Assessment of Maintenance Activities of Object-Oriented Software using Ripple Effect Analysis

Raghuraj Singh, Prabhat Verma, Isha Garg
Computer Science and Engineering Department
Harcourt Butler Technological Institute Kanpur, India

Vibhash Yadav
Computer Science and Engineering Department
Krishna Girls Engineering College Kanpur, India

Abstract—Maintenance activities cost about eighty percentage of the total project life cycle. But when we make changes at one place, changes propagate to other places as well leading to ripple Effect which leads to further increase in the cost of change. Object-Oriented Systems tend to be flexible but even then the ripple effect in these systems does exist. In this research work, we have proposed a computation model for the ripple effect encountered in Object-Oriented Systems. We have developed a tool JRECT which will compute the ripple effect for the projects according to our computation model. In our investigation, we have used seven versions of JavaMail: an open-source project to quantitatively relate the maintainability index with ripple effect through regression analysis.

Keywords—Ripple Effect; Object-Oriented Software Maintenance; Maintainability Index; Ripple Effect Computation Model

I. INTRODUCTION

According to Lehman’s Law of Continuing change “An E-type system must be continually adapted or it becomes progressively less satisfactory”. Various maintenance activities needs to be carried out in order to enhance the performance of existing software system or to debug the errors encountered in the software system. Different versions of a software product are launched by software manufacturers from time to time in order to address the issues encountered by the users in their software product and to add some alluring features in their software product so that their product doesn’t become obsolete. However the software products are not flexible. Whenever the changes are made at one place in the software code changes propagate to other places as well leading to ripple effect. If ripple effect is too high maintainability of software product becomes difficult.

Ripple Effect Analysis could play a major role in assessing the maintainability of a software project. If the level of ripple effect of a project is high then the maintainability of the project should be low and vice versa. In case of procedural languages, ripple effect is higher as compared to object-oriented systems. Though object-oriented paradigm offers a large number of features such as encapsulation, abstraction, inheritance and polymorphism, ripple effect is not negligible as the classes are coupled to each other. If the dependencies between the classes are high, then the ripple effect between them is high as well.

As a part of this work, a computational model is proposed here for the assessment of ripple effect in Object-Oriented Systems. Regression analysis has been used to measure the impact of ripple effect on maintainability.

II. RELATED WORK

Much of the work that is done on ripple effect analysis is based on procedural programming paradigm. F.M. Haney[1] described a technique called “Module Connection Analysis” which was
based on applied probabilities and elementary matrix algebra to estimate the total number of changes required to stabilize the systems. Yau and Collofello [2] considered ripple effect from two perspectives: logical ripple effect and performance ripple effect. Sue Black[3] reformulated Yau and Collofello’s ripple effect algorithm and validated its application in the software maintenance phase.

Haider Bilal and Sue Black [4] proposed a metric for the computation of ripple effect in object-oriented Systems by making use of Ripple Effect and Stability (REST) tool by removing all the classes from the code and converting all the member function and member data into regular C functions and global variables respectively. Nikolaos at el. [5] tried to predict the probability of change in object-oriented systems by considering the axes of changes between the classes.

Briand et al[6] related coupling measures to the ripple effect in order to perform impact analysis in Object-oriented systems.

YuQing Yan et al [7] linked ripple effect with requirements evolution and proposed a generic algorithm for identifying requirements dependencies.

III. COMPUTATIONAL MODEL

A. Relationship between ripple effect and dependencies between classes

Ripple Effect is strongly related to dependencies between the classes. If the dependencies between the classes increase, then the ripple effect for the classes should increase as well because the probability that the changes made in one class will propagate to the dependent class will increase.

B. Types of dependencies between classes

Two types of dependencies may exist between the classes-

- Inheritance based dependency,
- Non-inheritance based dependency.

In Java, a class A is said to be dependent on class B via inheritance dependency if

- B is an abstract class and A extends B,
- B is an interface and A implements B.

When a class A inherits class B it inherits all the public and protected members of class A. In first case, class B can override the methods and attributes of its superclass thus reducing the dependency on its superclass. However in second case, class B needs to define all the abstract methods in abstract class A unless it is also declared abstract and in the third case class B needs to implement all the methods in an interface. Also in second and third cases any changes in the declaration of abstract methods will be reflected in class A also.

In Java, a class A is said to be dependent on class B via non-inheritance dependency if it makes a reference to class B and accesses its public and protected (if classes A and B are in the same package) members.

