International Journal of System Dynamics Applications
Volume 11 • Issue 2 • July-December 2022

Enhancing Behavioral Dependency for Effective Computing in Software

Deepa Bura, Manav Rachna International Institute of Research and Studies, India
Amit Choudhary, Maharaja Surajmal Institute, India

ABSTRACT

Software plays an important role in effective computing and communication of any services. It becomes crucial to identify some critical parts of the software that can lead to enhanced computing and increases efficiency of the software. Dependency plays a significant role in finding relationship amongst classes and predicting change-prone classes. This paper aims to enhance behavioral dependency by defining six types of dependencies amongst classes. These are (1) direct behavioral dependency, (2) indirect behavioral dependency, (3) internal behavioral dependency, (4) external behavioral dependency, (5) indirect internal behavioral dependency, and (6) indirect external behavioral dependency. Evaluating these dependencies gives accurate results for the prediction of change-prone classes. Further, the paper compares the proposed approach with existing methods.

KEYWORDS

Behavioral Dependency, Change-Prone Classes, Communication, Computing, Fault-Prone Classes, Software Engineering, Software Project Management, Software Systems

INTRODUCTION

In the past few years, software industry has grown at a very fast pace. Software systems change constantly with time i.e. every developed software needs to be changed at some point of time in software life cycle. Software change is significant for any organization's progress. As each organization spends a lot of money on their software systems. For maintaining the value of these systems, change is required with changing customer needs.

In any software, there are some parts which are more frequently changed than others. These sensitive parts which are highly prone to changes are known as change prone classes in an object-oriented (OO) software. If such classes are identified early in a software it can help developers to pay more attention on peer review process, testing phase, requirements analysis, maintenance phase and restructuring efforts on particular classes.

UML is a de-facto standard for representing the design of software systems. UML class diagrams depict the dependency among different classes and methods involved in these classes. Thus, it plays a major role in the software development process. When a change in structure or behavior of a class

DOI: 10.4018/IJSDA.20220701.oa1

This article published as an Open Access article distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/4.0/) which permits unrestricted use, distribution, and production in any medium, provided the author of the original work and original publication source are properly credited.

affects the other related class, then there exists a dependency amongst the two classes. So it becomes important to find the change prone classes.

RELATED WORK

Abdeen et al. (2015) predicted improvement, revision, bug fixing and perfective maintenance are also some of the reasons of a software to change. This necessitates software change to be handled properly.

Mathur et al. (2014) analysed that many software projects fail for one or the other reason. One of the major reasons of software failure is incapability to understand the changing requirements and uncontrolled change propagation. Godara et al. (2018) suggested that classes which are prone to changes needs major consideration as these involve more effort and higher amount of maintenance costs and development costs.

Dependency can be defined as degree of association amongst two classes, if change in structure or behavior of one class affects other classes, dependency is said to exist between the classes. Sharafat and Tahvildari (2008) used Unified Modeling Language (UML) diagrams for the evaluation of dependency amongst classes. And the process of reverse engineering was used to find the degree of relationship amongst classes. Jflex software was used for evaluating the results. However, the evaluated results were based on several assumptions.

Lee et al. (2016) worked on co-change, i.e. if one class is changed it affects the other classes also. Research proposed an approach for prediction of co-change volume, using regression line co change was evaluated. Success rate achieved was around 82%. Research focused only on regression line however other factors were ignored. Arisholm et al. (2004) examined change prone classes using dynamic coupling feature. The proposed method was built on relating the amount of modifications in each class with dynamic coupling feature. Godara & Singh (2-15) proposed a new technique to find change in classes using Artificial Bee Colony algorithm.

Accordingly, the proposed model does not fit into the category of change prediction model as effort was not done to associate the anticipated metrics with changes in future versions. The research mainly focussed on finding the relations amongst dynamic coupling and change prone classes. Elish et al. (2014) used the same concept and extended the work of Arisholm et al. (2004) by removing the existing gap of not considering the changes in future versions. Research derived statistical correlation of coupling metrics and change proneness and indicated coupling metrics as a better indicator of change prone classes from one release to another. Software quality is related with software design. High quality software design can benefit in reduction of maintenance and testing costs. Eski et al. (2011) related change prone classes with quality of software. Research indicated software parts which have poor quality tend to change more frequently.

Bura et al. (2017) gave a dynamic measure of predicting change prone classes. Using run time information such as execution time, frequency of methods called, inter-dependency and popularity. Results were validated using OpenClinic and OpenHospital software. Godara and Singh (2014) gave a new hybrid approach for finding change prone classes, in which frequent item set mining algorithm is used to find how many times a method is being called by other methods and how many times a method calls another method. These rules are optimized using Artificial Bee Colony algorithm (ABC) and using decision tree a class is classified as change prone and non- change prone class.

Penta et al. (2008) focussed on prediction of sensitive parts which are more change prone and in addition to this predicted changes which mostly affect some specific classes in a software. The research was based on design patterns which were more affected by change than others. In software evolution, there are design patterns which are more likely to change than others. Considering earlier research, the research was different in the context, as it focussed on certain parts of design patterns rather than focussing on system's entire design pattern.

Godara and Singh (2014) gave a review of different techniques of finding change prone classes and discussed advantages and disadvantages of each method. Further, the paper proposed how

some of the techniques can be improved. Lu et al. (2012) applied statistical techniques for finding the relationships amongst 62 Object-Oriented attributes and change prone classes. The research covered several object-oriented metrics which can be divided in four zones: size, coupling, cohesion and inheritance. For analysing the relationships and combining the outcomes from several studies, the research used random effect model and statistical techniques. Research ranked these categories (size, coupling, cohesion and inheritance) based on their obtained results. Size metrics was given first rank as it can efficiently use in differentiating classes which are having high probability of change in future from other classes. Coupling and cohesion metrics was given second rank, as they have lesser predicting capability as compared to size metrics. Inheritance metrics was given third rank, as they have very poor predicting capability as compared to size metrics and coupling & cohesion metrics.

Godara & Singh (2014) used Behavioral Dependency (BD) to measures the degree of how much a class is dependent on another class. For example, Source Line of Code (SLOC) contains n number of classes, and a class constitutes n number of methods. A method can be called by several classes which causes dependency amongst classes. It is the primary factor of dependency amongst classes, there exists several factors which causes dependencies amongst classes. Bura and Choudhary(2020) proposed a system in which retrieval of classes can be improved by the prediction of change proneness.

Godara & Singh (2014) suggested evaluating dependencies, in the form of class diagrams can help software developers to predict impact of change made in one class on other classes. Predicting behavioral dependencies, in one version of software helps the software developers as they can focus more on such sensitive classes in next version of software. Research of Han et al.(2010), Godara & Singh (2017) suggests to transform these types of dependencies in the form of UML class diagrams in one version of software, such that it is available in design phase in next version of software which can help software developers to concentrate more on such classes and can allocate effort according to evaluated impact of change. Bura and Choudhary (2020) explained the model for finding change prone classes by taking some metrics derived from the software, the suitability of the metrics was proved by finding accuracy. Han et al. (2008) used the concept of behavioral dependency in sequence diagrams and class diagrams.

Galli (2020) gave a review of risk management on the applications of systems. Various factors were determined for the analysis of risk in a system. Herrera et al. (2019) studies the concept of supply chain and the behavior of customer on the applications of various factors. Shanbhag and Pardede (2019) studied the behavior of software when a new project or startup is initiated. Study shows that software is dynamic and it changes, study revealed varius factors on which a software can be evaluated. Al-Kadeem et al. (2017) studied the how the work system changes in an industry. Soni & Chorasia (2017) studied various policies related to training in higher sectors. Pradhan (2017) gave a model for minimizing the risk in any project.

INTRODUCTION TO UML

UML has emerged as a standard language in Object-Oriented software industry. It is used by software developers for design and development of Object-Oriented (OO) systems. This modelling language helps in visualizing, constructing and documenting the software system. It is used for developing software applications and it is applied in several domains. It is employed to design software, communicate software or business processes, and capture the necessary data for requirement and analysis phase. Garousi et al. (2006) analysed that UML acts as a bridge between idea formulation and implementation phase. It offers several types of diagrams. The arrival of the UML has brought numerous advantages since it has unified various Object-Oriented analysis and design methods into one standard modeling language. Due to the evident utilization and tool support to model the design, Inpirom & Prompoon (2013) discussed that UML occupies a vital part in design and coding part of software development life cycle. Over the years, it has been extensively used as a standard language in Object Oriented software industry.

UML Class Diagrams and Change Proneness

Prediction of change proneness in classes provides a vital information to software developers in the course of software development and maintenance. Such information is significant to software developers as they can make more flexible software by giving more attention on such classes. UML class diagrams plays a major role in providing information about change prone classes. Software developers can get insight of the classes which classes are more dependent on one another by analysing various levels of dependencies. This derived information from source code can help software developers to build a better software in next release of software.

