Available online at www.sciencedirect.com
ScienceDirect
The Journal of Systems and Software 80 (2007) 63–73
The Journal of Systems and Software
www.elsevier.com/locate/jss

Identifying and characterizing change-prone classes in two large-scale open-source products

A. Güneş Koru [a,*], Hongfang Liu [b]

[a] Department of Information Systems, University of Maryland, Baltimore County, UMBC—EASEL, Empirical and Applied Software Engineering Laboratory, 1000 Hilltop Circle, Baltimore, MD 21250, USA
[b] Georgetown University Medical Center, Department of Biostatistics, Bioinformatics, and Biomathematics, 4000 Reservoir Road, NW Suite 120, Washington, DC 20007, USA

Received 11 December 2005; received in revised form 3 May 2006; accepted 9 May 2006
Available online 27 June 2006

Abstract

Developing and maintaining open-source software has become an important source of profit for many companies. Change-prone classes in open-source products increase project costs by requiring developers to spend effort and time. Identifying and characterizing change-prone classes can enable developers to focus timely preventive actions, for example, peer-reviews and inspections, on the classes with similar characteristics in the future releases or products. In this study, we collected a set of static metrics and change data at class level from two open-source projects, KOffice and Mozilla. Using these data, we first tested and validated Pareto's Law which implies that a great majority (around 80%) of change is rooted in a small proportion (around 20%) of classes. Then, we identified and characterized the change-prone classes in the two products by producing tree-based models. In addition, using tree-based models, we suggested a prioritization strategy to use project resources for focused preventive actions in an efficient manner. Our empirical results showed that this strategy was effective for prioritization purposes. This study should provide useful guidance to practitioners involved in development and maintenance of large-scale open-source products.

Keywords: Static metrics; Open-source development; Object-oriented programming; Change-prone classes; Software maintenance

1. Introduction

Change-prone classes in software require particular attention because they require effort and increase development and maintenance costs. Identifying and characterizing those classes can enable developers to focus preventive actions such as, peer-reviews, testing, inspections, and restructuring efforts on the classes with the similar characteristics in the future. As a result, developers can use their resources more efficiently and deliver higher quality products in a timely manner.

Such strategies are not only useful for traditional closed-source development projects, but also for open-source projects as companies initiate and support open-source projects increasingly to gain business advantages (Golden, 2004). For example, Sun Microsystems supports the development of OpenOffice,[1] which is a full and freely available office suite. Apple supports the development of an open-source operating system, Darwin,[2] which is the core of its commercial product, Mac OS X.[3] Similarly the default Web browser on Mac OS X platform, Safari,[4] draws source code from the open-source Web browser, Konqueror.[5]

* Corresponding author. Tel.: +1 410 455 8843; fax: +1 410 455 3095.
E-mail addresses: gkoru@umbc.edu (A. Güneş Koru), hfliu@georgetown.edu (H. Liu).

[1] http://www.openoffice.org.
[2] http://developer.apple.com/darwin.
[3] http://www.apple.com/macosx.
[4] http://www.apple.com/safari.
[5] http://www.konqueror.org.

0164-1212/$ - see front matter © 2006 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2006.05.017

As open-source development model gets commercialized in many different areas from operating systems to desktop applications, schedule pressures and limited resources have become important concerns for open-source projects too. In addition, many widely used open-source products have also become large and complex software systems. Focusing on *all* product parts equally is difficult in open-source projects even with the existence of a large developer and user base. Therefore, it is desirable to identify and characterize the change-prone classes in open-source products to enable timely and focused preventive actions which can result in efficient use of resources.

Fortunately, a number of empirical studies conducted on closed-source products have shown that a large majority of problems (around 80%), such as, defects, change, rework, and so on, are actually rooted in a small proportion (around 20%) of the classes in a software system (Porter and Selby, 1990; Tian and Troster, 1998). This phenomenon has been commonly referred as Pareto's Law (2001) (also as the 80:20 rule).

Following these observations, this study mainly sought answers to the following two questions:

(1) Does Pareto's Law apply to open-source products? In other words, do change-prone classes constitute a small proportion in open-source products? Closed source products are often planned, designed, and documented. On the other hand, open-source products are often developed and maintained in an evolutionary way (Raymond, 1999). As a result, a different change distribution could be assumed for an open-source product. For example, it could be assumed that the changes are less localized and more evenly distributed over the classes of an open-source product. Validating Pareto's Law first motivates us for further research in this area.
(2) If Pareto's Law is applicable, how can the change-prone classes be identified and characterized? To be widely adopted, suggested techniques should be easy-to-use for practitioners. In addition, models obtained using those techniques should be simple and intuitive enough to be easily understood and interpreted by practitioners.

To obtain empirical evidence and answer the above questions, we analyzed a set of static metrics and change data that belonged to two large-scale open-source products, KOffice and Mozilla. The static metrics and change data were extracted from the CVS (Concurrent Versions System)[6] repositories of these two object-oriented products and transformed to class level. Structural properties of classes, as measured by static metrics, have been associated with change-proneness by previous studies as discussed later in Section 2. Therefore, in this study, we used static metrics in tree-based models to identify and characterize the change-prone classes in these two object-oriented products. We also suggested a prioritization strategy to use project resources for focused preventive actions in an efficient manner.

In the following, we start with the related work that investigated the relationship between static metrics and change-proneness. After that, we describe our data collection process. Section 4 explains the approach and techniques used in our data analyses. Then, Section 5 discusses the validation of Pareto's Law for KOffice and Mozilla, identification and characterization of change-prone classes in them, and the suggested prioritization strategy by presenting our empirical results. Finally, we present our conclusions and perspectives. The static metrics appeared in the tree-based models are explained in Appendix A. Appendix B includes detailed information about the pruning method used in the study to prune tree-based models.

