Practical Software Quality Prediction

Emad Shihab
Department of Computer Science and Software Engineering
Concordia University
eshihab@cse.concordia.ca

***Abstract*—Software systems continue to play an increasingly important role in our daily lives, making the quality of software systems an extremely important issue. Therefore, a signicant amount of recent research focused on the prioritization of software quality assurance efforts. One line of work that has been receiving an increasing amount of attention is Software Defect Prediction (SDP), where predictions are made to determine where future defects might appear. Our survey showed that in the past decade, more than 100 papers were published on SDP. Nevertheless, the practical adoption of SDP to date is limited.**

In this paper, I highlight the findings of my thesis, which identifies the challenges that hinder the adoption of SDP in practice. These challenges include the fact that the majority of SDP research rarely considers the impact of defects when performing their predictions, seldom provides guidance on how to use the SDP results, and is too reactive and defect-centric in nature. Therefore, I propose approaches that tackle these challenges. First, I present approaches that predict high-impact defects. Our approaches illustrate how SDP research can be tailored to consider the impact of defects when making their predictions. Second, I present approaches that simplify SDP models so they can be easily understood and illustrate how these simple models can be used to assist practitioners in prioritizing the creation of unit tests in large software systems. These approaches show how SDP research can provide guidance to practitioners using SDP. Then, I argue that organizations are interested in proactive risk management, which covers more than just defects. For example, risky changes may not introduce defects but they could delay the release of projects. Therefore, I present an approach that predicts risky changes, illustrating how SDP can be more encompassing (i.e., by predicting risk, not only defects) and proactive (i.e., by predicting changes before they are incorporated into the code base). Finally, I present a number of avenues for future research and discuss several lessons learned during the PhD degree process.

I. Introduction

Software systems are becoming increasingly complex and are being used in everything from mobile devices to space shuttles. The increasing importance and complexity of software systems in our daily lives makes their quality a critical, yet extremely difficult issue to address. The US National Institute of Standards and Technology (NIST) estimated that software faults and failures cost the US economy $59.5 billion a year [1]. Other studies show that an average Fortune 100 company maintains 35 million lines of code and that this amount of maintained code is expected to double every 7 years [19]. Software Quality Assurance (SQA), i.e., the set of activities that ensure software meets a specific quality level, is one area that takes up a large amount of this maintenance effort [9].

Therefore, a significant amount of recent research has focused on the prioritization of SQA efforts. One line of work that has been receiving increasing amounts of attention recently is Software Defect Prediction (SDP), where code and/or repository data (i.e., recorded data about the development process) is used to predict where defects might appear in the future (e.g., [18], [36]). In fact my literature review [25] shows that in the past decade more than 100 papers were published on SDP alone.

Nevertheless, the adoption of software engineering research, especially SDP, in practice has been a challenge [6], [9], [24]. My thesis hypothesized that the limited adoption of SDP is attributed to the fact that most SDP studies are not designed with a pragmatic view in mind [16], [31]. This hypothesis is supported through prior work and numerous discussions with software engineering practitioners from a large software company where I spent 1.5 years as a SQA specialist and one year as an embedded SQA researcher. Based on an extensive literature review of SDP research in the past decade, I observe that the following challenges play a key role in the limited adoption of SDP in practice:

1) **Rarely consider the impact of defects**: SDP research rarely considers the impact of defects when providing recommendations of software locations that should be addressed [4], [17]. This makes the SDP approaches less effective since, for example, a documentation defects tend to have far less impact than security defects.
2) **Seldom provide guidance for use of results in practice**: Very few SDP studies focus on what to do once the predictions are made. Practitioners are left with no guidance on how to make use of SDP results [9], [16].

In addition, the majority of SDP approaches are reactive in nature and only focus on predicting defects, i.e., they assume that defects are already in the code and flag code that these defects might exist in. However, organizations are interested in managing risk, which covers more than just defects. For example, risky changes may not introduce defects but they could delay the release of projects, and/or negatively impact customer satisfaction. At the same time, it would be ideal to proactively flag risky code and address it before is injected into the code base. I believe that proactive approaches that predict risky changes are needed [12].