C. Methodology

The proposed approach attempts to compute the ripple effect for Object-Oriented Software Systems in terms of dependencies between the classes. Significant dependencies between the classes are determined which are included for the computation of ripple effect. Different versions of an open-source java project are used for experimental analysis.

Ripple Effect for the whole project is computed in six steps that are described below.

Step 1 - Compute the Ripple Effect between individual pair of classes

In order to compute the ripple effect between individual pair of classes a matrix is formed where rows and columns represents classes and the value at the intersection of ith row and jth column represents the ripple effect between them(RE(i, j)).

Various approaches are considered and applied to the project in order to compute RE(i, j).

Approach 1:

In order to compute the ripple effect between the ith and jth class, first it is determined if ith class is related to jth class by inheritance dependency or non-inheritance dependency (if \(i \neq j\)).
If ith class is related to jth class by inheritance dependency, then there can be many possibilities.

Case 1: j is a non-abstract class and i extends non-abstract class j
In this case number of public methods (Sm pij) and protected methods (Sm pr ij) of jth class are counted. Number of public fields (Sap ij) and protected fields (Sap r ij) of jth class are also counted. Also, the number of overridden methods (Som ij) and overridden fields are also counted (Soa ij).

\[ RE(i, j) = \frac{Smp_j + Smpr_j + Sap_j + Sapr_j}{(Sm pij + Soa ij)} \]  

Case 2: j is an abstract class and i extends abstract class j
Here number of non-abstract public methods (Sm pij) and protected methods (Sm pr ij) of j are counted. Also number of public fields (Sap ij) and protected fields (Sap r ij) of j are counted. Number of abstract public methods (Sm apa ij) and protected methods (Sm pra ij) of j are counted since ith class will have to define the abstract methods unless it is declared abstract itself. Also the number of overridden fields (if any) (Soa ij) of jth class by ith class are counted.

\[ RE(i, j) = \frac{Smp_j + Sanpr_j + Sap_j + Sapr_j + Smpa_i + Smpra_i}{Soa ij} \]  

Case 3: i implements interface j
Here number of methods of j is counted (Sm j)

\[ RE(i, j) = Smj \]  

If ith class is not related to jth class by inheritance dependency and i \( \neq \) j then it is determined if ith class makes a reference to class j i.e. it invokes its methods (static or non-static), uses its attributes or assigns an object to the reference variable of type j. It is represented by a binary count (Bnr ij). Once it is determined that ith class is related to jth class then number of public methods (Sm pij) and protected methods (Sm pr ij) (if i and j are in the same package) of j that i invoked are counted.

\[ RE(i, j) = Bnr ij + Sm pij + Smpr ij \]  

Ripple Effect of a class with respect to itself (RE(i, i))

\[ RE(i, i) = 1 \]  

Approach 2:
A slight modification is done from Approach 1 in computing Ripple Effect of a class with respect to itself. A class is dependent on itself due to its fields (F(i)), methods (M(i)) and constructors (C(i)).

\[ RE(i, i) = F(i) + M(i) + C(i) \]  

Approach 3:
A slight modification is done while calculating the Ripple Effect between the classes unrelated by inheritance dependency. Here Bnrij is not considered since we are already counting the number of static and non-static methods of jth class invoked by ith class.

\[ RE(i, j) = Smp ij + Sm pr j \]  

Approach 4:
Here count of attributes and reference variables is not considered and Ripple Effect of a class with respect to itself is taken as 1.

\[ RE(i, i) = Smp ij + Sm pr j \]  

\[ RE(i, i) = 1 \]  

All the approaches are applied to the open-source project in order to determine the most appropriate approach.

Step 2 - Compute Ripple Effect of all the classes due to a single class i (REa) for \( 1 <= i <= n \) where n is the number of classes

It indicates total significant dependencies of all the classes due to single class.

\[ REa(i) = \sum RE(i, j) \]  

for \( 1 <= j <= n \) where n is the number of classes

Step 3 - Compute Ripple Effect of a class i due to all the classes (REv) for \( 1 <= i <= n \) where n is the number of classes
It indicates the total significant dependency of one class on all the classes.
\[ \text{REV}(j) = \Sigma \text{RE}(i,j) \text{ for } 1 \leq i \leq n \text{ where } n \text{ is the number of classes} \] (11)

Step 4 - Compute the total Ripple Effect of the project (REP)
It indicates the total number of significant dependencies in the project
\[ \text{REP} = \Sigma \text{REA}(i) = \Sigma \text{REV}(j) \text{ for } 1 \leq i \leq n \text{ and } 1 \leq j \leq n \text{ where } n \text{ is the number of classes} \] (12)

