Chapter 3

Software Reliability and Open Source Software

3.1 Introduction

The Institute of Electrical and Electronic Engineers (IEEE) [77] defines software reliability as the probability that software will not cause a system failure for a specified time under specified conditions. The probability is a function of the existence of faults in the software. The inputs to the system determine whether existing faults, if any, are encountered. John Musa of ATandT Bell Laboratories defines software reliability as the probability that a given software system operates for some time period without software error, on the machine for which it was designed, given that it is used within design limits. The measurement and analysis techniques include software metrics, software reliability models, and software analyses such as fault trees and failure modes effects and criticality analysis (FMECA) [78]. Software metrics are measures of some aspect of the software product or process it. Software reliability models, for the most part, model the failures occurring because of the software. Software analysis enables development personnel to find errors in the software while the software is still in a laboratory environment.

3.2 Objectives

The main objectives of this chapter are as follows:

• To analyze the process of software reliability and its usage.
• To understand the significance of software reliability.
• To understand various types of benchmarked models for software reliability.
• To understand architecture, development and growth of different types of open source software, especially related to ERP application.
• To analyze the characteristics and criteria for selection and evaluation of open source software.
• To understand opportunities and threats for various open source ERP package.
• To use and analyze OFBiz as well as its respective usage characteristics in terms of content, security, product, accounting privileges etc.
3.3 Significance of Software Reliability

There are at least five major reasons why reliable software has become a very important issue in the last decade or so.

1. Systems are becoming software-intensive. Mainly flight systems are becoming more software intensive than hardware intensive. Financial systems including teller, automated teller, and loan processing are software-intensive. Defense and energy systems are becoming more software-intensive. Everything from insurance rates to credit histories to hotel reservations to long-distance telephone calls is performed by software. Software affects our daily lives.

2. Many software-intensive systems are safety critical. Flight systems, electronic warfare systems, radar, air traffic control, medical systems, energy systems, and space systems are all software-intensive systems that are also safety critical. Even systems that are not safety critical may be mission critical, meaning that success is critical to some end purpose (such as defeating an enemy at war), or failure is extremely costly financially.

3. Customers now require more reliable software. Many government contracts are now requiring that an established level of software reliability be achieved. Software has also become part of the system reliability allocations on many government contracts. Commercial clients also require more reliable systems, and many are attempting to establish the same criteria as the government for developing of reliable software.

4. Software errors are not being tolerated by end users or by clients of end users. Financial institutions, medical institutions, the government, communication corporations, and other corporations are in a position of being legally liable for software that is not accurate, that causes potential loss of life or loss of mission, that causes inconvenience to end users, and that causes companies and end users to lose profits. In addition to being liable, users and developers of software are also facing increasing maintenance costs.

5. The cost of developing software is increasing. Data from a variety of sources show that for many systems developing the software is becoming one of the major costs of the system, if not the major cost. Much of the software cost can be associated with corrective action, particularly corrective action late in the
development cycle. The cost of maintaining software has been shown in some studies to be as much as 40-70% of the total development cost [143]. Some NASA and Air Forces have estimated it to be 50% of their development cost.

3.4 Software Reliability Modeling

A software reliability model specifies the general form of the dependence of the failure process on the principal factors that affect it: fault introduction, fault removal, and the operational environment [79]. These models are used to predict how much more time the software needs to be tested to achieve the desired failure intensity and to predict the expected reliability of the delivered software. The model parameters may be determined by means of the following: (1) Estimation: measures the reliability by applying statistical inferences to the collected failure data. This method validates the goodness of the model by assessing its current reliability. (2) Prediction: measures the future software reliability using the available software metrics. The failure data that is used in the reliability models may be of two types: (1) Failure count data which is expressed as the number of failures in each time interval. (2) Time between failure data which is expressed as the time interval between consecutive failures [80]. One type of input data can be transformed to the other alternate models either by using the cumulative failure data or by using some of the existing reliability tools such as CASRE and SMERFS. A well defined software reliability model can determine important characteristics of the failure process by incorporating expressions for the average number of failures experienced at any point in time, the average number of failures in a time interval, the failure intensity at any point in time, and the probability distribution of failure intervals. A good software reliability model, based on sound assumptions, gives better projection of future failure behavior, computes useful quantities, is simple, and is widely applicable [9].