To improve the adoption of SDP in practice, in my thesis, I propose approaches that demonstrate how prior SDP research can be tailored to deal with the aforementioned challenges

(i.e., considering the impact of defects and providing guidance on how to use the results), making SDP more pragmatic. The thesis contains three main parts, two parts focus on each of the two aforementioned challenges. The third part proposes an approach that demonstrates how SDP research can be more encompassing and proactive. Furthermore, I detail avenues for future work that can improve the adoption of SDP in practice. Finally, I discuss some insights on what did and did not work during the PhD process.

The rest of the paper is organized as follows. Section II presents the thesis contributions. Section III summarizes the contributions of the thesis. Section IV identifies several avenues for future work. Section V shares some insights on the PhD process.

II. Thesis Contributions

The main contributions of the thesis are divided into three parts. Each part focuses on tackling a specific challenge of SDP. It is important to note that when evaluating the different approaches presented in my thesis, I follow an empirical approach which requires access to historical data. Ideally, I would like to evaluate each approach on both, data from commercial and data from open source systems. However, this can be extremely difficult since some systems (e.g., commercial projects) have specific data that open source systems might not have. On the other hand, open source systems are more likely to share their data, whereas, data for commercial systems can be difficult to obtain (for confidentiality reasons). Therefore, I did my best to evaluate each approach presented in the thesis on as many projects as possible, however, some approaches are evaluated only on commercial systems while other approaches are evaluated only on open source systems.

Before presenting the thesis contributions, it is important to note that the thesis provides background on SDP and surveys the state-of-the-art in SDP. However, due to space limitations, this survey is omitted and interested readers are encouraged to read the survey in [25], Chapter 2.

A. Part 1: Considering Impact of Defects

A large body of prior work focuses on predicting post-release defects in open source and commercial systems [5], [18], [21], [35], [36]. One of the main reasons for the limited adoption of prior SDP research in practice is that even though they show promising accuracy results, all defects are considered to have the same negative impact. This is not realistic, because, for example, documentation defects tend to have far less impact than security defects. Therefore, I believe that there is a need for SDP approaches to consider impact when making their predictions [4], [17]. In this part, I present approaches that focuses on predicting the highest-impacting defects. I consider three possible definitions of high-impact defects: breakages (defects that occur in functionality that customers are used to), surprises (defects that occur in locations where practitioners did not expect) and re-opened defects (defects that have to be fixed more than once). My work illustrates how SDP approaches can be tailored to consider the impact of defects.

1) Studying and Predicting Breakage and Surprise Defects: The relationship between various software-related phenomena (e.g., code complexity) and post-release software defects has been thoroughly examined [5], [18], [21], [35], [36]. However, to date these predictions have limited adoption in practice. The most commonly cited reason is that the prediction identifies too much code to review without distinguishing the impact of these defects. In this chapter, I aim to address this challenge by focusing on *high-impact defects* for customers and practitioners. *Customers* are highly impacted by defects that break pre-existing functionality (breakage defects), whereas *practitioners* are caught off-guard by defects in files that had relatively few pre-release changes (surprise defects) [32].

I perform an empirical study on a large commercial software system to study and predict high-impact defects. I mine the project's repositories and extract a number of factors related to code and process, factors related to co-changes and factors related to time pressures. I present models that can effectively identify files containing breakage and surprise defects. In addition, I perform analysis to identify and quantify the effect of the various factors on the likelihood of a file containing a breakage or surprise defect.

Our study addresses a number of research questions such as *Can we effectively predict which files will have breakage/surprise defects?*, *Which factors are important for the breakage/surprise defect prediction models?* and *How much effort savings do specialized models that focus on breakage/surprise defect provide over state-of-the-art models?*. The main recommendations based on the findings of the work on breakage and surprise defects are:

- Practitioners need to consider both, breakage and surprise defects, separately since they are rare, unique and different. Surprise defects have high severity and indicate problems in the requirements.
- Using specialized defect prediction models can effectively predict breakage and surprise defects, yielding sizeable effort savings (3.3% for breakages and 30% for surprise defects.) over using simple post-release defect prediction models.
- Traditional defect prediction factors (i.e., the number of pre-release defects and file size) are good predictors of breakage defects. However, the number of co-changed files, the size of recently co-changed files and the time since the last change should be used to predict surprise defects.