Step 5 - Compute average Ripple Effect per class of the project (AREP)
It indicates the average number of significant dependencies per class
\[ \text{AREP} = \text{REP} / n \text{ where } n \text{ is the number of classes} \] (13)

Step 6 - Compute average of average Ripple Effect per class of the project (AAREP)
It indicates the average Ripple Effect of one class due to other class i.e. the average dependency on one class due to other class.
\[ \text{AAREP} = \text{AREP} / n \text{ where } n \text{ is the number of classes} \] (14)

D. Maintainability Index
Maintainability Index is a metrics that indicates maintainability of the software. Higher the maintainability index, higher is the maintainability of the software. It comprises of metrics such as Halstead metrics, McCabe's cyclomatic complexity, lines of code and optional use of comments.

Originally, Omar and Hagmeister[8] proposed a 3-metric Maintainability Index (MI) formula
\[ \text{MI} = 171 - 3.42 \ln (\text{aveE}) - 0.23(\text{g}') - 16.2 \ln (\text{aveLOC}) + 0.99 \text{ ave CM} \] (15)

Where aveE is average Halstead Program Efforts per module, g’ is average cyclomatic complexity per module, aveLOC is average lines of code per module and aveCM is average comment lines per module.

Some academic opinion places more confidence in Halstead Program volume than Halstead Program Efforts so maintainability Index formula was further modified to use Halstead program volume metrics rather than Halstead Program Efforts.

\[ \text{MI} = 171 - 5.2 \ln (\text{aveV}) - 0.23(\text{g}') - 16.2 \ln (\text{aveLOC}) \] (17)

Where aveV is average Halstead Program Volume per module, g’ is average cyclomatic complexity per module and aveLOC is average lines of code per module.

Later on comments were also included in the maintainability index formula. However, inclusion of comments was optional.

\[ \text{MI} = 171 - 5.2 \ln (\text{aveV}) - 0.23(\text{g}') - 16.2 \ln (\text{aveLOC}) + 50.0 \sin (\sqrt{2.46 \text{perCM}}) \] (18)

Where aveV is average Halstead Program Volume per module, g’ is average cyclomatic complexity per module, aveLOC is average lines of code per module and perCM is percentage of comment lines per module.

IV. IMPLEMENTATION OF COMPUTATIONAL MODEL
The proposed computation model is applied to the open source java projects. All the four proposed approaches discussed in previous section for the computation of Ripple Effect between individual pair of classes are applied to the project so that we can analyze all the approaches and determine the most appropriate approach among them.
A. Experimental Setup
The proposed computation model is applied to the open source java projects. All the four proposed approaches discussed in previous section for the computation of Ripple Effect between individual pair of classes are applied to the project so that we can analyze all the approaches and determine the most appropriate approach among them.

1) Open Source Java Project
Different versions of Open source java project JavaMail were downloaded from the site https://java.net/projects/javamail/downloads/directory/source
Seven versions of JavaMail open-source project (JavaMail 1.4.2, JavaMail 1.4.3, JavaMail 1.4.4, JavaMail 1.4.6, JavaMail 1.4.7, JavaMail 1.5.0 and JavaMail 1.5.1) were used for our experiments.

2) Integrated Development Environment
Integrated Development Environment used for experiments is Eclipse Indigo version 3.6

3) Operating System
Windows 7 operating system is used.

4) Third Party Tools
Third party tool used for our implementation of Computation Model are -
   a) CodeproAnalytics- It is an Eclipse plugin which is used to compute various metrics for our software project such as Halstead metrics, lines of code, number of classes, cyclomatic complexity and number of methods.
   b) MS-Excel- It is a part of MS-Office and is used for performing regression analysis and storing experimental results for different versions of software project.
   c) Org.apache.bcel package- This package is used in our self-developed tool which enables the tool to parse java classes.

5) Self-developed tool JRECT
As a part of our research work, a console-based tool JRECT (Java Ripple Effect and Computation Tool) is developed which takes as input jar file of a project and provides detailed class by class ripple effect analysis in an MS-excel friendly format. It also computes the overall and average ripple effect of the entire project.

B. Experimental Results of Ripple Effect Computations
All the four approaches proposed for the computation of Ripple Effect between individual pair of classes is applied to the seven versions of JavaMail project- JavaMail1.4.2(J42), JavaMail1.4.3(J43), JavaMail1.4.4(J44), JavaMail1.4.6(J46), JavaMail1.4.7(J47), JavaMail1.5.0(J50) and JavaMail1.5.1(J51).

