Software Defect Prediction Based on Source Code Stabilization Model

Semyon Igorevich Kirnosenko, Vitaly Alekseevich Egunov, Andrey Evgenevich Andreev and Dmitriy Nikolaevich Zharikov

Volgograd State Technical University, Volgograd, Russia

Submitted: Aug 5, 2013; Accepted: Sep 15, 2013; Published: Sep 25, 2013

Abstract: The ability to predict which files in a software system are most likely to contain the largest numbers of faults in the next release can be a very valuable asset for quality increasing before release. There are many special prediction methods for such purpose. Most of them operate with black-box calculation models that do not give interpretation of actions during prediction process. We propose a source code stabilization model. It allows to describe programming process in form of theory of probability task and to estimate probability of presence of logical errors in various source code fragments. We show applicability of this model for building defect prediction model. The results of experiments show that such defect prediction model may be accurate enough for practical usage in industrial environment. We also consider some possible ways of model improvements in context of changes they require.

Key words: Defects • Defect prediction • Quality • Metrics • Version control systems • Source code

INTRODUCTION

Software Defect Prediction: Under conditions of iterative complex software development, coding, testing and operation stages are often either aligned or successive. Defect detecting and correcting process is also continuous. Under these conditions, methods for predicting defects allow us to estimate the defects distribution between the individual software components. Subsequently, this information can be used to locate resources by testing those components, in which the probability of having undetected defects is highest. This allows you to more effectively organize the testing process and, as a consequence, reduce the time and effort to it.

There are many methods for predicting defects. All of them, in one way or another, based on the information about previously discovered defects use. The first defect predicting methods that have received wide application became methods, based on reliability growth models [1]. The first of these methods was proposed in the seventies of the last century and has since gained considerable experience in the use of such methods in academic and in industrial areas. The reliability growth calculation model basis is the use of a reliability function, which is a monotonically increasing function for which the abscissa axis is time and the ordinate axis is the number of defects, detected to the appropriate point in time. Usually to get this function, well-known distribution law is selected, such as exponential and its parameters are estimated based on previous experience and current information on the detected defects. By using this distribution we can estimate the number of undetected defects. Reliability growth models are quite simple, but based on a number of highly questionable assumptions. Also, due to such model peculiarities, its application to iteratively developing software often either difficult or even impossible.

A number of studies refer to the Pareto principle observance fact, when about 20% of the system components will comprise about 80% of undetected defects [2]. In contrast to methods based on reliability growth models, statistical techniques are well suited to projects being developed in accordance with the iterative approach. The disadvantages of these methods include the fact that they require the metrics set measurement and sometimes expert estimates. In addition, many of these methods, exploiting the "black box" model, do not give a clear interpretation of the processes performed in predictions, which contributes to the assumptions, they are based on, concealment [3].

Corresponding Author: Kirnosenko, Volgograd State Technical University, Lenin Avenue, 28, 400005, Volgograd, Russia.
Let the distribution of defective lines in terms of the writing a single line of source code is an individual source code line will be considered consequence of which would be a software defect that is applied. When writing a program, in particular on each ability to manage them separately, version control system having defects in a source code subset: To combine these pieces into a coherent whole and the code added with certain fragment. The probability of process, the source code fragments added gradually. of source code subsets, each of which contains source code does not appear immediately. It is the product of source code can be distributed among a certain set of sequence of special form lines. Moreover, all the program means the individual files. For each fragment, added continuous operation, perhaps a lot of people. In the components. As a result, each component comprises a set code, eliminating the identified defects. The software creating process in fact it is to write the source code in some programming language. This process can be methodologically difficult organized and it can involve a lot of people doing different types of work: design, coding, testing, etc. However, the coding is the only activity form in software development, cause-and-effect relationship of which, with the defects presence fact in the final product is not difficult to formalize. In addition, all other forms of activity ultimately find expression in the same coding. Since the results of the design may be the terms of reference, specifications, diagrams, UML, but in the future, these materials will be used to write the source code in accordance with the requirements. Similarly, the test results can be found defects and their causes, but again, this information will be used to write the source code, eliminating the identified defects.

