Malaysian Journal of Computer Science, Vol. 14 No. 2, December 2001, pp. 106-115

CONCEPTS FOR SLICING OBJECT-ORIENTED PROGRAMS

Nor Adnan Yahaya
Telekom R & D Sdn. Bhd.
UPM-MTDC Technology Incubation Centre
43400 Serdang, Selangor
Malaysia
Tel: 603 - 89423566
email: n.yahaya@computer.org

Hamed J. Al-Fawareh
Computer Science Dept.
Zarka Private University
Zarka/Jordan
Tel: 05 - 3656100/ext. 1145
email: fawareh@hotmail.com
fawareh@zpu.edu.jo

Abdul Azim Abd. Ghani
Faculty of Computer Science & IT
Universiti Putra Malaysia
43400 Serdang, Selangor
Malaysia
Tel: 603 - 89486101
email: azim@fsktm.upm.edu.my

ABSTRACT

This paper proposes several concepts that form the basis for slicing object-oriented programs. In the case of object-oriented languages, new relations occur between language constructs such as classes, methods, and messages. Thus new forms of dependencies have to be considered in addition to the traditional control and data dependencies that form the basis of most software maintenance tools that are currently available for maintaining procedure-oriented programs. We show how the proposed slicing concepts can be applied within the software maintenance process by giving an illustration through an example of Java program.

Keywords: ***Software Maintenance, Object-Oriented Programs, Program Slicing, Program Understanding***

1.0 INTRODUCTION

The last decade of the twentieth century has seen a rapid increase in the use of object-oriented approaches to software development. This trend is expected to continue in this millennium in light of the continuing progress and utilisation of the Java-based technology, particularly in the area of distributed computing. Eventhough advocates of the object-oriented approach generally believe that it can help in improving the readability of programs, the basic maintenance tasks to be carried out on them are something that still cannot be avoided. In other words, during the maintenance phase, object-oriented programs still need to be understood and later modified for the purpose of performing corrective or adaptive maintenance, functional enhancement as well as efficiency improvement. Therefore, if thus far various software maintenance systems have been developed to help in maintaining software systems developed through the use of the traditionally popular procedure-oriented approaches, we would expect that similar situation is also applicable to the maintenance of object-oriented systems.

Analysing dependencies between software components is one of basic activities used by a software maintenance engineer to identify various relationships among program elements. In the case of procedure-oriented programs, control and data dependencies are normally sufficient for helping him understand and trace their behaviour [1-5]. Although the expected benefits that one can gain through object-oriented development can be high, maintaining object-oriented programs can be problematic if not done systematically. The salient features of object-oriented techniques such as polymorphism, inheritance, encapsulation and dynamic binding are the main reasons for many maintenance problems. These additional features create additional dependencies between program elements and thus make the problem of understanding object-oriented systems more tedious and can be more complex than the procedure-oriented counterparts.

In this paper, we adapt the concept of program slicing to capture dependencies that are useful for maintaining object-oriented programs. To support this, we propose several new concepts that form the basis for slicing object-oriented programs in general. This extends the original notion of program slice [6, 7] to cover several types of program fragments that are believed to be useful in carrying out the maintenance of object-oriented programs. We show how the proposed slicing concepts can be applied within the software maintenance process by giving an illustration through an example of Java program.

2.0 DEPENDENCIES IN OBJECT-ORIENTED PROGRAMS

Object-oriented techniques introduce several new features to be incorporated into the resulting object-oriented programs. Hence, as a result of applying these techniques in developing large programs, several new dependencies between program components are important to the program maintenance tasks. These dependencies involve additional program components such as classes, methods and messages, which can be very difficult to capture due to additional types of relationships such as inheritance, polymorphism and dynamic binding.

