ACSIJ Advances in Computer Science: an International Journal, Vol. 3, Issue 4, No.10 , July 2014
ISSN : 2322-5157
www.ACSIJ.org

ACSIJ
WWW.ACSIJ.ORG

Explicit Object-Oriented Program Representation for Effective Software Maintenance

Bassey Isong
Department of Computer Science, University of Venda
Thohoyandou, Limpopo, South Africa
bassey.isong@univen.ac.za

Abstract

Today, object-oriented (OO) technology is a *de facto* approach in software development and several OO software applications are presently in use. For these systems to remain useful, they have to be effectively and efficiently maintained. As changes are both important and risky, Change impact analysis (CIA) is used to preserve the quality of the software system. OO software have complex dependencies and change types that often affect their maintenance in terms of ripple-effects identification or may likely introduce some faults which are hard to detect. Existing CIA proffers little or no clear information to represent the software for effective change impact prediction and components' fault-proneness is not considered. Consequently, changes made where dependencies and fault-proneness are not understood may have some undesirable effects elsewhere in the system or may increase its risks to fail. Therefore, this paper proposes an approach called OO Component Dependency Networks (OOComDN) which explicitly represent the software and allow its structural complexity to be quantified using complex networks. The objective is to enhance static CIA and facilitate program comprehension. To assess the effectiveness, a controlled experiment was conducted using students' project with respect to maintenance duration and correctness. The results obtained were significant, indicating OOComDN is practicable for impact analysis.

Keywords: *Impact Analysis, Software Change, Complex Networks, Faults.*

1. Introduction

Changes are an indispensable property of software which plays a crucial role in their evolution. Software during development or its life-time are subject to changes in order to continue to remain useful and meets its operational requirements. Factors that motivate software change on existing systems include activities such as defects fixing, new features addition to meet customers' changing requirements, environmental adaptation or internal code quality enhancement [1]. Despite the benefits associated with these activities, changes have possible high risks. Regardless 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 [2][3]. In particular, changes that are carried out frequently can destroy the architectures of the software and even increase source code and architecture inconsistency. In real-life software maintenance, this situation manifest especially, when the program dependencies and components' fault-proneness are ignored. This goes with the fact that making changes to software components while neglecting their dependencies and fault-proneness may have some unexpected effects on the quality of the later which may increase their risks to fail [2][3]. The emanating risk during the change is a function of the impact resulting from a given change made.

As today software applications have grown more and more in size and complexity, making these changes has become recognized as a challenging task. To be successful in performing changes, a good comprehension of the component dependencies as well as their fault-proneness probability is vital to avoid unintended effects in the system [4]. In the sphere of software development today, OO approach is overwhelmingly gaining momentum and widespread use. OO approach has the benefits of producing a clean, well-understood design characterized by easier to understand, test, maintain and extend [5]. Thus, it is important that OO software systems have to be effectively and efficiently maintained if they are to remain useful. However, the acclaim benefits of the OO technology do not on their own ensure quality, guard against developer's mistakes and prevent faults or failures. OO approach introduces new concepts whose features often affect its maintenance or increases its chances of becoming faulty.

Software change impact analysis (CIA) is the technique that is used as leverage. CIA tries to identify or estimate the consequences of the proposed change impact from the analysis of software product [4]. It is used to curb the risks and costs associated with unidentified effects of changes. Nevertheless, the complex relationships of OO features frequently create difficult situation that adversely impedes engineers' ability to anticipate and detect changes' ripple-effects or potential faults in the system [6]. The ripple-effects of a change or errors in one part of the system may

spread to other unchanged parts via the various complex dependencies. As a result, the maintainer would spend huge amount of time and efforts trying to locate the source of the failing effect. Several CIA approaches exist in the literature today such as static [7][8][9], dynamic [3][10][11] or hybrid approaches [19] as well as fault prediction models to predict the fault-proneness of high risks components, especially for large software systems. But these CIA approaches provides little or no information on how to explicitly represent OO software for effective comprehension and CIA. Additionally, faulty software components are not taken into consideration during the changes analysis. In this case, change impact and faults predictions are disjointed activities during software maintenance which make OO software modification more complex.