Changes in a software can be due to two reasons. Firstly, it can be due to addition of source code i.e. if a new class is added, or existing class gets deleted. Change can spread from other classes also. Such as if change is implemented in one class, and this change has impact on other related classes also. These changes in software are termed as "spread changes". UML class diagrams helps in predicting such type of changes in software systems. By analysing various types of dependencies, change prone classes can be effectively predicted which can also help in finding impact of change. Amongst various types of dependencies paper considers behavioral dependency for predicting change proneness of classes and estimating impact of change occurring in related classes.

Early finding of change proneness in classes allows the programmers and software professionals to devote their valued time and resources on these domains of software. Predicting such sensitive classes can help in developing a stable software, such as software developers can focus on these classes and can choose alternate design before implementation starts. UML class diagrams plays a major role in predicting such classes in design phases of software development life cycle. Changes made to a class are not restricted to that class only, it affects other classes also, therefore it becomes important to analyse changes in individual classes and show changes in related classes. For this purpose, various types of relationships existing in UML diagrams helps significantly in predicting how change propagates to other classes. Change prediction in source files which are highly sensitive to change can aid in the effective distribution of efforts and software resources.

The possibility of occurrence of change in a class is referred to as change proneness of class to future changes. The independent variables are a set of object oriented metrics listed below:

- **Coupling between object (CBO):** It is count of classes whose attributes are used by a particular class in addition to that classes which uses the attributes or methods of that particular class.
- **Number of Children (NOC):** It is the count of direct child of a class.
- **Number of Attributes (NOA):** It is the count of attributes defined in a class.
- **Number of Instance Variable (NIV):** It is a measure of relationship of a class with other objects in the software system.
- **Depth of Inheritance Tree (DIT):** It is the count of levels from class node to the parent of the tree.
- **Number of Methods (NOM):** It is the count of methods defined in each class.
- **Number of Instance Method (NIM):** Number of Instance Methods.
- **Number of Local Methods (NLM):** It is the count of local methods i.e. number of methods which are not inherited.
- **Response for Class (RFC):** It is the count of class's method and methods called by class's methods. Sum of these methods is response for a class.
- **Number of Local Default Visibility Methods (NLDM):** It is the count of local default visibility methods.
- **Number of Private Methods (NPRM):** It is the count of local private methods in a class which are not inherited.
- **Number of Protected Methods (NPROM):** It is the count of local protected methods in a class.
- **Number of Public Methods (NPM):** It is the count of public methods which are not inherited.
- **Lack of Cohesion amongst methods (LCOM):** It is the count of methods that uses the data field of that particular class. It calculates how much a class is related with its field.

Above discussed Object-Oriented metrics helps in the prediction of change prone classes. Earlier, most of the researcher have used these existing metrics for finding the change prone classes. Information regarding most of these metrics can be generated from UML class diagrams. ObjectAid UML Explorer can be utilized for generating the class diagrams which are used for finding the dependencies. This tool depicts the Java source code and libraries in UML class diagrams.

BEHAVIORAL DEPENDENCY

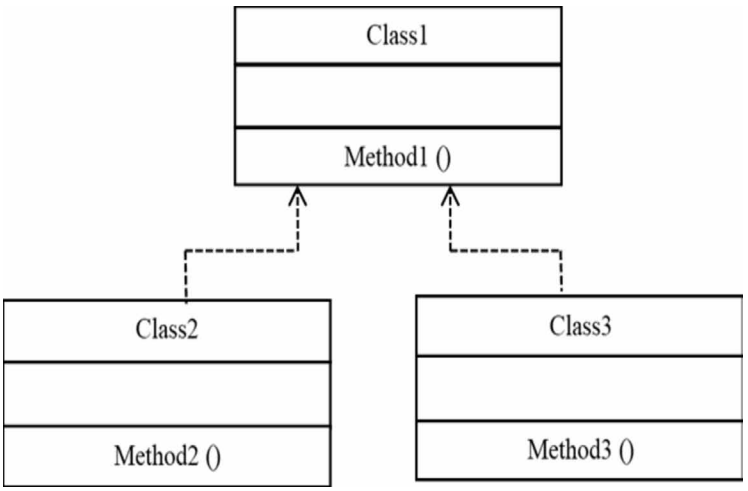
Dependency is a type of relationship amongst classes. If a class is dependent on other class, it is not necessary that other class is also dependent on the first class. A class which affects other is known as dependent class and classes that are dependent on this class are known as depending classes. Any changes made in dependent class affects the functionality of depending class. Thus, it becomes necessary to deal with such types of dependencies amongst classes.

Dependency is a directed relationship which is utilized to depict that certain UML elements or a group of elements depends on other elements for specification or implementation. Changing a dependent class impacts the related depending classes and it can be predicted by evaluating dependencies between dependent class and depending classes. For example, when a method of one class is called in other class, method invocation in first class and values returned by other class is a type of dependencies between classes.

Behavior is defined as a direct consequence of the actions of objects. It states how the states of the contributing objects alter over time. Behavior specifications can be utilized to define state or demonstrate the behavior of an object. A number of methods are available in UML to state behaviors such as use case diagrams, sequence diagram, etc. This research work uses UML class diagrams to determine behavioral dependency.

Inheritance and polymorphism are two most important factors for calculating behavioral dependency. Because of these two factors behavioral dependency amongst classes become more complex. Thus, it becomes necessary to consider both these factors for evaluation of dependency amongst classes, which is further crucial for evaluation of change prone classes. In the class diagram of Figure 1, the $class2$ and $class3$ are dependent on class1, which shows that the classes $class2$ and $class3$ calls the $method1()$. This dependency information can be derived from the UML class diagram. If the methods in $class1$ changes then it is propagated to the dependent classes also.

Figure 1. Dependency



To measure the behavioral dependency six types of behavioral dependencies are defined and used in this paper. These are (i) direct behavioral dependency (ii) indirect behavioral dependency (iii) internal behavioral dependency (iv) external behavioral dependency (v) indirect internal behavioral dependency and (vi) Indirect External Behavioral Dependency. Evaluating all these types of dependencies, considers all possible types of inter dependencies that exists between classes. Thus, prediction of change impact analysis and change prone classes gives accurate results when these depicted dependencies are implemented for finding change prone classes.

Research methodology is to start with the first version of software, find all the six specified behavioral dependencies between classes. Dependency gives a measurement of how much a class is related to other. If the dependency comes out be larger, it indicates that the class is highly prone and if the dependency is less it indicates the class is less prone. Accordingly, the class can be divided onto one of the two groups i.e. change prone and non-change prone.

Direct Behavioral Dependency

Definition: Given two classes' $C1$ and $C2$. $C2$ has a direct behavioral dependency on $C1$. If $C2$ needs some service of $C1$ it achieves it by calling some methods of $C1$ and $C1$ returns the values to $C2$. The direct behavioral dependency is denoted by $\rightarrow$. Here $C2 \rightarrow C1$.

Figure 2 illustrates direct dependency. Here there are two classes $C1$ and $C2$ where $C2$ is dependency class and $C1$ is dependent class. The class $C2$. exhibits direct dependency on $C1$ since it gets some services of $C1$ by calling the methods of $C1$. The changes made in $C1$ are propagated to $C2$ as $C2$ is based on $C1$.

Indirect Behavioral Dependency

Definition: Given three classes' $C1$, $C2$ and $C3$, $C3$ has a direct behavioral dependency on $C2$, $C2$ as a direct behavioral dependency on $C1$ (i.e. $C3$ has an indirect behavioral dependency on $C1$). If $C3$ needs some service of $C1$ it calls the methods of $C1$ via $C2$ and $C1$ returns values to $C3$ through $C2$. The indirect behavioral dependency is denoted by $\rightsquigarrow$. Here $C2 \rightarrow C1$, $C3 \rightarrow C2$ and $C3 \rightsquigarrow C1$.

Figure 3 depicts the indirect behavioral dependency between various classes. There are three classes $C1$, $C2$ and $C3$ where $C2$ is dependency class of $C1$ and $C3$ is dependency class of $C2$.

