Pragmatic Prioritization of Software Quality Assurance Efforts

Emad Shihab
Software Analysis and Intelligence Lab (SAIL)
Queen's University, Kingston, ON, Canada
emads@cs.queensu.ca

ABSTRACT

A plethora of recent work leverages historical data to help practitioners better prioritize their software quality assurance efforts. However, the adoption of this prior work in practice remains low. In our work, we identify a set of challenges that need to be addressed to make previous work on quality assurance prioritization more pragmatic. We outline four guidelines that address these challenges to make prior work on software quality assurance more pragmatic: 1) Focused Granularity (i.e., small prioritization units), 2) Timely Feedback (i.e., results can be acted on in a timely fashion), 3) Estimate Effort (i.e., estimate the time it will take to complete tasks), and 4) Evaluate Generality (i.e., evaluate findings across multiple projects and multiple domains). We present two approaches, at the code and change level, that demonstrate how prior approaches can be more pragmatic.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Software Quality Assurance

Keywords

Unit Testing, Software Metrics, Change Risk

1. INTRODUCTION

Prior studies show that more than 90% of the software development cost is spent on maintenance and evolution activities [2, 11]. Other studies showed 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 [10]. Software Quality Assurance (SQA) is one area that takes up a large amount of this maintenance effort [5]. Therefore, practitioners are in dire need of pragmatic approaches to assist them effectively prioritize SQA efforts.

ICSE '11, May 21–28, 2011, Waikiki, Honolulu, HI, USA

A significant amount of recent research has focused on the prioritization of SQA efforts, fueled by the formation of the Mining Software Repositories (MSR) field and the wide use of software repository data. The majority of this work focuses on code level quality (e.g., predicting buggy software locations), while other work explores change level quality (e.g., predicting buggy changes). However, the adoption of these approaches into practice remains low [3, 5]. This low adoption may be attributed to many reasons. For example, most defect prediction work provides predictions at the file level, a granularity deemed too coarse for practical use [5].

In this work, we study the prioritization of SQA efforts at the code and change levels from a pragmatic point of view in order to provide software practitioners with pragmatic approaches that help them in the prioritization of SQA efforts. Our approache is built on four guidelines that practitioners demand from SQA approaches to significantly increase their chance of adoption in practice. They are:

1. **Focused Granularity**: the unit of prioritization must be small enough (i.e., function/method level) such that tasks can be easily assigned [5],
2. **Timely Feedback**: must be able to act on the feedback/results of the approach in a timely fashion to maximize the usefulness of the feedback [7],
3. **Estimate Effort**: practitioners should be aware of the amount of effort needed for the task at hand to better allocate resources [9],
4. **Evaluate Generality**: the approaches must be evaluated on multiple projects that cover multiple domains (i.e., open source and commercial) to clearly understand the implications of the findings [5].

In addition to the mention of these guidelines in previous research work, these guidelines are the outcome of numerous discussions with software engineering practitioners from a large software company where I spent 1.5 years as a SQA specialist and 1 year as an embedded SQA researcher.

The code level and change level work complement each other. **Code level** quality assurance will assure a high quality of already implemented code (e.g., evolving and legacy code). On the other hand, **change level** quality assurance takes a more pro-active approach to assure a high level of quality for new code changes (e.g., new features).

We show how we can extend and enhance prior work to consider these pragmatic guidelines. For example, at the code level we provide an approach to help practitioners prioritize the creation of unit tests (i.e., we tell them which functions to create unit tests for first). At the change level, we propose a pragmatic approach that flags risky code changes for closer examination (e.g., code review).

The rest of the paper is organized as follows. Section 2 provides a motivating example for the work. Section 3 highlights our research hypothesis. Section 4 discusses the current state of the work. Section 5 examines related work. Section 6 concludes the paper.

2. MOTIVATING EXAMPLE

Lisa manages a large software team that consists of a number of developers and testers. Lisa wants to improve the overall quality of the software produced by her team.