2. Related work

In the literature, several static metrics for software have been proposed (Briand et al., 1998, 1999; Card and Glass, 1990; Chidamber and Kemerer, 1994) and found to be associated with various risk factors, such as change, defects, and effort. In this section, we mainly summarize the related work about change-proneness because change data was used in this study. Change is seen as an essential characteristic of software systems (Lehman and Belady, 1985), however, it is also perceived as an important risk element (Boehm, 1991) because changes require effort, and they increase development and maintenance costs.

Henry and Kafura found strong correlations between their metrics based on information flow among system components and the number of changed source lines in the Unix operating system (Henry and Kafura, 1981; Kafura and Henry, 1982). Kitchenham et al. were not able to validate Henry and Kafura's information flow metrics (Kitchenham et al., 1990). However, they found that change was related to some other metrics such as fanout, size, and number of branches. In another study, Kafura and Reddy validated their metric by using the subjective quality evaluations of the experts as the response variable (Kafura and Reddy, 1987).

Basili and Hutchens used a syntactic complexity metric (SynC) and observed a correlation between this metric and change when they used the data from the development of 19 student projects (Basili and Hutchens, 1983). Binkley and Schach found that their *Coupling Dependency Metric* (CDM) and change were correlated (Binkley and Schach, 1998). Li and Henry used some object-oriented metrics and observed correlations to change on two systems which had 39 and 71 classes (data points) (Li and Henry, 1993). Lindvall found that large C++ classes were more change prone (Lindvall, 1998). Lindvall and Sandhall studied

[6] CVS is a configuration management system commonly used for source code control purposes. See http://ximbiot.com/cvs/wiki.

how effectively software developers could predict change-prone C++ classes (Lindvall and Sandahl, 1998), and they concluded that characterization of change-prone classes could be useful for better prediction performance.

In summary, as discussed by Shepperd and Ince in detail (Shepperd and Ince, 1993), some researchers found strong relationships between static measures and change or defects but some researchers observed weaker relationships. Most of those studies mentioned by Shepperd and Ince involved correlation analysis or linear regression models. Recently, more sophisticated techniques to identify problematic modules, for example, Neural Networks (Khoshgoftaar et al., 1997; Khoshgoftaar and Szabo, 1996), Principal Component Analysis (Briand et al., 2000; Khoshgoftaar et al., 1996), Tree Based Models (Koru and Tian, 2003; Porter and Selby, 1990, 2001), and Optimized Set Reduction (Briand et al., 1993). Koru and Tian stressed that such techniques are needed because the relationship between static metrics and change count is usually not a monotonic linear relationship, therefore, models based on simple linear regression and correlation will not be sufficient (Koru and Tian, 2005).

The related work in the literature suggests that static metrics are generally associated with change-proneness. This study focuses on developing understandable and interpretable models that characterize change-prone classes in open-source products by using static metrics. Understandability and interpretability of the resulting models are important because practitioners need to understand the characteristics of change-prone modules from those models and plan preventive actions. On these issues, the tree-based modeling technique, which is explained in Section 4, has advantages over some other techniques, for example, neural networks.

3. Data collection

We analyzed the change and static metrics data that belonged to two product releases, KOffice 1.1 and Mozilla 1.0, referred to as KOffice and Mozilla, respectively, in this paper. More information about the history, development environments, and development processes of these open-source products can be found at their project Websites.[7] These specific releases were chosen only because they were accepted as major releases by the project communities. Both products were long-lived and large-scale products. Working on long-lived products prevents research results from being biased by the potential data fluctuations experienced during short periods of time. Studying large-scale products provides us with a large number of data points, a desirable situation for statistical analyses.

Another common characteristic of these products is that they are both object-oriented products written largely in C++. Object-oriented development techniques and programming languages have gained wide-spread use which also influenced our product selection criteria in this study. Clearly, there might be other products that are eligible according to these criteria. However, the number of products was limited to two in order to make this study feasible.

We treated each C++ class in these products as a single data point in our analysis. The number of data points obtained after some preprocessing steps is given in Section 5.1. Each data point included a large set of *static metrics* used as the features and post-release *change count* used as the response variable. Change data was recorded accurately because of the consistent use of CVS commits. We considered the possibility of using defect data too. However, there were several issues regarding the quality of the defect data collected in the defect databases of the projects, such as consistency in data collection, completeness of the defect records, problems with mapping defects onto classes, and so on, as discussed in Koru and Tian (2004).

To obtain static metrics for KOffice, a reverse engineering tool, Understand for C++ (Scientific Toolworks Inc., 2003), was used. Understand for C++ extracted static metrics from source code and provided 37 class-level static metrics essential for our purposes. The static metrics for Mozilla were obtained using the Columbus tool.[8] (Ferenc et al., 2002; FrontEndArt, 2003). The 67 class-level static metrics provided by Columbus included some size metrics (Lamb and Abounader, 1997), coupling metrics (Briand et al., 1999), cohesion metrics (Briand et al., 1998), and inheritance metrics (Chidamber and Kemerer, 1994). Since the list of the static metrics collected for KOffice and Mozilla is a long list, we made it available on-line at http://gkoru.ifsm.umbc.edu/KM/staticmetrics.html for space considerations. Both Understand for C++ and Columbus are stable enterprise measurement tools. Since the metrics are well defined metrics in the literature, both tools provided very similar results. It should be noted that the data points from these two products were not mixed to be analyzed together. Columbus needed preprocessed files to produce metrics, and as a result, it was slower. Therefore, after using Columbus for Mozilla, we switched to Understand for C++ for KOffice.

We developed a set of PERL programs that calculated the *change count* for C++ classes. *Change count* value for a particular class shows the number of times that class was changed after the product release dates until our data extraction date. The release dates for KOffice 1.1 and Mozilla 1.0 were August 29, 2001 and June 5, 2002, respectively. We ran our data extraction programs on January 30, 2005. Therefore, in this context, change-prone classes means classes that were changed more between these release and data extraction dates. The programs extracted the relevant change data from the CVS repositories of the KOffice and Mozilla projects. Only the changes in the

[7] http://www.koffice.org and http://www.mozilla.org.

[8] http://www.frontendart.com.

main CVS trunk, where the essential development and maintenance activities took place, were considered.

