7 Software Reliability

Software reliability. Does it exist ? Computer program instructions cannot break or wear out. Hence predicting software reliability is a different concept as compared to predicting hardware reliability. Software becomes more reliable over time, instead of wearing out. It becomes obsolete as the environment for which it was developed changes. Hardware redundancy allows us to make a system reliable as we desire, if we use large number of components with given reliability. We do not have such techniques in software and we may not get such techniques in foreseeable future.

Initially, problems in programming were blamed to the severe constraints imposed by the hardware. However, hardware has now become more reliable, flexible and versatile, but the problems with programming have not decreased.

7.1 BASIC CONCEPTS

The term reliability is often misunderstood in the software field since software does not break or wear-out in the physical sense. It either works in a given environment or it does not. Hence traditional "bath tub" curve of hardware reliability is not applicable here. The "bath tub curve" is given in Fig. 7.1.

As indicated, there are three phases in the life of any hardware component *i.e.,* burn-in, useful life and wear-out. In burn-in phase, failure rate is quite high initially, and it starts decreasing gradually as the time progresses. It may be due to initial testing in the premises of the organisation. During useful life period, failure rate is approximately constant. Failure rate increases in wear-out phase due to wearing out/aging of components. The best period is useful life period. The shape of this curve is like a "bath tub" and that is why it is known as bath tub curve.

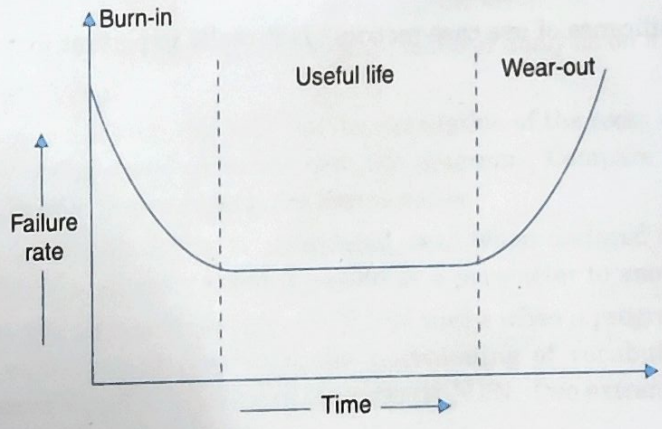


Fig. 7.1: Bath tub curve of hardware reliability

We do not have wear out phase in software. The expected curve for software is given in Fig. 7.2.

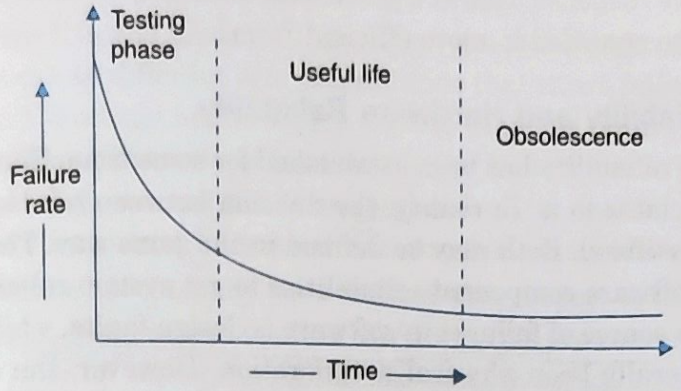


Fig. 7.2: Software reliability curve (failure rate versus time)

Software may be retired only if it becomes obsolete. Some of contributing factors are given below:

- change in environment
- change in infrastructure/technology
- major change in requirements
- increase in complexity
- extremely difficult to maintain
- deterioration in structure of the code
- slow execution speed
- poor graphical user interfaces.

7.1.1 What is Software Reliability?

According to Bev Littlewood [LITT 79]: "Software reliability means operational reliability. Who cares how many bugs are in the program ? We should be concerned with their effect on its operations".

As per IEEE standard [IEEE 90]: "Software reliability is defined as the ability of a system or component to perform its required functions under stated conditions for a specified period of time".

Software reliability is also defined as the probability that a software system fulfills its assigned task in a given environment for a predefined number of input cases, assuming that the hardware and the inputs are free of error [KOPE 79]. Hence it is the probability that the software will work without failure for a specified period of time in a given environment. Here environment and time is fixed. Reliability is for fixed time under given environment or stated conditions. So, reliability value is always for a well defined domain.

The most acceptable definition of software reliability is: "It is the probability of a failure free operation of a program for a specified time in a specified environment [MUSA87]".

For example, a time-sharing system may have a reliability of 0.95 for 10 hr when employed by the average user. This system, when executed for 10 hr, would operate without failure for

95 of these periods out of 100. As a result of the general way in which we defined failure, note that the concept of software reliability incorporates the notion of performance being satisfactory. For example, excessive response time at a given load level may be considered unsatisfactory so that a routine must be recorded in more efficient form.

7.1.2 Software Reliability and Hardware Reliability

The field of hardware reliability has been established for some time. Hence, one might ask how software reliability relates to it. In reality, the division between hardware and software reliability is somewhat artificial. Both may be defined in the same way. Therefore, one may combine hardware and software component reliabilities to get system reliability. Both depend on the environment. The source of failures in software is design faults, while the principal source in hardware has generally been physical deterioration. However, the concepts and theories developed for software reliability could really be applied to any design activity, including hardware design.

Once a software (design) defect is properly fixed, it is in general fixed for all time. Failure usually occurs only when a program (design) is exposed to an environment that it was not developed or tested for. Although manufacturing can affect the quality of physical components, the replication process for software (design) is trivial and can be performed to very high standards of quality. Since introduction and removal of design faults occurs during software development, software reliability may be expected to vary during this period.

The *design reliability* concept has not been applied to hardware to that extent. The probability of failure due to wear and other physical causes has usually been much greater than that due to an unrecognised design problem. It was possible to keep hardware design failures low because hardware was generally less complex logically than software. Hardware design failures had to be kept low because retrofitting of manufactured items in the field was very expensive. Awareness of the work that is going on in software reliability, plus a growing realisation of the importance of design faults, may now be having an effect on hardware reliability too. This growing awareness is strengthened by the parallels that people are starting to draw between software engineering and chip design.

A final characteristic of software reliability is that it tends to change continually during test periods. This happens either as new problems are introduced, when new code is written or when repair action removes problems that exist in the code. Hardware reliability may change during certain periods, such as initial burn-in or the end of useful life. However, it has a much greater tendency than software toward a constant value.

Despite the foregoing differences, we can develop software reliability theory in a way that is compatible with hardware reliability theory. Thus system reliability figures may be computed using standard hardware combinatorial techniques [SHOO86]. Hardware and software reliability share many similarities and some differences [LLOY77]. One must not err on the side of assuming that software always presents unique problems, but one must also be careful not to carry analogies too far.

7.1.3 Failures and Faults

What do we mean by the term software failure? It is the departure of the external results of program operation from requirements. So our *failure* is something dynamic. The program has

to be executing for a failure to occur. The term failure relates to the behaviour of the program. This very general definition of failure is deliberate. It can include such things as deficiency in performance attributes and excessive response time.

A *fault* is the defect in the program that, when executed under particular conditions, causes a failure. There can be different sets of conditions that cause failures, or the conditions can be repeated. Hence a fault can be the source of more than one failure. A fault is a property of the program rather than a property of its execution or behaviour. It is what we are really referring to in general when we use the term bug. A fault is created when a programmer makes an error. It's very important to make the failure-fault distinction!

Reliability quantities have usually been defined with respect to time, although it would be possible to define them with respect to other variables. We are concerned with three kinds of time. The *execution time* for a program is the time that is actually spent by a processor in executing the instructions of that program. The second kind of time is *calendar time*. It is the familiar time that we normally experience. Execution time is important, because it is now generally accepted that models based on execution time are superior. However, quantities must ultimately be related back to calendar time to be meaningful to engineers or managers. Sometimes the term *clock time* is used for a program. It represents the elapsed time from start to end of program execution on a running computer. It includes wait time and the execution time of other programs. Periods during which the computer is shut down are not counted. If computer utilisation by the program, which is the fraction of time the processor is executing the program, is constant, clock time will be proportional to execution time.

There are four general ways of characterising failure occurrences in time:

1. time of failure,
2. time interval between failures,
3. cumulative failures experienced upto a given time,
4. failures experienced in a time interval.

These are illustrated in Tables 7.1 and 7.2.

Note that all the foregoing four quantities are random variables. By *random* we mean that the values of the variables are not known with certainty. There are many possible values each associated with a probability of occurrence. For example, we don't really know when the next failure will occur. If we did, we would try to prevent or avoid it. We only know a set of possible times of failure.

There are at least two principal reasons for this randomness. First, the commission of errors by programmers, and hence the introduction of faults, is a very complex, unpredictable process. Hence the location of faults within the program are unknown. Second, the condition of what next? In addition, the relationship between program function requested and code path executed, although theoretically determinable, may not be so in practice because it is so complex. Since failures are dependent on the presence of a fault in the code and its execution in the execution of a program, these are generally unpredictable. For example, with a telephone switching system, how do you know what type of call will be made next in the context of certain machine states.

A third complicating element is thus introduced that argues for the randomness of the failure process.

Table 7.1: Time based failure specification

Failure number	*Failure time(sec)*	*Failure interval(sec)*
1	8	8
2	18	10
3	25	7
4	36	11
5	45	9
6	57	12
7	71	14
8	86	15
9	104	18
10	124	20
11	143	19
12	169	26
13	197	28
14	222	25
15	250	28

Table 7.2: Failure based failure specification

Time (sec)	*Cumulative failures*	*Failures in interval (30 sec)*
30	3	3
60	6	3
90	8	2
120	9	1
150	11	2
180	12	1
210	13	1
240	14	1

Table 7.3 illustrates a typical probability distribution of failures that occurs within a time period of execution. Each possible value of the random variable of number of failures is given alongwith its associated probability. The probabilities, of course, add to 1. Note that here the random variable is discrete, as the number of failures must be an integer. Note that the most probable number of failures is 2 for $t = 1$ hr. The mean or average number of failures can be computed. We multiply each possible value by the probability it can occur and add all the products. The mean is 3.04 failures for $t = 1$ hour.

Table 7.3: Probability distribution at times t_A and t_B

Value of random variable (failures in time period)	*Probability*	
	Elapsed time t_A = 1 hr	*Elapsed time t_B = 5 hr*
0	0.10	0.01
1	0.18	0.02
2	0.22	0.03
3	0.16	0.04
4	0.11	0.05
5	0.08	0.07
6	0.05	0.09
7	0.04	0.12
8	0.03	0.16
9	0.02	0.13
10	0.01	0.10
11	0	0.07
12	0	0.05
13	0	0.03
14	0	0.02
15	0	0.01
Mean failures	3.04	7.77

We will look at the time variation from two different viewpoints, the mean value function and the failure intensity function. The mean value function represents the average cumulative failures associated with each time point. The failure intensity function is the rate of change of the mean value function or the number of failures per unit time. For example, you might say 0.01 failure/hr or 1 failure/100 hr. Strictly speaking, the failure intensity is the derivative of the mean value function with respect to time, and is an instantaneous value.

A random process whose probability distribution varies with time is called non-homogeneous. Most failure processes during test fit this situation. Fig. 7.3 illustrates the mean value and the related failure intensity functions at time t_A and t_B. Note that the mean failures experienced increases from 3.04–7.77 between these two points, while the failure intensity decreases.

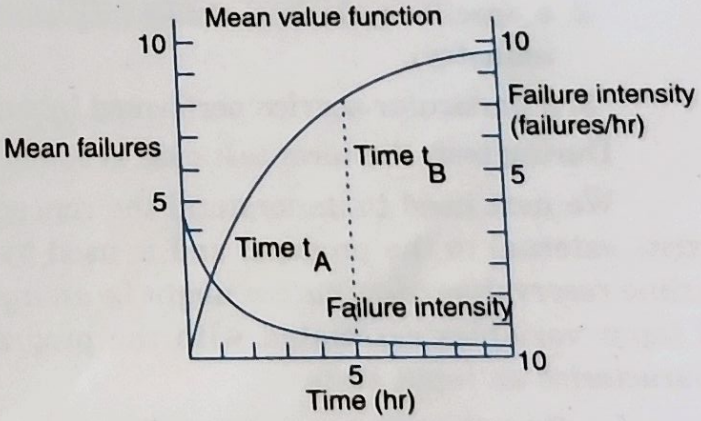


Fig. 7.3: Mean value & failure intensity functions

Failure behaviour is affected by two principal factors:

1. the number of faults in the software being executed,
2. the execution environment or the operational profile of execution.

The number of faults in software is the difference between the number introduced and the number removed.

Faults are introduced when the code is being developed by programmers. They may introduce the faults during original design or when they are adding new features, making design changes, or repairing faults that have been identified. In general, only code that is new or modified results in faults introduction. Code that is inherited from another application does not usually introduce any appreciable number of faults, except possibly in the interfaces. It generally has been thoroughly debugged in the previous application. Note that the process of faults removal introduces some new faults because it involves modification or writing of new code.

Faults removal obviously can't occur unless you have some means of detecting the fault in the first place. Thus fault removal resulting from execution depends on the occurrence of the associated failure. Occurrence depends both on the length of time for which the software has been executing and on the execution environment or operational profile. When different functions are executed, different faults are encountered and the failures that are exhibited tend to be different; thus the environmental influence. We can often find faults without execution. They may be found through inspection, compiler diagnostics, design or code reviews, or code reading.

7.1.4 Environment

Let us scrutinise the term environment. The environment is described by the operational profile. We need to build up to the concept of the operational profile through several steps. It is possible to view the execution of a program as a single entity. The execution can last for months or even years for a real time system. However, it is more convenient to divide the execution into runs. The definition of run is somewhat arbitrary, but it is generally associated with some function that the program performs. Thus, it can conveniently describe the functional environment of the program. Runs that are identical repetitions of each other are said to form a run type. The proportion of runs of various types may vary, depending on the functional environment. Examples of a run type might be:

1. a particular transaction in an airline reservation system or a business data processing system,
2. a specific cycle in a closed loop control system (for example, in a chemical process industry),
3. a particular service performed by an operating system for a user.

During test, the term test case is sometimes used instead of run type.

We next need to understand the concept of the input variable. This is a variable that exists external to the program and is used by the program in executing its function. For an airline reservation, *destination* might be an input variable. One generally has a large quantity of input variables associated with the program, and each set of values of these variables characterise an input state.

In effect, the input state identifies the particular run type that you're making. Therefore, runs can always be classified by their input states. Again, taking the case of the airline reservation system, the input state might be characterised by particular values of origin,

destination, airline, day and flight number. The set of all possible input states is known as the input space.

Similarly, an output variable is a variable that exists external to a program and is set by it. An output state is a set of values of all output variables associated with a run of a program. In the airline reservation system, an output state might be the set of values of variables printed on the ticket and on different reports used in operating the airline. It can now be seen that a failure involves a departure of the output state from what it is expected to be.

The run types required of the program by the environment can be viewed as being selected randomly. Thus, we define the operational profile as the set of run types that the program can execute alongwith possibilities with which they will occur. In Fig. 7.4, we show two of many possible input states A and B, with their probabilities of occurrence. The part of the operational profile for just these two states is shown in Fig. 7.5. In reality, the number of possible input states is generally quite large. A realistic operational profile is illustrated in Fig. 7.6. Note that the input states have been located on the horizontal axis in order of the probabilities of their occurrence. This can be done without loss of generality. They have been placed close together so that the operational profile would appear to be a continuous curve.

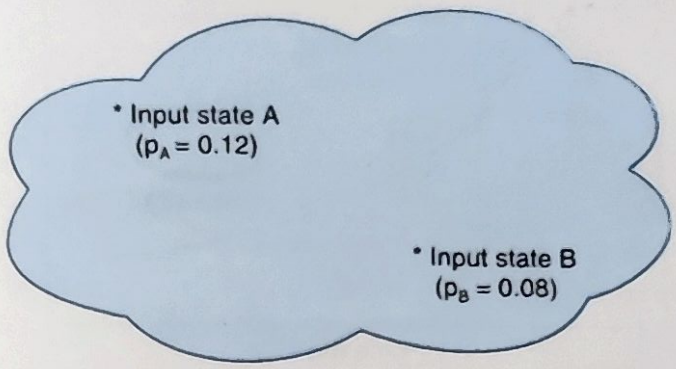


Fig. 7.4: Input Space

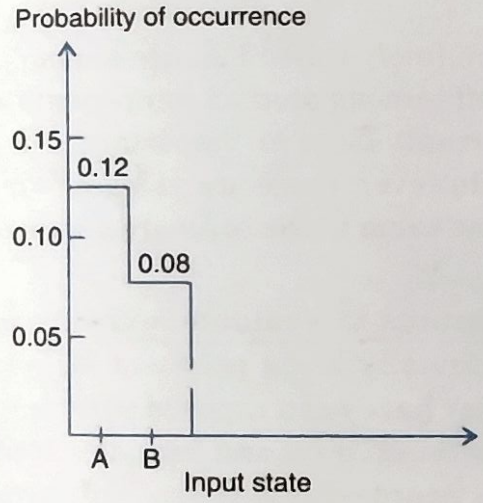


Fig. 7.5: Portion of operational profile

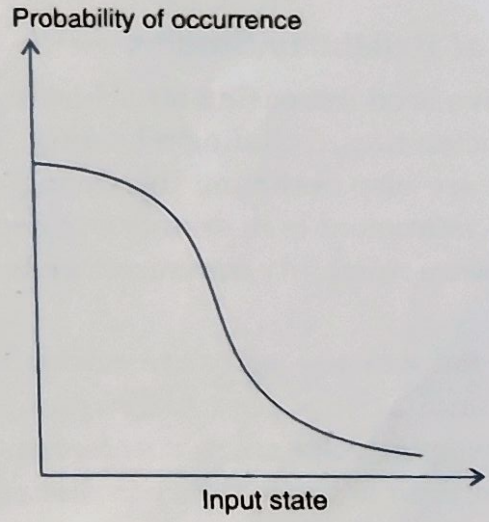


Fig. 7.6. Operational profile

Failure intensity is an alternative way of expressing reliability. We have discussed the example of the reliability of a particular system being 0.95 for 10 hr of time. An equivalent

statement could be the failure intensity of 0.05 failure/hr. Each specification has its advantages. The failure intensity statement is more economical, as you only have to give one number. However, the reliability statement is better suited to the combination of reliabilities of components to get system reliability. If the risk of failure at any point in time is of paramount concern, failure intensity may be the more appropriate measure. Such would be the case for a nuclear power plant. When proper operation of a system to accomplish some function with a time duration is required reliability specification is often best. An example would be a space flight to the moon. Fig. 7.7 shows how failure intensity and reliability typically vary during a test period, as faults are removed. Note that we define failure intensity, just like we do reliability with respect to a specified environment.

