The Journal of Systems and Software 103 (2015) 412–422

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Investigating the effect of “defect co-fix” on quality assurance resource allocation: A search-based approach

Hadi Hemmati[a,*], Meiyappan Nagappan[b], Ahmed E. Hassan[c]

[a] Department of Computer Science, University of Manitoba, Canada
[b] Department of Software Engineering, Rochester Institute of Technology, USA
[c] School of Computing, Queen’s University, Canada

ARTICLE INFO

Article history:
Received 2 December 2013
Revised 25 October 2014
Accepted 15 November 2014
Available online 9 December 2014

Keywords:
Defect prediction
Co-fix
Search-based prioritization

ABSTRACT

Allocation of resources to pre-release quality assurance (QA) tasks, such as source code analysis, peer review, and testing, is one of the challenges faced by a software project manager. The goal is to find as many defects as possible with the available QA resources prior to the release. This can be achieved by assigning more resources to the more defect-prone artifacts, e.g., components, classes, and methods. The state-of-the-art QA resource allocation approaches predict the defect-proneness of an artifact using the historical data of different software metrics, e.g., the number of previous defects and the changes in the artifact. Given a QA budget, an allocation technique selects the most defect-prone artifacts, for further investigation by the QA team. While there has been many research efforts on discovering more predictive software metrics and more effective defect prediction algorithms, the cost-effectiveness of the QA resource allocation approaches has always been evaluated by counting the number of defects per selected artifact. The problem with such an evaluation approach is that it ignores the fact that, in practice, fixing a software issue is not bounded to an artifact under investigation. In other words, one may start reviewing a file that is identified as defect-prone and detect a defect, but to fix the defect one may modify not only the defective part of the file under review, but also several other artifacts that are somehow related to the defective code (e.g., a method that calls the defective code). Such co-fixes (fixing several defects together) during analyzing/reviewing/testing of an artifact under investigation will change the number of remaining defects in the other artifacts. Therefore, a QA resource allocation approach is more effective if it prioritizes the artifacts that would lead to the smallest number of remaining defects.

Investigating six medium-to-large releases of open source systems (Mylyn, Eclipse, and NetBeans, two releases each), we found that co-fixes happen quite often in software projects (30–42% of the fixes modify more than one artifact). Therefore, in this paper, we first introduce a new cost-effectiveness measure to evaluate QA resource allocation, based on the concept of “remaining defects” per file. We then propose several co-fix-aware prioritization approaches to dynamically optimize the new measure, based on the historical defect co-fixes. The evaluation of these approaches on the six releases shows that (a) co-fix-aware QA prioritization approaches improve the traditional defect prediction-based ones, in terms of density of remaining defects per file and (b) co-fix-aware QA prioritization can potentially benefit from search-based software engineering techniques.

1. Introduction

Typical pre-release software quality assurance (QA) activities involve tasks such as code inspection, peer review, and testing of software artifacts. The goal of these QA activities is detecting and fixing defects before they are perceived as failures by the users. Allocating pre-release QA resources to the QA tasks, is one of the critical duties of a software project manager. Resources are always limited and managers need to prioritize their resources and fix as many defects as possible, within a limited time, before release. Some bugs are already identified by the development team, while unit/integration testing. So if there is any known unfixed bugs in the system, they will get priority to be fixed (by fault localization and debugging). However, many other bugs may slip through the initial testing. The goal of the QA team, at this stage, is ensuring that the existing QA budget

* Corresponding author. Tel.: +1 2044749254.
E-mail addresses: hemmati@cs.umanitoba.ca (H. Hemmati), mei@se.rit.edu (M. Nagappan), ahmed@cs.queensu.ca (A.E. Hassan).

http://dx.doi.org/10.1016/j.jss.2014.11.040
0164-1212/© 2014 Elsevier Inc. All rights reserved.

is wisely spent, even if there is no reported bug yet. Therefore, the QA team, usually estimates riskiness of each artifact (e.g., classes) and prioritize the artifact for further QA investigations (which mostly is a combination of inspection and testing before each release).

One of the current practices for estimating riskiness, which is well studied in the literature, is using defect prediction approaches (Fenton and Neil, 1999; Hall et al., 2011). Such approaches usually estimate the number of defects per artifact[1] (e.g., components, classes, and methods). These estimates are then used, to prioritize the allocation of resources to the most defect-prone artifacts. In this paper, we call this process, QA prioritization.

To improve QA prioritization, several prediction techniques and many product/process/social metrics have been studied on many software systems (Fenton and Neil, 1999; Hall et al., 2011). A QA prioritization is typically evaluated by its effectiveness in terms of properly assigning resources to the artifacts that contain more defects. A typical effectiveness measure for evaluating a QA prioritization approach is the number of defects per file. Assume we are prioritizing source code files of a system for further QA investigation. Assume the top three most defective files (A, B, and C) contain (20, 15, and 10) defects, respectively. The most effective QA prioritization approach would rank the files as A, B, and then C (i.e., from most to least number of defects). This ranking assures that with any budget between one to three files to investigate, the QA task hits the areas of the code with the most number of defects.

Assume that the investigation of A's defects results in modifying some other files due to code dependency of the defects. In other words, while fixing the defects of A, several defects of other files are also fixed. We call this phenomenon, a *co-fix*. Now consider a scenario where 10 of B's defects are already fixed by the time that we are done fixing A's defects and none of the C's defects are touched. The traditional effectiveness measure (the original number of defects per file) still insists that B is the next best choice for QA allocation. However, B only contains five remaining defects, whereas C still has 10 untouched defects.