In the light of this research gap, this paper therefore, proposes and constructs an approach that will assist software maintainers in performing the OO software maintenance effectively. The approach will be effective in the reduction of maintenance efforts and cost in terms of change impact and faults prediction. To this end, we propose the use of complex networks to build an intermediate representation (IR) of an entire OO program which will explicitly reveals its implicit dependencies and allow for quantitative measurement of the software quality. The objective is to improve static CIA approach. By effectively representing OO program using the IR called OO component dependency networks (OOComDN), it would go a long way to facilitating program comprehension and CIA while preserving the quality of the software with less cost in terms of time and effort. The approach was evaluated using students' project in terms of maintenance duration and correctness and the results obtained were promising, indicating the IR is efficient for CIA.

The rest of this paper is organized as follows: Section 2 gives the background information, Section 3 discusses the IR of the OO program and Section 4 is proposed OOComDN. Section 5 is the empirical evaluation of the IR, Section 6 is the study discussion, Section 7 is the validity threat and Section 8 is the conclusion.

2. Background Information

Change to a software system is inevitable and software maintainers would be making changes in the dark if they don't understand what, how and where of the changes to be performed. CIA is an important technique that is used to determine the consequences of such software changes and preserve its quality. As an important property of software, changes are necessary either in the requirements, design or source codes. Several CIA approaches have been proposed, developed and used in the literature [7]. Among these approaches is the static CIA approach which constructs a static representation of the software and reveals the structural dependencies from the source code [5].

OO software systems are composed of separate but linking entities known as components. They include the fields, methods/functions, and classes which are usually the components that are used for analysis. With the inevitability of change, when a change is made on one component, it may propagate to other components not changed. The task of CIA technique in this case, is to find the initial change component and other components that are thought to be truly affected by the change. OO program on the other hand, have complex dependencies that often make it cumbersome to identify the impact of the change or faults during their maintenance [5].The drivers of this complexity are the OO features such as encapsulation, inheritance, polymorphism and dynamic binding which distinguishes it from structured-oriented paradigm [5]. (See Figure 1) Thus, a change in one component will inevitably cause undesirable effects in other components in a manner not anticipated.

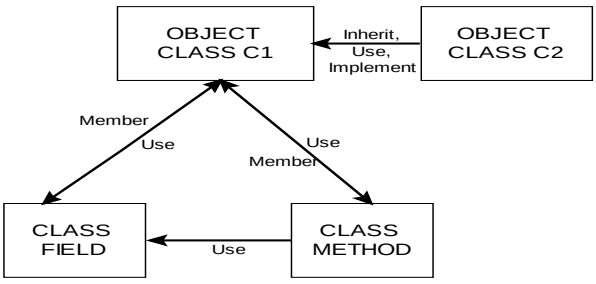

Fig. 1 OO program component dependencies

Existing static CIA approaches in the literature proffer little or no information on how to explicitly represent OO software for effective program comprehension and CIA process. This is because failing to understand all impacts of a change through component dependencies before the actual change implementation can have undesirable effects on the system. In addition, existing CIA approaches and software fault prediction activities are performed separately. No known approaches in this regard exist. Hence failing to take faults propagation into consideration during changes may results in same undesirable effect. The knowledge of component's fault-proneness probability is vital during CIA because, it is well-known that OO classes are not faults or failures free [6][12][13][14][15][16]. A fault is a defect in a software system that may cause an executable product to fail. Therefore, 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 lead to software failure. The early identification of the high-risk components will allow mitigating actions to be employed before change can be implemented. Understanding the different dependencies and source code change types of the OO software system as well as the probability of class fault-proneness will go a long way to reducing the effect of costly software failure.

As one of the benefits of this study, the teaching and learning of Software Engineering at the undergraduate level have long been centered on coding. This is evident in several students' course projects both published and unpublished. Software maintenance is one area where much attention has not been paid as students are not taught how to maintain a system. Today OO approach is becoming the mainstream in software development and several OO software applications exist and in use. Undergraduate students therefore, must be taught how to maintain a system alongside its coding. This will better equip the students with the necessary core competencies and technical skills expected of every software engineer when they graduate. They are the future software developers who may be involved in the maintenance of OO software systems. Hence, a simple approach that will assist them to understand OO program and carryout maintenance tasks successfully is indispensable. That is, an approach that will effectively represent OO program by exposing their structural characteristics which can aid comprehension and facilitate CIA. In this paper, we propose the use of IR of OO software using the complex networks. Details of the proposed approach are given in subsequent sections.