The following convention is used throughout this paper:

1. P denotes the program to be maintained.
2. $Ent(P)$ denotes the set { id | id is either the symbol "_", "*" , an identifier (name) of a class or method, or a labelling of any statement or variable in P }. The special symbols "_" and "*" denote any or all other members in $Ent(P)$ respectively, that can satisfy certain given properties.
3. $Var(A)$ denotes the set { v | v is a public, protected, or private variable declared in the fields of class A }. $A.v$ denotes the variable v Î $Var(A)$.
4. $Mtd(A)$ denotes the set {m| m is a method declared in the fields of class A }. $A.m$ denotes the method m Î $Mtd(A)$. $A.m.s$ and $A.m.v$ denote the statement s in m Î $Mtd(A)$ and variable v declared in m respectively.
5. $Var(P)$ denotes the set { $v \mid v \in Var(A)$ and A is a class in P }.
6. $Dep(P)$ denotes the set { d | d is a dependence relation that is defined for classes and their components in P }.
7. $use(x)$ denotes the set of variables which may be referenced when x is being executed.
8. $def(x)$ denotes the set of variables which may be modified when x is being executed.

3.0 BASIC RELATIONSHIPS

Various relationships between statements, variables and methods within and across classes are the cause of dependencies, which are of interest to the maintenance process. For our purpose, we have identified ten basic (direct) relationships which are defined as follows:

Execution sequence: Let A be a class, s and t be any two statements occurring in a method m of A. The execution flow relation, denoted by $A.m.s \xrightarrow{e} A.m.t$ holds if and only if t is executed immediately after s for some execution of s. This relationship is undefined for statements between two different methods.

Usage relationships: This type of relationship occurs between methods, statements and variables. Let A and B be two classes, mÎ $Mtd(A)$, s is a statement in m, n Î $Mtd(B)$ and v Î $Var(B)$. The usage relationships include the following:

1. Statement using method: $A.m.s \xrightarrow{u} B.n$ if and only if s contains a message that corresponds to n.
2. <u>Method using a method:</u> $A.m \xrightarrow{u} B.n$ if and only if there exists a statement s in m such that $A.m.s \xrightarrow{u} B.n$.
3. Statement using variable: $A.m.s \xrightarrow{u} B.v$ if and only if any of the following holds:
   i. $B.v$ Î $use(A.m.s)$
   ii. \$ m¢ Î $Mtd(B)$ such that $A.m.s \xrightarrow{u} B.m$¢ and $B.v$ Î $use(B.m$¢$.s$¢$)$ for some s¢ in m¢.
4. <u>Method using variable</u>: $A.m \xrightarrow{u} B.v$ if and only if there exists a statement s in m such that $A.m.s \xrightarrow{u} B.v$.

Affect relationships: Our main concern here is how a statement or method affects the state of other elements. In this case, we view the change in state as being the change in the value of any variables that are within the scope of the latter. Let A and B be two classes, $m Î Mtd(A)$, s is a statements in m, $n\ Î\ Mtd(B)$ and $v\ Î\ Var(B)$.

1. Statement affecting variable: $A.m.s \xrightarrow{a} B.v$ if and only if any of the following holds:
   - $i.$ $B.v\, Î\ def(A.m.s)$
   - $ii.$ $\$\ m¢\, Î\ Mtd(B)$ such that $A.m.s \xrightarrow{u} B.m¢$ and $B.v\, Î\ def(B.m¢.s¢)$ for some $s¢$ in $m¢$.
2. Method affecting variable: $A.m \xrightarrow{a} B.v$ if and only if there exists a statement s in m such that $A.m.s \xrightarrow{a} B.v$.

Class relationships: Three main relationships between classes need to be represented. The last two are merely extensions to the use and affect relationships to the class level. They are:

1. Inheritance: $A \xrightarrow{i} B$ if and only if A is a subclass (derived class) of B.
2. Class using class: $A \xrightarrow{u} B$ if and only if either one of the following holds:
   - $i.$ $\$\, m Î\ Mtd(A)\ and\ v\, Î\ Var(B)$ such that $A.m \xrightarrow{u} B.v$.
   - $ii.$ $\$\, m Î\ Mtd(A),\ n Î\ Mtd(B)$, such that $A.m \xrightarrow{u} B.n$.
3. Class affecting class: $A \xrightarrow{a} B$ if and only if $\$\ m Î\ Mtd(A)$ and $v Î\ Var(B)$ such that $A.m \xrightarrow{a} B.v$.