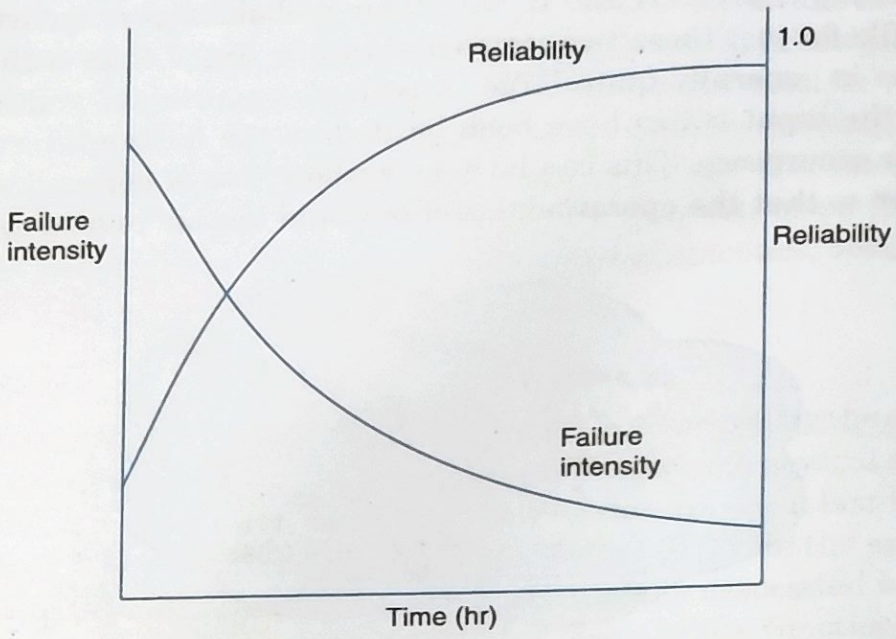


Fig. 7.7: Reliability and failure intensity

7.1.5 Uses of Reliability Studies

Pressures have been increasing for achieving a more finely tuned balance among product and process characteristics, including reliability. Trade-offs among product components with respect to reliability are also becoming increasingly important. Thus an important use of software reliability measurement is in system engineering. However, there are at least four other ways in which software reliability measures can be of great value to the software engineer, manager or user.

First, you can use software reliability measures to evaluate software engineering technology quantitatively. New techniques are continually being proposed for improving the process of developing software, but unfortunately they have been exposed to little quantitative evaluation. Because of the inability to distinguish between good and bad, new technology has often led to a general resistance to change that is counter productive. Software reliability measures offer the promise of establishing at least one criterion for evaluating the new technology. For example, you might run experiments to determine the decrease in failure intensity (failures per unit time) at the start of system test resulting from design reviews.

A quantitative evaluation such as this makes the benefits of good software engineering technology highly visible.

Second, software reliability measures offer you the possibility of evaluating development status during the test phases of a project. Methods such as intuition of designers or test team percent of tests completed, and successful execution of critical functional tests have been used to evaluate testing progress. None of these have been really satisfactory and some have been quite unsatisfactory. An objective reliability measure (such as failure intensity) established from test data provides a sound means of determining status. Reliability generally increases with the amount of testing. Thus, reliability can be closely linked with project schedules. Furthermore, the cost of testing is highly correlated with failure intensity improvement. Since two of the key process attributes that a manager must control are schedule and cost, reliability can be intimately tied in with project management.

Third, one can use software reliability measures to monitor the operational performance of software and to control new features added and design changes made to the software. The reliability of software usually decreases as a result of such changes. A reliability objective can be used to determine when, and perhaps how large, a change will be allowed. The objective would be based on user and other requirements. For example, a freeze on all changes not related to debugging can be imposed when the failure intensity rises above the performance objective.

Finally, a quantitative understanding of software quality and the various factors influencing it and affected by it enriches into the software product and the software development process. One is then much more capable of making informed decisions.

7.2 SOFTWARE QUALITY

Our objective of software engineering is to produce good quality maintainable software in time and within budget. Here quality is very important. What do we understand with the term "quality" ? It is not easy to define quality. People understand quality, appreciate quality but may not be able to clearly express the same. It is like beauty which is very much in the eyes of the beholder.

Different people understand different meanings of quality like:

- conformance to requirements
- fitness for the purpose
- level of satisfaction

If a product is meeting its requirements, we may say it is a good quality product. We expect that requirements are clearly stated and cannot be misunderstood. Everything is measured with respect to requirements and if it matches, product is a quality product. If a car is designed with a maximum speed of 150 km/hour and if it fails to achieve in the field, then it is not meeting the requirements. If two cars are designed with different style, performance and economy and both are up to standards set for them, then both are quality cars.

When user uses the product and finds the product fit for its purpose, he/she feels that product is of good quality. If product is meeting user requirements, a feeling of satisfaction may emerge and this satisfaction is nothing but the satisfaction for quality.

In a broad sense, the users' views of quality must deal with the product's ease of installation, operational efficiency, and convenience. If the product is easy to handle and we comfortably remember how to use it, then customer satisfaction level may increases.

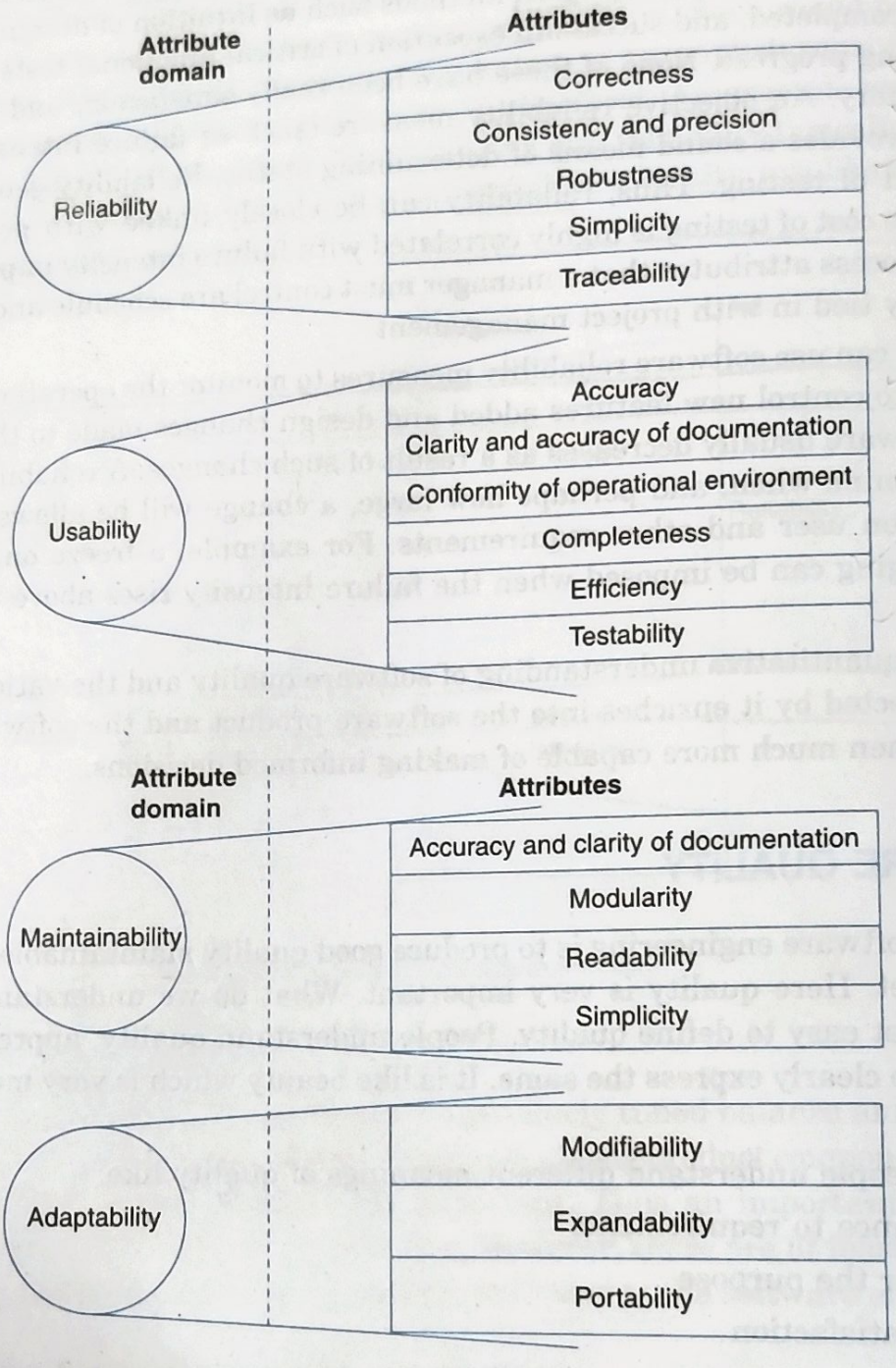


Fig. 7.8: Software quality attributes

Quality has many characteristics and some are related to each other. In software, the quality is commonly recognised as "lack of bugs" in the program. If a software has too many functional defects, then, it is not meeting its basic requirement of functionality. This is usually expressed in two ways:

(*i*) **Defect rate:** number of defects per million lines of source code, per function point or any other unit.

(*ii*) **Reliability:** generally measured as number of failures per '*t*' hours of operation, mean time to failure or probability of failure free operation in a specified time under specified environment.

When we deal with software quality, a list of attributes is required to be defined that are appropriate for software. There are four attribute domains [DUNN90], which should be defined. These are usually the ones most entrusted by the customer:

- Reliability
- Usability
- Maintainability
- Adaptability

These four attribute-domains can be divided into attributes that are more commonly understood by the software community and are given in Fig. 7.8. The details of software quality attributes are given in Table 7.4.

Table 7.4: Software quality attributes

1.	Reliability	The extent to which a software performs its intended functions without failure.
2.	Correctness	The extent to which a software meets its specifications.
3.	Consistency and precision	The extent to which a software is consistent and give results with precision.
4.	Robustness	The extent to which a software tolerates the unexpected problems.
5.	Simplicity	The extent to which a software is simple in its operations.
6.	Traceability	The extent to which an error is traceable in order to fix it.
7.	Usability	The extent of effort required to learn, operate and understand the functions of the software.
8.	Accuracy	Meeting specifications with precision.
9.	Clarity and accuracy of documentation	The extent to which documents are clearly and accurately written.
10.	Conformity of operational environment	The extent to which a software is in conformity of operational environment.
11.	Completeness	The extent to which a software has specified functions.
12.	Efficiency	The amount of computing resources and code required by software to perform a function.
13.	Testability	The effort required to test a software to ensure that it performs its intended functions.

(*Contd....*)

14.	Maintainability	The effort required to locate and fix an error during maintenance phase.
15.	Modularity	It is the extent of ease to implement, test, debug and maintain the software.
16.	Readability	The extent to which a software is readable in order to understand.
17.	Adaptability	The extent to which a software is adaptable to new platforms and technologies.
18.	Modifiability	The effort required to modify a software during maintenance phase.
19.	Expandability	The extent to which a software is expandable without undesirable side effects.
20.	Portability	The effort required to transfer a program from one platform to another platform.

The attribute domain and attributes are also named as the factor and criteria. There are many models of software quality and some are discussed here.

7.2.1 McCall Software Quality Model

McCall et. al [MCCA77] model of software quality was introduced in 1977 and many quality factors were incorporated. The model distinguishes between two levels of quality attributes. Higher level quality attributes are known as quality factors. These are external attributes and can be measured directly. The second level of quality attributes is named as quality criteria. Quality criteria can be measured either subjectively or objectively. The software quality factors are organised in three product quality factors as shown in Fig. 7.9.

Total: 11.

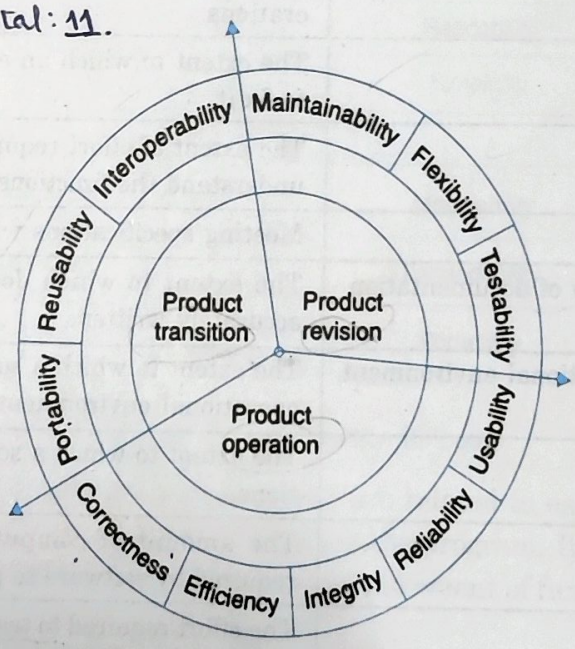


Fig. 7.9: Software quality factors

(*i*) **Product Operation:** Here, factors which are related to the operation of a product are combined. The factors are:

- Correctness
- Efficiency
- Integrity
- Reliability
- Usability.

These five factors are related to operational performance, convenience, ease of usage and its correctness. These factors play a very significant role in building customer's satisfaction.

(*ii*) **Product Revision:** The factors which are required for testing and maintenance are combined and are given below:

- Maintainability
- Flexibility
- Testability.

These factors pertain to the testing and maintainability of software. They give us idea about ease of maintenance, flexibility and testing effort. Hence, they are combined under the umbrella of product revision.

(*iii*) **Product Transition:** We may have to transfer a product from one platform to another platform or from one technology to another technology. The factors related to such a transfer are combined and are given below:

- Portability
- Reusability
- Interoperability.

Most of the quality factors are explained in Table 7.4. The remaining factors are given in Table 7.5.

Table 7.5: Remaining quality factors (others are in Table 7.4)

Sr. No.	*Quality factor*	*Purpose*
1.	Integrity	The extent to which access to software or data by the unauthorised persons can be controlled.
2.	Flexibility	The effort required to modify an operational program.
3.	Reusability	The extent to which a program can be reused in other applications.
4.	Interoperability	The effort required to couple one system with another.

Quality criteria

The second level of quality attributes are termed as quality criteria. We have eleven quality factors and each quality factor has many second level of quality attributes which are shown in Fig. 7.10. Second level attributes are internal attributes. However, users and managers are interested in the higher level external quality attributes. For example, we may not directly

not measure the reliability of a software system. We may however, directly measure the number of defects encountered so far. This direct measure can be used to obtain insight into the reliability of the system. This involves a theory of how the number of defects encountered relates to reliability, which can be ascertained on good grounds. For most other aspects of quality though, the relation between the attributes that can be measured directly and the external attributes we are interested in is less obvious, to say the least [VLIE 02]. The relationship between quality factors and quality criteria are given in Table 7.5(a). The definitions and details of these quality criteria are discussed in Table 7.5(b).