2) Studying and Predicting Re-opened Defects: Defect fixing accounts for a large amount of the software maintenance resources. Generally, defects are reported, fixed, verified and closed. However, in some cases defects have to be re-opened. Re-opened defects increase maintenance costs, degrade the overall user-perceived quality of the software and lead to unnecessary rework by practitioners.

Therefore, I study and predict re-opened defects through a case study on three large open source projects – namely Eclipse, Apache and OpenOffice. I build prediction models that effectively predict re-opened defects [23], [27]. Then, I

analyze the prediction models to determine which factors are the most important indicators of whether or not a defect will be re-opened. In particular, I ask two main research questions: *Which factors indicate, with high probability, that a defect will be re-opened?* and *Can we accurately predict whether a defect will be re-opened using the extracted factors?*. The main recommendations based on the findings of the work on re-opened defects are:

- The occurrence of re-opened defects should be minimized since they take considerably longer to resolve.
- Practitioners can leverage decision tree prediction models to accurately predict re-opened defects. Predicting re-opened defects in three different projects, I was able to achieve a precision between 49.9-78.3% and a recall in the range of 72.6-93.5%.
- The factors that best indicate re-opened bugs vary based on the project. The comment text is the most important factor for the Eclipse and OpenOffice projects, while the last status is the most important one for Apache. All of these factors can be extracted from the bug reports.

B. Part 2: Making Use of SDP Results

Most SDP research today provides black-box type of models, i.e., a list of defect-prone software locations is given without any explanation as to why. This makes it difficult to understand why these models are making their predictions. To make prediction models easier to understand, I present an approach that simplifies prediction models. In addition, I present an approach to prioritize the creation of unit tests in large software systems (i.e., which parts of the code we should write unit tests for) to show how SDP results can be applied in practice (i.e., applying SDP to determine the most defect-prone functions so they can have unit tests created for them).

1) Simplifying and Understanding SDP Models: Research studying the quality of software applications continues to grow rapidly with researchers building regression models that combine a large number of factors. However, these prediction models are hard to deploy in practice due to the cost associated with collecting all the needed factors, the complexity of the models and the black box nature of the models. For example, techniques such as Principle Component Analysis (PCA) are commonly used to merge a large number of factors into composite factors that are no longer easy to explain.

I use a statistical approach recently proposed by Cataldo *et al.* to create and operationalize explainable regression models [30]. In addition, I show that the approach is able to quantify the impact of the used factors in a prediction model on the likelihood of finding post-release defects. Finally, I demonstrate that the simple models achieve comparable performance over more complex PCA-based models while providing practitioners with intuitive explanations on how to make use of the results. The main recommendations based on the findings of the work on simplifying and understanding SDP models are:

- Practitioners should use a small number of metrics in their prediction models since it makes the prediction models simple and easy to understand.
- Using a small number of metrics can achieve prediction and explanative powers similar to more complex models. On a case study using the Eclipse project, using 3 or 4 metrics achieves precision and recall values that are comparable to more complex models that use 34 metrics.

2) Prioritizing the Creation of Unit Tests: One major challenge of SDP research is that it does not provide any guidance on how to make use of their results. I believe that this challenge is due to the fact that this SDP work is not designed with a specific scenario in mind.

I use factors extracted from the development history of software projects to build simple SDP models that prioritize the creation of unit tests [28], [29]. The approach is different from traditional SDP studies in that it performs its predictions at the function level and it takes into consideration the effort required to create the unit tests. This approach illustrates how software development and testing managers can leverage SDP to efficiently allocation their limited SQA resources. The main recommendations based on the findings of the work on the prioritization and creation of unit tests are:

- Indeed, practitioners can leverage the history of a project to effectively prioritize the creation of unit tests for their large software projects.
- Using historical data can achieve a three-fold improvement over a naive strategy that randomly selects functions to write unit tests for.
- Performing a case study on a large commercial and open source project, we find that the size of a function should be used to prioritize the creation of unit tests.

