A FRAMEWORK TO MEASURE CLASS COHESION

Xu Baowen* ** Chen Zhenqiang* Zhou Yuming*

*(Department of Computer Science and Engineering, Southeast University, Nanjing 210096)
**(State Key Laboratory of Software Engineering, Wuhan University, Wuhan 430072)

Abstract Classes are the basic modules in Object-Oriented (OO) software, which consist of attributes and methods. Thus, in OO environment, the cohesion is mainly about how tightly the attributes and methods of classes cohere with each other. This letter discusses the relationships between attributes and attributes, attributes and methods, methods and methods of a class, and the properties of these relationships. Based on these properties, the letter proposes a new framework to measure the cohesion of a class. The approach overcomes the limitations of previous class cohesion measures, which consider only one or two of the three relationships in a class.

Key words Object-Orientation(OO); Software measurement; Dependence analysis; Cohesion

I. Introduction

Cohesion is one of the most important features during software development. In an object-oriented (OO) programming language, classes are the basic facilities to support OO features, which consist of a set of attributes to represent the states and a set of methods to represent operations on attributes. Thus, in OO environment, the cohesion is mainly about how tightly the attributes and methods of classes cohere with each other.

There are many approaches proposed in literatures to measure the class cohesion[1–6]. Most approaches focus on the interaction between methods and attributes. The cohesion is measured as the number of the interactions. Generally only the references from methods to attributes are considered. And none care about the interactions of attributes to attributes and methods to methods at the same time. This might lead to bias when measuring the cohesion of a class.

In our previous works, we have done some research in measuring class cohesion[7–10]. This letter extends these works to measure the tightness of the attributes and methods of a class in a more clear view, and goes deep in the relationships between attributes and attributes, methods and attributes, methods and methods. Thus, the approach overcomes the limitations of previous class cohesion measures, which consider only one or two of the three facets.

II. Basic Definitions and Properties

In this letter, the relationships among attributes and methods are defined as three dependencies: inter-attribute, inter-method and method-attribute dependencies.

Definition 1 In a method of a class, if the definition (modification) of attribute A uses (refer, but not modify) the attribute B directly or indirectly, or whether A can been defined is determined by the state of B, A depends on B, which is denoted by $A \rightarrow B$.

1Manuscript received date: January 21, 2002; revised date: April 11, 2002.
Supported in part by the National Natural Science Foundation of China(NSFC) (No.60073012), Natural Science Foundation of Jiangsu (BK2001004).
Generally, if \( B \) is used in the \(<\text{condition part}\>\) of a control statement, such as "if" and "while" statement, and the definition of \( A \) is in the inner statement of the control statement, the definition of \( A \) depends on \( B \)'s state. According to Definition 1, we have

**Property 1** The dependencies among attributes are transitive, i.e., \( A \rightarrow B, B \rightarrow C \Rightarrow A \rightarrow C \).

According to Property 1, we define inter-attribute transitive dependence as Definition 2:

**Definition 2** If there exist attributes \( A_1, \ldots, A_n, (n > 1), A_1 \rightarrow A_2, \ldots, A_{i-1} \rightarrow A_i, \ldots, A_{n-1} \rightarrow A_n, A_1 \) transitively depends on \( A_n \), which is denoted by \( A_1 \rightarrow^* A_n \).

**Definition 3** If attribute \( A \) is referred in \( M \), \( M \) depends on \( A \), which is denoted by \( M \rightarrow A \).

**Definition 4** There are two types of dependencies between methods: call dependence and potential dependence. If \( P \) is called in \( M \), \( M \) call depends on \( P \), which is denoted by \( M \rightarrow P \). If there is an attribute used in \( M \) before it is defined, and \( A \) is defined in \( P \), \( M \) potentially depends on \( P \), which is denoted by \( M \rightarrow^* P \).

Given a class, we can not assume which method might be invoked before other methods except some special methods, such as constructor, initialization and so on. We assume all these methods can be invoked at any time except the constructor, initialization and destructor. Therefore, if \( M \) might use an attribute \( A \) and \( A \) is defined in \( P \), if \( P \) is invoked first and then \( M \) is invoked, \( M \) might use the \( A \) defined in \( P \), i.e., \( M \) potentially depends on \( P \).

To obtain the dependencies between methods, we introduce two attribute sets: OUT and IN for method \( M \), where \( \text{IN}(M) \) is an attribute set, each element of which is an attribute referred before modifying its value in \( M \). \( \text{OUT}(M) \) is an attribute set, each element of which is an attribute modified in \( M \). Thus, we have: \( A \in \text{IN}(M), A \in \text{OUT}(P) \Rightarrow M \rightarrow P \).

**Property 2** The calls among methods are transitive, i.e., \( M \rightarrow P, P \rightarrow Q \Rightarrow M \rightarrow Q \).

**Property 3** The inter-method potential dependencies are not transitive, i.e., \( M \rightarrow P, P \rightarrow Q \) might not hold.

If \( M \rightarrow P \) and \( P \rightarrow Q \) are introduced by unrelated and different attributes, \( M \) might has no relation with \( Q \). Thus, inter-method potential dependence is not transitive.

The intransitivity among inter-method potential dependencies leads to great difficulties to analyze. Thus, we redefine the dependence among methods.

**Definition 5** If the attribute \( A \) used in \( M \) is defined in \( P \), denoted by \( M \rightarrow^* P \), where \( (A,A) \) is named as a tag. For the call edge, add a tag \((*,*)\), i.e., if \( P \rightarrow Q \), \( P \rightarrow^* Q \).

Definition 5 is the basic definition. For the dependencies between attributes are transitive, we can obtain a more general definition according to Property 4.

To obtain this dependence, we introduce another two sets: DEP-A and DEP-A-OUT, each element of which has the form of \(<A,B>\), where \( A, B \) are attributes of the class. \( \text{DEP-A}(M) \) represents the dependencies from the attributes referred in \( M \) to the attributes defined out \( M \). \( \text{DEP-A-OUT}(M) \) records the dependencies from the attributes referred in \( M \) to the attributes defined out \( M \) when exiting \( M \).

In general, the intermediate results are invisible outside, and an attribute might be modified many times in a method. We introduce DEP-A-OUT to improve the precision. Obviously, we have \( \text{DEP-A-OUT}(M) \subseteq \text{DEP-A}(M) \). According to the definition above, we have the following properties:

**Property 4** \(<A,B> \in \text{DEP-A}(M), B \in \text{OUT}(P) \Rightarrow M \rightarrow^* P \).