Change count values for classes were calculated by following the procedure below, which is given at a high level here, for KOffice and Mozilla:

(1) Identify the source code commits made from the release date to the data collection date given above.
(2) For each commit:
   (a) Find the C++ header and/or implementation files involved in the commit. For any such file f, determine the changed lines in the old revision f_{old} by executing a diff command between f_{old} and, the new revision, f_{new} (see the man page for the diff command on any Unix or Linux based system).
   (b) If f_{old} includes source code that belongs to some C++ classes, determine those classes with their boundaries (start and end lines) in f_{old}. Then, detect the changed classes by using the changed lines information obtained at Step 2 (a) and the class boundaries found at this step. If a change is detected for a class, increment the change count value for that class by one.
(3) Record *change count* values along with the static metrics for each class.

Class boundaries can change during successive revisions of a file, and the above procedure took boundary changes into account. When a class was deleted, change count was incremented by one for one last time (in Step 2(b)) for that class. The classes added after the release dates were not included in data analysis because they did not exist at the release time and they were not measured.

4. Data analysis techniques

The first step in data analysis was aimed at answering the first question explained in Section 1, namely, "what is the validity of Pareto's Law for open-source products?". At this step, the classes were sorted according to change count in decreasing order. Then, as we followed this order, the increase in the cumulative change count was observed. Note that Pareto's Law, in this context, is related to the distribution of change count over classes, therefore, only change count data at class-level was needed at this step.

To answer the second question, about the identification and characterization of change-prone classes, we used both change count and static metrics data at class level and created tree-based models. To create tree-based models, we used the rpart package in the statistical analysis tool R (R Development Core Team, 2003). Tree-based models can be understood and interpreted by developers easily, and they can serve as excellent guidance and decision support tools in software development projects where human interpretation and judgement are important (Tian, 2001). The practical and theoretical aspects of tree-based models are discussed in Breiman et al. (1984). The use of tree-based models in social sciences has been common. In the software engineering field, they were introduced by Porter and Selby (1990).

In the tree-based models of this study, the static metrics discussed in Section 3 were used as *features*. The response variable was *change count* (*cc*). While creating tree-based models, first, all of the data points in a product's data set were collected in one node called root node. Then, recursive binary splitting was applied starting from the root node for the purpose of improving *purity* in terms of *cc*. Deviance of *cc* was used as a metric of impurity. Maximum deviance reduction was obtained at each split by picking the best feature and its cut-off value among *all possible* combinations. At a node k, where the *cc* of a data point i was represented by cc_i and the mean change count by μ_{cc}, the deviance was calculated by

$$R_k = \sum_i (cc_i - \mu_{cc})^2 \tag{1}$$

Therefore, at the node k to be split into the left child, 'l', and the right child 'r', the deviance reduction,

$$\Delta = 1 - \left(\frac{R_l + R_r}{R_k}\right) \tag{2}$$

was maximized to improve purity. By doing so, the most relevant feature–cutoff value pairs were collected at the top levels of the tree.

The above procedure, when repeated continuously without any stopping criteria, usually results in a very large tree. Such a tree needs to be "pruned" because it overfits the data set, and it is usually not useful for our identification and characterization purposes. For pruning, rpart provides the minimal cost-complexity pruning detailed in Appendix B. Pruning was achieved by using the argument called complexity parameter in rpart. In the resulting model, the leaf nodes are the clusters of classes with similar characteristics and *cc* values.

The samples of such trees can be seen in Fig. 2 of Section 5.3 where we discuss the obtained models for KOffice and Mozilla projects.

5. Analysis and results

In this section, we first explain the preprocessing steps performed on the data sets. Then, we introduce the results regarding the validity of Pareto's Law for the analyzed products. Following that, the results about the identification and characterization of change-prone classes are reported. Finally, we discuss a prioritization strategy which can be used by practitioners to use limited testing and inspection resources in an efficient manner.

5.1. Preprocessing

Considering the experience gathered from our previous work (Koru and Liu, 2005; Koru and Tian, 2003; Koru

and Tian, 2005), similar studies in the literature, and our initial observations on the KOffice and Mozilla data sets, we applied the preprocessing steps given below before our analyses:

(1) Removing zero change classes: We performed our analysis on the classes that were changed at least once. We obtained the change count per class within around 3.5 and 2.5 years following the release dates for KOffice 1.1 and Mozilla 1.0, respectively. Including the classes that did not change during these long time periods in the data analysis would have introduced a considerable amount of noise and caused mischaracterizations because this category of classes largely included library classes, reused or adopted classes, or some very small classes. Therefore, we only used the data points for evolving classes in our analyses. Removing the noisy data points that can cause repeated mischaracterizations of others is a common practice in data mining as suggested by Witten and Frank (2000).
(2) Outlier analysis: Outlier analysis was necessary to produce more generalizable models as discussed in various statistical and data mining literature (Barnett and Price, 1995; Rousseeuw and Leroy, 1987; Theodoridis and Koutroumbas, 2003; Witten and Frank, 2000). We identified the classes that were outliers considering the change count and removed them from our data sets. This process was performed visually by producing box plots and sorting the classes according to their change count. We removed the outlier data points one by one until the generated models became stable. With this process, one outlier data point was removed from KOffice and two from Mozilla data sets.
(3) Removal of metrics with zero values: We removed the static metrics which took zero values across all data points because they had no contribution to the identification and characterization of change-prone classes. The values of some coupling metrics suggested by Briand et al. (1996, 1999) were all zeros as measured by Columbus. These were *AMMIC*, *DMMEC*, *FMMEC*, *IFMMIC*, *OMMEC*, and *OMMIC*. Therefore, although these metrics possessed the properties of coupling metrics stated by Briand et al., the zero values prevented us from doing further analysis with them. A similar strategy for the removal of metrics that take zero values was applied in the previous studies (Briand et al., 2000; Emam et al., 2001). Weyuker's first desirable property for metrics (Weyuker, 1988) and Tian and Zelkowitz's fifth axiom about the discriminating power of metrics also justify this action (Tian and Zelkowitz, 1995).
(4) Correlation analysis: We observed that some static metrics were very highly correlated to size as measured by lines of code. We removed the metrics whose correlation to size exceeds 0.9 from our analysis because using the actual size metric instead of those metrics resulted in more generalizable and intuitive models. In KOffice, those metrics were *CountLineBlank* and the cyclomatic complexity metrics, *SumCyclomatic*, *SumCyclomaticModified*, and *SumCyclomaticStrict*. In Mozilla, the metric weighted methods per class (*WMC*), which takes the cyclomatic complexity as weight, was very highly correlated and removed.

