

Module

11

# Software Project Planning

# Lesson 27

## Project Planning and Project Estimation Techniques

## Specific Instructional Objectives

At the end of this lesson the student would be able to:

- Identify the job responsibilities of a software project manager.
- Identify the necessary skills required in order to perform software project management.
- Identify the essential activities of project planning.
- Determine the different project related estimates performed by a project manager and suitably order those estimates.
- Explain what is meant by Sliding Window Planning.
- Explain what is Software Project Management Plan (SPMP).
- Identify and explain two metrics for software project size estimation.
- Identify the shortcomings of function point (FP) metric.
- Explain the necessity of feature point metric in the context of project size estimation.
- Identify the types of project-parameter estimation technique.

## Responsibilities of a software project manager

Software project managers take the overall responsibility of steering a project to success. It is very difficult to objectively describe the job responsibilities of a project manager. The job responsibility of a project manager ranges from invisible activities like building up team morale to highly visible customer presentations. Most managers take responsibility for project proposal writing, project cost estimation, scheduling, project staffing, software process tailoring, project monitoring and control, software configuration management, risk management, interfacing with clients, managerial report writing and presentations, etc. These activities are certainly numerous, varied and difficult to enumerate, but these activities can be broadly classified into project planning, and project monitoring and control activities. The project planning activity is undertaken before the development starts to plan the activities to be undertaken during development. The project monitoring and control activities are undertaken once the development activities start with the aim of ensuring that the development proceeds as per plan and changing the plan whenever required to cope up with the situation.

## Skills necessary for software project management

A theoretical knowledge of different project management techniques is certainly necessary to become a successful project manager. However, effective software project management frequently calls for good qualitative judgment and decision taking capabilities. In addition to having a good grasp of the latest software project management techniques such as cost estimation, risk management, configuration management, project managers need good communication skills and the ability get work done. However, some skills such as tracking and

controlling the progress of the project, customer interaction, managerial presentations, and team building are largely acquired through experience. None the less, the importance of sound knowledge of the prevalent project management techniques cannot be overemphasized.

## Project planning

Once a project is found to be feasible, software project managers undertake project planning. Project planning is undertaken and completed even before any development activity starts. Project planning consists of the following essential activities:

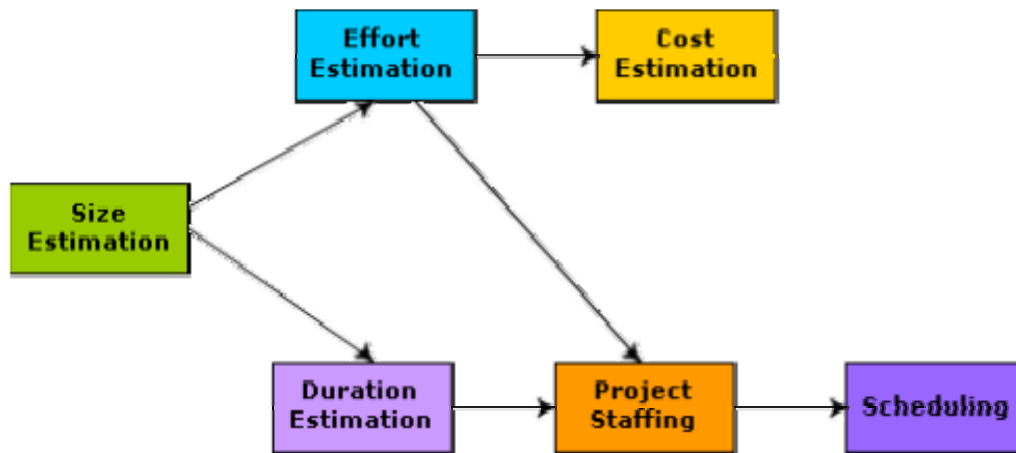
- Estimating the following attributes of the project:
  - Project size:** What will be problem complexity in terms of the effort and time required to develop the product?
  - Cost:** How much is it going to cost to develop the project?
  - Duration:** How long is it going to take to complete development?
  - Effort:** How much effort would be required?

The effectiveness of the subsequent planning activities is based on the accuracy of these estimations.

- Scheduling manpower and other resources
- Staff organization and staffing plans
- Risk identification, analysis, and abatement planning
- Miscellaneous plans such as quality assurance plan, configuration management plan, etc.

