6 Software Metrics

An eminent physicist, Lord Kelvin said, "when you can measure what you are speaking about, and can express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind; it may be the beginning of knowledge, but you have scarcely in your thoughts advanced to the stage of science."

It is clear from Lord Kelvin's statement that everything should be measurable. If it is not measurable, we should make an effort to make it measurable. Thus, the area of measurement is very significant and important in all walks of life. When we discuss metrics for software engineering, situation is entirely different. Some feel that metrics are valuable management and engineering tools, while others feel that they are useless and expensive exercises in pointless data collection.

6.1 SOFTWARE METRICS: WHAT AND WHY ?

Science begins with quantification; we cannot do physics without a notion of length and time; we cannot do thermodynamics until we measure temperature. All engineering disciplines have metrics (such as metrics for weight, density, wave length, pressure and temperature) to quantify various characteristics of their products. The most fundamental question we can ask is "how big is the program"? Without defining what big means, it is obvious that it makes no sense to say, "this program will need more testing than that program" unless we know "how big they are relative to one another. Comparing two strategies also needs a notion of size. The number of tests required by a strategy should be normalized to size. For example A needs 1.4 tests per unit of size, while strategy B needs 4.3 tests per unit of size.

What is meant by size was not obvious in the early phases of science development. Newton's use of mass instead of weight was a breakthrough for physics, and early researchers in thermodynamics had heat, temperature, and entropy hopelessly confused. Size is not obvious for the software. Metrics must be objective in the sense that the measurement process is algorithmic and will yield the same results no matter who applies it [BEIZ90]. To see what kinds of metrics, we need, let us ask some questions.

1. How to measure the size of a software?
2. How much will it cost to develop a software?
3. How many bugs can we expect?
4. When can we stop testing?
5. When can we release the software?

6. What is the complexity of a module?
7. What is the module strength and coupling?
8. What is the reliability at the time of release?
9. Which test technique is more effective?
10. Are we testing hard or are we testing smart?
11. Do we have a strong program or a weak test suite?

If we want an answer to the above questions, we will have to do our own measuring and fit our own empirical laws to the measured data. Most of the metrics are aimed at getting empirical laws that relate program size (however it be measured) to expected number of bugs, expected number of tests required to find bugs, test technique effectiveness, resource requirement, release instant, reliability and quality requirement, etc.

The groundwork of software metrics was laid in the seventies. The earliest paper on the subject was of course published in 1968 [RUBE68]. From these earlier works, interesting results have emerged in the eighties. The term software metrics designates here "a unit of measurement of a software product or software related process [HAME85]."

The terms measure, measurement and metrics are often used interchangeably. However, the difference amongst these should be understood clearly. Pressman explained [PRESO5] as : "A measure provides a quantitative indication of the extent, amount, dimension, capacity, or size of some attribute of a product or process. Measurement is the act of determining a measure. The metric is a quantitative measure of the degree to which a system, component, or process possesses a given attribute". A measure may be number of errors found in a module during testing. Measurement is the result of such data collection. A software metric should relate the individual measures in some way like : average number of errors found per hour of testing.

We want to measure various characteristics of software. Some are direct measures and others are indirect measures. Fenton [FENTO4] defined measurement as :

"It is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules."

Here, we want to collect information about attributes of entities. An entity is an object (such as car, hospital) or an event (such as journey or surgical operation in hospital) in the real world. An attribute is a feature or property of an entity. Examples are colour of a car, type of hospital, cost of journey, seriousness of the operation etc.

Some of the direct measures of software process are time and effort required, maturity of the process etc. Some of the direct measures of software product are lines of code produced, number of defects found, execution speed etc. Indirect measures of software product may include complexity, efficiency, reliability, portability, maitainability etc.

6.1.1 Definition

Software metrics can be defined as [GOOD93] *"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".*

This definition covers quite a lot. Software metrics are all about measurements which, in turn, involve numbers, the use of numbers to make things better, to improve the process of developing software and to improve all aspects of the management of that process. Software metrics are applicable to the whole development life cycle from initiation, when costs must be estimated, to monitoring the reliability of the end product in the field, and the way that product changes over time with enhancement. It covers the techniques for monitoring and controlling the progress of the software development, such that the fact that it is going to be six months late is recognized as early as possible rather than the day before delivery is due. It even covers organizations determining which of its software products are the cash cows and which are the dogs.

6.1.2 Areas of Applications

The most established area of software metrics is cost and size estimation techniques. There are many proprietary packages in the market that provide estimates of software system size, cost to develop a system, and the duration of the development or enhancement of the project. These packages are based on estimation models, like COCOMO81, COCOMO-II, developed by Berry Boehm [BOEH81]. Various techniques that do not require the use of readymade tools are also available. There has been a great deal of research carried out in this area, and this research continues in all important software industries and other organizations. One thing that does come across strongly from the results of this research work is that organizations cannot rely, solely, on the use of proprietary packages.

Controlling software development projects through measurement is an area that is generating a great deal of interest. This has become much more relevant with the increase in fixed price contracts and the use of penalty clauses by customers who deal with software developers.

The prediction of quality levels for software, often in terms of reliability, is another area where software metrics have an important role to play. Again, there are proprietary models in the market that can assist this, but debate continues about their accuracy. The requirement is there both from the customer's point of view and that from the developer's, who needs to control testing and other costs. Various techniques can be used now, and this area will become more and more important in future.

The use of software metrics to provide quantitative checks on software design is also a well-established area. Much research has been carried out, and some organizations have used such techniques to very good effect. This area of software metrics is also being used to control software products, which are in place and are subject to enhancement.

Software metrics are also used to provide management information. This includes information about productivity, quality and process effectiveness. It is important to realize that this should be seen as an ongoing activity. Snapshots of the current situation have their place, but the most valuable information comes when we can see trends in data. Is productivity or quality getting better or worse over time? Why is this happening? What can management do to improve things? The provision of management information is as much an art as a science. Statistical analysis is a part of it, but the information must be presented in a way that managers find it useful, at the right time and for the right reasons. All this shows that software metrics is a vast field and have wide variety of applications throughout the software life cycle [GOOD93].

6.1.3 Problems During Implementation

Implementing software metrics in an organization of any size is difficult. There are many problems that have to be overcome and decisions that have to be made, often with limited information on hand. The first decision concerns the scope of the work. There is a rule in software development that we do not try something new on a large or critical system and this translates, in the software metrics area, to *'do not try to do too much'*. On the other hand, there is evidence that concentrating on too small an area can result in such a limited pay back as to invalidate software metrics in an organization. It is always said, *'do not bet your career on a single metric'*.

Another problem during implementation arises if we start measuring the performance of individuals. There was an organization, which used individual productivity, in terms of functionality divided by effort, as a major determinant of salary increase. While this may appear attractive to some managers, the organization later stated that this was one of the worst mistakes it ever made. Using measurement in this way is counter productive, divisive and simply ensures that developers will rig the data they supply. We may say that management has one chance and one chance only. The first time that a manager uses data supplied by an individual against that individual is the last time that the manager will get accurate or true data from that person. The reasoning behind such a statement is simple, *"employees do not like upsetting the boss!"*

There is a great temptation to seek the "silver bullet", the single measure that tells all about the software development process. Currently, this does not exist. We must realize that implementing software metrics means changing the way in which people work and think. The software engineering industry is maturing. Customers are no longer willing to accept poor quality and late deliveries. Management is no longer willing to pour money into the black hole of IT, and more management control is now a key business requirement. Competition is growing. Developers have to change and mature as well, and this can be a painful experience as they find tenets of their beliefs being challenged and destroyed. Some simple statements that could be made by many in our industry together with interpretation of current management trends will illustrate the following points [GOOD93].

- **Statement** : Software development is so complex; it cannot be managed like other parts of the organization.
  Management view: Forget it, we will find developers and managers who will manage that development.
- **Statement** : I am only six months late with this project.
  Management view: Fine, you are only out of a job.
- **Statement** : But you cannot put reliability constraints in the contract.
  Management view: Then we may not get the contract.

The list is almost endless and the message is clear.

Software metrics cannot solve all our problems, but they can enable managers to improve their processes, to improve productivity and quality, to improve the probability of survival. But it is not an easy option. Many metrics programs fail. One reason for this is that organizations and individuals who would never dream of introducing a new system without a

structured approach ignore the problems of introducing change inherent in software metrics implementation. Only by treating the implementation of software metrics as a project or programme in its own right with plans, budgets, resources and management commitment can make such an implementation succeed. Hence the use of software metrics does not ensure survival, but it improves the probability of survival.

6.1.4 Categories of Metrics

There are three categories of software metrics which are given below:

(*i*) **Product metrics:** describe the characteristics of the product such as size, complexity, design features, performance, efficiency, reliability, portability, etc.

(*ii*) **Process metrics:** describe the effectiveness and quality of the processes that produce the software product. Examples are:

- effort required in the process
- time to produce the product
- effectiveness of defect removal during development
- number of defects found during testing
- maturity of the process.

(*iii*) **Project metrics:** describe the project characteristics and execution. Examples are:

- number of software developers
- staffing pattern over the life cycle of the software
- cost and schedule
- productivity

Some metrics belong to multiple categories like quality metric may belong to all three categories. It focuses on the quality aspects of the product process, and the project. Some important metrics are discussed in subsequent sections of the chapter.

6.2 TOKEN COUNT

Two important size metrics (LOC and Function count) are discussed in chapter 4 (software project planning). These metrics have established their applicability in the field, specifically as a key input in the costing models like COCOMO, COCOMO-II, Putnam resource allocation model etc. The major problem with LOC measure is that it is not consistent because some lines are more difficult to code than others. A program is considered to be a series of tokens and if we count the number of tokens, some interesting results may emerge.

In the early 1970s, the late Professor Maurice Halstead and his co-workers at Purdue University developed the software science family of measures [HALS77]. Tokens are classified as either operators or operands. All software science measures are functions of the counts of these tokens.

Generally, any symbol or keyword in a program that specifies an algorithmic action is considered an operator, while a symbol used to represent data is considered an operand. Most punctuation marks are also categorized as operators. Variables, constants and even labels are operands. Operators consist of arithmetic symbols such as +, –, /, * and command names such

as "while", "for", "printf", special symbols such as :=, braces, parentheses, and even function names such as "eof" (end of file). The size of the vocabulary of a program, which consists of the number of unique tokens used to build a program is defined as:

$$\eta = \eta_1 + \eta_2 \quad (6.1)$$

where η : vocabulary of a program

η_1 : number of unique operators

η_2 : number of unique operands

The length of the program in terms of the total number of tokens used is

$$N = N_1 + N_2 \quad (6.2)$$

where N : program length.

N_1 : total occurrences of operators

N_2 : total occurrences of operands

It should be noted that N is closely related to the lines of code (LOC) measure of program. For machine language programs where each line consists of one operator and one operand, the program length is

$$N = 2 * LOC \quad (6.3)$$

Additional metrics are defined using these basic terms. Another measure for size of the program is called the volume.

$$V = N * \log_2 \eta \quad (6.4)$$

The unit of measurement of volume is the common unit for size "bits". It is the actual size of a program if a uniform binary encoding for the vocabulary is used. Volume may also be interpreted as the number of mental comparisons needed to write a program of length N, assuming a binary search method is used to select a member of the vocabulary of size η. Since an algorithm may be implemented by many different but equivalent programs, a program that is minimal in size is said to have the potential volume V*. Any given program with volume V is considered to implement at the program level L, which is defined by

$$L = V^* / V \quad (6.5)$$

The value of L ranges between zero and one, with L = 1 representing a program written at the highest possible level (*i.e.*, with minimum size). The inverse of the program level is termed the difficulty. That is

$$D = 1 / L \quad (6.6)$$

As the volume of an implementation of a program increases, the program level decreases and the difficulty increases. Thus, programming practices such as redundant usage of operands, or the failure to use higher-level control constructs will tend to increase the volume as well as the difficulty.

The effort required to implement a program increases as the size of the program increases. It also takes more effort to implement a program at a lower level (higher difficulty) when

compared with another equivalent program at a higher level (lower difficulty). Thus the effort in software science is defined as

$$E = V / L = D * V \tag{6.7}$$

The unit of measurement of E is elementary mental discriminations.

6.2.1 Estimated Program Length

As stated earlier, size of the program is the total number of tokens, referred to as program length, N. The first hypothesis of software science is that the length of a well-structured program is a function only of the number of unique operators and operands. This function, called the estimated length equation is denoted by $\hat{N}$ and is defined by

(HALSTEAD)

$$\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2 \tag{6.8}$$

For example, the sorting program in Fig. 4.2 (given in chapter 4) has 14 unique operators and 10 unique operands. Suppose that these numbers are known before the completion of the program, possibly by using a program design language and knowing the programming language chosen. It is then possible to estimate the length N of the program in number of tokens using equation 6.8.

$$\hat{N} = 14 \log_2 14 + 10 \log_2 10$$
$$= 53.34 + 33.22 = 86.56$$

Thus, even before constructing the sorting program, equation 6.8 predicts that there will be about 87 tokens (operators and operands) used in the completed program, which is within 10% of the program's actual length N of 91 tokens (refer Table 6.2).

There are several major flaws in the derivation of the length equation (equation 6.8). These flaws are discussed in [SHEN83]. Most of empirical support for software science is based on analysis of the relationship between estimated and actual lengths. Researchers frequently report correlation of 0.95 or higher between these quantities [FITZ78, CONT86].

The length equation can be viewed as a hypothesis whose credibility rests on several independent experiments, which have indicated that $\hat{N}$ is a reasonable estimator of N. In the process of studying Halstead's length estimator the following alternate expressions have been published to estimate program length.

(JENSEN)

$$N_j = \log_2 (\eta_1!) + \log_2 (\eta_2!) \tag{6.9}$$

This equation was proposed by Jensen & Variavan [JENS85] and found to be a much better approximation for their data set. Another equation was derived by Mehndiratta and Grover [MEHN86], which is the slight modification of Halstead length estimator.

(MG)

$$N_B = \eta_1 \log_2 \eta_2 + \eta_2 \log_2 \eta_1 \tag{6.10}$$

The Halstead estimator was further modified by Card and Agresti [CARD87] by substituting $\sqrt{\eta_1}$ for $\log_2 \eta_1$ and $\sqrt{\eta_2}$ for $\log_2 \eta_2$, thus

(CA)

$$N_C = \eta_1 \sqrt{\eta_1} + \eta_2 \sqrt{\eta_2} \tag{6.11}$$

It is claimed that for small values of η, $\sqrt{\eta}$ and $\log_2 \eta$ behave similarly. Unfortunately, the accuracy of the estimator does not improve.

Although it is easy to construct a pathological program to make $\hat{N}$ a poor predictor of N, there is overwhelming evidence using existing analysers to suggest the validity of the length equation in several languages. Shen et.al. [SHEN83] suggested that a misclassification of any token has virtually no effect on the final estimate.

