3

Software Metrics

Software metrics are used to assess the quality of the product or process used to build it. The metrics allow project managers to gain insight about the progress of software and assess the quality of the various artifacts produced during software development. The software analysts can check whether the requirements are verifiable or not. The metrics allow management to obtain an estimate of cost and time for software development. The metrics can also be used to measure customer satisfaction. The software testers can measure the faults corrected in the system, and this decides when to stop testing.

Hence, the software metrics are required to capture various software attributes at different phases of the software development. Object-oriented (OO) concepts such as coupling, cohesion, inheritance, and polymorphism can be measured using software metrics. In this chapter, we describe the measurement basics, software quality metrics, OO metrics, and dynamic metrics. We also provide practical applications of metrics so that good-quality systems can be developed.

3.1 Introduction

Software metrics can be used to adequately measure various elements of the software development life cycle. The metrics can be used to provide feedback on a process or technique so that better or improved strategies can be developed for future projects. The quality of the software can be improved using the measurements collected by analyzing and assessing the processes and techniques being used.

The metrics can be used to answer the following questions during software development:

1. What is the size of the program?
2. What is the estimated cost and duration of the software?
3. Is the requirement testable?
4. When is the right time to stop testing?
5. What is the effort expended during maintenance phase?
6. How many defects have been corrected that are reported during maintenance phase?
7. How many defects have been detected using a given activity such as inspections?
8. What is the complexity of a given module?
9. What is the estimated cost of correcting a given defect?
10. Which technique or process is more effective than the other?
11. What is the productivity of persons working on a project?
12. Is there any requirement to improve a given process, method, or technique?

The above questions can be addressed by gathering information using metrics. The information will allow software developer, project manager, or management to assess, improve, and control software processes and products during the software development life cycle.

3.1.1 What Are Software Metrics?

Software metrics are used for monitoring and improving various processes and products in software engineering. The rationale arises from the notion that "you cannot control what you cannot measure" (DeMarco 1982). The most essential and critical issues involved in monitoring and controlling various artifacts during software development can be addressed by using software metrics. Goodman (1993) defined software metrics as:

> The continuous application of measurement based techniques to the software development process and its products to supply meaningful and timely management information, together with the use of those techniques to improve that process and its products.

The above definition provides all the relevant details. Software metrics should be collected from the initial phases of software development to measure the cost, size, and effort of the project. Software metrics can be used to ascertain and monitor the progress of the software throughout the software development life cycle.

3.1.2 Application Areas of Metrics

Software metrics can be used in various domains. One of the key applications of software metrics is estimation of cost and effort. The cost and effort estimation models can be derived using the historical data and can be applied in the early phases of software development.

Software metrics can be used to measure the effectiveness of various activities or processes such as inspections and audits. For example, the project managers can use the number of defects detected by inspection technique to assess the effectiveness of the technique. The processes can be improved and controlled by analyzing the values of metrics. The graphs and reports provide indications to the software developers and they can decide in which direction to move.

Various software constructs such as size, coupling, cohesion, or inheritance can be measured using software metrics. The alarming values (thresholds) of the software metrics can be computed and based on these values and then the required corrective actions can be taken by the software developers to improve the quality of the software.

One of the most important areas of application of software metrics is the prediction of software quality attributes. There are many quality attributes proposed in the literature such as maintainability, testability, usability, and reliability. The benefits of developing the quality models is that they can be used by software developers, project managers, and management personnel in the early phases of software development for resource allocation and identification of problematic areas.

Testing metrics can be used to measure the effectiveness of the test suite. These metrics include the number of statements, percentage of statement coverage, number of paths covered in a program graph, number of independent paths in a program graph, and percentage of branches covered.

Software metrics can also be used to provide meaningful and timely information to the management. The software quality, process efficiency, and people productivity can be computed using the metrics. Hence, this information will help the management in

making effective decisions. The effective application of metrics can improve the quality of the software and produce software within the budget and on time. The contributions of software metrics in building good-quality system are provided in Section 3.9.1.

3.1.3 Characteristics of Software Metrics

A metric is only relevant if it is easily understood, calculated, valid, and economical:

1. Quantitative: The metrics should be expressible in values.
2. Understandable: The way of computing the metric must be easy to understand.
3. Validatable: The metric should capture the same attribute that it is designed for.
4. Economical: It should be economical to measure a metric.
5. Repeatable: The values should be same if measured repeatedly, that is, can be consistently repeated.
6. Language independent: The metrics should not depend on any language.
7. Applicability: The metric should be applicable in the early phases of software development.
8. Comparable: The metric should correlate with another metric capturing the same feature or concept.

3.2 Measurement Basics

Software metrics should preserve the empirical relations corresponding to numerical relations for real-life entities. For example, for "taller than" empirical relation, ">" would be an appropriate numeric relation. Figure 3.1 shows the steps of defining measures. In the first step, the characteristics for representing real-life entities should be identified. In the third step, the empirical relations for these characteristics are identified. The third step

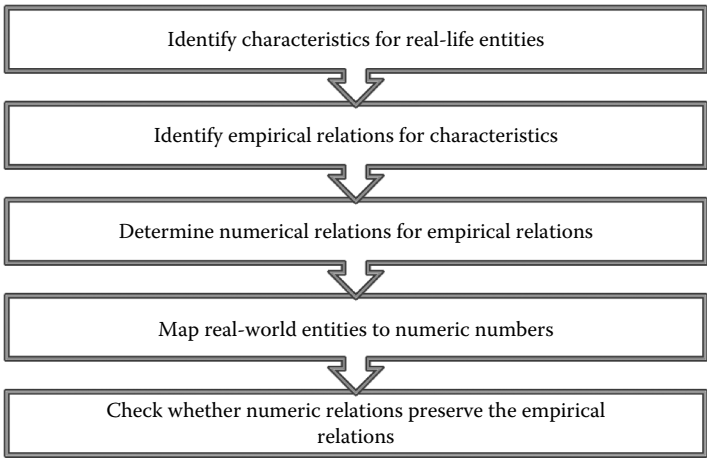

FIGURE 3.1
Steps in software measurement.

determines the numerical relations corresponding to the empirical relations. In the next step, real-world entities are mapped to numeric numbers, and in the last step, we determine whether the numeric relations preserve the empirical relation.

3.2.1 Product and Process Metrics

The entities in software engineering can be divided into two different categories:

1. Process: The process is defined as the way in which the product is developed.
2. Product: The final outcome of following a given process or a set of processes is known as a product. The product includes documents, source codes, or artifacts that are produced during the software development life cycle.

The process uses the product produced by an activity, and a process produces products that can be used by another activity. For example, the software design document is an artifact produced from the design phase, and it serves as an input to the implementation phase. The effectiveness of the processes followed during software development is measured using the process metrics. The metrics related to products are known as product metrics. The efficiency of the products is measured using the product metrics.

The process metrics can be used to

1. Measure the cost and duration of an activity.
2. Measure the effectiveness of a process.
3. Compare the performance of various processes.
4. Improve the processes and guide the selection of future processes.

For example, the effectiveness of the inspection activity can be measured by computing costs and resources spent on it and the number of defects detected during the inspection activity. By assessing whether the number of faults found outweighs the costs incurred during the inspection activity or not, the project managers can decide about the effectiveness of the inspection activity.

The product metrics are used to measure the effectiveness of deliverables produced during the software development life cycle. For example, size, cost, and effort of the deliverables can be measured. Similarly, documents produced during the software development (SRS, test plans, user guides) can be assessed for readability, usability, understandability, and maintainability.

The process and product metrics can further be classified as internal or external attributes. The internal attribute concerns with the internal structure of the process or product. The common internal attributes are size, coupling, and complexity. The external attributes concern with the behavior aspects of the process or product. The external attributes such as testability, understandability, maintainability, and reliability can be measured using the process or product metrics.

The difference between attributes and metrics is that metrics are used to measure a given attribute. For example, size is an attribute that can be measured through lines of source code (LOC) metric.

The internal attributes of a process or product can be measured without executing the source code. For instance, the examples of internal attributes are number of paths, number of branches, coupling, and cohesion. External attributes include quality attributes of the system. They can be measured by executing the source code such as the number of failures,

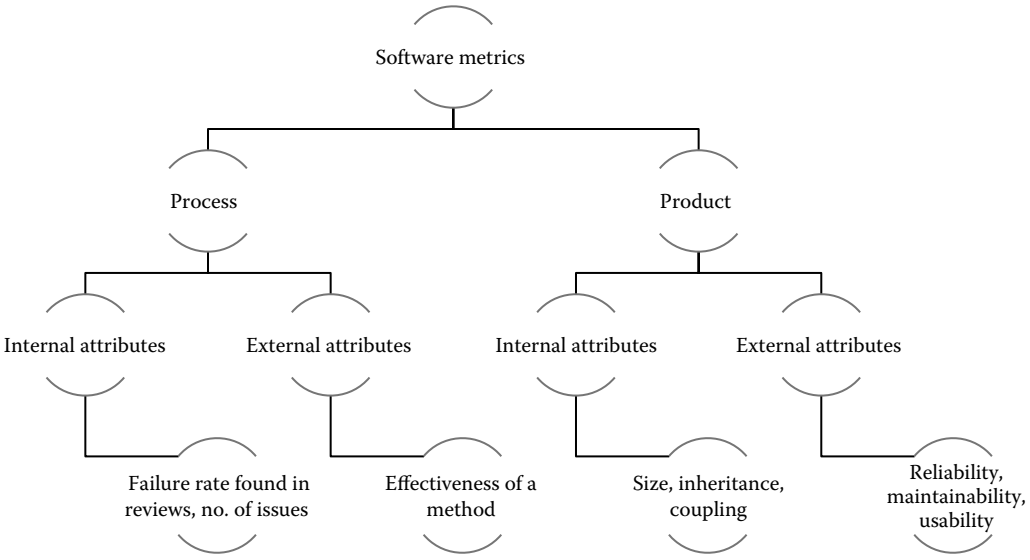

FIGURE 3.2
Categories of software metrics.

response time, and navigation easiness of an item. Figure 3.2 presents the categories of software metrics with examples at the lowest level in the hierarchy.