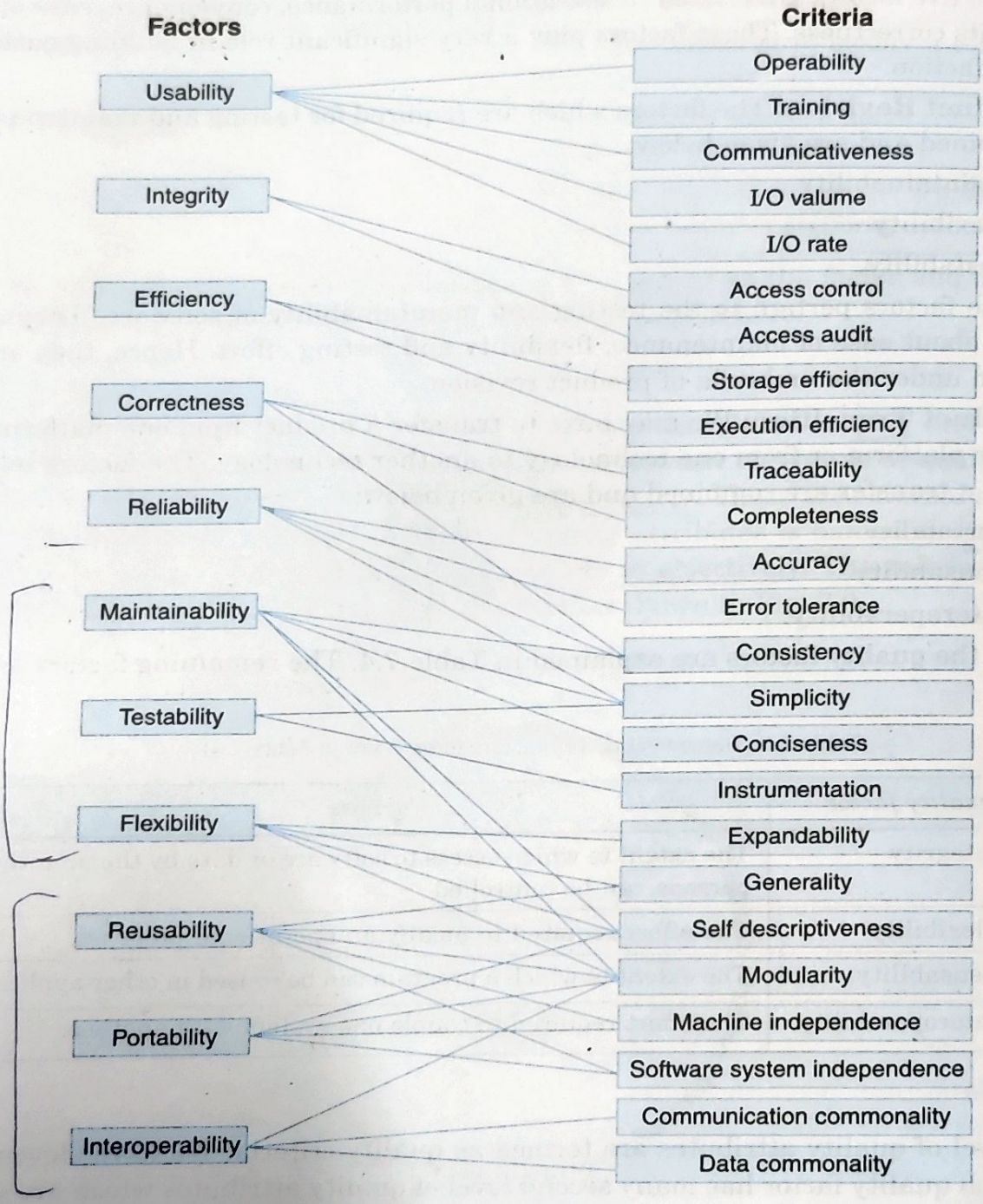


Fig. 7.10: McCall's quality model

Table 7.5(a): Relation between quality factors and quality criteria

Sr. No.	Quality criteria	Usability	Integrity	Efficiency	Correctness	Reliability	Maintainability	Testability	Flexibility	Reusability	Portability	Interoperability
1.	Operability	×										
2.	Training	×										
3.	Communicativeness	×										
4.	I/O volume	×										
5.	I/O rate	×										
6.	Access control		×									
7.	Access Audit		×									
8.	Storage efficiency			×								
9.	Execution efficiency			×								
10.	Traceability				×							
11.	Completeness				×							
12.	Accuracy					×						
13.	Error tolerance					×						
14.	Consistency				×	×	×					
15.	Simplicity					×	×	×				
16.	Conciseness						×					
17.	Instrumentation							×				
18.	Expandability								×			
19.	Generality								×	×		
20.	Self-descriptiveness						×		×	×	×	
21.	Modularity						×		×	×	×	×
22.	Machine independence									×	×	
23.	S/W system independence									×	×	
24.	Communication commonality											×
25.	Data commonality											×

Table 7.5(b): Software quality criteria

Sr. No.	Quality criteria	Definition / Purpose
1.	Operability	The ease of operation of the software
2.	Training	The ease with which new users can use the system
3.	Communicativeness	The ease with which inputs and outputs can be assimilated.

(Contd....)

Sr. No.	Quality criteria	Definition / Purpose
4.	I/O volume	It is related to the I/O volume.
5.	I/O rate	It is the indication of I/O rate.
6.	Access control	The provisions for control and protection of the software and data.
7.	Access audit	The ease with which software and data can be checked for compliance with standards or other requirements.
8.	Storage efficiency	The run-time storage requirements of the software.
9.	Execution efficiency	The run-time efficiency of the software.
10.	Traceability	The ability to link software components to requirements.
11.	Completeness	The degree to which a full implementation of the required functionality has been achieved.
12.	Accuracy	The precision of computations and output.
13.	Error tolerance	The degree to which continuity of operation is ensured under adverse conditions.
14.	Consistency	The use of uniform design and implementation techniques and notations throughout a project.
15.	Simplicity	The ease with which the software can be understood.
16.	Conciseness	The compactness of the source code, in terms of lines of code.
17.	Instrumentation	The degree to which the software provides for measurements of its use or identification of errors.
18.	Expandability	The degree to which storage requirements or software functions can be expanded.
19.	Generability	The breadth of the potential application of software components.
20.	Self-descriptiveness	The degree to which the documents are self explanatory.
21.	Modularity	The provision of highly independent modules.
22.	Machine independence	The degree to which software is dependent on its associated hardware.
23.	Software system independence	The degree to which software is independent of its environment.
24.	Communication commonality	The degree to which standard protocols and interfaces are used.
25.	Data commonality	The use of standard data representations.

It is not easy to measure many quality factors. We may have to apply software metrics, if possible, to measure such factors. The quality factors are not independent, but may overlap. Some factors may impact others in a positive sense, while others may do so negatively. We

may have to study and understand such relationships very carefully. A subjective assessment of some criteria can be obtained by giving a rating on a scale from, say, 0 (extremely bad) to 10 (extremely good). Such a subjective metric is difficult to use. Different people assessing the same criterion are likely to give different ratings. This renders a proper quality assessment almost impossible.

There are other methods to assessing quality like decomposing criterion into objectively measurable properties of the system. It is also difficult to measure every property objectively but aim is to define correctly and measure effectively.

7.2.2 Boehm Software Quality Model

Boehm introduced his quality model in 1978 [BOEH78]. He has defined three levels for quality attributes which are given in Fig. 7.11. These levels are primary uses, intermediate constructs and primitive constructs. Intermediate constructs are similar to McCall's quality factors and primitive constructs are similar to quality criteria. Hence Boehm's model is similar to McCall's in that it presents a hierarchy of characteristics, each of which contributes to overall quality. Boehm's notion of successful software includes the needs and expectations of users, as does McCall's, however, it also includes characteristics of hardware performance that are missing in McCall's model [PFLE 02].

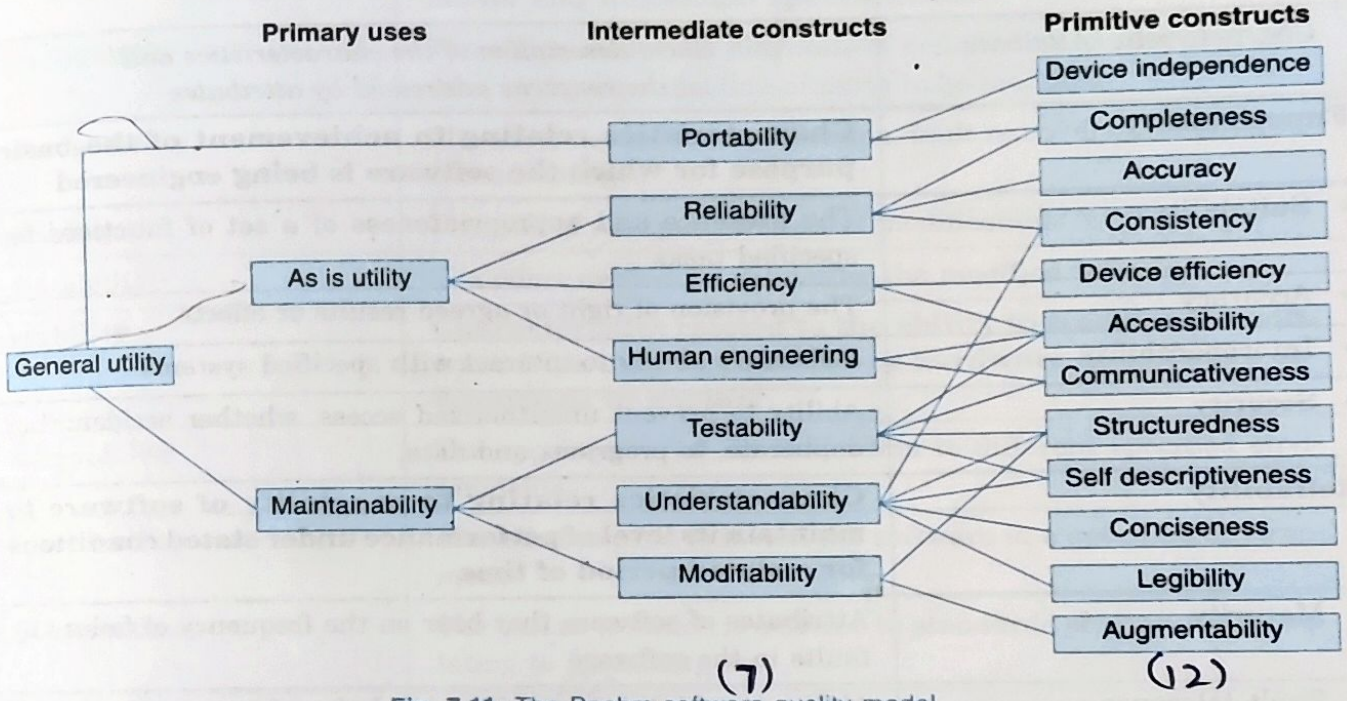


Fig. 7.11: The Boehm software quality model

The users must find the system easy to use. Thus human engineering aspect can sometime be the most critical. A system may be very good from performance point of view, but if user cannot understand how to use it, the system is a failure.

The Boehm's model asserts that quality software is software that satisfies the needs of the users. It reflects an understanding of quality where the software [PFLE 02]:

- does what the user wants it to do
- uses resources correctly and efficiently

- is easy for the user to learn and use
- is well designed, well coded, easily tested and maintained.

7.2.3 ISO 9126

The software engineering community was in the search of a single model to standardise the quality factors since 1980. The advantage of such a universal model is quite obvious; it makes easier to compare one product with another product. The result was ISO 9126 in 1992 a hierarchical model with six major attributes contributing to quality. These attributes are:

- Functionality
- Reliability
- Usability
- Efficiency
- Maintainability
- Portability.

The standard claims that these six factors are comprehensive; that is, any component of software quality can be described in terms of some aspect of one or more of the six factors.

Table 7.6: Software quality characteristics and attributes—The ISO 9126 view

Characteristic/Attribute	*Short description of the characteristics and the concerns addressed by attributes*
Functionality	**Characteristics relating to achievement of the basic purpose for which the software is being engineered**
• Suitability	The presence and appropriateness of a set of functions for specified tasks
• Accuracy	The provision of right or agreed results or effects
• Interoperability	Software's ability to interact with specified systems
• Security	Ability to prevent unauthorized access, whether accidental or deliberate, to programs and data.
Reliability	**Characteristics relating to capability of software to maintain its level of performance under stated conditions for a stated period of time**
• Maturity	Attributes of software that bear on the frequency of failure by faults in the software
• Fault tolerance	Ability to maintain a specified level of performance in cases of software faults or unexpected inputs
• Recoverability	Capability and effort needed to reestablish level of performance and recover affected data after possible failure.
Usability	**Characteristics relating to the effort needed for use, and on the individual assessment of such use, by a stated implied set of users.**

(Contd.)...

Characteristic/Attribute	*Short description of the characteristics and the concerns addressed by attributes*
• Understandability	The effort required for a user to recognize the logical concept and its applicability
• Learnability	The effort required for a user to learn its application, operation, input, and output
• Operability	The ease of operation and control by users.
Efficiency	**Characteristic related to the relationship between the level of performance of the software and the amount of resources used, under stated conditions**
• Time behaviour	The speed of response and processing times and throughput rates in performing its function
• Resource behaviour	The amount of resources used and the duration of such use in performing its function
Maintainability	**Characteristics related to the effort needed to make modifications, including corrections, improvements or adaptation of software to changes in environment, requirements and functional specifications**
• Analyzability	The effort needed for diagnosis of deficiencies or causes of failures, or for identification of parts to be modified
• Changeability	The effort needed for modification, fault removal or for environmental change
• Stability	The risk of unexpected effect of modifications
• Testability	The effort needed for validating the modified software.
Portability	**Characteristics related to the ability to transfer the software from one organization or hardware or software environment to another**
• Adaptability	The opportunity for its adaptation to different specified environments
• Installability	The effort needed to install the software in a specified environment
• Conformance	The extent to which it adheres to standards or conventions relating to portability
• Replaceability	The opportunity and effort of using it in the place of other software in a particular environment.

The hierarchical model is given in Fig. 7.12. The software quality attributes are explained in Table 7.6.

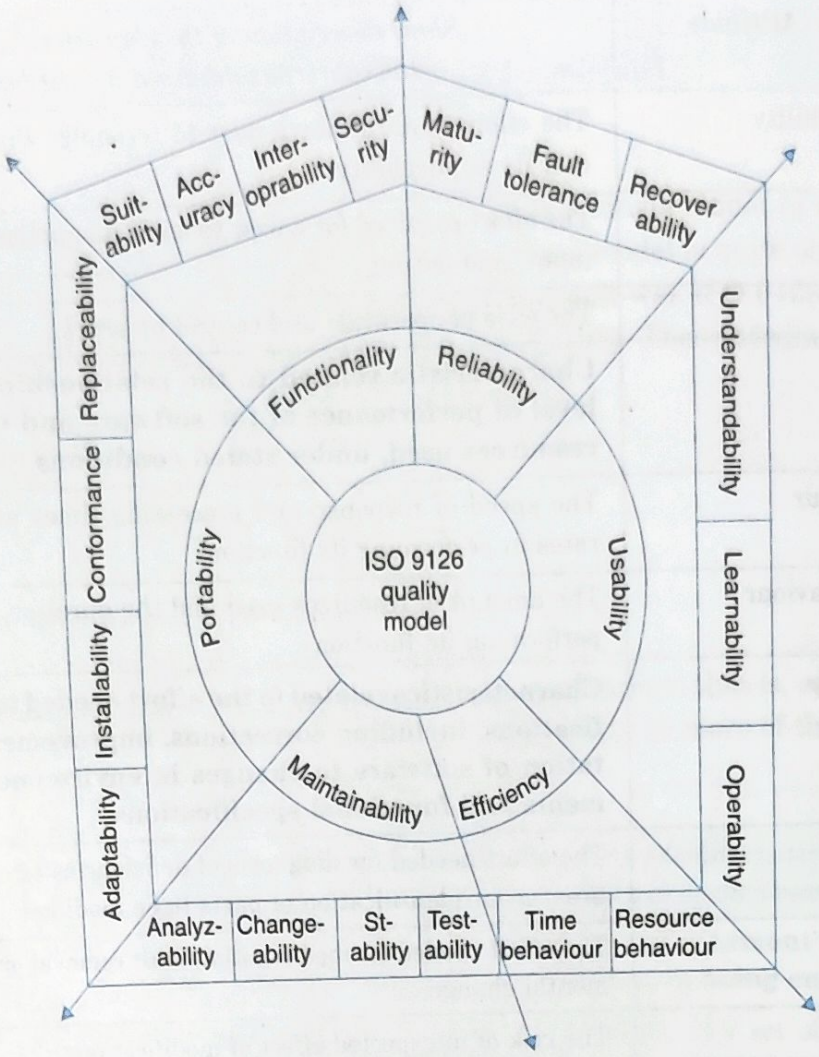


Fig. 7.12: ISO 9126 quality model

One major difference between ISO model and those of McCall and Boehm is that the ISO hierarchy is strict. Each characteristic is related only to one attribute. More-over, outer attributes are related to user view of the software rather than to an internal, developer view.

These models are helpful in articulating just what it is that we value in the software we build and use. But there are a number of difficulties in the direct application of any of the above models. We have defined quality as "meeting requirements", and it is not possible from practical point of view to define one generic model, which can fit into all types of application domain.

These models take for granted that all high level quality attributes are independent of each other. Based on this assumption, they have decomposed higher level quality factors into lower level ones independently [NIHA 01]. Thus, design of a generic model is not easy due to various attributes of quality. Sometimes, even, quality is difficult to express, although easy to feel and experience. These problems also make it difficult for us to determine how much a given model is effective and complete.

7.3 SOFTWARE RELIABILITY MODELS

To model software reliability one must first consider the principal factors that affect it: fault introduction, fault removal, and the environment. Fault introduction depends primarily on

the characteristics of the developed code (code created or modified for the application) and development process characteristics, which include software engineering technologies and tools used and level of experience of personnel. Note that code can be developed to add features or remove faults. Fault removal depends upon time, operational profile, and the quality of repair activity. The environment directly depends on the operational profile. Since some of the foregoing factors are probabilistic in nature and operate over time, software reliability models are generally formulated in terms of the random processes. The models are distinguished from each other in general terms by the nature of the variation of the random process with time.