Figure 2. Direct Behavioral Dependency

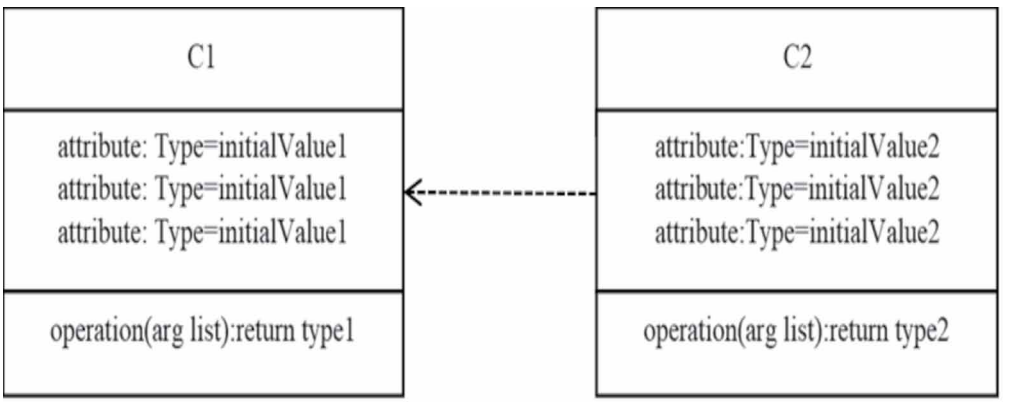
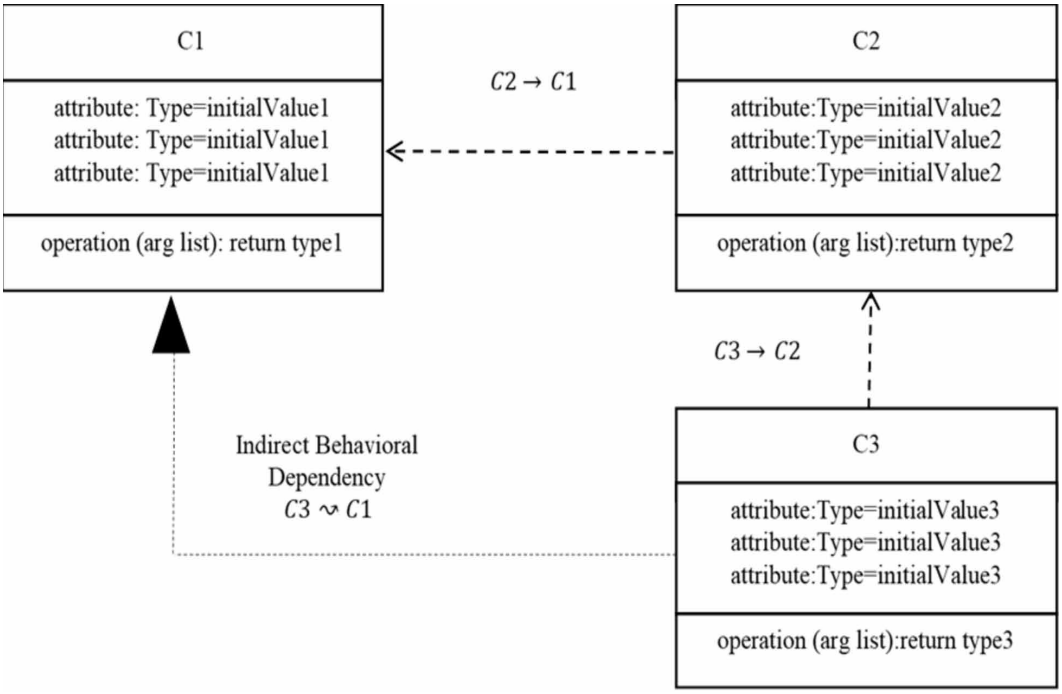


Figure 3. Indirect Behavioral Dependency



$C1$ is the dependent class. The class $C2$ exhibits direct dependency on $C1$ since it gets some services of $C1$ by calling the methods of $C1$. Class $C3$ invokes methods of $C2$ which in turn invokes the methods of $C1$. Hence, $C3$ exhibits indirect dependency on $C1$. Any change made in the $C1$ affects $C2$ which in turn affects $C3$.

Algorithm Tracing direct and indirect behavioral dependency

```
Input:  Java files
Output: behavioral dependency class count (BDCNT), behavioral
        dependency class name, behavioral dependency interface
        names, dependent class name DCLS, depending class name
        (DNGCL), dependent interface name (IDCLS), depending
        interface name (IDNGCLS).
Step 1: for each java program
Step 2: Start function – dependency finder
Step 3: Initialize the value of BDCNT as 1
Step 4: Repeat till the end of readline
Step 5: Start
Step 6: If class is present and if extends or implements is present
Step 7: Increment the value of BDCNT
Step 8: If BDCLS contains extends then the class that precede
        the keyword extends is depending class. Store the class
        name of depending class. The class that comes after
        extends is dependent class, store the class name of dependent class
```

Step 9: If $BDINTERF$ contains implements then the class that precede the keyword implements is depending interface. Store the interface name of depending interface name $IDCLS$. The interface that comes after implements is dependent interface, Store the name of dependent interface as $IDNGCLS$

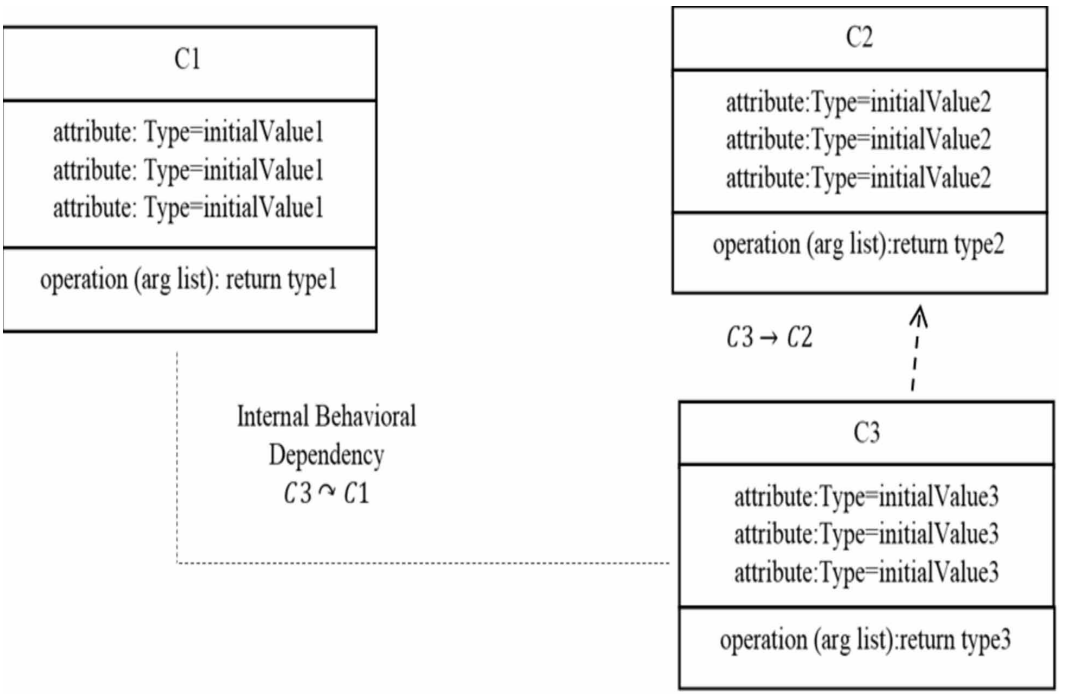
Step 10: End the function

Internal Behavioral Dependency

Definition: Given three classes' $C1$, $C2$, and $C3$. $C3$ has a direct behavioral dependency on $C2$ and $C1$ is an independent class. If $C3$ needs some service of $C2$ by calling the methods of $C2$, $C2$ returns the values to $C3$. If $C3$ needs some service of $C1$ it gets it by creating object of $C1$ and then using $C1$ object to invoke methods of $C1$. Finally, $C1$ returns values to $C3$. The internal behavioral dependency is denoted by $\curvearrowright$. Here $C3 \rightarrow C2$ and $C3 \curvearrowright C1$. Figure 4 illustrates the internal behavioral dependency.

Here there are three classes $C1$, $C2$ and $C3$ where $C3$ is dependency class of $C2$. $C1$ is the dependent class. The class $C3$ exhibits direct dependency on $C2$ since it gets some services of $C2$ by calling the methods of $C2$. Class $C3$ invokes methods of $C1$ by creating objects of $C1$. Hence, $C3$ exhibits internal dependency on $C1$. Any change made in $C1$ affects $C3$.