3.2.2 Measurement Scale

The data can be classified into two types—metric (continuous) and nonmetric (categorical). Metric data is of continuous type that represents the amount of magnitude of a given entity. For example, the number of faults in a class or number of LOC added or deleted during maintenance phase. Table 3.1 shows the LOC added and deleted for the classes A, B, and C.

Nonmetric data is of discrete or categorical type that is represented in the form of categories or classes. For example, weather is sunny, cloudy, or rainy. Metric data can be measured on interval, ratio, or absolute scale. The interval scale is used when the interpretation of difference between values is same. For example, difference between 40°C and 50°C is same as between 70°C and 80°C. In interval scale, one value cannot be represented as a multiple of other value as it does not have an absolute (true) zero point. For example, if the temperature is 20°C, it cannot be said to be twice hotter than when the temperature was 10°C. The reason is that on Fahrenheit scale, 10°C is 50 and 20°C is 68. Hence, ratios cannot be computed on measures with interval scale.

Ratio scales provide more precision as they have absolute zero points and one value can be expressed as a multiple of other. For example, with weight 200 pounds A is twice

TABLE 3.1
Example of Metrics Having Continuous Scale

Class#	LOC Added	LOC Deleted
A	34	5
B	42	10
C	17	9

heavier than B with weight 100 pounds. Simple counts are represented by absolute scale. The example of simple counts is number of faults, LOC, and number of methods. In absolute type of scale, the descriptive statistics such as mean, median, and standard deviation can be applied to summarize data.

Nonmetric type of data can be measured on nominal or ordinal scales. Nominal scale divides metric into classes, categories, or levels without considering any order or rank between these classes. For example, Change is either present or not present in a class.

$$\text{Change} = \begin{cases} 0, \text{ no change present} \\ 1, \text{ change present} \end{cases}$$

Another example of nominal scale is programming languages that are used as labels for different categories. In ordinal scale, one category can be compared with the other category in terms of "higher than," "greater than," or "lower than" relationship. For example, the overall navigational capability of a web page can be ranked into various categories as shown below:

$$\text{What is the overall navigational capability of a webpage?} = \begin{cases} 1, \text{ excellent} \\ 2, \text{ good} \\ 3, \text{ medium} \\ 4, \text{ bad} \\ 5, \text{ worst} \end{cases}$$

Table 3.2 summarizes the differences between measurement scales with examples.

TABLE 3.2

Summary of Measurement Scales

Measurement Scale	Characteristics	Statistics	Operations	Transformation	Examples
Interval	• =, <, > • Ratios not allowed • Arbitrary zero point	Mode, mean, median, interquartile range, variance, standard deviation	Addition and subtraction	$M = xM' + y$	Temperatures, date, and time
Ratio	• Absolute zero point		All arithmetic operations	$M = xM'$	Weight, height, and length
Absolute	• Simple count values		All arithmetic operations	$M = M'$	LOC
Nominal	• Order not considered	Frequencies	None	One-to-one mapping	Fault proneness (0—not present, 1—present)
Ordinal	• Order or rank considered • Monotonic increasing function (=, <, >)	Mode, median, interquartile range	None	Increasing function $M(x) > M(y)$	Programmer capability levels (high, medium, low), severity levels (critical, high, medium, low)

Example 3.1

Consider the count of number of faults detected during inspection activity:

1. What is the measurement scale for this definition?
2. What is the measurement scale if number of faults is classified between 1 and 5, where 1 means very high, 2 means high, 3 means medium, 4 means low, and 5 means very low?

Solution:

1. The measurement scale of the number of faults is absolute as it is a simple count of values.
2. Now, the measurement scale is ordinal since the variable has been converted to be categorical (consists of classes), involving ranking or ordering among categories.

3.3 Measuring Size

The purpose of size metrics is to measure the size of the software that can be taken as input by the empirical models to further estimate the cost and effort during the software development life cycle. Hence, the measurement of size is very important and crucial to the success of the project. The LOC metric is the most popular size metric used in the literature for estimation and prediction purposes during the software development. The LOC metric can be counted in various ways. The source code consists of executable lines and unexecutable lines in the form of blank and comment lines. The comment lines are used to increase the understandably and readability of the source code.

The researchers may measure only the executable lines, whereas some may like to measure the LOC with comment lines to analyze the understandability of the software. Hence, the researcher must be careful while selecting the method for counting LOC. Consider the function to check greatest among three numbers given in Figure 3.3.

The function "find-maximum" in Figure 3.3 consists of 20 LOC, if we simply count the number of LOC.

Most researchers and programmers exclude blank lines and comment lines as these lines do not consume any effort and only give the illusion of high productivity of the staff that is measured in terms of LOC/person month (LOC/PM). The LOC count for the function shown in Figure 3.2 is 16 and is computed after excluding the blank and comment lines. The value is computed following the definition of LOC given by Conte et al. (1986):

> A line of code is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program headers, declarations, and executable and non-executable statements.

In OO software development, the size of software can be calculated in terms of classes and the attributes and functions included in the classes. The details of OO size metrics can be found in Section 3.5.6.

```
/*This function checks greatest amongst three numbers*/
int find-maximum (int i, int j, int k)
        {
                int max;
/*compute the greatest*/
                if(i>j)
                    {
                      if (i<k)
                            max=i;
                      else
                            max=k;
                    }
                      else if (j>k)
                            max=j;
                      else
                            max=k;

/*return the greatest number*/
                     return (max);
        }
```

FIGURE 3.3
Operation to find greatest among three numbers.

3.4 Measuring Software Quality

Maintaining software quality is an essential part of the software development and thus all aspects of software quality should be measured. Measuring quality attributes will guide the software professionals about the quality of the software. Software quality must be measured throughout the software development life cycle phases.

3.4.1 Software Quality Metrics Based on Defects

Defect is defined by IEEE/ANSI as "an accidental condition that causes a unit of the system to fail to function as required" IEEE/ANSI (Standard 982.2). A failure occurs when a fault executes and more than one failure may be associated with a given fault. The defect-based metrics can be classified at product and process levels. The difference of the two terms fault and the defect is unclear from the definitions. In practice, the difference between the two terms is not significant and these terms are used interchangeably. The commonly used product metrics are defect density and defect rate that are used for measuring defects. In the subsequent chapters, we will use the terms fault and defect interchangeably.

3.4.1.1 Defect Density

Defect density metric can be defined as the ratio of the number of defects to the size of the software. Size of the software is usually measured in terms of thousands of lines of code (KLOC) and is given as:

$$\text{Defect density} = \frac{\text{Number of defects}}{\text{KLOC}}$$

The number of defects measure counts the defects detected during testing or by using any verification technique.

Defect rate can be measured as the defects encountered over a period of time, for instance per month. The defect rate may be useful in predicting the cost and resources that will be utilized in the maintenance phase of software development. Defect density during testing is another effective metric that can be used during formal testing. It measures the defect density during the formal testing after completion of the source code and addition of the source code to the software library. If the value of defect density metric during testing is high, then the tester should ask the following questions:

1. Whether the software is well designed or developed?
2. Whether the testing technique is effective in defect detection?

If the reason for high number of defects is the first one then the software should be thoroughly tested to detect the high number of defects. However, if the reason for high number of defects is the second one, it implies that the quality of the system is good because of the presence of fewer defects.

3.4.1.2 Phase-Based Defect Density

It is an extension of defect density metric where instead of calculating defect density at system level it is calculated at various phases of the software development life cycle, including verification techniques such as reviews, walkthroughs inspections, and audits before the validation testing begins. This metric provides an insight about the procedures and standards being used during the software development. Some organizations even set "alarming values" for these metrics so that the quality of the software can be assessed and monitored, thus appropriate remedial actions can be taken.

3.4.1.3 Defect Removal Effectiveness

Defect removal effectiveness (DRE) is defined as:

$$\text{DRE} = \frac{\text{Defects removed in a given life cycle phase}}{\text{Latent defects}}$$

For a given phase in the software development life cycle, latent defects are not known. Thus, they are calculated as the estimation of the sum of defects removed during a phase and defects detected later. The higher the value of the DRE, the more efficient and effective is the process followed in a particular phase. The ideal value of DRE is 1. The DRE of a product can also be calculated by:

$$\text{DRE} = \frac{D_B}{D_B + D_A}$$

where:

D_B depicts the defects encountered before software delivery

D_A depicts the defects encountered after software delivery

3.4.2 Usability Metrics

The ease of use, user-friendliness, learnability, and user satisfaction can be measured through usability for a given software. Bevan (1995) used MUSIC project to measure usability attributes. There are a number of performance measures proposed in this project and the metrics are defined on the basis of these measures. The task effectiveness is defined as follows:

$$\text{Task effectiveness} = \frac{1}{100} \times (\text{quantity} \times \text{quality})\%$$

where:

Quantity is defined as the amount of task completed by a user

Quality is defined as the degree to which the output produced by the user satisfies the targets of a given task

Quantity and quality measures are expressed in percentages. For example, consider a problem of proofreading an eight-page document. Quantity is defined as the percentage of proofread words, and quality is defined as the percentage of the correctly proofread document. Suppose quantity is 90% and quality is 70%, then task effectiveness is 63%.

The other measures of usability defined in MUSIC project are (Bevan 1995):

$$\text{Temporal efficiency} = \frac{\text{Effectiveness}}{\text{Task time}}$$

$$\text{Productive peroid} = \frac{\text{Task time} - \text{unproductive time}}{\text{Task time}} \times 100$$

$$\text{Relative user efficiency} = \frac{\text{User efficiency}}{\text{Expert efficiency}} \times 100$$

There are various measures that can be used to measure the usability aspect of the system and are defined below:

1. Time for learning a system
2. Productivity increase by using the system
3. Response time

In testing web-based applications, usability can be measured by conducting a survey based on the questionnaire to measure the satisfaction of the customer. The expert having knowledge must develop the questionnaire. The sample size should be sufficient enough to build the confidence level on the survey results. The results are rated on a scale. For example, the difficulty level is measured for the following questions in terms of very easy, easy, difficult, and very difficult. The following questions may be asked in the survey:

- How the user is able to easily learn the interface paths in a webpage?
- Are the interface titles understandable?
- Whether the topics can be found in the 'help' easily or not?