Considering the encoding process, it can be said that in the course programmer write a source code in the form of text on using programming language. The program is a sequence of special form lines. Moreover, all the program code does not appear immediately. It is the product of continuous operation, perhaps a lot of people. In the process, the source code fragments added gradually. To combine these pieces into a coherent whole and the ability to manage them separately, version control system is applied. When writing a program, in particular on each line, the programmer can make a logical error, the consequence of which would be a software defect that manifests itself in incorrect operation of the program under certain conditions. Let us formulate a number of assumptions on which to base our conclusions:

- Each individual source code line will be considered as defective if corrections of existing defects in software require this line modification and non-faulty line otherwise.
- Let writing a single line of source code is an independent test with some unknown probability of success $P_{LS}$ and the probability of failure $P_{LF} = 1 - P_{LS}$. Success is writing a non-faulty source code line and the failure is writing a faulty source code line.
- Let the distribution of defective lines in terms of the source code at regular intervals.
- Let the new source code addition does not affect the defect probability in the previously added code.

Then, the probability that when adding a new source code fragment of volume $LOC_{x}$ lines, defects have been introduced is the probability of presence in this source code volume at least one defective line:

$$P(D_{xt}) = 1 - (1 - P_{LS})^{LOC_{x}}$$

where $D_{xt}$ - event consisting in the presence of defects in the newly added source code fragments of volume $LOC_{d}$ lines, the cause of which is a natural defect density as an inevitable coding errors logic consequence.

If all the source code contain $M$ fragments, then the probability that it contains defects, according with the joint events probability addition theorem, is calculated by the following recursive formula:

$$P(D_{xt}) = (P(D_{xt})_{m} - P(D_{xt})_{m+1} + P(D_{xt})_{m}) / (P(D_{xt})_{m-1} - P(D_{xt})_{m})$$

where $P(D_{xt})_{m}$ - the probability of having defects in the software, calculated taking into account the $m$ fragments; $P(D_{xt})_{m}$ - the probability of having defects in the $m$-th fragment in accordance with formula 1.

Besides fragments, the source code is also distributed among the components by which usually means the individual files. For each fragment, added source code can be distributed among a certain set of components. As a result, each component comprises a set of source code subsets, each of which contains source code added with certain fragment. The probability of having defects in a source code subset:

$$P(D_{xt}) = P(D_{xt}) \frac{LOC_{xt}}{LOC_{xt}}$$

where $D_{xt}$ - the event consisting in the defect presence of a source code subset; $P(D_{xt})$ - the having defects probability in the fragment, which was added to the
source code of the relevant subsets; \( LOC_{\text{init}} \) - the source code lines number added to this subset; \( LOC_{\text{add}} \) - the source code lines number, added to the fragment.

If the component source code contains \( S \) subsets, then the probability that a component is defective according the joint events probability addition theorem is calculated by the following recursive formula:

\[
P(D_{AC_c})_s = (P(D_{AC_c})_{s-1}) + P(D_{AC_c})_s - (P(D_{AC_c})_{s-1}) \cdot P(D_{AC_c})_s,
\]

(4)

where \( P(D_{AC_c})_s \) - the defect presence probability, calculated taking into account the \( s \) source code subsets that make up the component; \( P(D_{Ac})_s \) - the defect presence probability in the source code \( s \)-th subset, according to formula 3.

The above formula does not take into account the fact that the source code is constantly introducing corrections and changes that affect the probability of having undetected defects in any subset of the source code. To take this into account we can try to estimate the probability that existing in the source code defects were left in it to a certain point in time \( t \), measured from the starting development point, which is the moment of entering into the version control system the first source code fragment.

Let us consider the original code of a single fragment. Separate fragment source code structural is much more complicated because contains the source code of the different fragments. Therefore, as a component in introducing a new source code and corrections previously made defects, defect density, including observed for the source code component is always changing: rises and falls. Source code fragment, once added, can not be changed. It can be partially or completely removed, but may be supplemented with a new source code, since new source code will already be part of new fragments. As a consequence, the unknown total defect density for separate fragment source code remains constant, but known, measured, increases monotonically, approaching to full as in the source code fragment defects are detected and corrected.

