CHAPTER 3
NEW COMPLEXITY MODEL FOR CLASSES

3.1 Introduction
3.2 Overview of Software Complexity and its Measurement
   3.2.1 Software Complexity
   3.2.2 Software Complexity Measurement
3.3 Proposed Complexity Model
   3.3.1 Base of Proposed Complexity Model
   3.3.2 Property of Inheritance
   3.3.3 Metrics for Computing Method Complexity
3.4 Case Study and Experimental Results
3.5 Analysis of Experimental Results
3.6 Summary

3.1 INTRODUCTION

Today most of the software development is based on object oriented methodology. Followers of object oriented methodology state that professional software production becomes notably simplified using this technique, which results in a significant cost decrease [Fothi03]. Object oriented methodology is also the base of other modern software development methodologies such as CBSD and AOSD. Modern object oriented software development technologies such as .NET and Java are rich from features that are capable of developing highly maintainable, reusable, testable and reliable software. However, extremely large size and dependencies among classes makes the software very hard to understand and maintain. Software complexity always remains a major issue in all the software development paradigms because it affects all other attributes of the software. It is accepted by both researchers and developers that complexity of software can be controlled in much better way by using object oriented approach as compared to the
procedure oriented approach. Object oriented approach controls complexity of a system by supporting hierarchical decomposition through both data and procedural abstraction [Da-wai07]. Nevertheless, the complexity of software is an essential property, not an accidental one [Brooks95]. Therefore, it is important to understand the software complexity in context of various factors that affects it. Traditional software complexity measures and models are not appropriate for object oriented software due to distinguish features mainly inheritance and polymorphism. This chapter proposes a complexity model for classes in object oriented software system, which defines Class Complexity (CC) as a sum of complexity of its methods. The Method Complexity (MC) is further defined in terms of complexity due to logic of method, method calls and data calls. One traditional Cyclomatic Complexity metric and two new metrics namely, Total Method Call Complexity (TMCC) and Total Data Call Complexity (TDCC) are used in this account. CFC of a method is computed using cyclomatic complexity metric. TMCC of a method is sum of Method Call Complexity (MCC) of each method call used in the method. MCC of a method call depends upon the type of method call. Different complexity level is assigned to different type of method calls. Similarly, TDCC of a method is sum of Data Call Complexity (DCC) of each data call used in the method. DCC of a data call depends upon the way data is called. Different complexity level is assigned to different type of data calls. Higher value of CC makes the class hard to understand and maintain. Rest of the chapter is organized as follows: Section 3.2 presents various definitions, classifications and measurement of software complexity. Existing complexity metrics and models are also discussed in this section. Section 3.3 explains the proposed model and its various metrics. In Section 3.4 model is applied on an application Sorter Searcher Selector (SSS) of Java language and results are compared with four CK metrics [CK94]: WMC, RFC, DIT and CBO. Section 3.5 presents critical analysis of experimental results. Finally, Section 3.6 presents summary of whole chapter.
3.2. OVERVIEW OF SOFTWARE COMPLEXITY AND ITS MEASUREMENT