The excluded metrics are also indicated in the list of static metrics mentioned in Section 3 with their reasons for exclusion. As a result of the above steps, we obtained 702 data points for KOffice and 1955 data points for Mozilla.

5.2. Validating Pareto's Law

Pareto's Law as applied to the software engineering domain was introduced in Section 1 and states that a great majority of problems (e.g. defects, changes, etc.) are located in a small proportion of classes. Fig. 1 shows the resulting plots for KOffice and Mozilla, respectively. In

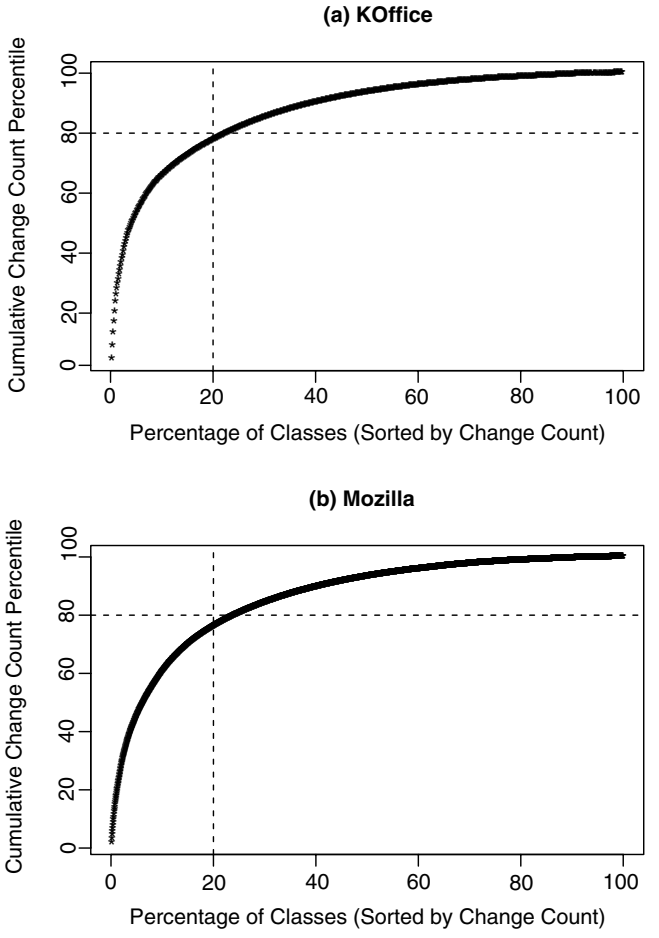

Fig. 1. Validating Pareto's Law for (a) KOffice: 702 classes and 15,948 total change count. (b) Mozilla: 1955 classes and 49,342 total change count.

these plots, the data points in KOffice and Mozilla were sorted according to the change count in descending order and aligned on the x-axis from left to right. The tick marks on the x-axis show the percentile of top-change classes. y-axis is the *cumulative* change count percentile. The dashed lines are drawn to show the intersection of the 80th percentile for cumulative change count and 20th percentile for the top-change classes.

As shown in Fig. 1, the plots for both KOffice and Mozilla show that around 80% of the total change was rooted in around 20% of the changed classes. Both curves almost meet the intersection of the dashed lines. These results strongly support Pareto's Law and provide strong evidence that a great majority of the changes are actually located in a small proportion of the changed classes in these projects. Identifying and characterizing the small proportion of problematic classes, and then focusing on them could significantly improve the quality of these products, reduce the product-release cycles, and the development and maintenance costs. These empirical results are important for companies whose business depends on making profits by developing and maintaining open-source products.

5.3. Identification and characterization results

The tree-based models obtained using the KOffice and Mozilla data sets can be seen in Fig. 2. The full names of the static metrics that appeared in these models are listed in Appendix A. In tree-based models, the highest three or four levels are usually enough to make generalizable characterizations. Therefore, the nodes at the deeper levels were pruned using the pruning mechanism explained in Appendix B.

In KOffice and Mozilla tree-based models, the size metric, lines of code, appeared at the top level, and it was used in partitioning the root node. In both models, the resulting two clusters after the first partitioning had a big difference in terms of the average change count. The classes that are larger than 2,167 lines of code in KOffice and those larger than 1711 in Mozilla were much more change-prone compared to the classes that are smaller then those cut-off values. In both models, the lines of code metric was also picked up when creating the leftmost nodes. The cut-off values were 313 and 397.5[9] for KOffice and Mozilla, respectively. The least change-prone classes had a size smaller than those cut-off values.

The leftmost nodes in our trees always had the lowest average change count and included around 76% and 82% of the data points in KOffice and Mozilla data sets, respectively. These observations are aligned with Pareto's Law which suggests that around 80% of the classes in a software system can be expected to cause minor problems. Surely, the number of data points in the leftmost node of the tree can be changed according to the pruning parameters explained in Appendix B. However, with all of the reasonable pruning parameters that result in intuitive and explanatory models, the leftmost nodes always included around 80% of the classes. They were not partitioned any more—although partitioning continued to take place in other nodes—because no further deviance reduction in change count would have been obtained by partitioning them.

This interesting observation implied that, a simple characterization of the change-prone classes could be made as those having the opposite characteristics of the classes in the leftmost leaf nodes. For example in KOffice, the classes that had more than 313 lines of code *or* those whose number of all declared methods was greater than 103.5 could be categorized as change-prone. Similarly, in Mozilla, the classes that had more than 397.5 lines of code could be categorized as change-prone.

