Implementation of UML2.0 Based Change Proneness Prediction in OO Software through Dependency

Deepa Godara\textsuperscript{a}\textsuperscript{*} and R.K. Singh\textsuperscript{b}

\textsuperscript{a}CSE, Uttarakhand Technical University, Dehradun, India
\textsuperscript{b}ECE, Uttarakhand Technical University, Dehradun, India

Accepted 01 October 2014, Available online 20 October 2014, \textit{Vol.4, No.6 (October 2014)}

Abstract

Predicting change prone class in software is a difficult software engineering process. In order to keep up with the pace of modern day expansion, change in any software is inevitable. Software enhancement and defects are two main reasons for software change. In the earlier research, of predicting change prone classes the stress was given only on static characteristics rather than dynamic characteristics. In this paper we aim to predict change prone classes using execution time of methods, trace events and dependencies between classes. Software prediction models based on these results can help us identify change prone classes of software which would lead to more rigorous testing and better results. We evaluate dependency between classes using UML2.0 sequence and class diagrams. Further, the results are obtained using a hospital application and implementing in Java.

Keywords: Change prone, class dependency, UML2.0 sequence diagram, UML2.0 class diagrams

1. Introduction

When adapting a system to new-fangled treatment patterns or technologies, it is essential to predict what such version of architectural plan entails in terms of system excellence (Aida Omerovic et al., 2010). Software systems are incessantly subjected to modification. This is all the more essential for addition of novel traits, to become accustomed to contemporary environment, to put right bugs or to re-factor the source code (Daniele Romano et al., 2011). Modification may be on account of diverse factors like improvement, modification, perfect upkeep or do away with drawbacks. Several elements of the software may be susceptible to modifications than their counterparts. A proper understanding of the classes which are change-prone is highly advantageous as the change-proneness may be some sign of particular fundamental quality issues (James M. Bieman et al., 2003). Managing change is one of the pivotal factors in the realm of software engineering. Evolutionary growth has been suggested as a competent method to tackle risks like modern technology and vague or varying needs (Erik Arisholm et al., 2000). If a preservation method has the competence to ascertain the components of the software which are change-prone then explicit corrective steps can be initiated. Therefore, a sound knowledge of the domain which had maximum modification over a certain interval will go a long way in spotting the crucial change-prone classes and interactions, then it is easy for development procedure to concentrate its attention on them (James M. Bieman et al., 2003).

\textsuperscript{*}Corresponding author: Deepa Godara

Even though the failure paths of several software development projects take divergent routes, all of them converge at one place while sharing general indications.

These include imprecise comprehension of end-user requirements, incapacity to tackle changing requirements, complicated software which is not easily preserve, delayed recognition of grave scheme defects, irresistible intricacy, plan and execution, unrestrained modification spread or inadequate testing (Mario Kušek et al., 2001). A small change can have considerable and unexpected effects on the system (M.K. Abdi et al., 2009). Change-prone classes in software call for meticulous attention as they need endeavor and augment growth and preservation overheads.

Recognizing and typifying them enables developers to take remedial measures like peer-reviews, testing, inspections, and streamlining endeavors on the classes with the parallel traits in the coming years. Consequently, developers are capable of exploiting their wherewithal more professionally and dispense superior quality products in an appropriate way (A. Güneş Koru et al., 2007). If defective classes are recognized in the initial stages of the growth project’s life phase, extenuating remedies can be considered including alert inspections. Forecast patterns by means of plan metrics can be employed to recognize defective classes in advanced stages (Daniela Glasberg et al., 2000)]. The accuracy of the forecast effect decides the accuracy of cost evaluation and quality of project preparation (Mikael Lindvall, 1999).