Since $\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2 \approx (\eta \log_2 \eta) / 2$ regardless of how the vocabulary of size η is divided into operators and operands. Hence

(SHEN) $$N_S = (\eta \log_2 \eta) / 2 \qquad (6.12)$$

The definitions of unique operators, unique operands, total operators and total operands are not specifically delineated. Bulut [BULU73] has considered the counting methods used for FORTRAN, ALGOL & machine languages. Conte [CONE86] suggested counting rules for PASCAL and James Elshoff [JAME78] reported rules for PL/1 programs. Counting rules for C languages are suggested [YOGE95] and are given below:

Counting rules for C language

1. Comments are not considered.
2. The identifier and function declarations are not considered.
3. All the variables and constants are considered operands.
4. Global variables used in different modules of the same program are counted as multiple occurrences of the same variable.
5. Local variables with the same name in different functions are counted as unique operands.
6. Function calls are considered as operators.
7. All looping statements *e.g.*, do{...} while ( ), while ( ) {...}, for ( ) {...}, all control statements *e.g.*, if ( ) {...}, if ( ) {...} else {...}, etc. are considered as operators.
8. In control construct switch ( ) { case : ...}, switch as well as all the case statements are considered as operators.
9. The reserve words like return, default, continue, break, size of, etc., are considered as operators.
10. All the brackets, commas, and terminators are considered as operators.
11. GOTO is counted as an operator and the label is counted as an operand.
12. The unary and binary occurrence of "+" and "–" are dealt separately. Similarly "*" (multiplication operator), and "*" (de referencing operator), "&" (bitwise AND operator) and "&" (address operator) are dealt with separately.
13. In the array variables such as "array-name[index]" "array-name" and "index" are considered as operands and [ ] is considered as operator.
14. In the structure variables such as "struct-name, member-name" or "struct-name →member-name", struct-name, member-name are taken as operands and '.', '→' are taken as operators. Same names of member elements in different structure variables are counted as unique operands.
15. All the hash directives are ignored.

The counting method presented here is fairly complete, however, "C" is so rich and complex that probability of additions and exceptions is always there.

6.2.2 Potential Volume

Many different but equivalent programs may implement an algorithm. Amongst all these programs, the one that has minimal size is said to have the potential volume V*. Halstead argued that the minimal implementation of any algorithm was through a reference to a procedure that had been previously written. The implementation of this algorithm would then require nothing more than invoking the procedure and supplying the operands for its input and output parameters. It is shown that the potential volume of an algorithm implemented as a procedure call could be expressed as

$$V^* = (2 + \eta_2^*) \log_2 (2 + \eta_2^*) \quad (6.13)$$

The first term in the parentheses, 2, represents the two unique operators for the procedure call — the procedure name and a grouping symbol that separates the procedure name from its parameters. The second term, η_2^* represents the number of conceptually unique input and output parameters. η_2^* can probably be determined for small application programs, it is much more difficult to compute for large programs, such as compiler or an operating system. In these cases it is difficult to identify precisely the conceptually unique operands.

6.2.3 Estimated Program Level / Difficulty

Halstead offered an alternate formula that estimates the program level.

$$\hat{L} = 2\eta_2 / (\eta_1 N_2) \quad (6.14)$$

Hence $$\hat{D} = \frac{1}{\hat{L}} = \frac{\eta_1 N_2}{2\eta_2}$$

An intuitive argument for this formula is that programming difficulty increases, if additional operators are introduced (*i.e.*, if $\eta_1/2$ increases) and if an operand is used repetitively (*i.e.*, if N_2/η_2 increases) . Every parameter in equation 6.14 may be obtained by counting the operators and operands in a finished computer program.

6.2.4 Effort and Time

Halstead hypothesized that the effort required to implement a program increases as the size of the program increases. It also takes more effort to implement a program at a lower level (higher difficulty) than another equivalent program at a higher level (lower difficulty). Thus, effort E measured in elementary mental discriminations is:

$$E = V / \hat{L} = V^* \hat{D}$$
$$= (\eta_1 N_2 N \log_2 \eta) / 2\eta_2 \quad (6.15)$$

A major claim for software science is its ability to relate its basic metrics to actual implementation. A psychologist, John Stroud, suggested that the human mind is capable of making a limited number of elementary discriminations per second [STRO67]. Stroud claimed that this number β (called the stroud number) ranges between 5 and 20. Since effort E uses elementary mental discriminations as its unit of measure, the programming time T of a program in seconds is simply

$$T = E / \beta \quad (6.16)$$

β is normally set to 18 since this seemed to give best results in Halstead's earliest experiments, which compared the predicted times with observed programming times, including the time for design, coding, and testing. Halstead claimed that this formula can be used to estimate programming time when a problem is solved by one proficient, concentrating programmer writing a single module program.

6.2.5 Language Level

There are now literally hundreds of programming languages. In some organizations, several languages may be used on a regular basis for software development. For example, in some large companies, software is being developed using FORTRAN, PASCAL, COBOL, and assembly language. There are proponents of each major language (including those of Ada, C, and Prolog) that argue that their favourite language is the best to use. These arguments suggest the need for a metric that expresses the power of a language [SHEN83].

Halstead hypothesized that, if the programming language is kept fixed, as V* increases L decreases in such a way that the product $L \times V^*$, remains constant. Thus, this product, which he called the language level, λ can be used to characterize a programming language. Lower value of λ means that the language is closer to the machine.

$$\lambda = L \times V^* = L^2 V \quad (6.17)$$

Using this formula, Halstead and other researchers determined the language level for various languages as shown in Table 6.1 [HALS77, SHEN83, YOGE95].

Table 6.1: Language levels

Language	*Language level* λ	*Variance* σ
PL/1	1.53	0.92
ALGOL	1.21	0.74
FORTRAN	1.14	0.81
CDC Assembly	0.88	0.42
PASCAL	2.54	–
APL	2.42	–
C	0.857	0.445

These averages, λ's follow the intuitive rankings of most programmers for these languages, but they all have large variances.

The current state of software science seems to be still that of a evolving theory. There are those who question (with good reason in most cases) some of its underlying assumptions. However, there is large body of published data that suggests that software science metrics may be useful; especially η_1 and η_2 have been shown to strongly correlate to program size and error rates. Researchers are therefore continuing to find these metrics useful as a basis for size and effort models. Thus, Halstead work served to stimulate a great deal of interest in software metrics, as well as to contribute some basic metrics that survive till today.

Example 6.1

Consider the sorting program given in Fig. 4.2 of chapter 4 (software project planning). List out the operators and operands and also calculate the values of software science measures like η, N, V, E, λ etc.

Solution

The list of operators and operands is given in Table 6.2.

Table 6.2: Operators and operands of sorting program of Fig. 4.2 of chapter 4.

Operators	*Occurrences*	*Operands*	*Occurrences*
int	4	SORT	1
( )	5	*x*	7
,	4	*n*	3
[ ]	7	*i*	8
if	2	*j*	7
<	2	save	3
;	11	im1	3
for	2	2	2
=	6	1	3
–	1	0	1
< =	2	—	—
+ +	2	—	—
return	2	—	—
{ }	3	—	—
$\eta_1 = 14$	$N_1 = 53$	$\eta_2 = 10$	$N_2 = 38$

Here $N_1 = 53$ and $N_2 = 38$. The program length $N = N_1 + N_2 = 91$

Vocabulary of the program $\eta = \eta_1 + \eta_2 = 14 + 10 = 24$

Volume $V = N \times \log_2 \eta$

$= 91 \times \log_2 24 = 417$ bits.

If a binary encoded scheme is used to represent each of the 24 items in the vocabulary, it would take 5 bits per item, since a 4 bit scheme leads to 16 unique codes (which is not enough), and a 5-bit scheme leads to 32 unique codes (which is more than sufficient). Each of the 91 tokens used in the program could be represented in order by a 5-bit code, leading to a string of $5 \times 91 = 455$ bits that would allow us to store the entire program in memory. Notice that size analysis is based on storing not a compiled version of the subroutine, but a binary translation of the original program. We could then say that this program occupies 455 bits of storage in its encoded form. However, also notice that a 5-bit scheme allows for 32 tokens

instead of just 24 tokens, so instead of using the integer 5, volume uses the non-integer $\log_2 \eta = 4.58$ to arrive at a slightly smaller volume of 417.

The estimated program length $\hat{N}$ of the program

$$= 14 \log_2 14 + 10 \log_2 10$$
$$= 14 * 3.81 + 10 * 3.32$$
$$= 53.34 + 33.2 = 86.45$$

Conceptually unique input and output parameters are represented by η_2^*

$\eta_2^* = 3$ {x : array holding the integer to be sorted. This is used both as input and output}.
{N : the size of the array to be sorted}.

The potential volume $V^* = 5 \log_2 5 = 11.6$

Since $L = V^* / V$

$$= \frac{11.6}{417} = 0.027$$

$$D = I / L$$

$$= \frac{1}{0.027} = 37.03$$

Estimated program level

$$\hat{L} = \frac{2}{\eta_1} \times \frac{\eta_2}{N_2} = \frac{2}{14} \times \frac{10}{38} = 0.038$$

which is not very close to the 0.027 level, we determined earlier using the conceptually unique operands.

We may use another formula

$$\hat{V}^* = V \times \hat{L} = 417 \times 0.038 = 15.67$$

The discrepancy between V^* and $\hat{V}^*$ does not inspire confidence in the application of this portion of software science theory to more complicated programs.

$$E = V/\hat{L} = \hat{D} \times V$$
$$= 417 / 0.038 = 10973.68$$

Therefore, 10974 elementary mental discriminations are required to construct the program.

$$T = E / \beta = \frac{10974}{18} = 610 \text{ seconds} \approx 10 \text{ minutes}$$

This is probably a reasonable time to produce the program, which is very simple.

Table 6.3

```
#include < stdio.h >
#define MAXLINE 100
int getline(char line[],int max);
int strindex(char source[],char search for[]);
char pattern[ ]="ould";
int main()
{
    char line[MAXLINE];
    int found = 0;
    while(getline(line,MAXLINE)>0)
        if(strindex(line, pattern)>=0)
        {
            printf("%s",line);
            found++;
        }
    return found;
}
int getline(char s[],int lim)
{
    int c,i=0;
    while(--lim > 0 && (c=getchar())!= EOF && c!='\n')
        s[i++]=c;
    if(c=='\n')
        s[i++] = c;
    s[i] = '\0';
    return i;
}
int strindex(char s[],char t[])
{
```

```
        int i,j,k;
        for(i=0;s[i] !='\0';i++)
        {
            for(j=i,k=0;t[k] != '\0',s[j] ==t[k];j++,k++);
            if(k>0 && t[k] =='\0')
                return i;
        }
        return -1;
    }
```

Example 6.2

Consider the program shown in Table 6.3. Calculate the various software science metrics.

Solution

List of operators and operands are given in Table 6.4.

Table 6.4

Operators	*Occurrences*	*Operands*	*Occurrences*
main ()	1	—	—
–	1	**Extern variable** pattern	1
for	2	**Main function** line	3
= =	3	found	2
! =	4	**Getline function** s	3
getchar	1	lim	1
( )	1	*c*	5
&&	3	*i*	4
– –	1	**Strindex function** *s*	2
return	4	*t*	3
+ +	6	*i*	5
printf	1	*j*	3
> =	1	*k*	6
strindex	1	**Numerical operands** 1	1
If	3	Maxline	1
>	3	0	8
getline	1	'\0'	4

(Contd.)...

while	2	'\n'	2
{ }	5	**Strings "ould"**	1
=	10	—	—
[ ]	9	—	—
,	6	—	—
;	14	—	—
EOF	1	—	—
$\eta_1 = 24$	$N_1 = 84$	$\eta_2 = 18$	$N_2 = 55$

Program vocabulary $\eta = 42$

Program length $N = N_1 + N_2$

$= 84 + 55 = 139$

Estimated length $\hat{N} = 24 \log_2 24 + 18 \log_2 18 = 185.115$

% error $= 24.91$

Program volume $V = 749.605$ bits

Estimated program level $= \frac{2}{\eta_1} \times \frac{\eta_2}{N_2}$

$= \frac{2}{24} \times \frac{18}{55} = 0.02727$

Minimal volume $V^* = 20.4417$

Effort $= V/\hat{L}$

$= \frac{749.605}{.02727}$

$= 27488.33$ elementary mental discriminations.

Time $T = E/\beta = \frac{27488.33}{18}$

$= 1527.1295$ seconds

$= 25.452$ minutes

6.3 DATA STRUCTURE METRICS

Essentially, the need for software development and other activities are to process data. Some data is input to a system, program, or module; some data may be used only internally; and some data is the output from a system, program, or module. A few examples of input, internal, and output data appear in Fig. 6.1.

Program	*Data input*	*Internal data*	*Data output*
Payroll	Name / Social Security No. / Pay Rate / Number of hours worked	Withholding rates Overtime factors Insurance premium Rates	Gross pay withholding Net pay Pay ledgers
Spread sheet	Item Names / Item amounts / Relationships among items	Cell computations Sub-totals	Spreadsheet of items and totals
Software Planner	Program size / No. of software developers on team	Model parameters Constants Coefficients	Est. project effort Est. project duration

Fig. 6.1: Some examples of input, internal, and output data [CONT86]

Thus, an important set of metrics is that, which captures the amount of data input to, processed in and output from software. For example, assume that a problem can be solved in two ways, resulting in programs A and B. A has 25 input parameters, 35 internal data items, and 10 output parameters. B has 5 input parameters, 12 internal data items, and 4 output parameters. We can assume that A is probably more complicated, took more time to program, and has a greater probability of errors than B.

A count of the amount of data input to, processed in, and output from software is called a data structure metric. This section presents several data structure metrics. Some concentrate on variables (and even constants) within each module and ignore the input/output dependencies. Others concern themselves primarily with the input/output situation. There is no general agreement on how the line of code measure (the classical and best-known software metric) is to be counted. Thus, it is not surprising that there are various methods for measuring data structures as well. In the following subsections, we will discuss the metrics proposed to measure the amount of data, the usage of data within modules, and the degree to which data is shared among modules [CONT86].

6.3.1 The Amount of Data

Most compilers and assemblers have an option to generate a cross-reference list, indicating the line where a certain variable is declared and the line or lines where it is referenced. Such a list is useful in debugging and maintenance, and can help determine the amount of data in the program. Consider a simple program which appears in Fig. 6.2. It inputs work hours and pay rates, and computes gross pay, taxes, and net pay. The C compiler produces a cross-reference listing for this program, which appears in Fig. 6.3.

One method for determining the amount of data is to count the number of entries in the cross-reference list. Be careful to exclude from the count those variables that are defined but never used. The definitions of these variables may be made for future reference, but they do not affect the operational characteristics of the program or, more importantly, the difficulty of development, and should not be counted. Such a count of variables will be referred to as VARS. Thus, for the program payday appearing in the Fig. 6.2 VARS = 7. For the sample program

SORT in Fig. 4.2 VARS = 6 (from a cross-reference listing that identified X, N, I, J, SAVE and IM1 as the variables in the program). The count of variables VARS depends on the following definition:

A variable is a string of alphanumeric characters that is defined by a developer and that is used to represent some value during either compilation or execution.

```
1.  #include < stdio. h >
2.  struct check
3.    {
4.      float gross, tax, net;
5.    } pay;
6.    float hours, rate;
7.    void main ( )
8.      {
9.       while (! feof (stdin))
10.        {
11.          scanf("%f %f", & hours, & rate);
12.          pay. gross = hours * rate;
13.          pay. tax = 0.25 * pay. gross;
14.          pay. net = pay. gross - pay. tax;
15.          printf("%f %f %f/n", pay. gross, pay. tax, pay. net);
16.        }
17.     }
```

Fig. 6.2: Payday program

check	2				
gross	4	12	13	14	15
hours	6	11	12		
net	4	14	15		
pay	5	12	13	13	14
	14	14	15	15	15
rate	6	11	12		
tax	4	13	14	15	

Fig. 6.3: A cross reference of program payday

Although a simple way to obtain VARS is from a cross-reference list, it can also be generated using a software analyzer that counts the individual tokens [CONT86].