3. 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 work by [7] and [17]. In this paper, we used the idea of complex networks to model OO software system's structure.

3.1 Complex Networks in Software Systems

Complex networks in recent decades have gained increasing momentum and software system is not an exception due to its topological structure [17][18]. 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 is that today software systems especially OO program has exponentially grown in size and complexity with structure becoming more and more complicated such that a change or fault in one component often requires 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. In this case, the better the structure of the software, the lesser would the cost of the development be. Therefore, analyzing OO software system's structure using complex network will help the maintainer to achieve the following objectives:

A. 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.

B. 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. By analyzing the software structure quantitatively would help the maintainer to know in advance, the quality of the system and the risk posed by the propagation of faults from one component to the other. This is vital to allow a maintainer to take mitigating actions where necessary in order to reduce the cost of software failure when changes are implemented.

4. OO Component Dependency Networks

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

4.1 Change Diffusion Networks

In 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 when taking both the semantics and syntactic structure into consideration. CDN is used to represent the software components and their relationships for onward maintenance task, perhaps, CIA. It explicitly represents the structure of the OO program source code that will assists the software maintainer in quantifying which components will be truly affected by a change. In other words, the representation is basically used to discover the evolution mechanism of the OO software system.

4.1.1 Dependencies Types

In this study, we identified four types of dependencies, $\mathbf{D}^{\mathbf{Type}}$ that exist in OO program: *inheritance (H), usage (U), invocation (V),* and *membership (M)* [2][7]. They are determining factor 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 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, 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 $\mathbf{D}^{\mathbf{Type}}$ the following definition of OOComDN is considered: *OOComDN-1*.

Definition 1: *[OOComDN -1]*
Given an OO program, P let G = $\langle (N, D^E), D^{Type} \rangle$ *represent OOComDN given by:*

OOComDN-1 = $\langle \mathbf{(N, D^E), D^{Type}} \rangle$

Where $N = NP_k + N_C + N_M + N_F$ are the nodes and $D^E = N \times N \times \mathbf{D}^{\mathbf{Type}}$ represents the set of various edges with dependencies types, $\mathbf{D}^{\mathbf{Type}}$. $\mathbf{D}^{\mathbf{Type}}$ is called the weight of the graph and NP_k, 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.

