Determining “Grim Reaper” Policies to Prevent Languishing Bugs

Patrick Francis
Industrial Software Systems
ABB Corporate Research
Raleigh, NC, USA
patrick.francis@us.abb.com

Laurie Williams
Department of Computer Science
North Carolina State University
Raleigh, NC, USA
williams@csc.ncsu.edu

Abstract—Long-lived software products commonly have a large number of reported defects, some of which may not be fixed for a lengthy period of time, if ever. These so-called languishing bugs can incur various costs to project teams, such as wasted time in release planning and in defect analysis and inspection. They also result in an unrealistic view of the number of bugs still to be fixed at a given time. The goal of this work is to help software practitioners mitigate their costs from languishing bugs by providing a technique to predict and pre-emptively close them. We analyze defect fix times from an ABB program and the Apache HTTP server, and find that both contain a substantial number of languishing bugs. We also train decision tree classification models to predict whether a given bug will be fixed within a desired time period. We propose that an organization could use such a model to form a “grim reaper” policy, whereby bugs that are predicted to become languishing will be pre-emptively closed. However, initial results are mixed, with models for the ABB program achieving F-scores of 63-95%, while the Apache program has F-scores of 21-59%.

Keywords—languishing bugs, software management, policy

I. INTRODUCTION

Inevitably, through the course of software development engineers inject defects into program code. As a software project sees increasing adoption, its creators may then be “rewarded” with an increasing number of defect reports. The defect tracking database of mature software projects, therefore, may contain several thousands of defects, if not more.

Unfortunately, most software projects have a finite amount of resources to devote to bug fixing, which may be exceeded by the quantity of reported bugs. In such cases, a project may accumulate an ever-expanding backlog of unfixed defects. Software managers must prioritize maintenance efforts, often based on factors such as defect severity, effort to fix, importance to the customer, or impact on the code base. However, this prioritization activity may result in defects that are repeatedly given a low priority or deferred to a future release. We use the term languishing bug to refer to the defects that remain open and unfixed for a lengthy period of time.

In many cases, project members understand that these languishing bugs are unlikely to ever be fixed. However, various costs may be incurred by having these open defects languishing in the bug database. For example, these defects may need to be repeatedly analyzed to determine whether they should be targeted for an upcoming release. They likewise distort a project’s bug metrics and hinder efforts to form a realistic view of the remaining work. Furthermore, any time spent investigating and debugging a defect that is ultimately never resolved is largely wasted effort.

This problem of an overwhelming stream of incoming tasks will be familiar to anyone with an overflowing email inbox. Many people receive emails faster than they can process them, until they have collected an infeasible backlog of unanswered messages. One strategy for dealing with this problem is to declare “email bankruptcy”, and simply delete (or ignore) any unanswered emails older than some given age. [5] Any email that was actually important will presumably be resent. We hypothesize that a similar approach can be effective for managing languishing bugs.

The goal of this work is to help software practitioners mitigate their costs from languishing bugs by providing a technique to predict and pre-emptively close them. We propose that software teams develop a “grim reaper” policy, named after the personification of death. This policy would use a predictive model to identify potentially languishing bugs, which could then be closed, acknowledging the reality that they are unlikely to be fixed. The bugs could be reopened if there are any future duplicate reports, but in the meantime developers are free from having to manage them.

To further our goal, we formulate several research questions:

RQ1: How prevalent are languishing bugs in a software project?

RQ2: How well can a model be learned from the defect data to predict which defects will languish?

To answer these questions we examined the defect repositories from two subject programs: an embedded program developed by ABB Inc., and the Apache HTTP server. We measured the duration that each defect was open and analyzed the distributions of these values. We have also devised and evaluated a technique for constructing a grim reaper policy, based on historical defect data. The technique includes a decision tree model trained to identify bugs that are unlikely to be fixed within some desired period of time, and are therefore considered languishing. We have trained several models for
varying values of parameters and assessed their accuracy at predicting languishing bugs.

The remainder of the paper is organized as follows. Section II discusses related work. Section III presents data on languishing bugs in our subject programs. Section IV discusses grim reaper policies and our results in mining them from historical data. Section V concludes and discusses future work.

II. RELATED WORK

Many previous researchers have looked at predicting measures about bugs, such as the fix time. For example, Anbalagan and Vouk [1] predicted fix times for bugs in Ubuntu based on the number of people involved in the report. Hooimeijer and Weimer [7] analyzed Firefox bugs to predict whether they would be triaged within a given time. Bhattacharya and Neamtiu [3] trained fix time models on several large open source projects and found that features that other authors had reported as predictive performed poorly on these projects. Guo et al. [6] studied the characteristics, such as the reputation of the submitter, that distinguish bugs that get fixed from those that do not. They also trained a logistic regression model to predict whether a given bug will ever be fixed. Our work differs from these in that it seeks to predict whether a bug will be fixed, rather than just triaged, and fixed within a given time period. Also, we use only the data directly available in the bug report, rather than computing our own features, such as the submitter reputation.