While it may sound simple to determine the value of VARS — certainly, counting the number of variables in a program seems straightforward—there are some items in the cross-reference listing of the program in Fig. 6.2 that have been deleted. These items are listed in Fig. 6.4. The items feof stdin are related to I/O. The cross-referencing software called the name of the program payday a variable. However, because none of these are variables in the sense of variables that we create to produce a program, we deleted them from Fig. 6.3, but it should be clear that this "algorithmic" metric VARS is, in fact, a little subjective. In determining this metric, as with all other software metrics including lines of code, we attempt to establish guidelines that eliminate as much subjectivity as possible. But, the reader is well advised to realize that total objectivity is impossible.

feof	9
stdin	10

Fig. 6.4: Some items not counted as VARS

Among all the variable names in line 13 of Fig. 6.2 is the constant 0.25. This program assumes that all pay will be taxed at a 25% rate. Also, in line 05 of Fig. 4.2 of chapter 4, the constant 2 is used to avoid sorting arrays with less than two elements. None of these constants 0.25 or 2 are counted as VARS, and yet they play special purposes in the programs. Furthermore, mathematical constants such as τ and ε are important for programs involving trigonometric or logarithmic applications. Even array references with an explicit index, such as A[11], may indicate some special meaning for that particular location.

Halstead [HALS77] introduced a metric that he referred to as η_2 to be a count of the operands in a program—including all variables, constants, and labels. Thus,

$$\eta_2 = \text{VARS} + \text{unique constants} + \text{labels}.$$

The sample SORT program in Fig. 4.2 which is analysed in Table 6.2 has 6 variables (X, N, I, J SAVE, IM1), 3 constants (1, 2, 0) and 01 labels (SORT) so that $\eta_2 = 10$. The name of the subroutine SORT is treated as a label since it is the label that will be used by any other program or sub-program that wants to access SORT. The program payday in Fig. 6.2 has 7 variables (check, gross, hours, net, pay, rate, tax), 1 constant 0.25, and no label. Thus, its η_2 is 8 note that η_2 is the count of the number of unique operands. Thus, this metric fails to capture an important feature of the "amount of data"—namely, the total operand usage. For example, given the 8 operands in Fig. 6.2, it is possible to construct the program shown or to construct a much larger and more complicated program in order to measure the quantity of usage of the operands. Halstead further defined the metric total occurrence of operands, and named it N_2. Fig. 6.5 repeats Fig. 6.2 and encloses each operand occurrences in brackets [CONT86].

The program payday uses the 8 operands 30 times: some are used several times (like pay) and some are used sparingly (0.25 is used only once). Thus, $N_2 = 30$ for this program.

The metrics VARS, η_2, and N_2 are the most popular data structure measures. They seem to be robust, slight variations in algorithm computation schemes for computing them do not seem to affect inordinately other measures based upon them.

```
1  # include < stdio. h >
2  struct [check]
3  {
4    float [gross], [tax], [net];
5  } [pay];
6  float [hours], [rate);
7  void main ( )
8  {
9  while (! feof (stdin))
10   {
11     scanf("% f % f", & [hour], & [rate]);
12     [pay] . [gross] = [hours] * [rate];
13     [pay] . [tax] = 0.25 * [pay] . [gross];
14     [pay] . [net] = [pay] . [gross] - [pay] . [tax];
15     printf("% f % f % f/n", [pay] . [gross] [pay] . [tax], [pay] . [net]);
16   }
17 }
```

Fig. 6.5: Program payday with operands in brackets

6.3.2 The Usage of Data within a Module

In Fig. 6.6 the program "bubble" inputs two related integer arrays (a and b) of the same size up to 100 elements each. It uses a bubble sort on the a-array, interchanges the b-array values to keep them with the accompanying a-array values, and outputs the results. Prior to Fig. 6.6, all of our examples have illustrated small, single-module programs or subroutines. Fig. 6.6 contains a main program in lines 11–37 and a sub-program procedure swap in lines 3–9.

Several metrics may be computed for individual modules. In order to characterize the intra-module data usage, we may use the metrics live variables and variable spans that are discussed below.

Live variables: While constructing program "bubble", the developer created a variable "last". Analyse Fig. 6.6 carefully to see that all array elements beyond the "last" one are sorted. While the program is running, if size = 25 and last = 14, then all items a[15]–a[25] and b[15]–b[25] are in order even though the first 14 elements of each array are not yet sorted. A beginning value for "last" is established in line 17, decremented in line 22, and used in the logical expression in line 24.

There are only three statements in this program in which "last" appears, excluding the declaration in line 13. Does this mean that we do not need to be concerned with "last" while constructing the statements other than 17, 22, and 24? Certainly not. Between statements 17 and 24, it is important to keep in mind what "last" is doing. For example statements 18–19 are

used to set up a potentially never-ending loop. However, even though these statements never mention "last", the developer realized that each time on a-value "bubbles down" to its appropriate position. "Last" will be decremented by one. Eventually on some cycle through the a-values none will be swapped and the loop beginning in statement 19 will be exited. As we will show later, "last" has life span that begins at statement 17 and extends through statement 24.

Thus, a developer must constantly be aware of the status of a number of data items during the development process. A reasonable hypothesis is: more the data items that a developer must keep track of when constructing statements, the more difficult it is to construct. Thus, our interest lies in the size of the set of those data items called live variables (LV) for each statement in the program.

As suggested earlier, the set of live variables for a particular statement is not limited to the number of variables referenced in that statement. For example, the statement being considered may be just one of the several that set up the parameters for a complex procedure. The developer must be aware of entire list of parameters to know that they are being set up in an orderly fashion, so that any statement in the group disturbs no variables later. Therefore, there are several possible definitions of a live variable [DUNS579].

1. A variable is live from the beginning of a procedure to the end of the procedure.

2. A variable is live at a particular statement only if it is referenced a certain number of statements before or after that statement.

3. A variable is live from its first to its last references within a procedure.

The first definition, while computationally simple, does not correspond to the idea of the live variable. According to this definition, both the variable "last" with a 8-statement life span (lines 17–24) and the variable "size" with a 22 statement life span (lines 14–35) can be considered alive throughout the procedure. The second definition might work, but there is no agreement on what a "certain number of statements" should be and no successful use has been reported.

The third definition meets the spirit of the live variable idea and is easy to compute algorithmically. In fact, a computer program (a software analyzer) can produce live variable counts for all statements in a program or procedure.

```
1   #include < stdio. h >
2
3   void swap (int x [ ], int K)
4     {
5       int t;
6       t = x[K];
7       x[K] = x[K + 1];
8       x[K + 1] = t;
9     }
```

(Contd.)...

```
10
11  void main ( )
12  {
13  int i, j, last, size, continue, a[100], b[100];
14  scanf("% d", & size);
15  for (j = 1; j < = size; j + +)
16  scanf("%d %d", & a[j], & b[j];
17  last = size;
18  continue = 1;
19  while(continue)
20  {
21   continue = 0;
22   last = last-1;
23   i = 1;
24  while (i < = last)
25  {
26   if (a[i] > a[i + 1])
27   {
28    continue = 1;
29    swap (a, i);
30    swap (b, i);
31   }
32    i = i + 1;
33  }
34  }
35  for (j = 1; j < = size; j ++)
36  printf("%d %d\n", a[j], b[j]);
37  }
```

Fig. 6.6: Bubble sort program

It is thus possible to define the average number of live variables ($\overline{LV}$), which is the sum of the count of live variables divided by the count of executable statements in a procedure. This is a complexity measure for data usage in a procedure or program. The live variables in the program in Fig. 6.6 appear in Fig. 6.7 the average live variables for this program is $\frac{124}{34} = 3.647$.

Line	*Live variables*	*Count*
4	—	0
5	—	0
6	*t,x,k*	3
7	*t,x,k*	3
8	*t,x,k*	3
9	—	0
10	—	0
11	—	0
12	—	0
13	—	0
14	size	1
15	size, *j*	2
16	size, *j*, *a*, *b*	4
17	size, *j*, *a*, *b*, last	5
18	size, *j*, *a*, *b*, last, continue	6
19	size, *j*, *a*, *b*, last, continue	6
20	size, *j*, *a*, *b*, last, continue	6
21	size, *j*, *a*, *b*, last, continue	6
22	size, *j*, *a*, *b*, last, continue	6
23	size, *j*, *a*, *b*, last, continue, *i*	7
24	size, *j*, *a*, *b*, last, continue, *i*	7
25	size, *j*, *a*, *b*, continue, *i*	6
26	size, *j*, *a*, *b*, continue, *i*	6
27	size, *j*, *a*, *b*, continue, *i*	6
28	size, *j*, *a*, *b*, continue, *i*	6
29	size, *j*, *a*, *b*, *i*	5
30	size, *j*, *a*, *b*, *i*	5

(Contd.)...

31	size, *j*, *a*, *b*, *i*	5
32	size, *j*, *a*, *b*, *i*	5
33	size, *j*, *a*, *b*	4
	size, *j*, *a*, *b*	4
35	size, *j*, *a*, *b*	4
36	*j*, *a*, *b*	3
37	—	0

Fig. 6.7: Live variables for the program in Fig. 6.6

As shown live variables depend on the order of statements in the source program, rather than the dynamic execution-time order in which they are encountered. A metric based on run-time order would be more precisely related to the life of the variable, but would be much more difficult to define algorithmically (especially in a non-structured programming language).

Variable spans: Two variables can be alive for the same number of statements, but their use in a program can be markedly different. For example, Fig. 6.8 lists all of the statements in a C program that refer to the variables "a" and "b". Both variables are alive for the same 40 statements (21–60), but "a" is referred to three times while "b" is mentioned only once. A metric that captures some of the essence of how often a variable is used in a program is called the span (SP). This metric is the number of statements between two successive references of the same variable [ELSH76]. The span is related to the third definition of live variables. For a program that references a variable in n statements, there are n-1 spans for that variable. Thus, in Fig. 6.8 "a" has 4 spans and "b" has only 2. Intuitively this tells us that 'a' is being used more than 'b'.

```
...
21        scanf (" %d %d," & a, & b);
...
32        x = a;
...
45        y = a – b;
...
53        z = a;
...
60        printf (" %d %d," a, b);
...
```

Fig. 6.8: Statements in ac program referring to variables a and b

Furthermore, the size of a span indicates the number of statements that pass between successive uses of a variable. A large span can require the developer to remember during the construction process a variable that was last used in the program. In Fig. 6.8 "*a*" has 4 spans of 10, 12, 7, and 6, statements, while for "*b*" has 2 spans of 23 and 14 statements. It is simple to extend this metric to "average span size", ($\overline{SP}$) in which case "*a*" has an average span size of 8.75 and '*b*' has an average span size of 18.5.

Making program-wide metrics from intra-module metrics: Each of the metric discussed in this section is intended to be used within a module, as indicated. But it is possible to extend each one into an inter-module metric. For example if we want to characterize the average number of live variables for a program having modules, we can use this equation.

$$\overline{LV}_{program} = \frac{\sum_{i=1}^{m} \overline{LV}_i}{m}$$

where $\overline{LV}_i$ is the average live variable metric computed from the ith module.

Furthermore, the average span size ($\overline{SP}$) for a program of n spans could be computed by using the equation.

$$\overline{SP}_{program} = \frac{\sum_{i=1}^{n} \overline{SP}_i}{n}$$

6.3.3 Program Weakness

A program consists of modules. Using the average number of live variables ($\overline{LV}$) and average life of variables (γ), the module weakness has been defined as [YOGE98]:

$$WM = \overline{LV} * \gamma$$

Average number of live variables and average life of variables can be found using some automated tools. Even most compilers and assemblers have an option to generate a cross-reference list, indicating the line number where a certain variable is declared and the line or lines where it is referenced. Such a list may be used to compute the value of LV and γ. Using these two values, weakness of a module can be computed. The weakness of the module can be used to estimate the testability and maintainability. If weakness of a module is more, testability will be better and vice-versa. Weakness will also have effect on maintainability. A weaker module will be more difficult to maintain.

As we all know a program is normally a combination of various modules, hence program weakness can be a useful measure and is defined as:

$$WP = \frac{\left(\sum_{i=1}^{m} WM_i\right)}{m}$$

where, WM_i : weakness of ith module.
WP : weakness of the program
m : number of modules in the program.

Example 6.3

Consider a program for sorting and searching. The program sorts an array using selection sort and than search for an element in the sorted array. The program is given in Fig. 6.8. Generate cross–reference list for the program and also calculate $\overline{LV}$, γ, and WM for the program.

Solution

The given program is of 66 lines and has 11 variables. The variables are *a, i, j,* item, min, temp, low, high, mid, loc and option.

```
/*************************************************************************/
/**** PROGRAM TO SORT AN ARRAY USING SELECTION SORT & THEN SEARCH *******/
/**** FOR AN ELEMENT IN THE SORTED ARRAY ********/
/*************************************************************************/

#include <stdio.h>
#define MAX 10

main ()
{
  int a[MAX];
  int i,j,item,min,temp;
  int low=0,high,mid,loc;
  char option;

  for (i=0;i<MAX;i++)
   {
   printf("Enter a[%d]:",i);
   scanf("%d",&a[i]);
   }
  /* selection sort */
   for (i=0;i<(MAX-1);i++)
   {
   min=i;
   for (j=i+1; j<MAX;j++)
     {
     if (a[min]>a[j])
       {
       temp=a[min];
       a[min]=a[j];
       a[j]=temp;
```

(Contd.)...

```
32          }
33        }
34      }
35  printf("\n The Sorted Array:\n");
36  for (i=0; i<MAX;i++)
37    printf("\n a[%d]=%d",i,a[i]);
38  printf("\n Do you want to search any element in the array
                                                    (Y/N):");
39  fflush(stdin);
40  scanf("%c",& option);
41  if (toupper(option)=='Y')
42     {
43     printf("\n Enter the item to be searched :");
44     scanf("%d", &item);
45     high=MAX;
46     mid=(int)(low+high)/2;
47     while ((low<=high)&&(item!=a[mid]))
48     {
49       if (item>a[mid])
50       low=mid+1;
51       else high=mid-1;
52       mid=(int) (low+high)/2;
53     }
54     if(low>high)
55     {
56       loc=0;
57       printf("\n No such item is present in the array\n");
58     }
59     if (item==a[mid])
60     {
61       loc=mid;
62       printf("\n The item %d is present at location %d in the sorted
         array\n",item,loc);
63     }
64    }
65   printf("\n Sorting & Searching done");
66  }
```

Fig. 6.8: Sorting and searching program

Cross-Reference list of the program is given below:

a	11	18	19	27	27	29	30	30	31	37	47	49	59	
i	12	16	16	16	18	19	22	22	22	24	36	36	36	37	37
j	12	25	25	25	27	30	31								
item	12	44	47	49	59	62									
min	12	24	27	29	30										
temp	12	29	31												
low	13	46	47	50	52	54									
high	13	45	46	47	51	52	54								
mid	13	46	47	49	50	51	52	59	61						
loc	13	56	61	62											
option	14	40	41												

Live variables per line are calculated as:

Line number	*Live variables on the line*	*Count*
13	low	1
14	low	1
15	low	1
16	low, *i*	2
17	low, *i*	2
18	low, *i*, *a*	3
19	low, *i*, *a*	3
20	low, *i*, *a*	3
22	low, *i*, *a*	3
23	low, *i*, *a*	3
24	low, *i*, *a*, min	4
25	low, *i*, *a*, min, *j*	5
26	low, *i*, *a*, min, *j*	5
27	low, *i*,*a*, min, *j*	5
28	low, *i*, *a*, min, *j*	5
29	low, *i*, *a*, min, *j*, temp	6
30	low, *i*, *a*, min, *j*, temp	6
31	low, *i*, *a*, *j*, temp	5
32	low, *i*, *a*,	3
33	low, *i*, *a*	3
34	low, *i*, *a*	3
35	low, *i*, *a*	3

(Contd.)...