4.0 DEPENDENCE RELATIONSHIPS

In general, we say that a program element is dependent on the other if some maintenance tasks is being done to any of them, then the program's behaviour involving the other may be affected. The maintenance tasks may involve removal and modifications made to the element in question. Based on the basic relationships defined earlier, several types of dependencies that are intuitively important for understanding the structure and behaviour of object-oriented programs are formulated. These dependencies are defined recursively as follows:

Control dependence: Let s and t be any two statements in a method m defined in a class A. We say that t is control dependent on s, denoted by $A.m.s \overset{e}{\Rightarrow} A.m.t$ if and only if any of the following holds:

$i.$ $A.m.s \xrightarrow{e} A.m.t$

$ii.$ There exists a statement $s¢$ also in m such that $A.m.s \overset{e}{\Rightarrow} A.m.s¢$ and $A.m.s¢ \xrightarrow{e} A.m.t$.

Control dependence relation is undefined for statements between two different methods.

Usage dependence

1. Statement dependence on a method/variable: A statement s in a method m of a class A is said to be dependent on Y which is either a method n or a variable v in a class B, denoted by $A.m.s \overset{u}{\Rightarrow} Y$ if and only if any of the following holds:
   - $i.$ $A.m.s \xrightarrow{u} Y$
   - ii. $\$\ m¢ \in Mtd(A')$ such that $A.m.s \overset{u}{\Rightarrow} A¢.m¢$ and $A¢.m¢ \xrightarrow{u} Y$ for a class A' in P.
   - iii. $\$\ s¢$ in m and a variable $w \in Var(A)$ such that $A.m.s¢ \overset{e}{\Rightarrow} A.m.s$, $A.m.s¢ \xrightarrow{a} A.w$, $A.m.s \xrightarrow{u} A.w$, and $A.m.s¢ \overset{u}{\Rightarrow} Y$.
2. Method dependence on method/variable: Let A and B be two classes and Y be either method $B.n$ or variable $B.v$. $A.m \overset{u}{\Rightarrow} Y$, if and only if $\$\ s$ in m such that $A.m.s \overset{u}{\Rightarrow} Y$.

Affect dependence:

1. Variable dependence on statement: A variable v defined in a class B is said to be dependent on a statement s in a method m of a class A, denoted by $A.m.s \overset{a}{\Longrightarrow} B.v$ if and only if any of the following holds:
   i. $A.m.s \xrightarrow{a} B.v$
   ii. $\$\ m¢\ Î\ Mtd(A¢)$ such that $A.m.s \overset{u}{\Longrightarrow} A¢.m¢$ and $A¢.m¢ \xrightarrow{a} B.v$.
   iii. There exists a statement s' in m and variable $w\ Î\ Var(A)$ such that $A.m.s \overset{e}{\Longrightarrow} A.m.s¢$, $A.m.s \xrightarrow{a} A.w, A.m.s' \xrightarrow{u} A.w,$ and $A.m.s¢ \overset{a}{\Longrightarrow} B.v$.

2. Variable dependence on a method: A variable v defined in a class B is said to be dependent on a method m defined in a class A, denoted by $A.m \overset{a}{\Longrightarrow} B.v$ if and only if $\$\ s$ in m such that $A.m.s \overset{a}{\Longrightarrow} B.v$.

Class dependence: The following dependencies are defined:

1. Dependence through inheritance: $A \overset{i}{\Longrightarrow} B$ if and only if any of the following holds:
   i. $A \xrightarrow{i} B$
   ii. There exists a class $A¢$ such that $A \overset{i}{\Longrightarrow} A¢$ and $A¢ \xrightarrow{i} B$.
2. Dependence through usage: $A \overset{u}{\Longrightarrow} B$ if and only if any of the following holds:
   i. $\$\ m\ Î\ Mtd(A)$ and $\$\ n\ Î\ Mtd(B)$ such that $A.m \overset{u}{\Longrightarrow} B.n$.
   ii. $\$\ m\ Î\ Mtd(A)$ and $\$\ v\ Î\ Var(B)$ such that $A.m \overset{u}{\Longrightarrow} B.v$.
3. Dependence through causing side effects: $A \overset{a}{\Longrightarrow} B$ if and only if $\$\ m\ Î\ Mtd(A)$ and $v\ Î\ Var(B)$ such that $A.m \overset{a}{\Longrightarrow} B.v$.