## Precedence ordering among project planning activities

Different project related estimates done by a project manager have already been discussed. Fig. 11.1 shows the order in which important project planning activities may be undertaken. From fig. 11.1 it can be easily observed that size estimation is the first activity. It is also the most fundamental parameter based on which all other planning activities are carried out. Other estimations such as estimation of effort, cost, resource, and project duration are also very important components of project planning.



**Fig. 11.1:** Precedence ordering among planning activities

## Sliding Window Planning

Project planning requires utmost care and attention since commitment to unrealistic time and resource estimates result in schedule slippage. Schedule delays can cause customer dissatisfaction and adversely affect team morale. It can even cause project failure. However, project planning is a very challenging activity. Especially for large projects, it is very much difficult to make accurate plans. A part of this difficulty is due to the fact that the proper parameters, scope of the project, project staff, etc. may change during the span of the project. In order to overcome this problem, sometimes project managers undertake project planning in stages. Planning a project over a number of stages protects managers from making big commitments too early. This technique of staggered planning is known as Sliding Window Planning. In the sliding window technique, starting with an initial plan, the project is planned more accurately in successive development stages. At the start of a project, project managers have incomplete knowledge about the details of the project. Their information base gradually improves as the project progresses through different phases. After the completion of every phase, the project managers can plan each subsequent phase more accurately and with increasing levels of confidence.

## Software Project Management Plan (SPMP)

Once project planning is complete, project managers document their plans in a Software Project Management Plan (SPMP) document. The SPMP document should discuss a list of different items that have been discussed below. This list can be used as a possible organization of the SPMP document.

Organization of the Software Project Management Plan (SPMP) Document

## **1. Introduction**

- (a) Objectives
- (b) Major Functions
- (c) Performance Issues
- (d) Management and Technical Constraints

## **2. Project Estimates**

- (a) Historical Data Used
- (b) Estimation Techniques Used
- (c) Effort, Resource, Cost, and Project Duration Estimates

## **3. Schedule**

- (a) Work Breakdown Structure
- (b) Task Network Representation
- (c) Gantt Chart Representation
- (d) PERT Chart Representation

## **4. Project Resources**

- (a) People
- (b) Hardware and Software
- (c) Special Resources

## **5. Staff Organization**

- (a) Team Structure
- (b) Management Reporting

## **6. Risk Management Plan**

- (a) Risk Analysis
- (b) Risk Identification
- (c) Risk Estimation
- (d) Risk Abatement Procedures

## **7. Project Tracking and Control Plan**

## **8. Miscellaneous Plans**

- (a) Process Tailoring
- (b) Quality Assurance Plan
- (c) Configuration Management Plan

- (d) Validation and Verification
- (e) System Testing Plan
- (f) Delivery, Installation, and Maintenance Plan

## Metrics for software project size estimation

Accurate estimation of the problem size is fundamental to satisfactory estimation of effort, time duration and cost of a software project. In order to be able to accurately estimate the project size, some important metrics should be defined in terms of which the project size can be expressed. The size of a problem is obviously not the number of bytes that the source code occupies. It is neither the byte size of the executable code. The project size is a measure of the problem complexity in terms of the effort and time required to develop the product.

Currently two metrics are popularly being used widely to estimate size: lines of code (LOC) and function point (FP). The usage of each of these metrics in project size estimation has its own advantages and disadvantages.

### Lines of Code (LOC)

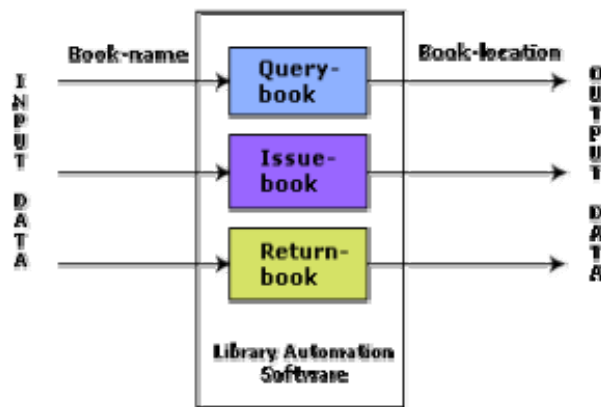
LOC is the simplest among all metrics available to estimate project size. This metric is very popular because it is the simplest to use. Using this metric, the project size is estimated by counting the number of source instructions in the developed program. Obviously, while counting the number of source instructions, lines used for commenting the code and the header lines should be ignored.