36	low, *i*, *a*	3
37	low, *i*, *a*	3
38	low, *a*	2
39	low, *a*	2
40	low, *a*, option	3
41	low, *a*, option	3
42	low, *a*	2
43	low, *a*	2
44	low, *a*, item	3
45	low, *a*, item, high	4
46	low, *a*, item, high, mid	5
47	low, *a*, item, high, mid	5
48	low, *a*, item, high, mid	5
49	low, *a*, item, high, mid	5
50	low, *a*, item, high, mid	5
51	low, *a*, item, high, mid	5
52	low, *a*, item, high, mid	5
53	low, *a*, item, high, mid	5
54	low, *a*, item, high, mid	5
55	*a*, item, mid	3
56	*a*, item, mid, loc	4
57	*a*, item, mid, loc	4
58	*a*, item, mid, loc	4
59	*a*, item, mid loc	4
60	item, mid, loc	3
61	item, mid, loc	3
62	item, loc	2
63		0
64		0
65		0
66		0
	Total	174

Thus Avg. number of Live Variables $(\overline{LV}) = \dfrac{\text{Sum of count of live variables}}{\text{Count of executable statements}}$

$$\overline{LV} = \frac{174}{53} = 3.28$$

$$\Rightarrow \quad \overline{LV} = 3.28$$

$$\gamma = \frac{\text{Sum of count of live variables}}{\text{Total no. of variables}}$$

$$\Rightarrow \qquad \gamma = \frac{174}{11} = 15.8$$

$$\Rightarrow \qquad \gamma \text{ (}i.e.\text{ Avg. life of variables)} = 15.8$$

Module weakness

$$WM = \overline{LV} \times \gamma$$

where WM is the module weakness

$\overline{LV}$ is the Avg. no. of live variables

& γ is the Avg. life of variables

$$\Rightarrow \qquad WM = 3.28 \times 15.8 = 51.8$$

$$WM = 51.8$$

Example 6.4

Consider a program given in Fig. 6.9 that draws a circle using midpoint algorithm. Generate cross reference list of variables and also calculate average number of live variables ($\overline{LV}$), average life of variables (γ) and program weakness (WM).

Solution

There are 9 variables declared in the program. The variables are rad, p_0, p_1, x, y, x_c, y_c, d and m.

```
\\To scan convert a circle using midPoint Algorithm
1.   #include <iostream.h>
2.   #include <conio.h>
3.   #include <graphics.h>
4.   void circle (int, int, int, int);
5.   void  main ()
6.   {
7.   int rad;
8.   int p0, p1;
9.   int x, y;
10.  int xc, yc;
11.  int d,m;
12.  d = DETECT;
13.  initgraph (&d, &m," ");
14.  setbkcolour(BLACK);
15.  clrscr ();
16.  cout <<" enter the radius of circle:";
17.  cin >> rad;
18.  cout <<"Enter the value of center co-ordinates:";
```

(Contd.)...

```
19.   cin >> xc >> yc;
20.   x = 0;
21.   y = rad;
22.   circle(xc, yc, x, y);
23.   p0 = 1 - rad;
24.   while (x <y)
25.   {
26.   if(p0 <0)
27.   {
28.   x ++ ;
29.   p0 = p0 + 2*(x + 1) + 1;
30.   circle (x, y, xc, yc);
31.   }
32.   else
33.   {
34.   x ++;
35.   y --;
36.   p0 = p0 + 2* (x - y) + 1;
37.   circle(x, y, xc, yc);
38.   }
39.   }
40.   getch();
41.   }
42.   void circle(int x, int y, int xc, int yc)
43.   {
44.   putpixel(xc + x, yc + y, 4);
45.   putpixel(xc - x, yc + y, 4);
46.   putpixel(xc + x, yc - y, 4);
47.   putpixel(xc - x, yc - y, 4);
48.   putpixel(xc + y, yc + x, 4);
49.   putpixel(xc - y, yc + x, 4);
50.   putpixel(xc + y, yc - x, 4);
51.   putpixel(xc - y, yc - x, 4);
52.   }
```

Fig. 6.9: Program for drawing a circle using midpoint algorithm

The cross reference list is given below:

Variable	*Reference a Line Number*
rad	7, 17, 21, 23,
p_0	8, 23, 26, 29, 29,36, 36,
p_1	8
x	9, 20, 22, 24, 28, 29, 30, 34, 36, 37, 42, 44, 45, 46, 47, 48, 49, 50, 51
y	9, 21, 22, 24, 30, 35, 36, 37, 42, 44, 45, 46, 47, 48, 49, 50, 51
x_c	10, 19, 22, 30, 37, 42, 44, 45, 46, 47, 48, 49, 50, 51
y_c	10, 19, 22, 30, 37, 42, 44, 45, 46, 47, 48, 49, 50, 51
d	11, 12, 13
m	11, 13

Live variables per line are given below:

Line number	*Live variables*	*Count*
12	d	1
13	d,m	2
14	–	0
15	–	0
16	–	0
17	rad	1
18	rad	1
19	rad, x_c, y_c	3
20	rad, x_c, y_c,x	4
21	rad, x_c, y_c, x, y	5
22	rad, x_c, y_c, x, y	5
23	rad, x_c, y_c, x, y, p_0	6
24	x_c, y_c, x, y, p_0	5
25	x_c, y_c, x, y, p_0	5
26	x_c, y_c, x, y, p_0	5
27	x_c, y_c, x, y, p_0	5
28	x_c, y_c, x, y, p_0	5
29	x_c, y_c, x, y, p_0	5
30	x_c, y_c, x, y, p_0	5
31	x_c, y_c, x, y, p_0	5
32	x_c, y_c, x, y, p_0	5
33	x_c, y_c, x, y, p_0	5
34	x_c, y_c, x, y, p_0	5
35	x_c, y_c, x, y, p_0	5

(Contd.)...

36	x_c, y_c, x, y, p_0	5
37	x_c, y_c, x, y	4
38	x_c, y_c, x, y	4
39	x_c, y_c, x, y	4
40	x_c, y_c, x, y	4
41	x_c, y_c, x, y	4
42	x_c, y_c, x, y	4
43	x_c, y_c, x, y	4
44	x_c, y_c, x, y	4
45	x_c, y_c, x, y	4
46	x_c, y_c, x, y	4
47	x_c, y_c, x, y	4
48	x_c, y_c, x, y	4
49	x_c, y_c, x, y	4
50	x_c, y_c, x, y	4
51	x_c, y_c, x, y	4
52		0
	Total	153

$$\overline{LV} = \text{Average number of live variables}$$

$$= \frac{\text{Sum of count of live variables}}{\text{Count of executable statements}}$$

$$= \frac{153}{41} = 3.73$$

$$\gamma = \text{Average life of variables}$$

$$= \frac{\text{Sum of count of live variables}}{\text{Number of unique variables}}$$

$$= \frac{153}{9} = 17$$

$$\text{Program weakness} = \overline{LV} \times \gamma$$

$$= 3.73 \times 17 = 63.41.$$

6.3.4 The Sharing of Data Among Modules

As discussed earlier, a program normally contains several modules and share coupling among modules. However, it may be desirable to know the amount of data being shared among the modules.

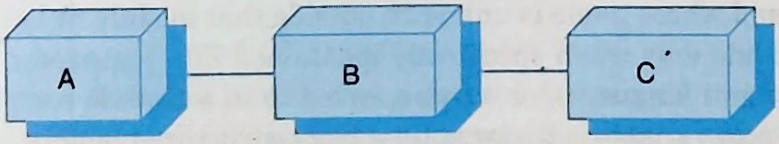


Fig. 6.10: Three modules from an imaginary program

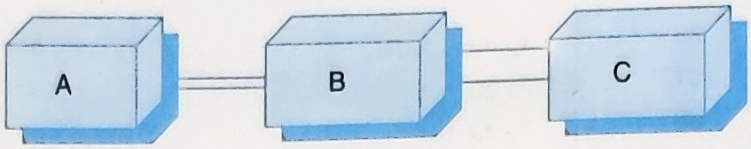


Fig. 6.11: "Pipes" of data shared among the modules

For example, Fig. 6.10 shows a schematic of a program consisting of the three subprograms A, B, and C. If we include information that represents the amount of data "passed" among the subprograms, we could envision pipes between the subprograms. The diameter of each pipe could represent the quantity or volume, of data sent from one subprogram to be used in the other. Fig. 6.11 shows the same three subprograms with pipes representing data shared between A and B and between B and C. Note that the A-B shared data is implicitly less than the B-C shared data. Fig. 6.12 shows a pictorial representation of the data shared between the main program of bubble and procedure swap both from Fig. 6.6 in bubble, the main program invokes the procedure swap in order to get it to swap the ith and $(i + 1)$ th members of the a-vector and the b-vector.

Here, we will introduce metrics that can be used for measuring this concept of sharing of data between modules. Keep in mind that the "bigger the pipe" in between any two modules, the more complex is their relationship. In theory, every module in a program is related to every other module.

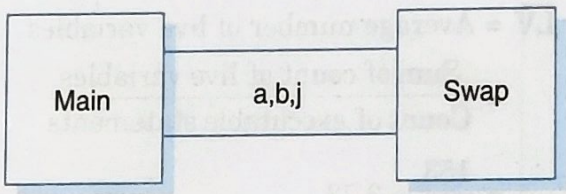


Fig. 6.12: The data shared in program bubble

If this were not true, the modules would not have been put into the same program. But, in practice, some modules may share no data directly with another, as shown by modules A and C in Fig. 6.11 The data structure employed in A should have little effect on module C, while the data structure employed in B is obviously important to C since they have such a large pipe between them.

The relationship between modules is simple if two variables are passed between them. However, a module with long list of parameters of different types, and some "global" data structure, which is shared, with several modules should be more difficult to construct or comprehend.

We assume that a global variable is one that is available to any and all modules in a program. Most programming languages allow the declaration of variables that can be accessed anywhere in the program. Contrast a global variable with a local variable, which is declared in a specific module, and whose name is unknown outside that module. A local variable is available to another module only when specifically mentioned in a parameter list passed to that module (in a procedural language), or when referred to in a module completely contained in the one where the local variable is declared (in a block-structured language). Global variables are not so limited, and are known and usable everywhere in the program.

6.4 INFORMATION FLOW METRICS

The other set of metrics we would like to consider are generally known as 'Information Flow' (IF) metrics. The basis of IF metrics is found upon the following premise. All but the simplest systems consist of components, and it is the work that these components do and how they are fitted together that influences the complexity of a system. If a component has to do numerous discrete tasks, it is said to lack 'cohesion'. If it passes information to, and/or accepts information from many other components within the system, it is said to be highly 'coupled'. Systems theory tells us that components that are highly coupled and that lack cohesion tend to be less reliable and less maintainable than those that are loosely coupled and that are cohesive. The following are the working definitions of the terms used above:

Component : Any element identified by decomposing a (software) system into its constituent parts.

Cohesion : The degree to which a component performs a single function.

Coupling : The term used to describe the degree of linkage between one component to others in the same system.

This systems-view map to software systems is extremely easy to understand as most engineers today use, or are at least familiar with, top down design techniques that produce a hierarchical view of system components. Even the more modern 'middle out' design approaches produce this structured type of deliverable, and here again IF metrics can be used.

Information flow metrics model the degree of cohesion and coupling for a particular system component. How that model is constructed can justifiably range from the simple to the complex. We intend to start with the most simple representation of IF metrics to illustrate the basic concepts, how to derive information using the metrics and how to use that information.

In terms of applying IF metrics to software systems, the pioneering work was done by Henry and Kafura [HENR81]. They looked at the UNIX operating system, and found a strong association between the IF metrics and the level of maintainability ascribed to components by developers. Other individuals who tried to apply these principles found difficulties in using the Henry and Kafura approach. Further work was done in the UK by Professor Darrell Ince and Martin Shepperd [INCE89], among others, which resulted in a more practical IF model. This work was complimented by Barbara Kitchenham [KITC90], who addressed the same problem, and who also presented a clear approach to the question of interpretation.

6.4.1 The Basic Information Flow Model

Information flow metrics are applied to the components of a system design. Fig. 6.13 shows a fragment of such a design, and for component 'A' we can define three measures, but remember that these are the simplest models of IF.

1. 'FAN IN' is simply a count of the number of other components that can call, or pass control, to Component A.

2. 'FANOUT' is the number of components that are called by component A.

3. This is derived from the first two by using the following formula. We will call this measure the INFORMATION FLOW index of component A, abbreviated as IF(A).

$$IF(A) = [FAN\ IN(A) \times FAN\ OUT\ (A)]^2.$$

The formula includes a power component to 'model the non-linear nature of complexity' as most texts of IF metrics describe it. (The assumption is that if something is more complex than something else, then resultant is much more complex). Given that assumption, we could raise to a power three or four or whatever we want but, on the principle that the simpler the model the better, then two is a good enough choice. In our view, raising to two makes it easier, as will be seen, to pick out the potential bad guys [GOOD93].

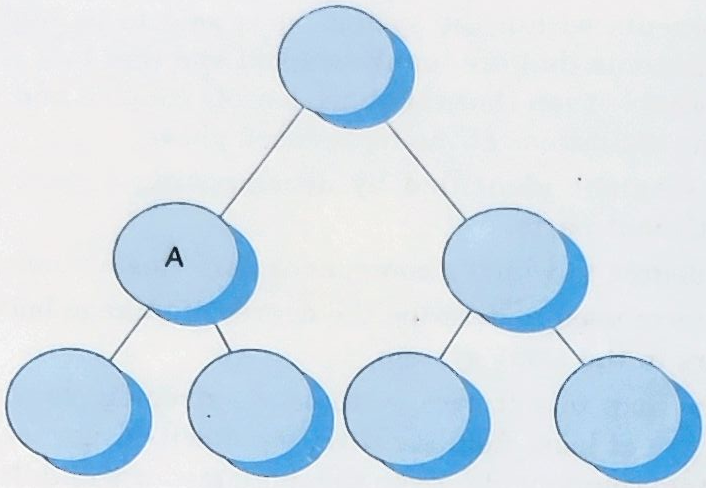


Fig. 6.13: Aspects of complexity

Given functional decomposition it will be seen that there is one additional attribute possessed by each component, namely its level in the decomposition. The following is a step-by-step guide to deriving these most simple of IF metrics.