3.2.1 Software Complexity

In literature, software complexity is defined by many researches in different ways. [Basili80] defines complexity as a measure of the resources expended by a system while interacting with a piece of software to perform a given task. The interacting system may be a human being or machine. In case of computer as an interacting system, complexity is defined in terms of execution time and storage space required to perform the computation. In case of programmer as an interacting system, complexity is defined by the difficulty of performing tasks such as coding, debugging, testing or modifying the software. The term software complexity is often applied to the interaction between a program and a programmer working on some programming task. This definition of software complexity is too broad. According to Zuse [Zuse91] software complexity is poorly defined and complexity measure is a misnomer. The true meaning of the term software complexity is the difficulty to maintain, change and understand software. It deals with the psychological complexity of programs. According to Henderson-Sellers [Sellers96] the cognitive complexity of software refers to those characteristics of software that affect the level of resources used by a person performing a given task on it. It is not possible to define software complexity completely by a single definition because it is multidimensional attribute of software. In addition, different stakeholders hold different views on software complexity definitions. Therefore, no standard definition exists in literature for software complexity. However, the importance of studying software complexity is very much clear. Software complexity is indicator of efforts (development, testing, maintenance etc.), fault-prone modules, reliability and defect rate. Complex software is difficult to develop, test, debug, maintain and has higher defect rate. In literature different type of classifications of software complexity are available as shown in Figure 3.1. According to Bill Curtis [Curtis80] there are two types of software complexities: Algorithmic and Psychological. Algorithmic (or computational) complexity characterizes the run-time performance of an algorithm whereas psychological complexity affects the performance of programmers trying to understand or modify a code module. According to Brooks [Brooks95] the complexity of software is an essential...
property, not an accidental one. Essential complexity arises from the nature of the problem and how deep a skill set is needed to understand the problem. Accidental complexity is the result of poor attempts to solve the problem and may be equivalent to what some are calling complication. Applying the wrong design pattern or selecting an inappropriate data structure adds accidental complexity to a problem. Wood's task complexity model [Wood86] has three dimensions: component, coordinative and dynamic complexity. Component complexity refers to the number of distinct information cues that must be processed in the execution of a task. Coordinative complexity takes into account the forms, strength and interdependencies of the relationships between the information cues. Dynamic complexity arises from both the changes in information cues and their interrelationships over time. De Tran-Cao et al. [Tran-Cao04] maps first two dimensions of wood's task complexity model to software functional complexity model which consist of system and component complexity. System complexity refers to data communications between functions. Component complexity refers to data groups and relationships between inputs and outputs. W. Xia et al. [Xia04] discusses two types of project complexities in terms of project management and risk factors. 1) Organizational complexity: It includes number and types of relationships among hierarchical levels,
formal organizational units and specialization. 2) Technological complexity: It includes number and types of relationships among inputs, outputs, tasks, and technologies. Both Organizational and Technological complexities may be static as well as dynamic. Correlation analysis between complexity and project performance attributes (project delivery time, cost, system functionality and end-user satisfaction) shows that higher complexity causes poor project performance. [McGregor06] discusses the differences between complex and complicated. Complex and complicated represents the modular and non-modular system respectively. A complex system consists of many parts and interfaces between them. Higher number of interfaces between the parts makes the system more time consuming to understand. Some good features such as encapsulation, information hiding and modularity makes the software as a quality product but also introduce some complexities due to interactions and relationships among various modules. A complicated system is difficult to understand and analyze because of no architectural coherence. Complicated system may or may not be complex and may be more efficient as compared to complex system. According to [Da-wai07] there are four types of software complexities: Domain, Scale, Artificial and Functional. Domain complexity is inherited in the problem. Scale complexity introduced due to size of problem. Artifacts used in building software cause artificial complexity. Functional complexity is candidate variable for computing efforts to develop the software function.

