I.J. Information Technology and Computer Science, **2015, 04, 28-41**
Published Online March 2015 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijitcs.2015.04.03

A Framework for Effective Object-Oriented Software Change Impact Analysis

Bassey Isong
University of Venda, Department of Computer Science, Thohoyandou, 0950, South Africa
E-mail: bassey.isong@univen.ac.za

Obeten Ekabua
North-West University, Department of Computer Sciences, Mmabatho, 2735, South Africa
E-mail: obeten.ekabua@nwu.ac.za

Abstract**—** Object-oriented (OO) software have complex dependencies and different change types which frequently affect their maintenance in terms of ripple-effects identification or may likely introduce some faults which are hard to detect. As change is both important and risky, change impact analysis (CIA) is a technique used to preserve the quality of the software system. Several CIA techniques exist but they provide little or no clear information on OO software system representation for effective change impact prediction. Additionally, OO classes are not faults or failures-free and their fault-proneness is not considered during CIA. There is no known CIA approach that incorporates both change impact and fault prediction. Consequently, making changes to software components while neglecting their dependencies and fault-proneness may have some unexpected effects on their quality or may increase their failure risks. Therefore, this paper proposes a novel framework for OO software CIA that allows for impact and fault predictions. Moreover, an intermediate OO program representation that explicitly represents the software and allows its structural complexity to be quantified using complex networks is proposed. The objective is to enhance static CIA and facilitate program comprehension. To assess its effectiveness, a controlled experiment was conducted using students' project with respect to maintenance duration and correctness. The results obtained were promising, indicating its importance for impact analysis.

Index Terms**—** Impact Analysis, Software Change, Complex Networks, Faults, Matrix

I. Introduction

In today's software development world, object-oriented (OO) technologies are increasing gaining momentum. Currently, the technology has amass popularity worldwide in several small, medium and large software organizations and several OO software applications are in used [1,2,3,4]. OO paradigm approaches are believed to provide better maintainable and reusable systems. They proffer the benefits of producing a clean, well-understood design characterized by easy to understanding, test, maintain and extend [5]. Given the critical context, it is of the essence that the software systems are maintained effectively and efficiently if they are to continue to remain useful. Change is an indispensable property of software which plays a central role in its evolution [6]. Software often undergoes changes during development or life-time in order to fix faults, add new features and enhance internal code quality of the system [7,8]. Despite the benefits, changes come with possible high risks. Regardless of the change size, changes have the ability to introduce unanticipated side-effects and errors elsewhere in the system, degrade the quality of software or cause the software to fail [8,9]. In real-life software maintenance, the situation is worsened especially if the program dependencies are ignored. The fact remains that making changes to software components while neglecting their dependencies and fault-proneness may have some unexpected effects on its quality or may increase their risks to fail [8,9].

With the exponential growth in the size and complexity of today's software applications, maintenance tasks have been quite challenging. Changes are performed successfully when there is a good comprehension of the system's component dependencies as well as their fault-proneness probability which are vital to avoid unintended effects in the system [7,9]. Software change impact analysis (CIA) is a technique that is used to identify or estimate the consequences of a proposed change impact through the analysis of software product [8]. It is used to curb the risks and costs associated with unidentified effects of changes. In the perspective of OO software maintenance, the paradigm's acclaimed benefits do not pledged software quality on its own, guard against developer's mistakes or prevent faults and failures. Features that are specific to OO software such as encapsulation, inheritance, polymorphism and dynamic binding often time affect their maintenance. The complexity offered by these features often makes it cumbersome to pinpoint the impact of changes or it is likely that they might introduce some types of faults which are difficult to detect. Consequently, the ripple-effects of changes or errors in one part of the system may spread to other unchanged parts via the various complex dependencies. All these could lead the maintainer spending huge amount of time and efforts trying to locate the source of the failing effect.

There are several CIA approaches that exist as well as fault prediction models to predict the fault-proneness of high risks classes, especially for large software systems.

However, these CIA approaches provide little or no information on how to represent OO program for effective maintenance. Furthermore, OO classes are not faults or failures free [1,2,3,4,10,11,12,13,14]. Obviously, the two activities are carried out separately and there are no known approaches of CIA that incorporates both change impact and fault prediction. Thus, the intuition is that if a fault-prone class is changed without fixing the existing faults, it may increase the efforts and costs of the maintenance or could lead to software failure. In the realm of project management, time, cost and scope constitutes the three "stalagmites" where quality is the goal [1][12]. As faults during development are inevitable, the earlier they are found and fixed, the lesser it costs and the higher the quality of the products delivered [1,2,3].

This paper therefore proposes a framework for an effective OO software CIA that will assist software maintainers to carry out maintenance effectively. The objective is to improve static CIA technique in order to reduce maintenance efforts and cost in terms of faults and change impact prediction. In addition, the paper proposes the use of complex networks to build an intermediate representation (IR) of the entire OO program which will explicitly reveals its implicit structures and dependencies. The effective representation of OO program through the IR is of the essence in facilitating program comprehension and CIA while preserving the quality of the software with less cost in terms of time and effort. To assess the effectiveness of the IR for CIA, it was evaluated using students' project in terms of maintenance duration and correctness and the results obtained were promising, indicating that IR is efficient for CIA.

The rest of this paper is organized as follows: Section II is CIA, III and IV is the proposed CIA framework and IR respectively. Section V is the empirical evaluation, VI is the discussion and VII are the validity threats while VIII is the conclusion.

II. Software Change Impact Analysis

Software changes are both important and risky when they are made. CIA is a technique that is often used to preserve the quality of the affected system. According to Bohner and Arnold [8], CIA is defined as the:

"....determination of the potential effects to a subject system resulting from a proposed software change".

It is a process that is used to quantify which software component will be affected by a change proposal or likely to be changed when a component is changed. CIA underpinning principle stemmed from the believe that, irrespective of the change size, they have the ability to introduce unanticipated side-effects, errors elsewhere in the system, degrade the quality of software or cause the software to fail [5,6]. In particular, changes that are carried out frequently can destroy the architecture of the software or even increases source code and architecture inconsistency. In this case, CIA is used by engineers to allow for more effective prioritization of change requests, accuracy resource estimation, development schedules, and to reduce the amount of corrective maintenance by reducing the number of errors introduced as a by-product of the maintenance effort [6,7,8]. The process that is used to achieve CIA is captured in Fig. 1.

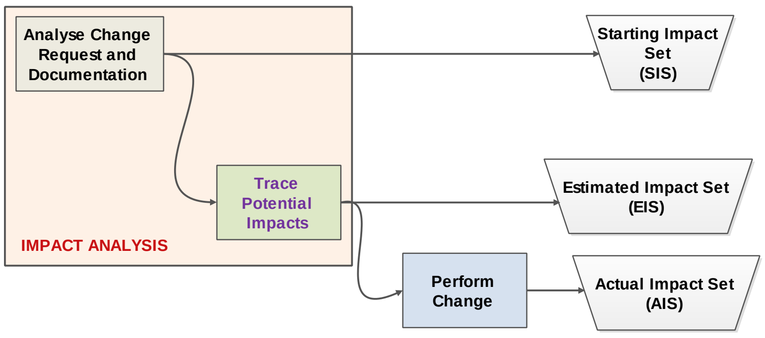
Analyse Change Request and Documentation
Trace Potential Impacts
IMPACT ANALYSIS
Starting Impact Set (SIS)
Estimated Impact Set (EIS)
Perform Change
Actual Impact Set (AIS)

Fig. 1. Impact analysis process

The process is performed iteratively and is applied to discover both direct and indirect impacts of changes. In Fig. 1, the inputs are the *change set* that originates from the change proposal while the outputs are the *impact set* [7]. For instance, change set elements at the source code level would have computed impact set such as classes, methods/functions and fields depending on the level of granularity employed. The activities that are performed during the course of CIA are the SIS, EIS and the AIS [6,7]. Existing CIA approaches are the static [14][15], dynamic [14,15] or hybrid approaches [15]. Static CIA is based on call or program dependencies graph which is known to be safe but less precise with the generation of large impact set [7]. On the other hand, dynamic CIA computes impact set based on the information collected during the execution, more precise but with less safe when compared to static approach [7].