1. Note the level of each component in the system design.
2. For each component, count the number of calls to that component — this is the FAN IN of that component. Some organizations allow more than one component at the highest level in the design, so for components at the highest level which should have a FAN IN of zero, assign a FAN IN of one. Also note that a simple model of FAN IN can penalize reused components.
3. For each component, count the number of calls from that component. For components that call no other, assign a FAN OUT value of one.
4. Calculate the IF value for each component using the above formula.
5. Sum the IF value for all components within each level which is called as the LEVEL SUM.
6. Sum the IF values for the total system design which is called the SYSTEM SUM.
7. For each level, rank the components in that level according to FAN IN, FAN OUT and IF values. Three histograms or line plots should be prepared for each level.
8. Plot the LEVEL SUM values for each level using a histogram or line plot.

This may sound like a great deal of work, but for most commercial systems, provided the documentation exits, this data can be derived and the analysis done within one engineering day. If the systems are larger, then it will obviously take longer, but remember that once done, it is very easy to keep up-to-date. Depending upon the environment it may even be possible to automate the calculations.

Having got the information, we now need to utilize it. It must be realized that, for IF metrics, there are no absolute values of good or bad. Information Flow metrics are relative indicators. This means that the value for one system may be higher than the other system, but this does not mean that one system is worse. Nor does a high metric value guarantee that a component will be unreliable and un-maintainable. It is only that it will probably be less reliable and maintainable than its fellows.

The rub is that, in most systems, less reliable and less maintainable means that it is potentially going to cost significant amounts of money to fix and enhance. Potentially, it could even be a nightmare component.

A nightmare component is the one that the system administrator has nightmares about, because he or she knows that if anyone touches that component, the whole system is going to crash, and it will take weeks to fix because designers have already left and no one is available to guide.

So the strength of IF metrics is not in the numbers themselves, but in how the information is used. As a guide, 25 per cent of components with the highest scores for FAN IN, FAN OUT and IF values should be investigated. Now in practice it may well be that a certain number of components stick out like a sore thumb, especially on the IF values. If this group is more or less than the 25 per cent guide, then do not worry about it, concentrate on those that seem to be odd according to the metric values rather than following any 25 per cent rule slavishly.

High FAN IN values indicate components that lack cohesion. It may well be that the functions have not been broken out to a great enough degree. Basically, these components are often called because they are doing more than one job.

High levels of FAN OUT also indicate a lack of cohesion or missed levels of abstraction. Here design was stopped before it was finished, and this is reflected in the high number of calls from the component. Generally speaking, FAN OUT appears to be a better indicator of problem components than FAN IN, but it is early days yet and we would not wish to discount FAN IN.

High IF values indicate highly coupled components. These components need to be looked at in terms of FAN IN and FAN OUT to see how to reduce the complexity level. Sometimes a 'traffic centre' may be hit. This is a component where, for whatever reasons, there is a high IF value, but things cannot be improved. Switching components often exhibit this. Here there is a potential problem area which, it is also a large component, may be very error prone. If the complexity cannot be reduced, then at least make sure that component is thoroughly tested.

Looking at the LEVEL SUM plot of values, we should see a fairly smooth curve showing controlled growth in IF across the levels. Sudden increases in these values across levels can indicate a missed level of abstraction within the general design. For systems where the design has less than ten levels, then a simple count of components at each level seems to work equally well.

The final item of information flow metrics is the SYSTEM SUM value. This gives an overall complexity rating for the design in terms of IF metrics. Most presentations on this topic will say that this number can be used to assess alternative design proposals.

6.4.2 A More Sophisticated Information Flow Model

We have looked at the most simple form of IF metrics, but the original proposals put forward by Henry and Kafura [HENR81] were more sophisticated than the control flow-based variant discussed above. As mentioned earlier, Ince and Shepperd [INCE89] and Kitchenham [KITC90] have done a great deal of work to help in the practical application of Henry and Kafura's pioneering proposals, and it is a distillation of that work, that has been summarized by Goodman [GOOD93] into the more sophisticated IF model. It should, however, be realized that this is a model, and it will need to be tailored to one's organization's design mechanisms before it is used. Such a tailoring process should not take more than two days for counting rule derivation and documentation of these rules, provided a well-defined design notation is used together with a competent engineer who knows that notation.

The only difference between the simple and the sophisticated IF models lies in the definition of FAN IN and FAN OUT.

For a component A let:

a = the number of components that call A.

b = the number of parameters passed to A from components higher in the hierarchy;

c = the number of parameters passed to A from components lower in the hierarchy;

d = the number of data elements read by component A.

Then:

$$\text{FAN IN(A)} = a + b + c + d$$

Also let:

e = the number of components called by A;

f = the number of parameters passed from A to components higher in the hierarchy;

g = the number of parameters passed from A to components lower in the hierarchy;

h = the number of data elements written to by A.

Then:

$$\text{FAN OUT (A)} = e + f + g + h$$

Other than those changes to the basic definitions, the derivation, analysis and interpretation remain the same. It is advisable for any organization starting to apply IF metrics to build up confidence by using the simpler form. If these work, then leave it at that. If, and only if, the simpler form fails in environment, in other words we feel confident that no significant relationship exists between the simple measures and the levels of reliability and maintainability, then spend the effort to tailor and pilot the more sophisticated form.

It is encouraging to know that there have been a number of experimental validations of IF metrics that seem to support the claims made for them. Programming groups that have been introduced to IF metrics have been able to make use of them and they also report benefits in the area of design, quality control and system management. They seem to work, but there appears to be some reluctance in the industry as a whole to make use of IF metrics. Perhaps one reason is that managers feel they are a bit 'techie'. Perhaps others feel that they are not yet ready to use sophisticated techniques like IF metrics.

6.5 OBJECT ORIENTED METRICS

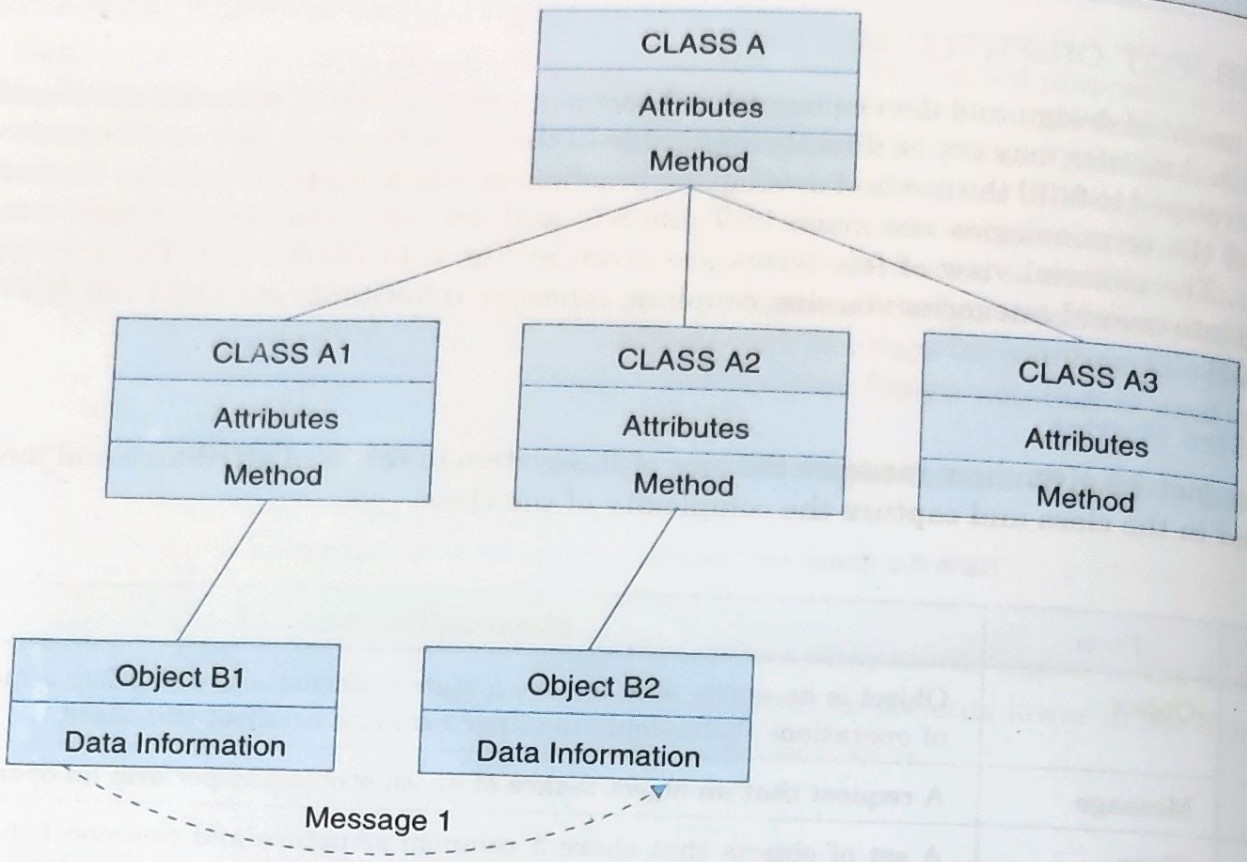
Object oriented design and development has become a popular way of software development. Traditional metrics may not be directly applicable in this area. Hence, a new set of metrics has been developed to fulfil the needs of developers, practioners, researchers and quality controllers. Some of the terminologies are given in Table 6.5, and are very common in object oriented metrics. The pictorial view of few terms are given in Fig. 6.14 [ROSE 97]. The metrics are divided into several categories viz. size, coupling, cohesion, inheritance etc. [ARV106, SHYA91, SHYA94].

6.5.1 Size Metrics

The size metrics given here measure the size of the system in terms of attributes and methods included in the class and capture the complexity of the class.

Table 6.5 Some terminologies used in object oriented metrics

S. No.	*Term*	*Meaning / Purpose*
1.	Object	Object is an entity able to save a state (information) and offers a number of operations (behaviour) to either examine or affect this state.
2.	Message	A request that an object makes of another object to perform an operation.
3.	Class	A set of objects that share a common structure and common behaviour manifested by a set of methods; the set serves as a template from which object can be created.
4.	Method	An operation upon an object, defined as part of the declaration of a class.
5.	Attribute	Defines the structural properties of a class and unique within a class.
6.	Operation	An action performed by or on an object, available to all instances of class, need not be unique.
7.	Instantiation	The process of creating an instance of the object and binding or adding the specific data.
8.	Inheritance	A relationship among classes, wherein an object in a class acquires characteristics from one or more other classes.
9.	Cohesion	The degree to which the methods within a class are related to one another.
10.	Coupling	Object A is coupled to object B, if and only if A sends a message to B.



Noté: 1 class A1 is a child class of A and inherits attributes and methods of A.

2. Object B1 contains data and structure from class A1 and class A through inheritance.
3. Object B2 contains data and structure from class A2 and class A through inheritance.
4. Object B1 passes a message 1 to object B2. Hence class A1 is coupled to class A2 through message 1.

Fig. 6.14: Pictorial view of few object oriented terms

(*a*) **Number of attributes per class (NOA):** It counts the number of attributes defined in a class. Fig. 6.15 shows the class diagram of book information system. In this system, Number of Attributes (NOA) for publication class is 2. So NOA = 2 for publication class.

(*b*) **Number of methods per class (NOM):** It counts the number of methods defined in the class. In Fig. 6.15, class publication has two methods getdata ( ) and display ( ). Hence NOM = 2 for publication class.

(*c*) **Weighted methods per class (WMC):** The WMC is the count of sum of complexities of all methods in a class. The method complexity in measured using cyclomatic complexity. This should be normalised so that nominal complexity for a method is taken as unity. Consider a class K_1, with methods $M_1, M_2 \ldots\ldots M_n$ that are given in the class. Let $C_1, C_2 \ldots\ldots, C_n$ be the complexities of the methods. WMC is defined as :

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

If method's complexities are nominal (value = 1), then WMC = n, which is equal to number of methods.

In Fig. 6.15, WMC for book is 3, sale is 2 and publication is 2 (considering each method complexity to be unity).

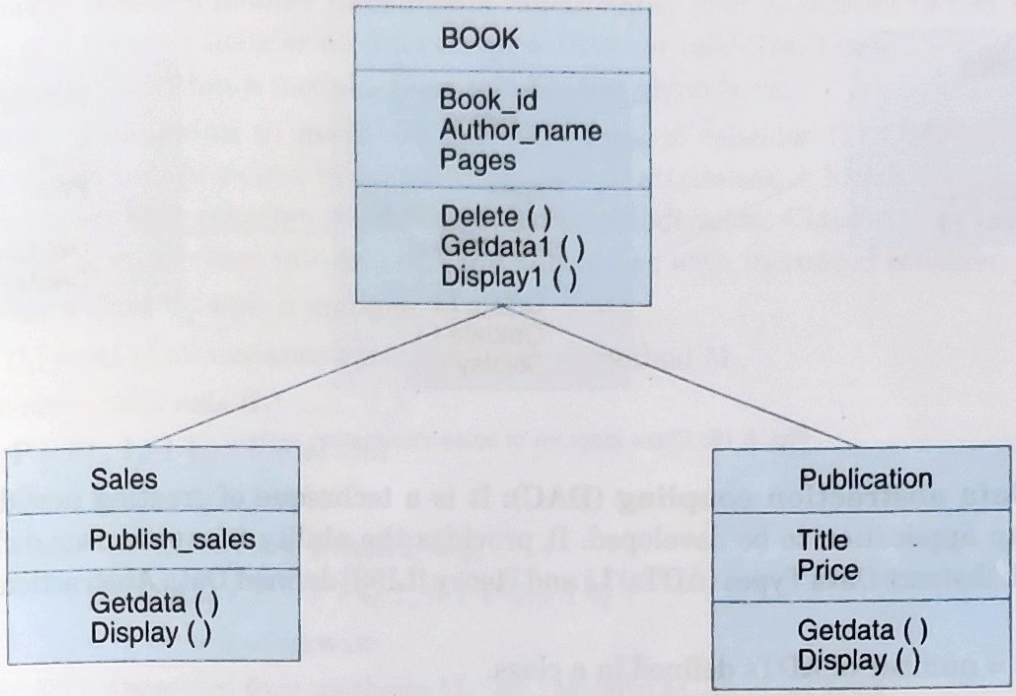


Fig. 6.15: Class diagram of book Information system

(*d*) **Response for a class (RFC):** The RFC is the count of the set of all methods that can be invoked in response to a message to an object of the class or by some methods in the class. This includes all methods accessible within class hierarchy. This metric gives us the idea about complexity of a class through number of methods and the amount of communication with other classes. The larger the number of methods that can be invoked from a class through messages, the greater the complexity of the class. If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes complicated. It requires more understanding and effort on the part of testing team. The RFC for class Book (refer Fig. 6.15) is the number of methods that can be invoked in response to messages by itself, by sales class and by publication class. Hence RFC for Book = 3 (self) + 2 (sales) + 2 (publications) = 7.

