

CHAPTER – 5

SOFTWARE RELIABILITY





INTRODUCTION:

Software reliability is one of the most important element of the overall quality of any software even after accomplishment of a software work. If it fails to meet its actual performance after its deployment, then the software is considered as unreliable software. We cannot expect better performance from such software.

Software reliability is defined in statistical terms as the probability of failure free operation of a computer program in a specific environment for specific time for a software to be reliable, it must performs operation based upon its analysis and design, available resources, reusability (if reusable components are involved) and so on.

If the design standards are not maintained, efficient algorithms are not implemented, necessary resources are not available or the supply is not at proper times, then there is high probability of software failure or there is degradation in quality of performance.





Therefore, in order to make a software reliable, we should take some measures or earlier stages as follows:

-  Designing software project based on the available resources.
-  Develop software that best fits the current environment conditions.
-  Estimating costs that might be required even after its implementation.
-  Estimating the actual performance upon using reusable resources.

SOFTWARE SAFETY:

Software safety is a SQA activity that focuses on the identification and assessment of potential hazards that may affect software negatively and cause an entire system to fail. If hazards can be identified early in the software process, software design features can be specified that will either eliminate or control potential hazards.

A modeling and analysis process is conducted as part of software safety. Initially, hazards are identified and categorized by critically and risk. For example: some of the hazards associated with a computer base cruise control for an automobile might be:

-  Causes uncontrolled acceleration that cannot be stopped.
-  Does not respond to depression of brake pedal (by turning off)
-  Does not engage when switch is activated.
-  Slowly loses or gain speed.

Once such hazards are identified analysis techniques are used to assign severity and probability of occurrence.

Software reliability uses statistical analysis to determine the likelihood that a software failure will cause. However, the occurrence of failure does not necessarily result in a mishap (accident).




Software safety examines the ways in which failure result in conditions that can lead to a mishap. That is, failures are not considered in vacuum but are evaluated in the context of an entire computer based system and its environment.

MEASURES OF RELIABILITY AND AVAILABILITY:

If we consider a computer based system, a simple measure of reliability is meantime between failures.

$$MTBF = MTTF + MTTR$$

$$\text{Availability} = \frac{MTTF}{MTTF + MTTR} * 100\%$$

-  MTBF = Meantime Between Failure
-  MTTF = Meantime to Failure
-  MTTR = Meantime to Repair

SOFTWARE RELIABILITY MODELS:

The basic goal of software engineering is to produce high quality software at low cost. With growth in size and complexity of software, management issues began dominating. Reliability is one of the representative qualities of software development process. Reliability of software is basically defined as the probability of expected operation over specified time interval.

Software Reliability is an important attribute of software quality, together with functionality, usability, performance, serviceability, capability, install ability, maintainability, and documentation. Reliability is the property of referring 'how well software meets its requirements' & also 'the probability of failure free operation for the specified period of time in a specified environment. Software reliability defines as the failure free operation of computer program in a specified environment for a specified time. Software Reliability is hard to achieve, because the complexity of software tends to be high.

A proliferation of software reliability models have emerged as people try to understand the characteristics of how and why software fails, and try to quantify software reliability. Software Reliability Modelling techniques can be divided into two subcategories: Prediction modelling and Estimation modelling. Both kinds of modelling techniques are based on observing and accumulating failure data and analyzing with statistical inference.

1. PREDICTION MODELS:

This model uses historical data. They analyze previous data and some observations. They are usually made prior to development and regular test phases. The model follow the concept phase and the predication from the future time.

a. Musa Model:

This prediction technique is used to predict, prior to system testing, what the failure rate will be at the start of system testing. This prediction can then later be used in the reliability growth

modelling. For this prediction method, it is assumed that the only thing known about the hypothetical program is a prediction of its size and the processor speed.

This model assumes that failure rate of the software is a function of the number of faults it contains and the operational profile. The number of faults is determined by multiplying the number of developed executable source instructions by the fault density. Developed excludes re-used code that is already debugged. Executable excludes data declarations and compiler directives. For fault density at the start of system test, a value for faults per KSLOC needs to be determined. For most projects the value ranges between 1 and 10 faults per KSLOC. Some developments which use rigorous methods and highly advanced software development processes may be able to reduce this value to 0.1 fault/KSLOC.

b. Putnam Model:

Created by Lawrence Putnam, Sr. the Putnam model describes the time and effort required to finish a software project of specified size. SLIM (Software Lifecycle Management) is the name given by Putnam to the proprietary suite of tools his company QSM, Inc. has developed based on his model. It is one of the earliest of these types of models developed, and is among the most widely used. Closely related software parametric models are Constructive Cost Model (COCOMO), Parametric Review of Information for Costing and Evaluation – Software (PRICE-S), and Software Evaluation and Estimation of Resources – Software Estimating Model (SEER-SEM).