With the CIA process discussed above, it is quite clear that the process is specifically used to predict the impact of changes while components' fault-proneness is not taken into account before the actual changes are made. Since OO software components are not fault or failure free, the position of this paper is that, if changes not meant to fix existing components' faults are made, they could create some undesirable effects or increase the likelihood of the software to fail. This is therefore, the basis for this paper.

III. The CIA Framework

During software development, changes are made to realize various change proposals of software systems. Based on the change proposal, the task of the maintainer is to analyze and evaluate the system in order to effectively predict the impacts of the change. However, it has been revealed that about 70% of the total development cost of a system is expended on maintenance [5,6]. Moreover, OO software components have complex dependencies that often time adversely impact maintenance and their components, classes in particular, are not exempted from being faulty. Hence, it is vital that during CIA and before actual changes are implemented, change impact prediction be performed along affected components' fault-proneness prediction. This is necessary to ensure that the risks and cost of the change implementation are reduced to the minimum or eliminated. Predicting faults early would allow mitigating actions to be focused on the high risks components or take alternative actions before changes are made.

A. Description

The proposed framework incorporates two activities: impact and fault prediction for OO software and is dependent on the software system size. This framework is unique and it involve activities of components analysis and complex dependencies extraction, change impact analysis, early faults or failure prediction and change implementation (See Fig. 2). The goal is to proffer guidance to the software maintainer when maintaining OO software.

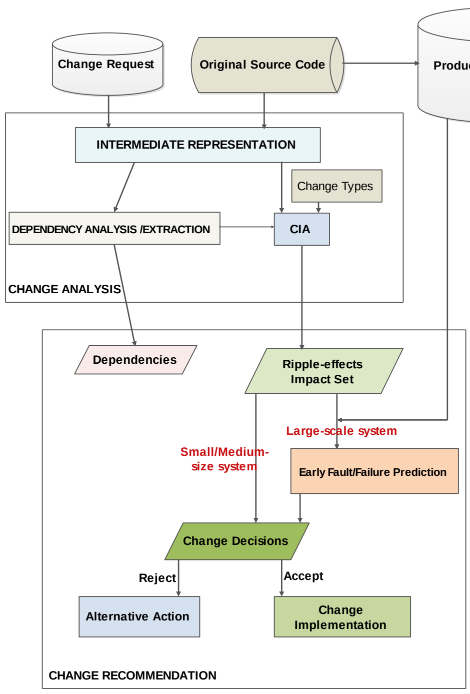

Fig. 2. Proposed CIA framework

The details are discussed as follows:

1) *Dependencies Analysis and Exttraction:* This is the first stage which is aimed at facilitating OO program comprehension and effective CIA. On the proposed framework, the original OO source code has to be analyzed by constructing an intermediate source code representation (IR). The IR should be simple and clearly reveals all the possible components (*classes*, *methods* and *fields*), and their dependencies (*inheritance*, *membership*, *invocation* and usage) [14]. It should also permit the quantification of the overall program complexity. The essences is to provide a good understanding of how components relate to one another and to facilitate CIA activities in the next stage. The representation is based on the complex software networks. The goals in this stage is to assist maintainers to:
   - Visualize the structure and dependencies of the system,
   - Compute the degree of components' coupling,
   - Determine the impact of a change alongside dependencies and impact diffusing of change types, and
   - Quantify the risk propagation of each component with respect to fault in small sized systems.

These activities will help the maintainer to take appropriate decisions and actions during the course of CIA in the later stage.

2) *Change Impact Prediction:* After the construction of the IR, the next and crucial task is to perform the actual CIA. The objective is to help the maintainer quantify or determine which OO software components in the original software systems will truly be affected by the change proposal or which will bring inconsistencies to the software if changes are made. With the IR, this stage ensures that the impacts of changes are localized as possible. Based on the nature of OO software, we have proposed a technique called *impact diffusion* which will be used to precisely predict the impact of changes. Thus, the *impact diffusion* is based on three influential factors:
   - The type change performed on the object components,
   - The type of dependencies that links one component to another, and
   - The behavior and impact range of each change and the type of dependencies.

The rationale in this case is that, in OO program unlike non-OO program, the effect of changes are dependent on the change type performed and the nature of the dependencies between the components affected by the changes. These determinant factors are to be taken into account in order to precisely predict the effect of a change and to allow decisions to be taken as early as possible on whether to implement or reject a change. The goal in this case is to improve the accuracy and precision of the predicted *impact set* which is the output of the stage.

3) *Early Fault/Failure Prediction:* With the impact set at hand, the goal of this stage is to determine the affected components or the predicted *impact set* for fault-proneness or which of them may lead to failure when changes are made. This prediction is based on

probability and the process is based on the size of the system. Since OO software systems are of different sizes: small, medium and large, we recommend an approach that will be applied when maintaining the systems.

a) *Small/medium sized systems:* For small or medium sized systems, the quality of the components identified as impact set can be assessed by computing the probability of fault propagation using their dependencies in the *complex software networks*. In this case, the risks components pose to other components they connect to are computed and the value obtained is used to take decisions during change implementation. With the computed values, the higher the probability the higher would the risk of the fault propagation be. In the same vein, a smaller risk value would signify a fault in such component poses no serious impact on the other components and modification can be performed hitch-free. The knowledge of the risks value will assist the software maintainer to take extra care during the course of implementing the actual change.
b) *Large-scale systems:* In the perspective of large OO software systems, using the complex software networks might not be appropriate. In this case, the quality of the systems can be assessed via pure prediction using software metrics such as code metrics, past change and fault histories as well as suitable fault prediction model. Several empirical studies in the literature have confirmed the relationship between product and process metrics and fault-proneness [10,11,12]. To carry out the prediction, all the measures extracted from either the previous or current version of the software stored in the database will be used to predict whether a component affected by a change will be faulty or not. The motivation is that software quality is known to play a crucial role in the success and failure of any software organization. However, in large software systems, providing high quality in development has been deemed complex and a laborious activity [12]. In this case, it is important that the available resources are focused on the most critical parts of the system to ensure customers' satisfaction. That is to say, the early identification of faulty components before changes are made is of importance for the reduction of maintenance efforts, costs and risks while preserving software quality. This will in turn facilitate software testing and inspection activities.

4) *Change Decisions and Implementation:* After identifying the impact set and assessed their overall quality, the next step is to take decisions on whether to implement the change or not. In other words, this is the acceptance or rejection stage. Deciding on whether to implement a change or not is important because, for example, if a change proposal is known to trigger significant ripple-effects over the entire system or undesirable effects and majority of the affected classes are fault-prone, one decision could be to reject the change or to consider an additional change plan or redesign the system through strategies like refactoring, or accept the change proposal. A change is only implemented if the impact and the risks are known to be small or after validation and verification activities have been performed on the affected faulty parts. Otherwise, it is rejected if it is known to have deteriorating effects on the whole system. The essence is also to reduce the cost of risky changes.

IV. The Proposed Intermediate Representation

This section discusses the proposed IR of OO program that will assist software maintainers in facilitating program understanding and CIA. The approach is based on the initial work of [14] and [16]. However, in this paper we modeled OO software system's structure using *complex software networks*.