Further characterizations were made for other leaf nodes by combining the conditions encountered along the way to the root node with a logical AND operation (&). For example, in Mozilla, as seen in Fig. 2b, there was a leaf node that contained 11 classes with the highest average change count, 425.27, among all leaf nodes. The classes in that leaf node were characterized with the logical rule below:

$$\text{LOC} \geqslant 1711 \quad \& \quad \text{OCMEC} < 5.5 \quad \& \quad \text{LCOM3} \geqslant 52.5$$

Such characterizations for the individual leaf nodes can be useful considered that average change count values in leaf nodes generally show quite variation. A prioritization strategy based on these observations is explained next.

5.4. A prioritization strategy

In our prioritization scheme, leaf nodes corresponded to priority categories for preventive actions. Leaf nodes were sorted according to their average change count in descending order. Leftmost leaves, which generally include non-risky classes as explained in Section 5.3, were excluded. In a model, the leaf node with the highest average change count became the category with the highest priority (number 1) for preventive actions. We should note that, the rightmost node in a model, which typically includes classes with the highest values of some static metrics, might not have the highest average change count (e.g. see the rightmost node in Mozilla model) as observed and reported by Koru and Tian in a previous study (Koru and Tian, 2005).

Then, we examined which ratio of classes in priority categories fell into top 20 percentile according to change count ranking. Higher ratios would show that our strategy was effective in finding top-change classes. While calculating these ratios, using all of the available data points in a data set to both create and test tree-based models would result in optimistic estimates. Therefore, we used 10-fold cross-

[9] The fractional value produced by the tool R means that the classes with 397 lines of code were sent to the left child.

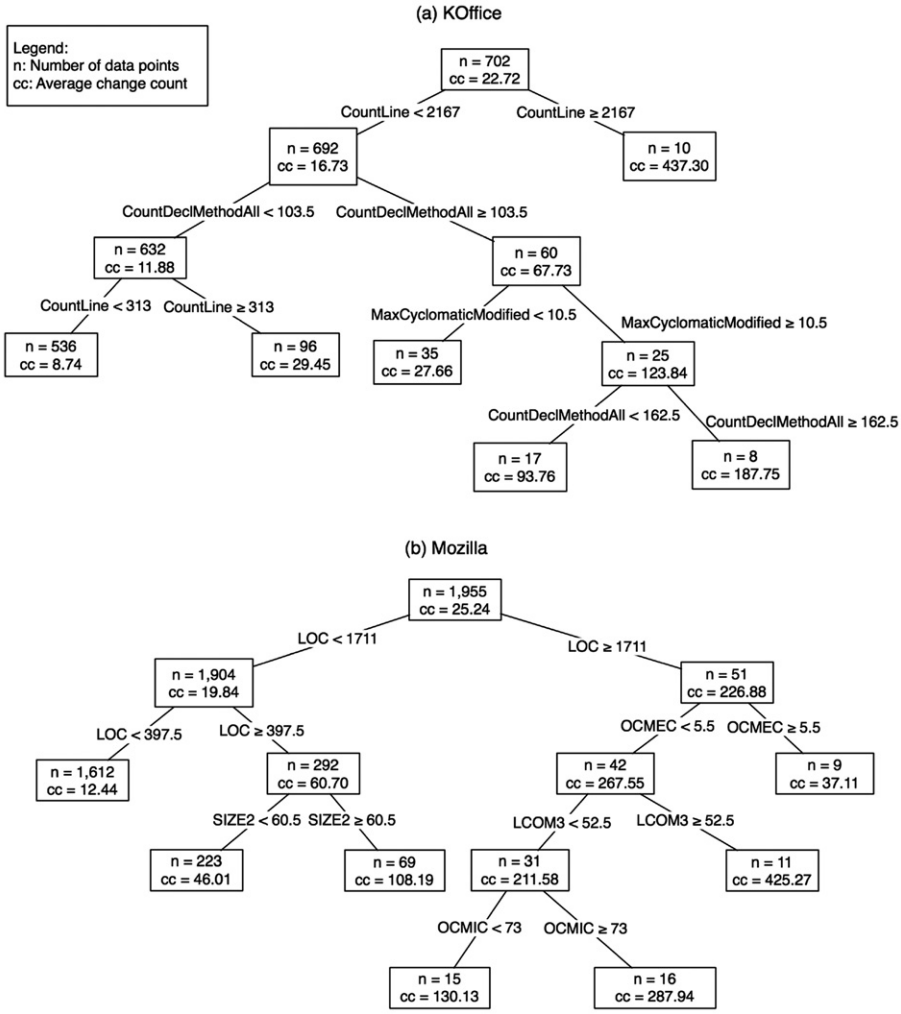

Fig. 2. Tree-based model for (a) KOffice (b) Mozilla.

validation, which is a commonly used method to obtain more realistic estimates (Witten and Frank, 2000). For each fold, 9/10th of the data points were used as the learning set to create a distinct tree-based model. The remaining 1/10th of the data points were used as the test set and placed in the leaves of the model considering their static metrics. The placement of these data points in the leaves determined their priority category—provided that they were not placed in the leftmost node. Over 10 folds, the data points that fell into different priority categories were accumulated. Then, for each one of the top six priority categories, the ratio of the classes in the top 20 percentile according to change ranking was calculated.

Normally, each 10-fold cross-validation experiment produced a slightly different set of estimates for the priority categories. Therefore, we preferred to run 100 experiments and draw box plots that summarize our overall results. For the priority categories 1–6, the box plots that show the ratios of top 20% classes in change rankings are given in Fig. 3. Higher ratios for higher priority categories depicted in Fig. 3 for both KOffice and Mozilla present strong evidence that the tree-based models can provide valuable guidance while planning the allocation of limited project resources for focused preventive actions. For example, in Mozilla, around 90% of the classes detected to be in the Priority 1 category were in the top 20 percentile according to change count ranking.