A software reliability model specifies the general form of the dependence of the failure process on the factors mentioned. We have assumed that it is, by definition, time based (this is not to say that non-time-based models may not provide useful insights). The possibilities for different mathematical forms to describe the failure process are almost limitless.

As with any emerging discipline, software reliability has produced its share of models. Software reliability models are propounded to assess reliability of software from either specified parameters which are assumed to be known or from software-error generated data.

The past few years have seen the introduction of a number of different software reliability models. These models have been developed in response to the urgent need of software engineers, system engineers, and managers to quantify the concept of software reliability. In the reliability models, our emphasis is on failures rather than faults. The failures occur during execution of the program. Hence notion of time plays an important role. As we know, reliability is the probability that the program will not fail during a certain period of time under stated conditions.

Normally, we deal with calendar time. We may like to know the probability that a given system will not fail in one week time period. But models are generally based on execution time. The execution time is the time spent by the machine actually executing the program. Reliability models based on execution time yield better results than those based on calendar time. In many cases, a posterior translation of execution time to calendar time is possible. To emphasize this distinction, execution time will be denoted by τ and calendar time by t.

When the software fails, we try to locate and repair the fault that caused this failure. In particular, this situation arises during the testing phase of the life cycle. Most of the reliability models are applicable for system testing, when the individual modules have been integrated into one system.

During system testing, failure behaviour may not follow a constant pattern and may change over time, since faults detected are subsequently repaired. As we know, a stochastic process whose probability distribution changes over time is called non-homogeneous [VLIE02]. The variation in time between successive failures can be described in terms of a function $\mu(\tau)$ which denotes the average number of failures upto time τ. Alternatively, we may define:

$\lambda(\tau)$ = failure intensity function (Average number of failures per unit of time at time τ).

The reliability of a program increases through fault correction and hence the failure intensity decreases.

7.3.1 Basic Execution Time Model

The model was developed by J.D. MUSA [MUSA79] in 1979 and is based on execution time. It is assumed that failures may occur according to a non-homogeneous poisson process (NHPP).

Real world events may be described using poisson processes. Examples of poisson processes are:

- number of telephone calls expected in a given period of time.
- expected number of road accidents in a given period of time.
- expected number of persons visiting in a shopping mall in a given period of time.

In this case processes are non-homogeneous, since failure intensity changes as a function of time.

In this model, the decrease in failure intensity, as a function of the number of failures observed, is constant and is given as:

$$\lambda(\mu) = \lambda_0 \left(1 - \frac{\mu}{V_0}\right) \tag{7.1}$$

where λ_0 : Initial failure intensity at the start of execution.

V_0 : Number of failures experienced, if program is executed for infinite time period.

μ : Average or expected number of failures experienced at a given point in time.

The relationship between failure intensity (λ) and mean failures experienced (μ) is given in Fig. 7.13.

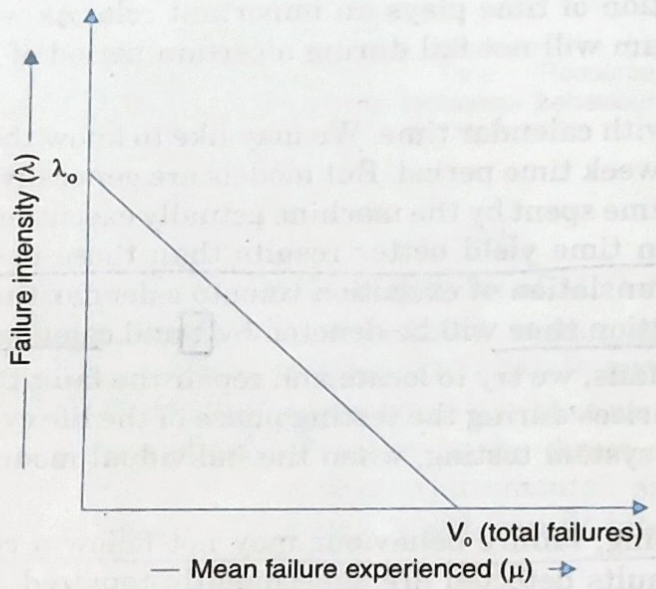


Fig. 7.13: Failure intensity λ as a function of μ for basic model

The slope of the failure intensity can be obtained by finding its first derivative and is given as:

$$\frac{d\lambda}{d\mu} = \frac{-\lambda_0}{V_0} \tag{7.2}$$

This model implies a uniform operational profile. If all input classes are selected equally often, the various faults have an equal probability of manifesting themselves. The correction of any of those faults then contributes an equal decrease in the failure intensity. The negative sign shows that there is a negative slope meaning thereby a decrementing trend in failure intensity.

The relationship between execution time (τ) and mean failures experienced (μ) is shown in Fig. 7.14.

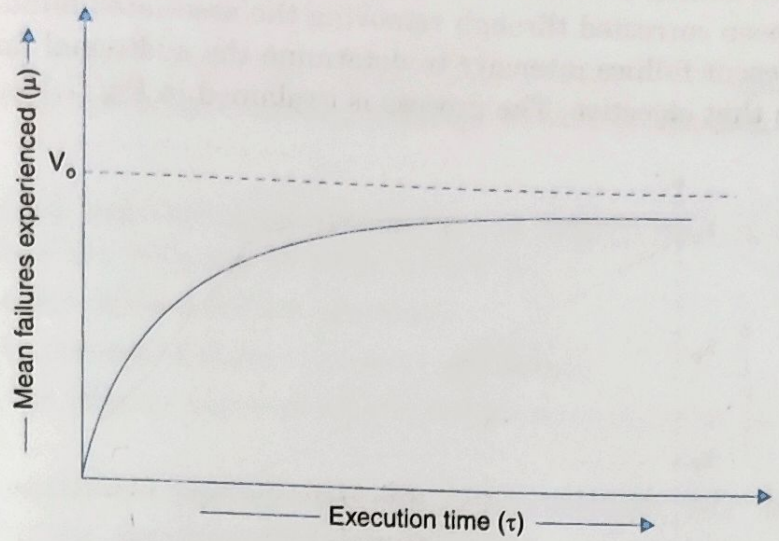


Fig. 7.14: Relationship between τ and μ for basic model

For a derivation of this relationship, equation 7.1 can be written as:

$$\frac{d\mu(\tau)}{d\tau} = \lambda_0 \left(1 - \frac{\mu(\tau)}{V_0}\right)$$

The above equation can be solved for μ(τ) and results in:

$$\mu(\tau) = V_0 \left(1 - \exp\left(\frac{-\lambda_0 \tau}{V_0}\right)\right) \tag{7.3}$$

The failure intensity as a function of execution time is shown in Fig. 7.15.

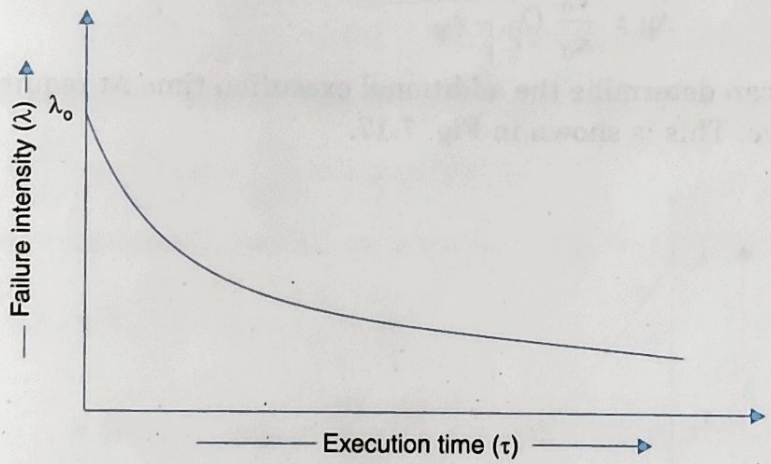


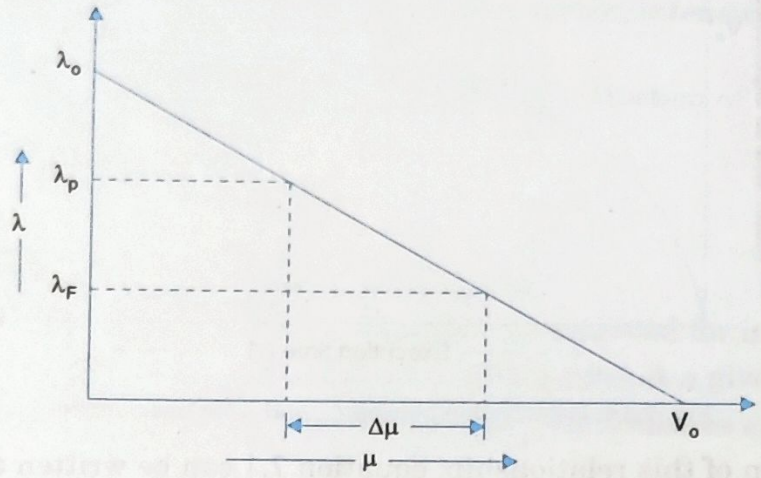
Fig. 7.15: Failure intensity versus execution time for basic model

The relationship is useful for determining the present failure intensity at any given value of execution time. This relationship can just be derived by differentiating equation 7.3 and results in:

$$\lambda(\tau) = \lambda_0 \exp\left(\frac{-\lambda_0 \tau}{V_0}\right) \tag{7.4}$$

Derived quantities

Assume that we have chosen a failure intensity objective for the software product. Assume, some failures have been corrected through removing the associated faults. Then, we can use the objective and present failure intensity to determine the additional failures that must be experienced to reach that objective. The process is explained in Fig. 7.16.



λ_0 : Initial failure intensity
λ_P : Present failure intensity
λ_F : Failure intensity objective
$\Delta\mu$: Expected number of additional failures to be experienced to reach failure intensity objective.

Fig. 7.16: Additional failures required to be experienced to reach the objective

$\Delta\mu$ can be derived in mathematical form as

$$\Delta\mu = \frac{V_0}{\lambda_0}(\lambda_P - \lambda_F) \tag{7.5}$$

Similarly, we can determine the additional execution time $\Delta\tau$ required to reach the failure intensity objective. This is shown in Fig. 7.17.

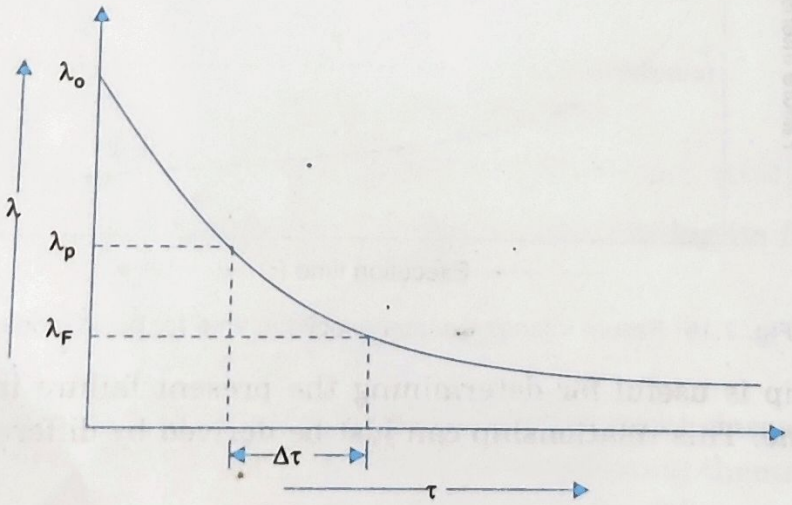


Fig. 7.17: Addition time required to reach the objective

This can be derived in mathematical form as:

$$\Delta\tau = \frac{V_0}{\lambda_0} \text{Ln}\left(\frac{\lambda_P}{\lambda_F}\right) \tag{7.6}$$

Hence, $\Delta\mu$ and $\Delta\tau$ give us the idea about failure correction workload. Both are used in making estimates of the additional calendar time required to reach the failure intensity objective.

Example 7.1

Assume that a program will experience 200 failures in infinite time. It has now experienced 100. The initial failure intensity was 20 failures/CPU hr.

(*i*) Determine the current failure intensity.

(*ii*) Find the decrement of failure intensity per failure.

(*iii*) Calculate the failures experienced and failure intensity after 20 and 100 CPU hrs. of execution.

(*iv*) Compute additional failures and additional execution time required to reach the failure intensity objective of 5 failures/CPU hr.

Use the basic execution time model for the above mentioned calculations.

Solution

Here $V_0 = 200$ failures

$\mu = 100$ failures

$\lambda_0 = 20$ failures/CPU hr.

(*i*) Current failure intensity:

$$\lambda(\mu) = \lambda_0\left(1 - \frac{\mu}{V_0}\right)$$

$$= 20\left(1 - \frac{100}{200}\right) = 20(1 - 0.5) = 10 \text{ failures/CPU hr.}$$

(*ii*) Decrement of failure intensity per failure can be calculated as:

$$\frac{d\lambda}{d\mu} = \frac{-\lambda_0}{V_0} = -\frac{20}{200} = -0.1/\text{CPU hr.}$$

(*iii*) (*a*) Failures experienced and failure intensity after 20 CPU hr:

$$\mu(\tau) = V_0\left(1 - \exp\left(\frac{-\lambda_0\tau}{V_0}\right)\right)$$

$$= 200\left(1 - \exp\left(\frac{-20 \times 20}{200}\right)\right) = 200(1 - \exp(-2))$$

$$= 200(1 - 0.1353) \approx 173 \text{ failures.}$$

$$\lambda(\tau) = \lambda_0 \exp\left(\frac{-\lambda_0\tau}{V_0}\right)$$

$$= 20 \exp\left(\frac{-20 \times 20}{200}\right) = 20 \exp(-2) = 2.71 \text{ failures/CPU hr.}$$

(*b*) Failures experienced and failure intensity after 100 CPU hr:

$$\mu(\tau) = V_0\left(1 - \exp\left(\frac{-\lambda_0\tau}{V_0}\right)\right)$$

$$= 200\left(1 - \exp\left(\frac{-20 \times 100}{200}\right)\right) \approx 200 \text{ failures (almost)}$$

$$\lambda(\tau) = \lambda_0 \exp\left(\frac{-\lambda_0\tau}{V_0}\right)$$

$$= 20 \exp\left(\frac{-20 \times 100}{200}\right) = 0.000908 \text{ failures/CPU hr.}$$

(*iv*) Additional failures ($\Delta\mu$) required to reach the failure intensity objective of 5 failures/CPU hr.

$$\Delta\mu = \left(\frac{V_0}{\lambda_0}\right)(\lambda_P - \lambda_F) = \left(\frac{200}{20}\right)(10 - 5) = 50 \text{ failures.}$$

Additional execution time required to reach failure intensity objective of 5 failures/CPU hr.

$$\Delta\tau = \left(\frac{V_0}{\lambda_0}\right)\text{Ln}\left(\frac{\lambda_P}{\lambda_F}\right)$$

$$= \frac{200}{20}\text{Ln}\left(\frac{10}{5}\right) = 6.93 \text{ CPU hr.}$$

7.3.2 Logarithmic Poisson Execution Time Model

This model is also developed by Musa et. al. [MUSA79]. The failure intensity function is different here as compared to Basic model. In this case, failure intensity function (decrement per failure) decreases exponentially whereas it is constant for basic model.

The failure intensity function is given as:

$$\lambda(\mu) = \lambda_0 \exp(-\theta\mu) \tag{7.7}$$

where θ is called the failure intensity decay parameter. The relationship between failure intensity (λ) and mean failures experienced (μ) is shown in Fig. 7.18.

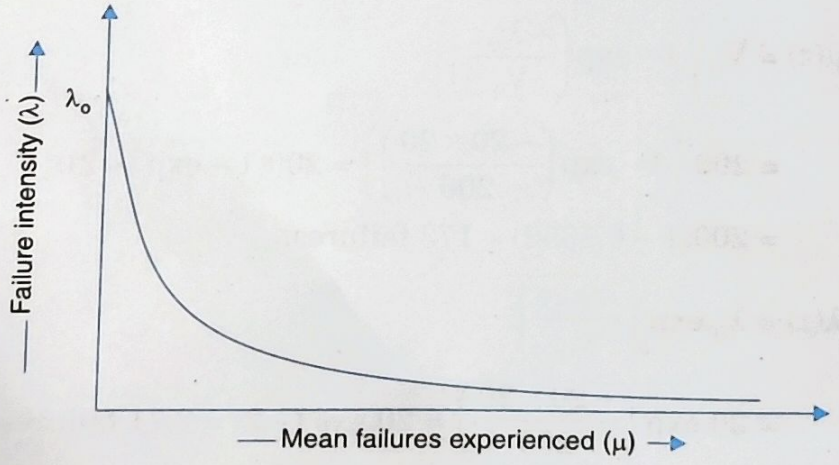


Fig. 7.18: Relationship between μ and λ

The 'θ' represents the relative change of failure intensity per failure experienced. The slope of failure intensity function is:

$$\frac{d\lambda}{d\mu} = -\lambda_0\theta \exp(-\mu\theta)$$

$$\frac{d\lambda}{d\mu} = -\theta\lambda \tag{7.8}$$

The relationship between execution time and mean failures experienced is given in Fig. 7.19.