The testing team focuses on testing already-implemented code (e.g., legacy code). To improve the effectiveness of their testing in finding bugs, Lisa wants to be able to prioritize for which functions to write unit tests. Since this already-implemented code has a rich set of data in the software repositories, Lisa can leverage this data to know which functions to write unit tests for.

Lisa needs an approach that analyzes her historical data and provides her with a list of functions (*Focused Granularity*) to write unit tests for. Lisa must be able to act on the results of this analysis in a timely fashion (*Timely Feedback*), so that tests can be created quickly. Furthermore, Lisa wants to make sure she knows how much time it will take to write the unit tests (*Estimate Effort*). Lastly, Lisa would like to explore the generality of the approach (*Evaluate Generality*) to better understand how to apply the approach in different projects that her team manages (e.g., can a model created using one project be used in other projects).

On the other hand, the development team makes code changes that either implement new features or perform general maintenance (e.g., bug fixes). Lisa wants to be more pro-active and find problems before they even get into the code. She wants to prioritize the review of the riskiest changes.

In this case, Lisa needs an approach that can determine the overall risk of a particular change by analyzing the history of the code being changed. For each change, the functions/methods (*Focused Granularity*) that were changed are shown, so her team knows exactly what to review. Since the analysis is being done at the change level, the results of the approach need to be returned as soon as a change happens, such that developers can act while the changes are still fresh in their mind (*Timely Feedback*). In order to balance the reviewing work with the development work, the effort required to review the change must be provided (*Estimate Effort*). Finally, Lisa would like to explore the generality of the metrics used to determine the risk of the change, to know whether these metrics are project specific or if they can work on different projects (*Evaluate Generality*).

3. RESEARCH HYPOTHESIS

Software repositories, such as source control and bug repositories, are commonly used in practice for record keeping purposes. However, the information in these repositories can be mined and analyzed to assist in future decision making [5]. A large amount of prior work in the area of Empirical Software Engineering and Mining Software Repositories leverages this data to validate various hypotheses and build useful techniques. For example, Zimmermann *et al.* [18] leverage the data stored in source control and bug repositories to identify buggy files in the Eclipse Open Source project.

In this thesis, we leverage historical data stored in software repositories to propose two approaches - one at the code level and one at the change level. These approaches differ from previous work in that they are designed to be pragmatic (according to our aforementioned guidelines). Our underlying research hypothesis is:

> *Pragmatic SQA prioritization approaches are needed in practice. These approaches need to provide recommendations at a fine-granularity, offer results that can be acted on in a timely manner, provide an estimate of the effort required to complete the task and be examined across various projects.*

4. CURRENT STATE OF THE WORK

In this section, we report the current state of the work. We have completed the majority of the code level quality work that is related to the prioritization of unit test creation. We are currently working on the change level quality work.

Code Level: Prioritization of Unit Test Creation

Motivation: To improve the quality of their software, practitioners of large software systems test their systems extensively. This testing is done in multiple forms, for example, by performing functional testing, unit testing and performance testing. In this work, we focus on the problem of prioritization of unit test creation (i.e., which parts of the code we should write unit tests for).

This problem is a very important one, since in most large software systems, it is not practically feasible to write unit tests for all of the code base. For example, if a team has enough resources to write unit tests to assess the quality of 100 lines of code per day, it would take them 27 years to write unit tests for a 1 million lines of code (LOC) system [14].

Approach: We present an approach that uses the history of a project to prioritize the writing of unit tests. The approach uses heuristics to recommend a prioritized list of functions to write unit tests for. Our approach is pragmatic, in that 1) provides results with a fine-grained granularity (i.e., at the function/method level), 2) takes into consideration the effort required to write the unit test when recommending functions to create unit tests for, 3) provides a list of functions to a practitioner can act on immediately and 4) our results were verified on a large open source project (i.e., Eclipse) and a large commercial system.

In a nutshell, the approach extracts a project's historical knowledge, calculates various heuristics (listed below) and recommends a prioritized list of functions to write unit tests for.

Extracting Historical Data: We combine source code change information from the source code control system with bug data stored in the bug tracking system and map the code changes to the actual source code functions that changed (*Focused Granularity*). We are able to achieve this focused granularity by extracting the source code of the changed files, comparing them to their previous version and mapping the changed functions to the change itself.