The charts, such as bar charts, pie charts, scatter plots, and line charts, can be used to depict and assess the satisfaction level of the customer. The satisfaction level of the customer must be continuously monitored over time.

3.4.3 Testing Metrics

Testing metrics are used to capture the progress and level of testing for a given software. The amount of testing done is measured by using the test coverage metrics. These metrics can be used to measure the various levels of coverage, such as statement, path, condition, and branch, and are given below:

1. The percentage of statements covered while testing is defined by statement coverage metric.
2. The percentage of branches covered while testing the source code is defined by branch coverage metric.
3. The percentage of operations covered while testing the source code is defined by operation coverage metric.
4. The percentage of conditions covered (both for true and false) is evaluated using condition coverage metric.
5. The percentage of paths covered in a control flow graph is evaluated using condition coverage metric.
6. The percentage of loops covered while testing a program is evaluated using loop coverage metric.
7. All the possible combinations of conditions are covered by multiple coverage metrics.

NASA developed a test focus (TF) metric defined as the ratio of the amount of effort spent in finding and removing "real" faults in the software to the total number of faults reported in the software. The TF metric is given as (Stark et al. 1992):

$$\text{TF} = \frac{\text{Number of STRs fixed and closed}}{\text{Total number of STRs}}$$

where:
STR is software trouble report

The fault coverage metric (FCM) is given as:

$$\text{FCM} = \frac{\text{Number of faults addressed} \times \text{severity of faults}}{\text{Total number of faults} \times \text{severity of faults}}$$

Some of the basic process metrics used to measure testing are given below:

1. Number of test cases designed
2. Number of test cases executed
3. Number of test cases passed
4. Number of test cases failed
5. Test case execution time
6. Total execution time
7. Time spent for the development of a test case
8. Testing effort
9. Total time spent for the development of test cases

On the basis of above direct measures, the following additional testing-related metrics can be computed to derive more useful information from the basic metrics as given below.

1. Percentage of test cases executed
2. Percentage of test cases passed
3. Percentage of test cases failed
4. Actual execution time of a test case/estimated execution time of a test case
5. Average execution time of a test case

3.5 OO Metrics

Because of growing size and complexity of software systems in the market, OO analysis and design principles are being used by organizations to produce better designed, high–quality, and maintainable software. As the systems are being developed using OO software engineering principles, the need for measuring various OO constructs is increasing.

Features of OO paradigm (programming languages, tools, methods, and processes) provide support for many quality attributes. The key concepts of OO paradigm are: classes, objects, attributes, methods, modularity, encapsulation, inheritance, and polymorphism (Malhotra 2009). An object is made up of three basic components: an identity, a state, and a behavior (Booch 1994). The identity distinguishes two objects with same state and behavior. The state of the object represents the different possible internal conditions that the object may experience during its lifetime. The behavior of the object is the way the object will respond to a set of received messages.

A class is a template consisting of a number of attributes and methods. Every object is the instance of a class. The attributes in a class define the possible states in which an instance of that class may be. The behavior of an object depends on the class methods and the state of the object as methods may respond differently to input messages depending on the current state. Attributes and methods are said to be encapsulated into a single entity. Encapsulation and data hiding are key features of OO languages.

The main advantage of encapsulation is that the values of attributes remain private, unless the methods are written to pass that information outside of the object. The internal working of each object is decoupled from the other parts of the software thus achieving modularity. Once a class has been written and tested, it can be distributed to other programmers for reuse in their own software. This is known as reusability. The objects can be maintained separately leading to easier location and fixation of errors. This process is called maintainability.

The most powerful technique associated to OO methods is the inheritance relationship. If a class B is derived from class A. Class A is said to be a base (or super) class and class B is said to be a derived (or sub) class. A derived class inherits all the behavior of its base class and is allowed to add its own behavior.

Polymorphism (another useful OO concept) describes multiple possible states for a single property. Polymorphism allows programs to be written based only on the abstract interfaces of the objects, which will be manipulated. This means that future extension in the form of new types of objects is easy, if the new objects conform to the original interface.

Nowadays, the software organizations are focusing on software process improvement. This demand led to new/improved approaches in software development area, with perhaps the most promising being the OO approach. The earlier software metrics (Halstead, McCabe, LOCs) were aimed at procedural-oriented languages. The OO paradigm includes new concepts. Therefore, a number of OO metrics to capture the key concepts of OO paradigm have been proposed in literature in the last two decades.

3.5.1 Popular OO Metric Suites

There are a number of OO metric suites proposed in the literature. These metric suites are summarized below. Chidamber and Kemerer (1994) defined a suite of six popular metrics. This suite has received widest attention for predicting quality attributes in literature. The metrics summary along with the construct they are capturing is provided in Table 3.3.

Li and Henry (1993) assessed the Chidamber and Kemerer metrics given in Table 3.3 and provided a metric suite given in Table 3.4.

Bieman and Kang (1995) proposed two cohesion metrics loose class cohesion (LCC) and tight class cohesion (TCC).

Lorenz and Kidd (1994) proposed a suite of 11 metrics. These metrics address size, coupling, inheritance, and so on and are summarized in Table 3.5.

Briand et al. (1997) proposed a suite of 18 coupling metrics. These metrics are summarized in Table 3.6. Similarly, Tegarden et al. (1995) have proposed a large suite of metrics based on variable, object, method and system level. The detailed list can be found in Henderson-Sellers (1996). Lee et al. (1995) has given four metrics, one for measuring cohesion and three metrics for measuring coupling (see Table 3.7).

The system-level polymorphism metrics are measured by Benlarbi and Melo (1999). These metrics are used to measure static and dynamic polymorphism and are summarized in Table 3.8.

TABLE 3.3

Chidamber and Kemerer Metric Suites

Metric	Definition	Construct Being Measured
CBO	It counts the number of other classes to which a class is linked.	Coupling
WMC	It counts the number of methods weighted by complexity in a class.	Size
RFC	It counts the number of external and internal methods in a class.	Coupling
LCOM	Lack of cohesion in methods	Cohesion
NOC	It counts the number of immediate subclasses of a given class.	Inheritance
DIT	It counts the number of steps from the leaf to the root node.	Inheritance

TABLE 3.4

Li and Henry Metric Suites

Metric	Definition	Construct Being Measured
DAC	It counts the number of abstract data types in a class.	Coupling
MPC	It counts a number of unique send statements from a class to another class.	Coupling
NOM	It counts the number of methods in a given class.	Size
SIZE1	It counts the number of semicolons.	Size
SIZE2	It is the sum of number of attributes and methods in a class.	Size

TABLE 3.5

Lorenz and Kidd Metric Suites for measuring Inheritance

Metric	Definition
NOP	It counts the number of immediate parents of a given class.
NOD	It counts the number of indirect and direct subclasses of a given class.
NMO	It counts the number of methods overridden in a class.
NMI	It counts the number of methods inherited in a class.
NMA	It counts the number of new methods added in a class.
SIX	Specialization index

TABLE 3.6

Briand et al. Metric Suites

IFCAIC	These coupling metrics count the number of interactions between classes.
ACAIC OCAIC	These metrics distinguish the relationship between the classes (friendship, inheritance, none), different types of interactions, and the locus of impact of the interaction.
FCAEC DCAEC OCAEC IFCMIC ACMIC DCMIC FCMEC DCMEC OCMEC IFMMIC AMMIC OMMIC FMMEC DMMEC OMMEC	The acronyms for the metrics indicates what interactions are counted: • The first or first two characters indicate the type of coupling relationship between classes: A: ancestor, D: descendents, F: friend classes, IF: inverse friends (classes that declare a given class A as their friend), O: others, implies none of the other relationships. • The next two characters indicate the type of interaction: CA: There is a class–attribute interaction if class x has an attribute of type class y. CM: There is a class–method interaction if class x consist of a method that has parameter of type class y. MM: There is a method–method interaction if class x calls method of another class y, or class x has a method of class y as a parameter. • The last two characters indicate the locus of impact: IC: Import coupling, counts the number of other classes called by class x. EC: Export coupling, count number of other classes using class y.

TABLE 3.7

Lee et al. Metric Suites

Metric	Definition	Construct Being Measured
ICP	Information flow-based coupling	Coupling
IHICP	Information flow-based inheritance coupling	Coupling
NIHICP	Information flow-based noninheritance coupling	Coupling
ICH	Information-based cohesion	Cohesion

Yap and Henderson-Sellers (1993) have proposed a suite of metrics to measure cohesion and reuse in OO systems. Aggarwal et al. (2005) defined two reusability metrics namely function template factor (FTF) and class template factor (CTF) that are used to measure reuse in OO systems. The relevant metrics summarized in tables are explained in subsequent sections.

TABLE 3.8

Benlarbi and Melo Polymorphism Metrics

Metric	Definition
SPA	It measures static polymorphism in ancestors.
DPA	It measures dynamic polymorphism in ancestors.
SP	It is the sum of SPA and SPD metrics.
DP	It is the sum of DPA and DPD metrics.
NIP	It measures polymorphism in noninheritance relations.
OVO	It measures overloading in stand-alone classes.
SPD	It measures static polymorphism in descendants.
DPD	It measures dynamic polymorphism in descendants.

3.5.2 Coupling Metrics

Coupling is defined as the degree of interdependence between modules or classes. It is measured by counting the number of other classes called by a class, during the software analysis and design phases. It increases complexity and decreases maintainability, reusability, and understandability of the system. Thus, interclass coupling must be kept to a minimum. Coupling also increases amount of testing effort required to test classes (Henderson-Sellers 1996). Thus, the aim of the developer should be to keep coupling between two classes as minimum as possible.

Information flow metrics represent the amount of coupling in the classes. Fan-in and fan-out metrics indicate the number of classes collaborating with the other classes:

1. Fan-in: It counts the number of other classes calling class X.
2. Fan-out: It counts the number of classes called by class X.