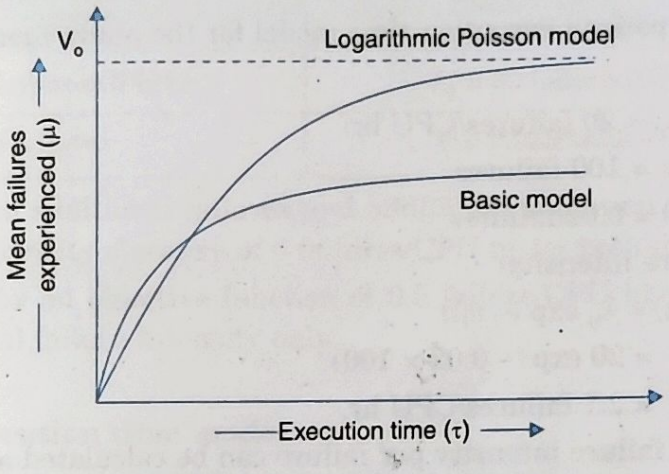


Fig. 7.19: Relationship between τ and μ

The expected number of failures for this model is always infinite at infinite time. The relation for number of failures is given by:

$$\mu(\tau) = \frac{1}{\theta} \text{Ln}(\lambda_0\theta\tau + 1) \tag{7.9}$$

In we observe Fig. 7.15 and 7.18, it is clear that the failure intensity of the logarithmic poisson execution time model drops more rapidly initially than that of the basic model and later it drops more slowly.

The expression for failure intensity is given as:

$$\lambda(\tau) = \lambda_0/(\lambda_0\theta\tau + 1) \tag{7.10}$$

The relations for the additional number of failures and additional execution time in this model are:

$$\Delta\mu = \frac{1}{\theta} \text{Ln}\left(\frac{\lambda_P}{\lambda_F}\right) \tag{7.11}$$

and

$$\Delta\tau = \frac{1}{\theta}\left[\frac{1}{\lambda_F} - \frac{1}{\lambda_P}\right] \tag{7.12}$$

where λ_P = Present failure intensity

λ_F : Failure intensity objective

Hence, at larger values of execution time, the logarithmic poisson model will have larger values of failure intensity than the basic model.

Example 7.2

Assume that the initial failure intensity is 20 failures/CPU hr. The failure intensity decay parameter is 0.02/failures. We have experienced 100 failures up to this time.

(*i*) Determine the current failure intensity.

(*ii*) Calculate the decrement of failure intensity per failure.

(*iii*) Find the failures experienced and failure intensity after 20 and 100 CPU hrs. of execution.

(*iv*) Compute the additional failures and additional execution time required to reach the failure intensity objective of 2 failures/CPU hr.

Use logarithmic poisson execution time model for the above mentioned calculations.

Solution

$\lambda_0 = 20$ failures/CPU hr.

$\mu = 100$ failures

$\theta = 0.02$/failures

(*i*) Current failure intensity:

$$\lambda(\mu) = \lambda_0 \exp(-\theta\mu)$$
$$= 20 \exp(-0.02 \times 100)$$
$$= 2.7 \text{ failures/CPU hr.}$$

(*ii*) Decrement of failure intensity per failure can be calculated as:

$$\frac{d\lambda}{d\mu} = -\theta\lambda$$
$$= -.02 \times 2.7 = -.054\text{/CPU hr.}$$

(*iii*) (*a*) Failures experienced and failure intensity after 20 CPU hr:

$$\mu(\tau) = \frac{1}{\theta} \text{Ln}(\lambda_0\theta\tau + 1)$$
$$= \frac{1}{0.02} \text{Ln}(20 \times 0.02 \times 20 + 1) = 109 \text{ failures}$$
$$\lambda(\tau) = \lambda_0/(\lambda_0\theta\tau + 1)$$
$$= (20)/(20 \times .02 \times 20 + 1) = 2.22 \text{ failures/CPU hr.}$$

(*b*) Failures experienced and failure intensity after 100 CPU hr:

$$\mu(\tau) = \frac{1}{\theta} \text{Ln}(\lambda_0\theta\tau + 1)$$
$$= \frac{1}{0.02} \text{Ln}(20 \times .02 \times 100 + 1) = 186 \text{ failures}$$
$$\lambda(\tau) = \lambda_0/(\lambda_0\theta\tau + 1)$$
$$= (20)/(20 \times .02 \times 100 + 1) = 0.4878 \text{ failures/CPU hr.}$$

(*iv*) Additional failures ($\Delta\mu$) required to reach the failure intensity objective of 02 failures/CPU hr.

$$\Delta\mu = \frac{1}{\theta} \text{Ln} \frac{\lambda_P}{\lambda_F} = \frac{1}{0.02} \text{Ln}\left(\frac{2.7}{2}\right) = 15 \text{ failures.}$$

$$\Delta\tau = \frac{1}{\theta}\left[\frac{1}{\lambda_F} - \frac{1}{\lambda_P}\right] = \frac{1}{0.02}\left[\frac{1}{2} - \frac{1}{2.7}\right] = 6.5 \text{ CPU hr.}$$

Example 7.3

The following parameters for basic and logarithmic Poisson models are given:

Basic execution time model	*Logarithmic poisson execution time model*
$\lambda_0 = 10$ failures/CPU hr.	$\lambda_0 = 30$ failures/CPU hr.
$V_0 = 100$ failures	$\theta = 0.025$/failure

(*a*) Determine the additional failures and additional execution time required to reach the failure intensity objective of 5 failures/CPU hr for both models.

(*b*) Repeat this for an objective function of 0.5 failure/CPU hr. Assume that we start with the initial failure intensity only.

Solution

(*a*) (*i*) **Basic execution time model**

$$\Delta\mu = \frac{V_0}{\lambda_0}(\lambda_P - \lambda_F)$$

$$= \frac{100}{10}(10 - 5) = 50 \text{ failures.}$$

λ_P(Present failure intensity) in this case is same as λ_0 (initial failure intensity).

Now, $$\Delta\tau = \frac{V_0}{\lambda_0} \text{Ln}\left(\frac{\lambda_P}{\lambda_F}\right)$$

$$= \frac{100}{10} \text{Ln}\left(\frac{10}{5}\right) = 6.93 \text{ CPU hr.}$$

(*ii*) **Logarithmic execution time model**

$$\Delta\mu = \frac{1}{\theta} \text{Ln}\left(\frac{\lambda_P}{\lambda_F}\right)$$

$$= \frac{1}{0.025} \text{Ln}\left(\frac{30}{5}\right) = 71.67 \text{ failures}$$

$$\Delta\tau = \frac{1}{\theta}\left(\frac{1}{\lambda_F} - \frac{1}{\lambda_P}\right)$$

$$= \frac{1}{0.025}\left(\frac{1}{5} - \frac{1}{30}\right) = 6.66 \text{ CPU hr.}$$

Logarithmic model has calculated more failures in almost some duration of execution time initially.

(*b*) Failure intensity objective (λ_F) = 0.5 failures/CPU hr.

(*i*) **Basic execution time model**

$$\Delta\mu = \frac{V_0}{\lambda_0}(\lambda_P - \lambda_F)$$

$$= \frac{100}{10}(10 - 0.5) = 95 \text{ failures}$$

$$\Delta\tau = \frac{V_0}{\lambda_0} \text{Ln}\left(\frac{\lambda_P}{\lambda_F}\right)$$

$$= \frac{100}{10} \text{Ln}\left(\frac{10}{0.5}\right) = 30 \text{ CPU hr.}$$

(*ii*) **Logarithmic execution time model**

$$\Delta\mu = \frac{1}{\theta} \text{Ln}\left(\frac{\lambda_P}{\lambda_F}\right)$$

$$= \frac{1}{0.025} \text{Ln}\left(\frac{30}{0.5}\right) = 164 \text{ failures}$$

$$\Delta\tau = \frac{1}{\theta}\left(\frac{1}{\lambda_F} - \frac{1}{\lambda_P}\right)$$

$$= \frac{1}{0.025}\left(\frac{1}{0.5} - \frac{1}{30}\right) = 78.66 \text{ CPU hr.}$$

Note that as failure intensity objectives get smaller, the $\Delta\mu$ and $\Delta\tau$ required to reach the objectives become substantially larger for logarithmic model than the basic model.

7.3.3 Calendar Time Component

The calendar time component relates execution time and calendar time by determining the calendar time to execution time ratio at any given point in time. The ratio is based on the constraints that are involved in applying resources to a project. To obtain calendar time, one integrates this ratio with respect to execution time. The calendar time component is of greatest significance during phases where the software is being tested and repaired. During this period one can predict the dates at which various failure intensity objectives will be met. The calendar time component exists during periods in which repair is not occurring and failure intensity is constant. However, it reduces in that case to a constant ratio between calendar time and execution time.

In test, the rate of testing at any time is constrained by the failure identification or test team personnel, the failure correction or debugging personnel, or the computer time available. The quantities of these resources available to a project are usually more or less established in its early stages. Increases are generally not feasible during the system test phase because of the long lead times required for training and computer procurement. At any given value of execution time, one of these resources will be limiting. The limiting resource will determine the rate at which execution time can be spent per unit calendar time. A test phase may consist of from one to three periods, each characterized by a different limiting resource.

The following is a common scenario. At the start of testing one identifies a large number of failures separated by short time intervals. Testing must be stopped from time to time to let the people who are fixing the faults keep up with the load. As testing progresses, the intervals between failures become longer and longer. The time of the failure correction personnel is no longer completely filled with failure correction work. The test team becomes the bottleneck. The effort required to run tests and analyse the results is occupying all their time. That paces the amount of testing done each day. Finally, at even longer intervals, the capacity of the computing facilities becomes limiting. This resource then determines how much testing is accomplished.

The calendar time component is based on a debugging process model. This model takes into account:

1. resources used in operating the program for a given execution time and processing an associated quantity of failures.
2. resource quantities available, and
3. the degree to which a resource can be utilised (due to bottlenecks) during the period in which it is limiting.

Table 7.7 will help in visualising these different aspects of the resources, and the parameters that result.

Resource usage

Resource usage is linearly proportional to execution time and mean failures experienced. Let x_r be the usage of resource r. Then

$$x_r = \theta_r \tau + \mu_r \mu \tag{7.13}$$

Note that θ_r is the resource usage per CPU hr. It is nonzero for failure identification personnel (θ_I) and computer time (θ_c). The quantity μ_r is the resource usage per failure. Be careful not to confuse it with mean failures experienced μ. It is nonzero for failure identification personnel (μ_I), failure correction personnel (μ_f), and computer time (μ_c).

Table 7.7: Calendar time component resources and parameters

	Usage parameters requirements per		*Planned parameters*	
Resource	CPU hr	Failure	Quantities available	Utilization
Failure identification personnel	θ_I	μ_I	P_I	1
Failure correction personnel	0	μ_f	P_f	ρ_f
Computer time	θ_c	μ_c	P_c	ρ_c

Hence, to be more precise, we have

$X_C = \mu_c \Delta\mu + \theta_c \Delta\tau$ (for computer time)

$X_f = \mu_f \Delta\mu$ (for failure correction)

$X_I = \mu_I \Delta\mu + \theta_I \Delta\tau$ (for failure identification)

For failure correction (unlike identification), resources required are dependent only on the mean failures experienced. However, computer time is used in both identification and correction of failures. Hence, computer time used will usually depend on both the amount of execution time and the number of failures.

Note that since failures experienced is a function of execution time, resource usage is actually a function of execution time only. The intermediate step of thinking in terms of failures experienced and execution time is useful in gaining physical insight into what is happening.

Computer time required per unit execution time will normally be greater than 1. In addition to the execution time for the program under test, additional time will be required for the execution of such support programs as test drivers, recording routines, and data reduction packages.

Consider the change in resource usage per unit of execution time. It can be obtained by differentiating the equation 7.13 with respect to execution time.

We obtain $$dx_r/d\tau = \theta_r + \mu_r \lambda \quad (7.14)$$

Since the failure intensity decreases with testing, the effort used per hour of execution time tends to decrease with testing. It approaches the execution time coefficient of resource usage asymptotically as execution time increases.

Calendar time to execution time relationship

Resource quantities and utilizations are assumed to be constant for the period over which the model is being applied. This is a reasonable assumption, as increases are usually not feasible [BROO75].

The instantaneous ratio of calendar time to execution time can be obtained by dividing the resource usage rate of the limiting resource by the constant quantity of resources available that can be utilized. Let to be calendar time. Then

$$dt/d\tau = (1/P_r\rho_r)\, dx_r/d\tau \quad (7.15)$$

The quantity P_r represents resource available. Note that ρ_r is the utilization. The above ratio must be computed separately for each resource-limited period. Since x_r is a function of τ, we now have a relationship between t and τ in each resource limited period.

The form of the instantaneous calendar time to execution time ratio for any given limiting resource and either model is shown in Fig. 7.20. It is readily obtained from Equations (7.14) and (7.15) as

$$dt/d\tau = (\theta_r + \mu_r \lambda)/P_r \rho_r \quad (7.16)$$

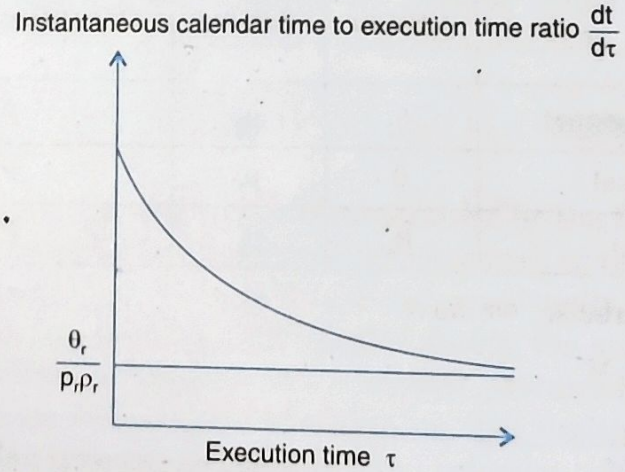


Fig. 7.20: Instantaneous calendar time to execution time ratio

The shape of this curve will parallel that of the failure intensity. The curve approaches an asymptote of $(\theta_r/P_r\rho_r)$. Note that the asymptote is 0 for the failure correction personnel resource. At any given time, the maximum of the ratios for the three limiting resources actually determines the rate at which calendar time is expanded; this is illustrated in Fig. 7.21. The maximum is plotted as a solid curve. When the curve for a resource is not maximum (not limiting), it is plotted thin. Note the transaction points FI and IC. Here, the calendar time to execution time ratios of two resources are equal and the limiting resource changes. The point FC is a potential but not true transition point. Neither resource F nor resource C is limiting near this point.

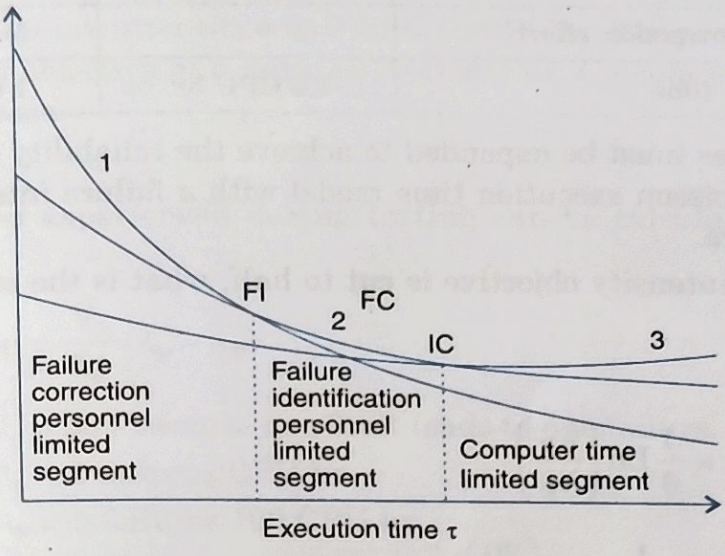


Fig. 7.21: Calendar time to execution time ratio for differrent limiting resources

The calendar time component allows you to estimate the calendar time in days required to meet the failure intensity objective. The value of this interval is particularly useful to software managers and engineers. One may determine it from the additional execution time and additional number of failures needed to meet the objective that we found for the execution time component. Second, one now determines the date on which the failure intensity objective will be achieved. This is a simple variant of the first quantity that takes account of things like weekends and holidays. However, it is useful quantity because it speaks in terms managers and engineers understand.

Example 7.4

A team runs test cases for 10 CPU hrs and identifies 25 failures. The effort required per hour of execution time is 5 person hr. Each failure requires 2 hr. on an average to verify and determine its nature. Calculate the failure identification effort required.

Solution

As we know, resource usage is:

$$X_r = \theta_r \tau + \mu_r \mu$$

Here $\theta_r = 15$ person hr., $\mu = 25$ failures

$\tau = 10$ CPU hrs., $\mu_r = 2$ hrs./failure

Hence, $X_r = 5(10) + 2(25)$

$= 50 + 50 = 100$ person hr.

Example 7.5

Initial failure intensity (λ_0) for a given software is 20 failures/CPU hr. The failure intensity objective (λ_F) of 1 failure/CPU hr. is to be achieved. Assume the following resource usage parameters.

Resource usage	*Per hour*	*Per failure*
Failure identification effort	2 Person hr.	1 Person hr.
Failure Correction effort	0	5 person hr.
Computer time	1.5 CPU hr.	1 CPU hr.

(*a*) What resources must be expended to achieve the reliability improvement ? Use the logarithmic poisson execution time model with a failure intensity decay parameter of 0.025/failure.