The work that is perhaps most similar to ours is by Zhang et al. [9]. They analyzed several commercial projects and used a k-Nearest Neighbor model to predict whether a given bug would be fixed slowly or quickly, i.e. above or below a given time threshold. However, our work is distinguished by our proposal to use such a model to form a policy for pre-emptively closing bugs.

III. PREVALENCE OF LANGUISHING BUGS

Before developing solutions for managing languishing bugs, we first need to assess how common they are. Therefore, our first research question is: How prevalent are languishing bugs in a software project?

A. Subject Programs

We examined two subject programs. The first is a proprietary embedded program developed by ABB Inc., which we refer to as ABB1. The program contains approximately 2.1 million lines of code, written primarily in C and C++, and has been developed since 2000. We extracted the defect data directly from the bug database and included all available data as of July 2012.

The second subject program is the open source Apache HTTP server, version 2.x. [2] This program contains 340 thousand lines of code, written in C and C++. We collected the defect data from the project’s Bugzilla web page and included all available data as of March 2013.

For each bug, we calculated its Fix Time, which we define as the time (in days) between when the bug was opened and when it was closed. For ABB1, the bug data includes both of these dates as fields. Apache includes only the Opened Date, and so we extracted the Closed Date from the change history, using the date when the Status field was most recently set to a closed status. For bugs that were still open, and therefore had no Closed Date, we instead used the date that we extracted the data from the database. Therefore, the “Fix Time” for an open bug is equivalent to the age of the bug.

B. Distribution of Defect Fix Times

For both subject programs, we found that the fix times are generally skewed toward short fix times. Nevertheless, a substantial proportion of the defects have fix times that are much longer than average. Fig. 1 summarizes the distributions of fix times for defects that are currently open (as of the data extraction date mentioned in Section III.A) and ones that are closed. For these boxplots, any defects that are closed as invalid are excluded (e.g. they are marked as Not Repeatable, or Works As Designed, or similar). For ABB1, 23.7% of the defects are open, while for Apache 33.3% of the defects are open.

As shown in the figure, the fix times for open defects are generally much higher than for defects that were closed. For ABB1, the median fix time for open defects is 862 days, versus 131 days for closed defects. In fact, 88% of the open defects have fix times longer than the 75th percentile time for the closed defects. Apache is similar; the median fix time for open defects is 1242 days, versus 202 days for closed defects. Furthermore, 73% of the open defects have fix times longer than the 75th percentile time of the closed defects.

For both programs, the closed defects do have a number of outliers with very long fix times. However, defects with long fix times are much more likely to remain open rather than be closed, and both projects contain many defects with long fix times. These results show that both subject programs contain a substantial number of languishing bugs that are unlikely to be fixed.

IV. “GRIM REAPER” POLICY

To avoid the costs and problems associated with languishing bugs, organizations should strive to prevent them from languishing in the first place. However, the obvious solution of resolving all bugs in a timely manner is often not feasible due to resource constraints or market realities. We therefore propose that organizations develop a “grim reaper” policy (named after the personification of death). This policy would use a model to predict whether a given bug is likely to be fixed within some chosen period of time, or whether it will languish. Those bugs that are predicted to be languishing can be pre-emptively closed.
Bugs closed in this way can be given a disposition such as “Languishing”, “Inactive”, or “Expired”, to distinguish them from bugs that have been resolved. Some defect management systems already have a disposition of “Won’t Fix”, to indicate defects that are legitimate but that nevertheless will not be fixed. This idea is the same as what we propose, and the grim reaper policy may serve as a formalized means to apply this disposition.

A. Policy Parameters

We consider two primary parameters in constructing a grim reaper policy: the policy application date, and the languishing criteria. Values for these parameters are necessary to train the classification model.

The first parameter is the policy application date, which specifies the date at which a given defect should be assessed to see if it should be closed. Two approaches an organization could use to set this parameter include:

- Bug age: apply the policy to a bug once it has been open for a certain amount of time. The chosen age might be a fixed time, such as one year or two years, or based on the project data, such as the mean bug fix time plus X%.
- Release cycle: apply the policy to all open bugs at a given point in the release cycle, e.g. one month after a release, or mid-way between two releases.

The second parameter for a grim reaper policy is the set of languishing criteria. These criteria determine which of the bugs in the repository should be labeled as languishing and are used to train the classification model. That is, at what point should a bug be considered “languishing”? Three possible criteria to determine the languishing bugs are:

- Bug age: label as languishing those bugs that have been open for more than some defined period of time. This period might be an arbitrary cutoff such as two years or five years, or it might be determined based on the historical likelihood of a bug being closed after that age.
- Release cycle: bugs that have remained open for more than X releases of the product are considered languishing.
- User interest: bugs that have not been re-reported for some given period of time are considered languishing.

An organization may also choose to combine several languishing criteria.

B. Experimental Setup

To answer RQ2, we have trained a number of decision tree models to predict languishing bugs. As described above, many possible settings for the parameters of policy application date and languishing criteria exist. We selected one scheme for each and evaluated several values to assess the sensitivity of the models to these parameters.