6.5.2 Coupling Metrics

Coupling relations increase complexity, reduce encapsulation, potential reuse, and limit understanding and maintainability. An improvement of modularity is achieved when inter object class couples in minimized. Evidently, the large number of couples, the higher the sensitivity to changes in other parts of design and less is the possibility of reuse of the class. Some metrics are discussed below :

(*a*) **Coupling between objects (CBO):** CBO for a class is the count of the number of other classes to which it is coupled. Two classes are coupled when methods declared in one class use methods or instance variables by the other class. The excessive coupling is detrimental to modular design and prevents reuse. The more independent a class is, the easier it is to reuse in another application. In Fig. 6.16, the Book class contains instances of the classes

publication and sales. The book class delegates its publication and sales issues to instances of the publication and sales classes. The value of metric CBO for class book is 2 and for class publication and class sales is zero.

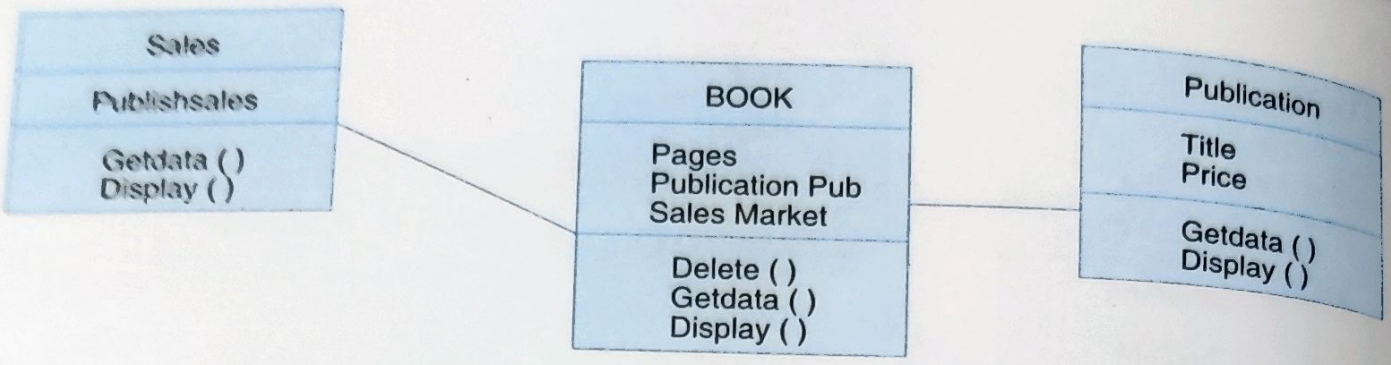


Fig. 6.16: Class diagram of sales information system

(*b*) **Data abstraction coupling (DAC):** It is a technique of creating new data types suited for an application to be developed. It provides the ability to create user defined data types called Abstract Data Types (ADTs) Li and Henry {LI93] defined Data Abstraction Coupling (DAC) as :

DAC = number of ADTs defined in a class.

In Fig. 6.16, there are two ADTs in class book, pub and market. DAC for book class is 2.

(*c*) **Message passing coupling (MPC):** Li and Henry [LI 93] defined Message Passing Coupling (MPC) metric as "number of send statements defined in a class". So, if two different methods in class A access the same method in class B, then MPC = 2. In Fig. 6.16, MPC value of class book is 4 as methods in class book calls sales :: getdata ( ), sales : : display ( ), publication : : getdata ( ), publications : : display ( ).

(*d*) **Coupling factor (CF):** Coupling can be due to message passing (dynamic coupling) or due to semantic association links (static coupling) among class instances.

It is desirable to reduce communication amongst classes and even if they communicate, very little information should be exchanged. It is defined as :

$$CF = \frac{\sum_{i=1}^{TC} \sum_{J=1}^{TC} [Is\_client\ (C_i, C_j)]}{TC^2 - TC}$$

where TC is the total number of classes.

$$Is\_client\ (C_i, C_j) = \begin{cases} 1 & \text{if } C_i \text{ and } C_j \text{ are coupled} \\ 0 & \text{otherwise} \end{cases}$$

Coupling due to the use of the inheritance is not included in CF, because a class is heavily coupled to its ancestors via inheritance. If no classrd are coupled, CF = 0%. If all classes are coupled with all other classes, CF = 100%.

6.5.3 Cohesion Metrics

Cohesion is a measure of the degree to which the elements of a module are functionally related. A strongly cohesive module implements functionality that is related to one feature of the software and requires little or no interaction with other modules. Thus, we want to maximize the interactions within a module. Four metrics are given here.

(*a*) **Lack of cohesion in methods (LCOM):** Lack of cohesion (LCOM) measures the dissimilarity of methods in a class by instance variable or attributes. A highly cohesive module should stand alone ; high cohesion indicates good class subdivision. Classes with low cohesion should probably be subdivided into two or more subclasses with increased cohesion.

Consider a class C_1 with n methods $M_1, M_2 \ldots\ldots M_n$.

Let $[I_i]$ = set of all instance variables used by method M_i.

There are n such sets $\{I_1\}, \ldots\ldots \{I_n\}$.

Let $P = \{(I_i, I_j) \mid I_i \cap I_j = \phi\}$ and

$Q = \{(I_i, I_j) \mid I_i \cap I_j \neq \phi\}$

If all n sets $\{I_1\}, \ldots, \{I_n\}$ are ϕ then $P = \phi$.

$$LCOM = |P| - |Q|, \text{ if } |P| > |Q|$$
$$= 0 \text{ otherwise}$$

In Fig. 6.17, there are four methods M_1, M_2, M_3 and M_4 in class book.

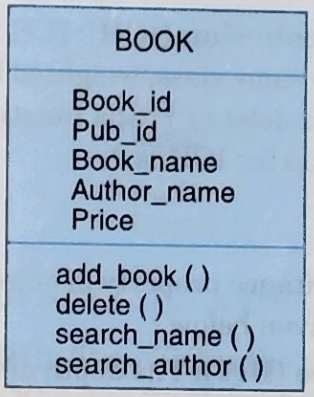


Fig. 6.17: Class diagram of class book in library management system

I_1 {add_book ( )} = {Book_id, Pub_id, Book_name, Author_name, Price}

I_2 {delete ( )} = { Book_id}

I_3 {search_name ( )} = {Book_name}

I_4 {search_author ( )} = {Author_name}

$I_1 \cap I_2$, $I_1 \cap I_3$, $I_1 \cap I_4$ are non-null sets

But $I_2 \cap I_3$, $I_2 \cap I_4$ and $I_3 \cap I_4$ are null sets

LCOM = 0, if number of null interactions are not greater than number of non-null interactions.

Hence, in this case, LCOM = 0 [$|P| = |Q| = 3$]. A positive high value of LCOM implies that classes are less cohesive. Hence, low value of LCOM is desirable.

(*b*) **Tight class cohesion (TCC):** It is defined as the percentage of pairs of public methods of the class with common attribute usage. In Fig. 6.17, methods defined access the following attributes :

add_book ( ) = {Book_id, Pub_id, Book_name, Author_name, Price}
delete ( ) = {Book_id}
Search_name ( ) = {Book_name}
Search_author ( ) = {Author_name}

All methods in class book are public. Number of pairs of methods = 6.

Methods pairs with common attribute usage
= {add_book ( ), delete ( )}, {add_book, search_name}
and {delete ( ), search_author}

Hence, $TCC = \frac{3}{6} \times 100 = 50.$

(*c*) **Loose class cohesion (LCC):** In addition to direct attributes, this measure considers attributes indirectly used by a method. Method m directly or indirectly invokes a method m', which uses attribute a. LCC is same as TCC except that this metric also considers indirectly connected methods. The LCC is defined as the percentage of pairs of public methods of the class, which are directly or indirectly connected. In Fig. 6.17, LCC for class book is same as TCC *i.e.*, 50% as there are no indirect invocations by the methods of class book.

(*d*) **Information flow based cohesion (ICH):** ICH for a class is defined as the number of invocations of other methods of the same class, weighted by the number of parameters of the invoked method. In Fig. 6.17, method delete ( ) calls function search_name, which in also the method of the same class. Hence value for ICH is 1.

6.5.4 Inheritance Metrics

These metrics are based on the inheritance property of object oriented software. These metrics are easy to calculate and some are given below :

(*a*) **Depth of inheritance tree (DIT):** The depth of a class within the inheritance hierarchy is the maximum number of steps from the class node to the root of the tree and is measured by the number of ancestor classes. In cases involving multiple inheritances, the DIT will be the maximum length from the node to the root of the tree. In Fig. 6.18, DIT for result class is 2 as it has 2 ancestor classes Internal_Exam/External_Exam and student. DIT for Internal_Exam and External_Exam is 1 as it has one ancestor class student.

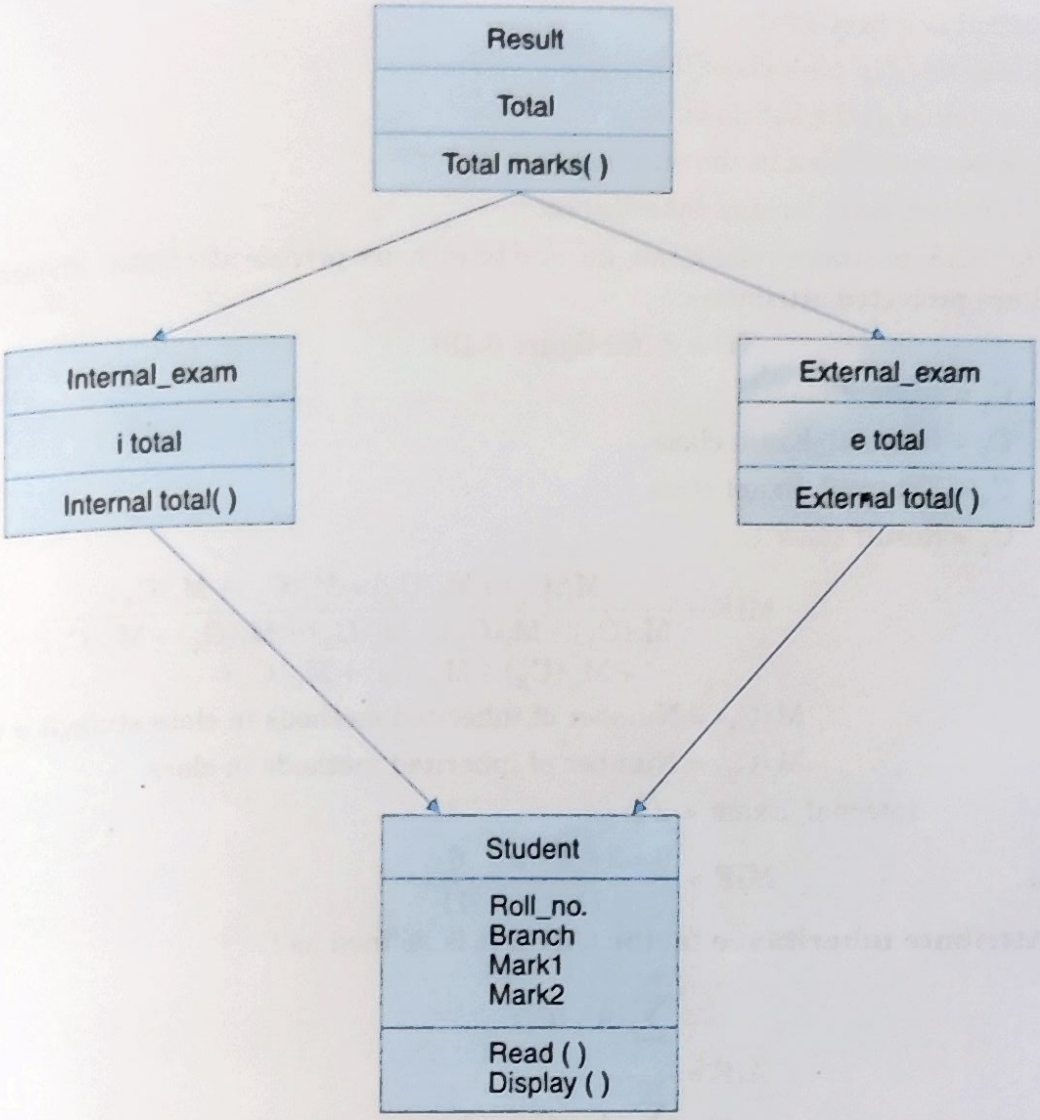


Fig. 6.18: Class diagram of result management system

(*b*) **Number of children (NOC):** The NOC is the number of immediate subclasses of a class in a hierarchy. In Fig. 6.18, NOC value for class student is 2.

(*c*) **Method inheritance factor (MIF):** It is system level metrics and is defined as :

$$MIF = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_a(C_i)}$$

where, $M_a(C_i) = M_i(C_i) + M_d(C_i)$

TC = total number of classes

$M_d(C_i)$ = Number of methods declared in a class

$M_i(C_i)$ = Number of methods inherited in a class.

A method is inherited if:

- It is defined in base class.
- It is visible in the subclass.
- It is not overridden in the subclass.

MIF is 0% for class lacking inheritance.

In Fig. 6.18, in student class, roll_no. and branch are private attributes whereas mark1 and mark2 are protected attributes.

TC = 4 (for figure 6.18)

Let C_1 = student class

C_2 = Internal_Exam class

C_3 = External_Exam class

C_4 = Result class

$$MIF = \frac{M_i(C_1) + M_i(C_2) + M_i(C_3) + M_i(C_4)}{M_i(C_1) + M_i(C_2) + M_i(C_3) + M_i(C_4) + M_d(C_1) + M_d(C_2) + M_d(C_3) + M_d(C_4)}$$

$M_i(C_1)$ = Number of inherited methods in class student = 0

$M_i(C_2)$ = Number of inherited methods in class Internal_Exam = 2

Thus, $$MIF = \frac{0+2+2+2}{11} = \frac{6}{11}$$

(*d*) **Attribute inheritance factor (AIF).** It is defined as :

$$AIF = \frac{\sum_{i=1}^{TC} A_d(C_i)}{\sum_{i=1}^{TC} A_a(C_i)}$$

where, $A_a(C_i) = A_i(C_i) + A_d(C_i)$

TC = total number of classes

$A_d(C_i)$ = number of attributes declared in a class

$A_i(C_i)$ = number of attributes inherited in a class

An attribute is inherited if:

- It is defined in the base class.
- It is visible in the subclass.
- It is not overridden in the subclass.

AIF is 0% for class lacking inheritance.

For the figure 12.7, TC = 4

Let C_1 = student

C_2 = Internal_Exam

C_3 = External_Exam

C_4 = Result

$$AIF = \frac{A_i(C_1) + A_i(C_2) + A_i(C_3) + A_i(C_4)}{A_i(C_1) + A_i(C_2) + A_i(C_3) + A_i(C_4) + A_d(C_1) + A_d(C_2) + A_d(C_3) + A_d(C_4)}$$

$A_i(C_1)$ = Number of inherited attributes in class student = 0

$A_i(C_2) = 2$, as two attributes mark1 and mark2 are inherited by subclass Internal_Exam.