We are of the opinion that a lion’s share of the software metrics appraise the grade of object-orientation or calculate immobile traits of the plan, which do not appear to be advantageous in finding a key to the
perplexing dilemma regarding the existence or otherwise of high-quality in regard to a explicit plan. While attempting to face such an issue, an expert would evaluate the conformance of the plan to well recognized rules of thumb, heuristics, and doctrines (Nikolaos Tsantalis et al., 2005). Behavioral Dependency Analysis (BDA) estimates the scope to which the functionality of one system unit is reliant on other units. According to the mine of data used to execute a BDA, we can segregate the BDA methods into three groups such as code-based, execution-trace-based, and model-based. To obtain behavioral reliance measures between two disseminated objects, we undertake a methodical scrutiny of messages sent between them in a group of sequence diagrams (SDs) For example, when an object sends a synchronous communication to another object and waits for a reply, we say that the former object is behaviorally reliant on the latter (Vahid Garousi et al., 2006).

UML is now extensively acknowledged in the software engineering circle as a general notational benchmark. It extends a helping hand to object-oriented plans which on the other hand promote module reprocess. It is competent to furnish numerous outlooks of the system under blueprint (Kathy Dang Nguyen et al., 2007). The UML based plan enables us to execute prescribed authentication and corroboration method (András Pataricza et al., 2003). The unified modeling language (UML) is a graphical language for visualizing, denoting, building, and recording software-intensive techniques. UML offers a typical method of scripting system plans, covering abstract things, classes written in a definite programming language, database schemes and reusable software components (Mario Kušek et al., 2001). UML has appeared assuming the role of the software industry’s leading language and is, by now, an Object Management Group (OMG) benchmark. It symbolizes a set of finest engineering exercises that have been established as triumphant in the modeling of mega and intricate techniques. OMG is now recommending the UML pattern for global homogeny for information technology (Kleanthis C. Thramboulidis, 2001). As the application of object-oriented plan and programming grows up in the industry, we see that legacy and polymorphism are being utilized more often to perk up internal reprocess in a system and to assure conservation (Erik Arisholm et al., 2004).

The rest of the paper is organized as follows: Section gives overview of related work. Section 3 briefly describes UML diagrams and the feature class dependency used to predict change proneness. Section 4 describes the feature execution time and trace events for change proneness prediction. Section 5 gives results and discussions. Section 6 concludes and outlines future research.

2. Related Work

In the course of the growth and preservation of object-oriented (OO) software, the data on the classes which are more prone to change is highly advantageous. Developers and maintainers are able to create further adaptable software by changing the segment of classes which are susceptible to modifications. Conventionally, nearly all change-proneness forecast has been investigated according to source codes. Nevertheless, change-proneness forecast in the initial stage of software growth can offer an easier method for evolving durable software by changing the existing plan or selecting substitute plans prior to execution. To tackle this requirement, (Ah-Rim Han et al., 2008) have offered an innovative and a systematic method for estimating the behavioral dependency measure (BDM) which enables proper forecast of change-proneness in UML 2.0 brand.

(Ali R. Sharafat et al., 2008) have come out with a novel method to forecast modifications in an object-oriented software mechanism. The key dilemma in software growth procedure is to evolve inaccuracy recognition to initial stages of the software life span. With this end in view, the Verification and Validation (V&V) of UML diagrams undertake a very significant function in identifying defects at the plan stage itself. It has a discrete relevance for software safety, where it is highly essential to spot safety faults before they can be subjugated. (V. Lima et al., 2009) have played a vital role in this regard by offering a formal V&V method for one of the most admired UML diagrams viz. sequence diagrams. (Mehdi Amoui et al., 2009) have established that an awareness of probable time of occurrence of modifications will motivate managers and developers to design their preservation functions with superior proficiency. (Malan V. Gaikwad et al., 2011) have the credit of introducing a novel a method of employing class hierarchy technique which is easily comprehend-able and executable. (Malan V. Gaikwad et al., 2011) have intelligent focused on locating reliance of software that may be obtained by assessing the proneness of Object Oriented Software.

Recognizing change-prone classes enables developers to devote further interest to classes with parallel traits in the future and thus investigation resources and time can be utilized more efficiently. (Xiaoyan Zhu et al., 2013) have gathered a group of static metrics and modification data at class level from an open-source software product, Dacarow. Their test outcomes have shown beyond doubt that their categorization outcomes are functional for recognizing change-prone classes and therefore can extend a helping hand to the developers to perk up their efficiency. Moreover, (Emanuel Giger et al., 2012) have presented a paper for capturing the fine-grained Source Code Changes (SCC) and their semantics and also (Ali R. Sharafat et al., 2007) have proposed a novel method for the prediction of changes in object oriented software system, in which the quality aspects were qualified by the probability of change in each class.