Figure 3.4 depicts the values for fan-in and fan-out metrics for classes A, B, C, D, E, and F of an example system. The values of fan-out should be as low as possible because of the fact that it increases complexity and maintainability of the software.

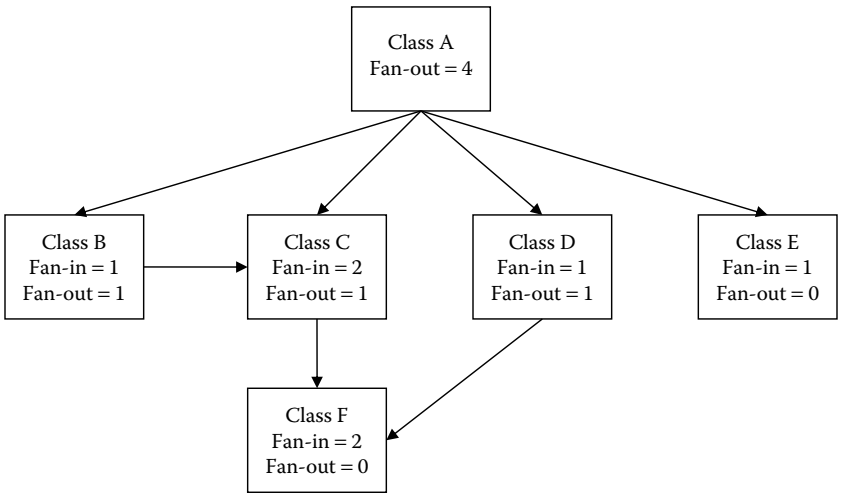

FIGURE 3.4
Fan-in and fan-out metrics

Chidamber and Kemerer (1994) defined coupling as:

> Two classes are coupled when methods declared in one class use methods or instance variables of the other classes.

This definition also includes coupling based on inheritance. Chidamber and Kemerer (1994) defined coupling between objects (CBO) as "the count of number of other classes to which a class is coupled." The CBO definition given in 1994 includes inheritance-based coupling. For example, consider Figure 3.5, three variables of other classes (class B, class C, and class D) are used in class A, hence, the value of CBO for class A is 3. Similarly, classes D, F, G, and H have the value of CBO metric as zero.

Li and Henry (1993) used data abstraction technique for defining coupling. Data abstraction provides the ability to create user-defined data types called abstract data types (ADTs). Li and Henry defined data abstraction coupling (DAC) as:

$$\text{DAC} = \text{number of ADTs defined in a class}$$

In Figure 3.5, class A has three ADTs (i.e., three nonsimple attributes). Li and Henry defined another coupling metric known as message passing coupling (MPC) as "number of unique send statements in a class." Hence, if three different methods in class B access the same method in class A, then MPC is 3 for class B, as shown in Figure 3.6.

Chidamber and Kemerer (1994) defined response for a class (RFC) metric as a set of methods defined in a class and called by a class. It is given by RFC = |RS|, where RS, the response set of the class, is given by:

$$\text{RS} = I_i \cup all\ j\{E_{ij}\}$$

where:

I_i = set of all methods in a class (total i)

$R_i = \{R_{ij}\}$ = set of methods called by M_i

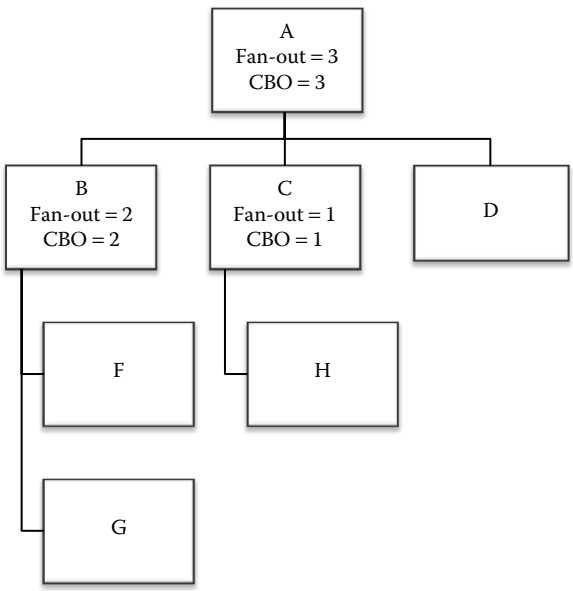

FIGURE 3.5
Values of CBO metric for a small program.

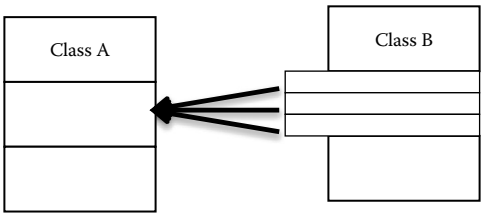

FIGURE 3.6
Example of MPC metric.

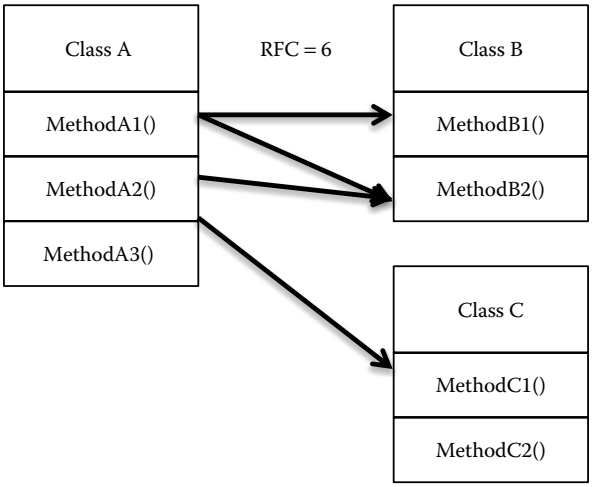

FIGURE 3.7
Example of RFC metric.

For example, in Figure 3.7, RFC value for class A is 6, as class A has three methods of its own and calls 2 other methods of class B and one of class C.

A number of coupling metrics with respect to OO software have been proposed by Briand et al. (1997). These metrics take into account the different OO design mechanisms provided by the C++ language: friendship, classes, specialization, and aggregation. These metrics may be used to guide software developers about which type of coupling affects the maintenance cost and reduces reusability. Briand et al. (1997) observed that the coupling between classes could be divided into different facets:

1. Relationship: It signifies the type of relationship between classes—friendship, inheritance, or other.
2. Export or import coupling (EC/IC): It determines the number of classes calling class A (export) and the number of classes called by class A (import).
3. Type of interaction: There are three types of interactions between classes—class–attribute (CA), class–method (CM), and method–method (MM).
   i. CA interaction: If there are nonsimple attributes declared in a class, the type of interaction is CA. For example, consider Figure 3.8, there are two nonsimple

attributes in class A, B1 of type class B and C1 of type class C. Hence, any changes in class B or class C may affect class A.

ii. CM interaction: If the object of class A is passed as parameter to method of class B, then the type of interaction is said to be CM. For example, as shown in Figure 3.8, object of class B, B1, is passed as parameter to method M1 of class A, thus the interaction is of CM type.

iii. MM interaction: If a method M_i of class K_i calls method M_j of class K_j or if the reference of method M_i of class K_i is passed as an argument to method M_j of class K_j, then there is MM type of interaction between class K_i and class K_j. For example, as shown in Figure 3.8, the method M2 of class B calls method M1 of class A, hence, there is a MM interaction between class B and class A. Similarly, method B1 of class B type is passed as reference to method M3 of class C.

The metrics for CM interaction type are IFCMIC, ACMIC, OCMIC, FCMEC, DCMEC, and OCMEC. In these metrics, the first one/two letters denote the type of relationship (IF denotes inverse friendship, A denotes ancestors, D denotes descendant, F denotes friendship, and O denotes others). The next two letters denote the type of interaction (CA, CM, MM) between classes. Finally, the last two letters denote the type of coupling (IC or EC).

Lee et al. (1995) acknowledged the need to differentiate between inheritance-based and noninheritance-based coupling by proposing the corresponding measures: noninheritance information flow-based coupling (NIH-ICP) and information flow-based inheritance coupling (IH-ICP). Information flow-based coupling (ICP) metric is defined as the sum of NIH-ICP and IH-ICP metrics and is based on method invocations, taking polymorphism into account.

```
class A
{
B B1; // Nonsimple attributes
C C1;
public:
void M1(B B1)
{
}
};
class B
{
public:
void M2()
{
A A1;
A1.M1();// Method of class A called
}
};
class C
{
void M3(B::B1) //Method of class B passed as parameter
{
}
};
```

FIGURE 3.8
Example for computing type of interaction.

3.5.3 Cohesion Metrics

Cohesion is a measure of the degree to which the elements of a module are functionally related to each other. The cohesion measure requires information about attribute usage and method invocations within a class. A class that is less cohesive is more complex and is likely to contain more number of faults in the software development life cycle. Chidamber and Kemerer (1994) proposed lack of cohesion in methods (LCOM) metric in 1994. The LCOM metric is used to measure the dissimilarity of methods in a class by taking into account the attributes commonly used by the methods.

The LCOM metric calculates the difference between the number of methods that have similarity zero and the number of methods that have similarly greater than zero. In LCOM, similarity represents whether there is common attribute usage in pair of methods or not. The greater the similarly between methods, the more is the cohesiveness of the class. For example, consider a class consisting of four attributes (A1, A2, A3, and A4). The method usage of the class is given in Figure 3.9.