Calculating Heuristics: We use the extracted historical data to calculate various heuristics that are used to generate the prioritized list of functions for testing. We choose to use heuristics that can be extracted from a project's history for two main reasons: 1) evolving software systems have a rich history that we can use to our advantage and 2) previous work in fault prediction showed that history-based heuristics are good indicators of future bugs (e.g., [1, 13]).

The used heuristics are as follows:

- Most Freq. Modified (MFM): Functions that were modified the most since the start of the project.
- Most Recently Modified (MRM): Functions that were most recently modified.
- Most Freq. Fixed (MFF): Functions that were fixed the most since the start of the project.
- Most Recently Fixed (MRF): Functions that were most recently fixed.

- Largest Modified (LM): The largest modified functions, in terms of total lines of code.
- Largest Fixed (LF): The largest fixed functions, in terms of total lines of code.
- Size Risk (SR): Riskiest functions, defined as the number of bug fixing changes divided by the size of the function in lines of code.
- Change Risk (CR): Riskiest functions, defined as the number of bug fixing changes divided by the total number of changes.
- Random: Randomly selects functions to write unit tests for.

Generating a List of Functions: Following the heuristic calculation phase, we use the heuristics to generate a prioritized list of functions for which unit tests should be written for first. Each heuristic generates a different prioritized list of functions. For example, one of the used heuristics (i.e., MFM) recommends that we write tests for functions that have been modified the most since the beginning of the project. Another heuristic recommends that we write tests for functions that are fixed the most (i.e., MFF). Once the initial data processing is done, the lists can be generated immediately. As soon as the practitioner would like to write unit tests, he/she can generate a list on demand and start to write unit tests for functions immediately, i.e., no need to wait for further data, such as static analysis or runtime information (*Timely Feedback*).

The size of a function, in lines of code, is provided as an indicator of the amount of effort needed to write the unit test for a function (*Estimate Effort*). Previous work also used size as an indicator of effort [9].

Result: To figure out which heuristic we should use to effectively prioritize the creation of unit tests, we measure the performance of the different heuristics using two measures: Usefulness and Percentage of Optimal Performance (POP). Due to space limitations, we only present the results for usefulness. More details on POP are available in [14].

Usefulness: The first question that comes up after we write unit tests for a set of functions is - was writing the tests for these functions worth the effort? For example, if we write unit tests for functions that have no bugs after the tests are written, then our effort may be wasted. Ideally, we would like to write unit tests for functions that have the highest risk of having a bug in the future.

We define the usefulness metric as the percentage of functions for which we write unit tests that end up having at least one bug after the tests are written. The usefulness metric indicates how much of our effort on writing unit tests is actually worth the effort.

The median usefulness values for each of the heuristics in the commercial system and the Eclipse OSS system are listed in Table 1 (*Evaluate Generality*). The last row of the table shows the usefulness achieved by the random heuristic. The heuristics are ranked from 1 to 9, with 1 indicating the best performing heuristic and 9 the worst.

Our findings show that the LF, LM, MFF and MFM heuristics should be used to prioritize the creation of unit tests. The approach was designed in conjunction with a development team from an industrial partner that praised it for its pragmatic promise.

Table 1: Median Usefulness Results of the Commercial (Comm) and Open Source (OSS) Systems

Heuristic	Med. Usefulness		Imp. over rand.		Rank	
	Comm (%)	OSS (%)	Comm	OSS	Comm	OSS
LF	87.0	44.7	3.1X	5.3X	1	1
LM	84.7	32.9	3.1X	3.9X	2	2
MFF	83.8	32.3	3.0X	3.8X	3	3
MFM	80.0	28.1	2.9X	3.3X	4	4
MRF	56.9	16.0	2.1X	1.9X	5	6
CR	55.0	17.4	2.0X	2.0X	6	5
SR	48.8	12.6	1.8X	1.5X	7	7
MRM	43.1	9.9	1.6X	1.2X	8	8
Random	27.7	8.5	-	-	9	9

Change Level: Change Risk Analysis

We are currently working on the second part of the thesis, which is concerned with change level quality. This approach focuses on flagging risky code changes for further examination.