Determining the LOC count at the end of a project is a very simple job. However, accurate estimation of the LOC count at the beginning of a project is very difficult. In order to estimate the LOC count at the beginning of a project, project managers usually divide the problem into modules, and each module into submodules and so on, until the sizes of the different leaf-level modules can be approximately predicted. To be able to do this, past experience in developing similar products is helpful. By using the estimation of the lowest level modules, project managers arrive at the total size estimation.

### Function point (FP)

Function point metric was proposed by Albrecht [1983]. This metric overcomes many of the shortcomings of the LOC metric. Since its inception in late 1970s, function point metric has been slowly gaining popularity. One of the important advantages of using the function point metric is that it can be used to easily estimate the size of a software product directly from the problem specification. This is in contrast to the LOC metric, where the size can be accurately determined only after the product has fully been developed.

The conceptual idea behind the function point metric is that the size of a software product is directly dependent on the number of different functions or features it supports. A software product supporting many features would certainly be of larger size than a product with less number of features. Each function when invoked reads some input data and transforms it to the corresponding output data. For example, the issue book feature (as shown in fig. 11.2) of a Library Automation Software takes the name of the book as input and displays its location and the number of copies available. Thus, a computation of the number of input and the output data values to a system gives some indication of the number of functions supported by the system. Albrecht postulated that in addition to the number of basic functions that a software performs, the size is also dependent on the number of files and the number of interfaces.



**Fig. 11.2:** System function as a map of input data to output data

Besides using the number of input and output data values, function point metric computes the size of a software product (in units of functions points or FPs) using three other characteristics of the product as shown in the following expression. The size of a product in function points (FP) can be expressed as the weighted sum of these five problem characteristics. The weights associated with the five characteristics were proposed empirically and validated by the observations over many projects. Function point is computed in two steps. The first step is to compute the unadjusted function point (UFP).

$$\text{UFP} = (\text{Number of inputs}) * 4 + (\text{Number of outputs}) * 5 + \\ (\text{Number of inquiries}) * 4 + (\text{Number of files}) * 10 + \\ (\text{Number of interfaces}) * 10$$

**Number of inputs:** Each data item input by the user is counted. Data inputs should be distinguished from user inquiries. Inquiries are user commands such as print-account-balance. Inquiries are counted separately. It must be noted that individual data items input by the user are not considered in the calculation of the number of inputs, but a group of related inputs are considered as a single input.

For example, while entering the data concerning an employee to an employee pay roll software; the data items name, age, sex, address, phone number, etc. are together considered as a single input. All these data items can be considered to be related, since they pertain to a single employee.

**Number of outputs:** The outputs considered refer to reports printed, screen outputs, error messages produced, etc. While outputting the number of outputs the individual data items within a report are not considered, but a set of related data items is counted as one input.

**Number of inquiries:** Number of inquiries is the number of distinct interactive queries which can be made by the users. These inquiries are the user commands which require specific action by the system.

**Number of files:** Each logical file is counted. A logical file means groups of logically related data. Thus, logical files can be data structures or physical files.

**Number of interfaces:** Here the interfaces considered are the interfaces used to exchange information with other systems. Examples of such interfaces are data files on tapes, disks, communication links with other systems etc.

Once the unadjusted function point (UFP) is computed, the technical complexity factor (TCF) is computed next. TCF refines the UFP measure by considering fourteen other factors such as high transaction rates, throughput, and response time requirements, etc. Each of these 14 factors is assigned from 0 (not present or no influence) to 6 (strong influence). The resulting numbers are summed, yielding the total degree of influence (DI). Now, TCF is computed as  $(0.65 + 0.01 \times DI)$ . As DI can vary from 0 to 70, TCF can vary from 0.65 to 1.35. Finally,  $FP = UFP \times TCF$ .

## Shortcomings of function point (FP) metric

LOC as a measure of problem size has several shortcomings:

- LOC gives a numerical value of problem size that can vary widely with individual coding style – different programmers lay out their code in different ways. For example, one programmer might write several source instructions on a single line whereas another might split a single instruction across several lines. Of course, this problem can be easily overcome by counting the language tokens in the program rather than the lines of code. However, a more intricate problem arises because the length of a program depends on the choice of instructions used in writing the program. Therefore, even for the same problem, different programmers might come up with programs having different LOC counts. This situation does not improve even if language tokens are counted instead of lines of code.