There are few problems related to LCOM metric, proposed by Chidamber and Kemerer (1994), which were addressed by Henderson-Sellers (1996) as given below:

1. The value of LCOM metric was zero in a number of real examples because of the presence of dissimilarity among methods. Hence, although a high value of LCOM metric suggests low cohesion, the zero value does not essentially suggest high cohesion.
2. Chidamber and Kemerer (1994) gave no guideline for interpretation of value of LCOM. Thus, Henderson-Sellers (1996) revised the LCOM value. Consider m methods accessing a set of attributes Di ($i = 1,\ldots,n$). Let $\mu(D_i)$ be the number of methods that access each datum. The revised LCOM1 metric is given as follows:

$$\text{LCOM1} = \frac{(1/N)\sum_{i=1}^{n}\mu(D_i) - m}{1 - m}$$

$M1 = \{A1, A2, A3, A4\}$
$M2 = \{A1, A2\}$
$M3 = \{A3\}$
$M4 = \{A3, A4\}$
$M5 = \{A2\}$
$M1 \cap M2 = 1$
$M1 \cap M3 = 1$
$M1 \cap M4 = 1$
$M1 \cap M5 = 1$
$M2 \cap M3 = 0$
$M2 \cap M4 = 0$
$M2 \cap M5 = 1$
$M3 \cap M4 = 1$
$M3 \cap M5 = 0$
$M4 \cap M5 = 0$
$\text{LCOM} = 4 - 6, \text{ Hence, } \text{LCOM} = 0$

FIGURE 3.9
Example of LCOM metric.

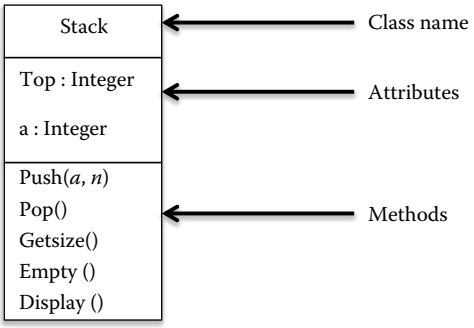

FIGURE 3.10
Stack class.

The approach by Bieman and Kang (1995) to measure cohesion was based on that of Chidamber and Kemerer (1994). They proposed two cohesion measures—TCC and LCC. TCC metric is defined as the percentage of pairs of directly connected public methods of the class with common attribute usage. LCC is the same as TCC, except that it also considers indirectly connected methods. A method M1 is indirectly connected with method M3, if method M1 is connected to method M2 and method M2 is connected to method M3. Hence, transitive closure of directly connected methods is represented by indirectly connected methods. Consider the class stack shown in Figure 3.10.

Figure 3.11 shows the attribute usage of methods. The pair of public functions with common attribute usage is given below:

{(empty, push), (empty, pop), (empty, display), (getsize, push), (getsize, pop), (push, pop), (push, display), (pop, display)}

Thus, TCC for stack class is as given below:

$$\mathrm{TCC}(\mathrm{Stack}) = \frac{8}{10} \times 100 = 80\%$$

The methods "empty" and "getsize" are indirectly connected, since "empty" is connected to "push" and "getsize" is also connected to "push." Thus, by transitivity, "empty" is connected to "getsize." Similarly "getsize" is indirectly connected to "display."
LCC for stack class is as given below:

$$\mathrm{LCC}(\mathrm{Stack}) = \frac{10}{10} \times 100 = 100\%$$

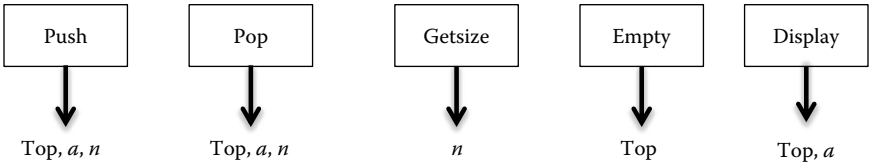

FIGURE 3.11
Attribute usage of methods of class stack.

Lee et al. (1995) proposed information flow-based cohesion (ICH) metric. ICH for a class is defined as the weighted sum of the number of invocations of other methods of the same class, weighted by the number of parameters of the invoked method.

3.5.4 Inheritance Metrics

The inheritance represents parent–child relationship and is measured in terms of number of subclasses, base classes, and depth of inheritance hierarchy by many authors in the literature. Inheritance represents form of reusability. Chidamber and Kemerer (1994) defined depth of inheritance tree (DIT) metric as maximum number of steps from class to root node in a tree. Thus, in case concerning multiple inheritance, the DIT will be counted as the maximum length from the class to the root of the tree. Consider Figure 3.12, DIT for class D and class F is 2.

The average inheritance depth (AID) is calculated as (Yap and Henderson-Sellers 1993):

$$\text{AID} = \frac{\sum \text{depth of each class}}{\text{Total number of classes}}$$

In Figure 3.11, the depth of subclass D is 2 ([2 + 2]/2).

The AID of overall inheritance structure is: 0(A) + 1(B) + 1(C) + 2(D) + 0(E) + 1.5(F) + 0(G) = 5.5. Finally, dividing by total number of classes we get 5.5/6 = 0.92.

Chidamber and Kemerer (1994) yet proposed another metric, number of children (NOC), which counts the number of immediate subclasses of a given class in an inheritance hierarchy. A class with more NOC requires more testing. In Figure 3.12, class B has 1 and class C has 2 subclasses. Lorenz and Kidd (1994) proposed number of parents (NOP) metric that counts the number of direct parent classes for a given class in inheritance hierarchy. For example, class D has NOP value of 2. Similarly, Lorenz and Kidd (1994) also developed number of descendants (NOD) metric. The NOD metric defines the number of direct and indirect subclasses of a class. In Figure 3.12, class E has NOD value of 3 (C, D, and F). Tegarden et al. (1992) define number of ancestors (NA) as the number of indirect and direct parent classes of a given class. Hence, as given in Figure 3.12, NA(D) = 4 (A, B, C, and E).

Other inheritance metrics defined by Lorenz and Kidd include the number of methods added (NMA), number of methods overridden (NMO), and number of methods inherited (NMI). NMO counts number of methods in a class with same name and signature as in its

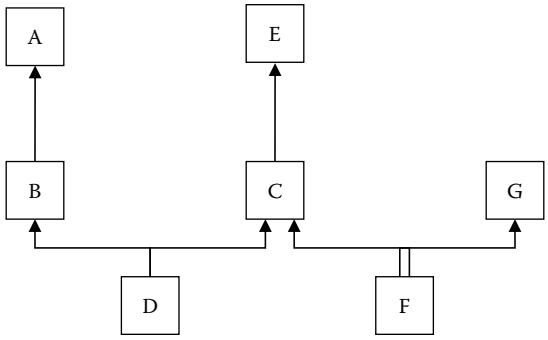

FIGURE 3.12
Inheritance hierarchy.

parent class. NMA counts the number of new methods (neither overridden nor inherited) added in a class. NMI counts number of methods inherited by a class from its parent class. Finally, Lorenz and Kidd (1994) defined specialization index (SIX) using DIT, NMO, NMA, and NMI metrics as given below:

$$\text{SIX} = \frac{\text{NMO} \times \text{DIT}}{\text{NMO} + \text{NMA} + \text{NMI}}$$

Consider the class diagram given in Figure 3.13. The class employee inherits class person. The class employee overrides two functions, addDetails() and display(). Thus, the value of NMO metric for class student is 2. Two new methods is added in this class (getSalary() and compSalary()). Hence, the value of NMA metric is 2.

Thus, for class Employee, the value of NMO is 2, NMA is 2, and NMI is 1 (getEmail()). For the class Employee, the value of SIX is:

$$\text{SIX} = \frac{2 \times 1}{2 + 2 + 1} = \frac{2}{5} = 0.4$$

The maximum number of levels in the inheritance hierarchy that are below the class are measured through class to leaf depth (CLD). The value of CLD for class Person is 1.

3.5.5 Reuse Metrics

An OO development environment supports design and code reuse, the most straightforward type of reuse being the use of a library class (of code), which perfectly suits the

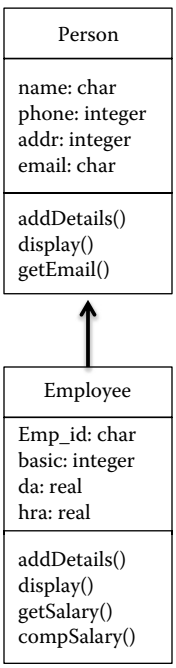

FIGURE 3.13
Example of inheritance relationship.

requirements. Yap and Henderson-Sellers (1993) discuss two measures designed to evaluate the level of reuse possible within classes. The reuse ratio (U) is defined as:

$$\mathrm{U} = \frac{\text{Number of superclasses}}{\text{Total number of classes}}$$

Consider Figure 3.13, the value of U is 1/2. Another metric is specialization ratio (S), and is given as:

$$\mathrm{S} = \frac{\text{Number of subclasses}}{\text{Number of superclasses}}$$

In Figure 3.13, Employee is the subclass and Person is the parent class. Thus, S = 1.

Aggarwal et al. (2005) proposed another set of metrics for measuring reuse by using generic programming in the form of templates. The metric FTF is defined as ratio of number of functions using function templates to total number of functions as shown below:

$$\mathrm{FTF} = \frac{\text{Number of functions using function templates}}{\text{Total number of functions}}$$

Consider a system with methods $F_1,\ldots,F_n$. Then,

$$\mathrm{FTF} = \frac{\sum_{i-1}^{n} uses\_FT\left(F_i\right)}{\sum_{i-1}^{n} F}$$

where:

$$uses\_FT\left(F_i\right)\begin{cases}1, \text{ iff function uses function template}\\ 0, \text{ otherwise}\end{cases}$$

In Figure 3.14, the value of metric FTF = (1/3).

The metric CTF is defined as the ratio of number of classes using class templates to total number of classes as shown below:

$$\mathrm{CTF} = \frac{\text{Number of classes using class templates}}{\text{Total number of classes}}$$

```
void method1(){
.....}
template<class U>
void method2(U &a, U &b){
.....}
void method3(){
.....}
```

FIGURE 3.14
Source code for calculation of FTF metric.

```
class X{
.....};
template<class U, int size>
class Y{
U ar1[size];
....};
```

FIGURE 3.15
Source code for calculating metric CTF.

Consider a system with classes $C_1,\ldots,C_n$. Then,

$$\text{CTF} = \frac{\sum_{i-1}^{n} uses\_CT(C_i)}{\sum_{i-1}^{n} C_i}$$

where:

$$uses\_CT(C_i)\begin{cases}1 \text{ iff class uses class template} \\ 0 \text{ otherwise}\end{cases}$$

In Figure 3.15, the value of metric CTF $= \frac{1}{2}$.