3. UML Diagrams and Class dependency

UML Diagrams are one of the diagrams supporting our work, from which we can compute the features of the application. UML (Unified Modeling Language) is a general-purpose modeling language, which helps to represent the structure of complex software in a visual form, and utilize in software engineering. UML diagrams provide communication between the customer, system analysts and programmers, who write the source code. So, it acts as a mediator or information provider to find out the features. In general, 8 kinds of UML diagrams are
generated. But for our purpose we need only 2 kinds of UML diagrams – (i) Sequence Diagram (ii) Class diagram.

**UML Sequence Diagram:** - Sequence diagram is a kind of interaction diagram, which represents a sequence of interaction between the objects.

**UML Class Diagram:** - They help to create graphical logical models of a system, further used to create the source code for the classes represented on the diagram. Class diagrams represent the relationship between classes and interfaces.

Class Dependency is one of the important features to predict the change proneness. In a source code, if one class gets changes, it also affects the other class. It can be found only through the dependencies between the classes or objects.

The relationship between the sender object and receiver object is an example of the class dependency, while sending a message between two objects. The changes in the class of the receiver object also affect the class of the sender object. The inheritance and polymorphism are also taken into account, during the measurement of class dependency. The higher changes in class dependency indicate the possibility of more changes to happen. Two kinds of class dependencies are:

**Direct Class Dependency:** - Consider two objects O1 and O2. If O1 wants services to be get from O2, then a synchronous message is sent to O2 and O1 waits for a reply that received from O1. This kind of dependency is called as direct class dependency. This is denoted as, \( O_1 \rightarrow O_2 \).

**Indirect Class Dependency:** - Consider n objects \( O_1, O_2, O_3, ..., O_n \). Indirect dependency between the objects \( O_1 \) and \( O_2 \) is denoted as, \( O_1 \Rightarrow O_2 \), except \( n=2 \) that represents direct class dependency \( O_1 \rightarrow O_2 \). Because the indirect class dependency is represented as \( (O_1 \rightarrow O_2) \Rightarrow (O_1 \rightarrow O_3) \Rightarrow ... \Rightarrow (O_1 \rightarrow O_n) \). Here, \( \Rightarrow \) indicates the External service request relation. For example, if an object \( O_1 \) needs a service from the object \( O_2 \) through \( O_3 \), then it is indicated as \( (O_1 \rightarrow O_2) \Rightarrow (O_2 \rightarrow O_3) \).

**Fig. 1:** General structure of Sequence Diagrams (a) SD-1 (b) SD-2

A synchronous message has the dependency between the sender and receiver objects, since the sender object depends on receiver object by waiting for the reply from the receiver object. It also indicates that the reply from the receiver object affects the sender object. But an asynchronous message does not have the dependency between the sender and receiver objects, since the sender object does not wait for the reply from the receiver object, its process continues. We have to compute the class dependency as a feature for our work, so we consider only the synchronous messages. The calculation of class dependency states more the class dependency more the changes in object oriented software.

### 3.1 Measuring class dependency

To measure the class dependency from the source code, we generate a UML Sequence and Class diagram for the source codes. From this, we then measure the class dependency by using the following methods.

(i) Construct Dependency model of Object

(ii) Construct Dependency model of system

(iii) Form reachable path table

(iv) Calculate the weighted sum of reachable paths

(v) Calculate the Class Dependency

i) Dependency model of Object: messages are exchanged between instances of classes. Each message is a combination of three parts: reverse traceable message this is valid incase of indirect dependency, probabilistic execution (The probabilistic execution rate of a message is a probability of execution rate of a message in an alt combined fragment of a sequence diagram) and expected execution rate (probability of the execution rate of a sequence diagram). 