Putnam's observations about productivity levels to derive the software equation:

$$\frac{B^{1/3} \cdot \text{Size}}{\text{Productivity}} = \text{Effort}^{1/3} \cdot \text{Time}^{4/3}$$

Where:

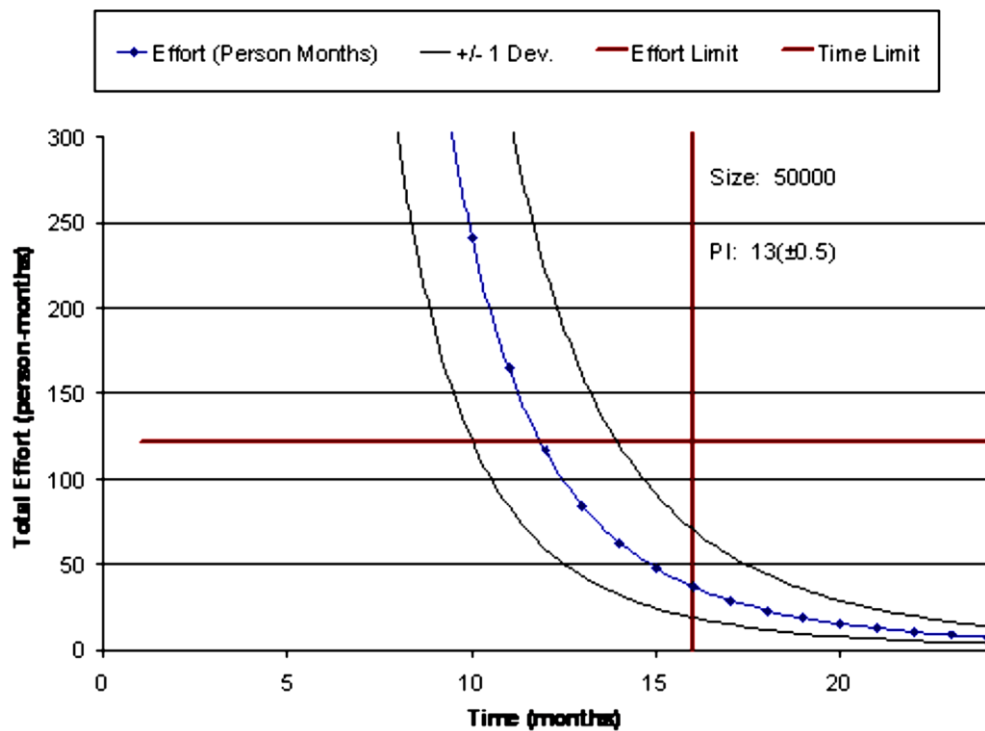
- ✚ Size is the product size (whatever size estimate is used by your organization is appropriate). Putnam uses ESLOC (Effective Source Lines of Code) throughout his books.
- ✚ B is a scaling factor and is a function of the project size Productivity is the Process Productivity, the ability of a particular software organization to produce software of a given size at a particular defect rate.
- ✚ Effort is the total effort applied to the project in person-years.
- ✚ Time is the total schedule of the project in years.

In practical use, when making an estimate for a software task the software equation is solved for *effort*:

$$\text{Effort} = \left[\frac{\text{Size}}{\text{Productivity} \cdot \text{Time}^{4/3}} \right]^3 \cdot B$$

An estimated software size at project completion and organizational process productivity is used. Plotting *effort* as a function of *time* yields the *Time-Effort Curve*. The points along the curve represent the estimated total effort to complete the project at some *time*. One of the distinguishing features of the Putnam model is that total effort decreases as the time to complete

the project is extended. This is normally represented in other parametric models with a schedule relaxation parameter.



Thus estimating method is fairly sensitive to uncertainty in both size and process productivity. Putnam advocates obtaining process productivity by calibration.

c. Rome Lab TR-92-52 Model:

Software Reliability Measurement and Test Integration Techniques Method RL-TR-92-52 contain empirical data that was collected from a variety of sources, including the Software Engineering Laboratory. The model consists of 9 factors that are used to predict the fault density of the software application. The nine factors are:

Factor	Measure	Range of values	Applicable Phase*	Tradeoff Range
A – Application	Difficulty in developing various application types	2 to 14 (defects/KSLOC)	A-T	None – fixed
D - Development organization	Development organization, methods, tools, techniques, documentation	.5 to 2.0	If known at A, D-T	The largest range
SA - Software anomaly management	Indication of fault tolerant design	.9 to 1.1	Normally, C-T	Small
ST - Software	Traceability of design and code to requirements	.9 to 1.0	Normally, C-T	Large
SQ - Software quality	Adherence to coding standards	1.0 to 1.1	Normally, C-T	Small
SL - Software	Normalizes fault density by language type	Not applicable	C-T	N/A
SX - Software	Unit complexity	.8 to 1.5	C-T	Large
SM - Software	Unit size	.9 to 2.0	C-T	Large
SR - Software standards review	Compliance with design rules	.75 to 1.5	C-T	Large

There are certain parameters in this prediction model that have tradeoff capability. This means that there is a large difference between the maximum and minimum predicted values for that particular factor. Performing a tradeoff means that the analyst determines where some changes can be made in the software engineering process or product to experience an improved fault density prediction. A tradeoff is valuable only if the analyst has the capability to impact the software development process. The output of this model is a fault density in terms of faults per KSLOC. This can be used to compute the total estimated number of inherent defects by simply multiplying by the total predicted number of KSLOC.

d. Rayleigh Model:

This model predicts fault detection over the life of the software development effort and can be used in conjunction with the other prediction techniques. Software management may use this profile to gauge the defect status. This model assumes that over the life of the project that the faults detected per month will resemble a Raleigh curve.