Figure 4. Internal Behavioral Dependency



External Behavioral Dependency

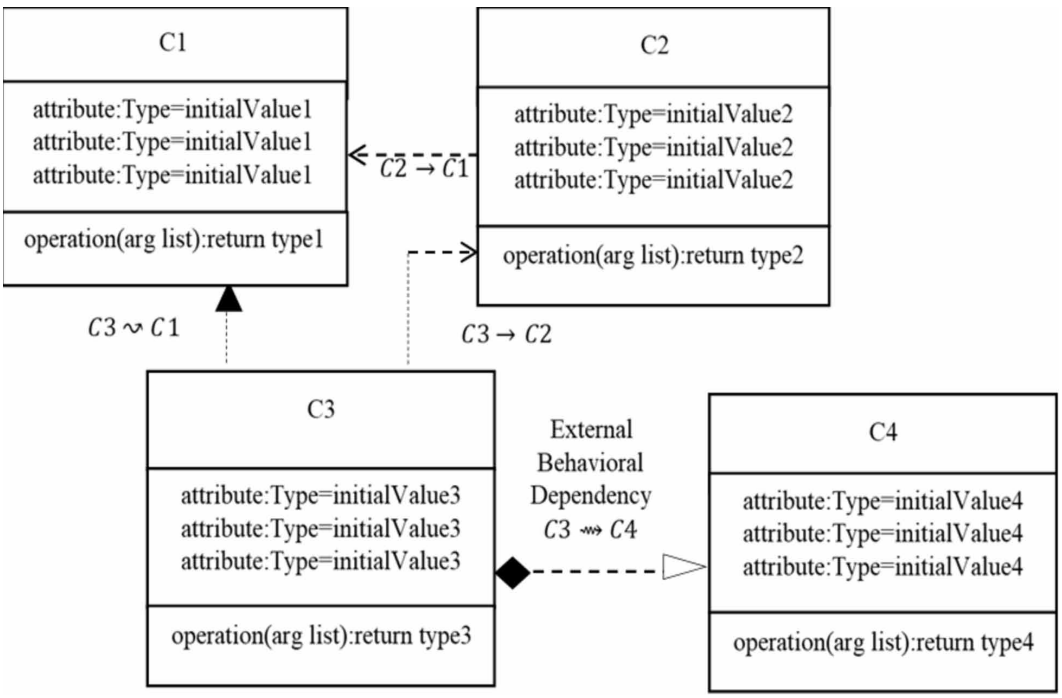
Definition: Given four classes' $C1$, $C2$, $C3$ and $C4$. $C3$ has a direct behavioral dependency on $C2$, $C2$ has a direct behavioral dependency on $C1$ (i.e. $C3$ has an indirect behavioral dependency on $C1$). $C4$ is the independent class of another package. If $C3$ needs some service of $C4$ by importing another package, then object of $C4$ is created to invoke methods of $C4$ and $C4$ returns values to $C3$. This type of dependency between $C3$ and $C4$ is known as external behavioral dependency. It is denoted by $\rightsquigarrow$. Here $C2 \rightarrow C1$ and $C3C1$, $C3 \rightsquigarrow C4$.

Figure 5 depicts the external behavioral dependency. Here there are four classes $C1$, $C2$, $C3$ and $C4$ where $C2$ is dependency class of $C1$ and $C3$ is dependency class of $C2$. $C4$ is an independent class belonging to another package. $C1$ is the dependent class. $C3$ invokes methods of $C4$ by importing the package that has $C4$ and then invoking it in $C3$. Class $C2$ exhibits direct dependency on $C1$ since it gets some services of $C1$ by calling the methods of $C1$. Any change made in the classes $C1$, $C2$ or $C4$ affects $C3$. Hence, the change is propagated through classes.

Algorithm Find internal and external behavioral dependency

```
Input:  Java files
Output: behavioral dependency class count BDCNT, behavioral
        dependency class name, behavioral dependency interface
        names, dependent class name DCLS, depending class name DNGCL.
1.  Start function – dependency finder
2.  Initialize BDCNT -> 1
```

Figure 5. External Behavioral Dependency



3. Repeat until readline
4. If keyword method or class-> found
4a: If keyword "*new*" -> found or *classname* ->found
4b: Store *BDFNAME* & $BDCNT = BDCNT + 1$. Store *BDCLSNAME*.
5: For each *BDCLS*:
5a: Initialize count =1
5b. If count I is 1, then it is dependent class. Otherwise it is depending class name.
6: Increment I, Go to 5b.
7. Stop

Indirect Internal Behavioral Dependency

Definition: Given two classes $C1$ and $C2$. $C2$ has a direct behavioral dependency on class $C1$, if $C2$ needs some service of class $C1$ by calling some parameterized constructor methods of $C1$ and $C1$ returning some values to $C2$ via super method or keyword. The indirect internal behavioral dependency is denoted by $\rightsquigarrow$. Here $C2 \rightarrow C1$ and $C2 \rightsquigarrow C1$. Figure 6 illustrates indirect internal behavioral dependency.

There are two classes $C1$ and $C2$, in which $C2$ invokes methods of $C1$ by using parameterized constructors and hence $C2$ is changed when the $C1$ is changed. Here $C2$ is internally and indirectly depending on $C1$.

Indirect External Behavioral Dependency

Definition: Given four classes $C1$, $C2$, $C3$, $C4$. $C3$ has a direct behavioral dependency on $C2$, $C2$ has a direct behavioral dependency on $C1$ (i.e. $C3$ has an indirect behavioral dependency on $C1$). $C4$ is the independent class of another package.

If $C3$ needs some service of $C1$ by calling some methods of $C1$ via $C2$ and $C1$ returns some values to $C3$ via $C2$. If $C3$ needs some service of $C4$ by importing another package, then invoking some parameterized constructor methods without Object of $C4$ and $C4$ returning some values of $C3$ leads to indirect external behavioral dependency. It is denoted by $\leftrightarrow$. Here $C2 \rightarrow C1$, $C3 \rightsquigarrow C1$ and $C3 \rightsquigarrow C4$.

Figure 6. Indirect Internal Behavioral Dependency

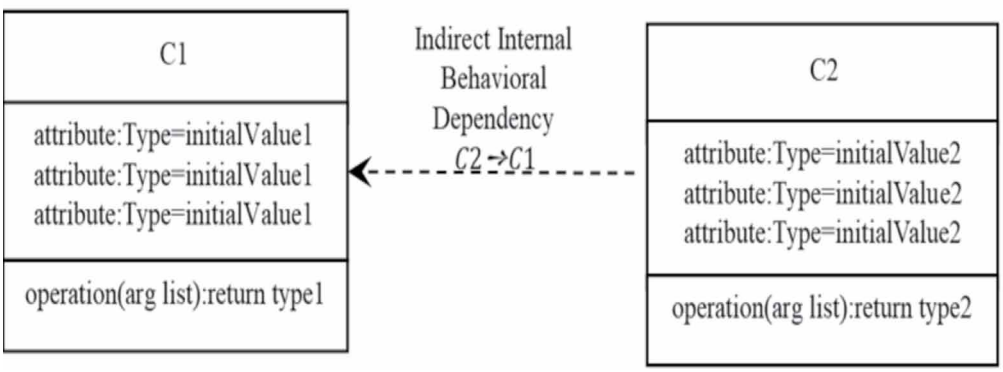
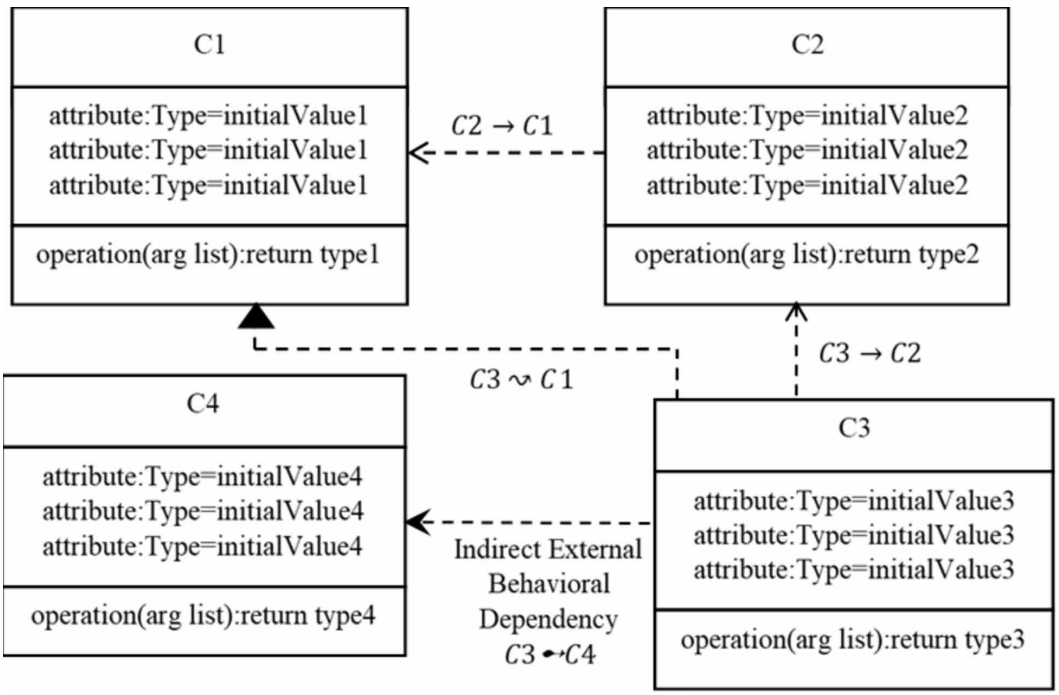


Figure 7 portrays indirect external behavioral dependency. Here there are four classes $C1$, $C2$, $C3$ and $C4$ where $C2$ is dependency class of $C1$. $C4$ is an independent class belonging to another package. $C3$ invokes methods of $C4$ by importing the package that has $C4$ and then invokes parameterized constructors in $C3$. Any change made in the classes $C1$, $C2$ or $C4$ affects $C3$. Also, the change is propagated through classes.