- A good problem size measure should consider the overall complexity of the problem and the effort needed to solve it. That is, it should consider the local effort needed to specify, design, code, test, etc. and not just the coding effort. LOC, however, focuses on the coding activity alone; it merely computes the number of source lines in the final program. We have already seen that coding is only a small part of the overall software development activities. It is also wrong to argue that the overall product development effort is proportional to the effort required in writing the program code. This is because even though the design might be very complex, the code might be straightforward and vice versa. In such cases, code size is a grossly improper indicator of the problem size.
- LOC measure correlates poorly with the quality and efficiency of the code. Larger code size does not necessarily imply better quality or higher efficiency. Some programmers produce lengthy and complicated code as they do not make effective use of the available instruction set. In fact, it is very likely that a poor and sloppily written piece of code might have larger number of source instructions than a piece that is neat and efficient.
- LOC metric penalizes use of higher-level programming languages, code reuse, etc. The paradox is that if a programmer consciously uses several library routines, then the LOC count will be lower. This would show up as smaller program size. Thus, if managers use the LOC count as a measure of the effort put in the different engineers (that is, productivity), they would be discouraging code reuse by engineers.
- LOC metric measures the lexical complexity of a program and does not address the more important but subtle issues of logical or structural complexities. Between two programs with equal LOC count, a program having complex logic would require much more effort to develop than a program with very simple logic. To realize why this is so, consider the effort required to develop a program having multiple nested loop and decision constructs with another program having only sequential control flow.
- It is very difficult to accurately estimate LOC in the final product from the problem specification. The LOC count can be accurately computed only after the code has been fully developed. Therefore, the LOC metric is little use to the project managers during project planning, since project planning is carried out even before any development activity has started. This possibly is the biggest shortcoming of the LOC metric from the project manager's perspective.

## Feature point metric

A major shortcoming of the function point measure is that it does not take into account the algorithmic complexity of a software. That is, the function point metric implicitly assumes that the effort required to design and develop any two functionalities of the system is the same. But, we know that this is normally not true, the effort required to develop any two functionalities may vary widely. It only takes the number of functions that the system supports into consideration without distinguishing the difficulty level of developing the various functionalities. To overcome this problem, an extension of the function point metric called feature point metric is proposed.

Feature point metric incorporates an extra parameter algorithm complexity. This parameter ensures that the computed size using the feature point metric reflects the fact that the more is the complexity of a function, the greater is the effort required to develop it and therefore its size should be larger compared to simpler functions.

## Project Estimation techniques

Estimation of various project parameters is a basic project planning activity. The important project parameters that are estimated include: project size, effort required to develop the software, project duration, and cost. These estimates not only help in quoting the project cost to the customer, but are also useful in resource planning and scheduling. There are three broad categories of estimation techniques:

- Empirical estimation techniques
- Heuristic techniques
- Analytical estimation techniques

### Empirical Estimation Techniques

Empirical estimation techniques are based on making an educated guess of the project parameters. While using this technique, prior experience with development of similar products is helpful. Although empirical estimation techniques are based on common sense, different activities involved in estimation have been formalized over the years. Two popular empirical estimation techniques are: Expert judgment technique and Delphi cost estimation.

#### Expert Judgment Technique

Expert judgment is one of the most widely used estimation techniques. In this approach, an expert makes an educated guess of the problem size after analyzing the problem thoroughly. Usually, the expert

estimates the cost of the different components (i.e. modules or subsystems) of the system and then combines them to arrive at the overall estimate. However, this technique is subject to human errors and individual bias. Also, it is possible that the expert may overlook some factors inadvertently. Further, an expert making an estimate may not have experience and knowledge of all aspects of a project. For example, he may be conversant with the database and user interface parts but may not be very knowledgeable about the computer communication part.

A more refined form of expert judgment is the estimation made by group of experts. Estimation by a group of experts minimizes factors such as individual oversight, lack of familiarity with a particular aspect of a project, personal bias, and the desire to win contract through overly optimistic estimates. However, the estimate made by a group of experts may still exhibit bias on issues where the entire group of experts may be biased due to reasons such as political considerations. Also, the decision made by the group may be dominated by overly assertive members.