3.2.2 Software Complexity Measurement

Software complexity is unavoidable; it can’t be removed fully, only it can be controlled. Controlling software complexity needs metrics to measure it. Many metrics have been proposed by various researchers to measure different aspects of software complexity with the goal of evaluating, predicting and controlling it. The best known early approach to measure software complexity is Halstead’s software science metrics, McCabe’s Cyclomatic Complexity metric and Kafura & Henry’s Fan-in Fan-out complexity metrics. Many unique features mainly inheritance and polymorphism of object oriented software are not addressed by traditional software complexity metrics therefore, traditional complexity metrics are not suitable for measuring complexity of object oriented software. Various metrics and models have been proposed in literature to
compute the complexity of object oriented software. Chidamber and Kemerer’s [CK94] proposed a suite of six metrics WMC, DIT, NOC, CBO, LCOM and RFC. These metrics measures the complexity of object oriented software by design of classes. WMC do not specify any particular metric to measure the complexity rather simply defines complexity of a class as a sum of complexity of individual methods. Higher value of DIT and NOC is indicator of higher complexity due to involvement of many methods. CBO for a class is the total number of other classes to which it is coupled. CBO is useful to determine the complexity of testing. Lower value of LCOM is indicator of higher complexity. RFC indicates that higher cardinality of set of methods that can be potentially executed in response to a message received by an object of a class is analogous to higher complexity. Fothi et al. [Fothi03] computes complexity of class by complexity of control structures, data and relationships between data and control structures. Authors proposed a metric based on the depth of nesting in each method, number of data and control nodes. [Dawei07] proposed a model for computing complexity of object oriented systems. This model describes complexity at variable, method, object and system level. For each level of complexity various measures are identified which considers both procedural and data characteristics of an object oriented system. [Mishra07] proposed a metric to calculate the complexity of a class at method level by considering the internal structure of method. The proposed metric measures the complexity using cognitive weight of basic control structure (sequence, branch and iteration) and message call. Yadav et al. [Yadav09] proposed a metric to calculate the overall complexity of design hierarchy caused by inherited methods and computes complexity based on count of inherited methods only. However, internal characteristics of methods are not considered by this metric.

3.3 PROPOSED COMPLEXITY MODEL

3.3.1 Base of Proposed Complexity Model

Object oriented software is a collection of various classes and the relationships among them. A class is further consisting of data members and methods. Data members, methods and relationships among classes all contributes to the complexity of software. The model introduced in this chapter determines the class complexity by complexity of its methods. However, there is no single measure that can measure the complexity of the
method. Logic of the method, type of method call (polymorphic or non-polymorphic), type of data call and relationships all are constituents of the method complexity. Higher complexity of method's logic makes it difficult to understand and maintain. Higher number of classes involved in a method call and data call are also indicator of higher complexity. The relationship through which method or data is called also contributes to the complexity of the method. To compute the above complexities it is important to understand the Method and Data Call Graph (MDCG). To construct MDCG of any method only method and data calls within that method are considered. For example, Figure 3.2 shows the MDCG of a method M that calls two other methods M1, M2 and two data member D1, D2. The solid circle shaped node represents the non-polymorphic method call, dotted circle shaped node represents polymorphic method call and box shaped node represents the data call. Each arc represents how method/data is called. M1 is called by class name, M2 and D2 are called by object and D1 is called by calling object of the method M. If nothing is specified on arc then by default calling object of the method M is used to call method/data. In case of the polymorphic call a weight is assigned to corresponding node. Weight of the node is total number of potential method that can be called in polymorphic call. In case of non-polymorphic method call, weight of node is 1.

In general, one of following three ways can be used to call any method:

1. Class.method()
2. Object.method()
3. Method()

First type of method call is used for static method of class. In this case, method is a member of the same class with which method is called. Second type of method call is generally used to call methods of coupled class. In third case object to call the method is not specified. In this case by default the calling object is used to call the method. Similarly data member can be called by three ways. Each type of call has different level of complexity.
On the basis of above discussion, three measures (Control Flow Complexity, Total Method Call Complexity and Total Data Call Complexity) are derived to compute complexity of a method. Figure 3.3 illustrates the proposed complexity model.

FIGURE 3.2 MDCG OF METHOD M

FIGURE 3.3 PROPOSED COMPLEXITY MODEL
Method Complexity (MC) = Control Flow Complexity (CFC) + Total Method Call Complexity (TMCC) + Total Data Call Complexity (TDCC)

\[
\text{Class Complexity}(CC) = \sum_{\text{for each method of the class}} \text{MC}
\]

### 3.3.2 Property of Inheritance

Before understanding the complexity measures of method, it is necessary to understand the “Property of Inheritance”. According to the property of inheritance, if any method/data member is not found in the current class then it is searched from its base class. If the required method/data member is not present in the base class then it is searched from base of base class and this procedure continues until the required method/data member is found.