Case 1: Approach 1 is applied to selected JavaMail projects

Results obtained are summarized in Table I.

<table>
<thead>
<tr>
<th>Projects</th>
<th>Number of classes</th>
<th>RE</th>
<th>avg RE/class</th>
<th>Class with highest RE</th>
<th>Class with highest RE</th>
</tr>
</thead>
<tbody>
<tr>
<td>J42</td>
<td>343</td>
<td>5774</td>
<td>.095</td>
<td>Folder.java</td>
<td>IMA PFolder.java</td>
</tr>
<tr>
<td>J43</td>
<td>358</td>
<td>5985</td>
<td>.338</td>
<td>Folder.java</td>
<td>IMA PFolder.java</td>
</tr>
<tr>
<td>J46</td>
<td>380</td>
<td>6346</td>
<td>.652</td>
<td>Folder.java</td>
<td>IMA PFolder.java</td>
</tr>
<tr>
<td>J47</td>
<td>401</td>
<td>7515</td>
<td>.481</td>
<td>Folder.java</td>
<td>IMA PFolder.java</td>
</tr>
<tr>
<td>J50</td>
<td>402</td>
<td>7515</td>
<td>18.843</td>
<td>Folder.java</td>
<td>IMA</td>
</tr>
</tbody>
</table>
Table II. Results obtained for Case 2

<table>
<thead>
<tr>
<th>Projects</th>
<th>Number of classes</th>
<th>RE</th>
<th>avg Re/Class</th>
<th>Class with highest REa</th>
<th>Class with highest REv</th>
</tr>
</thead>
<tbody>
<tr>
<td>J4 2</td>
<td>343</td>
<td>9128.095</td>
<td>26.612</td>
<td>Folder.java</td>
<td>IMAPFolder.java</td>
</tr>
<tr>
<td>J4 3</td>
<td>358</td>
<td>9551.338</td>
<td>26.680</td>
<td>Folder.java</td>
<td>IMAPFolder.java</td>
</tr>
<tr>
<td>J4 4</td>
<td>380</td>
<td>1018.7652</td>
<td>26.810</td>
<td>Folder.java</td>
<td>IMAPFolder.java</td>
</tr>
<tr>
<td>J4 6</td>
<td>401</td>
<td>1158.2481</td>
<td>28.884</td>
<td>Folder.java</td>
<td>IMAPFolder.java</td>
</tr>
<tr>
<td>J4 7</td>
<td>402</td>
<td>1158.4426</td>
<td>28.817</td>
<td>Folder.java</td>
<td>IMAPFolder.java</td>
</tr>
<tr>
<td>J5 0</td>
<td>406</td>
<td>1172.0258</td>
<td>28.868</td>
<td>Folder.java</td>
<td>IMAPFolder.java</td>
</tr>
<tr>
<td>J5 1</td>
<td>419</td>
<td>1215.6202</td>
<td>29.012</td>
<td>Folder.java</td>
<td>IMAPFolder.java</td>
</tr>
</tbody>
</table>

Case 3: Approach 3 is applied to selected open source projects. Results obtained are summarized in Table III.

Table III. Results obtained for Case 3

<table>
<thead>
<tr>
<th>Projects</th>
<th>Number of classes</th>
<th>RE</th>
<th>avg Re/Class</th>
<th>Class with highest REa</th>
<th>Class with highest REv</th>
</tr>
</thead>
<tbody>
<tr>
<td>J4 2</td>
<td>343</td>
<td>7981.095</td>
<td>23.268</td>
<td>Folder.java</td>
<td>IMAPFolder.java</td>
</tr>
<tr>
<td>J4 3</td>
<td>358</td>
<td>8355.338</td>
<td>23.339</td>
<td>Folder.java</td>
<td>IMAPFolder.java</td>
</tr>
<tr>
<td>J4 4</td>
<td>380</td>
<td>8924.652</td>
<td>23.486</td>
<td>Folder.java</td>
<td>IMAPFolder.java</td>
</tr>
<tr>
<td>J4 6</td>
<td>401</td>
<td>1023.2481</td>
<td>25.517</td>
<td>Folder.java</td>
<td>IMAPFolder.java</td>
</tr>
<tr>
<td>J4 7</td>
<td>402</td>
<td>1023.443</td>
<td>25.459</td>
<td>Folder.java</td>
<td>IMAPFolder.java</td>
</tr>
<tr>
<td>J5 0</td>
<td>406</td>
<td>1036.0258</td>
<td>25.518</td>
<td>Folder.java</td>
<td>IMAPFolder.java</td>
</tr>
<tr>
<td>J5 1</td>
<td>419</td>
<td>1075.3202</td>
<td>25.664</td>
<td>Folder.java</td>
<td>IMAPFolder.java</td>
</tr>
</tbody>
</table>