3.5.6 Size Metrics

There are various conventional metrics applicable to OO systems. The traditional LOC metric measures the size of a class (refer Section 3.3). However, the OO paradigm defines many concepts that require additional metrics that can measure them. Keeping this in view, many OO metrics have been proposed in the literature. Chidamber and Kemerer (1994) developed weighted methods per class (WMC) metric as count of number of methods weighted by complexities and is given as:

$$\text{WMC} = \sum_{i=1}^{n} C_i$$

where:
$M_1,\ldots M_n$ are methods defined in class K_1
$C_1,\ldots C_n$ are the complexities of the methods

Lorenz and Kidd defined number of attributes (NOA) metric given as the sum of number of instance variables and number of class variables. Li and Henry (1993) defined number of methods (NOM) as the number of local methods defined in a given class. They also defined two other size metrics—namely, SIZE1 and SIZE2. These metrics are defined below:

SIZE1 = number of semicolons in a class
SIZE2 = sum of NOA and NOM

3.6 Dynamic Software Metrics

The dynamic behavior of the software is captured though dynamic metrics. The dynamic metrics related to coupling, cohesion, and complexity have been proposed in literature. The difference between static and dynamic metrics is presented in Table 3.9 (Chhabra and Gupta 2010).

3.6.1 Dynamic Coupling Metrics

Yacoub et al. (1999) developed a set of metrics for measuring dynamic coupling—namely, export object coupling (EOC) and import object coupling (IOC). These metrics are based on executable code. EOC metric calculates the percentage of ratio of number of messages sent from one object o_1 to another object o_2 to the total number of messages exchanged between o_1 and o_2 during the execution of a scenario. IOC metric calculates percentage of ratio of number of messages sent from object o_2 to o_1 to the total number of messages exchanged between o_1 and o_2 during the execution of a scenario. For example, four messages are sent from object o_1 to object o_2 and three messages are sent from object o_2 to object o_1, $EOC(o_1) = 4/7 \times 100$ and $IOC(o_1) = 3/7 \times 100$.

Arisholm et al. (2004) proposed a suite of 12 dynamic IC and EC metrics. There are six metrics defined at object level and six defined at class level. The first two letters of the metric describe the type of coupling—import or export. The EC represents the number of other classes calling a given class. IC represents number of other classes called by a class. The third letter signifies object or class, and the last letter signifies the strength of coupling (D—dynamic messages, M—distinct method, C—distinct classes). Mitchell and Power developed dynamic coupling metric suite summarized in Table 3.10.

3.6.2 Dynamic Cohesion Metrics

Mitchell and Power (2003, 2004) proposed extension of Chidamber and Kemerer's LCOM metric as dynamic LCOM. They proposed two variations of LCOM metric: runtime simple LCOM (RLCOM) and runtime call-weighted LCOM (RWLCOM). RLCOM considers instance variables accessed at runtime. RWLCOM assigns weights to each instance variable by the number of times it is accessed at runtime.

TABLE 3.9

Difference between static and dynamic metrics

S. No.	Static Metrics	Dynamic Metrics
1	Collected without execution of the program	Collected at runtime execution
2	Easy to collect	Difficult to collect
3	Available in the early phases of software development	Available in later phases of software development
4	Less accurate as compared to dynamic metrics	More accurate
5	Inefficient in dealing with dead code and OO concepts such as polymorphism and dynamic binding	Efficient in dealing with all OO concepts

TABLE 3.10

Mitchell and Power Dynamic Coupling Metric Suite

Metric	Definition
Dynamic coupling between objects	This metric is same as Chidamber and Kemerer's CBO metric, but defined at runtime.
Degree of dynamic coupling between two classes at runtime	It is the percentage of ratio of number of times a class A accesses the methods or instance variables of another class B to the total no of accesses of class A.
Degree of dynamic coupling within a given set of classes	The metric extends the concept given by the above metric to indicate the level of dynamic coupling within a given set of classes.
Runtime import coupling between objects	Number of classes assessed by a given class at runtime.
Runtime export coupling between objects	Number of classes that access a given class at runtime.
Runtime import degree of coupling	Ratio of number of classes assessed by a given class at runtime to the total number of accesses made.
Runtime export degree of coupling	Ratio of number of classes that access a given class at runtime to the total number of accesses made.

3.6.3 Dynamic Complexity Metrics

Determining the complexity of a program is important to analyze its testability and maintainability. The complexity of the program may depend on the execution environment. Munson and Khoshgoftar (1993) proposed dynamic complexity metric.

3.7 System Evolution and Evolutionary Metrics

Software evolution aims at incorporating and revalidating the probable significant changes to the system without being able to predict a priori how user requirements will evolve. The current system release or version can never be said to be complete and continues to evolve. As it evolves, the complexity of the system will grow unless there is a better solution available to solve these issues.

The main objectives of software evolution are ensuring the reliability and flexibility of the system. During the past 20 years, the life span of a system could be on average 6–10 years. However, it was recently found that a system should be evolved once every few months to ensure it is adapted to the real-world environment. This is because of the rapid growth of World Wide Web and Internet resources that make it easier for users to find related information.

The idea of software evolution leads to open source development as anybody could download the source codes and, hence, modify it. The positive impact in this case is that a number of new ideas would be discovered and generated that aims to improve the quality of system with a variety of choices. However, the negative impact is that there is no copyright if a software product has been published as open source.

Over time, software systems, programs as well as applications, continue to develop. These changes will require new laws and theories to be created and justified. Some models would also require additional aspects in developing future programs. Innovations,

improvements, and additions lead to unexpected form of software effort in maintenance phase. The maintenance issues also would probably change to adapt to the evolution of the future software.

Software process and development are an ongoing process that has a never-ending cycle. After going through learning and refinements, it is always an arguable issue when it comes to efficiency and effectiveness of the programs.

A software system may be analyzed by the following evolutionary and change metrics (suggested by Moser et al. 2008), which may prove helpful in understanding the evolution and release history of a software system.

3.7.1 Revisions, Refactorings, and Bug-Fixes

The metrics related to refactoring and bug-fixes are defined below:

- *Revisions:* Number of revisions of a software repository file
- *Refactorings:* Number of times a software repository file has been refactored
- *Bug-fixes:* Number of times a file has been associated with bug-fixing
- *Authors:* Number of distinct or different authors who have committed or checked in a software repository

3.7.2 LOC Based

The LOC-based evolution metrics are described as:

- *LOC added:* Sum total of all the lines of code added to a file for all of its revisions in the repository
- *Max LOC added:* Maximum number of lines of code added to a file for all of its revisions in the repository
- *Average LOC added:* Average number of lines of code added to a file for all of its revisions in the repository
- *LOC deleted:* Sum total of all the lines of code deleted from a file for all of its revisions in the repository
- *Max LOC deleted:* Maximum number of lines of code deleted from a file for all of its revisions in the repository
- *Average LOC deleted:* Average number of lines of code deleted from a file for all of its revisions in the repository

3.7.3 Code Churn Based

- *Code churn:* Sum total of (difference between added lines of code and deleted lines of code) for a file, considering all of its revisions in the repository
- *Max code churn:* Maximum code churn for all of the revisions of a file in the repository
- *Average code churn:* Average code churn for all of the revisions of a file in the repository

3.7.4 Miscellaneous

The other related evolution metrics are:

- *Max change set:* Maximum number of files that are committed or checked in together in a repository
- *Average change set:* Average number of files that are committed or checked in together in a repository
- *Age:* Age of repository file, measured in weeks by counting backward from a given release of a software system
- *Weighted Age:* Weighted Age of a repository file is given as:

$$\frac{\sum_{i=1}^{N} \text{Age}(i) \times \text{LOC added}(i)}{\sum \text{LOC added}(i)}$$

where:

i is a revision of a repository file and N is the total number of revisions for that file

3.8 Validation of Metrics

Several researchers recommend properties that software metrics should posses to increase their usefulness. For instance, Basili and Reiter suggest that metrics should be sensitive to externally observable differences in the development environment, and must correspond to notions about the differences between the software artifacts being measured (Basili and Reiter 1979). However, most recommended properties tend to be informal in the evaluation of metrics. It is always desirable to have a formal set of criteria with which the proposed metrics can be evaluated. Weyuker (1998) has developed a formal list of properties for software metrics and has evaluated number of existing software metrics against these properties. Although many authors (Zuse 1991, Briand et al. 1999b) have criticized this approach, it is still a widely known formal, analytical approach.

Weyuker's (1988) first four properties address how sensitive and discriminative the metric is. The fifth property requires that when two classes are combined their metric value should be greater than the metric value of each individual class. The sixth property addresses the interaction between two programs/classes. It implies that the interaction between program/class A and program/class B is different than the interaction between program/class C and program/class B given that the interaction between program/class A and program/class C. The seventh property requires that a measure be sensitive to statement order within a program/class. The eighth property requires that renaming of variables does not affect the value of a measure. Last property states that the sum of the metric values of a program/class could be less than the metric value of the program/class when considered as a whole (Henderson-Sellers 1996). The applicability of only the properties for OO metrics are given below:

Let u be the metric of program/class P and Q

Property 1: This property states that

$$(\exists P),(\exists Q)\left[u(p) \neq u(Q)\right]$$

It ensures that no measure rates all program/class to be of same metric value.

Property 2: Let c be a nonnegative number. Then, there are finite numbers of program/class with metric c. This property ensures that there is sufficient resolution in the measurement scale to be useful.

Property 3: There are distinct program/class P and Q such that $u(p) = u(Q)\cdot$

Property 4: For OO system, two programs/classes having the same functionality could have different values.

$$(\exists P)(\exists Q)\left[P \equiv Q \text{ and } u(P) \neq (Q)\right]$$

Property 5: When two programs/classes are concatenated, their metric should be greater than the metrics of each of the parts.

$$(\forall P)(\forall Q)\left[u(P) \leq u(P+Q) \text{ and } u(Q) \leq u(P+Q)\right]$$

Property 6: This property suggests nonequivalence of interaction. If there are two program/class bodies of equal metric value which, when separately concatenated to a same third program/class, yield program/class of different metric value.