In accordance with the above assumptions, the chance of having undetected defects in the previously added source code is only possible as a consequence of the removal of this code in part or in full. There are two reasons for the source code removal: refactoring and defects correction. Again, relying on the assumptions made, we can say that in proportion to the removed code volume decreases the probability of having undetected defects in the code remained, since it is likely to accidentally delete a faulty code. Let us suppose that initially to the fragment \( LOC_{\text{add}} \), lines of source code were added and later from this source has been removed \( LOC_{\text{add}} \) lines in the result of refactoring by the moment of time \( t \). Then, the probability that a change in existing undetected defects were not completely removed as a result of refactoring by the time \( t \), can be calculated as:

\[
P(D_{R_M}(t)) = \frac{LOC_{\text{add}} - LOC_{\text{add}}(t)}{LOC_{\text{add}}},
\]

(5)

Let us suppose that initially to the fragment \( LOC_{\text{add}} \), lines of source code were added and later from this source has been removed \( LOC_{\text{add}} \), lines in the result of defect correction and \( LOC_{\text{add}} \), lines as a result of refactoring by the moment of time \( t \). If in the source code fragment remained undetected defects, which means that the total number of defective lines in the initial code was larger than \( LOC_{\text{add}} \). Of course, the total number of defective lines of initial source code cannot be greater than \( LOC_{\text{add}} \). And given that the source code can be removed not only from the bug fixes, but also as a result of refactoring, the total number of defective lines of source code cannot be greater than \( LOC_{\text{add}} - LOC_{\text{add}}(t) \). Consequently, the probability that the available source code fragment defects were not completely corrected at time \( t \) can be calculated as the probability that the total number of defective rows source was greater than number \( LOC_{\text{add}}(t) \) and not greater than \( LOC_{\text{add}} - LOC_{\text{add}}(t) \). This probability can be calculated using Laplace’s integral theorem [5]:

\[
P(D_{FG}(t)) = \frac{1}{\sqrt{2\pi}} \int_{x}^{x'} e^{-x'^2/2} \, dx',
\]

(6)

where,

\[
x = \frac{b}{p}, \quad x' = \frac{b}{p}, \quad b = b_1 = LOC_{FG}(t) + LOC_{AC} - LOC_{FG}(t) + LOC_{AC}, \quad \text{and} \quad p = LOC_{FG} + LOC_{AC - FG}(t)
\]

the event consists in the fact that undetected defects, presented in a source code fragment, were not completely corrected by the time \( t \).

So, in the source code fragment operation, the undetected defect presence probability may affect the two events: removal of the source code by refactoring and as a result of bug fixes. Considering these events independent, we can calculate the probability that found in the source code fragment defects have been removed or corrected by the time \( t \):

\[
P(D_{SC}(t)) = P(D_{AC}(t)) \cdot P(D_{FG}(t)).
\]

(7)
Performing components classification. The result of searching the defective source code. Carried out by calculating the probabilities of finding undetected refactoring methods can be applied for finding the minimum removed by the time t as a result of defects correction and containing undetected defects. Thus sampling fragment there were defects; these defects have not been prediction is component sample that is classified as containing undetected defects at time t is calculated as: identifying correcting fragments in the version software systems some of its versions were considered. Therefore, the model can be used to construct a method for forecasting the defects, which will include the following steps:

- Searching the defective source code. Carried out by identifying correcting fragments in the version control system [6].
- Calculating the probabilities of finding undetected defects on components of the software in accordance with the proposed model.
- Performing components classification. The result of prediction is component sample that is classified as containing undetected defects. Thus sampling methods can be applied for finding the minimum number of components containing the maximum number of undetected defects [2], or a different approach.

It remains to note that for the described above model application it is required an estimate of the introducing defects probability when writing a single line of source code $P_{LF}$. For some time moment t we can simply estimate from the total number of added source code lines $LOC_{AT}$ and the total number of source code lines, proved to be defective $LOC_{LF}$ by the time t. Then the statistical estimate of the required probability is calculated as the relative frequency of defective source code lines:

$$P_{LF} = W_{LF}(t) = \frac{LOC_{LF}(t)}{LOC_{AT}}$$  \hspace{1cm} (11)

**Experiment:** To verify the method prediction accuracy, based on proposed model, experiment was conducted in which all components of considered software were classified as containing undetected defects, or as not having any. For the experiments, the source code of 7 different software systems with open source code was taken. These included the following: django (www.djangoproject.com/, a platform for web-based application), gnome-terminal (www.gnome.org/, terminal emulator), gnuplot (www.gnuplot.info/charting), httpd (httpd.apache.org/, http-server), nhibernate (nhforge.org/, ORM for .NET platform), pgadmin3 (www.pgadmin.org/, PostgreSQL database administration), wordpress (wordpress.org/, content management system). Selection of these software systems is due, above all, by considerations of obtaining sample data, which could be considered representative on various aspects. The selected software systems have different meanings, are implemented using different programming languages and methodologies, have different size of the source code, the number of active developers, the development period, the number of versions, etc.