4.1.2 Typical Illustration

A typical illustration of the OOComDN is shown in Figure 3 using the program, P written in Java of Figure 2. The various shapes used to represent each component in the OOComDN-1 are also shown in Figure 3.

```
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. 2 Sample program

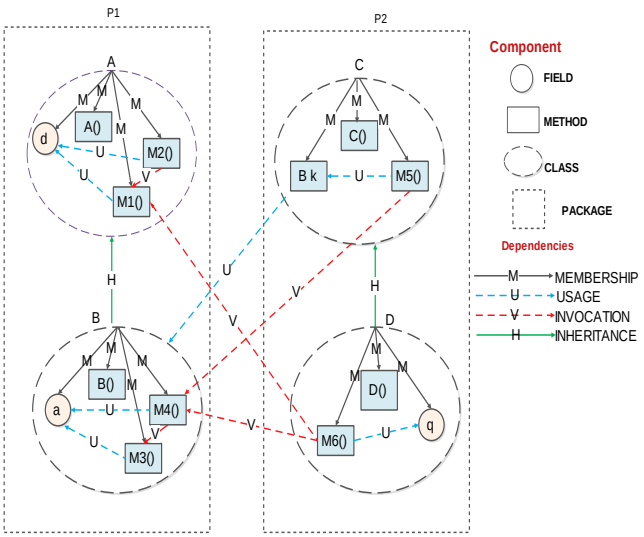

Fig. 3 OOComDN of the sample program in Figure 2

Figure 3 shows the representation of the OO program captured in Figure 2. 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, there will be an edge emanating 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* to be performed on a given component. The weight of each directed edge will indicate the probability that a change in one component say A, may or may not impact other component, D.

4.2 Degree of OOComDN-1

After the construction of the OO program as OOComDN-1, the first thing is to compute its degree, Z. Z of a node in OOComDN-1 is the number of dependencies a component has against other components connected to it or it is connected to. Two types of Z exist: *in-degree* and the *out-degree*. The computation of the 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. (see Figure 4) This is important as it gives an insight into how components are related to one another in terms of coupling and what need to be done to accomplish a change, when a change is consider in one component. Degree computation is done at the class level which is done after pruning OOComDN-1 leaving only classes and their dependencies types as shown in Figure 4. The definitions for Z are stated as follows:

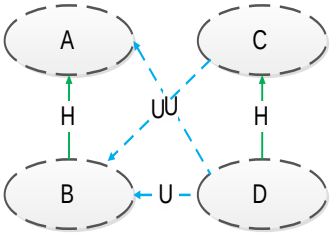

Fig. 4 Class level OOComDN-1

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

Given, OOComDN-1, < **(N, D^E), D^{Type}>***, with an adjacency matrix* $\mathbf{A_{ij}}$*, 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* $\mathbf{A_{ji}}$*.*

$$Z_i^{out} = \sum_j A_{ji} \quad \text{.....1}$$

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* $\mathbf{A_{ij}}$*.*

$$Z_i^{in} = \sum_j A_{ij} \quad \text{.....2}$$

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

$$Z_i^{tot} = Z_i^{in} + Z_i^{out} \quad \text{.....3}$$

In other words, $Z^{in}(n_i)$ indicates the number of classes that has dependency on class $n_j \in N$ and $\mathbf{Z}^{out}(n_i)$ the number of classes on which class $n_i \in N$ depends on. The in-degree and out-degree for the program shown in Figure 2 is captured in Table 1.

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

Node, n_i	Z_i^{in}	Z_i^{out}	Z_i^{tot}
A	(B,A) = 1, (D,A) = 1	-	2
B	(C,B) = 1, (D,B) = 1	(B,A) = 1	3
C	(D,C) = 1	(C,B) = 1	2
D	(D,A) = 1, (D,B) = 1 (D,C) = 1	-	3

As shown in Table 1, for instance, class A has one in-degree for the ordered paired (B,A) and (D,A) and no out-degree. In addition, Z^{tot} is a measure of the overall complexity of the program. The presentation clearly shows the nature of coupling in A which will assist a maintainer to know in advance, the complexity of the class before performing CIA. As 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 [12][16], the degree of a class, Z in a software network would actually shows the degree to which each class depends on other classes. In this paper, Z is used to measure the degree of coupling in a small or medium sized system.

4.3 Fault Diffusion Networks

Fault diffusion network (FDN) is similar to the one proposed by [17] and is represented just as CDN. As used in the OOComDN, 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. 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. 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:

$$\text{OOComDN-2} = <N_C, D_C, P_b>$$

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 5).

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 $\boldsymbol{P_b}$ **(i,j)**. In this paper, we defined it as follows:*

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

Where **CM(i,j)** is the set of members in class *j* which faults will propagate to the members in class *i*, that 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()*}

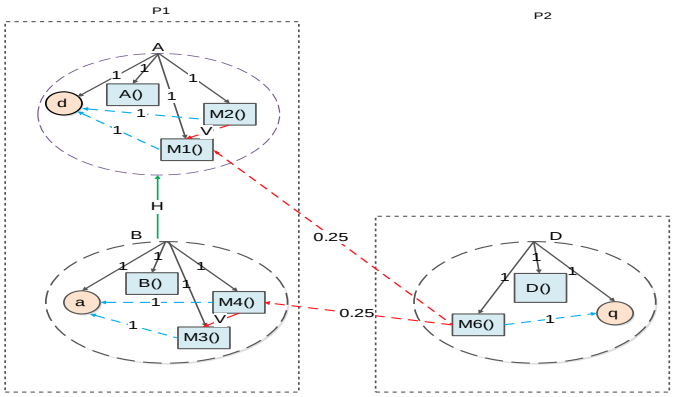

Fig. 5 Class fault propagation probability

As shown above, Figure 5 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 Figure 5, it is clear that class **D** depends on class **A** and **B** as follows:

$$(D.M6(),A) = \{M1()\} = D.M6() \rightarrow A.M1()$$

$$(D.M6(),B) = \{M4()\} = D.M6() \rightarrow B.M4()$$