For program/class P, Q, R

$$(\exists P)(\exists Q)(\exists R)\left[u(P) = u(Q) \text{ and } u(P+R) \neq u(Q+R)\right]$$

Property 7: This property is not applicable for OO metrics (Chidamber and Kemerer 1994).

Property 8: It specifies that "if P is a renaming of Q, then $u(P) = u(Q)$."

Property 9: This property is not applicable for OO metrics (Chidamber and Kemerer 1994).

3.9 Practical Relevance

Empirical assessment of software metrics is important to ensure their practical relevance in the software organizations. Such analysis is of high practical relevance and especially beneficial for large-scale systems, where the experts need to focus their attention and resources to problem areas in the system under development. In the subsequent section, we describe the role of metrics in research and industry. We also provide the approach for calculating metric thresholds.

3.9.1 Designing a Good Quality System

During the entire life cycle of a project, it is very important to maintain the quality and to ensure that it does not deteriorate as a project progresses through its life cycle. Thus, the project manager must monitor quality of the system on a continuous basis. To plan

and control quality, it is very important to understand how the quality can be measured. Software metrics are widely used for measuring, monitoring, and evaluating the quality of a project. Various software metrics have been proposed in the literature to assess the software quality attributes such as change proneness, fault proneness, maintainability of a class or module, and so on. A large portion of empirical research has been involved with the development and evaluation of the quality models for procedural and OO software.

Software metrics have found a wide range of applications in various fields of software engineering. As discussed, some of the familiar and common uses of software metrics are scheduling the time required by a project, estimating the budget or cost of a project, estimating the size of the project, and so on. These parameters can be estimated at the early phases of software development life cycle, and thus help software managers to make judicious allocation of resources. For example, once the schedule and budget has been decided upon, managers can plan in advance the amount of person-hours (effort) required. Besides this, the design of software can be assessed in the industry by identifying the out of range values of the software metrics. One way to improve the quality of the system is to relate structural attribute measures intended to quantify important concepts of a given software, such as the following:

- Encapsulation
- Coupling
- Cohesion
- Inheritance
- Polymorphism

to external quality attributes such as the following:

- Fault proneness
- Maintainability
- Testing effort
- Rework effort
- Reusability
- Development effort

The ability to assess quality of software in the early phases of the software life cycle is the main aim of researchers so that structural attribute measures can be used for predicting external attribute measures. This would greatly facilitate technology assessment and comparisons.

Researchers are working hard to investigate the properties of software measures to understand the effectiveness and applicability of the underlying measures. Hence, we need to understand what these measures are really capturing, whether they are really different, and whether they are useful indicators of quality attributes of interest? This will build a body of evidence, and present commonalities and differences across various studies. Finally, these empirical studies will contribute largely in building good quality systems.

3.9.2 Which Software Metrics to Select?

The selection of software metrics (independent variables) in the research is a crucial decision. The researcher must first decide on the domain of the metrics. After deciding the domain, the researcher must decide the attributes to capture in the domain. Then,

the popular and widely used software metrics suite available to measure the constructs is identified from the literature. Finally, a decision on the selection must be made on software metrics. The criterion that can be used to select software metrics is that the selected software metrics must capture all the constructs, be widely used in the literature, easily understood, fast to compute, and computationally less expensive. The choice of metric suite heavily depends on the goals of the research. For instance, in quality model prediction, OO metrics proposed by Chidamber and Kemerer (1994) are widely used in the empirical studies.

In cases where multiple software metrics are used, the attribute reduction techniques given in Section 6.2 must be applied to reduce them, if model prediction is being conducted.

3.9.3 Computing Thresholds

As seen in previous sections, there are a number of metrics proposed and there are numerous tools to measure them (see Section 5.8.3). Metrics are widely used in the field of software engineering to identify problematic parts of the software that need focused and careful attention. A researcher can also keep a track of the metric values, which will allow to identify benchmarks across organizations. The products can be compared or rated, which will allow to assess their quality. In addition to this, threshold values can be defined for the metrics, which will allow the metrics to be used for decision making. Bender (1999) defined threshold as "Breakpoints that are used to identify the acceptable risk in classes." In other words, a threshold can be defined as a benchmark or an upper bound such that the values greater than a threshold value are considered to be problematic, whereas the values lower are considered to be acceptable.

During the initial years, many authors have derived threshold values based on their experience and, thus, those values are not universally accepted. For example, McCabe (1976) defined a value of 10 as threshold for the cyclomatic complexity metric. Similarly, for the maintainability index metric, 65 and 85 are defined as thresholds (Coleman et al. 1995). Since these values are based on intuition or experience, it is not possible to generalize results using these values. Besides the thresholds based on intuition, some authors defined thresholds using mean (μ) and standard deviation (σ). For example, Erni and Lewerentz (1996) defined the minimum and maximum values of threshold as $T = \mu + \sigma$ and $T = \mu - \sigma$, respectively. However, this methodology did not gain popularity as it used the assumption that the metrics should be normally distributed, which is not applicable always. French (1999) used Chebyshev's inequality theorem (not restricted to normal distribution) in addition to mean (μ) and standard deviation (σ) to derive threshold values. According to French, a threshold can be defined as $T = \mu + k \times \sigma$ (k = number of standard deviations). However, this methodology was also not used much as it was restricted to only two-tailed symmetric distributions, which is not justified.

A statistical model (based on logistic regression) to calculate the threshold values was suggested by Ulm (1991). Benlarbi et al. (2000) and El Emam et al. (2000b) estimated the threshold values of a number of OO metrics using this model. However, they found that there was no statistical difference between the two models: the model built using the thresholds and the model built without using the thresholds. Bender (1999) working in the epidemiological field found that the proposed threshold model by Ulm (1991) has some drawbacks. The model assumed that the probability of fault in a class is constant when a metric value is below the threshold, and the fault probability increases according to the logistic function, otherwise. Bender (1999) redefined the threshold effects as an acceptable risk level. The proposed threshold methodology was recently used by Shatnawi (2010)

to identify the threshold values of various OO metrics. Besides this, Shatnawi et al. (2010) also investigated the use of receiver operating characteristics (ROCs) method to identify threshold values. The detailed explanation of the above two methodologies is provided in the below sub sections (Shatnawi 2006). Malhotra and Bansal (2014a), evaluated the threshold approach proposed by Bender (1999) for fault prediction.

3.9.3.1 Statistical Model to Compute Threshold

The Bender (1999) method known as value of an acceptable risk level (VARL) is used to compute the threshold values, where the acceptable risk level is given by a probability Po (e.g., Po = 0.05 or 0.01). For the classes with metrics values below VARL, the risk of a fault occurrence is lower than the probability (Po). In other words, Bender (1999) has suggested that the value of Po can be any probability, which can be considered as the acceptable risk level.

The detailed description of VARL is given by the formula for VARL as follows (Bender 1999):

$$\text{VARL} = \frac{1}{\beta}\left[\ln\left(\frac{\text{Po}}{1-\text{Po}}\right) - \alpha\right]$$

where:

α is a constant
β is the estimated coefficient
Po is the acceptable risk level

In this formula, α and β are obtained using the standard logistic regression formula (refer Section 7.2.1). This formula will be used for each metric individually to find its threshold value.

For example, consider the following data set (Table A.8 in Appendix I) consisting of the metrics (independent variables): LOC, DIT, NOC, CBO, LCOM, WMC, and RFC. The dependent variable is fault proneness. We calculate the threshold values of all the metrics using the following steps:

Step 1: Apply univariate logistic regression to identify significant metrics.

The formula for univariate logistic regression is:

$$P = \frac{e^{g(x)}}{1 + e^{g(x)}}$$

where:

$$g(x) = \alpha + \beta x$$

where:

x is the independent variable, that is, an OO metric
α is the Y-intercept or constant
β is the slope or estimated coefficient

Table 3.11 shows the statistical significance (sig.) for each metric. The "sig." parameter provides the association between each metric and fault proneness. If the "sig."

TABLE 3.11

Statistical Significance of Metrics

Metric	Significance
WMC	**0.013**
CBO	**0.01**
RFC	**0.003**
LOC	**0.001**
DIT	0.296
NOC	0.779
LCOM	**0.026**

value is below or at the significance threshold of 0.05, then the metric is said to be significant in predicting fault proneness (shown in bold). Only for significant metrics, we calculate the threshold values. It can be observed from Table 3.11 that DIT and NOC metrics are insignificant, and thus are not considered for further analysis.

Step 2: Calculate the values of constant and coefficient for significant metrics.

For significant metrics, the values of constant (α) and coefficient (β) using univariate logistic regression are calculated. These values of constant and coefficient will be used in the computation of threshold values. The coefficient shows the impact of the independent variable, and its sign shows whether the impact is positive or negative. Table 3.12 shows the values of constant (α) and coefficient (β) of all the significant metrics.

Step 3: Computation of threshold values.

We have calculated the threshold values (VARL) for the metrics that are found to be significant using the formula given above. The VARL values are calculated for different values of Po, that is, at different levels of risks (between Po = 0.01 and Po = 0.1). The threshold values at different values of Po (0.01, 0.05, 0.08, and 0.1) for all the significant metrics are shown in Table 3.13. It can be observed that the threshold values of all the metrics change significantly as Po changes. This shows that Po plays a significant role in calculating threshold values. Table 3.13 shows that at risk level 0.01 and 0.05, VARL values are out of range (i.e., negative values) for all of the metrics. At Po = 0.1, the threshold values are within the observation range of all the metrics. Hence, in this example, we say that Po = 0.1 is the appropriate risk level and the threshold values (at Po = 0.1) of WMC, CBO, RFC, LOC, and LOCM are 17.99, 14.46, 52.37, 423.44, and 176.94, respectively.