6. Conclusions and perspectives

Many companies have started to make profits and/or gain business advantages by developing and maintaining open-source software. For many application domains from operating systems to desktop applications, as an alternative to closed-source industrial software, there are equally large and complex open-source products. Currently, in the absence of systematic approaches that can guide early focused preventive actions, such as peer-reviews and testing, open-source developers are usually in a situation to deal with problems as they arise. In some projects (e.g. Debian, Kontact, etc.), when release-critical problems become intolerable and threaten project success, developers

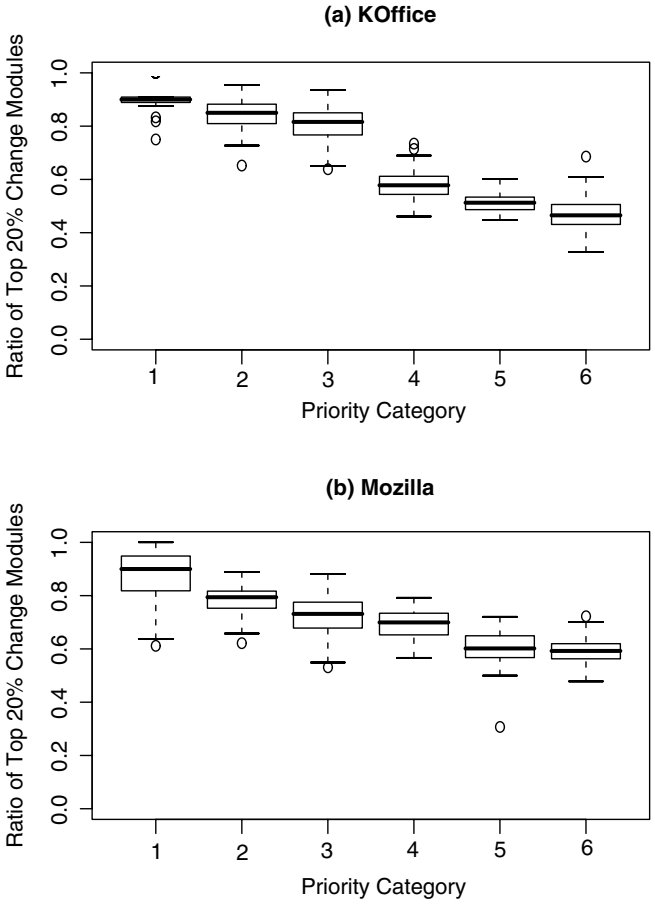

Fig. 3. The ratio of top 20% classes in leaf nodes.

organize weekend-long bug-squashing parties. During these events, they meet on-line using chat programs and try to attack problems in an ad-hoc manner.[10]

More systematic approaches that can help open-source practitioners focus on change-prone classes earlier and enable them to apply preventive actions would be useful. Currently, return-on-investment results from such efforts is not available, however, our industrial colleagues often mentioned their importance and potential benefits. In this study, as the first step, we focused on identifying and characterizing change-prone classes by using static metrics and change data collected from two large-scale and long-lived open-source products, KOffice and Mozilla. We found that:

- There is strong evidence supporting the applicability of Pareto's Law (also known as 80:20 rule) for both open-source products. When we sorted the changed classes according to their change count, we observed that those in the top 20 percentile included almost 80% of the total change count. Therefore, we concluded that these open-source projects are similar to their closed-source counterparts in this sense, and it would be a worthwhile effort to investigate how to identify and characterize the change prone classes in open-source software.
- The tree-based models obtained were easy to understand and interpret. Such models can serve as excellent guidance and decision support tools for practitioners in open-source projects because they enable practitioners to understand the characteristics of problematic classes. As a result, practitioners can focus on the classes with similar characteristics in the future releases or products. In our models, size metrics and the static metrics related to size (e.g. number of methods) were more associated to change count compared to some other static metrics. These observations are generally aligned with those made by the other researchers who performed similar studies on the closed source products and concluded that size was a good predictor of problems (Emam et al., 2001a; Lindvall, 1998).
- Some of the static metrics other than size metrics, for example coupling and cohesion metrics, also appeared in our models as secondary features. Therefore, including them in our analysis was useful for our prioritization purposes. An example was shown in Section 5.3 about the characterization made for the leaf node with the highest change count, where the metrics, OCMEC and LCOM3, played a role in addition to the size metric, LOC. More specific characterizations, such as those made with the help of the metrics other than size metrics, can be helpful for practitioners as also mentioned by Lindvall and Sandahl (1998).
- Practitioners can easily create priority categories for their focused preventive actions by using tree-based models. As shown by our data analysis, a large portion of the classes included in the high priority categories were top-change classes. Practitioners can start from the classes with the highest priority category and then proceed to those with lower priorities. Previously, Zhao and Elbaum conducted a survey about the quality assurance activities in open-source projects (Zhao and Elbaum, 2003). They found that quality assurance practices are still evolving and large projects spend considerable amount of time in testing. The prioritization scheme described in this paper can be helpful in creating a strategy for focused and effective preventive actions.
- In Fig. 2, the models for KOffice and Mozilla showed similarities as we discussed in the paper but they had some differences too. Each open-source project has a unique setting with its own process and people characteristics (e.g. skill level and expertise of developers, peer review and testing practices, etc.). Therefore, it is necessary to collect data and produce models for each individual project. A model obtained for a particular project might not be highly effective in another project. However, accumulation of similar models and results from different open-source projects over time can reveal

[10] http://people.debian.org/~vorlon/rc-bugsquashing.html.

some well-understood common patterns in the future. In that case, those models and the conclusions drawn from them can be used as a starting point for new projects or for projects that have no historical data.

Based on our findings, we suggest that open-source practitioners start to collect static metrics and change data as a part of their development and maintenance efforts. Models developed using historical static metrics and change data collected from evolving classes can be useful when prioritizing the focused preventive actions on the classes that are still under development.