Case 4: Approach 4 is applied to selected open source projects. Results obtained are summarized in Table IV.
### TABLE IV. RESULTS OBTAINED FOR CASE 4

<table>
<thead>
<tr>
<th>Projects</th>
<th>Number of classes</th>
<th>RE</th>
<th>avg Re/class</th>
<th>Class with highest REa</th>
<th>Class with highest REv</th>
</tr>
</thead>
<tbody>
<tr>
<td>J42</td>
<td>343</td>
<td>4627.095</td>
<td>13.490</td>
<td>Folder.java</td>
<td>IMAPFolder.java</td>
</tr>
<tr>
<td>J43</td>
<td>358</td>
<td>4789.338</td>
<td>13.378</td>
<td>Folder.java</td>
<td>IMAPFolder.java</td>
</tr>
<tr>
<td>J44</td>
<td>380</td>
<td>5083.652</td>
<td>13.378</td>
<td>Folder.java</td>
<td>IMAPFolder.java</td>
</tr>
<tr>
<td>J46</td>
<td>401</td>
<td>6165.481</td>
<td>15.375</td>
<td>Folder.java</td>
<td>IMAPFolder.java</td>
</tr>
<tr>
<td>J47</td>
<td>402</td>
<td>6165.426</td>
<td>15.337</td>
<td>Folder.java</td>
<td>IMAPFolder.java</td>
</tr>
<tr>
<td>J50</td>
<td>406</td>
<td>6237.258</td>
<td>15.363</td>
<td>Folder.java</td>
<td>IMAPFolder.java</td>
</tr>
<tr>
<td>J51</td>
<td>419</td>
<td>6466.202</td>
<td>15.432</td>
<td>Folder.java</td>
<td>IMAPFolder.java</td>
</tr>
</tbody>
</table>

From the results of all the four cases we can observe that average ripple effects of J42, J43 and J44 are comparable and so are the average ripple effects of J46, J47, J50 and J51. During the code analysis it was observed that this is due to the introduction of new package imap in JavaMail API. Also, the most vulnerable class is ‘IMAPFolder.java’ and the class which is likely to have maximum impact on other classes is ‘Folder.Class’.

### C. Maintainability Index of Open-Source Projects

Maintainability Index of different versions of open-source project JavaMail is computed using Eclipse Plugin CodePro Analytics. We have used formula (17) for maintainability index calculation. Comments are not included as maintainability index is overly sensitive to comments.

### TABLE V. MAINTAINABILITY INDEX RESULTS FOR JAVAMAIL PROJECTS

<table>
<thead>
<tr>
<th>Projects</th>
<th>aveV</th>
<th>g^</th>
<th>aveLOC</th>
<th>MI</th>
</tr>
</thead>
<tbody>
<tr>
<td>J42</td>
<td>3249.650</td>
<td>18.463</td>
<td>80.536</td>
<td>53.608</td>
</tr>
<tr>
<td>J43</td>
<td>3371.813</td>
<td>18.942</td>
<td>82.586</td>
<td>52.898</td>
</tr>
<tr>
<td>J44</td>
<td>3514.820</td>
<td>19.317</td>
<td>84.746</td>
<td>52.219</td>
</tr>
<tr>
<td>J46</td>
<td>3553.169</td>
<td>19.099</td>
<td>84.547</td>
<td>52.172</td>
</tr>
<tr>
<td>J47</td>
<td>3544.517</td>
<td>19.059</td>
<td>84.547</td>
<td>52.231</td>
</tr>
<tr>
<td>J50</td>
<td>3555.132</td>
<td>19.841</td>
<td>84.313</td>
<td>52.317</td>
</tr>
<tr>
<td>J51</td>
<td>3558.355</td>
<td>19.011</td>
<td>84.632</td>
<td>52.206</td>
</tr>
</tbody>
</table>

### D. Regression Analysis

In order to assess the impact of ripple Effect on maintainability, regression analysis is performed with maintainability index as a dependent variable and average ripple effect/class (avgRe/class) as an independent variable. All the four cases of computation model which are implemented on the JavaMail projects are taken into account. We have obtained the plots for all the four cases. Non-linear regression analysis is done keeping in mind real-world scenario.

