 (0.010)	0.572 (0.020)	0.503 (0.015)
(P_2)	$log(\text{Muse}_{repo})$	0.171 (0.010)	−0.288 (0.020)	−0.135 (0.013)
(P_3)	$log(\text{MTBC}_{repo})$	0.012 (0.002)	−0.018 (0.004)	
(P_4)	$log(\text{ContribStartRel}_{repo} + 0.1)$	0.014 (0.001)		−0.021 (0.001)
(P_5)	$log(\overline{\text{CommitVersatility}}_{repo})$	0.028 (0.009)		0.033 (0.013)
(P_6)	$log(\text{ContribDuration}_{repo} + 0.1)$	0.030 (0.002)	−0.050 (0.005)	−0.018 (0.002)
(P_7)	$log(\text{DeveloperVersatility}_{repo})$	−0.205 (0.010)	0.394 (0.025)	0.243 (0.013)
(P_8)	$log(\text{VersatilityLevel}_{repo})$	0.181 (0.012)	0.483 (0.025)	0.437 (0.017)
	Constant	−0.986 (0.019)	−3.092 (0.048)	−1.462 (0.020)
	R^2	**0.832**	**0.640**	**0.759**
	Observations	27,850	27,850	27,850

In all three models, Commits_{repo} (the total number of commits made by a developer to the given repository) was the most powerful predictor, and accounted for a great of deal the high R^2 values.

Corrective profile (table IV, column 1): In contrast to other predictors, *Versatility*$_d$ has a negative coefficient indicating that developers with higher *Versatility*$_d$ values are likely to perform less corrective commits given that other predictors remain fixed. The fact that *Muse* and *Versatility*$_d$ are both versatility based metrics, yet have opposite signs in this model, supports our assumption that *Muse* and *Versatility*$_d$ capture different kinds of information. In addition, it is also evident that developers who commit less frequently (higher MTBC), join the project later, remain active for longer, and have more commits with distinct change type patterns, are likely to

have a higher corrective profile (i.e., perform more corrective commits).

Perfective profile (see Table IV, column 2): Similarly to the corrective model, the signs of *Versatility*$_d$ and *Muse* are opposite, but in contrast to the former, it is now *Muse* that has a negative sign. This indicates that developers with higher *Muse* values are likely to perform less perfective commits given that other predictors remain fixed. Also, developers who commit more frequently (lower MTBC), and remain active for shorter time (lower ContribDuration), but have more commits with distinct change type patterns (i.e., higher values for DistinctChangeTypeSets) are likely to have a higher perfective profile (i.e., perform more perfective commits).

Adaptive profile (see Table IV, column 3): The adaptive model is more similar to the perfective one than to the corrective one, with *Versatility* having a positive sign and *Muse* a negative sign. Adaptive commits are likely to favour developers with lower Muse, who join the project earlier (lower ContribStartRel), remain active for shorter time (lower ContribDuration), have more commits with distinct change type patterns, and have a greater versatility (as defined in Table II).

VII. Discussion & Applications

Identifying anomalies in development process. The manager of a large software project should aim to control and manage its maintenance activity profiles. Monitoring for unexpected spikes in maintenance activity profiles and investigating the reasons (root cause) behind them would assist managers and other stakeholders to plan ahead and identify areas that require additional resource allocation. For example, lower corrective profiles could imply that developers are neglecting bug fixing. Higher corrective profiles could imply an excessive bug count. Finding the root cause in cases of significant deviations from predicted values may reveal essential issues whose removal can improve projects' health. Similarly, exceptionally well performing projects can also be a good subject for investigation in order to identify positive patterns.

Improving development team's composition. Building a successful software team is hardly a trivial task as it involves a delicate balance between technological and human aspects [25,26]. We believe that developers' maintenance activity profiles could assist in composing a more balanced team. We conjecture that composing a team that heavily favors a particular maintenance activity over the others could lead to an unbalanced development process and adversely affect the team's ability to meet typical requirements such as developing a sustainable number of product features, adhering to quality standards, and minimizing technical debt so as to facilitate future changes.

VIII. Threats to validity

Threats to Statistical Conclusion Validity is the degree to which conclusions about the relationship among variables based on the data are reasonable. Our results are based on nearly 30,000 observations, and the predictors are statistically significant with *p-value* < 0.01.