A. Complex Software Networks

Complex networks in recent decades have gained increasing momentum and software system is not an exception as a result of their topological structure [16][17]. Software systems can be modeled as complex networks where software components are represented as nodes and their interactions as edges. The representation is possible due to the design structure of OO software which is better explained by its structural properties in terms of components and the relationships. The components are the *fields*, *methods*, *classes* and *packages* while their interactions are the different dependencies that exist between these components.

The importance of the IR stemmed from the fact that today software systems especially OO program has exponentially grown in size and complexity with structures becoming more and more complicated. In this case, changes or faults in one component often require changes/faults to several other parts in a way not anticipated. Consequently, the complex structure posed by the complex relationships makes it difficult to quantify the overall quality of the final software product. As it has been known that the better the structure of the software, the lesser would the cost of the development be, analyzing OO software system's structure using complex network will help the maintainer to achieve the following goals:

1) To visualize software components and their complex dependencies. This will help the maintainer to have an understanding of which components will be impacted by a change when a change request is considered on a component. Consequently, change will be limited to few components as possible.
2) To quantitatively analyze the quality of the entire OO program structure. This involves measuring the degree of the components in terms of coupling and their fault propagation from one component to another. Analyzing the software structure quantitatively would help the maintainer to assess software quality and the risk of faults propagation from one component to another. The essence is to enable a maintainer take some mitigating actions where necessary in order to

reduce the cost of software failure when making changes.

B. OO Component Dependencies Networks

The proposed IR is called the OOComDN and is used to represent OO software components and their relationships. In the OOComDN, the components are the nodes and the interaction or relationships between every pair of the components is a *weighted* directed edge with an edge type indicating the probability that a change or fault in one component may propagate to the other component. In this paper, OOComDN is twofold: *change* and *fault* diffusion networks.

1) *Change diffusion networks:* In the change diffusion network (CDN), OO software system is represented using a *weighted* directed graph, G where components are the vertices and the dependencies among the components are the edges, taking both the semantics and syntactic structure into consideration. It is used to represent the software components and their relationships for effective maintenance, perhaps, CIA. It explicitly represents the structure and the dependencies in the OO program source code which will be used to quantifying the components that are truly affected by a change. In other words, the representation is basically used to discover the evolution mechanism of the OO software system.

a) *Dependencies types:* In this study, we identified four types of dependencies, $\mathbf{D^{Type}}$ that exist in OO program: inheritance (H), usage (U), invocation (V), and membership (M) [9,14]. They constitute one of the determining factors of change ripple-effects. Their details are discussed as follows: Given an OO program with two classes C_1 and C_2, methods m_1 and m_2 and fields, f, the dependencies that may exist are as follows:

- *Inheritance (H):* ***H*** exists if: C_2 inherits from C_1, C_1 inherits from C_2 or C_2 indirectly inherits from C_1.
- *Usage (U):* ***U*** exist if: C_1 uses C_2, C_1 aggregates or contains C_2, or C_1 aggregates or contains C_2 by value or reference.
- *Invocation (V):* ***V*** is the type of dependencies between methods, m of a class. If m_1 and m_2 are methods in a class, therefore, V exists if: m_1 calls m_2 or m_1 overrides m_2 and so on.
- *Membership (M):* ***M*** is one that exists between the class and its member. That is, dependencies between the class and its members (methods and fields).

These dependencies are the non-numeric weight assigned to the edges of the OOComDN-1 and constitutes the links by which a change or fault transmits from one component to other once a change is consider on a specific component. Based on the CDN and the $\mathbf{D^{Type}}$ the following definition of OOComDN is considered.

Definition 1: *[OOComDN -1]*
Given an OO program, P let G = <(N,D^E), D^Type > represent OOComDN given by:

$$\text{OOComDN-1} = <(\mathbf{N}, \mathbf{D^E}), \mathbf{D^{Type}}>$$

Where $N = N_{Pk} + N_C + N_M + N_F$ are the nodes and $D^E = N \times N \times \mathbf{D^{Type}}$ represents the set of various edges with dependencies types, $\mathbf{D^{Type}}$. $\mathbf{D^{Type}}$ is called the weight of the graph and N_P, N_C, N_M and N_F represent the set of packages, classes, member methods and fields respectively. Each component is represented by only one node and the weighted-directed edge between two nodes indicates that a component is a member of the class or uses, invokes or inherits the other components.

b) *Typical illustrations:* A typical illustration of the OOComDN is shown in Fig. 4 using the program, P written in Java of Fig. 3. The various shapes used to represent each component in the OOComDN-1 are also shown in Fig. 4.

```
package p1;
public class A {

public A(){};
private int d;

public void M1()
{ d=2; }

public int M2(int x)
{ M1();
x= d + 10;
return x; }}

public class B extends
A {
public B() {};
private int a;

public void M3()
{ a=5; }

public int M4(int b)
{ M3();
int c = a+b+10;
return c; }}
```

```
package p2;
import p1.*;

public class C {
    public C(){};
    private p1.B k;

public void M5()
{   k.M4();  }}

class D extends C {
public D() {};

private String q;

public void M6()
{ q="Boy!";
B j ;  j.M4();
A p; p.M1();
}
}
```

Fig. 3. Sample program

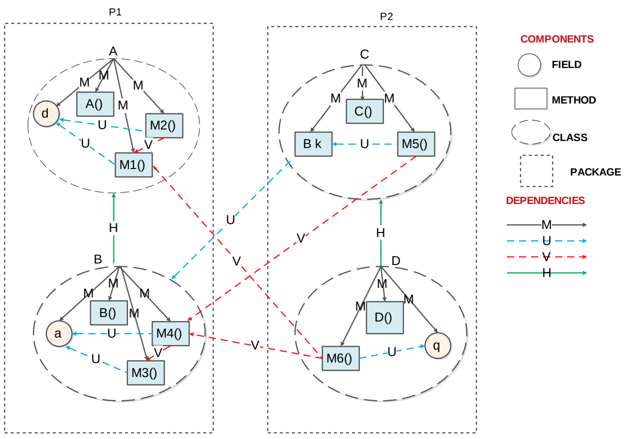

Fig. 4. OOComDN of the sample program in Figure 3

Fig. 4 shows the representation of the OO program captured in Fig. 3. In the OOComDN-1 A, B, C and D are the classes in P while ***H***, ***V***, ***M*** and ***U*** are the dependencies types. In this way, if a component says D uses or inherits or invokes a class say A, an edge would originate from the node D to node A. Furthermore, the multiplicities of these dependencies are very important and are taken into account depending on the *type of change* is performed on a given component. The weight of each directed edge will determine the probability that a change in one component say A, may or may not impact other component, D.

