# Using Software Evolution History to Facilitate Development and Maintenance

Pamela Bhattacharya
Department of Computer Science and Engineering
University of California, Riverside, CA, USA
pamelab@cs.ucr.edu

## ABSTRACT

Much research in software engineering have been focused on improving software quality and automating the maintenance process to reduce software costs and mitigating complications associated with the evolution process. Despite all these efforts, there are still high cost and effort associated with software bugs and software maintenance, software still continues to be unreliable, and software bugs can wreak havoc on software producers and consumers alike. My dissertation aims to advance the state-of-art in software evolution research by designing tools that can measure and predict software quality and to create integrated frameworks that helps in improving software maintenance and research that involves mining software repositories.

## Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management—*productivity*

## General Terms

Software, Evolution

## Keywords

Software quality; developer productivity; software evolution; empirical studies

## 1. INTRODUCTION

Software development and software maintenance are time, labor and resource intensive processes. The costs associated with software bugs are huge: a survey in 2002 by the National Institute of Standards and Technology estimates the annual cost of software bugs to about $59.5 billion [8]. percent to 90 percent of total costs. Even for projects where costs are not a primary issue, e.g., in open source settings, maintenance is still a lengthy, arduous process [3]. Existing research has focused on several aspects to benefit the maintenance process. These include designing better programming languages and adaptable Integrated Development Environments (IDE) to improve programmer productivity, building automatic debugging tools or by enabling stronger security policies etc. However, it has not been clear how much of these research achievements have actually helped in reducing maintenance costs. In fact, despite all these advances in research associated with software development and maintenance, they are still ad-hoc process with unsatisfactory results: costs associated with evolution are high, yet new releases contain bugs and fail to operate as desired and we still need to restart most programs to enable updates. In this context, my dissertation has three main research objectives: (1) studying the effects of programming language on quality and developer productivity, (2) designing graph based models to predict maintenance effort and other software metrics, and (3) developing an automated framework to integrate various artifacts in OSS to facilitate software repository mining.


*ICSE* '11, May 21–28, 2011, Honolulu, Hawaii, USA


## 2. CURRENT PROGRESS

In this section we discuss the two different problems we have worked on, how our work helps in identifying factors that affect maintenance costs and tools we have built to reduce this cost and effort significantly.

### 2.1 Impact of Programming Language on Development and Maintenance

Quantitatively assessing the impact of programming language on development and maintenance has been a long-standing challenge [6]. Recently there has been a shift in the language choice for new applications: with the advent of Web 2.0, language popularity statistics show that dynamic, high-level languages are gaining more and more attraction [5]. These languages raise the level of abstraction, promising accelerated development of higher-quality software. However, the lack of tools for static checking, absence of mature analysis and verification tools makes software written in these languages potentially more prone to error and harder to maintain. Prior efforts on analyzing the impact of choice of programming language suffer from one or more deficiencies with respect to the applications they consider and the manner they conduct their studies. For example, they consider applications built by different teams in different languages, hence they fail to control for developer competence, or they consider small-sized, infrequently-used, short-lived projects. Using such methodologies often results in analyses, which cannot be generalized to large real-world applications. In our study we address all these shortcomings by presenting a novel methodology for assessing the impact of programming language on development and maintenance [2].

### 2.2 Automated Framework to Improve Bug Triaging

Popular software projects receive hundreds of bug reports every day [3]. Ideally, each bug gets assigned to a developer who can fix it

in the least amount of time. This process of assigning bugs, known as *bug triaging*, is complicated by several factors: if done manually, triaging is labor-intensive, time-consuming and fault-prone. Moreover, for open source projects, it is difficult to keep track of active developers and their expertise. Prior work [1] has used machine learning techniques to automate bug triaging but has employed a narrow band of tools which can be ineffective in large, long-lived software projects. To redress this situation, we employ a comprehensive set of machine learning tools and analyses that lead to very accurate predictions, and lay the foundation for the next generation of machine learning-based bug triaging [3].