ii) Dependency model of system: We need to find the dependency for the whole input application of source code. For this purpose, all these separated dependencies are combined together to build one big System Dependency Model to find the class dependency feature.

iii) Reachable path table: The paths between each pair of objects in the system dependency model are traversed from the source object to destination object and the paths are tabulated in a reachable path table. To find the reachable paths, traversal starts from a message incoming to the destination object to a message outgoing from the source object in reverse. These messages that are found via traversal are added in the reachable path table. If the source and destination objects have direct dependency between them, then the name of the incoming message to the destination object and the name of the outgoing message from the source object are equal. So, only one message name is included in the reachable path table for the direct dependency objects. But for the indirect dependency objects, the traversal from the incoming message to the destination object is carried out by iteratively substituting it with a backward navigable message and then we can reach the outgoing messages from the source object.

iv) weighted sum of reachable paths: Sum of reachable paths can be calculated as:

\[
Sum = \sum_{i=1}^{N} 1 \cdot F_{rez} \cdot F_{exe}
\]

where, \( N \) - Number of messages in the respective reachable path 

\( F_{rez} \) - Probabilistic execution rate of first message in the reachable path

\( F_{exe} \) - Expected execution rate of first message in the reachable path
v) Class dependency: The Dependency feature for a particular class $C_i$, is calculated for getting the reachable path by summing the pair of instance of the corresponding class $(C_i, C_j)$, where, $C_i, 1 \leq j \leq n$, and $n$ is the total number of classes.

4. Execution Time and Trace Events

This is the next feature used in predicting change prone class. From the input source code, we can find this feature value, time. In source code, many classes and methods are presented. For a class, more number of methods is included in it. Each will be called by other classes also. To get the time feature, we have to calculate the execution time of each methods of each class.

Third feature that facilitates the prediction of change prone class is trace events. Some of the methods in the source code are not executed in the run time. During runtime of the source code, we have to identify the following things,

- What are the methods executed during the runtime?
- How many times, these methods are executed during the runtime?

Using these details, the feature trace event is computed directly from a given application.

5. Results and Discussions

Our proposed work for predicting change proneness is implemented using Java. Hospital Management application is given as the input for our proposed work. In this application, a large number of classes and methods are presented. These are taken for the process of our proposed work.

Table I: Output for time and trace events

<table>
<thead>
<tr>
<th>Classes</th>
<th>Feature - Time</th>
<th>Feature – Trace Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>a11</td>
<td>0.008</td>
<td>3</td>
</tr>
<tr>
<td>a12</td>
<td>0.001</td>
<td>3</td>
</tr>
<tr>
<td>a13</td>
<td>0.001</td>
<td>3</td>
</tr>
<tr>
<td>a20</td>
<td>0.0</td>
<td>2</td>
</tr>
<tr>
<td>Eight</td>
<td>0.001</td>
<td>1</td>
</tr>
<tr>
<td>Fifth</td>
<td>0.01</td>
<td>32</td>
</tr>
</tbody>
</table>

Fig. 2 UML sequence diagram generated for our proposed work

The next feature behavioral dependency is generated by converting the input into UML sequence and class diagram and then the feature is identified by constructing a model of OBDM and SOBDM from these UML diagrams. The sample UML diagrams that are generated from the input application are given below in fig2.

Table II: Output for class dependency feature values

<table>
<thead>
<tr>
<th>Classes</th>
<th>Feature – Class Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>a11</td>
<td>2.690476</td>
</tr>
<tr>
<td>a12</td>
<td>0.896825</td>
</tr>
<tr>
<td>a13</td>
<td>2.690476</td>
</tr>
<tr>
<td>a20</td>
<td>0.309523</td>
</tr>
<tr>
<td>Eight</td>
<td>0.246031</td>
</tr>
<tr>
<td>Fifth</td>
<td>0.198412</td>
</tr>
</tbody>
</table>

Table II shows the result class dependency calculate using UML diagrams

Conclusion and Future Research

We have proposed a method to predict change prone values using dynamic characteristics of object oriented software such as dependencies between classes, execution time of each method and trace events.

In future we will try to inculcate more features to increase the accuracy of change proneness prediction.

References