(*b*) If the failure intensity objective is cut to half, what is the effect on requirement of resources ?

Solution

(*a*) $$\Delta\mu = \frac{1}{\theta} \text{Ln}\left(\frac{\lambda_P}{\lambda_F}\right)$$

$$= \frac{1}{0.025} \text{Ln}\left(\frac{20}{1}\right) = 119 \text{ failures.}$$

$$\Delta\tau = \frac{1}{\theta}\left(\frac{1}{\lambda_F} - \frac{1}{\lambda_P}\right)$$

$$= \frac{1}{0.025}\left(\frac{1}{1} - \frac{1}{20}\right) = \frac{1}{0.025}(1 - 0.05) = 38 \text{ CPU hrs.}$$

Hence $X_I = \mu_I \Delta\mu + \theta_I \Delta\tau$

$= 1(119) + 2(38) = 195$ Person hrs.

$X_F = \mu_F \Delta\mu$

$= 5(119) = 595$ person hr.

$X_C = \mu_c \Delta\mu + \theta_c \Delta\tau$

$= 1(119) + (1.5)(38) = 176$ CPU hr.

(*b*) Now $\lambda_F = 0.5$ failures/CPU hr.

$$\Delta\mu = \frac{1}{0.025} \text{Ln}\left(\frac{20}{0.5}\right) \simeq 148 \text{ failures}$$

$$\Delta\tau = \frac{1}{0.025}\left(\frac{1}{0.5} - \frac{1}{20}\right) = 78 \text{ CPU hrs.}$$

So, $X_I = 1(148) + 2(78) = 304$ Person hrs.

$X_F = 5(148) = 740$ person hrs.

$X_C = 1(148) + (1.5)(78) = 265$ CPU hrs.

Hence, if we cut failure intensity objective to half, resources requirements are not doubled but they are some what less. Note that $\Delta\tau$ is approximately doubled but increases logarithmically. Thus, the resources increase will be between a logarithmic increase and a linear increase for changes in failure intensity objective.

Example 7.6

A program is expected to have 500 faults. It is also assumed that one fault may lead to one failure only. The initial failure intensity was 2 failures/CPU hr. The program was to be released with a failure intensity objective of 5 failures/100 CPU hr. Calculate the number of failures experienced before release.

Solution

The number of failures experienced during testing can be calculated using the equation mentioned below:

$$\Delta\mu = \frac{V_0}{\lambda_0}(\lambda_P - \lambda_F)$$

Here $V_0 = 500$ because one fault leads to one failure.

$\lambda_0 = 2$ failures/CPU hr.

$\lambda_F = 5$ failures/100 CPU hr.

$= 0.05$ failures/CPU hr.

So, $$\Delta\mu = \frac{500}{2}(2 - 0.05)$$

$= 487$ failures.

Hence 13 faults are expected to remain at the release instant of the software.

7.3.4 The Jelinski-Moranda Model

The Jelinski-Moranda model [JELI72] is the earliest and probably the best-known reliability model. It proposed a failure intenstiy function in the form of

$$\lambda(t) = \phi(N - i + 1) \tag{7.17}$$

where

ϕ = Constant of proportionality

N = Total number of errors present

i = number of errors found by time interval t_i.

This model assumes that all failures have the same failure rate. It means that failure rate is a step function and there will be an improvement in reliability after fixing a fault. So, every failure contributes equally to the overall reliability.

Here, failure intensity is directly proportional to the number of remaining errors in a program. The relation between time and failure intensity is shown in Fig. 7.22.

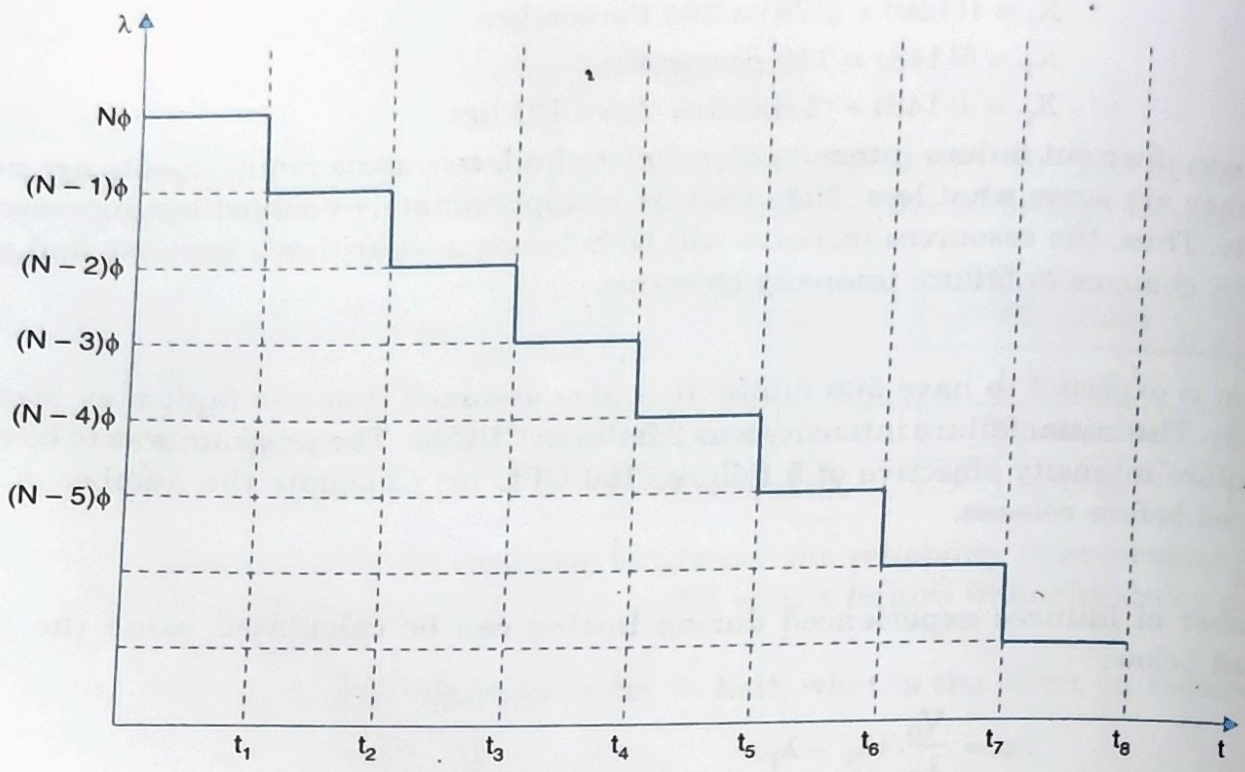


Fig. 7.22: Relation between t and λ

The time interval $t_1, t_2 \ldots t_k$ may vary in duration depending upon the occurrence of a failure.

Between $(i-1)$th and ith failure, failure intensity function is $(N - i + 1)\phi$.

Example 7.7

There are 100 errors estimated to be present in a program. We have experienced 60 errors. Use Jelinski-Moranda model to calculate failure intensity with a given value of $\phi = 0.03$. What will be failure intensity after the experience of 80 errors ?

Solution

$$N = 100 \text{ errors}$$
$$i = 60 \text{ failures}$$
$$\phi = 0.03$$

We know

$$\lambda(t) = \phi(N - i + 1)$$
$$= 0.03(100 - 60 + 1)$$
$$= 1.23 \text{ failures/CPU hr.}$$

After 80 failures

$$\lambda(t) = 0.03(100 - 80 + 1)$$
$$= 0.63 \text{ failures/CPU hr.}$$

Hence, there is continuous decrease in the failure intensity as the number of failures experienced increases.

7.3.5 The Bug Seeding Model

The so-called bug seeding model is an outgrowth of a technique used to estimate the number of animals in a wild life population or fish in a pond [FELL57]. The technique is best illustrated by discussing the estimation of the number of a specific species of fish, say, bass, N, in a small pond which contains no other type of fish. We begin by procuring a suitable number of bass, N_t from a fish hatchery and tag each one with a means of identification which will remain reliably attached for the length of the measurement period. The N_t tagged fish are then added to the N original fish and allowed to mix and disperse. After an appropriate number of days a sample is fished from the lake (by hook or net) and separated into n_t tagged bass and n untagged bass. If we assume that there was no difference in the dispersion or ease of catching of the tagged and untagged fish, then we can set up the following equation, which equals the proportions of tagged fish in the fished sample to the original fraction seeded.

$$\frac{N_t}{N+N_t} = \frac{n_t}{n+n_t} \tag{7.18}$$

We may solve the above equation for N in terms of the known quantities N_t, n, and n_t yielding the following:

$$\hat{N} = \frac{n}{n_t} N_t \tag{7.19}$$

By direct analogy we may consider N to be the unknown number of bugs in the program at the start of debugging and $N_s = N_t$ to be the number of seeded bugs (unknown to the debugger). After γ months of debugging, the bugs which have been removed are examined by someone (other than the debugger) who has a list of the seeded bugs. This tester classifies them as n_s which come from the seeded group and n which were not seeded. Direct substitution yields.

$$N = \frac{n}{n_s} N_s \tag{7.20}$$

The possibility of seeding bugs in a program and of using this technique to measure the initial bug content was first suggested by Mills [MILL72]. The results of the early experiments in this area were inconclusive for two reasons. First of all, it was difficult to makeup realistic bugs. Consider the analogous fish problem of matching ages of the bass, not to mention the realistic situation where one has, say, three types of fish—bass, perch, and pickerel. Now one must also match the unknown ratios in seeding the tagged fish unless all types of fish are equally easy to catch. Also, are hatchery-bred fish as easy to catch as pond-bred fish? The second problem with the early seeding experiments was that the measuring technique of bug seeding was used to evaluate the effectiveness of coder reading (as opposed to machine testing) as a means of debugging programs. Thus, the results of the two experiments somewhat clouded a crisp evaluation of either.

A different approach was suggested by Hyman [HYMA73] which circumvented the problem of seeding bugs. He proposed that one employ two (or more) independent debuggers to work on the same program initially. Suppose that it is estimated that debugging will take 4 months and that debugger number 1 is assigned to the program for 4 months (or the duration of the job). Debugger number 2 is assigned to the job for only one or two months at the beginning of the project. The two debuggers work independently, and after a few weeks the results of their efforts are evaluated by a third analyst, who estimates the number of program bugs N.

The estimates are repeated every few weeks, and when the third analyst is satisfied that the value N is sufficiently well estimated, the results of debugger number 2's work are given to debugger number 1 and debugger number 2 is reassigned. Now, reasonable estimate of the total number of bugs in the program, and knowing the number of bugs already removed, by subtraction we obtain the number of remaining bugs. In addition, only a portion of debugger number 2's findings duplicate debugger number 1's. Thus, debugger number 1 is able to rapidly incorporate much of debugger number 2's work, thereby producing almost a step change in the number of bugs found. In most cases the benefits should far outweigh the costs—one or two extra man-months of debugging time (the cost may be minor, zero, or negative if debugger number 2 really finds many independent errors which shorten debugger number 1's efforts). We now discuss a detailed development of the estimation formulas.

In order to develop our two-debugger estimation procedure, we begin with the following notation:

γ = development time in months; interval 0 to γ_1 is the measurement period

B_0 = number of bugs in program at $\gamma = 0$

B_1 = number of bugs found in program by debugger number 1 upto time γ_1

b_0 = number of bugs which debugger number 2 finds upto time γ_1 which are common, *i.e.*, in set B_1

b_1 = number of bugs which programmer number 2 finds upto time γ_1 which are independent, *i.e.*, not in set B_1

$B_2 = b_0 + b_1$ = number of bugs found in program by debugger number 2 upto time γ_1.

If we really believe that the bugs we would seed in a bug seeding experiment are identical to the indigenous bugs, then instead of seeding, we could merely locate a certain number of bugs and tag them (identify them). Of course the problem here is that much work must transpire in order to identify a group of bugs. Since it should not matter which set of bugs we choose as the tagged set (as long as they are representative), we should be able to treat the identified set as if they were a tagged set of bugs. In order to proceed, we must make some fundamental assumptions, which we will state as hypotheses:

(1) **Bug Characteristics unchanged as debugging proceeds:** When a large program is debugged, the bugs found during the first several weeks (months) are representative of the total bug population.

(2) **Independent debugging results in similar programs:** When two independent debuggers work on a large program, the evolution of the program is such that the difference between their two versions are small enough that they can be neglected.

(3) **Common bugs versus representative:** When two independent debuggers work on large program, the bugs which they find in common are representative of the total populatio

Assuming that the above hypotheses are valid, we can treat the quantity B_1 as if it we the selected group Ns. Similarly, b_c becomes n_s and B_2 becomes n. Substituting these quantiti solving for the original number of bugs $B_0 = N$ yields

$$\hat{B}_0 = \frac{B_2}{b_c} B_1 \qquad (7.$$

7.4 CAPABILITY MATURITY MODEL

The capability maturity model (CMM) is not a software life cycle model. Instead, it is a strategy for improving the software process, irrespective of the actual life cycle model used. The CMM was developed by Software Engineering Institute (SEI) of Carnegie-Mellon University in 1986.

CMM is used to judge the maturity of the software processes of an organization and to identify the key practices that are required to increase the maturity of these processes. The CMM is organized into five maturity levels as shown in Fig. 7.23 [PAUL94, SCHA96].

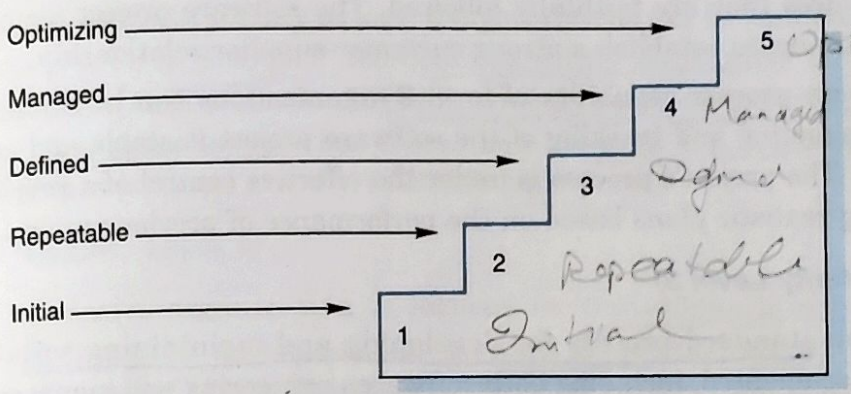


Fig. 7.23: Maturity levels of CMM

7.4.1 Maturity Levels

1. *Initial (Maturity Level 1)*

At this, the lowest level, there are essentially no sound software engineering management practices in place in the organization. Instead, everything is done on an adhoc basis. If one specific project happens to be staffed by a competent manager and a good software development team, then that project may be successful. However, the usual pattern is time and cost over runs caused by a lack of sound management in general, and planning in particular. As a result, most activities are responses to crisis, rather than preplanned tasks. In maturity level 1 organizations, the software process is unpredictable, because it depends totally on the current staff; as the staff changes, so does the process. As a consequence, it is impossible to predict, with any accuracy, the important items such as the time it will take to develop a product or the cost of that product. It is unfortunate fact that the vast majority of software organizations all over the world are level 1 organizations.

2. *Repeatable (Maturity Level 2)*

At this level, policies for managing a software project and procedures to implement those policies are established. Planning and managing new projects is based on experience with similar projects. An objective in achieving level 2 is to institutionalize effective management processes for software projects, which allow organizations to repeat successful practices developed on earlier projects, although the specific processes implemented by the projects may differ. An effective process can be characterized as practiced, documented, enforced, trained, measured, and amenable to be proved.

Instead of functioning in crisis mode as in level 1, managers identify problems as they arise and take immediate corrective action to prevent them from becoming crisis. The key point is that, without measurement it is impossible to detect problems before they get out of hand. Measurements may include the careful tracking of costs, schedules and functionality. Also, measurements taken during one project can be used to draw up realistic duration and cost schedules for future projects.

Projects in level 2 organizations have installed basic software management controls. Realistic project commitments are based on the results observed on previous projects and on the requirements of the current projects. Software project standards are defined, and the organization ensures they are faithfully followed. The software project team works with its subcontractors, if any, to establish a strong customer-supplier relationship.

The software process capability of level 2 organizations can be summarized as disciplined because planning and tracking of the software project is stable and earlier successes can be repeated. The project's process is under the effective control of a project management system, following realistic plans based on the performance of previous projects.

3. Defined (Maturity Level 3)

At this level, the standard process for developing and maintaining software across the organization is documented, including both software engineering and management processes. This standard process is referred to throughout the CMM as the organization's standard software process. Processes established at level 3 are used (and changed, as appropriate) to help the software managers and technical staff to perform more effectively. The organization exploits effective software engineering practices while standardizing its processes. An organization wide training program is implemented to ensure that the staff and managers have the knowledge and skills required to fulfill their assigned roles.

Projects tailor the organization's standard software process, to develop their own defined software process, which accounts for the unique characteristics of the project. This tailored process is referred to, in the CMM as the project's defined software process. Defined software contains a coherent, integrated set of well-defined software engineering and management processes. Because the software process is well defined, management has good control over technical progress of all projects.

The software process capability of level 3 organizations can be summarized as "standard" and "constituent" because both software engineering and management activities are stable and repeatable. Within established product lines, cost, schedule, and functionality are under control, and software quality is tracked. This process capability is based on a common, organization-wide understanding of the activities, roles, and responsibilities in a defined software process.