C. Degree of OOComDN-1

After the construction of the OO program as OOComDN-1, the next step is to compute its degree, **Z**. **Z** of a node in OOComDN-1 is the number of dependencies a component has on other components connected to it or it is connected to. Two types of **Z** exist: *in-degree* and the *out-degree*. **Z** is used to identify the degree of coupling of each component in the program as well as the structural complexity of the software at the class level. The importance is to give an insight into how components are related to one another and what need to be done to accomplish a change on a given component. This computation is only done at the class level and it is done after pruning the OOComDN-1 leaving only classes and their dependencies types as shown in Fig. 5. The formal definitions for Z are stated as follows:

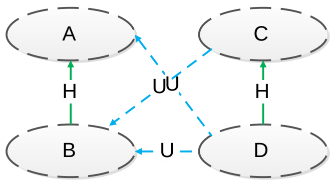

Fig. 5. Class level OOComDN-1

Definition 2: [*Degree of OOComDN-1***]**

Given, OOComDN-1, < **(N, D^E), D^{Type}***>, with an adjacency matrix* $\boldsymbol{A_{ij}}$*, the degree of a vertex,* Z_i*, we defined the out-degree of an OO program component as the number of edges or connections originating from that component. It is given by* $|Z^{out}(n_i)|$ *which is the sum of the* i^{th} *column of the* $\boldsymbol{A_{ji}}$*.*

$$Z^{\ out} = \sum_j A_{ji} \tag{1}$$

On the other hand, the in-degree of an OO software component, n_i *is the total number of edges or connections onto that node and it is given by* $|Z^{in}(n_i)|$ *which is the sum of the* i^{th} *row of the* $\boldsymbol{A_{ij}}$*.*

$$Z^{\ in} = \sum A_{ij} \tag{2}$$

$\boldsymbol{Z^{tot}}(n_i)$ *is the total number of directed edges into and out of the node,* $\boldsymbol{n_i}$ **∈N.** *It is simply the sum of* $Z^{\ in}$ *and* $Z^{\ out}$*.*

$$Z^{\ tot} = Z^{\ in} + Z^{\ out} \tag{3}$$

As stated in definition 2 above, $Z^{in}(n_i)$ would indicates the number of classes that has dependency on class n_j ∈N and $\mathbf{Z}^{out}(n_i)$ the number of classes on which class n_i ∈N depends on. The Z^{in} and Z^{out} for the program shown in Fig. 2 are captured in Table 1.

Table 1. In-degree and Out-degree in OOComDN-1 of Figure 4

Node, n_i	$Z^{\ in}$	$Z^{\ out}$	$Z^{\ tot}$
A	2	-	2
B	2	1	3
C	1	1	2
D	3	-	3

As shown in Table 1, for instance, class A has one Z^{in} for the ordered paired (B,A) and (D,A) and no Z^{out}. In addition, Z^{tot} is a measure of the overall complexity of the program. This shows the nature of coupling in A which will assist a maintainer in identifying the complexity of the classes prior to performing CIA. As the complex relationships among OO software components often lead to structural complexity of the software system as well as cognitive complexity, being similar to Chidamber-Kemerer's (CK) Coupling between Object Classes (CBO) metric [10,12], Z in the software networks would show the degree to which each class depends on other classes. Thus, we used Z to measure the degree of coupling in a small or medium sized system.

D. Dependencies Matrices

This section discusses the strategies for identifying initial *impact set* of a change during CIA on the OOComDN-1. It is based on *adjacency matrix* representation. The objective is to provide a high-level identification of the relationship between the classes or members of the original program. That is, the designed will assist in the identification of the **SIS** with respect to the change proposal. The correct identification of **SIS** is crucial to the correct computation of the **EIS** which is geared towards improvement of the overall precision. The strategy involves the transformation of the OOComDN-1 into three separate dependency matrices:

- Class dependency matrix (CDM),
- Intra-membership relation matrix (MRM), and
- Inter-membership relation matrix (IRM).

With these matrices, CDM is a high-level matrix that is extracted from the high-level structure of the entire system that is only composed of classes and their dependencies (See Fig. 5), while MRM and IRM are extracts of the CDM which involve class members' dependencies. Dependency and relation are used in the matrices to denote class-to-class relationship and member-to-member relationship respectively. Each matrix is explained as follows.

1) *Class dependency matrix:* CDM is an adjacency matrix representing both dependencies and relations among different classes of OO program. Based on the different source code change type of OO programs, it is obvious that changes are not only limited to class members but also to other classes and packages. Thus, CDM is used for the basis of class changes. The following definition is given:

Definition 3: *[Class dependency matrix (CDM)]*

Given the ***OOComDN-1****, two classes* ***A, B*** *∈* ***N*** *for instance, we define CDM as follows:*

$$\text{CDM}= [M_{ij}]=\begin{cases} - = 1 & if\ there\ exist\ intra - CD, A \in N \\ + = 1 & if\ there\ exist\ inter - CD,\ \ A \rightarrow B \nleftarrow A, B \in N \\ 0 & Otherwise \end{cases}$$

where CD is the class dependency in the definition. The definition is three-fold and it indicates the following:

i. **M_{ij} = "-" =1** value denotes that there is a local or internal relationship within a class and its members, "-" Є D^E. The significance is that a change to a class affects the class itself. We called "-" the intra-dependency value.

ii. **M_{ij} = "+" =1** as long as i ≠j, indicates that there is external dependency "+"Є D^E for A →B. We called "+" the inter-class dependency value, indicating that a change to class B will affect A and any other classes related to it. And lastly,

iii. **M_{ij} = 0** value denotes that there is no dependency between class A and B. (see Table 2)

Table 2. Class dependency matrix for Figure 4.2

n_i/n_j	A	B	C	D
A	-/+/0	-/+/0	-/+/0	-/+/0
B	-/+/0	-/+/0	-/+/0	-/+/0
C	-/+/0	-/+/0	-/+/0	-/+/0
D	-/+/0	-/+/0	-/+/0	-/+/0

In Table 2, each matrix value shows implicitly the ***D^{type}*** (i.e. *usage, invocation and inheritance)* where the directed edges direction is from the column class to the row class.

2) *Intra and inter-class membership relation matrices:* These two matrices are used to represent the relationship between members of a class and members of other classes connected to it respectively. To understand these matrices, the following formal definitions are stated:

Definition 4: *[Intra- member Relation Matrix]*

*Given the OOComDN-1, a class, say **A= {a_1, a_2,…, a_m}ЄN** and a dependency **"-" Є N**, we thus define the intra- member relation matrix for dependency **"-"** of class **A** as follows:*

$$K_A = [k_{ij}] = \begin{cases} 1 \text{ If } a_i \text{ has relation with } a_j & 1 \leq i,\ j \\ 0\ \ otherwise \end{cases}$$

The definition is twofold:

i. K_{ij} =1 if a_i has a relationship with a_j, where a_i,a_j Є*A*; otherwise

ii. K_{ij} = 0, indicating there is no relationship between a_i and a_j.

Table 3. Intra-membership relation matrix

a_i/a_j	a_1	a_2	a_3	a_4
a_1	1/0	1/0	1/0	1/0
a_2	1/0	1/0	1/0	1/0
a_3	1/0	1/0	1/0	1/0
a_4	1/0	1/0	1/0	1/0

The intra-class membership relation matrix is used to represent the relationship within a class and its members. The matrix is captured in Table 3 and the intra-class membership relation matrix for OOComDN-1 in Fig. 4 is shown in Fig. 6.

A_i/A_j	d	A()	M1()	M2()
d	1	0	0	0
A()	0	1	0	0
M1()	1	0	1	0
M2()	1	0	1	1

B_i/B_j	a	B()	M3()	M4()
a	1	0	0	0
B()	0	1	0	0
M3()	1	0	1	0
M4()	1	0	1	1

C_i/C_j	B k	C()	M5
B k	1	0	0
C()	0	1	0
M5()	0	0	1

D_i/D_j	q	D()	M6()
q	1	0	0
D()	0	1	0
M6()	1	0	1

Fig. 6. Intra-membership relation matrixes for Figure 4

The intra-class membership relationship matrix which is used to represent the relationship between elements in each class given by: A_i/A_j, B_i/B_j, C_i/C_j and D_i/D_j for A, B, C, and D respectively. For instance, in A_i/A_j, all the zero values indicate that there is no relationship between members within the classes, while 1 indicates the presence of a relationship.