For convenience in symbolic manipulation, all the basic and dependence relations defined earlier are taken to be irreflexive. Also for simplicity, any relation r, $X \xrightarrow{r} Y \xrightarrow{r} Z,\ X \overset{r}{\Longrightarrow} Y \overset{r}{\Longrightarrow} Z$ denotes $X \xrightarrow{r} Y,\ Y \xrightarrow{r} Z$ and $X \overset{r}{\Longrightarrow} Y,\ Y \overset{r}{\Longrightarrow} Z$ respectively.

5.0 TECHNIQUES FOR SLICING OBJECT-ORIENTED PROGRAMS

The original concept of program slicing was introduced by Weiser [6, 7]. It refers to a technique of capturing statements that are relevant to a particular computation, which is to be specified by a slicing criterion. The resulting program fragment is called a program slice. Ever since the introduction of this concept of a program slice, various slightly different notions of program slices have been proposed, together with methods of computing them. A comprehensive survey of program slicing techniques can be found in [8].

For our purpose, we choose to extend the traditional concept of program slice to cater for program fragments that are more amenable to the problem of maintaining object-oriented programs. Slicing object-oriented programs would then involve capturing the various combinations of dependencies between classes as well as their components that are considered important to software maintenance. This section discusses definitions of several concepts of slices that bear strong relationships with dependencies of interest.

Convention:

i. [*A*] refers to class *A* without any elements removed from it.
ii. [*A/V*] refers to class *A* with the set of variables *V* and removal of elements not involved in using/defining any variables in *V*.

iii. [*A.m*] refers to class *A* with only the method *m* and the relevant associated declarations in *A*.
iv. [*A.m/V*] refers to [*A.m*] with the set of variables *V* and removal of elements not involved in using/defining any variables in *V*.
v. [*A.m/S*] refers to [*A.m*] with the set of statements *S* and the relevant associated declarations in *A*.

Definition: An elementary slicing criterion is a 4-tuple $C=< d, E_1, E_2, V >$ where,
1. *d Î Dep(P),*
2. E_1, E_2 *Î Ent(P)* and the expression $E_1\ d\ E_2$ is valid.
3. *V Í Var(P).*

The slicing criterion *C* can be used to specify the dependencies of interest that may exist between entity E_1 and E_2 in *P*. A slice *S* with respect to a slicing criterion $C=<d, E, E_2, V>$ denoted by $Slice(< d, E_1, E_2, V >)$, is a program fragment of *P* that is obtained through several slicing rules.

Dependencies can also be categorised according to levels. The first category is *class-level* involving a class to another class. The second category is *method-level* involving a method or a statement within a method to another method or variable in a class. The final category is *statement-level* which is basically intra-method involving statements within a given method. Slices, in turn, can also be categorised according to such levels of dependencies intended to be captured.

The most general type of slices are the *class-level* slices, which capture *class-level* dependencies. This type of slices intuitively will exclude a lesser number of elements compared to the lower-level slices of similar form of criterion. The general form of slicing criterion for them is $C = < \overset{d}{\Longrightarrow}, A, B, V >$, where $d \in \{ i, u, a \}$.

Class-level slicing based on dependence through inheritance is specified according to the following rule:

Slicing Rule 1: If $C= < \overset{i}{\Longrightarrow}, A, B, V >$ then $Slice(C)$ is $S1 \cup S2 \cup S3$ where:
i. $S1=\{[A]\}$
ii. $S2=\{ [Y / (V \cap Var(Y))] \mid A \overset{i}{\Longrightarrow} Y \overset{i}{\Longrightarrow} B \}$
iii. $S3=\{ [B/ (V \cap Var(B))] \}$ if $A \overset{i}{\Longrightarrow} B$

Slice(C) obtained through rule 1 shows how any variable in *V* is being used by the class *A* through class *B*. Essentially, it will capture classes *A*, *B* and all the intermediate classes linking *A* to *B* if *A* and *B* are related through *inheritance*. However, if this is true, only the parts of *B* and intermediate classes that are related to variables in *V* are captured.