Threats to Construct Validity consider the relationship between theory and observation, in case the measured variables do not measure the actual factors.

- Volatile Classification. While the method we used is commonly practiced ([2,19,20]), our experiments show it may be sensitive to the choice of keywords used as indicative for the various maintenance activity types.
- Semantic Change Extraction. ChangeDistiller ([15,17]) was used to extract semantic changes from the VCS. Unfortunately, like any other software, it may not be immune to bugs and malfunctions.
- Developer-level Metrics Computation. To compute developer-level metrics we used a novel VCS mining platform we had built to support this study. While we invested great effort into testing it to ensure its proper functionality, it may not be immune to bugs and malfunctions.

Threats to External Validity consider the generalization of our findings.

- Programming Language Bias. All analyzed commits were in the Java programming language. It is possible that developers who use other programming languages, have different maintenance activity patterns which have not been explored in the scope of this work.
- Open Source Bias. The repositories studied in this paper were all popular open source projects from GitHub ([27]). It may be the case that developers' maintenance activity profiles are different in an open source environment when compared to other environments.
- Popularity Bias. We intentionally selected the most popular, data rich repositories. This could limit our results to developers and repositories of high popularity, and potentially skew the perspective on characteristics found only in less popular repositories and their developers.

IX. Conclusions and Future Work

We have demonstrated that the developer-level metrics we have defined are statistically significant, and can be successfully used to model and predict the corrective, perfective and adaptive maintenance activity profiles with promising R^2 values of 0.83, 0.64 and 0.75 respectively. Our work is based on studying Big Code ([5]), and involved processing nearly 1000 highly popular open source GitHub repositories, comprising a corpus of 147 million LOC, maintained by 30,000 developers, spread over 2.5 million revisions and 30 million individual code change instances.

We believe that considering a project's characteristics as an aggregation of the characteristics of all the developers' that have taken part in its development, and modeling the former using the latter, is a useful technique. It might, for example, assist in predicting further project characteristics such as fault potential, which has traditionally relied on project-level metrics [10,28–32] or only limited developer-level metrics

[33–36] (e.g., numeric representations of experience, number of developers that changed a particular file or method).

In light of the promising results, we are in the process of conducting further studies that involve developer-level metrics. For example, we are working on comparing project-level and developer-level maintenance activity profiles. Preliminary results indicate that the *variance* in developer-level maintenance activity profiles is much greater than it is in the project-level ones.

We believe that the combination of Big Code and fined grained developer-level metrics, can open the door to a new range of applications and empirical studies. A particularly compelling direction is fusing more data sources such as bug tracking systems, social Q&A sites (e.g., StackOverflow [37]) and others, to study how these types of data relate to developers' characteristics such as commit patterns, temporal activity, fault potential, etc. Augmented with developer surveys to validate the relations, we hope these studies could shed new light on our understanding of software development and evolution.

ACKNOWLEDGEMENTS

We thank Ms Ilana Gelernter and the Tel Aviv University's statistical counseling service for the help with the statistical analysis, and Dr. Boris Levin for comments that greatly improved the manuscript. This research was supported by THE ISRAEL SCIENCE FOUNDATION, grant No. 476/11.

REFERENCES

[1] E. B. Swanson, "The dimensions of maintenance," in *Proceedings of the 2nd international conference on Software engineering*. IEEE Computer Society Press, 1976, pp. 492–497.

[2] A. Mockus and L. G. Votta, "Identifying reasons for software changes using historic databases," in *Software Maintenance, 2000. Proceedings. International Conference on*. IEEE, 2000, pp. 120–130.

[3] W. Meyers, "Interview with wilma osborne," *IEEE Software*, vol. 5, no. 3, pp. 104–105, 1988.

[4] B. P. Lientz, E. B. Swanson, and G. E. Tompkins, "Characteristics of application software maintenance," *Communications of the ACM*, vol. 21, no. 6, pp. 466–471, 1978.

[5] "Muse envisions mining big code to improve software reliability and construction," http://www.darpa.mil/news-events/2014-03-06a, [Online; accessed 18-April-2016].

[6] F. X. Diebold, "On the origin (s) and development of the term Big Data," PIER Working Paper, 2012.

[7] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from big code," in *ACM SIGPLAN Notices*, vol. 50, no. 1. ACM, 2015, pp. 111–124.