Motivation: Practitioners are interested in taking pro-active that catch bugs before they are injected into the code base. This can be done by closely identifying the risky changes (i.e., changes that have a high chance of introducing a bug) and rigourously reviewing them before they are incorporated into the code base. Since it is practically infeasible to review every change, having an approach that can estimate the risk of the changes and prioritize them according to their risk would benefit these practitioners in knowing which changes to focus on first.

Proposed Approach: We plan to extract data about a given change, in addition to using historical data about the changed functions or methods, to determine a particular risk of the change. As of now, we plan to calculate various metrics such as: 1) the amount of churn in each function due to the change, 2) the spread of the change (i.e., how many folders a change spans), 3) the number of total changes and bug-fixing changes done to the changed functions previously, and 4) the number of different developers that previously touched the changed functions and other metrics. For example, if a change has high churn in a function that is known to be buggy in the past, then it probably should be flagged as being risky and carefully reviewed.

To know which metrics have an impact on the risk of a change, we plan to extract the various metrics for changes that introduced bugs (identified using the SZZ algorithm [16]) and compare these metrics values to the metric values from a clean set of changes (training data set). Then we plan to build prediction models, using logistic regression, that use the metrics to predict whether or not a change will introduce a bug (on a separate testing data set). These models will tell us which metrics are good predictors of risky changes. We used a similar approach in our previous work [15] to narrow down a large set of process and product metrics into a few that had the greatest impact on the performance of the prediction models.

Using a similar technique to our code level work in [14], we will be able to map the change down to the changed functions (*Focused Granularity*). This way, practitioners will be able to know the functions or methods that they should closely review. In addition, practitioners will be able to generate the list of risky changes on demand and address the risky changes while they are fresh in the developers' minds (*Timely Feedback*).

The approach will use the number of different functions/methods, the size of these methods and their spread across different subsystems (i.e., folders) as an indicator of the effort required to review the changes (*Estimate Effort*). This effort can be leveraged by the practitioners to prioritize changes, not only on their risk level, but also on the amount of effort that their review requires.

To study the generality of our findings, we plan to perform this study on many projects, from both the open source and commercial domains (*Evaluate Generality*). We have partnered with an industrial partner who has provided us access to their source control and bug repositories and we are in the final stages of the data collec-

tion phase. As for open source projects, we are in the process of collecting the publicly available data.

Proposed Results: The outcome of the work will be an approach that can be used to determine risky changes. This approach will also provide an explanation for the assigned risk value of the change (e.g., too much churn). The approach will help practitioners take a pro-active role in assuring that a high level of software quality is achieved.

5. RELATED WORK

Code Level: Work by Arisholm *et al.* [1], Graves *et al.* [4] and Yu *et al.* [17] has shown that historical heuristics (e.g., prior changes and bug fixes) are good indicators of future bugs.

The work closest to our work used the idea of building a cache that recommends buggy code. Hassan and Holt [6] used change and fault metrics to generate a Top Ten List of subsystems (i.e., folders) that managers need to focus their testing resources on. Kim *et al.* [8] extended the work in [6] and used the idea of a cache that keeps track of locations that were recently added, recently changed and where faults were fixed to predict where future faults may occur. They performed their prediction at two levels of granularity: file- and method/function-level.

Our work prioritizes functions at a finer granularity than most previous work on fault prediction (except for *Kim et al.'s* approach [8]). Instead of identifying buggy files or subsystems, we identify buggy functions. Furthermore, our work considers the effort required to write the unit tests for the function/method and provides feedback that can be acted on within a short time frame. Lastly, we validated our study on two large projects from the commercial and open source domains. Our work puts forward a pragmatic approach to assist in the prioritization of unit test writing, given the knowledge about the history of the functions.