### Delphi cost estimation

Delphi cost estimation approach tries to overcome some of the shortcomings of the expert judgment approach. Delphi estimation is carried out by a team comprising of a group of experts and a coordinator. In this approach, the coordinator provides each estimator with a copy of the software requirements specification (SRS) document and a form for recording his cost estimate. Estimators complete their individual estimates anonymously and submit to the coordinator. In their estimates, the estimators mention any unusual characteristic of the product which has influenced his estimation. The coordinator prepares and distributes the summary of the responses of all the estimators, and includes any unusual rationale noted by any of the estimators. Based on this summary, the estimators re-estimate. This process is iterated for several rounds. However, no discussion among the estimators is allowed during the entire estimation process. The idea behind this is that if any discussion is allowed among the estimators, then many estimators may easily get influenced by the rationale of an estimator who may be more experienced or senior. After the completion of several iterations of estimations, the coordinator takes the responsibility of compiling the results and preparing the final estimate.

### Heuristic Techniques

Heuristic techniques assume that the relationships among the different project parameters can be modeled using suitable mathematical expressions. Once the basic (independent) parameters are known, the other (dependent) parameters can be easily determined by substituting the value of the basic parameters in the

mathematical expression. Different heuristic estimation models can be divided into the following two classes: single variable model and the multi variable model.

Single variable estimation models provide a means to estimate the desired characteristics of a problem, using some previously estimated basic (independent) characteristic of the software product such as its size. A single variable estimation model takes the following form:

$$\text{Estimated Parameter} = c_1 * e^{d_1}$$

In the above expression,  $e$  is the characteristic of the software which has already been estimated (independent variable). *Estimated Parameter* is the dependent parameter to be estimated. The dependent parameter to be estimated could be effort, project duration, staff size, etc.  $c_1$  and  $d_1$  are constants. The values of the constants  $c_1$  and  $d_1$  are usually determined using data collected from past projects (historical data). The basic COCOMO model is an example of single variable cost estimation model.

A multivariable cost estimation model takes the following form:

$$\text{Estimated Resource} = c_1 * e_1^{d_1} + c_2 * e_2^{d_2} + \dots$$

Where  $e_1, e_2, \dots$  are the basic (independent) characteristics of the software already estimated, and  $c_1, c_2, d_1, d_2, \dots$  are constants. Multivariable estimation models are expected to give more accurate estimates compared to the single variable models, since a project parameter is typically influenced by several independent parameters. The independent parameters influence the dependent parameter to different extents. This is modeled by the constants  $c_1, c_2, d_1, d_2, \dots$ . Values of these constants are usually determined from historical data. The intermediate COCOMO model can be considered to be an example of a multivariable estimation model.

### **Analytical Estimation Techniques**

Analytical estimation techniques derive the required results starting with basic assumptions regarding the project. Thus, unlike empirical and heuristic techniques, analytical techniques do have scientific basis. Halstead's software science is an example of an analytical technique. Halstead's software science can be used to derive some interesting results starting with a few simple assumptions. Halstead's software science is especially useful for estimating software maintenance efforts. In fact, it outperforms both empirical and heuristic techniques when used for predicting software maintenance efforts.

## Halstead's Software Science – An Analytical Technique

Halstead's software science is an analytical technique to measure size, development effort, and development cost of software products. Halstead used a few primitive program parameters to develop the expressions for over all program length, potential minimum value, actual volume, effort, and development time.

For a given program, let:

- $\eta_1$  be the number of unique operators used in the program,
- $\eta_2$  be the number of unique operands used in the program,
- $N_1$  be the total number of operators used in the program,
- $N_2$  be the total number of operands used in the program.

### Length and Vocabulary

The length of a program as defined by Halstead, quantifies total usage of all operators and operands in the program. Thus, length  $N = N_1 + N_2$ . Halstead's definition of the length of the program as the total number of operators and operands roughly agrees with the intuitive notation of the program length as the total number of tokens used in the program.

The program vocabulary is the number of unique operators and operands used in the program. Thus, *program vocabulary*  $\eta = \eta_1 + \eta_2$ .

### Program Volume

The length of a program (i.e. the total number of operators and operands used in the code) depends on the choice of the operators and operands used. In other words, for the same programming problem, the length would depend on the programming style. This type of dependency would produce different measures of length for essentially the same problem when different programming languages are used. Thus, while expressing program size, the programming language used must be taken into consideration:

$$V = N \log_2 \eta$$

Here the program volume  $V$  is the minimum number of bits needed to encode the program. In fact, to represent  $\eta$  different identifiers uniquely, at least  $\log_2 \eta$  bits (where  $\eta$  is the program vocabulary) will be needed. In this scheme,  $N \log_2 \eta$  bits will be needed to store a program of length  $N$ . Therefore, the volume  $V$  represents the size of the program by approximately compensating for the effect of the programming language used.