C. Part 3: Making SDP Approaches More Encompassing and Proactive

The majority of SDP research focuses on predicting defects and is reactive in nature, i.e., it assumes that the defects already exist in the code, and aim to identify the code that contains these defects [2], [3], [7], [8], [10], [11], [13], [14], [18], [20], [22], [33], [34], [36]. However, I believe that organizations are interested in more than just defects, they are interested in managing risk. Risk is more encompassing than defects. In this part, I present an approach to identify risky code changes, i.e., changes that require additional attention through careful code/design review and possibly more testing. The work illustrates how SDP approaches can be more encompassing and proactive

1) Studying and Predicting the Risk of Software Changes: Modelling and understanding defects has been the focus of much of the Software Engineering research today. However, organizations are interested in more than just defects. In particular, they are more concerned about managing risk, i.e., the likelihood that a code or design change will cause a negative impact on their products and processes, regardless of whether or not it introduces a defect.

I conduct a study to predict and better understand risky changes, i.e., changes for which developers believe that additional attention is needed in the form of careful code/design

reviewing and/or more testing [26]. The findings and models are being used today by an industrial partner to manage the risk of their software projects. The main recommendations based on the findings of the work on risky software changes are:

- The developer making the change and the team they belong to need to be considered when studying the risk of a software change.
- Developers are accurate 96.1% of the time when identifying changes that introduce defects. However, developers' identification of risky changes is less reliable when changes have many related changes.
- Practitioners should use factors such as the number of lines and chunks added by the changes, the bugginess of the files being changed, the number of bug reports linked to the change and the experience of the developer making the change to identify risky changes.

III. Summary of Thesis Contributions

To summarize, the major contributions of the thesis are listed below. The details regarding each contribution can be found in the respective publication cited above or the thesis document [25].

- An extensive review of the state of the art in SDP. Such a review is of paramount importance at this time since a large amount (more than 100 papers according to my review) of research related to SDP has been done in the last decade, making it a good time to reflect on what has been addressed and what remains as an open issue in the field today. This survey provides an empirical foundation and motivation for the work in my thesis.
- The development of an approach to predict and better understand high-impact defects. This approach can be followed by other researchers to tailor their SDP approaches so they can focus on high-impact defects. I believe that such an approach is more applicable than the state-of-the-art in SDP today, since impact is taken into consideration when making predictions.
- The development of an approach to simplify SDP models by reducing the number of used factors. This approach shows how SDP research can be simplified, making it easier to understand and use. The presented results show that my approach can significantly simplify SDP models and that these simple models are able to achieve comparable performance to models that are much more complex.
- The development of an approach to guide the prioritization of unit test creation. This approach shows how simple SDP models can be used to prioritize the creation of unit tests. The presented results show that my approach outperforms ad-hoc methods used by practitioners today.
- The proposal of an approach that makes SDP more encompassing and proactive. The approach identifies potentially risky code changes before they are incorporated into the code base.

IV. Future Work

I believe that my thesis makes a positive contribution towards the goal of making SDP research more pragmatic. However, there are still many open challenges that need to be tackled in order to increase the adoption of SDP in practice. I now highlight some avenues for future work.

A. Formally Investigating Reasons for Lack of SDP Adoption in Practice

In my thesis, I relied on my experience when deciding some of the challenges that hinder the adoption of SDP in practice. The reasons given in my thesis are by no means complete. I encourage future research to conduct more detailed and formal studies regarding the reasons that hinder the adoption of SDP in practice.

B. Considering Other Types of High-Impact Defects

In my thesis, I focused on three different types of high-impact software defects. I believe that this is a good start, however, there remains more work to do in this area. Different types of high-impact defects need to be examined. For example, another type of defect that might have a high impact is defects in software artifacts that many other artifacts depend on. I encourage future research to continue this line of work and study and predict other types of high-impact defects.

C. Building Tools to Guide Practitioners

Today, most SDP research proposes solutions and empirically evaluates them based on historical data. This type of work has significantly contributed to the research side of software engineering, however, very little work actually builds tools based on their research to advance and applicability of SDP research in practice. I encourage future research to focus more on how to build tools so that our research can be easier to incorporate in industry.