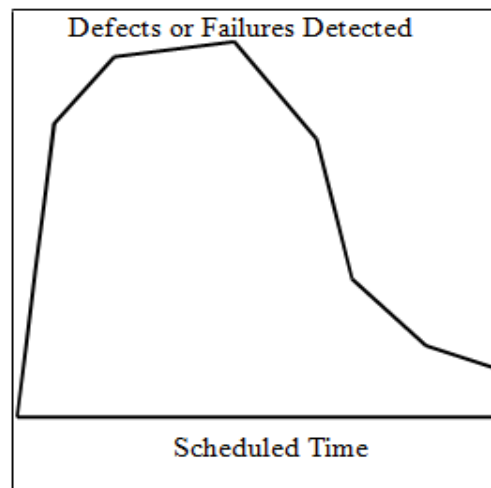
The Rayleigh model is the best suited model for estimating the number of defects logged throughout the testing process, depending on the stage when it was found (unit testing, integration testing, system testing, functional testing etc.). Defect prediction function is the following:

$$F(t)f(K_{\text{cum}}, t, t_m) = (1)$$

Parameter K is the cumulated defect density, t is the actual time unit and t_m is the time at the peak of the curve.

For example, the estimated number of defects impacts the height of the curve while the schedule impacts the length of the curve. If the actual defect curve is significantly different from the

predicted curve then one or both of these parameters may have been estimated incorrectly and should be brought to the attention of management.



2. ESTIMATION MODEL:

This model uses the current data from the current software development effort and doesn't use the conceptual development phases and can estimate at any time.

a. Weibull Model (WM):

This model is used for software/hardware reliability. The model incorporates both increasing/decreasing and failure rate due to high flexibility. This model is a finite failure model.

$$MTTF = \int (1-F(t)) dt = \int \exp(-bt^\alpha) dt \quad (4)$$

Where, MTTF = Mean Time to Failure

α , β = Weibull distribution parameters. t = Time of failure.

A Weibull-type testing-effort function with multiple change-points can be estimated by using the methods of least squares estimation (LSE) and maximum likelihood estimation (MLE). The LSE minimizes the sum of squares of the derivations between actual data patterns and predicted curve, while the MLE estimates parameters by solving a set of simultaneous equations.

b. Bayesian Models (BM):

The models are used by several organizations like Motorola, Siemens & Philips for predicting reliability of software. BM incorporates past and current data. Prediction is done on the bases of number of faults that have been found & the amount of failure free operations. The Bayesian SRGM considers reliability growth in the context of both the number of faults that have been detected and the failure-free operation. Further, in the absence of failure data, Bayesian models consider that the model parameters have a prior distribution, which reflects judgment on the unknown data based on history e.g. a prior version and perhaps expert opinion about the software.

c. J-M Model (JMM):

This model assumes that failure time is proportional to the remaining faults and taken as an exponential distribution. During testing phase the number of failures at first is finite. Concurrent mitigation of errors is the main strength of the model and error does not affect the remaining errors. Error removal is all human behavior which is irregular so it cannot be avoid by introducing new errors during the process of error removal.

$$(MTBF)_{t_i} = 1 / (N - (i - 1)) \quad (6)$$

Where, N= Total number of faults.

i = Number of fault occurrences. MTBF=Mean Time between failure.

t = Time between the occurrence of the $(i-1)^{st}$ and i^{th} fault occurrences.

This model assumes that N initial faults in the code prior to testing are a fixed but known value. Failures are not correlated and the times between failures are independent and exponentially distributed random variables. Fault removal on failure occurrences is instantaneous and does not introduce any new faults into the software under test. The hazard rate $z(t)$ of each fault is time invariant and a constant. Moreover, each fault is equally likely to cause a failure.

d. Goel-Okumoto Model (GOM):

G-O model takes the number of faults per unit time as independent random variables. In this model the number of faults occurred within the time and model estimates the failure time. Delivery of software within cost estimates is also decided by this model.

The Goel-Okumoto (G-O) non-homogeneous Poisson process (NHPP) model has slightly different assumptions from the J-M model. The significant difference between the two is the assumption that the expected number of failures observed by time t follows a Poisson distribution with a bounded and non-decreasing mean value function (t) . The expected value of the total number of failures observed in infinite time is a finite value N .

e. Thompson and Chelson's Model:

In this model the Number of failures detected in each interval (f_i) and Length of testing time for each interval i (T_i).

$$(f_i + f_0 + 1)'$$

$$(T_i + T_0)$$

Software is corrected at end of testing interval. Software is operational. Software is relatively fault free. The Thompson Chelson model can be used, If the failure intensity is decreasing, has the software been tested or used in an operational environment representative of its end usage, with no failures for a significant period of time.