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 denotes 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. With this computation, the higher the probability, the higher the risk of the fault propagation will be. In this case, 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. Therefore, it is important that during CIA, the risks propagation probability of all the affected classes identified as impact set should be

measured 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 the course of maintenance. It would in turn allow mitigating actions to be focused on those high risk components in time to avoid the cost of software failure.

5. 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 the OOComDN-1 was evaluated. Details are discussed in subsequent sections.

5.1 Study Setting, Subject, 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 third 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, strict measures were taken to blend the teams 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 successfully. 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 [3]. An overview of the experiment design is captured in Figure 6.

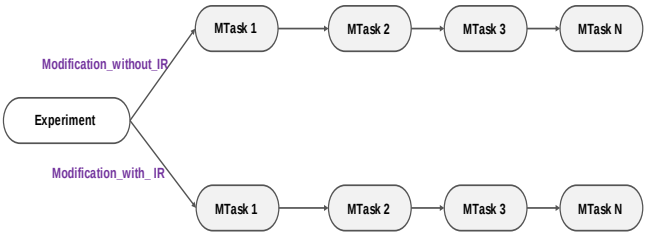

Fig. 6 Experimental design overview

5.2 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 Figure 6).

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.

5.3 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.

5.4 Statistical Technique and Specification

In this study, we used the paired-sample T-test called the dependent T-test statistical technique to test the hypotheses stated in Section 5.3. 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. 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 2. 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 2: 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 $H0_3$

5.5 Result Analysis

The main results obtained based on the task phases: *modification without IR* and *modification with IR* for MTask1 – Mtask4 are visualized in Figure 7 and Figure 8 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 the results, there are some clear indications that TaskPhase affect the CD, PC and NoE in the two phases. For instance, 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.

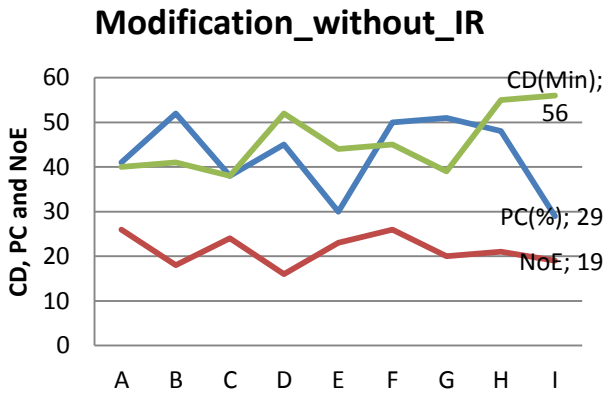

Fig. 7 Effect of TaskPhase on modification without IR

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 3. 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 was $p \leq 0.05$.

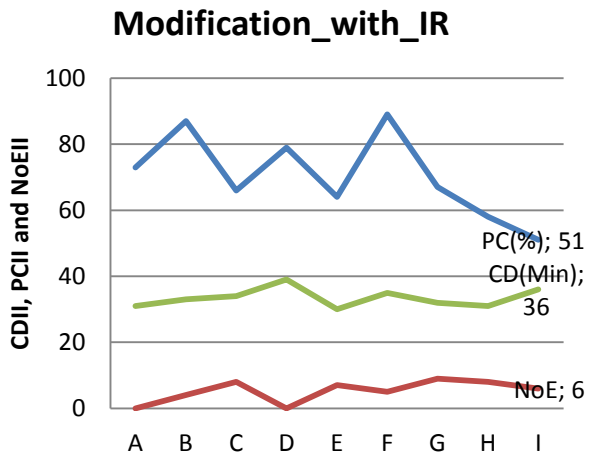

Fig. 8 Effects of TaskPhase on modification with IR

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

(1) For the impact of TaskPhase on CD, we rejected $H0_1$ since p-value $\approx 0.00 \leq 0.05$.

(2) For the impact of NoE introduced, we rejected $H0_2$ since p-value $\approx 0.00 \leq 0.05$, and lastly,

(3) For the impact of TaskPhase on PC, we rejected $H0_3$ since p-value $\approx 0.00 \leq 0.05$.

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.

Table 3: 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

6. Discussions

The results obtained from the experiment seem very interesting in terms of duration, program correctness and the number errors introduce after change were implemented for phase II. As shown in Figure 7 and 8 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 time 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.

7. Threats to Validity

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.

7.1 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 [20]. They pose a threat 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.

7.2 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:

7.2.1 Measuring Program Correctness, Change Duration and Number of Faults

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.

7.2.2 Task Phase

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.

7.3 External Validity

The threats to external validity concern conditions that limit generalization of the results obtained in the experiment [20][21]. Such threats are mainly from the participants, the settings and the nature of the system maintained.

7.3.1 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 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.

7.3.2 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 [22]. 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.

8. Conclusions