Change Level: The work that most closely relates to our proposed work on change risk is the prior work by Mockus and Weiss [12] and Kim *et al.* [7]. Mockus and Weiss [12] assess the risk of Initial Modification Requests (IMR) on the 5ESS commercial project. IMRs consist of multiple Modification Requests (MR), which are made up of multiple changes. Our work plans to complement the previous work in [12] by providing recommendations at a finer granularity (i.e., at the change level). Furthermore, we plan to consider the effort required to review the change. We plan to validate our results on commercial and open source projects.

Kim *et al.* [7] uses features to classify changes as being buggy or clean. We plan to complement the work by Kim *et al.* by providing a measure of the effort required to review a change, by providing tool support and by validating our results on commercial and open source projects. Overall, these enhancements will make effort prioritization more pragmatic.

Table 2 summarizes the comparison to the closest related work.

Table 2: Comparison to the Closest Related Work Using Our Pragmatic Guidelines

		Focused Granularity	Timely Feedback	Consider Effort	Evaluate Generality
Code	Hassan [6]	No	Yes	No	OSS only
	Kim [8]	Yes	Yes	No	OSS only
	Ours	Yes	Yes	Yes	OSS and Comm
Change	Mockus [12]	No	Yes	No	Comm only
	Kim [7]	Yes	Yes	No	OSS only
	Ours	Yes	Yes	Yes	OSS and Comm

6. CONCLUSION

Practitioners require pragmatic prioritization of quality assurance efforts. Our thesis outlines four guidelines that need to be adhered to make SQA research more pragmatic. We propose two approaches. one at the code- level and one at the change level, that are concerned with the prioritization of unit test creation and the code review of risky changes. We believe that the outcome of this thesis will assist practitioners improve the overall quality of their software.

7. REFERENCES

[1] E. Arisholm and L. 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.

[2] L. Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000.

[3] M. W. Godfrey, A. E. Hassan, J. Herbsleb, G. C. Murphy, M. Robillard, P. Devanbu, A. Mockus, D. E. Perry, and D. Notkin. Future of mining software archives: A roundtable. *Software, IEEE*, 26(1):67 –70, 2009.

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

[5] A. E. Hassan. The road ahead for mining software repositories. In *FoSM 2008.*, pages 48 –57, 2008.

[6] A. E. Hassan and R. C. Holt. The top ten list: Dynamic fault prediction. In *ICSM '05*, pages 263–272, 2005.

[7] S. Kim, E. J. Whitehead, Jr., and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Trans. Softw. Eng.*, 34(2):181–196, 2008.

[8] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller. Predicting faults from cached history. In *ICSE '07*, pages 489–498, 2007.

[9] T. Mende and R. Koschke. Revisiting the evaluation of defect prediction models. In *PROMISE'09*, pages 1–10, 2009.

[10] H. A. Müller, S. R. Tilley, and K. Wong. Understanding software systems using reverse engineering technology - perspectives from the rigi project. In *ACFAS'94*, pages 41–48, 1994.

[11] J. Moad. Maintaining the competitive edge. *Datamation*, 64(66):61–62, 1990.

[12] A. Mockus and D. M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, 2000.

[13] N. Nagappan and T. Ball. Using software dependencies and churn metrics to predict field failures: An empirical case study. In *ESEM '07*, pages 364–373, 2007.

[14] E. Shihab, Z. M. Jiang, B. Adams, A. E. Hassan, and R. Bowerman. Prioritizing unit test creation for test-driven maintenance of legacy systems. In *QSIC'10*, pages 132–141, 2010.

[15] E. Shihab, Z. M. Jiang, W. M. Ibrahim, B. Adams, and A. E. Hassan. Understanding the impact of code and process metrics on post-release defects: A case study on the Eclipse project. In *ESEM'10*, pages 1–10, 2010.

[16] J. Sliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *MSR'05*, pages 24–28, May 2005.

[17] T.-J. Yu, V. Y. Shen, and H. E. Dunsmore. An analysis of several software defect models. *IEEE Trans. Softw. Eng.*, 14(9):1261 –1270, 1988.

[18] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for Eclipse. In *PROMISE '07*, pages 9–15, 2007.