### 3.3.3 Metrics for Computing Method Complexity

Following notations are used to compute the method complexity:

- \( N_{\text{overload}} \) = Number of overloaded versions of a method.
- \( N_{\text{traverse,o}} \) = Total number of classes traversed to find the required object.
- \( N_{\text{traverse,m}} \) = Total number of classes traversed to find the required method.
- \( N_{\text{traverse,d}} \) = Total number of classes traversed to find the required data member.
- OCC = Object Call Complexity i.e. complexity of the object using which method/data member is called.

Following three metrics are used to compute the complexity of a method.

#### 3.3.3.1 Control Flow Complexity (CFC)

CFC can be computed by traditional McCabe’s cyclomatic complexity metric, using control flow graph.

#### 3.3.3.2 Total Method Call Complexity (TMCC)

TMCC of a method (say M) is sum of Method Call Complexity (MCC) of each method call used in M.

\[
\text{TMCC} = \begin{cases} 
0 & \text{if no methodcall} \\
\sum_{\text{for each methodcall}} \text{MCC} & \text{otherwise}
\end{cases}
\]
Method Call Complexity (MCC) of any method call depends upon the way method is called. As mentioned in section-3.3.1 any method can be called by one of three ways. Each type of method call has different level of complexity which can be computed as follows.

**Class.method()**
The complexity of this type of method call is computed as follows

\[
\text{MCC} = 1 + N_{\text{overload}}
\]

If method is not overloaded then \(N_{\text{overload}} = 0\). As class name is specified with method call therefore, only specified class contains the required method. Hence, only one class is traversed but, if the method is overloaded then further efforts are required to identify required method out of \(N_{\text{overload}}\) versions of method.

**Object.method()**
The complexity of this type of method call is computed as follows:

\[
\text{MCC} = \text{OCC} + (P \times N_{\text{traverse},m} + N_{\text{overload}})
\]

First the object with which method is called is searched from the same method in which this method call is used. If object found here then its complexity is 1 otherwise it is searched from the containing class if object found in that class then its complexity is 2 otherwise property of inheritance is used to search the object. Therefore,

\[
\text{OCC} = 1 + N_{\text{traverse},o}
\]

Called method is searched from the class whose object is used to call the method. If method does not exists in that class then property of inheritance is used to reach at method definition.

If method is not overloaded then \(N_{\text{overload}} = 0\). \(P\) is the total number of possible classes whose method can be bond during runtime. In case of static binding \(P = 1\).

**Method()**
The complexity of this type of method call is computed as follows:

\[
\text{MCC} = N_{\text{traverse},m} + N_{\text{overload}}
\]
In this case, method is searched from same class, otherwise property of inheritance is used to reach at the destination class. \( N_{\text{overload}} = 0 \) if method is not overloaded.

### 3.3.3.3 Total Data Call Complexity (TDCC)

TDCC of a method (say M) is sum of Data Call Complexity (DCC) of each data call used in M.

\[
TDCC = \begin{cases} 
0 & \text{if no data call} \\
\sum \text{DCC} & \text{otherwise}
\end{cases}
\]

One of following three ways can call any data member. Each type of data call has different level of complexity which can be computed as follows:

**Type.datamember**

The complexity of this type of data call is 1. This kind of data call is used for static data members. The specified type name (class, enum etc.) contains the required data member. Therefore, only one type is traversed to find the datamember, hence its complexity is 1.

**Object.datamember**

The complexity of this type of data call is computed as follows:

\[
DCC = OCC + N_{\text{traverse, } d}
\]

Where \( OCC = 1 + N_{\text{traverse, } d} \)

OCC is computed in same way as discussed above in case of object.method() call. This type of data member call is generally used to call data member of coupled class. In that case, data member is searched from the class whose object is used to call the data member. If data member does not exist in that class then property of inheritance is used to reach at data member definition.