Algorithm Find indirect internal and external behavioral dependency.

```
Input:  Java files
Output: behavioral dependency class count BDCNT, behavioral
        dependency class name, behavioral dependency interface
        names, dependent class name DCLS, depending class name DNGCL.
1.  for each java program
2.  Start function – dependency finder
3.  Initialize the value of BDCNT -> 1
4.  Repeat until readline
5.  If a method -> found or keyword class -> found or keyword
    super -> found
6a: If the keyword 'new' -> found or class name -> found
6b: Store BDFNAME and BDCNT = BDCNT + 1. Store the BDCLSNAME
7.  For each BDCLS do the following steps
8.  Initialize count value I = 1
9.  If count I is 1, class=dependent class, else class= depending class.
10. Increment I, Go to step 9
11. Stop
```

Figure 7. Indirect External Behavioral Dependency



Using all the above mentioned algorithms, dependency of a class is calculated which gives the total count of dependency as discussed in next section.

MEASURE OF BEHAVIORAL DEPENDENCY

$$BD\left(C_j\right) = \sum_{j \le n} Sum\ \ of\ \ D_D, D_I, D_{INT}, D_{EXT}, D_{II}, D_{IE} \tag{1}$$

where, $C_j, 1 \le j \le n$, and n is the total number of classes:

D_D – direct behavioral dependency
D_I – indirect behavioral dependency
D_{INT} – internal behavioral dependency
D_{EXT} – external behavioral dependency
D_{II} – indirect internal behavioral dependency
D_{IE} – indirect external behavioral dependency

Algorithm Algorithm to measure behavioral dependency

Step 1: Compute the direct dependency (D_D), indirect dependency (D_I).
Step 2: Compute the internal dependency (D_{INT}), external dependency (D_{EXT}).
Step 3: Compute the indirect internal dependency (D_{II}) and
indirect external dependency (D_{IE}).
Step 4: Behavioral dependency is computed by using equation (1).

DEPENDENCY MODEL FOR CHANGE IMPACT ANALYSIS

Research provides mathematical model for dependency evaluation in Object-Oriented Software systems. This model helps in evaluation of dependencies using various attributes, which can be used for finding relationship amongst classes. The evaluated relationships are used for finding the impact of changes amongst classes. Dependency model is build using behavioral dependencies which were defined in paper 5.

This dependency model gives a theoretical approach for finding similarity between various classes. Similarity is estimated by evaluating attribute and operations similarity between various levels of defined dependencies amongst classes. Estimation of dependency between various classes using this dependency model can help in early prediction of impact of changes amongst classes.

Direct Behavioral Dependency

Consider two classes C_1 and C_2 having direct dependency between them. Class similarity matrix is obtained using equation (2):

$$NIS(C_1, C_2) = v_n \times NSM(C_1, C_2) + v_i \times ISM(C_1, C_2) \tag{2}$$

where, C_1, C_2 are two classes:

v_n *and* v_i - Represent arbitrary weights assigned to the name similarity and internal similarity respectively.

NIS represents Name Internal Similarity (NIS), NSM represents Name Similarity Matrix which measures the similarity between the names of two classes, C_1 and C_2, based on their semantic similarity.

ISM is the Internal Similarity Matrix (ISM) which measures the internal similarity of two classes C_1 and C_2, as a weighted similarity of their attributes' and operations similarity.

NSM and ISM are evaluated using equation (3) and (4):

$$NSM\left(C_1, C_2\right) = SS\left(Name\left(C_1\right), Name\left(C_2\right)\right) \tag{3}$$

$$ISM(C_1, C_2) = v_a \times ASim(C_1, C_2) + v_m \times OSim(C_1, C_2) \tag{4}$$

v_a *and* v_m - Represent arbitrary weights assigned to the attributes and operations similarity respectively.

SS represents the semantic similarity which is stated as a metric over a set of documents, where the similarity is evaluated based upon their similar meaning rather than syntactical representation.

Attribute similarity and operations similarity is evaluated using equations (5) and (6) $ASim$ and $OSim$ between classes C_1 and C_2 in the interval of 1 to i is given by:

$$ASim(C_1, C_2) = Max\left[\forall \sum_{n=1}^{|A_1|} aSim\left(a_k, a_l\right)\right] / \left|A_2\right| \tag{5}$$

$$OSim(C_1, C_2) = Max\left[\forall \sum_{n=1}^{|O_1|} oSim\left(o_k, o_l\right)\right] / \left|O_2\right| \tag{6}$$

Finally, the neighbourhood similarity is derived from following equation (7):

$$NNHSM(C_1, C_2) = v_n \times NSM(C_1, C_2) + v_{nh} \times NHSM(C_1, C_2) \tag{7}$$

where, NNHSM is Name Neighbourhood Similarity matrix, NHSM is the Neighbourhood Similarity Matrix. It is used to calculate neighbourhood similarity in the same way.

Indirect Behavioral Dependency

Given three classes' $C1$, $C2$ and $C3$, $C3$ has a direct behavioral dependency on $C2$, $C2$ has a direct behavioral dependency on $C1$ (i.e. $C3$ has an indirect behavioral dependency on $C1$). If $C3$ needs some service of classname1 it calls the methods of $C1$ via $C2$ and $C1$ returns values to $C3$ through $C2$. The indirect behavioral dependency is denoted by $\rightsquigarrow$. Here $C2 \rightarrow C1$, $C3 \rightarrow C2$ and $C3 \rightsquigarrow C1$. Class similarity is obtained using following metrics in the classes which have indirect

behavioral dependency. Consider three classes C_1, C_2 and C_3 having indirect dependency between them. Suppose, there is an indirect dependency between C_1 and C_3 through C_2.

Name Similarity Matrix is given by the following equation (8):

$$NIS(C_1,C_2,C_3) = v_n \times NSM(C_1,C_2,C_3) + v_i \times ISM(C_1,C_2,C_3) \tag{8}$$

where, C_1, C_2 and C_3 are classes and C_3 is given by, $C_3 = C_1$ in terms of C_2.

v_n *and* v_i - Represent arbitrary weights assigned to the name similarity and internal similarity respectively.

Similarly, NSM, ISM, $ASim$, $OSim$ and NNHSM are evaluated using equations (9), (10), (11), (12),(13):

$$NSM(C_1,C_2,C_3) = SS(Name(C_1),Name(C_2),Name(C_3)) \tag{9}$$

$$ISM(C_1,C_2,C_3) = v_a \times ASim(C_1,C_2,C_3) + v_m \times OSim(C_1,C_2,C_3) \tag{10}$$

v_a *and* v_m - Represent arbitrary weights assigned to the attributes and operations similarity respectively:

$$ASim(C_1,C_2,C_3) = Max\left[\forall\sum_{n=1}^{|A_1|} aSim\left(a_k,a_l\right)\right] / \left|A_2\right| \tag{11}$$

$$OSim(C_1,C_2,C_3) = Max\left[\forall\sum_{n=1}^{|O_i|} oSim\left(o_k,o_l\right)\right] / \left|O_2\right| \tag{12}$$

$$NNHSM(C_1,C_2,C_3) = v_n \times NSM(C_1,C_2,C_3) + v_{nh} \times NHSM(C_1,C_2,C_3) \tag{13}$$

v_n *and* v_{nh} - Represent arbitrary weights assigned to the name similarity and neighborhood similarity respectively.

Here, all the values are examined between C_1 and C_3 through C_2.