The experimental procedure was as follows. For each software systems some of its versions were considered. For each version forecast was carried out and forecast...
accuracy was tested to obtain a number of accuracy characteristics. Then, for each characteristic average values were taken at its sample for all the separate versions. These values are given as the experimental results for the separate software systems. Using the method based on the proposed model prediction was implemented, the result of which was a sample of system components, which are in line with the forecast contain undetected defects. Then we searched the defective source code for the version under consideration by the previously proposed methods [6]. In this case search was carried out not only for the latest source code version, but also the source of all previous versions. As the result we get a selection of components that actually contained defects found and corrected after the considered version release. Thus we obtained two samples of software components: components do contain defects at the time of considered versions issuance and components containing defects at the time of release version under consideration in accordance with this outlook. On these samples basis, the prediction accuracy calculation was performed. The experimental results are shown in Table 1.

Table 1: Experimental results

<table>
<thead>
<tr>
<th>Software</th>
<th>P</th>
<th>R</th>
<th>A</th>
<th>AUC</th>
<th>Neg/Pos</th>
</tr>
</thead>
<tbody>
<tr>
<td>Django</td>
<td>0.80</td>
<td>0.54</td>
<td>0.70</td>
<td>0.82</td>
<td>1.09</td>
</tr>
<tr>
<td>Gnome-terminal</td>
<td>0.75</td>
<td>0.48</td>
<td>0.68</td>
<td>0.78</td>
<td>1.31</td>
</tr>
<tr>
<td>Gnuplot</td>
<td>0.61</td>
<td>0.70</td>
<td>0.74</td>
<td>0.82</td>
<td>2.36</td>
</tr>
<tr>
<td>Htpd</td>
<td>0.54</td>
<td>0.67</td>
<td>0.63</td>
<td>0.81</td>
<td>2.36</td>
</tr>
<tr>
<td>Nhibernate</td>
<td>0.36</td>
<td>0.33</td>
<td>0.81</td>
<td>0.70</td>
<td>9.72</td>
</tr>
<tr>
<td>Pgadmin3</td>
<td>0.67</td>
<td>0.61</td>
<td>0.64</td>
<td>0.70</td>
<td>1.75</td>
</tr>
<tr>
<td>Wordpress</td>
<td>0.91</td>
<td>0.58</td>
<td>0.62</td>
<td>0.74</td>
<td>0.29</td>
</tr>
</tbody>
</table>

CONCLUSIONS

Proposed source code stabilization model allows estimating the defects presence probability in various fragments of source code. The accuracy of such estimates is sufficient for them to perform prediction based on the defects presence in the software components. In turn, the accuracy of these predictions is sufficient for practical use in an industrial environment. At the same time, the resulting prediction accuracy can not be considered high, but this is largely due to several model simplifications from which to improve accuracy we can move. For example, estimate of the source code defective line writing probability \( P_{def} \), may be performed not only for the source code and for some of its fragments. This way you can consider various factors affecting this probability: source code complexity, developer experience and more. This has a positive effect on the prediction accuracy which has already received experimental confirmation. Revision of the model same assumptions is possible, but less promising direction of development. While all of these assumptions except the last one, in practice almost never met, they still allow you to create a minimum basis for further conclusions. Almost all of the potential ways of clarifying these assumptions, such as the source code faulty lines distribution assessment are associated with the need to perform source code static analysis. This means the need to implement such for each programming language, with which we want to deal with, that, in general, significantly narrows the model applicability scope. It should also be noted that the proposed model and the forecasting method, based on it, work without the need for expert evaluation or measurement of a large metrics number. This distinguishes them from most analogues.

All operations, necessary for the metrics calculation and defect forecasting were performed using a specially developed for this study software MSR Tools, which is available on the Internet (msr.sourceforge.net) along with the source code, is licensed under BSD.
REFERENCES