Definition 5: ***[Inter- membership Relation]***

*Given the **OOComDN-1**, two classes **A={a_1,a_2,…,a_m}**, **B={b_1,b_2, …,b_n}ЄN**, **A≠B**, and a dependency **"+" ЄD^E** for **A→B**. We then define the inter-membership relation matrix for dependency **"+"** as follows:*

$$P_{A\to B} = [p_{ij}] = \begin{cases} 1 \text{ If } a_i \text{ has relation with } b_j \\ 0 \text{ otherwise} \end{cases} 1 \le i \le m, 1 \le j \le n$$

Table 4. Inter-membership relation matrix

a_i/b_j	a_1	a_2	b3	a_n
b_1	1/0	1/0	1/0	1/0
b_2	1/0	1/0	1/0	1/0
b_3	1/0	1/0	1/0	1/0
b_m	1/0	1/0	1/0	1/0

Table 4 shows the inter-class membership relation matrix for a_i and b_j. The above definition indicates that $p_{ij} = 1$ if a_i in class A has a relation with b_j in class B, where a_i∈A, b_j∈B; otherwise $p_{ij}=0$ indicating there is no relationship that exists. The inter-class membership relation matrix for OOComDN-1 in Figure 4 is captured in Fig. 7. The relationships are given by A_i/D_j, B_i/D_j and A_i/C_j in the matrix. Like MRM, all the zero values indicate that there is no relationship between members of the two corresponding classes while the value 1 indicates the existence of a relationship.

A_i/D_j	d	A()	M1()	M2()
q	0	0	0	0
D()	0	0	0	0
M6()	0	0	1	0

B_i/D_j	a	B()	M3()	M4()
d	0	0	0	0
M1()	0	0	0	0
M2()	0	0	1	0

A_i/C_j	d	M1()	M2()
B k	0	0	0
C()	0	0	0
M5()	0	0	0

Fig. 7. Inter-membership relation matrixes for Figure 4

E. Fault Diffusion Networks

Fault diffusion network (FDN) represented just as CDN. The only difference is that the semantics of the relationship is neglected and every relationship has the same importance. FDN is used to characterize the risks a component poses on others due to the direct or indirect dependency existing between them. The rationale is that, though it is believed that a fault in one component will propagate to other components that depend on it, the case is not always true with respect to OO software systems [16]. The intuition is that, OO program class is composed of several fields and methods and a class is considered faulty if it has at least one fault emanating from either itself or its members. In this case, members of another class that depends on such faulty class do not all connect to the faulty member directly or indirectly. Hence, the propagation of fault from one component to another is based on probability. In this case, we adopt the approach proposed by [16]. The definition is stated as follows:

Definition 3: *[OOComDN-2]*

In FDN, the nodes represent the classes and a class is represented by only one node in the entire OOComDN-2. Interactions between classes are represented by directed numerically weighted edges.

Thus, OOComDN-2 can be described as:

$$OOComDN\text{-}2 = \langle N_C, D_C, P_b \rangle$$

Where $\mathbf{N_C}$ is the set of classes, $\mathbf{D_C}$ is the set of edges linking one class to another and $\mathbf{P_b}$ is the probability that a fault in a class will propagate to another. The interaction is based on the principle that, if members in class, say **D** use class members of **A**, **B**, an edge will originate from the node of the member in class **D** to the node in **A, B** and vice versa. For simplicity, in FDN, only the existence of dependency is considered while the $\mathbf{D^{Type}}$ is ignored. Additionally, the multiplicity of the dependencies regardless of how many times a class depend on another class and so on is ignored. Also, the numerical weight on each $\mathbf{D_C}$ in a class is the same which represents the probability that a fault in class will impact or spread to other classes they connects to. (see Figure 8).

Definition 4: ***[Fault Propagation Probability]***

*Let **P** be an OO program having class **i** and class **j**, where class **j** depends on class **i**. We therefore, define the probability of fault propagating from class **i** to class **j** as **P_b (i,j)** [16]. It is stated as follows:*

$$\mathbf{P_b(j,i)} = \frac{|\mathbf{CM(i,j)}|}{|\mathbf{MT_j}|} \tag{4}$$

According to [16], **CM(i,j)** is the set of members in class *j* which faults will propagate to the members in class *i*, which they are directly or indirectly linked to, thereby rendering the class faulty. On the other hand, $\mathbf{MT_j}$ is the total number of class members present in the class, *j*. They are shown as follows:

CM(D,A) = {*M1()*} and $\mathbf{MT_A}$ = {*d, A(), M1(), M2()*}

CM (D,B) = {*M4()*} and $\mathbf{MT_B}$ = {*a, B(), M3(), M4()*}

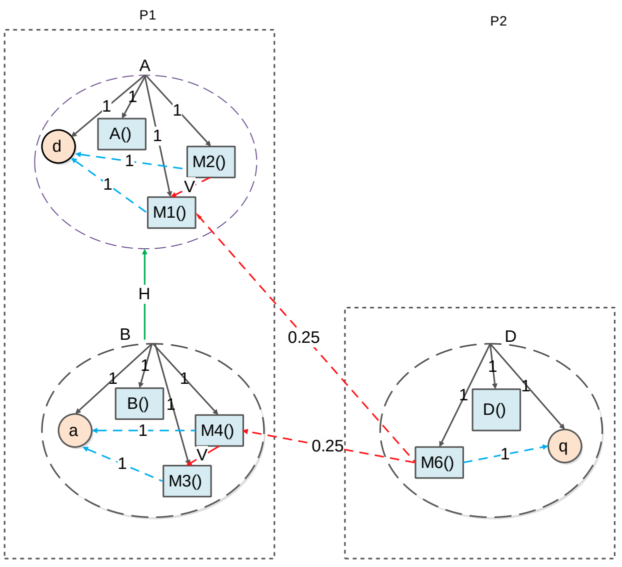

Fig. 8. Class fault propagation probability

As shown above, Fig. 8 captured the fault propagation probability in a class. The edges of all members in a class are denoted by 1. It indicates the probability that a member of the class will be faulty due to the dependency it has with a faulty member. That is, every member of a class has the same probability of being faulty if a member they depend on is faulty. However, for inter-class dependency, the case is not always true. Each class has its own probability value which is based on the number of members in that class that depends on the faulty class. For instance, as shown in Fig. 5, it is clear that class **D** depends on class **A** and **B** as follows:

(D.M6(),A) = {M1()} = D.M6() → A.M1()

(D.M6(),B) = {M4()} =D.M6() → B.M4()

Therefore,

$$\mathbf{P_b(D,A)} = \frac{|\mathbf{M1}()|}{|\{\boldsymbol{d},\boldsymbol{M1}(),\boldsymbol{M2}(),\boldsymbol{A}()\}|} = \frac{\mathbf{1}}{\mathbf{4}} = \mathbf{0.25}\text{, and}$$

$$\mathbf{P_b(D,B)} = \frac{|\mathbf{M4}()|}{|\{\boldsymbol{a},\boldsymbol{M3}(),\boldsymbol{M4}(),\boldsymbol{B}()\}|} = \frac{\mathbf{1}}{\mathbf{4}} = \mathbf{0.25}$$

The above computation is based on equation 4 where P_b(D, A) = P_b (D, B) = 0.25, 25%. This shows that, since M6() in class D depends on class A and B, the probability that a fault in class A or B will impact class D is only 25%. For inheritance dependency type, the probability will not be computed because members in the classes are not connected directly. The computation is based on the fact that, the higher the probability, the higher the risk of the fault propagation would be. Accordingly, a smaller risk value signifies that a fault in the measured component poses no serious impact on the other components and modification can be performed hitch-free. This idea stemmed from the fact that, if a class in which other classes depend on is faulty and was not detected before a change not meant to fix it is made, there is the probability that the faults may propagate to other components connected to it.