**Datamember**

The complexity of this type of data call is computed as follows:

\[
DCC = N_{\text{traverse, } d}
\]
In this case datamember is searched from same class, if data member does not exists in that class then property of inheritance is used to reach at the destination class.

Consider a class `Result` that is derived from the class `Exam` as shown below (partial code in Java):

```java
class Result extends Exam {
    char grade;
    -----
    -----
    public void findGrade()
    {
        int t;
        t=total();
        if(t<=Marks.getMin())
            grade='C'
        else if(t<Marks.getAvg())
            grade='B'
        else
            grade='A';
    }
}
```

**FIGURE 3.4 MDC GRAPH OF findGrade() METHOD**

Method `total()` belongs to `Exam` class and `findGrade()` belongs to `Result` class. The MDCG of method `findGrade()` is shown in Figure 3.4. CFC of `findGrade()` method is 3. `findGrade()` method calls three methods: `total()`, `getMin()`, `getAvg()` and one data member `grade`. Complexity of `Marks.getMin()` is 1 because it is called by class name `Marks`. 
Similarly complexity of `Marks.getMax()` is 1. Method call complexity of method call `total()` is 2 because two class are searched to reach at `total()` method. First the method is searched from `Result` class as it does not belongs to `Result()` class therefore, its base class `Exam` is traversed to reach at `total()` method. As MCC of `total()`, `Marks.getMin()`, `Mark.getAvg()` is 2, 1 and 1 respectively, therefore, TMCC of `findGrade()` method is 2+1+1=4. The method `findGrade()` use one data member i.e. `grade` which belongs to same class therefore, its DCC is 1.

CFC=3  
MCC=4  
DCC=1  

Complexity of `findGrade()` = CFC + MCC + DCC  
Hence complexity of `findGrade()` is 3+4+1=8.

**3.4 CASE STUDY AND EXPERIMENTAL RESULTS**

Proposed model is applied on small-scale extensible application SSS, developed in Java language. SSS accepts data in array; analyze pattern of data and selects suitable sorting or searching algorithm at runtime according to the pattern of data. The class diagram of SSS is shown in Figure 3.5. Same application is also analyzed using four CK metrics: WMC, RFC, DIT and CBO. Results are presented in Table 3.1, Table 3.2 and Table 3.3.

Following points are considered while applying the model

- If same method/data member is referenced more than once then it is counted only once.
- Library classes/methods/data-members are not taken into account.
- Try block is treated as a predicate statement.
- Enum and Interface type is treated as equivalent to class.
- Implementation relationship between class and interface is treated as coupling relationship.
FIGURE 3.5 CLASS DIAGRAM OF SSS
### CHAPTER 3
NEW COMPLEXITY MODEL FOR CLASSES

#### TABLE 3.1 RESULTS OF PROPOSED COMPLEXITY MODEL ON SSS

<table>
<thead>
<tr>
<th>Type Name</th>
<th>Method</th>
<th>TMCC</th>
<th>TDCC</th>
<th>CFC</th>
<th>MC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>Array()</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Array(int)</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>getElements()</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>displayElements()</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>toArray()</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>analysePattern()</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>getPatternTypeForSorter()</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>getPatternTypeForSearcher()</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>BinarySearchAsc</td>
<td>Search(Array,int)</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>BinarySearchDesc</td>
<td>Search(Array,int)</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>Chooser</td>
<td>ChooseSorter(Array)</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>ChooseSearcher(Array)</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>InsertionSort</td>
<td>Sort(Array)</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>ISearcher</td>
<td>Search(Array,int)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ISorter</td>
<td>Sort(Array)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>QuickSort</td>
<td>Sort(Array)</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Qsort(int [], int, int)</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>divide(int [], int, int)</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Register</td>
<td>Register(PatternType, ISorter)</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Register(PatternType, ISearcher)</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Searcher</td>
<td>Searcher(int)</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Search(int)</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>SearcherNotFoundException</td>
<td>SearcherNotFoundException(String)</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>SequentialSearch</td>
<td>Search(Array,int)</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>Sorter</td>
<td>Sorter(int)</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Sort()</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>SorterNotFoundException</td>
<td>SorterNotFoundException(String)</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>PatternType</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
TABLE 3.2 RESULTS OF PROPOSED COMPLEXITY MODEL AND CK METRICS ON SSS