In this paper, we have proposed an approach to represent OO program. The approach will assist in the facilitation of both program comprehension and onward software maintenance, CIA in particular. The OOComDN

constructed is quite simple, easy and do not analyze deeply into the methods' body. It clearly reveals the complex dependencies in the program. Unlike other dependency graphs, OOComDN is not complex and the components involved are countable. As a benefit, OOComDN can be used to teach beginners such as undergraduate student to understand the structure of OO software and perform CIA effectively during maintenance tasks. In addition, it can be used to quantify the structural complexity of the system especially for small or medium-size systems without having to use OO design metrics. To assess the significance of the IR, 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. Therefore, this paper concludes that the IR is 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 therefore, believe it will affect the results reported in this paper. However, necessary measures as discussed in the study validity threats were taken ensure quality in the experiments and the results presented are valid. The future work of this study will be on the implementation of the approach in order to automate the approach.

References

[1] Bohner, S. A. and Amold, R. S., "Software Change Impact Analysis," IEEE Computer Society Tutorial, IEEE Computer Society Press,1996

[2] 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

[3] Law, J., Rothermel, G., "Whole program path-based dynamic impact analysis", The Intl Conf. on Software Engineering, 2003.

[4] Bohner, S. A., "A Graph Traceability Approach to Software Change Impact Analysis," Ph.D. Dissertation George Mason University, Fairfax, VA, 1995

[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] Fenton, N., Ohlsson, N. Quantitative analysis of faults and failures in a complex software system. IEEE Transactions on Software Engineering, Vol. 26 no. 8, pp.797-814, 2000

[7] 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

[8] Badri, L. Badri, M and Yves, S. D. Supporting predictive change impact analysis: a control call graph based technique. In Proceedings of Asia-Pacific Software Engineering Conference, 2005

[9] Zhang, S., Gu, Z., Lin, Y. and Zhao, J. J. Change impact analysis for AspectJ programs. In Proceedings of International Conference on Software Maintenance, pages 87 – 96, 2008

[10] Law, J. and Rothermel, G. Incremental dynamic impact analysis for evolving software systems. In Proceedings of International Symposium on Software Reliability Engineering, 2003

[11] Apiwattanapong, T., Orso, A. and Harrold, M. J. Efficient and precise dynamic impact analysis using execute after sequences. In Proceedings of International Conference on Software Engineering, pages 432 – 441, 2005.

[12] Emam, K.E., Melo, W.L., Machado, J.C.: "The prediction of faulty classes using object-oriented design metrics". Journal of Systems and Software, No.56, pp. 63-75, 2001

[13] Malhotra, R., Kaur, A. and Singh, Y. Empirical validation of object-oriented metrics for predicting FP at different severity levels using support vector machines. International Journal System Assurrance Engineering Management. No.1, vol. 3, pp. 269–281, 2010.

[14] 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

[15] 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

[16] 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

[17] 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.

[18] 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.

[19] 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

[20] 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

[21] 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

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

Bassey Isong received B.Sc degree in Computer Science from the University of Calabar, Nigeria in 2004 and M.Sc. degree in both Computer Science and Software Engineering from Blekinge Institute of Technology, Sweden in 2008 and 2010 respectively. Presently, he is a PhD candidate of Computer Science, North-West University, South Africa. Since 2010, he has being a faculty member in the University of Venda, South Africa and a lecturer of Computer Science and Information Systems. His research interests include Requirements Engineering, Software Evolution Information Security, Software Testing, and Mobile Computing.