D. More Realistic Evaluations

As shown in my survey [25], the vast majority of SDP studies evaluate their approaches using the precision and recall measures. However, as pointed out by other researchers [15] and from my own industrial experience, standard statistical measures of performance such as precision and recall might not be the best way to evaluate the practical value of SDP approaches. Whenever possible, I strived to obtain feedback from practitioners about the proposed approaches. I encourage future research to investigate and propose evaluation criteria that practitioners use to measure the value of SDP approaches. I believe that using such criteria will provide a more realistic evaluation of SDP approaches and significantly improve the adoption of SDP in practice.

E. Examining Replicability

The majority of SDP research heavily depends on historical development data. Till now, the availability of open source data has been relatively easy, however, acquiring commercial data is still a challenge. The fact that commercial data is not widely

available makes it difficult to examine the repeatability of SDP studies. Examining repeatability is important since it indicates how generalizable the finding are. I believe that the entire software engineering research community needs to address this issue of making data (especially commercial data) available in order to facilitate the repeatability of proposed approaches.

V. Lessons Learned During the PhD Process

In this section, I share some of the lessons learned during the PhD. I divide the section into two parts. First, I discuss things that worked well during the PhD and second, I discuss things that I wish I did (most of which are things I did not realize until I started my faculty position).

A. Things That Worked Well

I list a number of things that worked well for me during the PhD. Obviously, this is not a complete list, however, it contains most things that I found to work.

1) Working with Industry: I was fortunate enough to work closely with industry during the PhD, and the feedback they provided was invaluable. Especially in an applied field like Software Engineering, feedback from industry on our research can be a very frank reality check. Although those reality checks can lead to disappointing outcomes sometimes (e.g., when the perfect algorithm does not perform well in a realistic setting), it often ends up saving a ton of time and ends up making the research a lot more practical.

Another positive aspect about collaborating with industry is that industry is where the majority of real problems occur. Hence, I often found interesting research problems to work on just by working or talking to industry. One word of caution however, is that industry can also hinder your research plan. In many cases their goals are different than that of a researcher. For example, in many cases (but not all), industry is interested in tools or in solving problems that have been solved in the research community. Therefore, it is critical to set expectations early in order to avoid disappointments for both, the researcher and the industrial partner.

2) Performing Community Service: My advisor always encouraged his students to review papers, volunteer to help at workshops and conferences, etc. Early on, I knew that doing this community service was a positive thing, if not for us, at least for the community. However, I never knew how important it was for my career. First, serving on program committees allowed me to see, first hand, high quality and not so high quality work. Such observations immediately allowed me to reflect on my own work and realize how I can improve it so that it can be accepted. Second, in most conferences I served as PC on, the review phase is often followed by a discussion phase. Being able to objectively argue for or against a paper, in my humble opinion, earns you respect from others in the research community. Having this respect from your colleagues plays a critical role once you graduate and are writing your own papers and grant proposals. Lastly, I learned a ton by reviewing other work, which in turn helped me improve my own research. From my own experience, I highly recommend that a PhD student perform some community service.

3) Building Depth and Breadth: During the PhD (and I assume this holds for most PhD students), I built a strong and deep understanding in my research topic. However, I strongly believe that depth is not enough during a PhD. A PhD student absolutely need to build breadth as well. Having breadth becomes very important because you will need to make sure you can do research in more than your area of specialization, especially once you graduate and become an independent researcher. This helps in securing funding for my research program, as well as, makes me attractive to a larger number of graduate students.

4) Keeping an Agenda: At the beginning of the PhD, I found it very difficult to get anything done, especially since a PhD follows a learn-do-succeed/fail process. At some points in time, it seemed like I did not get anything done. One of the best things to keep track of my progress and (at the least) prove to myself that I was indeed making progress was to keep an agenda of what I did. Keeping an agenda at least doubled my productivity. It also helped me realize how much time I spent on some tasks so that when someone like my advisor says“maybe you should pursue another avenue to solve this problem”, I can justify to myself that indeed, I have been spending too much time pursuing something that is fruitless.