Similarly, $A_i(C_3) = 2$, $A_i(C_4) = 2$

$A_d(C_1) = 4$, as four attributes are declared in class student,

Similarly, $A_d(C_2) = A_d(C_3) = A_d(C_4) = 1$

Thus, $$AIF = \frac{0+2+2+2}{13} = \frac{6}{13}.$$

6.6 USE-CASE ORIENTED METRICS

The use-case diagrams and use cases have become very popular techniques during software requirements analysis and specifications. They also provide foundations for software design activities. Some of the metrics are:

6.6.1. Counting Actors

The simplest is the number of actors in a use case model. More the actors more complex would be the system under consideration. The actors may also be further categorized as simple, average or complex, depending on their activities. Shinji Kusumoto et. al. [KUSUO4] defined weighting factor for each type of actors and is given in Table 6.6.

Table 6.6: Actor weighting factors

Type	*Description*	*Factor*
Simple	Program interface	1
Average	Interactive or protocol driven interface	2
Complex	Graphical Interface	3

A simple actor represents another system with a defined Interface. An average actor is either another system that interacts through a protocol such as TCP/IP or it is a person interacting through a text based interface (such as an old ASCII terminal). A complex actor is a person interacting through a GUI interface. The actors weight can be calculated by adding these values together.

6.6.2 Counting Use Cases

Use case count is also very simple metric, but gives us idea about size and complexity of the software under consideration.

A use case is categorized as simple, average or complex. The basis of this decision is the number of transactions in a use case, including alternate paths. A simple use case has 3 or fewer transactions, an average use case has 4–7 transactions and a complex use case has more than 7 transactions as given in Table 6.7 [KUSUO4].

Table 6.7: Transaction-based weighting factors

Type	*Description*	*Factor*
Simple	3 or fewer transactions	5
Average	4 to 7 transactions	10
Complex	Mone than 7 transactions	15

The number of each use case type is counted in the software under consideration and then each number is multiplied by a weighting factor as shown in Table 6.7. Finally, use case weight is calculated by adding these values together.

This is a new area and more research is required to develop new matrics. The effort estimation using use cases is very important area however much work is required to be carried out to design a widely acceptable model.

6.7 WEB ENGINEERING PROJECT METRICS

World wide web (WWW) is expanding day-by-day and influencing every one of us very heavily and deeply. It will be interesting to know and design metrics for various aspects of web applications. Some of the web attributes are size of the web, its connectivity, visibility of sites and the distribution of information. Few metrics are given below :

6.7.1 Number of Static Web Pages

Many web applications use static pages. The user has no control over the contents of a static page. Normally, static pages are simple and easy to construst. This metric gives us the idea about size and complexity of the web application.

6.7.2 Number of Dynamic Web Pages

A dynamic page is different from a static page ; where user actions prepares the customised contents and display on the page. Hence contents are dependent on the action (s) of the user. Dynamic pages are very common in most of the web applications. These pages are more complex and require more effort than static pages. This metric also provides the idea about size, complexity and effort to construct the web application.

6.7.3 Number of Internal Page Links

The links are basically connections that provide a connectivity to some other web pages within the web application. If link count is more, complexity will be more and web page dependency factor will increase. It is expected to have less number of connections for a good web application.

6.7.4 Word Count

The word count metric is the total words on a page. This provides the idea about the contents on a page. This may range from low to extra high.

6.7.5 Web Page Similarity

Web page similarity metrics measure the extent of relatedness between two or more web pages. We may classify similarity metrics into content-based, link-based and usage based metrics. Content based similarity is measured by comparing the text of documents. Pages with similar contents may be considered related and designated in the same group. Link based measures rely on the hyperlink structure of a web graph to obtain related pages. Usage based similarity is based on patterns of document access. The intent is to group pages or even users into meaningful groups that can aid in better organisation and accessibility of websites.

6.7.6 Web Page Search and Retrieval

These are metrics for evaluating and comparing the performance of web search and retrieval services. These may be used to measure performance of search engines. One of the measure is time taken to search a web page and retrieval of desired information. The effectiveness of a search activity is also dependent on the quality of retrieved information. Although it is not easy to measure the quality of the content due to its dependency on the perceptions of a user.

6.7.7 Number of Static Content Objects

This includes static text based objects such as graphics, audio/video information, pictures in the web application. Many content objects may appear in a single web page.

6.7.8 Number of Dynamic Content Objects

The dynamic content objects are dependent on the actions of a user. This may include graphics, audio/video information, pictures etc. which are generated in a customised web application. Many content objects may appear in a single web page.

The design of web metrics is an upcoming area for characterizing and quantifying information on the web. We may see good number of metrics in the coming years to quantify various characteristics of web.

6.8 METRICS ANALYSIS

There is a wide range of techniques that we can use to assess internal attributes of the software product. All of these techniques produce data and, all this data can be expressed in numerical form. But collecting data should be seen as only the first stage of software assessment. We also have to analyze the data to make deductions about the quality of the software product and determine if it is sufficiently good to be released to customers.

The quantity of data we collect will often be quite large. Many textual, data structural and test coverage metrics are defined at the component level. If we collect, say 20 metrics for each component and we have 100 components in a system, then we will have 2000 data points, excluding metrics defined at the subsystem and system levels. It would be difficult to imagine

a software assessor simply gazing at pages of figures and rationally arriving at a pass/fail decision for that particular product. We need to use statistics to understand the numbers, to make deductions and then produce evidence to support those deductions, In many cases simply expressing the data in pie charts and histograms can reveal a great deal about the software. Nevertheless, if we wish to uncover the relationships between metrics to validate theories, methods like regression and correlation will be more appropriate. However, when applying any kind of statistics we need to be very careful. Software metrics data is often considered to be unusual in that it does not conform to the normally made assumptions on which many statistical tests and methods are based. However, this does not preclude the meaningful application of statistical techniques. We have many tests at our disposal, which can accommodate other distributional assumptions. It is therefore, important for the researcher to determine the specific statistical tests and methods needed for each analysis in turn. Furthermore, analyzing failure data needs a very particular type of statistics and there is a range of models, which are specifically used to predict future reliability. Nevertheless, there are limitations to the kind of predictions we can make about reliability [MART94].

6.8.1 Using Statistics for Assessment

The primary purpose of applying statistics to metrics data is to gain understanding. Therefore the choice of statistics we use will ultimately be determined by the audience for whom the statistics are intended. If we are providing an evaluation of a software product for project managers then generally we will want to use simple descriptive statistics. We can use simple graphs and tables to point out areas of high and low software quality and make comparisons with other projects or predefined target values.

We will also want to use the metrics data collected from different projects to define and refine the criteria on which the assessments are made. This means deciding the ranges of values that the metrics ought to have. Such an activity is essentially internal to the Quality Assurance department or test laboratory. In this case there is no need to restrict us to simple descriptive statistics, although these will still have a role. There are other more powerful techniques such as regression, correlation and multivariate techniques.

The advent of commercial statistical packages over the last few years has meant that people who have little or no knowledge of the underlying mathematics can readily use statistics. Many of the complex calculations that in the past had to be done manually or hard programmed into specific tools are no longer a concern. One particular facility provided by many such packages is the ability to generate graphs and diagrams automatically. Some packages are compatible with word processors so that these diagrams can be 'cut and pasted' into assessment reports.

The fact that statistical packages are so easy to use gives rise to the danger of applying inappropriate techniques. We still need to understand the assumptions on which a particular technique is based. If these are ignored then the conclusions are likely to be erroneous. It is all too easy to produce plausible-looking diagrams or correlations that are, in reality, totally meaningless.

Statistics is a large body of knowledge and it would certainly not be possible to describe all the methods that would conceivably be used with software metrics. We do describe here a range of techniques that have been specifically used in the field. These are:

Summary statistics such as mean, median, maximum and minimum; *graphical representations* such as histograms, pie charts and box plots; Principal components analysis to reduce the number of variables being analyzed; Regression and correlation techniques for uncovering relationships in the data; *Reliability* models for predicting future reliability — which are very different from the other techniques.

We want to see how each of these techniques can be used to interpret particular metrics data. But before we show how we can actually apply the statistics we explore why software metrics are different from other types of data on which statistics have traditionally been applied.

6.8.2 Problems with Metrics Data

Most statistical methods and tests make a number of assumptions about the data being analyzed. For example, the use of F-test in testing the significance of simple least-square regression (fitting a straight line to two variables) assumes that the data for both variables is at least interval and that the errors are approximately normally distributed. Both of these assumptions will often be false for many of the metrics we consider and so we should check carefully before applying any such technique.

Normal distribution

Many statistical tests are based on the assumption that the data under analysis is drawn from a normally distributed population. The frequency distribution for normal data has a bell-shaped curve as shown in Fig. 6.19.

Many types of physical measurement data have been shown to follow this kind of distribution and for this reason it underlies many techniques. However, this is often not the case for software metrics.

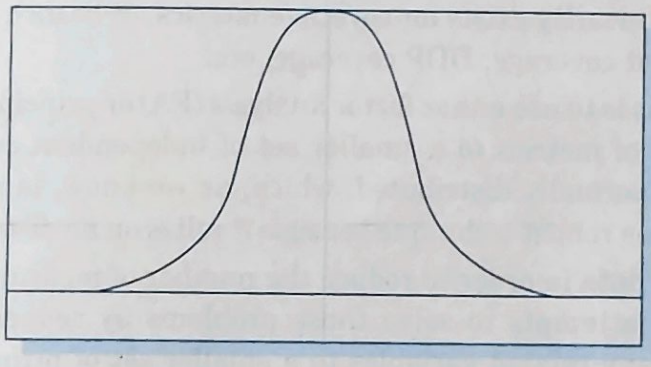


Fig. 6.19: The normal distribution

This has implications when considering testing for equivalence of mean averages from different samples. When using the t-test for this purpose it is necessary that the sample variable be normally distributed. We can also circumvent this problem by using robust techniques. These robust techniques (often called non-parametric) require few distributional assumptions or perform just as well even when their assumptions are violated.

Outliers

An outlier is a data point, which is outside the normal range of values of a given population. In non-software data, outliers are often due to errors in measurement or are caused by systematic

bias. For this reason they are frequently removed from the data set and subsequently ignored. In software metrics data, however, the outliers are often the most interesting data points. For example, if we are measuring component size, it is not uncommon to have a single software component, which is 10–20 times larger than any other component. It is precisely these large components, which may give rise to problems in maintainability and so they should certainly not be removed from the data set.

Measurement scale

The scale on which metrics are defined will generally determine which statistics can be meaningfully applied. Most physical measures are ratio, *e.g.*, length, mass, voltage, pressure. Therefore people would be forgiven for assuming the same about software metrics. This is often not the case.

To decide which scale a particular metric is defined on, we need to go back to the actual definition of the metric and reason about the relationship imposed by the attribute being measured.

Multicollinearity

Many multivariate techniques such as multivariate regression require that the variables (*i.e.*, metrics) are independent of each other. Independent in this sense means that they do not correlate with one another. Unfortunately most static flow graph and textual metrics are correlated with size. It does not mean that these metrics all measure size-they measure many different attributes like number of decisions or nesting-but these attributes tend to be highly correlated with size and by implication with each other. This is because a component with a high number of decisions or a high level of nesting is also likely to be large. The same phenomenon of multicollinearity exists for coverage metrics, so branch coverage will be highly correlated with statement coverage, DDP coverage, etc.

The solution to this is to use either factor analysis (FA) or principal components analysis (PCA) to reduce the set of metrics to a smaller set of independent components. FA usually requires the data to be normally distributed, which, as we know, is rarely going to be true. PCA on the other hand is a robust technique because it relies on no distributional assumptions.

PCA is applied to data in order to reduce the number of measures used and to simplify correlation patterns. It attempts to solve these problems by reducing the dimensionality represented by these many related variables to a smaller set of principal components while retaining most of the variation from the original variables. These new principal components are uncorrelated and can act as substitutes for the original variables with little loss of information.

6.8.3 The Common Pool of Data

We often want to compare metrics from one software product with typical values observed in many other products. This requires creating a common pool of data from previous projects. If metrics are defined at the system level then we will have one value per software project. However, metrics defined at finer levels of granularity will contribute to the common pool in varying degrees. The pool of metric values defined at the component level will be influenced

more by those software projects, which have a greater number of components. When establishing the common pool we need to be aware of three points.

1. The selection of projects should be representative and not all come from a single application domain or development styles. The exception is when a QA department with a fixed range of developments or a defined process uses the common pool.
2. No single very large project should be allowed to dominate the pool; this is less likely as the number of projects increases.
3. For some projects, certain metrics may not have been collected; we should ensure this is not a source of bias.

Only when these three points are satisfied can we be sure about making comparisons with the common pool.

6.8.4 A Pattern for Successful Applications

Successful applications of metrics abound but are not much talked about in the public literature. Mature metrics can help us to predict expected number of latent bugs, help us to decide how much testing is enough and how much design effort, cost, elapsed time, and all the rest we expect from metrics. Here's what it takes to have a success [BEIZ90].

1. *Any Metric Is Better Than None:* Use simplest possible metric like weight of program listings first and then implement "token counting" and "function counting" as the next step. Worry about the fancy metrics later.

2. *Automation Is Essential:* Any metrics project that relies on having the developers fill out long questionnaires or manually calculate metric values is doomed. They never work. If they work once, they won't the second time. If we've learned anything, it's that a metric whose calculation isn't fully automated isn't worth doing; it probably harms productivity and quality more than any benefit we can expect from metrics.

3. *Empiricism Is Better Than Theory:* Theory is at best a guide to what makes sense to include in a metrics project — there's no theory sufficiently sound today to warrant the use of a single, specific metric above others. Theory tells us what to put into the empirical pot. It's the empirical, statistical data that we must use for guidance.

4. *Use Multifactor Rather Than Single Metrics:* All successful metrics programs use a combination (typically linear) of several different metrics with weights calculated by regression analysis.

5. *Don't Confuse Productivity Metrics with Complexity Metrics:* Productivity is a characteristic of developers and testers. Complexity is a characteristic of programs. It's not always easy to tell them apart. Examples of productivity metrics incorrectly used in lieu of complexity metrics are: number of shots it takes to get a clean compilation, percentage of components that passed testing on the first attempt, number of test cases required to find the first bug. There's nothing wrong with using productivity metrics as an aid to project management, but that's a whole different story. Automated or not, a successful metrics program needs developer cooperation. If complexity and productivity are mixed up, be prepared for either or both to be sabotaged to uselessness.

6. *Let Them Mature:* It takes a lot of projects and a long time for metrics to mature to the point where they're trustworthy. If the typical project takes 12 months, then it takes ten to fifteen projects over a 2–3 year period before the metrics are any use at all.

7. *Maintain Them:* As design methods change, as testing gets better, and as QA functions to remove the old kind of bugs, the metrics based on that past history lose their utility as a predictor of anything. The weights given to metrics in predictor equations have to be revised and continually reevaluated to ensure that they continue to predict what they are intended to predict.