Class-level slicing based on dependence through usage or causing side effects is specified according to the following rule:

Slicing Rule 2: If $C= < \overset{d}{\Longrightarrow}, A, B, V >$ where $d \in \{u, a\}$ then $Slice(C)$ is $S1 \cup S2 \cup S3$ where:
i. $S1=\{[A]\}$
ii. $S2=\{ [Y / (V \cap Var(Y))] \mid A \overset{u}{\Longrightarrow} Y \overset{u}{\Longrightarrow} B\}$
iii. $S3=\{ [B / V'] \mid V' \subseteq V$ and $"v \in V', \$\ m \in Mtd(A)$ such that $A.m \overset{d}{\Longrightarrow} B.v \}$

Slice(C) obtained through rule 2 shows how any variable in *V* is being used or affected by the class *A* through class *B*. Essentially it will capture classes *A*, *B* and all the intermediate classes linking *A* to *B* if *A* and *B* are related through such dependencies. However, if this is true, only the parts of *B* and intermediate classes that are related to variables in *V* with respect to the dependence relation of interest are captured.

Method-level slicing gives a more refined view of the use and affect dependencies between classes. They can be obtained through four slicing rules. Before we will present these rules, we will present the only rule for performing statement-level slicing.

Definition: A *definition-use chain* from statements s to t in a given class method is an ordered set of statements $d_u(s, t)=(s, x_1, x_2,\ldots,x_n, t)$ where $s \overset{e}{\Rightarrow} x_1 \overset{e}{\Rightarrow} x_2 \overset{e}{\Rightarrow} \ldots x_n \overset{e}{\Rightarrow} t$ such that $def(s) \cap use(x_1)$ ^{1} $\emptyset$, $def(x_n) \cap use(t)$ ^{1} $\emptyset$, and for every I satisfying $(1<I\leq n)$, $def(x_{I-1}) \cap use(x_I)$ ^{1} $\emptyset$.

<u>**Slicing Rule 3**</u>: If $C= < \overset{e}{\Rightarrow}, A.m.s, A.m.t, V >$ then $Slice(C)$ is $[A.m /(S1 \cup S2 \cup S3)]$ where

i. $S1= \{x|\, A.m.s \overset{e}{\Rightarrow} x \overset{e}{\Rightarrow} A.m.t$ and \$ $v\in (V \cap (use(x) \cup def(x))) \}$.

ii. $S2= \{x|\, x\in d_u(y, z)$ where $A.m.s \overset{e}{\Rightarrow} y \overset{e}{\Rightarrow} z \overset{e}{\Rightarrow} A.m.t$ and \$ $x'\in \{ S1 \cap d_u(y, z) \}\}$.

iii $S3= \{A.m.s\}$ if \$ $x\in (S1 \cup S2)$ such that $def(A.m.s) \cap use(x)$ ^{1} $\emptyset$

iv $\{A.m.t\}$ if \$ $x\in (S1 \cup S2)$ such that $def(x) \cap use(A.m.t)$^{1} $\emptyset$.

Slicing rule 3 captures the statements between $A.m.s$ and $A.m.t$ inclusive, which involve in using or defining any of the variables in V either directly or indirectly through *def()-use()* chains.