5) Teaching: Towards the end of the PhD, I knew that I wanted to pursue a faculty position. Although research is an important part of a professor’s job, teaching is of high importance as well. Therefore, at the end of the PhD, I asked to teach an undergraduate course. It was one of the most difficult things I did during the PhD. However, it made the first teaching experience as a professor so much more effective. Teaching as a PhD student helped me realize how much time and effort teaching requires and prepared me in figuring out the balance between teaching, research and service when I started as a faculty member. I highly recommend teaching towards the end of your PhD, especially if you plan on pursuing an academic career after your PhD.

B. Things I Wish I Did Before Graduating

On the other hand there are some tings that I only realized were important after I completed the PhD and started to work as a faculty member. Below are things I wish I was aware of during the PhD.

1) Writing Grants: It is probably no surprise, but funding is crucial for any successful research program. Unfortunately, writing grants to secure funding is something that PhD students are rarely exposed to. Therefore, something that I wish I could have gotten more experience with is writing grants, since it gives you a big advantage when you start your academic career. If you get a chance to be involved in any grant writing during your PhD, make sure you take advantage of it.

2) Recruiting Students: Another big question that faced me after starting the faculty position is - how can I recruit the best students? This is often another aspect that is critical to the success of a faculty position, which is never taught in the PhD program. If you ever have a chance to be involved in the recruitment of junior students, make sure you get involved.

Being able to (or even just getting familiar with) evaluate student applications and know how to determine good students is an invaluable skill to have as a faculty.

ACKNOWLEDGEMENTS

I thank my advisor Dr. Ahmed E. Hassan for his guidance, support and advice during the PhD process. Also, I thank all my collaborators, co-authors and colleagues for all their help. I thank the Natural Sciences and Engineering Research Council of Canada (NSERC) for funding parts of my research.

REFERENCES

[1] The economic impacts of inadequate infrastructure for software testing. http://www.nist.gov/director/planning/upload/report02-3.pdf.

[2] Erik Arisholm and Lionel C. Briand. Predicting fault-prone components in a java legacy system. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, ISESE '06, pages 8–17, 2006.

[3] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, 1996.

[4] Marco D'Ambros, Michele Lanza, and Romain Robbes. On the relationship between change coupling and software defects. In *Proc. Working Conference on Reverse Engineering (WCRE'09)*, pages 135–144, 2009.

[5] Khaled El Emam, Walcelio Melo, and Javam C. Machado. The prediction of faulty classes using object-oriented design metrics. *Journal of Systems and Software*, 56:63–75, February 2001.

[6] Michael W. Godfrey, Ahmed E. Hassan, James Herbsleb, Gail C. Murphy, Martin Robillard, Prem Devanbu, Audris Mockus, Dewayne E. Perry, and David Notkin. Future of mining software archives: A roundtable. *IEEE Software*, 26(1):67 –70, Jan.-Feb. 2009.

[7] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661, July 2000.

[8] Tibor Gyimothy, Rudolf Ferenc, and Istvan Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31:897–910, October 2005.

[9] A.E. Hassan. The road ahead for mining software repositories. In *Frontiers of Software Maintenance, 2008*, pages 48 –57, Oct. 2008.

[10] Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 78–88, 2009.

[11] Israel Herraiz, Jesus M. Gonzalez-Barahona, and Gregorio Robles. Towards a theoretical model for software growth. In *Proc. International Workshop on Mining Software Repositories (MSR'07)*, pages 21–28, 2007.

[12] Y. Kamei, E. Shihab, B. Adams, AE. Hassan, A Mockus, A Sinha, and N. Ubayashi. A large-scale empirical study of just-in-time quality assurance. *Software Engineering, IEEE Transactions on*, 39(6):757–773, June 2013.

[13] Taghi M. Khoshgoftaar, Edward B. Allen, Wendell D. Jones, and John P. Hudepohl. Data mining for predictors of software quality. *International Journal of Software Engineering and Knowledge Engineering*, 9(5):547–564, 1999.

[14] Marek Leszak, Dewayne E. Perry, and Dieter Stoll. Classification and evaluation of defects in a project retrospective. *Journal of Systems and Software*, 61:173–187, April 2002.