<table>
<thead>
<tr>
<th>Type Name</th>
<th>Proposed Model</th>
<th>CK Metrics Suite</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CC</td>
<td>WMC</td>
</tr>
<tr>
<td>Array</td>
<td>31</td>
<td>17</td>
</tr>
<tr>
<td>BinarySearchAsc</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>BinarySearchDesc</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>Chooser</td>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td>InsertionSort</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>Isearcher</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Isorter</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>QuickSort</td>
<td>13</td>
<td>8</td>
</tr>
<tr>
<td>Register</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Searcher</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>SearcherNotFoundException</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>SequentialSearch</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>Sorter</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>SorterNotFoundException</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>PatternType</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

TABLE 3.3 CORRELATION COEFFICIENT

<table>
<thead>
<tr>
<th>Correlation of CC with</th>
<th>Correlation Coefficient</th>
</tr>
</thead>
<tbody>
<tr>
<td>WMC</td>
<td>0.97</td>
</tr>
<tr>
<td>RFC</td>
<td>0.80</td>
</tr>
<tr>
<td>CBO</td>
<td>0.39</td>
</tr>
</tbody>
</table>

3.5 ANALYSIS OF EXPERIMENTAL RESULTS

Sum of CFC of all the methods of a class is same as WMC. RFC counts the total number of methods in the class and the methods that can be called by methods of class. RFC do not consider how methods are called. CBO counts the total number of coupled classes. MCC/DCC assigns different complexity level to method/data calls depending upon the way method/data is called. Correlation analysis shows that complexity computed by proposed model is more closely related to WMC and RFC value of class. The proposed complexity model computes the complexity of a class by combining complexity aspects of WMC, CBO, RFC and DIT. The correlation coefficient of CC with WMC, RFC and CBO is 0.97, 0.80, and 0.39 respectively. Finding correlation coefficient between CC and DIT needs data that should be more appropriate, but it is believed that DIT is also highly correlated with CC.
3.6 SUMMARY

Software complexity measurement is always an area of interest in any software development discipline. Many different views on software complexity are available in literature but all agree on that software complexity is an indicator of defect rate, reliability, development efforts and maintenance cost. Many metrics/models are available in literature to measure complexity of object-oriented software systems from different aspects such as inheritance, polymorphism, etc. This chapter presents a new complexity model for classes in object-oriented systems that integrates the complexity due to control flow, inheritance, and polymorphism. The given model computes the class complexity as a sum of complexity of its methods. Method complexity is further computed as a sum of Control Flow Complexity (CFC), Total Method Call Complexity (TMCC), and Total Data Call Complexity (TDCC). CFC of a method is computed using cyclomatic complexity metric. TMCC is the sum of Method Call Complexity (MCC) of each method call. Different types of method calls have different complexity which depends upon the way the method is called. TDCC is the sum of Data Call Complexity (DCC) of each data call. Different types of data calls have different complexity which depends upon the way the data is called. Method Data Call Graph (MDCG) is used to compute MCC and DCC. The basic principle behind computing MCC and DCC is to compute the total number of classes traversed to understand the method and data call. Different metrics are proposed to compute complexity of different types of method and data call. The proposed model shows that number of method/data calls, type of method/data calls, number of classes involved in method/data calls, and control flow of methods highly influence the complexity of classes. Higher complexity of classes makes it hard to understand and maintain. The proposed model and four CK metrics are applied on software developed in Java language. Comparison with four CK metrics shows that CC is highly positively correlated with WMC and RFC as compared to CBO. It is believed that CC is also highly positively correlated with DIT due to involvement of many classes in method/data calls.