4. Managed (Maturity Level 4)

At this level, the organization sets quantitative quality goals for both software products and processes. Productivity and quality are measured for important software process activities across all projects as part of an organizational measurement program. An organization-wide software process database is used to collect and analyze the data available from the project's defined software processes. Software processes are instrumented with well-defined and

consistent measurements at level 4. These measurements establish the quantitative foundation for evaluating the project's software processes and products. Projects achieve control over their products and processes by narrowing the variation in their process performance to fall within acceptable quantitative boundaries.

Meaningful variations in process performance can be distinguished from random variation (noise), particularly within established product lines. The risks involved in moving up the learning curve of a new application domain are known and carefully managed. One measure could be number of faults detected per 1000 lines of code. A corresponding objective is to reduce this quantity (number of faults) over time.

The software process capability at level 4 organizations can be summarized as "predictable" because the process is measured and operates within measurable limits. This level of process capability allows an organization to predict trends of process and product quality within quantitative bounds of these limits. When these limits are exceeded, action is taken to correct the situation. Software products are of predictably high quality.

5. Optimizing (Maturity Level 5)

At this level, the entire organization is focused on continuous process improvement. The organizations have the means to identify weaknesses and strengthen the process proactively, with the goal of preventing the occurrence of defects. Data of the effectiveness of the software process is used to perform cost benefit analysis of new technologies and proposes changes to the organization's software process. Innovations that exploit the best software engineering practices are identified and transferred throughout the organization.

Software project teams in level 5 organizations "analyze defects to determine their causes". Software processes are evaluated to prevent known types of defects from recurring, and lessons learned are disseminated to other projects.

The software process capability of level 5 organizations can be characterized as "continuously improving" because level 5 organizations are continuously striving to improve the range of their process capability, there by improving the process performance of their projects. Improvement occurs both by incremental advancements in the existing process and by innovations using new technologies and methods.

These five maturity models are summarized in Fig. 7.24 [SCHA96]

Maturity level	*Characterization*
Initial	Adhoc process
Repeatable	Basic project management
Defined	Process definition
Managed	Process measurement
Optimizing	Process control

Fig. 7.24: The five levels of CMM

Experience with the capability maturity model has shown that advancing a complete maturity level usually takes from 18 months to 3 years, but moving from level 1 to level 2

sometimes takes 3 or even 5 years. This is a reflection of how difficult it is to instill a methodical approach in an organization that up to now has functioned on a purely adhoc and reactive basis.

7.4.2 Key Process Areas

Except for level 1, each maturity level is decomposed into several key process areas that indicate the areas (KPAs) an organization should focus on to improve its software process. Key process areas identify the issues that must be addressed to achieve a maturity level. Each key process area identifies a cluster of related activities that, when performed collectively, achieve a set of goals considered important for enhancing process capability. The key process areas and their purposes are listed below. The name of each key process area is followed by its two-letter abbreviation [PAUL 94, PAUL 95, WIEG 98].

By definition there are no key process areas for level 1.

The key process areas at level 2 focus on the software project's concerns related to establishing basic project management controls, as summarized below:

1. Requirements Management (RM) — Establish a common relationship between the customer requirements and the developers in order to understand the requirements of the project.
2. Software Project Planning (PP) — Establish reasonable plans for performing the software engineering and for managing the software project.
3. Software Project Tracking and Oversight (PT) — Establish adequate visibility into actual progress so that management can take effective actions when the software project's performance deviates significantly from the software plans.
4. Software Subcontract Management (SM) — Select qualified software subcontractors and manage them effectively.
5. Software Quality Assurance (QA) — Provide management with appropriate visibility into the process being used by the software project and of the products being built.
6. Software Configuration Management (CM) — Establish and maintain the integrity of the products of the software project throughout the project's software life cycle.

The key process areas at level 3 address both project and organizational issues, as the organization establishes an infrastructure that institutionalizes effective software engineering and management processes across all projects, as summarized below:

1. Organization Process Focus (PF) — Establish the organizational responsibility for software process activities that improve the organization's overall software process capability.
2. Organization Process Definition (PD) — Develop and maintain a usable set of software process assets that improve process performance across the projects and provide a basis for cumulative, long-term benefits to the organization.

3. Training Program (TP)	Develop the skills and knowledge of individuals so that they can perform their roles effectively and efficiently.
4. Integrated Software Management (IM)	Integrate the software engineering and management activities into a coherent, defined software process that is tailored from the organization's standard software process and related process assets.
5. Software Product Engineering (PE)	Consistently perform a well-defined engineering process that integrates all the software engineering activities to produce correct, consistent software products effectively and efficiently.
6. Inter group Coordination (IC)	Establish a means for the software engineering group to participate actively with the other engineering groups so the project is better able to satisfy the customer's needs effectively and efficiently.
7. Peer Reviews (PR)	Remove defects from the software work products early and efficiently. An important corollary effect is to develop a better understanding of the software work products and of the defects that can be prevented.

The key process areas of level 4 focuses on establishing a quantitative understanding of both the software process and the software work products being built, as summarized below:

1. Quantitative Process Management (QP)	Control the process performance of the software project quantitatively.
2. Software Quality Management (QM)	Develop a quantitative understanding of the quality of the project's software products and achieve specific quality goals.

The key process areas at level 5 cover the issues that both the organization and the projects must address to implement continuous and measurable software process improvement, as summarized below:

1. Defect Prevention (DP)	Identify the causes of defects and prevent them from recurring.
2. Technology Change Management (TM)	Identify beneficial new technologies (i.e., tools, methods, and processes) and transfer them into the organization in an orderly manner.
3. Process Change Management (PC)	Continually improve the software processes used in the organization with the intent of improving software quality, increasing productivity, and decreasing the cycle time for product development.

7.4.3 Common Features

For convenience, common features organize each of the key process areas. The common features are attributes that indicate whether the implementation and institutionalization of a key process area are effective, repeatable, and lasting. The five common features, followed by their two-letter abbreviations, are listed below:

1. Commitment to Perform (CO)	Describes the actions the organizations must take to ensure that the process is established and will endure. It includes practices on policy and leadership.
2. Ability to Perform (AB)	Describes the preconditions that must exist in the project or organization to implement the software process competently. It includes practices on resources, organizational structure, training, and tools.
3. Activities Performed (AC)	Describes the role and procedures necessary to implement a key process area. It includes practices on plans, procedures, work performed, tracking, and corrective action.
4. Measurement and Analysis (ME)	Describes the need to measure the process and analyze the measurements. It includes examples of measurements.
5. Verifying Implementation (VE)	Describes the steps to ensure that the activities are performed in compliance with the process that has been established. It includes practices on management reviews and audits.

There is another reason for the growing importance of the CMM. Many software organizations have stipulated that any software development organization that wishes to be a subcontractor must conform to CMM level 3 or higher. Thus, there is pressure for organizations to improve the maturity of their software processes.

7.5 ISO-9000

As explained in the previous section, the SEI capability maturity model initiative is an attempt to improve software quality by improving the process by which software is developed. The term maturity is essentially a measure of the goodness of the process itself. A different attempt to improve software quality is based on International Standards Organization (ISO) 9000-Series standards.

The standard is important, as it is becoming the main focus in which customers can judge the competence of a software developer. It has been adopted for use by over 130 countries. One of the problems with ISO-9000 series standard is that it is not industry-specific. It is expressed in general terms, and can be interpreted by the developers to diverse products such as ball bearings, hair dryers, automobiles, televisions as well as software [INCE 94].

ISO-9000 series of standards is a set of documents dealing with quality systems that can be used for quality assurance purposes. ISO-9000 series is not just software standard. It is a series of five related standards that are applicable to a wide variety of industrial activities, including design/development, production, installation, and servicing. Within the ISO-9000 Series, standard ISO-9001 for quality system is the standard that is most applicable to software development.

Because of the broadness of ISO-9001, ISO has published specific guidelines to assist in applying ISO-9001 to software namely ISO-9000-3.

There is significant room for interpretation in using ISO-9001 in the software world. ISO-9000-3 is a guide to interpret ISO-9001, yet many-to-many relationships between their clauses raises the suspicion that liberties have been taken in creating this guidance.

7.5.1 Mapping ISO-9001 to the CMM

There are 20 clauses in ISO-9001, which are summarized and compared to the practices in the CMM in this section [PAUL 94].

1. *Management responsibility*

ISO-9001 requires that the quality policy be defined, documented, understood, implemented, and maintained; that responsibilities and authorities for all personnel specifying, achieving, and monitoring quality be defined; and that in-house verification resources be defined, trained, and funded. A designated manager ensures that the quality program is implemented and maintained.

In the CMM, management responsibility for quality policy and verification activities is primarily addressed in Software Quality Assurance, although software project planning and software project tracking and oversight also include activities that identify responsibility for performing all project roles.

2. *Quality system*

ISO-9001 requires that a documented quality system, including procedures and instructions, be established. ISO-9000-3 characterizes this quality system as an integrated process throughout the entire life cycle.

Quality system activities are primarily addressed in the CMM in Software Quality Assurance. The procedures that would be used are distributed throughout the key process areas in the various Activities Performed practices.

The specific procedures and standards that a software project would use are specified in the software development plan described in Software Project Planning. Compliance with these standards and procedures is assured in Software Quality Assurance and by the auditing practices in the Verifying Implementation common feature.

3. *Contract review*

ISO-9001 requires that contracts be reviewed to determine whether the requirements are adequately defined, agreed with the bid, and can be implemented.

Review of the customer requirements, as allocated to software, is described in the CMM in Requirements Management. The software organization (supplier) ensures that the system requirements allocated to software are documented and reviewed and that missing or ambiguous requirements are clarified. Since the CMM is constrained to the Software perspective, the customer requirements as a whole are beyond the scope of this key process area.

4. *Design control*

ISO-9001 requires that procedures to control and verify the design be established. This includes planning design activities, identifying inputs and outputs, verifying the design, and controlling design changes.

In the CMM, the life cycle activities of requirements analysis, design, code, and test are described in Software Product Engineering. Planning these activities is described in Software Project Planning. Software Project Tracking and Oversight describes control of these life cycle activities, and Software Configuration Management describes configuration management of software work products generated by these activities.

5. Document control

ISO-9001 requires that the distribution and modification of documents be controlled.

In the CMM, the configuration management practices characterizing document control are described in Software Configuration Management. The specific procedures, standards, and other documents that may be placed under configuration management in the CMM are distributed throughout the key process areas in the various Activities Performed practices.

6. Purchasing

ISO-9001 requires that purchased products conform to their specified requirements. This includes the assessment of potential subcontractors and verification of purchased products.

In the CMM, this is addressed in Software Subcontract Management.

7. Purchaser-supplied product

ISO-9001 requires that any purchaser-supplied material be verified and maintained.

Integrated Software Management is the only practice in the CMM describing the use of purchased software. It does so in the context of identifying off-the-shelf or reusable software as part of planning. Integration of off-the-shelf and reusable software is one of the areas where the CMM is weak.

8. Product identification and traceability

ISO-9001 requires that the product be identified and traceable during all stages of production, delivery, and installation. The CMM covers this clause primarily in Software Configuration Management.

9. Process control

ISO-9001 requires that production processes be defined and planned. This includes carrying out production under controlled conditions, according to documented instructions. Special processes that cannot be fully verified are continuously monitored and controlled.

The procedures defining the software production process in the CMM are distributed throughout the key process areas in the various Activities Performed practices (see section 7.4.3). Specific procedures and standards that would be used are specified in the software development plan.

10. Inspection and testing

ISO-9001 requires that incoming materials be inspected or certified before use and that in-process inspection and testing be performed. Final inspection and testing are performed prior to release of finished product. Records of inspection and testing are kept.

The issues surrounding the inspection of incoming material have already been discussed in clause 4.7 of ISO-9001. The CMM describes testing in Software Product Engineering. In-process inspections in the software development are addressed in Peer Reviews.

11. Inspection, measuring, and test equipment

ISO-9001 requires that equipment used to demonstrate conformance be controlled, calibrated, and maintained. When test hardware or software is used, it is checked before use and rechecked at prescribed intervals.

This clause is generically addressed in the CMM under the testing practices in Software Product Engineering.

12. Inspection and test status

ISO-9001 requires that the status of inspections and tests be maintained for items as they progress through various processing steps.

This clause is addressed in the CMM by the testing practices in Software Product Engineering and on problem reporting and configuration status, respectively, in Software Configuration Management.

13. Control of nonconforming product

ISO-9001 requires that nonconforming product be controlled to prevent inadvertent use or installation.

Design, implementation, testing, and validation are addressed in Software Product Engineering. Software Configuration Management addresses the status of configuration items, which would include the status of items that contain known defects not yet fixed. Installation is not addressed in the CMM.

14. Corrective action

ISO-9001 requires that the causes of nonconforming product be identified. Potential causes of nonconforming product are eliminated; procedures are changed resulting from corrective action.

The software development group should look at field defects, analyze why they occurred, and take corrective action. This would typically occur through software updates and patches distributed to the customers of the software. Under this interpretation, an appropriate mapping of this clause would be problem reporting, followed with controlled maintenance of baseline work products. Problem reporting is described in Software Configuration Management in the CMM.

15. Handling, storage, packaging, and delivery

ISO-9001 requires that procedures for handling, storage, packaging, and delivery be established and maintained. Replication, delivery and installation are not covered in the CMM.

16. Quality records

ISO-9001 requires that quality records be collected, maintained, and dispositioned. The practices defining the quality records to be maintained in the CMM are distributed throughout the key process areas in the various Activities Performed practices.

17. Internal quality audits

ISO-9001 requires that audits be planned and performed. The results of audits are communicated to management, and any deficiencies found are corrected.

The auditing process is described in Software Quality Assurance. Specific audits in the CMM are called out in the auditing practices of the Verifying Implementation common feature.

18. Training

ISO-9001 requires that training needs be identified and that training be provided, since selected tasks may require qualified personnel. Records of training are maintained.

Specific training needs in the CMM are identified in the training and orientation practices in the Ability to Perform common feature.

19. Servicing

ISO-9001 requires that servicing activities be performed as specified. Although the CMM intends to apply in both the software development and maintenance environments, the practices in the CMM do not directly address the unique aspects that characterize the maintenance environment. Maintenance is embedded throughout the practices of the CMM, and they must be appropriately interpreted in the development or maintenance contexts. Maintenance is not, therefore, a separate process in the CMM.

20. Statistical techniques

ISO-9001 states that appropriate, adequate statistical techniques are identified and should be used to verify the acceptability of process capability and product characteristics.

The practices describing measurement in the CMM are distributed throughout the key process areas. Product measurement is typically incorporated into the various Activities Performed practices (see section 7.4.3), and process measurement is described in the Measurement and Analysis common feature.

7.5.2 Contrasting ISO-9001 and the CMM

Clearly there is a strong correlation between ISO-9001 and the CMM, although some issues in ISO-9001 are not covered in the CMM, and some issues in the CMM are not addressed in ISO-9001.

The biggest difference, however, between these two documents is the emphasis of the CMM on continuous process improvement. ISO-9001 addresses the minimum criteria for an acceptable quality system. It should also be noted that the CMM focuses strictly on software, while ISO-9001 has a much broader scope: hardware, software, processed materials, and services [MARQ91].

The biggest similarity is that for both the CMM and ISO-9001, the bottom line is "Say what you do; do what you say." The fundamental premise of ISO-9001 is that every important process should be documented and every deliverable should have its quality checked through a quality control activity. ISO-9001 requires documentation that contains instructions or guidance on what should be done or how it should be done. The CMM shares this emphasis on processes that are documented and practiced as documented. Phrases such as conducted

"according to a documented procedure" and following "a written organizational policy" characterize the key process areas in the CMM.

The CMM also emphasizes the need to record information for later use in the process and for improvement of the process. This is equivalent to the quality records of ISO-9001 that document whether or not the required quality is achieved and whether or not the quality system operates effectively [TICK 92].

7.5.3 Conclusion

Although there are specific issues that are not adequately addressed in the CMM, in general the concerns of ISO-9001 are encompassed by the CMM. The converse is less true. ISO-9001 describes the minimum criteria for an adequate quality management system rather than process improvement, although future revisions of ISO-9001 may address this concern. The differences are sufficient to make a route mapping impractical, but the similarities provide a high degree of overlap.

Should software process improvement be based on the CMM, with perhaps some extensions for ISO-9001 specific concerns, or should the improvement effort focus on certification concerns? A market may require ISO-9001 certification, and level 1 organization would certainly profit from addressing the concerns of ISO-9001. It is also true that addressing the concerns of the CMM would help organizations prepare for an ISO-9001 audit. Although either document could be used to structure a process improvement program, the more detailed guidance and greater breadth provided to software organizations by the CMM suggest that it is the better choice (perhaps a biased answer).

In any case, building competitive advantage should be focused on improvement, not on achieving a score, whether the score is a maturity level or a certificate. We would advocate addressing the larger context encompassed by the CMM, but even then there is a need to address the still larger business context, as exemplified by Total Quality Management.

REFERENCES

[AVIZ77] Avizienis A. and Chen L., "*On the Implementation of N–version Programming for Software Fault Tolerance During Program Execution*", Proceedings of COMPSAC 77, Chicago, Nov.1977.

[BELL91] Belli F., and Jedrcejowicz P., "*An Approach to the Reliability Optimization of Software with Redundancy*", IEEE Trans. on Software Engineering, Vol.17,No.3, March 1991.