Some of the necessary change data have been already collected in open-source projects, however, it was mainly collected to support essential development activities. Such data is often not in a format that allows its immediate use for research and improvement purposes. To conduct this research, we developed a set of PERL programs and scripts for data collection. However, more functionally sophisticated and usable tools can be developed by open-source communities in the future. Some source code analysis and measurement programs are already publicly available (e.g. SourceNavigator[11]). The data analysis tool, R[12], used in this paper is an open-source program too. These freely available programs can be combined to provide tool support in open-source projects.

As an immediate follow-up to this study, we plan to conduct interviews with the selected open-source developers in KOffice and Mozilla projects in order to obtain some qualitative data about the troublesome modules in their products and learn how they could utilize our approach in their projects. We believe that such qualitative data and analysis results will be useful in incorporating our approach into the management of open-source projects.

Acknowledgements

We would like to thank the anonymous reviewers of this paper for their insightful comments and suggestions. We would also like to thank Khaled El Emam, Carolyn Seaman, Anthony F. Norcio, and Chris M. Law for reading our early drafts and making recommendations.

Appendix A. Static metrics used in tree-based models

The names of the static metrics that appeared in the tree-based models in Fig. 2 and their execution are explained here. We did not use any threshold value or perform any transformations on the raw metrics values. Note that the entire list of the metrics included in the study is made available on-line:[13]

(1) Metrics for KOffice
- CountLine: Lines of code. This metric was calculated by adding up the number of all lines in the declaration and implementation portions of a class. Class declarations are included in header files (with h extension) and implementations are included in implementation files (with .cpp extension).
- CountDeclMethodAll: Number of all declared methods. In calculating this metric, all of the methods of a class was counted including all inherited methods.
- MaxCyclomaticModified: The highest modified cyclomatic complexity of the methods defined in a class.

(2) Metrics for Mozilla
- LOC: Lines of code (see the explanation above).
- OCMEC: This is the count of class-method relationships between a class and the methods of the classes that are not friend or ancestor classes. The class is being used by the methods of the other classes.
- SIZE2: Number of attributes plus number of local methods.
- LCOM3: It is a lack of cohesion metric. The number of connected components in a graph where the vertices are the methods of a class and an edge exists if two methods uses at least one attribute in common.
- OCMIC: Same as OCMEC above but the class uses the methods of other classes.

Appendix B. Pruning tree-based models

rpart performs minimal cost-complexity pruning by closely following the method suggested by Breiman et al. in Breiman et al. (1984). The cost complexity metric R_α is defined as

$$R_\alpha = R_{\text{leaves}} + \alpha * size \tag{B.1}$$

where R_{leaves} is the sum of the deviance values in leaves. *size* is measured by the number of leaves. $\alpha \geqslant 0$ is the cost complexity parameter which is the complexity cost (or complexity penalty) per leaf node for having a tree with *size*. In minimal cost-complexity pruning, for any α, the subtree that minimizes R_α is selected among all possible subtrees. For example, if the complexity cost, α, is very small, then a very large overly fit tree with $R_{\text{leaves}} = 0$ will minimize R_α. As α gets larger, the leaves will be pruned and the minimizing tree will possess fewer leaves. If α is very large, than only one tree which has only one leaf, the root node, will minimize R_α.

To obtain better estimates of R_{leaves}, rpart uses 10-fold cross-validation by default. It divides the data set into

[11] http://sourcenav.sourceforge.net.
[12] http://www.r-project.org.
[13] http://gkoru.ifsm.umbc.edu/KM/staticmetrics.html.

almost ten equal parts and uses nine of them to build a tree. Then, it drops down the data points in the tenth set onto the tree to find R_{leaves}. This process is repeated 10 times and R_{leaves} values are averaged to find the point estimate of R_{leaves}. Then, an optimum minimizing subtree of each size is identified with its corresponding α and R_{leaves}. rpart uses normalized values of α and R_{leaves} (according to R_{root}). These are *cp* (complexity parameter) calculated as α/R_{root} and *xerror* calculated as $R_{\text{leaves}}/R_{\text{root}}$. Then, the original tree can be finally pruned by the analyst by providing his preferred *cp* value. Some analysts prefer to use the *cp* value that corresponds to the minimum *xerror*, whereas, some others prefer to use the 1-Standard Error rule, and they select the *cp* value whose *xerror* is within one standard deviation of the minimum *xerror*. We applied the 1-Standard Error rule.

References

Barnett, V., Price, T., 1995. Outliers in Statistical Data. John Wiley & Sons.

Basili, V.R., Hutchens, D.H., 1983. An empirical study of a syntactic complexity family. IEEE Transactions on Software Engineering 9 (6), 664–672.

Binkley, A.B., Schach, S.R., 1998. Validation of the Coupling Dependency Metric as a predictor of run-time failures and maintenance measures. In: International Conference on Software Engineering 1998, ICSE'98. pp. 452–455.

Boehm, B.W., 1991. Software risk management: principles and practices. IEEE Software 8 (1), 32–41.

Breiman, L., Friedman, J.H., Olshen, R.A., Stone, C.J., 1984. Classification and Regression Trees. Wadsworth & Brooks.

Briand, L.C., Basili, V.R., Hetmanski, C.J., 1993. Developing interpretable models with optimized set reduction for identifying high-risk software components. IEEE Transactions on Software Engineering 19 (11), 1028–1044.

Briand, L.C., Daly, J.W., Wüst, J.K., 1998. A unified framework for cohesion measurement in object-oriented systems. Journal of Empirical Software Engineering 3 (1), 65–117.

Briand, L.C., Daly, J.W., Wüst, J.K., 1999. A unified framework for coupling measurement in object-oriented systems. IEEE Transactions on Software Engineering 25 (1), 91–121.

Briand, L.C., Morasca, S., Basili, V.R., 1996. Property-based software engineering measurement. IEEE Transactions on Software Engineering 22 (1), 68–86.

Briand, L.C., Wüst, J., Daly, J.W., Porter, D.V., 2000. Exploring the relationships between design measures and software quality in object-oriented systems. Journal of Systems and Software 51 (3), 245–273.