To avoid such problem, it is important that during CIA, the risks propagation probability of all the affected classes identified as *impact set* should be computed before actual changes are made. The approach will assist the maintainer to quantitatively measure the structural quality of the software through the assessment of the potential risks. The essence is to allow the maintainer know which components affected by a change proposal will have a higher risk probability of transmitting faults to its neighbors during changes. It will in turn allow mitigating actions to be focused on those high risk components in time to avoid the cost of software failure.

V. Empirical Evaluation

In this section, we present the results of the empirical evaluation performed to assess the effectiveness and significance of the IR for facilitating CIA. In this study, only OOComDN-1 was evaluated. Details are discussed in subsequent sections.

A. Setting, Subjects and Tasks

In this study, we performed a controlled experiment using small-size systems developed by students in one of their semester's projects. The subjects were only undergraduate Computer Science students of our department and the study was in fulfillment of the Software Engineering curriculum with a focus on software maintenance techniques. The subjects in their final year of study were divided into nine groups (A, B, C, D, E, F, G, H and I) of five students each and each student had comparable levels of education and experience in software development, java programming in particular. For each team selected, measures were taken to blend each team with the required skills needed. In order to be effective in carrying out maintenance, subjects had a week of theoretical knowledge of software maintenance, the basic knowledge needed for CIA using IR of OO program and others. The goal of the controlled experiment was to demonstrate whether a good and effective representation of OO program can increase the understandability of the maintainer to perform modification tasks correctly and efficiently. In this case, to be able to maintain and change a system efficiently and correctly, the maintainer has to have an in-depth understanding of the systems' structure (source code). By *efficiency*, we mean the minimum time taken to carry out the change while *correctness* is the intended functionality and less side-effects of the change.

The characteristics of the system collected from the subjects are Team A, D, F, H, and I system's had 5 class each while team B, C, E, and G 6 classes each. The maintenance task was to perform modification task on other team's system. There were four maintenance tasks the subjects performed during the course of the experiment:

- $MTask_1$ - *one class change,*
- $MTask_2$ - *one class change,*
- $MTask_3$ - *two methods change,* and
- $MTask_4$ - *one field change.*

The changes were based on the different change types applicable for OO program [14]. An overview of the experiment design is captured in Fig. 9.

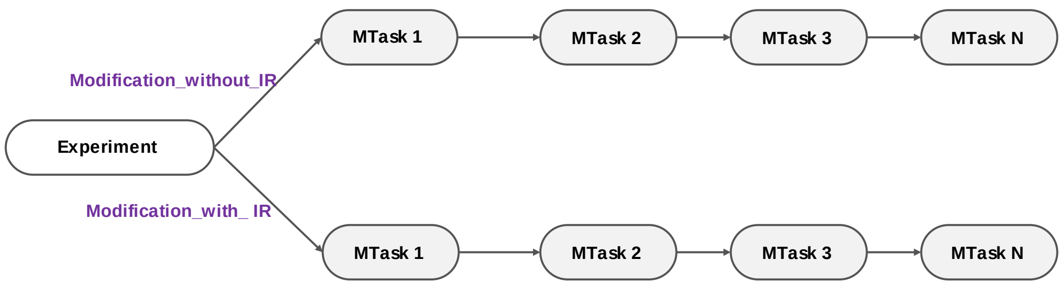

Fig. 9. Experimental design overview

B. Experimental Variables

During the course of the experiment, the variables that were of importance at each phase of the maintenance task are the change duration, program correctness, the number of errors the change introduced and the task phase. The change duration (CD) was computed by finding the difference between the starting and finishing time of the modification task. The program correctness (PC) was computed by grading each team with a grade between 0-100% based the outcome of the tasks and the correct program execution while the number of errors (NoE) was computed by counting the errors introduce by the modification task after the changes were made via recompiling the program. In this case, NoE were computed based on the number of lines affected as indicated on the development IDE used. These were all performed by the supervisor and the team members. Lastly, for the TaskPhase, two variables were important: *modification without IR* or *modification with IR* (MTask1- MTask4), (See Fig. 9).

Due to the programming skills of the subjects, we first assessed the each team's program for actual amount of time and complexity of classes that would be impacted by each change and the approximate time required to carry out the tasks. This was necessary in order to quantify the degree of difficulty of the change tasks. However, the results we obtained from the experiment put forward that this approach was adequately appropriate in this regard.

C. Study Hypotheses

In this study, hypotheses were tested in the experiment to assess the significance of the IR to CIA during the maintenance task. Thus, the null hypotheses of the experiment were as follows:

Impact of TaskPhase on Change Duration (CD):

$H0_1$: The time taken to perform maintenance task is equal for modification without IR and modification with IR.

Impact of TaskPhase on Number of Error Introduced:

$H0_2$: The number of error introduced in a changed program is equal for modification without IR and modification with IR.

Impact of TaskPhase on Program_Correctness (PC):

$H0_3$: The correctness of the program after maintenance task is the same for both modification without IR and modification with IR.

For the effect on duration (CD), the test was to evaluate if using IR constitutes a time wastage or not on the part of the maintainer while the effect on correctness (PC) would be to evaluate if using IR during maintenance contributes to program understanding or not. In this case, if correctness is equal for both, then it is not useful for CIA. However, if the program correctness is more for *modification with IR* than *modification without IR*, then it is useful for CIA and facilitates program comprehension. Furthermore, for NoE, the task would be to test if the number of errors introduced after modification is equal in both case or not. If it is lower with the TaskPhase, *modification with IR,* then it is useful, otherwise not useful for CIA.

D. Statistical Technique

In this study, we used the paired-sample T-test called the dependent T-test statistical technique to test the hypotheses stated in subsection C of section V. The choice of the dependent T-test statistical technique stems from the fact that it is used to analyze paired scores to determine if a difference exists between them. It compares measurements from the same participants by using two different measurement approaches. That is, it proffers a flexible approach for measuring the effectiveness of two different techniques using the same participants. *Modification_without_IR* and Modification*_with_IR* are the measurement techniques that were used in this study.

All the variables specified were normally distributed. We used the Shapiro-Wilk Test since it is appropriate for small sample sizes, say less than 50 (< 50). There were no transformations performed on the variables since they have no potential negative effect. The model specification is captured in Table 5. In the event that the underlying assumptions of the models are not violated, the related null hypothesis will be rejected if the presence of a significant model term corresponds to $p \leq 0.05$.

Table 5. Statistical technique specification

Variable	Distribution	Model Term	Use of Model Term
Duration	normal	TaskPhase	Test $H0_1$
Number of Errors	normal	TaskPhase	Test $H0_2$
Program Correctness	normal	TaskPhase	Test $H0_3$

E. Results Analysis

The main results obtained based on the task phases: *modification without IR* and *modification with IR* for MTask1 – Mtask4 are visualized in Fig.10 and Fig. 11 respectively. The change duration, % program correctness and a count of error are shown on the Y-axis, while the project group is shown on the X-axis. With these results, there are clear indications that TaskPhase affect the CD, PC and NoE in the two phases. For instance, it shows that a small amount of time was utilized to implement a change in the program when IR was used in phase II than when IR was not used in phase I. In the same vein, the correctness of the program was better when IR was utilized during the modification task and the same result is applicable to NoE introduced in both phases. However, for practical importance, it is essential to see if these differences are significant. To achieve this, the above stated hypotheses were tested.

As specified earlier, the paired-sample T-test was employed to test the hypotheses. The results obtained from the hypotheses testing with respect to the CD, PC and NoE for the modification tasks ($MTask_1$-$MTask_4$) in both phases are captured in Table 6. The results indicate that TaskPhase does have a significant effect on the program correctness, change duration and number of errors introduced. The level of significance used was $p \leq 0.05$.