Finally, the four slicing rules for performing method-level slicing are as follows:

<u>**Slicing Rule 4**</u>: If $C= < \overset{u}{\Rightarrow}, A.m, B.n, V >$ then $Slice(C)$ is $S1\cup S2\cup S3$ where:

i. $S1= \{[A.m]\}$

ii. $S2= \{ [Y.k] \mid A.m \overset{u}{\Rightarrow} Y.k \overset{u}{\Rightarrow} B.n$ and $B.n \xrightarrow{u} B.v$ for some $v \in V\}$

iii. $S3= \{[B.n/V']$ such that $V'\subseteq V$ and " $v\in V', A.m \overset{u}{\Rightarrow} B.n$ and $B.n \xrightarrow{u} B.v \}$

Slice(C) obtained through rule 4 shows how any variable in V is being used by method $A.m$ through method $B.n$. This slice captures all the intermediate classes, which could form more than one path leading to $B.n$, if $B.n$ references any of the variables in V. If this condition is untrue, the slice will contain only $[A.m]$ which implies that method m in class A is not dependent on method n in class B in order to reference any variable in V. However, if this condition is true, only a portion of $B.n$ will be of interest and captured. No other method in B will be captured.

Slicing Rule 5: If $C= < \overset{u}{\Rightarrow}, A.m.s, B.n, V >$ then $Slice(C)$ is $S1\cup S2\cup S3\cup S4$ where:

i. $S1= \{[A.m /\{A.m.s\}]\}$

ii. $S2= \{ [Y.k] \mid A.m.s \overset{u}{\Rightarrow} Y.k \overset{u}{\Rightarrow} B.n$ and $B.n \xrightarrow{u} B.v$ for some $v \in V \}$

iii. $S3= \{ [B.n] \mid V' \subseteq V$ and $\forall v \in V',\ A.m.s \overset{u}{\Rightarrow} B.n$ and $B.n \overset{u}{\Rightarrow} B.v \}$

iv. $S4= \bigcup_{i=1}^{k} Slice(<\overset{e}{\Rightarrow}, A.m.s_i, A.m.s, def(A.m.s_i)>)$ where $A.m.s_i \xrightarrow{u} y$ for some $y \in S2$.

Slice(C) obtained through rule 5 gives a more restricted view of the dependence of interest as specified by rule 4 above. Specifically, it seeks to show how any variable in V is being used by statement s in method m through method $B.n$. This rule requires the resulting slice to capture the intermediate classes leading to $B.n$ that are traceable to $A.m.s$ only, if $B.n$ references any of the variables in V. In addition, this type of slice, through $S4$ will capture only the relevant intermediate statements between $A.m.s$ and other statements logically above it, from which $A.m.s$ has obtained this dependency indirectly.

<u>**Slicing Rule 6**</u>: If $C= < \overset{d}{\Rightarrow}, A.m, B.v, \_ >$ where $d \in \{u, a\}$ then $Slice(C)$ is $S1\cup S2\cup S3$ where:

i. $S1= \{[A.m]\}$

ii. $S2=\{ [Y.k] \mid A.m \overset{u}{\Rightarrow} Y.k \overset{d}{\Rightarrow} B.v \}$

iii. $S3=\{ [B/\{v\}]\}$ if $A.m \overset{d}{\Rightarrow} B.v$

Slicing Rule 7: If $C = <\overset{d}{\Rightarrow}, A.m.s, B.v, \_>$ then $Slice(C)$ is $S1 \cup S2 \cup S3 \cup S4$ where:

i. $S1 = \{[A.m / \{A.m.s\}]\}$

ii. $S2 = \{ [Y.k] \mid A.m.s \overset{u}{\Rightarrow} Y.k \overset{d}{\Rightarrow} B.v \}$

iii. $S3 = \{ [B / \{v\}]\}$ if $A.m.s \overset{d}{\Rightarrow} B.v.$

iv. If d is u then

$$S4 = \bigcup_{i=1}^{k} Slice(<\overset{e}{\Rightarrow}, A.m.s_i, A.m.s, def(A.m.s_i)>) \text{ where } A.m.s_i \xrightarrow{u} y \text{ for some } y \in S2.$$

Else if d is a then

$$S4 = \bigcup_{i=1}^{k} Slice(<\overset{e}{\Rightarrow}, A.m.s, A.m.s_i, def(A.m.s)>) \text{ where } A.m.s_i \xrightarrow{u} y \text{ for some } y \in S2.$$