Card, D.N., Glass, R.L., 1990. Measuring Software Design Quality. Prentice-Hall, Englewood Cliffs, NJ.

Chidamber, S.R., Kemerer, C.F., 1994. A metrics suite for object oriented design. IEEE Transactions on Software Engineering 20 (6), 476–493.

Emam, K.E., Benlarbi, S., Goel, N., Rai, S.N., 2001a. The confounding effect of class size on the validity of object-oriented metrics. IEEE Transactions on Software Engineering 27 (7), 630–650.

Emam, K.E., Melo, W., Machado, J.C., 2001b. The prediction of faulty classes using object-oriented design metrics. Journal of Systems and Software 56 (1), 63–75.

Ferenc, R., Beszédes, A., Tarkiainen, M., Gyimóthy, T., 2002. Columbus—reverse engineering tool and schema for C++. In: Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2002). IEEE Computer Society, pp. 172–181.

FrontEndArt, Ltd., 2003. Setup and User's Guide to Columbus CAN. Hungary.

Golden, B., 2004. Succeeding with Open Source. Addison-Wesley Information Technology Series. Addison-Wesley Professional.

Henry, S., Kafura, D., 1981. Software structure metrics based on information flow. IEEE Transactions on Software Engineering 7 (5), 510–518.

Kafura, D., Henry, S., 1982. Software quality metrics based on interconnectivity. Journal of Systems and Software 2, 121–131.

Kafura, D., Reddy, G.R., 1987. The use of software complexity metrics in software maintenance. IEEE Transactions on Software Engineering 13 (3), 335–343.

Khoshgoftaar, T.M., Allen, E.B., Hudepohl, J., Aud, S., 1997. Applications of neural networks to software quality modeling of a very large telecommunications system. IEEE Transactions on Neural Networks 8 (4), 902–909.

Khoshgoftaar, T.M., Allen, E.B., Kalaichelvan, K.S., Goel, N., 1996. Early quality prediction: a case study in telecommunications. IEEE Software 13 (1), 65–71.

Khoshgoftaar, T.M., Szabo, R.M., 1996. Using neural networks to predict software faults during testing. IEEE Transactions on Reliability 45 (3), 456–462.

Kitchenham, B.A., Pickard, L.M., Linkman, S.J., 1990. An evaluation of some design metrics. IEE Software Engineering Journal 5 (1), 50–58.

Koru, A.G., Liu, H., 2005. An investigation of the effect of module size on defect prediction using static measures. In: International Workshop on Predictor Models in Software Engineering (PROMISE 2005). St. Louis, Missouri.

Koru, A.G., Tian, J., 2003. An empirical comparison and characterization of high defect and high complexity modules. Journal of Systems and Software 67 (3), 153–163.

Koru, A.G., Tian, J., 2004. Defect handling in medium and large open source projects. IEEE Software 21 (4), 54–61.

Koru, A.G., Tian, J., 2005. Comparing high change modules and modules with the highest measurement values in two large-scale open-source products. IEEE Transactions on Software Engineering 31 (8), 625–642.

Lamb, D.A., Abounader, J.R., 1997. Data Model for Object-Oriented Design Metrics.

Lehman, M., Belady, L.A., 1985. Program Evolution: Processes of Software Change. Academic Press Inc.

Li, W., Henry, S., 1993. Object-oriented metrics that predict maintainability. Journal of Systems and Software 23 (2), 111–122.

Lindvall, M., 1998. Are large C++ classes change-prone? An empirical investigation. Software–Practice and Experience 28 (15), 1551–1558.

Lindvall, M., Sandahl, K., 1998. How well do experienced software developers predict software change? Journal of Systems and Software 43 (1), 19–27.

Pareto, V., 2001. On the distribution of wealth and income, in roots of the Italian School of Economics and Finance: from Ferrara (1857) to Einaudi (1944), M. Baldassarri and P. Ciocca, (EDS.), vol. 2, Houndmills, Palgrav.

Porter, A.A., Selby, R.W., 1990. Empirically guided software development using metric-based classification trees. IEEE Software 7 (2), 46–54.

Porter, A.A., Selby, R.W., 1990. Evaluating techniques for generating metric-based classification trees. Journal of Systems and Software 12 (3), 209–218.

R Development Core Team, 2003. R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria, ISBN 3-900051-00-3. URL http://www.R-project.org.

Raymond, E.S., 1999. The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary. O'Reilly and Associates, Sebastopol, CA, USA.

Rousseeuw, P., Leroy, A., 1987. Robust Regression and Outlier Detection. John Wiley & Sons.

Scientific Toolworks, Inc., 2003. Understand for C++: User Guide and Reference Manual. 321 N. Mall Drive Suite I-201, St. George, UT 84790. URL http://www.scitools.com.

Shepperd, M., Ince, D., 1993. Derivation and Validation of Software Metrics. Clarendon Press, Oxford, Oxford University Press, Oxford.

Theodoridis, S., Koutroumbas, K., 2003. Pattern Recognition. Academic Press.

Tian, J., 2001. Quality assurance alternatives and techniques: a defect-based survey and analysis. Software Quality Professional 3 (3), 6–18.

Tian, J., Nguyen, A., Allen, C., Appan, R., 2001. Experience with identifying and characterizing problem prone modules in telecommunication software systems. Journal of Systems and Software 57 (3), 207–215.

Tian, J., Troster, J., 1998. A comparison of measurement and defect characteristics of new and legacy software systems. Journal of Systems and Software 44 (2), 135–146.

Tian, J., Zelkowitz, M.V., 1995. Complexity measure evaluation and selection. IEEE Transactions on Software Engineering 21 (8), 641–650.

Weyuker, E.J., 1988. Evaluating software complexity measures. IEEE Transactions on Software Engineering 14 (9), 1357–1365.

Witten, I.H., Frank, E., 2000. Data Mining: Practical Machine Learning Tools with Java Implementations. Morgan Kaufmann, San Francisco.

Zhao, L., Elbaum, S., 2003. Quality assurance under the open source development model. Journal of Systems and Software 66 (1), 65–75.