Table 6. Dependent T-test results

Paired variable	T	DF	P-value Sig.
CD - CDII	-8.541	8	0.000
NoE - NoEII	10.509	8	0.000
PC - PCII	5.646	8	0.000

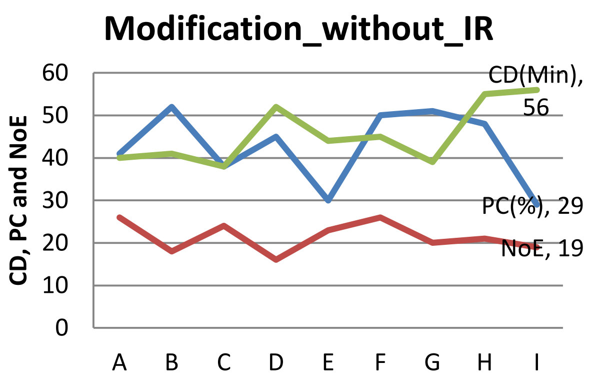

Fig. 10. Effect of TaskPhase on modification without IR

The summary of the results of the hypotheses tests is as follows:

- For the impact of TaskPhase on CD, we rejected $H0_1$ since p-value $\approx 0.00 \leq 0.05$.
- For the impact of NoE introduced, we rejected $H0_2$ since p-value $\approx 0.00 \leq 0.05$, and lastly,
- For the impact of TaskPhase on PC, we rejected $H0_3$ since p-value $\approx 0.00 \leq 0.05$.

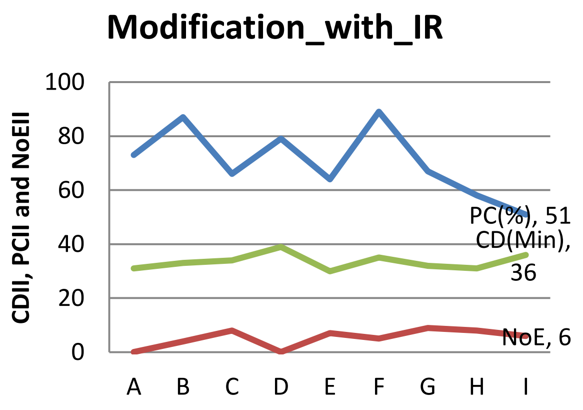

Fig. 11. Effects of TaskPhase on modification with IR

In conclusion, at the significance level of $\alpha = 0.05$, there exists enough evidence that there is a huge difference in the mean CD, PC and NoE of both phases of the of maintenance tasks (*modification without IR* and *modification with IR*). These results therefore, demonstrate that the IR of OO program is effective and useful in the facilitation of CIA.

VI. Discussion

The results obtained from the experiment performed in this work seem very interesting in terms of duration, program correctness and the number errors introduce after change were implemented in phase II. As shown in Fig. 10 and 11 respectively, it is obvious that the time taken by the subjects to perform the maintenance task in phase II (36 min maximum) were significantly smaller than the modification duration of phase I (56 min maximum). Accordingly, the correctness of the maintenance task (correct solutions) was significantly higher for phase II (56% minimum) than for the phase I (51% minimum). Moreover, the number of errors introduced after the changes were made was significantly lower for phase II (6 maximum) when the *modification with IR* was used as opposed to *modification without IR* (19 minimum).

The results further suggest the effectiveness of the IR for CIA. With these results, it is quite clear that using the IR of OO program during CIA will actually reduce the time needed to make changes by effectively identifying components affected by a change and their dependencies, the correctness of the solution and the number of errors that will be introduced after the change. Accordingly, the interpretation of these results requires care. This is because, though we took good measures to blend each team with skillful and experienced subjects, the experiment actually did not took care of such experiences and skills in term of the team. In this case, the level of skill and experience of each team differs and may affect the maintenance task in terms of efficiency and comprehension. Factor that could also affects the results are the system's structural properties such as coupling, cohesion and inheritance. Though, inheritances were not utilized in the subject's programs, it is true that a good design involves having low coupling and high cohesion in a system in order for maintenance to be effective. Unfortunately, the reverse: high coupling and low cohesion is known to have negative effect on change propagation across systems. Consequently, much time could be spent by each team in order to understand and carry out changes correctly. Moreover, while some errors still remained in most of the team's program after changes were made could be as a result of either undiscovered indirect impacts resulting from the system's structural properties or the programming experience of the subjects.

VII. Validity Threats

Experiments are always associated with potential risks that can affect the validity of results. In this section, we discuss the important possible threats to the validity of the controlled experiment and what has been done to reduce them.

A. Internal Validity

Internal validity threats are effects that can affect the independent variable (TaskPhase) with respect to causality, without the knowledge of the researcher's in an experiment [18]. They pose threats to the conclusion about a possible causal relationship between treatment and outcome. In this study, the experiment was performed in two phases and in the same location and setting. Thus, lack of randomization of the TaskPhase assignment could result in skill differences between the participating teams, which in turn would render the results biased. However, this potential threat was addressed by assigning each subject to a team based on their previous performances to ensure that each team was balanced. In addition, since the same participants were involved in both phases, the dependent t-test proved most suitable for testing the stated hypotheses.

B. Construct Validity

Construct validity deals with the degree to which conclusions are justified from the perspective of the observed participants, study settings, and dependent and independent variables. These threats are as follows:

1) *Measusing PC, CD and NoE:* In the experiment, three simple measures were used as dependent variables: PC, CD and NoE. The variable PC, a measure of the program correctness, was a mark given which shows whether the subjects obtained a correct solution after change tasks MTask1 – 4 were carried out. To show the quality of the marks given, an independent expert was consulted. The programs were thoroughly tested and the program code was also inspected. This was to ensure that the program measure was appropriate. The CD measured the time spent to perform maintenance tasks correctly for the modification tasks MTask1 – 4. Though time was measured as a difference between the finish time and start time, we believe it might be affected by factors such as calling the attention of the supervisor and so on, during the experiment. However, we took every step to reduce this threat. Also, NoE is a count of the number of faults found on the IDE after implementing the changes for modification tasks MTask1 – 4. During compilation, necessary steps were taken to count the actual faults that originated. In addition, though PC, CD and NoE are the important pointers of program maintainability that reflect maintenance cost, however, several other maintainability dimensions were not covered such as faults severity, the design quality of the program and so on. To eliminate these threats, only quality programs were selected for the experiment.
2) *Task phases:* The division of the experiment into phases; *modification_without_IR* and *modification_with_IR* could be another important threat to the construct validity in the experiment. In this case, the trend was to determine whether the variable TaskPhase has satisfactory construct validity. In the context of the experiment, to check the construct validity we quantified beforehand the difficulty of modification tasks in terms of amount of class each program had and their complexity and the time needed to implement the changes.

C. External Validity

The threats to external validity concern conditions that limit generalization of the results obtained in the experiment [18][19]. Such threats are mainly from the participants, the settings and the nature of the system maintained.

1) *Application and tasks:* The systems used for the experiment were very small in size, maximum of two packages, 6 classes which are not up to a thousand lines of code (KLOC). Thus they were small-sized applications compared with industrial OO program systems. In addition, the modification tasks were relatively simple, small in size and time. However, program characterized in this manner poses limitation to controlled experiments and is dependent on the research question being asked as well as to the extent to which the results are supported by theory [20][21]. In the experiment, we showed a clear impact of TaskPhase, notwithstanding the small size of the applications and modification tasks. Its generalization to larger applications and tasks can be made with the support of existing program comprehension research theories. Additionally, it is possible that the task phases and their effects on project team's performance would be different for larger systems and complex maintenance tasks since larger systems will often require larger cognitive complexity. Also, if the experiment had lasted longer the results may have been different.
2) *Subject sample:* All the participants used in the experiment were only undergraduate students of computer science and thus fell in the class of "novices" or as "advanced beginners" as stipulated by [20]. Similar results might also be obtained by subjects having a similar background. Due to the small sample size of about 45 students in nine teams involved, caution is needed when interpreting the results. Also participants varied because of their individual programming skills and experience. However, due to the blending of the teams with skillful and experienced subjects, it is believed the presence of differences had no significant impact on the results obtained.