Plot 1: When the computed maintainability index is plotted against avgRe/class computed according to Case 1 (Fig. 1).

![Fig. 1. MI vs. avgRe/class plot 1](image-url)
Plot 2: When the computed maintainability index is plotted against avgRe/class computed according to Case 2 (Fig. 2).

![Image of MI vs avgRe/class plot 2](image)

Plot 3: When the computed maintainability index is plotted against avgRe/class computed according to Case 3. (Fig. 3).

![Image of MI vs avgRe/class plot 3](image)

Plot 4: When the computed maintainability index is plotted against avgRe/class computed according to Case 4 (Fig. 4).

![Image of MI vs avgRe/class plot 4](image)

All the four graphs indicate that as average ripple effect per class increases, maintainability index decreases. This implies that ripple effect has negative impact on maintainability.

V. RESULTS AND DISCUSSION
We have proposed four approaches for the computation of ripple effect in object-oriented systems. All the four approaches are applied to the different versions of JavaMail project. We also obtained four graphs by performing regression analysis with average ripple effect per class as an independent variable and maintainability index as a dependent variable. Now we have to determine that which approach is the most suitable approach and which plot best depicts the relationship between maintainability and ripple effect.

A. Determining the Relevant Approach for Computation
After studying the four graphs obtained by us after performing regression analysis with average ripple effect per class as independent variable and maintainability index as a dependent variable. All the four graphs indicate that maintainability index decreases with increase in average Ripple Effect per class but in order to determine the most appropriate trend...
line we observed the R-square values of these graphs. We found that
R-square (plot1) < R-square (plot4) < R-square (plot2) < R-square (plot3)
R-square value indicates the goodness of fit. Hence, since R-square value for graph obtained in plot 3 is highest. Hence we have concluded that Approach 3 proposed for the computation of Ripple Effect between individual pair of classes is the most relevant approach and results obtained in Case 3 are the most relevant ripple effect experimental results for the open-source projects.

B. Impact of ripple effect on maintainability
In order to assess and establish the relationship between ripple effect and maintainability regression analysis is performed. The four graphs are obtained after performing regression analysis. All the four trend lines in the graphs indicate that as average ripple effect per class increases maintainability index decreases. Most relevant equation relating MI and avg. re/class

\[ \text{MI} = -8.042 \ln \left( \text{avg re/class} \right) + 78.27 \]

VI. VALIDATION OF RIPPLE EFFECT COMputation MODEL
In order to validate our computation model we studied the relationship between our computed ripple Effect and maintainability. Theoretically ripple effect increases with decrease in maintainability. If our computation model is valid then it should exhibit the same behavior. Therefore, we performed regression analysis with average ripple effect per class (avgre/class) as an independent variable and maintainability index as a dependent variable. The trend line obtained by us indicated that as avgre/class increases maintainability index decreases. This means our experimental results are in conformance with theoretical results and goodness of fit is pretty decent. Hence our computation model is valid.

VII. CONCLUSION
We have determined the significant interclass and intraclass dependencies by performing various experiments on the open-source Java projects. We also performed regression analysis in order to determine the most relevant experiment and to assess the impact of ripple effect on maintainability.

We have proposed a computation model for the computation of ripple effect. We have also developed a tool JRECT which will compute the ripple effect for the project according to our computation model. We have also tried to relate maintainability index with ripple effect through regression analysis.

Ripple Effect analysis can be a useful tool for determining the maintainability of a software product. If the ripple effect in a software product is low then maintainability will be high as fewer efforts will be required to carry out the proposed changes and the additional maintenance activities that are required to handle these ripple effects will be reduced. Ripple effect analysis should be done before carrying out the proposed changes in the software project in order to study the impact of the proposed changes on the software product. Thus ripple Effect analysis can play a significant role in the assessment of maintenance activities.

VIII. FUTURE WORK
In future, the computation model can be applied to more projects which will further validate our project. The computation model can be applied to other programming paradigms such as aspect-oriented programming paradigm. It can be applied to other object-oriented languages such as C++, C# as well. Moreover, here we have tried to assess the impact of ripple effect on maintainability. We can take other quality parameters such as reusability, reliability etc. as well and assess the impact of ripple effect on these quality parameters as well.
References