In this paper, we first argue that the current effectiveness measure, based on the number of defects per file, is not the best evaluation mechanism for QA prioritization, due to the co-fixes. We then study six medium-to-large releases of open source systems (Mylyn, Eclipse, and NetBeans; two releases each), and realize that there are several cases of co-fix in these releases (30–42% of the fixes modify more than one file[2]). To consider the co-fix effect on the defect-proneness of artifacts, in this paper, we first introduce a new cost-effectiveness measure for QA prioritization, *density of remaining defects* (DRD), per artifact (per file, in our case). We use density rather than absolute number of defects, since it better captures the cost involved in detecting/repairing of the defects per artifact (Arisholm and Briand, 2006; Kamei et al., 2010; Mende and Koschke, 2009, 2010; Shihab et al., 2013). However, the novelty of DRD is in introducing the concept of *remaining defects* vs. all defects, as the effectiveness measure.

To optimize the new cost-effectiveness measure for QA prioritization (DRD), we propose a novel approach for ranking files, which dynamically updates the ranks, based on the current remaining defects per file (co-fix-aware approach). We then compare two variations of a traditional QA prioritization approach: *TradNum* (sorting files based on their number of predicted defects) (Marcus et al., 2008; Neuhaus et al., 2007; Shihab et al., 2010; Zimmermann and Nagappan, 2008) and *TradDens* (sorting files based on their density of predicted defects) (Arisholm and Briand, 2006; Kamei et al., 2010; Mende and Koschke, 2009, 2010; Shihab et al., 2013) with two different variations of our *co-fix-aware* approach *CoFixNum* (a co-fix-aware ranking based on the number of predicted defects) and *CoFixDens* (a co-fix-aware ranking based on the density of predicted defects).

To empirically evaluate the performance of our proposed approaches, we study, in particular, the following research question:

Research Question: Can we improve the cost-effectiveness of traditional QA prioritization, using a co-fix-aware approach?

Applying the four approaches (*TradNum*, *TradDens*, *CoFixNum*, and *CoFixDens*) on the releases of the open source systems, we show that *CoFixDens* is significantly more effective than the alternatives in ranking files with higher DRDs, in most of our case studies. We also show the feasibility of search algorithms on this research domain, by investigating the applicability of a simple hill climbing local search algorithm for QA resource prioritization and comparing it with the *CoFixDens* approach. Therefore, the contributions of this paper can be summarized as:

1. Introducing the concept of remaining defects as the basis for measuring the effectiveness of QA resource prioritization.
2. Introducing the concept of co-fix-awareness to dynamically rank source code files.
3. Proposing two co-fix-aware ranking algorithms (*CoFixNum*, and *CoFixDens*).
4. Empirically comparing several QA resource prioritization algorithms.
5. Investigating the applicability of heuristic-based search algorithms on QA resource prioritization.

2. The effect of co-fix on QA prioritization: motivating example

Fig. 1 is an example from the browsing package in Eclipse version 3.0. As it is shown, an issue/bug, with an id of 63036, was fixed by repairing three defective files (PackagesViewHierarchalContentProvider.java, PackagesViewFlatContentProvider.java, and JavaBrowsingPart.java). This is an example of co-fix, where several source code defects in different files are repaired together during one single fix commit. Co-fixes may have several reasons. One reason is that several defects are logically related to a single issue, i.e., fixing the issue requires fixing all the related defects. A co-fix may also happen due to code dependencies. For example, two defects in two methods of two distinct classes need to be fixed together, since one method is calling the other. Regardless of the cause, one may study the defect co-fix distribution among artifacts (source code files in this paper) and its effect on the QA prioritization.

As an example of *co-fix* distribution, let us look at four fixes that affect three files from the Netbeans project as follows:

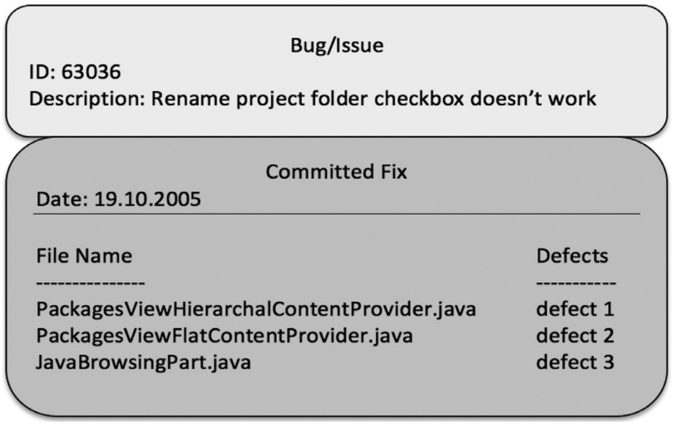
Bug/Issue
ID: 63036
Description: Rename project folder checkbox doesn't work
Committed Fix
Date: 19.10.2005
File Name
Defects
PackagesViewHierarchalContentProvider.java
defect 1
PackagesViewFlatContentProvider.java
defect 2
JavaBrowsingPart.java
defect 3

Fig. 1. An example of Fix–Defect relationship from Eclipse version 3.0 browsing package. For the sake of simplicity, for each file, only the defects related to the fix is shown.

[1] Granularity of the artifacts depends on many factors such as the availability of historical data, the cost of QA per artifact, and the QA policies in place in the project.

[2] Since all these systems are in Java, each main class is usually a separate file. So the granularity of artifacts in our analysis can be considered as file or class level.