$Slice(C)$ obtained through rule 6 and 7 above further gives a more restricted view, in the sense that the main interest is how a particular variable $B.v$ is used or affected by statement $A.m$ and $A.m.s$ respectively. In this case, even if $A.m \overset{d}{\Rightarrow} B.v$ (for rule 6) and $A.m.s \overset{d}{\Rightarrow} B.v$ (for rule 7) is true, [$B/ \{v\}$] may not include any method in B if the dependency is through direct access on $B.v$ by a statement in a method in $S2$. In C++, $B.v$ must be either protected or public. On other hand, unlike rule 4 and 5, portions of more than one method in B may be captured. For rule 7, like rule 5, it will capture only the relevant intermediate statements between $A.m.s$ and other statements logically above or below it when the dependence relation d is u or a respectively, from which $A.m.s$ has obtained this dependency indirectly.

6.0 ILLUSTRATION OF USE

The concept proposed in this paper can be applied within software maintenance tasks particularly in understanding the dependencies among elements of object-oriented programs. We foresee the normal approach would be to formulate meaningful queries related to various aspects of interests. These queries are, in turn, transformed into various slicing criteria that will be used to obtain slices that can be analysed and utilised for maintenance purposes.

Consider the program in Appendix A as the input program that needs to be sliced.

Example 1: Slicing using criteria

C = $Slice\ (<\ \overset{e}{\Rightarrow}, Discount.increase.1, Discount.increase.9, \{rate\}>)$ will involve the following steps according to Rule 3:

1) $S1$ = {2, 2.1, 3, 3.1, 5}
2) The maximal *definition-use* chains between statements 1 and 9 involving any statement is S1 are {(2.1, 5, 6, 7), (3.1, 5, 6, 7)}
3) $S2$ = {2.1, 3.1, 5, 6, 7}
4) $S3$ = {1} since $def(1) \cap use(2)\ {}^{1}\ \emptyset$
5) $S4$ = {9} since $def(5) \cap use(9)\ {}^{1}\ \emptyset$

Therefore, based on the above computations, *Slice* $(< \overset{e}{\Rightarrow}$, *Discount.increase.1, Discount.increase.9, {rate}>)* includes the following:

```
1      status = MyInput.readString();
2      if(status == “student”)
1.1        rate = 20/100;
3      if(status == “pensioner”)
3.1        rate = 17/100;
5        node.Total=node.Total-node.Total*rate;
6        discount=Total - node.Total;
7        System.out.println(“discount :” +discount);
9        System.out.println(“Total After Discount:”  + node.Total);
```

Fig. 1: Result of Slicing

Example 2: Computing *Slice* $(< \overset{u}{\Rightarrow}$, *TestComputeArea.main, circle.findArea, {radius} >)* will involve the following steps according to Rule 4:

1) *S1={[TestComputeArea.main]}*
2) *S2={ }* since there are no intermediate classes through usage relationships.
3) *S3={[circle.findArea/{radius}]}* Since *TestComputeArea.main* $\xrightarrow{u}$ *circle.findArea* and *circle.findArea* $\xrightarrow{u}$ *Circle.radius.*

Slice$(< \overset{u}{\Rightarrow}$, *TestComputeArea.main, circle.findArea, {radius} >)* obtained based on the above steps is represented as the boldfaced portion of the original program in Appendix A.

7.0 CONCLUSION

This paper discusses concepts relevant for slicing object-oriented programs. In the case of object-oriented programs, data and control dependencies are no longer sufficient for supporting maintenance tasks. Class and component dependencies are more natural in supporting the process of understanding the structure and behaviour of object-oriented programs. Based on the identified dependence relations, several slicing rules have been developed for slicing object-oriented programs. We also highlight the potential application of this new slicing approach by providing an illustration with two examples. We leave the presentation of slices as a separate concern, although our present inclination is to present them as the highlighted parts of the original program.

REFERENCES

[1] D. Binkley and K. B. Gallagher, “Program Slicing”. *Advances in Computers*, Vol. 43, 1996, pp. 1-50.

[2] S. Horwitz , T. Reps, and D. Binkley, “Interprocedural Slicing Using Dependence Graphs”. *ACM Transactions on Programming Language and System*, Vol. 12, No. 1, January 1990, pp. 26-60.