Internal Behavioral Dependency

Name internal similarity and name neighbourhood similarity is evaluated for three classes using equations (14) and (15):

$$NIS(C_1,C_2,C_3) = v_n \times NSM(C_1,C_2,C_3) + v_i \times ISM(C_1,C_2,C_3) \tag{14}$$

$$NNHSM(C_1, C_2) = v_n \times NSM(C_1, C_2, C_3) + v_{nh} \times NHSM(C_1, C_2, C_3) \quad (15)$$

v_n and v_i - Represent arbitrary weights assigned to the name similarity and internal similarity respectively.

v_n and v_h - Represent arbitrary weights assigned to the name similarity and neighborhood similarity respectively.

where, C_1, C_2 and C_3 are classes and C_3 is given by:

$C_3 = C_2$ and $C_2 = C_1$ and $C3 \curvearrowright C1$ via C_2

Name similarity matrix and internal similarity matrix is given by equations (16) and (17):

$$NSM(C_1, C_2, C_3) = SS(Name(C_1), Name(C_2), Name(C_3)) \quad (16)$$

$$ISM(C_1, C_2, C_3) = v_a \times ASim(C_1, C_2, C_3) + v_m \times OSim(C_1, C_2, C_3) \quad (17)$$

v_a and v_m - Represent arbitrary weights assigned to the attributes and operations similarity respectively:

$$ASim(C_1, C_2, C_3) = Max\left[\forall \sum_{n=1}^{|A_1|} aSim\left(a_k, a_l\right)\right] / \left|A_2\right| + ... \quad (18)$$

$$OSim(C_1, C_2, C_3) = Max\left[\forall \sum_{n=1}^{|O_1|} oSim\left(o_k, o_l\right)\right] / \left|O_2\right| + ... \quad (19)$$

Using equations (18) and (19) attribute similarity and operations similarity of the classes is obtained.

A_1 and A_2 are two sets of attributes of Classes C_1, C_2, C_3 respectively:

$a_k \in A_1$ and $a_l \in A_2, \left|A_1\right| \leq \left|A_2\right|$

The attribute similarity $aSim(a_k, a_l)$ between two attributes, a_k and a_l is computed based on their semantic similarity as quantified by above equation (18).

Similarly:

$o_k \in O_1$ and $o_l \in O_2, \left|O_1\right| \leq \left|O_2\right|$

The operations similarity $OSim(o_k, o_l)$ between two attributes, $o_k and o_l$ is computed based on their semantic similarity as quantified by above equation (19).

External Behavioral Dependency

External behavioral dependency is denoted by $\rightsquigarrow$. Here $C2 \rightarrow C1$ and $C3 \rightsquigarrow C1$, $C3 \rightsquigarrow C4$.

The Name Similarity Matrix is given by the following equation (20):

$$NIS(C_1, C_2, C_3, C_4) = v_n \times NSM(C_1, C_2, C_3, C_4) + v_i \times ISM(C_1, C_2, C_3, C_4) \tag{20}$$

where, C_1, C_2, C_3 and C_4 are classes and $C_1 = C_3$ via $C_2 = C_4$ via C_3.

NSM, ISM, $ASim$ and $OSim$ are evaluated in the following equations (21), (22), (23), (24):

$$NSM(C_1, C_2, C_3, C_4) = SS(Name(C_1), Name(C_2), Name(C_3), Name(C_4)) \tag{21}$$

$$ISM(C_1, C_2, C_3, C_4) = v_a \times ASim(C_1, C_2, C_3, C_4) + v_m \times OSim(C_1, C_2, C_3, C_4) \tag{22}$$

$$ASim(C_1, C_2, C_3, C_4) = Max\left[\forall \sum_{n=1}^{|A_1|} aSim\left(a_k, a_l\right)\right] / \left|A_2\right| \tag{23}$$

where, A_1 and A_2 are two sets of attributes of Classes C_1, C_2, C_3 and C_4 respectively.

$v_a \ and \ v_m$ - Represent arbitrary weights assigned to the attributes and operations similarity respectively:

$$a_k \in A_1 \ and \ a_l \in A_2, \left|A_1\right| \leq \left|A_2\right|$$

The similarity $ASim(a_k, a_l)$ between two attributes, $a_k \ and \ a_l$ is computed based on their semantic similarity as quantified by above equation 22:

$$OSim(C_1, C_2, C_3, C_4) = Max\left[\forall \sum_{n=1}^{|O_1|} oSim\left(o_k, o_l\right)\right] / \left|O_2\right| \tag{24}$$

where:

$$o_k \in O_1 \ and \ o_l \in O_2, \left|O_1\right| \leq \left|O_2\right|$$

The similarity $OSim(o_k, o_l)$ between two attributes, $o_k \ and \ o_l$ is computed based on their semantic similarity as quantified by above equation (25):

$$NNHSM(C_1, C_2, C_3, C_4) = v_n \times NSM(C_1, C_2, C_3, C_4) + v_{nh} \times NHSM(C_1, C_2, C_3, C_4) \quad (25)$$

Name neighbourhood similarity matrix is evaluated using equation (25).

Here, all the values was considered such as: 1) C_1, C_3 through C_2; 2) C_2, C_4 through C_3.

COMPARATIVE ANALYSIS

This section shows the comparative analysis with related work given by other authors. Mostly Researchers have compared two versions of software using source line of code. Such that the change prone classes have been predicted using attributes such as line inserted, deleted or modified. And compared different versions of software line by line using tool such as Winmerge software. Using this information researchers predicted class as change prone and non-change prone class. This work provides an advantage over related work, as it provides change prone classes using design measures, which is dynamic in nature.

Comparative Analysis With Existing Work

The efficiency of the proposed is compared with the existing technique Han et al. (2010) in terms of prediction of number of change prone classes. The behavioral dependency measure is evaluated on a diverse version of open-source software JFlex, which is implemented in Java and is taken from http://sourceforge.net. The concept behind the working of JFlex lexers is deterministic finite automata (DFAs). It is quick, do not introduce exclusive backtracking and is intended to operate along with the LALR parser generator by Scott Hudson, and the Java modification of Berkeley Yacc BYacc/J by Bob Jamison.

Table 1 provides the comparison of number of classes identified as change prone classes. From Figure 8, it is seen that for 8 packages, the proposed technique predicts 114 classes and the existing technique identified 106 classes. The proposed technique extracted classes based on the six behavioral dependencies whereas the existing method extracted classes with two behavioral dependencies. Thus, the proposed approach outperforms and gives better refined results based on the extended behavioral dependency concept.

The prediction capability of the proposed technique was also evaluated with existing approach Malhotra and Khanna (2013) against data sets obtained from two versions of open source software (Frinika and FreeMind).

FreeMind is leading software for mind-mapping which is implemented in Java. Source code of the software is available at www.sourceforge.net. In addition to mind map functionality, FreeMind is a hierarchical editor that is easy to use and focus more on folding. It is used to manage knowledge and content. Frinika is free software that offers entire music workstation which can run on Linux, Windows, Mac OSX and other operating systems. It is implemented in Java and has many characteristics like sequencer, soft-synths, real time effects and audio recording. Table 2 gives the number of change prone classes predicted using proposed and existing approaches .

Figure 9, gives the number of change prone classes in Frinika software version 0.2.0 and 0.6.0 and Freemind software versions 0.9.0 RC1 and 0.9.0 RC7 using proposed and existing approach.

Table 1. Number of change prone classes detected using proposed and existing methods

Metrics	Name	Version 1	Version 2	No. of Classes	Packages
Existing	Jflex	1.3	1.4.3	106	8
Proposed	Jflex	1.3	1.4.3	114	8

Figure 8. Change proneness prediction using proposed and existing Han et al. (2010) methods

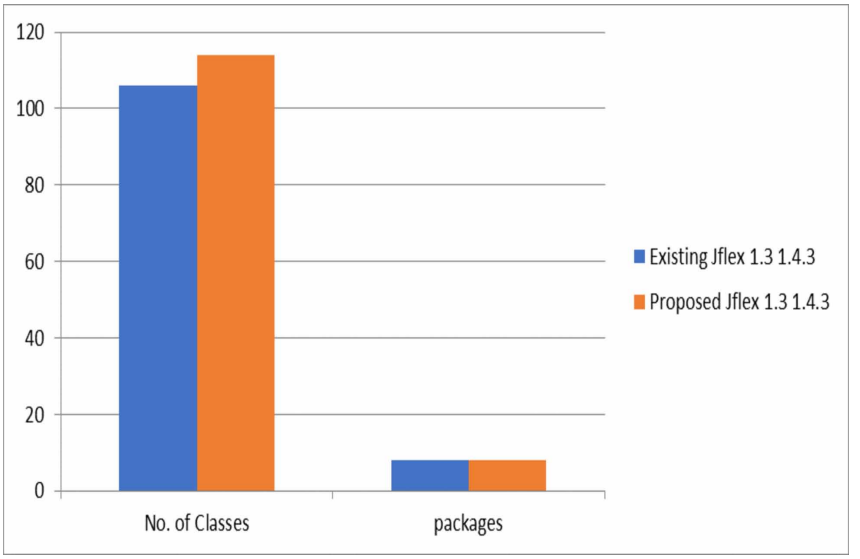
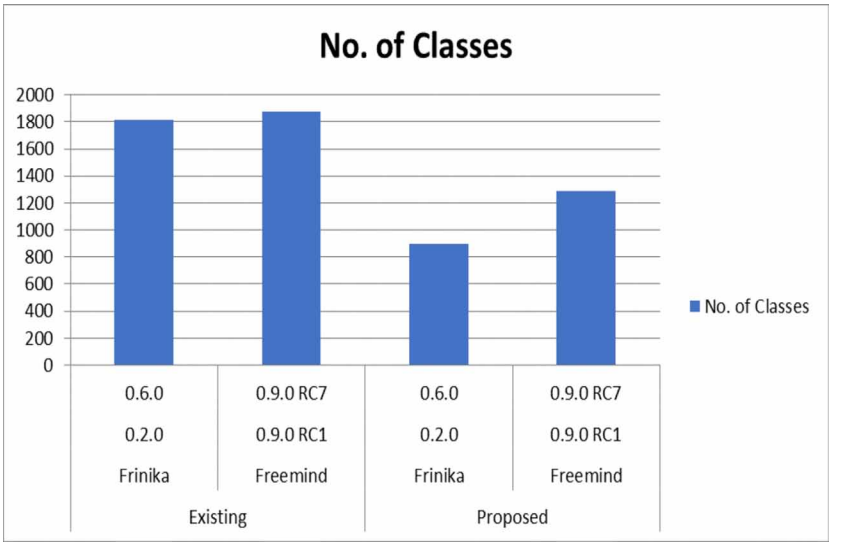