[BOEH78] Boehm B.W., et. al, "*Characteristics of Software Quality*", Amsterdam, North Holland, 1978.

[BROO75] Brooks, F.P., Jr., The Mythical Man Month, Addison Wesley, Reading, MA, 1975.

[DUNN90] Dunn R., "*Software Quality: Concepts and Plans*", Prentice Hall, 1990.

[ECKH 91] Eckhardt D.E., et al., "*An Experimental Evaluation of Software Redundancy as a Strategy for Improving Reliability*", IEEE Trans. on Software Engineering , Vol. 17, No.7, July 1991.

[FELL57] Feller, W., "*An Introduction to Probability Theory and its Applications*", 2nd ed., Vol.1, Wiley, New York, 1957.

[HYMA73] Hyman, Mort: "Private Communication", 1973.

[INCE94] Ince D., "*ISO-9001 and Software Quality Assurance*", McGraw Hill Book Co. 1994.

[JELI72] Jelinski Z. and Moranda P.B., "*Software Reliability Research*", in Statistical Computer Performance Evaluation, Academic Press, NY, PP465–84, 1972.

[JELI71] Jelinski Z. and Moranda P.B., "*Software Reliability Research in Statistical Computer Performance Evaluation*", ed. W. Freiberger, New York: Academic Press, pp.465–484, 1971.

[JONE 86] Jones, C., Programming Productivity, McGraw–Hill, N.Y., 1986.

[LLOY77] Lloyd, D.K., and Lipow M., "*Reliability Management, Methods, and Mathematics*", Redondo Beach, CA , 1977.

[MARQ91] Marquardt D., et. al., "*Vision 2000: The Strategy for the ISO-9000 Series Standards in the 90s*", ASQC Quality Progress, Vol. 24, No. 5, 25—31, May 1991.

[MACA76] McCabe T.J., "*A Complexity Measure*", IEEE Trans on Software Engineering, Vol.2, No.6, pp.308–320, Dec.1976.

[MILL72] Mills, Harlan D., "*Mathematical Foundations of Structured Programming*", IBM Federal Systems Division Document FSC72–6012, Gaithersburg, Md., February 1972.

[MUSA75] Musa J.D., "*A Theory of Software Reliability and its Applications*", IEEE Trans. on Software Engg., SE 1(3), pp.312–327, Sept.1975.

[MUSA79] Musa J.D., "*Validity of the Execution Time Theory of Software Reliability*" , IEEE Trans. on Reliability, R–28 (3), PP. 181–191, August 1979.

[MUSA79] Musa J.D., "*Software Reliability Data*", Report Available from Data and Analysis Center for Software, Rome Air Development center, Rome, N.Y., 1979.

[MUSA79] Musa J.D., "*Validity of the Execution Time Theory for Software Reliability*", IEEE Trans. on Reliability R–28(3), pp.181–191, Aug.1979.

[MUSA80] Musa J.D., "*Software Reliability measurements*", Journal of Systems and Software, 1(3), pp.223–241, 1980.

[MUSA87] Musa J.D., A.Iannino, K. Okumoto, "*Software Reliability Measurement, Prediction and Application*", McGraw-Hill Book Company, NY., pp.183–185, 1987.

[MUSA87] Musa J.D., et. al., "*Engineering and Managing Software with Reliability Measures*", McGraw-Hill, 1987.

[MCCA77] McCall J.A., et. al., "*Factors in Software Quality*", Vol. 1, 2 and 3, AD/A-049-014/015/055, springfield, VA: National Technical Information Service, 1977.

[PAUL94] Paulk M.C., "*A Comparision of ISO-9001 and the CMM for Software*", Technical Report CMU/SEI-94–TR-12, ESC–TR-94-12, SEI, Carnegie Mellon University, USA, 1994.

[PAUL95] Paulk M.C. et. al., "*The Capability Maturity Model: Guidelines for Improving the Software Process*", Reading, MA, Addison—Wesley, 1995.

[PFLE02] Pfleceger S.L., "*Software Engineering*", 2nd Edition, Pearson Eduction Asia, 2002.

[RAND75] Randill B., "*System Structure for Software Fault Tolerance*", IEEE Trans. on Software Engineering, Vol. SE–1, June 1975.

[SCHA96] Schach S., "*Classical and Object Oriented Software Engineering*", IRWIN, USA, 1996.

[VLIE02] Vliet H.V., "*Software Engineering*", John Wiley and Sons, 2002.

[WIEG98] Wiegers K.E., "*Molding the CMM to Your Organization*", Software Development Magazine, May, 1998.

MULTIPLE CHOICE QUESTIONS

Note: Choose most appropriate answer of the following questions.

7.1. Which one is not a phase of "bath tub curve" of hardware reliability
(a) burn-in (b) useful life
(c) wear-out (d) test-out.

7.2. Software reliability is
(a) the probability of failure free operation of a program for a specified time in a specified environment
(b) the probability of failure of a program for a specified time in a specified environment
(c) the probability of success of a program for a specified time in any environment
(d) none of the above.

7.3. Fault is
(a) defect in the program (b) mistake in the program
(c) error in the program (d) all of the above.

7.4. One fault may lead to
(a) one failure (b) two failures
(c) many failures (d) all of the above.

7.5. Which 'time' unit is not used in reliability studies
(a) execution time (b) machine time
(c) clock time (d) calendar time.

7.6. Failure occurrences can be represented as
(a) time to failure (b) time interval between failures
(c) failures experienced in a time interval (d) all of the above.

7.7. Maximum possible value of reliability is
(a) 100 (b) 10
(c) 1 (d) 0.

7.8. Minimum possible value of reliability is
(a) 100 (b) 10
(c) 1 (d) 0.

7.9. As the reliability increases, failure intensity
(a) decreases (b) increases
(c) no effect (d) none of the above.

7.10. If failure intensity is 0.005 failures/hour during 10 hours of operation of a software, its reliability can be expressed as
(a) 0.90 (b) 0.92
(c) 0.95 (d) 0.98.

7.11. Software Quality is
(a) conformance to requirements (b) fitness for the purpose
(c) level of satisfaction (d) all of the above.

7.12. Defect rate is
(a) number of defects per million lines of source code
(b) number of defects per function point

(c) number of defects per unit of size of software

(d) all of the above.

7.13. How many product quality factors have been proposed in McCall quality model ?

(a) 2 (b) 3

(c) 11 (d) 6.

7.14. Which one is not a product quality factor of McCall quality model ?

(a) product revision (b) product operation

(c) product specification (d) product transition.

7.15. The second level of quality attributes in McCall quality model are termed as

(a) quality criteria (b) quality factors

(c) quality guidelines (d) quality specifications.

7.16. Which one is not a level in Boehm software quality model ?

(a) primary uses (b) intermediate constructs

(c) primitive constructs (d) final constructs.

7.17. Which one is not a software quality model ?

(a) McCall model (b) Boehm model

(c) ISO-9000 (d) ISO-9126.

7.18. Basic execution time model was developed by

(a) Bev. Littlewood (b) J.D. Musa

(c) R. Pressman (d) Victor Baisili.

7.19. NHPP stands for

(a) non homogeneous poisson process (b) non hetrogeneous poisson process

(c) non homogeneous poisson product (d) non hetrogeneous poisson product.

7.20. In basic execution time model, failure intensity is given by

(a) $\lambda(\mu) = \lambda_0 \left(1 - \frac{\mu^2}{V_0}\right)$ (b) $\lambda(\mu) = \lambda_0 \left(1 - \frac{\mu}{V_0}\right)$

(c) $\lambda(\mu) = \lambda_0 \left(1 - \frac{V_0}{\mu}\right)$ (d) $\lambda(\mu) = \lambda_0 \left(1 - \frac{V_0}{\mu^2}\right)$.

7.21. In basic execution time model, additional number of failures required to achieve a failure intensity objective ($\Delta\mu$) is expressed as

(a) $\Delta\mu = \frac{V_0}{\lambda_0} (\lambda_P - \lambda_F)$ (b) $\Delta\mu = \frac{V_0}{\lambda_0} (\lambda_F - \lambda_P)$

(c) $\Delta\mu = \frac{\lambda_0}{V_0} (\lambda_F - \lambda_P)$ (d) $\Delta\mu = \frac{\lambda_0}{V_0} (\lambda_P - \lambda_F)$.

7.22. In basic execution time model, additional time required to achieve a failure intensity objective ($\Delta\tau$) is given as

(a) $\Delta\tau = \frac{\lambda_0}{V_0} \text{Ln} \left(\frac{\lambda_F}{\lambda_P}\right)$ (b) $\Delta\tau = \frac{\lambda_0}{V_0} \text{Ln} \left(\frac{\lambda_P}{\lambda_F}\right)$

(c) $\Delta\tau = \frac{V_0}{\lambda_0} \text{Ln} \left(\frac{\lambda_F}{\lambda_P}\right)$ (d) $\Delta\tau = \frac{V_0}{\lambda_0} \text{Ln} \left(\frac{\lambda_P}{\lambda_F}\right)$.

7.23. Failure intensity function of Logarithmic Poission execution model is given as
(a) $\lambda(\mu) = \lambda_0 \text{ Ln } (-\theta\mu)$
(b) $\lambda(\mu) = \lambda_0 \exp(\theta\mu)$
(c) $\lambda(\mu) = \lambda_0 \exp(-\theta\mu)$
(d) $\lambda(\mu) = \lambda_0 \log(-\theta\mu)$.

7.24. In Logarithmic Poisson execution model, 'θ' is known as
(a) failure intensity function parameter
(b) failure intensity decay parameter
(c) failure intensity measurement
(d) failure intensity increment parameter.

7.25. In Jelinski-Moranda model, failure intensity is defined as
(a) $\lambda(t) = \phi(N - i + 1)$
(b) $\lambda(t) = \phi(N + i + 1)$
(c) $\lambda(t) = \phi(N + i - 1)$
(d) $\lambda(t) = \phi(N - i - 1)$.

7.26. CMM level 1 has
(a) 6 KPAs
(b) 2 KPAs
(c) 0 KPAs
(d) none of the above.

7.27. MTBF stands for
(a) mean time between failures
(b) maximum time between failures
(c) minimum time between failures
(d) many time between failures.

7.28. CMM model is a technique to
(a) improve the software process
(b) automatically develop the software
(c) test the software
(d) all of the above.

7.29. Total number of maturing levels in CMM are
(a) 1
(b) 3
(c) 5
(d) 7.

7.30. Reliability of a software is dependent on number of errors
(a) removed
(b) remaining
(c) both (a) and (b)
(d) none of the above.

7.31. Reliability of software is usually estimated at
(a) analysis phase
(b) design phase
(c) coding phase
(d) testing phase.

7.32. CMM stands for
(a) capacity maturity model
(b) capability maturity model
(c) cost management model
(d) comprehensive maintenance model.

7.33. Which level of CMM is for basic project management ?
(a) initial
(b) repeatable
(c) defined
(d) managed.

7.34. Which level of CMM is for process control ?
(a) initial
(b) repeatable
(c) defined
(d) optimizing.

7.35. Which level of CMM is for process management ?
(a) initial
(b) defined
(c) managed
(d) optimizing.

7.36. CMM was developed at
(a) Harvard University
(b) Cambridge University
(c) Carnegie Mellon University
(d) Maryland University.

7.37. McCall has developed a
(*a*) quality model (*b*) process improvement model
(*c*) requirement model (*d*) design model.

7.38. The model to measure the software process improvement is called
(*a*) ISO-9000 (*b*) ISO-9126
(*c*) CMM (*d*) spiral model.

7.39. The number of clauses used in ISO-9001 are
(*a*) 15 (*b*) 25
(*c*) 20 (*d*) 10.

7.40. ISO-9126 contains definitions of
(*a*) quality characteristics (*b*) quality factors
(*c*) quality attributes (*d*) all of the above.

7.41. In ISO-9126, each characteristic is related to
(*a*) one attribute (*b*) two attributes
(*c*) three attributes (*d*) four attributes.

7.42. In McCall quality model; product revision quality factor consist of
(*a*) maintainability (*b*) flexibility
(*c*) testability (*d*) none of the above.

7.43. Which is not a software reliability model ?
(*a*) the Jelinski-Moranda Model (*b*) basic execution time model
(*c*) spiral model (*d*) none of the above.

7.44. Each maturity model in CMM has
(*a*) one KPA (*b*) equal KPAs
(*c*) several KPAs (*d*) no KPA.

7.45. KPA in CMM stands for
(*a*) key Process Area (*b*) key Product Area
(*c*) key Principal Area (*d*) key Performance Area.

7.46. In reliability models, our emphasis is on
(*a*) errors (*b*) faults
(*c*) failures (*d*) bugs.

7.47. Software does not break or wearout like hardware. What is your opinion ?
(*a*) true (*b*) false
(*c*) cannot say (*d*) not fixed.

7.48. Software reliability is defined with respect to
(*a*) time (*b*) speed
(*c*) quality (*d*) none of the above.

7.49. MTTF stands for
(*a*) mean time to failure (*b*) maximum time to failure
(*c*) minimum time to failure (*d*) none of the above.

7.50. ISO-9000 is a series of standards for quality management systems and has
(*a*) 2 related standards (*b*) 5 related standards
(*c*) 10 related standards (*d*) 25 related standards.

EXERCISE

7.1. What is software reliability? Does it exist?

7.2. Explain the significance of bath tube curve of reliability with the help of a diagram.

7.3. Compare hardware reliability with software reliability.

7.4. What is software failure? How is it related with a fault?

7.5. Discuss the various ways of characterising failure occurrences with respect to time.

7.6. Describe the following terms:

(*i*) Operational profile (*ii*) Input space

(*iii*) MTBF (*iv*) MTTF

(*v*) Failure intensity.

7.7. What are uses of reliability studies? How can one use software reliability measures to monitor the operational performance of software?

7.8. What is software quality? Discuss software quality attributes.

7.9. What do you mean by software quality standards? Illustrate their essence as well as benefits.

7.10. Describe the McCall software quality model. How many product quality factors are defined and why?

7.11. Discuss the relationship between quality factors and quality criteria in McCall's software quality model.

7.12. Explain the Boehm software quality model with the help of a block diagram.

7.13. What is ISO-9126? What are the quality characteristics and attributes?

7.14. Compare the ISO-9126 with McCall software quality model and highlight few advantages of ISO-9126.

7.15. Discuss the basic model of software reliability. How can $\Delta\mu$ and $\Delta\tau$ be calculated?

7.16. Assume that the initial failure intensity is 6 failures/CPU hr. The failure intensity decay parameter is 0.02/failure. We assume that 45 failures have been experienced. Calculate the current failure intensity.

7.17. Explain the basic and logarithmic Poission model and their significance in reliability studies.

7.18. Assume that a program will experience 150 failures in infinite time. It has now experienced 80. The initial failure intensity was 10 failures/CPU hr.

(*i*) Determine the current failure intensity.

(*ii*) Calculate the failures experienced and failure intensity after 25 and 40 CPU hrs. of execution.

(*iii*) Compute additional failures and additional execution time required to reach the failure intensity objective of 2 failures/CPU hr.

Use the basic execution time model for the above mentioned calculations.

7.19. Write a short note on Logarithmic Poisson Execution time model. How can we calculate $\Delta\mu$ and $\Delta\tau$?

7.20. Assume that the initial failure intensity is 10 failures/CPU hr. The failure intensity decay parameter is 0.03/failure. We have experienced 75 failures upto this time. Find the failures experienced and failure intensity after 25 and 50 CPU hrs. of execution.

7.21. The following parameters for basic and logarithmic poisson models are given:

Basic execution time model	*Logarithmic poisson execution time model*
λ_0 = 5 failures/CPU hr	λ_0 = 25 failures/CPU hr
V_0 = 125 failures	θ = 0.3/failure

Determine the additional failures and additional execution time required to reach the failure intensity objective of 0.1 failure/CPU hr. for both models.

7.22. Quality and reliability are related concepts but are fundamentally different in a number of ways. Discuss them.

7.23. Discuss the calendar time component model. Establish the relationship between calendar time to execution time.

7.24. A program is expected to have 250 faults. It is also assumed that one fault may lead to one failure. The initial failure intensity is 5 failures/CPU hr. The program is released with a failure intensity objective of 4 failures/10 CPU hr. Calculate the number of failures experienced before release.

7.25. Explain the Jelinski-Moranda model of reliability theory. What is the relation between '*t*' and 'λ'?

7.26. Describe the Mill's bug seeding model. Discuss few advantages of this model over other reliability models.

7.27. Explain how the CMM encourages continuous improvement of the software process.

7.28. Discuss various key process areas of CMM at various maturity levels.

7.29. Construct a table that correlates key process areas (KPAs) in the CMM with ISO-9000.

7.30. Discuss the 20 clauses of ISO-9001 and compare with the practices in the CMM.

7.31. List the difference of CMM and ISO-9001. Why is it suggested that CMM is the better choice than ISO-9001 ?

7.32. Explain the significance of software reliability engineering. Discuss the advantages of using any software standard for software development ?

7.33. What are various key process areas at defined level in CMM? Describe activities associated with one key process area.

7.34. Discuss main requirements of ISO-9001 and compare it with SEI capability maturity model.

7.35. Discuss the relative merits of ISO-9001 certification and the SEI CMM based evaluation. Point out some of the shortcomings of the ISO-9001 certification process as applied to the software industry.