VIII. Conclusion

Change impact analysis plays an important role in the reduction of the risks and costs associated with unidentified effects of changes during software maintenance. In this paper, we have proposed a novel framework for carrying out CIA in OO software systems during software maintenance. The framework combines change impact prediction as well as faults or failure predictions on different system sizes. In addition, we proposed a method to represent OO program that allows both CIA and a quantification of their structural complexities. The approach will assist engineers in the facilitation of both program comprehension and onward software maintenance. The intermediate program representation constructed is quite simple, easy and do not analyze deeply into the methods' body. It explicitly reveals the complex dependencies in the program. As a benefit, it can be used to teach undergraduate student to understand the structure of OO software and perform CIA effectively during maintenance tasks. Furthermore, quantifying the structural complexity of the system especially for small or medium-size systems is important and can serve as substitute to OO design metrics. To assess the significance of the representation, an empirical evaluation of the approach was conducted and the results obtained were significant for CIA in terms of maintenance effort reduction of effort and costs. We therefore conclude that the framework and the intermediate representation are effective and practicable for impact analysis of OO software systems.

The limitation of the study is that, small sized systems were used in the evaluation of the IR. In addition, the participants involved were students and are not as skillful as professionals. We believe this could impact the results reported in this paper. However, necessary measures as discussed in the study validity threats were taken to ensure quality in the experiments and the results presented are valid. Our future work will be the implementation of the approach in order to automate the CIA process.

References

[1] Xu, J., Ho, D. and Capretz, L.F. An Empirical Validation of Object-Oriented Design Metrics for Fault Prediction. Journal of Computer Science No.4, Vol 7, pp. 571-577, 2008. ISSN 1549-3636

[2] Subramanyam, R. and Krishnan, M.S.: Empirical Analysis of CK Metrics for Object- Oriented Design Complexity: Implications for Software Defects. IEEE Trans. Software Eng. No.29, pp. 297-310, 2003

[3] Janes, A. et al. Identification of defect-prone classes in telecommunication software systems using design metrics. International Journal of Information Sciences, 2006

[4] Al-Dallal, J. and Briand, L.C.: An object-oriented high-level design-based class cohesion metric. Information & Software Technology No. 52, pp.1346-1361, 2010

[5] Lee, M., Offutt, A.J. and. Alexander, R. T. "Algorithmic analysis of the impacts of changes to object-oriented software. *Proceedings of the Technology of Object-Oriented Languages and Systems* (TOOLS 00). Washington, DC, USA: IEEE Computer Society, pp. 61-70, 2000

[6] Jönsson, P. and Lindvall, M.: "Impact Analysis" *Engineering and Managing Software Requirements* Issue: 6, Springer-Verlag, pp. 117-142, 2005

[7] Bohner, S. A.: "Extending software change impact analysis into COTS components*" Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop*, Greenbelt, USA, pp.175 -182. (2002)

[8] Arnold, R.S., and Bohner, S.A.: "Impact analysis – towards a framework for comparison", The Intl Conf. on Software Maintenance, 1993

[9] Abdi, M. K., Lounis, H. and Sahraoui, H. "Analyzing change impact in object-oriented systems" *Proceedinds of 32nd Euromicro Conference on Software Engineering and Advanced*, 2006, pp.8

[10] Rathore, S.S. and Gupta, A. Validating the Effectiveness of Object-Oriented Metrics over Multiple Releases for Predicting FP. Proceedings of 19th Asia-Pacific Software Engineering Conference, IEEE. pp.350-355, 2012

[11] Zhou, Y., Xu, B. and Leung, H. On the ability of complexity metrics to predict fault-prone classes in object-oriented systems. The Journal of Systems and Software No. 83, pp. 660–674, 2010

[12] Isong, B.E. and Ekabua, O.O. "A Systematic Review of the Empirical Validation of Object-oriented Metrics towards Fault-proneness Prediction", *International Journal of Software Engineering and Knowledge Engineering* (IJSEKE) World Scientific Publishing Company December, 2013. Vol. 23, No.10, pp. 1513–1540

[13] Pan, W.F., Li B, Ma Y.T. et al: Measuring structural quality of object-oriented software via bug propagation analysis on weighted software networks. Journal of Computer Science and Technology, 25(6): 1202–1213 Nov. 2010.

[14] Sun, X., Li, B., Tao, C., Wen, W. and Zhang, S. "Change Impact Analysis Based on a Taxonomy of Change Types" 2010 IEEE Proceedings of 34th Annual Computer Software and Applications Conference (COMPSAC 2010), 2010. pp.373-82

[15] Oliveira, M. et al: "The Hybrid Technique for Object-Oriented Software Change Impact, Analysis" Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR 2010), IEEE Press, 2010, pp.252-255

[16] Pan, W.F., Li B, Ma Y.T. et al: Measuring structural quality of object-oriented software via bug propagation analysis on weighted software networks. Journal of Computer Science and Technology, 25(6): 1202–1213 Nov. 2010.

[17] Liu, J., Lu, J., He, K. and Li, B.: Characterizing the structural quality of general Complex software networks. International Journal of Bifurcation and Chaos, Vol. 18, No. 2 (2008) 605–613

[18] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. Experimentation in Software Engineering: An Introduction. Norwell, MA, USA: Kluwer Academic Publishers, 2000

[19] Kitchenham, B.A., Pfleeger, S.L., Pickard, L.M., Jones, P.W., Hoaglin, D.C., El Emam, K., Rosenberg, J. "Preliminary guidelines for empirical research in software engineering". IEEE Transactions on Software Engineering, vol. 28, no. 8, pp. 721–734, 2002

[20] Mayrhauser, A.V. and Vans, A.M. Program comprehension during software maintenance and evolution. *Computer* vol. 28, no. 8, pp.44–55, 1995

Authors' Profiles

Isong, Bassey received B.Sc. degree in Computer Science from the University of Calabar, Nigeria in 2004 and M.Sc. degrees in Computer Science and Software Engineering from Blekinge Institute of Technology, Sweden in 2008 and 2010 respectively. He also received a Ph.D in Computer Science in the North-West University, Mafikeng campus, South Africa in 2014. In addition, since 2010, he has being a faculty member of the University of Venda, South Africa and a lecturer of Computer

Science and Information Systems. His research interests include Software Engineering, Requirements Engineering, Software Measurement and Maintenance, Information Security, Software Testing, and Mobile Computing.

Obeten O. Ekabua is a Professor and Departmental Chair of the Department of Computer Science in the North West University, Mafikeng Campus, South Africa. He holds BSc (Hons), MSc and PhD degrees in Computer Science in 1995, 2003, and 2009 respectively. He started his lecturing career in 1998 at the University of Calabar, Nigeria. He is the former chair of the Department of Computer Science and Information Systems, University of Venda, South Africa. He has published several works in several International and National journals, and also in several career conferences. He has also pioneered several new research directions and made a number of landmarks contributions in his field and profession. He has received several awards to his credit. His research interest is in software measurement and maintenance, Cloud and GRID computing, Cognitive Radio Networks, Security Issues and Next Generation Networks.

How to cite this paper: Bassey Isong, Obeten Ekabua,"A Framework for Effective Object-Oriented Software Change Impact Analysis", International Journal of Information Technology and Computer Science(IJITCS), vol.7, no.4, pp.28-41, 2015. DOI: 10.5815/ijitcs.2015.04.03