Table 2. Number of change prone classes detected using proposed and existing Malhotra and Khanna (2013) methods

Techniques	Name	Version 1	Version 2	Number of classes
Existing	Frinika	0.2.0	0.6.0	1812
	Freemind	0.9.0 RC1	0.9.0 RC7	1875
Proposed	Frinika	0.2.0	0.6.0	894
	Freemind	0.9.0 RC1	0.9.0 RC7	1284

Figure 9. Number of change prone classes predicted using proposed and existing methods



Number of change prone classes predicted using proposed approach in Frinika software 0.2.0 and 0.6.0 are 894. However, using existing methodology Malhotra and Khanna (2013), number of change prone classes were predicted as 1812. While evaluating the efficiency of the proposed technique using two dissimilar versions 0.9.0 RC1 and 0.9.0 RC7 of Freemind software, the number of classes identified as change prone classes are 1284. However, the number of change prone classes were 1875 using existing technique for the Freemind software versions 0.9.0 RC1 and 0.9.0 RC7. Fig. 7.3 depicts the identification of number of classes as change prone using the proposed and existing techniques for two different versions of software Frinika and FreeMind. It is evident that the proposed technique predicts change prone classes with better accuracy as the existing technique considers just Object- Oriented metrics for evaluating change prone classes, however the proposed approach considers OO metrics and other features for predicting change proneness. The performance of the proposed technique was compared with existing technique Malhotra and Jangra (2013) in predicting the change proneness using data sets obtained from three open source software of different versions Art-of Illusion and Sweet Home-3D.

Art of Illusion is complete software that has all the features of 3D modelling, rendering, and animation studio. It is implemented completely in Java, and can work on all operating system. Sweet Home-3D is application software that provides interior design to draw house plans, arrange furniture and view the results in 3D. Source code for these software is available on website www.sourceforge.net. Table 3 mentions the number of change prone classes predicted using proposed and existing method.

From Figure 10, the classes exhibiting change in Sweet Home-3D software version 3.6 and 3.7 using the proposed technique was found to be 200 and the classes without change was 743. But, the existing technique detected 15 classes as classes exhibiting change and 333 classes without change for the two versions of Sweet Home-3D software version 3.6 and 3.7. While evaluating the performance of the proposed technique using two different versions 2.7 and 2.9.2 of software Art-of Illusion, the number of classes exhibiting change was 167 and classes without change was 1005. However, the numbers of classes with and without change were 131 and 303 respectively, using the existing technique for the 2.7 and 2.9.2 of software Art-of Illusion. The prediction of change prone classes of proposed system was better than the existing system. As a summary of this evaluation, the proposed approach for change proneness prediction can be used in intra system scenarios to predict change prone classes.

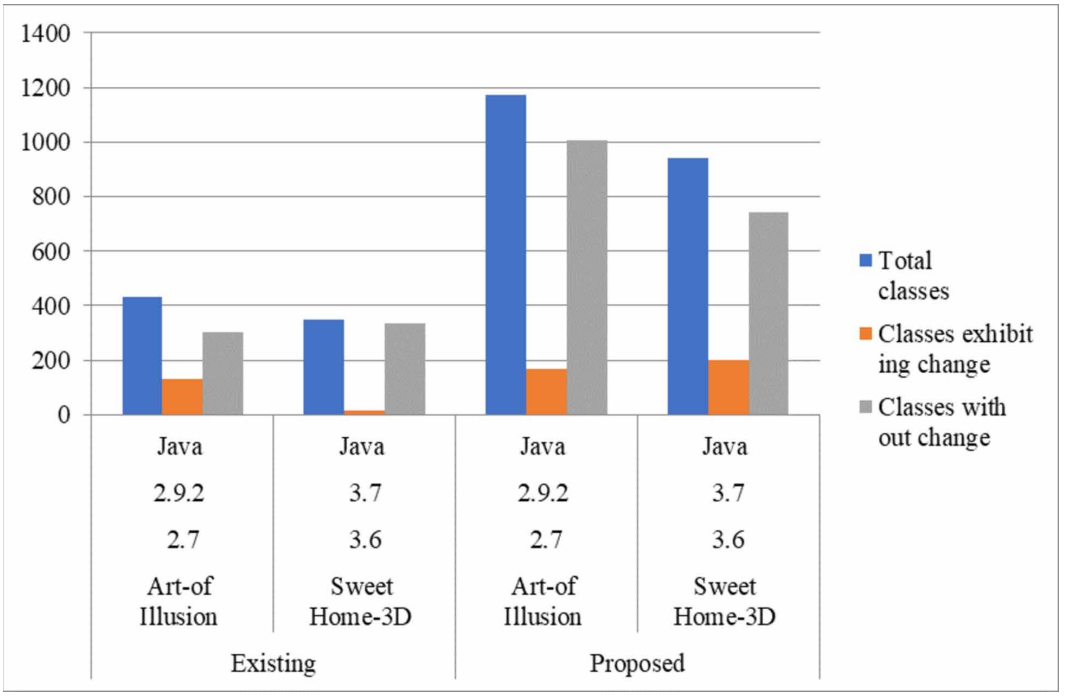
CONCLUSION

Early detection of change prone classes benefits software developers and maintainers such as they can devote significant time and effort on such sensitive classes. Keeping this view in mind, this paper made an attempt to understand use of UML diagrams in software development life cycle. Further, discussed various types of behavior diagrams in UML such as use case diagrams, interaction diagrams, state–chart diagrams, and activity diagrams. Because of the simplicity of UML class diagrams, research suggests software developers to use UML class diagrams for analyzing the types of dependencies that exists

Table 3. Number of change prone classes detected using proposed and existing Malhotra and Jangra (2013) methods

Technique	Software	Version 1	Version 2	P/L used	Total classes	Classes exhibiting change	Classes without change
Existing	Art-of Illusion	2.7	2.9.2	Java	434	131	303
	Sweet Home-3D	3.6	3.7	Java	348	15	333
Proposed	Art-of Illusion	2.7	2.9.2	Java	1172	167	1005
	Sweet Home-3D	3.6	3.7	Java	943	200	743

Figure 10. Number of change prone classes detected using proposed and existing methods



between classes. For getting a better understanding of UML class diagrams, a detailed study of these diagrams is given in above sections of this paper. Paper highlighted various types of relationships that exists between classes of UML class diagrams. And discussed advantages of using UML class diagrams in prediction of change prone classes. This research extends the concept of behavioral dependency in this paper and suggested six types of behavioral dependencies: direct behavioral dependency, indirect behavioral dependency, internal behavioral dependency, external behavioral dependency, indirect internal behavioral dependency and indirect external behavioral dependency. Various definitions and algorithms for computing different types of behavioral dependency is discussed in this paper. With the help of these dependencies, UML class diagrams can help in finding how much a class is dependent on other class, such that in next release of software such sensitive classes can be more focused for efficient resource allocation and timely completion of software. Research aims to give predictions to next version of software.

Different types of behavioral dependencies given in this paper evaluates how much a class is dependent on another class. And when a change is made in a class how it impacts other related classes. Evaluating various types of given dependencies helps in predicting how change propagates amongst classes. Paper mainly discussed about spread changes and prediction of these spread changes using various types of dependencies at a detailed level.

This paper provides description of the most significant feature of change proneness prediction model, evaluation of these dependencies classifies a class into dependent and depending class. Further, for the evaluation of these dependencies a dependency model is given in paper.

Applications of the proposed research can be in the field of Reverse Engineering so that the maintenance phase takes less time and becomes easier. Secondly the study can be used to identify the classes in two groups that is change prone and non-change prone class, so that the prone classes can be more focused.