8. *Let Them Die:* Metrics wear out just like test suites do, for the same reason—the pesticide paradox. Actually, it's not that the metric itself wears out but that the importance we assign to the metric changes with time. We have seen projects go down the tubes because of worn-out metrics and the predictions based on them.

REFERENCES

[ALBR79] Albrecht A., "*Measuring Application Development Productivity*", Proc. IBM Application Development Symposium, Monterey, California Oct 14–17, 1979.

[ALBR83] Albrecht A., and Gaffney J.E., "*Software Function Source Lines of Code and Development Effort Prediction: A Software Science Validation*", IEEE Trans. Software Engineering, SE-9 639–648, 1983.

[ARVI06] Arvinder Kaur, "*Development of Techniques for Good Quality Object Oriented Software*", Ph.D. Thesis, University School of Information Technalogy, Guru Gobind Singh Indraprastha University, Delhi, India 2006.

[AGGA94] Aggarwal K.K. and Yogesh Singh, "*A Modified Approach for Software Science Measures*", ACM SIGSOFT Software Engineering Notes, USA, July, 1994.

[BACH90] Bache and Monica, "*Measures of Testability as a Basis for Quality Assurance*", Software Engineering Journal, March, PP-86–92, 1990.

[BAIL81] Bailey J.W., and Basili V.R.," *A meta Model for Software Development Resource Expenditures*", Proc. of the Int. Conf. on Software Engineering, 107–116, 1981.

[BASI75] Basili V.R. and Turner A.J., "*Iterative Enhancement: a Practical Technique for Software Development*," IEEE Trans. On Software Engg., SE-1, 390–396, Dec. 1975.

[BEIZ90] Boris Beizer, "*Software Testing Techniques*", Van Nostrand Reinhald International Co. Ltd., UK, 1990.

[BOEH81] Boehm B., "*Software Engineering Economics*", Prentice-Hall, Englewood Cliffs, NJ. 1981.

[BRUC92] Bruci I. Blum,"*Software Engineering — A Holistic View*", Oxford University Press, NY, 1992.

[BULU73] Bulut N., "*Invariant Proprties of Algorithms*", Ph.D. Thesis, Purdue University, August, 1973.

[CARD87] Card D.N. and Agresti W.W., "*Resolving Software Science Anomoly*", Journal of Systems and Software, Vol.7, 29–35, 1987.

[CHHA01] Chhabra Jitender Kumar, Aggarwal K.K., Yogesh Singh, "*Computing Program Weakness using Module Coupling*", ACM SIGSOFT Software Engineering Notes, Vol. 27, No. 1, January, 63–66, 2002.

[CONT86] Conte S.K., Dunsmore H.E., Shen V.Y., "*Software Engineering Metrics and Models*", The Benjamin/Cummings Pub. C. Inc., California, USA, 1986.

[CHHA2K] Chhabra Jitender Kumar, Dinesh Chutani, Aggarwal K.K., Yogesh Singh, "*Effect of Data Coupling on Program Weakness*", International Conference on Quality, Reliability and IT at the Turn of the Millennium, New Delhi, Dec. 2000.

[DUNS79] Dunsmore H.E. and Gannon J.D., "*Analysis of The Effects of Programming Factors on Programming Effort*", Journal of systems and software, 141–153, 1980.

[EJIO91] "*Software Engineering with Formal Metrics*", QED Information Sciences, Wellesley, Massachusetts 1991.

[ELSH76] Elshoff J.L., "*An Analysis of Some Commercial PI/1 Programs*", IEEE Transactions on Software Engineering", SE-2, 113–120, June, 1976.

[FENT04] N.E. Fenton, "*Software Metrics* (2 W)" Thomson Books, 2004.

[FITZ78] Fitzisimmory A. and Love T., "*A Review and Evaluation of Software Science*", ACM Computing Surveys, Vol 10, March, 1978.

[GHEZ94] Carlo Ghezzi et. al. "*Software Engineering*", PHI, 1994.

[GOOD93] Paul Goodman, "*Practical Implementation of Software Metrics*", McGraw Hill Book Company, UK, 1993.

[HALS77] Halstead M.H., "*Elements of Software Science*", New York, Elsevier North Holland, 1977.

[HAME85] Hamer P. and Frewin G., "*Software Metrics: A Critical Overview*", Pergamon Infotech State of the art report 13 (2), 1985.

[HENR81] Henry S. and Kafura D., "*Software Structure Metrics Based on Information Flow*", IEEE Trans. on Software Engineering SE-7, 5, 510–518, Sept 1981.

[INCE89] Ince D., "*Software Metrics: Measurement for Software Control and Assurance*", New York: Elsevier, 1989.

[JAME78] Elshoff J.L., "*An Inrestigation in to the Effects of the Counting Method Used on Software Science Measurements*," ACM SIGPLAN Notices, Vol 13, 30–45, Feb, 1978.

[JENS85] Jensen H.A. and Vairavan K., "*An Experimental Study of Software Metrics for Real Time Software*", IEEE Trans, on Software Engineering, 231–234, Feb, 1985.

[KITC90] Kitchenham B., "*Empirical Studies of Assumption Underlying Software Cost Estimation Models*", Proc. of European COCOMO User Group, 1990.

[KUSU04] S. Kusumoto et. al., "*Estimating Effort by Use Case Points: Method, Tool and Case Study*", Proc. of the 10th International Symposium on Software Metrics, 1530–1435/04, 2004.

[LI93] W Li and S. Henry "*Object Oriented Metrics that Predict Maintainability*", Journal of Systems Software, 23, 111—122, 1993.

[MART94] Martin Neh, "*Software Metrics for Product Assessment*", McGraw Hill Book Co., UK, 1994.

[MATS94] Matson E.M., et al., "*Software Development Cost Estimation Using Function Points*", IEEE Trans. on software Engineering", Vol 20, No.4. April, 1994.

[MEHN86] Mehnadiralta B., and Grover P.S., "*Measuring Computer Programs*", Proc. of CSI Annual Convention, India, 1986.

[RAMA88] Ramamurthy B. and Melton A., "*A Synthesis of Software Science Measures and the Cyclomatic Number*," IEEE Trans. on Software Engineering Vol. 14. No.8, August, 1988.

[ROSE97] Rosenberg L.H., "*Applying and Interpreting Object Oriented Metrics*", SATC Project of NASA on Object Oriented Matrics, USA, 1997.

[RUBE68] Rubey R.J. and R.D. Hartwick, "*Quantitative Measurement of Program Quality*", Proc. ACM Nat. Conf. PP 671–677, 1968.

[SESH61] Sheshu S. and Recd M.B., "*Linear Graphs and Electrical Networks*", Addision Wesely, USA, 1961.

[SHEN83] Shen V.Y., Conte S.D., and Dun Smore H.E." *Software Science Revisited: A Critical Analysis of the Theory and its Empirical Support*", IEEE Trans. on Software Engg., Vol. SE-9 No.2., 1983,155–165, March, 1983.

[STEP95] Stephen Treble and Neil Douglas, "*Sizing and Estimating Software in Practice*", McGraw Hill Book Company, London, 1995.

[SHYA91] Shyam R. Chidamber and Kemerer C.F., "*Towards a Metrics Suite for Object Oriented Design*", ACM OOPSLA, 171–211, 1991.

[SHYA94] Shyam R. Chidamber and Kemerer C.F., "*A Metrics Suite for Object Oriented Design*", IEEE Transactions on Software Engineering, vol. 20, No. 6, 476-493, June 1994.

[STRO67] Stroud J.M., "*The Fine Structure of Psychological Time*", Annuals of New York Academy of Science 138, 2, 623–631, 1967.

[THEB83] Thebaut S.M., "*The Saturation Effect in Large Scale Software Development Its Impact and Control*", Ph.D. Thesis, Department of Computer Science, Purdue University, West Lafeyette, IN, May, 1983.

[TRAC88] Tracz W., "*Software Reuse Emerging Technologies*", IEEE Computer Society Press, Washington DC, 1988.

[YOGE95] Singh Yogesh, "*Metrics and Design Techniques for Reliable Software*", Ph.D Thesis, Kurukshetra University, Kurukshetra (India) , July, 1995.

[YOGE98] Singh Yogesh and Pradeep Bhatia, "*Module Weakness — A New Measure*", ACM SIGSOFT Software Engineering Notes, 81, July 1998.

MULTIPLE CHOICE QUESTIONS

Note: *Choose most appropriate answer of the following questions.*

6.1. Which one is not a category of software metrics ?

(*a*) product metrics (*b*) process metrics

(*c*) project metrics (*d*) people metrics.

6.2. Software science measures are developed by

(*a*) M. Halstead (*b*) B. Littlewood

(*c*) T.J. McCabe (*d*) G. Rothermal.

6.3. Vocabulary of a program is defined as:

(*a*) $\eta = \eta_1 + \eta_2$ (*b*) $\eta = \eta_1 - \eta_2$

(*c*) $\eta = \eta_1 \times \eta_2$ (*d*) $\eta = \eta_1 / \eta_2$.

6.4. In Halstead theory of software science, volume is measured in bits. The bits are

(*a*) number of bits required to store the program

(*b*) actual size of a program if a uniform binary encoding scheme for vocabulary is used

(*c*) number of bits required to execute the program

(*d*) none of the above.

6.5. In Halstead theory, effort is measured in

(*a*) parson-months (*b*) hours

(*c*) elementary mental discriminations (*d*) none of the above

6.6. Language level is defined as
(a) $\lambda = L^3V$ (b) $\lambda = LV$
(c) $\lambda = LV^*$ (d) $\lambda = L^2V$

6.7. Program weakness is
(a) $WM = \overline{LV} \times \gamma$ (b) $WM = \overline{LV} / \gamma$
(c) $WM = \overline{LV} + \gamma$ (d) none of the above.

6.8. 'FAN IN' of a component A is defined as
(a) count of the number of components that can call, or pass control, to component A
(b) number of components related to component A
(c) number of components dependent on component A
(d) none of the above.

6.9. 'FAN OUT' of a component A is defined as
(a) number of components related to component A
(b) number of components dependent on component A
(c) number of components that are called by component A
(d) none of the above.

6.10. Which is not a size metric?
(a) LOC (b) function count
(c) program length (d) cyclomatic complexity.

6.11. Which one is not a measure of software science theory?
(a) vocabulary (b) volume
(c) level (d) logic

6.12. A human mind is capable of making how many number of elementary mental descriminations per second (*i.e.*, stroud number)?
(a) 5–20 (b) 20–40
(c) 1–10 (d) 40–80

6.13. Minimal implementation of any algorithm was given the following name by Halstead:
(a) volume (b) potential volume
(c) effective volume (d) none of the above.

6.14. Program volume of a software product is
(a) $V = N \log_2 n$ (b) $V = (N/2) \log_2 n$
(c) $V = 2N \log_2 n$ (d) $V = N \log_2 n + 1$

6.15. Which one is the international standard for size measure?
(a) LOC (b) function count
(c) program length (d) none of the above.

6.16. Which one in not on object oriented metric ?
(a) RFC (b) CBO
(c) DAC (d) OBC.

6.17. Which metric also considers indirect connected methods ?
(a) TCC (b) LCC
(c) both of the above (d) none of the above.

6.18. Depth of inheritance tree (DIT) can be measured by:

(*a*) number of ancestor classes (*b*) number of successor classes

(*c*) number of failure classes (*d*) number of root classes.

6.19. A dynamic page is:

(*a*) where contants are not dependent on the actions of the user.

(*b*) where contents are dependent on the actions of the user

(*c*) where contents cannot be displayed

(*d*) none of the above.

6.20. Which one is not a size measure ?

(*a*) LOC

(*b*) FP

(*c*) cyclomatic complexity

(*d*) program length.

EXERCISE

6.1. Define software metrics. Why do we really need metrics in software?

6.2. Discuss the areas of applications of software metrics. What are the problems during implementation of metrics in any organization ?

6.3. What are various categories of software metrics ? Discuss with the help of suitable examples.

6.4. Explain the Halstead theory of software science. Is it significant in today's scenario of component based software development ?

6.5. What is the importance of language level in Halstead theory of software science?

6.6. Give Halstead's software science measures for:

(*i*) program length (*ii*) program volume

(*iii*) program level (*iv*) effort

(*v*) language level.

6.7. For a program with number of unique operators $\eta_1 = 20$ and number of unique operands $\eta_2 = 40$, compute the following:

(*i*) program volume (*ii*) effort and time

(*iii*) program length (*iv*) program level.

6.8. Develop a small software tool that will perform a Halstead analysis on a programming language source code of your choice.

6.9. Write a program in C and also PASCAL for the calculation of the roots of a quadratic equation. Find out all software science metrics for both the programs. Compare the outcomes and comment on the efficiency and size of both the source codes.

6.10. How should a procedure identifier be considered, both when declared and when called? What about the identifier of a procedure that is passed as a parameter to another procedure?

6.11. It is interesting to examine how the ratio $(N-\hat{N}) / N$ varies when a program is divided into parts. Actually, some experiments show that the partitioning of vocabularies due to program modularization maintains the stability of the ratio $(N-\hat{N})/N$. Two extreme situations may occur:

a. η_1 and η_2 are the same for all parts.

b. η_1 and η_2 are partitioned into disjoint subsets by modularization.

Compute the variations of $\hat{N}$ for a program with $N = \hat{N} = 72$, $\eta_1 = 4$ and $\eta_2 = 16$ when the program is divided into two parts, under the two extreme assumptions.

Warning: It may be difficult to divide a program in such a way so as to satisfy the latter assumption. (At least one procedure definition and call must share an identifier.) For large values of the quantities involved, however, we can assume that, at least, η_1 and η_2 have small intersections with respect to their size [GHEZ 94].

6.12. Define data structure metrics. How can we calculate amount of data in a program ?

6.13. Describe the concept of module weakness. Is it applicable to programs also ?

6.14. Write a program for the calculation of roots of a quadratic equation. Generate cross reference list for the program and also calculate $\overline{LV}$, γ and WM for this program.

6.15. Show that the value of SP at a particular statement is also the value of LV at that point.

6.16. Discuss the significance of data structure metrics during testing.

6.17. What are information flow metrics ? Explain the basic information flow model.

6.18. Discuss the problems with metrics data. Explain two methods for the analysis of such data.

6.19. Show why and how software metrics can improve the software process. Enumerate the effect of metrics on software productivity.

6.20. Why does lines of code (LOC) not measure software nesting and control structures?

6.21. Several researchers in software metrics concentrate on data structure to measure complexity. Is data structure a complexity or quality issue, or both?

6.22. List the benefits and disadvantages of using Library routines rather than writing own code.

6.23. Compare software science measures and function points as measures of complexity. Which do you think more useful as a predictor of how much particular software's development will cost?

6.24. Some experimental evidence suggests that the initial size estimate for a project affects the nature and results of the project. Consider two different managers charged with developing the same application. One estimates that the size of the application will be 50,000 lines, while the other estimates that it will be 100,000 lines. Discuss how these estimates affect the project throughout its life cycle.

6.25. Which one is the most appropriate size estimation techinque and why?

6.26. Discuss the object oriented metrics. What is the importance of metrics in object oriented software development?

6.27. Define the following:
RFC, CBO, DAC, TCC, LCC and DIT.

6.28. What is the significance of use case metrics? Is it really important to design such metrics?