[15] Paul Luo Li, James Herbsleb, Mary Shaw, and Brian Robinson. Experiences and results from initiating field defect prediction and product test prioritization efforts at abb inc. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 413–422, 2006.

[16] Thilo Mende, Rainer Koschke, and Jan Peleska. On the utility of a defect prediction model during hw/sw integration testing: A retrospective case study. In *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering*, CSMR '11, pages 259–268, 2011.

[17] Audris Mockus, Ping Zhang, and Paul Luo Li. Predictors of customer perceived software quality. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 225–233, 2005.

[18] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 181–190, 2008.

[19] Hausi A. Müller, Scott R. Tilley, and Kenny Wong. Understanding software systems using reverse engineering technology perspectives from the rigi project. In *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '93, pages 217–226, 1993.

[20] Nachiappan Nagappan and Thomas Ball. Static analysis tools as early indicators of pre-release defect density. In *International Conference on Software Engineering(ICSE'05)*, pages 580–586, 2005.

[21] Nachiappan Nagappan and Thomas Ball. Using software dependencies and churn metrics to predict field failures: An empirical case study. In *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, ESEM '07, pages 364–373, 2007.

[22] Niclas Ohlsson and Hans Alberg. Predicting fault-prone software modules in telephone switches. *IEEE Transactions on Software Engineering*, 22(12):886–894, 1996.

[23] E. Shihab, A Ihara, Y. Kamei, W.M. Ibrahim, M. Ohira, B. Adams, AE. Hassan, and K.-i. Matsumoto. Predicting re-opened bugs: A case study on the eclipse project. In *Reverse Engineering (WCRE), 2010 17th Working Conference on*, pages 249–258, Oct 2010.

[24] Emad Shihab. Pragmatic prioritization of software quality assurance efforts. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1106–1109, 2011.

[25] Emad Shihab. *An Exploration of Challenges Limiting Pragmatic Software Defect Prediction*. Phd thesis, Queen's University, 2012.

[26] Emad Shihab, Ahmed E. Hassan, Bram Adams, and Zhen Ming 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*, FSE '12, pages 62:1–62:11, 2012.

[27] Emad Shihab, Akinori Ihara, Yasutaka Kamei, Walid M Ibrahim, Masao Ohira, Bram Adams, Ahmed E Hassan, and Ken-ichi Matsumoto. Studying re-opened bugs in open source software. *Emp. Software Engineering*, 18(5):1005–1042, 2013.

[28] Emad Shihab, Zhen Ming Jiang, Bram Adams, Ahmed E. Hassan, and Robert Bowerman. Prioritizing unit test creation for test-driven maintenance of legacy systems. In *Proc. International Conference on Quality Software (QSIC'10)*, QSIC '10, pages 132–141, 2010.

[29] Emad Shihab, Zhen Ming Jiang, Bram Adams, Ahmed E. Hassan, and Robert Bowerman. Prioritizing the creation of unit tests in legacy software systems. *Softw. Pract. Exper.*, 41(10):1027–1048, September 2011.

[30] Emad Shihab, Zhen Ming Jiang, Walid M. Ibrahim, Bram Adams, and Ahmed E. Hassan. Understanding the impact of code and process metrics on post-release defects: A case study on the eclipse project. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, pages 4:1–4:10, 2010.

[31] Emad Shihab, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. Is lines of code a good measure of effort in effort-aware models? *Information and Software Technology*, 55(11):1981–1993, 2013.

[32] Emad Shihab, Audris Mockus, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. High-impact defects: A study of breakage and surprise defects. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 300–310, 2011.

[33] Ramanath Subramanyam and M. S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Tran. on Software Engineering*, 29(4):297–310, 2003.

[34] T.-J. Yu, V. Y. Shen, and H. E. Dunsmore. An analysis of several software defect models. *IEEE Transactions on Software Engineering*, 14(9):1261–1270, 1988.

[35] Thomas Zimmermann and Nachiappan Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 531–540, 2008.

[36] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for Eclipse. In *PROMISE '07: Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, pages 1–7, 2007.