[8] S. R. Schach, B. Jin, L. Yu, G. Z. Heller, and J. Offutt, "Determining the distribution of maintenance categories: Survey versus measurement," *Empirical Software Engineering*, vol. 8, no. 4, pp. 351–365, 2003.

[9] R. E. Johnson and B. Foote, "Designing reusable classes," *Journal of object-oriented programming*, vol. 1, no. 2, pp. 22–35, 1988.

[10] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 78–88.

[11] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *Software Engineering, IEEE Transactions on*, vol. 31, no. 6, pp. 429–445, 2005.

[12] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, pp. 10–10, 2010.

[13] V. R. Basili, "Software modeling and measurement: the goal/question/-metric paradigm," 1992.

[14] B. Fluri and H. C. Gall, "Classifying change types for qualifying change couplings," in *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*. IEEE, 2006, pp. 35–45.

[15] H. C. Gall, B. Fluri, and M. Pinzger, "Change analysis with evolizer and changedistiller," *IEEE Software*, vol. 26, no. 1, p. 26, 2009.

[16] "About the search api," https://developer.github.com/v3/search/, [Online; accessed 11-April-2016].

[17] B. Fluri, E. Giger, and H. C. Gall, "Discovering patterns of change types," in *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*. IEEE, 2008, pp. 463–466.

[18] M. Martinez, L. Duchien, and M. Monperrus, "Automatically extracting instances of code change patterns with ast analysis," *arXiv preprint arXiv:1309.3730*, 2013.

[19] M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," in *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*. IEEE, 2003, pp. 23–32.

[20] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *ACM sigsoft software engineering notes*, vol. 30, no. 4. ACM, 2005, pp. 1–5.

[21] P. McCullagh and J. A. Nelder, *Generalized linear models*. CRC press, 1989, vol. 37.

[22] R Development Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2008, ISBN 3-900051-07-0. [Online]. Available: http://www.R-project.org

[23] E. Shihab, "An exploration of challenges limiting pragmatic software defect prediction," Ph.D. dissertation, Citeseer, 2012.

[24] A. E. Camargo Cruz and K. Ochimizu, "Towards logistic regression models for predicting fault-prone code across software projects," in *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*. IEEE Computer Society, 2009, pp. 460–463.

[25] N. Gorla and Y. W. Lam, "Who should work with whom?: building effective software project teams," *Communications of the ACM*, vol. 47, no. 6, pp. 79–82, 2004.

[26] P. J. Guinan, J. G. Cooprider, and S. Faraj, "Enabling software development team performance during requirements definition: A behavioral versus technical approach," *Information Systems Research*, vol. 9, no. 2, pp. 101–125, 1998.

[27] "Github - the largest open source community in the world," https://github.com/open-source, [Online; accessed 18-April-2016].

[28] T. J. McCabe, "A complexity measure," *Software Engineering, IEEE Transactions on*, no. 4, pp. 308–320, 1976.

[29] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *Software Engineering, IEEE Transactions on*, vol. 33, no. 1, pp. 2–13, 2007.

[30] R. M. Bell, T. J. Ostrand, and E. J. Weyuker, "Does measuring code change improve fault prediction?" in *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*. ACM, 2011, p. 2.

[31] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*. IEEE, 2005, pp. 284–292.

[32] T. J. Ostrand and E. J. Weyuker, "Predicting bugs in large industrial software systems." in *ISSSE*. Springer, 2011, pp. 71–93.

[33] R. M. Bell, T. J. Ostrand, and E. J. Weyuker, "The limited impact of individual developer data on software defect prediction," *Empirical Software Engineering*, vol. 18, no. 3, pp. 478–505, 2013.

[34] S. Matsumoto, Y. Kamei, A. Monden, K.-i. Matsumoto, and M. Nakamura, "An analysis of developer metrics for fault prediction," in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*. ACM, 2010, p. 18.

[35] E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang, "An industrial study on the risk of software changes," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 62.

[36] T. Jiang, L. Tan, and S. Kim, "Personalized defect prediction," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 2013, pp. 279–289.

[37] "Stackoverflow - a language-independent collaboratively edited question and answer site for programmers," http://stackoverflow.com/, [Online; accessed 11-April-2016].