# 3. FUTURE WORK

## 3.1 Graph-based Metrics as Defect and Maintenance Predictors

Graph-based metrics have been used in software maintenance studies earlier for defect prediction. Zimmermann et al. [10] used function call graphs to predict the failure probability of files. Pinzger et al. [9] build networks of developers connected via code artifacts to predict failures. Nagappan et al. [7] have used function call graphs for failure prediction by extracting complexity metrics. However, none of these studies has proposed any graph-based metrics to predict maintenance effort or identify critical spots in the source code, which are more prone to severe bugs. We plan to construct graph-based models of the software (function call graphs and module collaboration graphs) to compute the relations between various software elements and use novel and existing techniques to improve software quality and decrease maintenance effort by analyzing the structure of the software. This work is in progress and we provide brief discussion of our hypotheses and preliminary results.

*Prioritizing bug fixes.* When a bug is reported, the maintainers review the bug and assign it a severity rank based on how badly it affects the program. For a software provider, it should be a priority to not only minimize the total number of bugs, but also try to ensure that those bugs that do occur are low-severity, rather than Blocker or Critical. To prioritize bug severity, we propose using a metric called NodeRank to help identify critical functions, i.e., functions that, when buggy, are likely to exhibit high-severity bugs. Analogous to the PageRank algorithm [4], our NodeRank algorithm is a link analysis algorithm that assigns a numerical weight to each node in a graph, i.e., to each function in the function call graph. NodeRank measures the relative importance of that node in the graph, which in our case translates to the importance of that function in the software. The higher the NodeRank of a function, the more important it is for the program and, accordingly, any bug associated with a high-ranked node has high severity. NodeRank can give maintainers a fast and accurate way of identifying a critical function or module by knowing its NodeRank. Our preliminary results suggest that NodeRank is an effective predictor of bug severity, and can be used to identify "critical" spots in the source code.

*Estimating maintenance effort.* A leading cause for high software maintenance costs is the difficulty associated with changing the source code, e.g., for adding new functionality or refactoring. We propose to identify problematic, difficult-to-change modules using a module-level metric called Modularity Ratio. We define the modularity ratio of a module as the ratio between the coupling and the cohesion values of that module. Our preliminary results have shown that, as the cohesion/coupling ratio increases for a module (which means the software structure improves), there is an associated decrease in maintenance effort for that module.

*Committer experience vs. bug severity.* We want to study if developer expertise—measured as the number of bugs fixed and patches committed—correlates with number of bug introducing changes made by the developer and the average severity of those bugs introduced. Our hypothesis is that code introduced by expert developers would be less prone to bugs or will introduce bugs of low severity.

*Effects of refactoring.* We want to identify structural changes in the call graph that result from refactoring and test how these changes improved code quality and maintenance. We hypothesize that functions or modules, which has been refactored will undergo reduction in average bug severity and maintenance effort in the succeeding versions.

## 3.2 Framework to Integrate Software Artifacts

Software projects use different repositories for storing project and evolution information such as source code, bugs and patches. An integrated system that combines these multiple repositories, along with efficient search techniques that can answer a broad range of queries regarding the project's evolution history would be beneficial to both software developers (for development and maintenance) and researchers (for empirical analyses). Integrating this information is a tedious, cumbersome, error-prone process when done manually, especially for large projects. Previous approaches to this problem use frameworks that limit the user to a set of predefined query templates, or use query languages with limited power. In this work, we argue the need for a framework built with recursively enumerable languages that can answer temporal queries, and supports negation and recursion. As a first step toward such a framework, we use a Prolog-based system that we built, along with an evaluation of real-world integrated data from the Firefox project. Our system allows for elegant and concise, yet powerful queries, and can be used by developers and researchers for frequent development and empirical analysis tasks.

# 4. CONCLUSION

My dissertation will study and propose improvements to the maintenance process during software evolution. We plan to use statistically significant empirical analyses to study software evolution and maintenance patterns and build frameworks that benefit software maintenance process and research in empirical studies.

# 5. REFERENCES


[1] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *ICSE*, pages 361–370, 2006.

[2] P. Bhattacharya and I. Neamtiu. Assessing programming language impact on development and maintenance: A study on C and C++. In *ICSE 2011*.

[3] P. Bhattacharya and I. Neamtiu. Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging. In *ICSM*, 2010.

[4] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 1998.

[5] DedaSys LLC. Programming Language Popularity.

[6] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri. Challenges in software evolution. In *IWPSE*, 2005.

[7] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *ICSE '06*.

[8] NIST. The economic impacts of inadequate infrastructure for software testing. Planning Report, May 2002.

[9] M. Pinzger, N. Nagappan, and B. Murphy. Can developer-module networks predict failures? In *FSE*, 2008.

[10] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *ICSE '08*.