TABLE 3.12

Constant (α) and Coefficient (β) of Significant Metrics

Metric	Coefficient (β)	Constant (α)
WMC	0.06	−2.034
CBO	0.114	−2.603
RFC	0.032	−2.629
LOC	0.004	−2.648
LCOM	0.004	−1.662

TABLE 3.13

Threshold Values on the basis of Logistic Regression Method

Metrics	VARL at 0.01	VARL at 0.05	VARL at 0.08	VARL at 0.1
WMC	−42.69	−15.17	−6.81	17.99
CBO	−17.48	−2.99	1.41	14.46
RFC	−61.41	−9.83	5.86	52.37
LOC	−486.78	−74.11	51.41	423.44
LCOM	−733.28	−320.61	−195.09	176.94

3.9.3.2 Usage of ROC Curve to Calculate the Threshold Values

Shatnawi et al. (2010) calculated threshold values of OO metrics using ROC curve. To plot the ROC curve, we need to define two variables: one binary (i.e., 0 or 1) and another continuous. Usually, the binary variable is the actual dependent variable (e.g., fault proneness or change proneness) and the continuous variable is the predicted result of a test. When the results of a test fall into one of the two obvious categories, such as change prone or not change prone, then the result is a binary variable (1 if the class is change prone, 0 if the class is not change prone) and we have only one pair of sensitivity and specificity. But, in many situations, making a decision in binary is not possible and, thus, the decision or result is given in probability (i.e., probability of correct prediction). Thus, the result is a continuous variable. In this scenario, different cutoff points are selected that make each predicted value (probability) as 0 or 1. In other words, different cutoff points are used to change the continuous variable into binary. If the predicted probability is more than the cutoff then the probability is 1, otherwise it is 0. In other words, if the predicted probability is more than the cutoff then the class is classified as change prone, otherwise it is classified as not change prone.

The procedure of ROC curves is explained in detail in Section 7.5.6, however, we summarize it here to explain the concept. This procedure is carried for various cutoff points, and values of sensitivity and 1-specificity is noted at each cutoff point. Thus, using the (sensitivity, 1-specificity) pairs, the ROC curve is constructed. In other words, ROC curves display the relationship between sensitivity (true-positive rate) and 1-specificity (false-positive rate) across all possible cutoff values. We find an optimal cutoff point, the cutoff point where balance between sensitivity and specificity is provided. This optimal cutoff point is considered as the threshold value for that metric. Thus, threshold value (optimal cutoff point) is obtained for each metric.

For example, consider the data set shown in Table A.8 (given in Appendix I). We need to calculate the threshold values for all the metrics with the help of ROC curve. As discussed, to plot ROC curve, we need a continuous variable and a binary variable. In this example, the continuous variable will be the corresponding metric and the binary variable will be "fault." Once ROC curve is constructed, the optimal cutoff point where sensitivity equals specificity is found. This cutoff point is the threshold of that metric. The thresholds (cutoff points) of all the metrics are given in Table 3.14. When the ROC curve, is constructed the optimal cutoff point is found to be 62. Thus, the threshold value of LOC is 62. This means that if a class has LOC value >62, it is more prone to faults (as our dependent variable in this example is fault proneness) as compared to other classes. Thus, focused attention can be laid on such classes and judicious allocation of resources can be planned.

TABLE 3.14

Threshold Values or the basis of ROC Curve Method

Metric	Threshold Value
WMC	7.5
DIT	1.5
NOC	0.5
CBO	8.5
RFC	43
LCOM	20.5
LOC	304.5

3.9.4 Practical Relevance and Use of Software Metrics in Research

From the research point of view, the software metrics have a wide range of applications, which help to design a better and much improved quality system:

1. Using software metrics, the researcher can identify change/fault-prone classes that
   a. Enables software developers to take focused preventive actions that can reduce maintenance costs and improve quality.
   b. Helps software managers to allocate resources more effectively. For example, if we have 26% testing resources, then we can use these resources in testing top 26% of classes predicted to be faulty/change prone.
2. Among a large set of software metrics (independent variables), we can find a suitable subset of metrics using various techniques such as correlation-based feature selection, univariate analysis, and so on. These techniques help in reducing the number of independent variables (termed as "data dimensionality reduction"). Only the metrics that are significant in predicting the dependent variable are considered. Once the metrics found to be significant in detecting faulty/change-prone classes are identified, software developers can use them in the early phases of software development to measure the quality of the system.
3. Another important application is that once one knows the metrics being captured by the models, and then such metrics can be used as quality benchmarks to assess and compare products.
4. Metrics also provide an insight into the software, as well as the processes used to develop and maintain it.
5. There are various metrics that calculate the complexity of the program. For example, McCabe metric helps in assessing the code complexity, Halstead metrics helps in calculating different measurable properties of software (programming effort, program vocabulary, program length, etc.), Fan-in and Fan-out metrics estimate maintenance complexity, and so on. Once the complexity is known, more complex programs can be given focused attention.
6. As explained in Section 3.9.3, we can calculate the threshold values of different software metrics. By using threshold values of the metrics, we can identify and focus on the classes that fall outside the acceptable risk level. Hence, during the

project development and progress, we can scrutinize the classes and prepare alternative design structures wherever necessary.

7. Evolutionary algorithms such as genetic algorithms help in solving the optimization problems and require the fitness function to be defined. Software metrics help in defining the fitness function (Harman and Clark 2004) in these algorithms.
8. Last, but not the least, some new software metrics that help to improve the quality of the system in some way can be defined in addition to the metrics proposed in the literature.

3.9.5 Industrial Relevance of Software Metrics

The software design measurements can be used by the software industry in multiple ways: (1) Software designers can use them to obtain quality benchmarks to assess and compare various software products (Aggarwal et al. 2009). (2) Managers can use software metrics in controlling and auditing the quality of the software during the software development life cycle. (3) Software developers can use the software metrics to identify problematic areas and use source code refactoring to improve the internal quality of the software. (4) Software testers can use the software metrics in effective planning and allocation of testing and maintenance resources (Aggarwal et al. 2009). In addition to this, various companies can maintain a large database of software metrics, which allow them to compare a specific company's application software with the rest of the industry. This gives an opportunity to relatively measure that software against its competitors. Comparing the planned or projected resource consumption, code completion, defect rates, and milestone completions against the actual consumption as the work progresses can make an assessment of progress of the software. If there are huge deviations from the expectation, then the managers can take corrective actions before it is too late. Also, to compare the process productivity (can be derived from size, schedule time, and effort [person-months]) of projects completed in a company within a given year against that of projects completed in previous years, the software metrics on the projects completed in a given year can be compared against the projects completed in the previous years. Thus, it can be seen that software metrics contribute in a great way to software industry.

Exercises

3.1 What are software metrics? Discuss the various applications of metrics.

3.2 Discuss categories of software metrics with the help of examples of each category.

3.3 What are categorical metric scales? Differentiate between nominal scale and ordinal scale in the measurements and also discuss both the concepts with examples.

3.4 What is the role and significance of Weyuker's properties in software metrics.

3.5 Define the role of fan-in and fan-out in information flow metrics.

3.6 What are various software quality metrics? Discuss them with examples.

3.7 Define usability. What are the various usability metrics? What is the role of customer satisfaction?

3.8 Define the following metrics:

a. Statement coverage metric

b. Defect density
c. FCMs

3.9 Define coupling. Explain Chidamber and Kemerer metrics with examples.

3.10 Define cohesion. Explain some cohesion metrics with examples.

3.11 How do we measure inheritance? Explain inheritance metrics with examples.

3.12 Define the following metrics:

a. CLD
b. AID
c. NOC
d. DIT
e. NOD
f. NOA
g. NOP
h. SIX

3.13 What is the purpose and significance of computing the threshold of software metrics?

3.14 How can metrics be used to improve software quality?

3.15 Consider that the threshold value of CBO metric is 8 and WMC metric is 15. What does these values signify? What are the possible corrective actions according to you that a developer must take if the values of CBO and WMC exceed these values?

3.16 What are the practical applications of software metrics? How can the metrics be helpful to software organizations?

3.17 What are the five measurement scales? Explain their properties with the help of examples.

3.18 How are the external and internal attributes related to process and product metrics?

3.19 What is the difference between process and product metrics?

3.20 What is the relevance of software metrics in research?

Further Readings

An in-depth study of eighteen different categories of software complexity metrics was provided by Zuse, where he tried to give basic definition for metrics in each category:

H. Zuse, *Software Complexity: Measures and Methods*, Walter De Gryter, Berlin, Germany, 1991.

Fenton's book on software metrics is a classic and useful reference as it provides in-depth discussions on measurement and key concepts related to metrics:

N. Fenton, and S. Pfleeger, *Software Metrics: A Rigorous & Practical Approach*, PWS Publishing Company, Boston, MA, 1997.

The traditional Software Science metrics proposed by Halstead are listed in:

H. Halstead, *Elements of Software Science*, Elsevier North-Holland, Amsterdam, the Netherlands, 1977.

Chidamber and Kemerer (1991) proposed the first significant OO design metrics. Then, another paper by Chidamber and Kemerer defined and validated the OO metrics suite in 1994. This metrics suite is widely used and has obtained widest attention in empirical studies:

S. Chidamber, and C. Kemerer, "A metrics suite for object-oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

Detailed description on OO metrics can be obtained from:

B. Henderson-Sellers, *Object Oriented Metrics: Measures of Complexity*, Prentice Hall, Englewood Cliffs, NJ, 1996.
M. Lorenz, and J. Kidd, *Object-Oriented Software Metrics*, Prentice Hall, Englewood Cliffs, NJ, 1994.

The following paper explains various OO metric suites with real-life examples:

K.K. Aggarwal, Y. Singh, A. Kaur, and R. Malhotra,"Empirical study of object-oriented metrics," *Journal of Object Technology*, vol.5, no. 8, pp. 149–173, 2006.

Other relevant publications on OO metrics can be obtained from:

www.acis.pamplin.vt.edu/faculty/tegarden/wrk-pap/ooMETBIB.PDF

Complete list of bibliography on OO metrics is provided at:

"Object-oriented metrics: An annotated bibliography," http://dec.bournemouth.ac.uk/ESERG/bibliography.html.