For the policy application date, in this paper we analyze a scheme based on the bug age, and selected values of one year, two years, and three years. That is, each bug would be evaluated against the policy when it reached the given age.

However, to train the models using the correct defect data, each bug needed to be reverted to the state it was in at the desired application age, rather than its most recent state. We reverted the data using the change history in the bug database, which records all the changes made to a given bug. We un-did each change individually, in reverse chronological order, until the bug had been reverted to the desired age (1 year, 2 years, etc.).

To determine which bugs should be labeled as languishing, we used a scheme based on the bug age, using cutoff values chosen based on the historical bug data. We selected the bug ages at which 80%, 90% or 95% of the bugs open at least that long were still open. This idea is illustrated in Fig. 2. The horizontal axis shows the minimum fix time. The vertical axis indicates, among the bugs that were open for at least that time, what proportion were eventually closed (light gray) and what proportion are still open (dark gray). We then identified the minimum ages at which at least 80%, 90% or 95% of the bugs remained open.

For each combination of parameter values, we trained five decision tree classification models, using five-fold cross validation to form the training and testing sets. The models were created using the rpart package in R [8], which implements the CART classification tree algorithm by Breiman, et al. [4].

We trained the models using only those defects that were still open at the given policy application age. We also excluded any defects that were currently open but had a fix time less than the given languishing age cutoff. (Since these defects have not been closed, but have also not existed for long enough to be considered languishing, they cannot be properly labeled as either languishing or not.) We trained the models using all the fields recorded in the bug database, such as Severity, Component or Reported Build. However, we excluded those fields that would be obviously spurious (such as the BugID, or Title) or would contain inaccurate data. For example, several fields from the Apache data were excluded because they were not recorded in the change history and therefore could not be properly reverted to their state at the policy application date. In total, we included 81 fields for ABB1, and 33 fields for Apache. (We do not list them here due to space constraints.)

C. Model Results

For our experiment, we used the two subject programs described earlier. For ABB1, the languishing criteria of 80%
open, 90% open, and 95% open correspond to 850 days, 1500 days, and 2000 days, respectively. Furthermore, 10.65% of the total bugs have fix times longer than 850 days, 5.19% longer than 1500 days, and 2.34% longer than 2000 days.

For Apache, the languishing criteria correspond to 2300 days, 2950 days, and 3200 days, respectively. Furthermore, 4.89% of the total bugs have fix times longer than 2300 days, 1.95% longer than 2950 days, and 1.29% longer than 3200 days.

The performance of our models is summarized in Tables I and II. Each cell shows the average of the Precision, Recall and F-Score for each model. (For ABB1, the combination of 1095 days and 80% open was omitted because the policy application date is higher than the languishing cutoff.) The models for ABB1 perform relatively well, with F-scores ranging from 63% up to 95%. The performance for Apache, however, is much worse; the F-scores range from 21% to 59%. For both subject programs, the accuracy of the model decreases as the time between the application date and the languishing cutoff date increases. That is, models that have to predict further into the future perform worse.

We also examined the potential costs of using these models in a grim reaper policy. One measure of this cost is the number of bugs that were predicted to languish, but were actually closed as fixed. In other words, how many bug fixes would have been “lost” by closing the bugs early? This number includes both false positives, i.e. bugs mis-predicted by the model, and open bugs that did reach the languishing cutoff age, but were later closed. For ABB1, 10% to 24% of the predicted languishing bugs were eventually actually fixed. With Apache, 34% to 47% of the predicted languishing bugs were fixed.

The poor performance of the models on Apache may be partly explained by the choice of languishing criteria. Both subject programs use the same criteria (80% open, etc.), but with Apache these correspond to much later ages. Since the policy application dates are the same, though, the intervals between the application dates and the cutoff dates are much longer, which correlates with worse predictive accuracy, as mentioned above. Models trained on Apache using an application date of 1 year and languishing cutoffs of 60% open (1050 days) and 70% open (1650 days) showed greatly improved F-scores: 73% and 57%, respectively.

V. CONCLUSIONS AND FUTURE WORK
In this paper we present an approach to help organizations mitigate their costs from languishing software bugs. Specifically, we propose a “grim reaper” policy whereby a model predicts which bugs will become languishing and the organization pre-emptively closes them. We analyzed two subject programs, one industrial and one open source, and found that they both contained a substantial number of languishing bugs. We then trained decision tree models for nine combinations of policy application date and languishing cutoff date. We found that the models predict languishing bugs with moderate accuracy for ABB1, but did not perform as well on Apache. For both programs, the models performed best when the policy application date and languishing cutoff were closest together. We also measured the costs incurred by the use of this policy, in terms of lost bug fixes. We found that 10% to 47% of predicted languishing bugs were actually fixed. Future work will include detailed study of the costs and benefits of the proposed grim reaper policies. We will also investigate alternative classification methods, such as improved machine learning algorithms or natural language analysis of the defect report text.

REFERENCES