Future Scope

Maintenance phase of Software Development Life Cycle takes around 65-70% percent of time. If change prone classes are identified in the initial phases of the lifecycle model. It becomes easier to handle such classes, in an effective manner. Even the process of Reverse Engineering becomes easier, as when the change is demanded by customer it can be easily handled. This paper gives an approach of identifying change prone classes by finding behavioral dependency between classes. In future, a model can be built in which more object oriented features can be combined with behavioral dependency to predict the proneness of a class.

REFERENCES

Abdeen, H., Bali, K., Sahraoui, H., & Dufour, B. (2015). Learning dependency-based change impact predictors using independent change histories. *Information and Software Technology*, *67*, 220–235. doi:10.1016/j.infsof.2015.07.007

Al-Kadeem, R., Backar, S., Eldardiry, M., & Haddad, H. (2017). Review on using system dynamics in designing work systems of project organizations: Product development process case study. *International Journal of System Dynamics Applications*, *6*(2), 52–70. doi:10.4018/IJSDA.2017040103

Arisholm, E., Briand, L. C., & Fyen, A. (2004). Dynamic Coupling Measurement for Object-Oriented Software. *IEEE Transactions on Software Engineering*, *30*(8), 491–506. doi:10.1109/TSE.2004.41

Bura, D., & Choudhary, A. (2020). A novel change impact model for enhancing project management. *International Journal of Project Organisation and Management*, *12*(2), 119–132. doi:10.1504/IJPOM.2020.106373

Bura, D., & Choudhary, A. (2020). Enhancing Information Retrieval System Using Change-Prone Classes. In *Critical Approaches to Information Retrieval Research* (pp. 40–68). IGI Global. doi:10.4018/978-1-7998-1021-6.ch003

Bura, D., Choudhary, A., & Singh, R. K. (2017). A Novel UML Based Approach for Early Detection of Change Prone Classes. *International Journal of Open Source Software and Processes*, *8*(3), 1–23. doi:10.4018/IJOSSP.2017070101

Elish, M., & Zouri, A. (2014). Effectiveness of Coupling Metrics in Identifying Change-Prone Object-Oriented Classes. *Proceedings of the 2014 International Conference on Software Engineering Research and Practice (SERP'14)*, 44-50.

Eski, S., & Buzluca, F. (2011). An Empirical Study on Object-Oriented Metrics and Software Evolution in Order to Reduce Testing Costs by Predicting Change-Prone Classes. *IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW),* 566 – 571.

Galli, B. J. (2020). Application of Systems Engineering to Risk Management: A Relational Review. *International Journal of System Dynamics Applications*, *9*(2), 1–23. doi:10.4018/IJSDA.2020040101

Garousi, V., Briand, L. C., & Labiche, Y. (2006). Analysis and visualization of behavioral dependencies among distributed objects based on UML models. *Proceedings of the 9th international conference on Model Driven Engineering Languages and Systems*, 365-379. doi:10.1007/11880240_26

Godara, D., Choudhary, A., & Singh, R. K. (2018). Predicting Change Prone Classes in Open Source Software. *International Journal of Information Retrieval Research*, *8*(4), 1–23. doi:10.4018/IJIRR.2018100101

Godara, D., & Singh, R. (2014). A new hybrid model for predicting change prone class in object oriented software. *International Journal of Computer Science and Telecommunications*, *5*, 1–6.

Godara, D., & Singh, R. K. (2014). A review of Studies on Change Proneness Prediction in Object Oriented Software. *International Journal of Computers and Applications*, *105*(3).

Godara, D., & Singh, R. K. (2014). *Implementation of UML2. 0 Based Change Proneness Prediction in OO Software through Dependency*. Academic Press.

Godara, D., & Singh, R. K. (2014, September). Improving change proneness prediction in UML based design models using ABC algorithm. In *2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI)* (pp. 1296-1301). IEEE. doi:10.1109/ICACCI.2014.6968320

Godara, D., & Singh, R. K. (2014). Understanding change prone classes in object oriented software. *International Journal of Computers and Applications*, *107*(1).

Godara, D., & Singh, R. K. (2015). Enhancing Frequency Based Change Proneness Prediction Method Using Artificial Bee Colony Algorithm. In *Advances in Intelligent Informatics* (pp. 535–543). Springer. doi:10.1007/978-3-319-11218-3_48

Godara, D., & Singh, R. K. (2017). Exploring the relationships between design measures and change proneness in object-oriented systems. *International Journal of Software Engineering*, *Technology and Applications*, *2*(1), 64–80.

Han, A. R., Jeon, S. U., Bae, D. H., & Hong, J. E. (2008, July). Behavioral dependency measurement for change-proneness prediction in UML 2.0 design models. In *2008 32nd Annual IEEE International Computer Software and Applications Conference* (pp. 76-83). IEEE.

Han, A. R., Jeon, S. U., Bae, D. H., & Hong, J. E. (2010). Measuring behavioral dependency for improving change-proneness prediction in UML-based design models. *Journal of Systems and Software*, *83*(2), 222–234. doi:10.1016/j.jss.2009.09.038

Herrera, M. M., Carvajal-Prieto, L. A., Uriona-Maldonado, M., & Ojeda, F. (2019). Modeling the Customer Value Generation in the Industry's Supply Chain. *International Journal of System Dynamics Applications*, *8*(4), 1–13. doi:10.4018/IJSDA.2019100101

Inpirom, A., & Prompoon, N. (2013, May). Diagram change types taxonomy based on analysis and design models in UML. In *Software Engineering and Service Science (ICSESS), 2013 4th IEEE International Conference on* (pp. 283-287). IEEE. doi:10.1109/ICSESS.2013.6615306

Lee, S. J., Lo, L. H., Chen, Y. C., & Shen, S. M. (2016). Co-changing code volume prediction through association rule mining and linear regression model. *Expert Systems with Applications*, *45*, 185–194. doi:10.1016/j.eswa.2015.09.023

Lu, H., Zhou, Y., Xu, B., Leung, H., & Chen, L. (2012). The ability of object-oriented metrics to predict change-proneness: A meta-analysis. *Empirical Software Engineering Journal*, *17*(3), 200–242. doi:10.1007/s10664-011-9170-z

Malhotra, R., & Khanna, M. (2013). Inter project Validation for Change Proneness Prediction using Object-Oriented Metrics. *Software Engineering*. *International Journal (Toronto, Ont.)*, *3*(1), 21–31.

Malhotra, R., & Jangra, R. (2013). Prediction & Assessment of Change Prone Classes Using Statistical & Machine Learning Techniques. *Journal of Information Processing Systems,* 1-26.

Mathur, S., Soni, A. K., & Sharma, G. (2014). C2MC: An Automated Tool of the Requirement Engineering Model for a Non Fading Data Warehouse. *International Journal of Data Mining And Emerging Technologies*, *4*(2), 111–119. doi:10.5958/2249-3220.2014.00009.3

Penta, M., Cerulo, L., Gueheneuc, Y., & Antoniol, G. (2008). An empirical study of the relationships between design pattern roles and class change proneness. *Proceedings of IEEE International Conference on Software Maintenance*, 217-226. doi:10.1109/ICSM.2008.4658070

Pradhan, P. L. (2017). Proposed Heuristics Model Optimizing the Risk on RTS. *International Journal of System Dynamics Applications*, *6*(2), 31–51. doi:10.4018/IJSDA.2017040102

Shanbhag, N., & Pardede, E. (2019). The Dynamics of Product Development in Software Startups: The Case for System Dynamics. *International Journal of System Dynamics Applications*, *8*(2), 51–77. doi:10.4018/IJSDA.2019040104

Sharafat, A. R., & Tahvildari, L. (2008). Change prediction in object-oriented software systems: A probabilistic approach. *Journal of Software*, *3*(5), 26–39. doi:10.4304/jsw.3.5.26-39

Soni, S., & Chorasia, B. K. (2017). Policy Planning in Higher Technical Education: A System Dynamic Approach. *International Journal of System Dynamics Applications*, *6*(3), 87–110. doi:10.4018/IJSDA.2017070105

Deepa Bura received her B.E. degree from Maharishi Dayanand University, Rohtak,India and M.Tech. degree in Information Technology from Guru Gobind Singh Indraprastha University, Delhi, India. She completed her Ph.D. degree from Uttarakhand Technical University, Dehradun, India. Her field of research includes Software Engineering, Database systems and Datawarehouse and data mining. She is working as Associate Professor in Manav Rachna International Institute of Research and Studies, India. She has more than 30 research papers to her credit.

Amit Choudhary is currently working as an Associate Professor and Head in the Department of Computer Science at Maharaja Surajmal Institute, New Delhi, India for the last 17 years. He has done MCA, M.Tech and M.Phil in Computer science and doctoral degree in Computer Science and Engineering from M. D. University, Rohtak, India. His research interest is focused on Machine Learning, Pattern Recognition and Artificial Intelligence. He has many international publications to his credit.