## Potential Minimum Volume

The potential minimum volume  $V^*$  is defined as the volume of most succinct program in which a problem can be coded. The minimum volume is obtained when the program can be expressed using a single source code instruction., say a function call like `foo( ) ;`. In other words, the volume is bound from below due to the fact that a program would have at least two operators and no less than the requisite number of operands.

Thus, if an algorithm operates on input and output data  $d_1, d_2, \dots d_n$ , the most succinct program would be  $f(d_1, d_2, \dots d_n)$ ; for which  $\eta_1 = 2, \eta_2 = n$ . Therefore,  $V^* = (2 + \eta_2)\log_2(2 + \eta_2)$ .

The program level  $L$  is given by  $L = V^*/V$ . The concept of program level  $L$  is introduced in an attempt to measure the level of abstraction provided by the programming language. Using this definition, languages can be ranked into levels that also appear intuitively correct.

The above result implies that the higher the level of a language, the less effort it takes to develop a program using that language. This result agrees with the intuitive notion that it takes more effort to develop a program in assembly language than to develop a program in a high-level language to solve a problem.

## Effort and Time

The effort required to develop a program can be obtained by dividing the program volume with the level of the programming language used to develop the code. Thus, effort  $E = V/L$ , where  $E$  is the number of mental discriminations required to implement the program and also the effort required to read and understand the program. Thus, the programming effort  $E = V^2/V^*$  (since  $L = V^*/V$ ) varies as the square of the volume. Experience shows that  $E$  is well correlated to the effort needed for maintenance of an existing program.

The programmer's time  $T = E/S$ , where  $S$  the speed of mental discriminations. The value of  $S$  has been empirically developed from psychological reasoning, and its recommended value for programming applications is 18.

## Length Estimation

Even though the length of a program can be found by calculating the total number of operators and operands in a program, Halstead suggests a way to determine the length of a program using the number of unique operators and operands used in the program. Using this method, the program parameters such as length, volume, cost, effort, etc. can be determined even before the start of any programming activity. His method is summarized below.

Halstead assumed that it is quite unlikely that a program has several identical parts – in formal language terminology identical

substrings – of length greater than  $\eta$  ( $\eta$  being the program vocabulary). In fact, once a piece of code occurs identically at several places, it is made into a procedure or a function. Thus, it can be assumed that any program of length  $N$  consists of  $N/\eta$  unique strings of length  $\eta$ . Now, it is standard combinatorial result that for any given alphabet of size  $K$ , there are exactly  $K^r$  different strings of length  $r$ .

Thus.

$$N/\eta \leq \eta^n \text{ Or, } N \leq \eta^{n+1}$$

Since operators and operands usually alternate in a program, the upper bound can be further refined into  $N \leq \eta \eta_1^{n_1} \eta_2^{n_2}$ . Also,  $N$  must include not only the ordered set of  $n$  elements, but it should also include all possible subsets of that ordered sets, i.e. the power set of  $N$  strings (This particular reasoning of Halstead is not very convincing!!!).

Therefore,

$$2^N = \eta \eta_1^{n_1} \eta_2^{n_2}$$

Or, taking logarithm on both sides,

$$N = \log_2 \eta + \log_2 (\eta_1^{n_1} \eta_2^{n_2})$$

So we get,

$$N = \log_2 (\eta_1^{n_1} \eta_2^{n_2})$$

(approximately, by ignoring  $\log_2 \eta$ )

Or,

$$\begin{aligned} N &= \log_2 \eta_1^{n_1} + \log_2 \eta_2^{n_2} \\ &= n_1 \log_2 \eta_1 + n_2 \log_2 \eta_2 \end{aligned}$$

Experimental evidence gathered from the analysis of larger number of programs suggests that the computed and actual lengths match very closely. However, the results may be inaccurate when small programs when considered individually.

In conclusion, Halstead's theory tries to provide a formal definition and quantification of such qualitative attributes as program complexity, ease of understanding, and the level of abstraction based on some low-level parameters such as the number of operands, and operators appearing in the program. Halstead's software science provides gross estimation of properties of a large collection of software, but extends to individual cases rather inaccurately.

Let us consider the following C program:

The unique operators are:

The unique operands are:

Therefore,

Volume = Length\*log(23)  
= 81\*4.52  
= 366