[3] K. H. Soo and Y. R. Kwon, “Restructuring Programs through Program Slicing”. *International Journal of Software Engineering and Knowledge Engineering*, Vol. 4, No. 3, 1994, pp. 349-368.

[4] A. D. Lucia and A. R. Faslino, “Understanding Function Behaviours Through Program Slicing”, in *Workshop on Program Comprehension, IEEE Press*, 1996.

[5] M. Weiser, “Programmers Use Slices When Debugging”. *Communications of the ACM*, Vol. 25 No. 7, July 1982, pp. 446-452.

[6] M. Weiser, *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. PhD Thesis, University of Michigan, Ann Arbor, 1979.

[7] M. Weiser, "Program Slicing". *IEEE Transactions on Software Engineering,* Vol. 10 No. 4, July 1984, pp. 352-357.

[8] F. Tip, "A Survey of Program Slicing Techniques". *Journal of Programming Languages 3*, 1995, pp. 121-189.

BIOGRAPHY

Nor Adnan Yahaya obtained his PhD in Computer Science from Northwestern University, USA in 1987. Currently, he is a principal researcher at Telekom R & D Sdn. Bhd. He is also an Adjunct Associate Professor at the Faculty of Computer Science & IT, Universiti Putra Malaysia. His research areas include Software Engineering and IT Security.

Hamed J. Al-Fawareh obtained his PhD in Software Engineering from Universiti Putra Malaysia. Currently, he is a lecturer in Computer Science Department, Zarka Private University/Jordan. His research areas include Software Engineering, Software Maintenance, Program Slicing, and Object-Oriented Programming.

Abdul Azim Abd. Ghani obtained his PhD in Software Engineering from Strathclyde University, UK. Currently, he is the Dean of the Faculty of Computer Science and IT, Universiti Putra Malaysia. His research areas include Software Engineering, Software Metrics, and System Engineering.

APPENDIX A: A SAMPLE JAVA SOURCE CODE

```
        public class MyInput {
        public static int readInt() {       …       }
        public static double readDouble(){… }
        public static String readString(){… }
          }
        public class Cylinder extends Circle {
        private double length;
        public Cylinder()   {
1               length = MyInput.readDouble();
          }
        public Cylinder(double r, double l)  {
2               length = 1;
          }
        public double getLength()  {
3             return length;
          }
        public double findVolume() {
4                 return findArea()*length;
          }
          }
        public class TestComputeArea {
        static double radius;
        static double area;
        static double PI = 3.14159;
        public static void main(String[] args) {
1           System.out.println("Enter radius");
2           radius = MyInput.readDouble();
3           Circle myCircle = new Circle(radius);
4           System.out.println(myCircle.findArea());
5           Cylinder myCylinder = new Cylinder();
6           System.out.println(myCylinder.findArea());
7        System.out.print(myCylinder.findVolume());
       }
       }

       class Circle {
        private double radius;
        String color;
        static double weight;
        public Circle(double r, String c, double w) {
1              radius = r;
2              color = c;
3              weight = w;
          }
         public Circle(double r, String c){
4              radius = r;
5              color = c;
          }
         public Circle() {
6              radius = MyInput.readDouble();
7              color = "white";
8              weight = 1.0;
          }
         public double getRadius() {
9              return radius;
          }
         public double findArea()  {
10           color = "Blue";
11           weight = 10;
12             return radius*radius*Math.PI;
          }
          }
          class Discount extends Salary{
         protected float rate;
         float Total, discount;
         String status;
         void increase() {
1                status = MyInput.readString();
2                if(status == "student")
2.1                   rate = 20/100;
3                if(status == "pensioner")
3.1                   rate = 17/100;
4                Total = node.Total;
5                node.Total=node.Total -node.Total*rate;
6                discount = Total - node.Total;
7                System.out.println("discount :" +discount);
8                System.out.println("Total Before Discount: " + Total);
9                System.out.println("Total After Discount: " + node.Total);
          }
          }
```