3.5 Software Reliability Model Classification

One of the early reliability models, which were based on hardware reliability concepts, was developed by [81]. In the seventies, many software reliability models were proposed, developed and widely used. Since then, many different software reliability models have been developed and numerous researchers in software
reliability engineering have attempted to categorize and classify them. Classify the reliability models in terms of five attributes [82]:

i. **Time domain**: either calendar or execution time.

ii. **Category**: either finite or infinite number of failures. For the finite number of failures category models, there are a number of classes depending on the functional form of the failure intensity in terms of time. For infinite failure category models, there are a number of families depending on the functional form of the failure intensity in term of the expected number of failures experienced.

iii. **Type**: the distribution of the number of failures experienced as a function of t.

iv. **Class**: functional form of the failure intensity expressed in terms of time (for finite failure category only).

v. **Family**: functional form of the failure intensity expressed in terms of the expected number of failures experienced (for infinite failure category only). For the sake of simplicity, Musa and Okumoto first separate the finite from the infinite models. They then incorporate the five attributes as a guide to finding the relationship between the models; thus clarifying the comparison between the models.

The simplicity of this classification explains its popularity of usage. Goel et al. [79] define two main categories of software reliability models: (1) software growth reliability models that estimate reliability using the error history, and (2) statistical models that estimate the reliability using the results (success or failure) of executing test cases. The software growth reliability models are classified based on the nature of the failures the time between failures models, failure counts models, and fault seeding and input domain based models. Categorize models into two major types [83]:

1. **Type I**: time between successive failures models, which breaks down to failure rate Type I-1 and random function Type I-2.

2. **Type II**: the number of failures up to a given time. Classify reliability models as follows [84]: (1) Data-domain models: A better reliability estimate can be achieved if all of the combinations of the inputs are identified and the outcomes are well observed. To implement this theory, this model category is decomposed into fault-seeding models and input-domain models. (2) Time-domain models:
model the failure process using the failure history to estimate the number of faults and the required test time to uncover these faults. Homogeneous Markov, non-homogeneous Markov, semi-Markov are models that belong to the time-domain model. Classify software reliability according to software development life cycle phases. Their classification is well defined and comprehensive.

### 3.5.1 Exponential Models

All software reliability models of the exponential class have a common set of assumptions. In addition to these common assumptions, each model has its own unique set of assumptions. The standard assumptions are [85]:

- The software is operated in a similar manner as that for which reliability predictions are to be made.
- This assumption is to ensure that the data collected in that particular environment is applicable to the environment in which the reliability projections are to be made.
- Every fault has the same chance of being encountered and is of the same severity as any other faults.
- This assumption is to ensure that the various failures all have the same distributional properties.
- One severity class might have different failure rates which may require separate reliability analysis. The failures are independent.
- A failure occurs when the faults are encountered, so having independent failures simplifies the maximum likelihood estimates.

Exponential class models have two major types: binomial-type and Poisson-type. In addition to the common assumptions for the exponential class models, the binomial-type models assume that the failures are removed from the software as soon as they occur and introduce no more faults during the fix. The Poisson-type models assume that the faults remaining in the software is a Poisson random variable. The principal difference between the binomial and Poisson type models is how the remaining number of faults are treated. According to the consideration made in exponential model, binomial-type models treat the number of remaining faults as a fixed number. While in the Poisson-type models, the number of remaining faults is treated as random variable [82]. Jelinski-Moranda, Musa and Schneidewind models are
discussed in the following subsections. Table 3.1 summaries the reliability functions for these models.

3.5.1.1 Jelinski-Moranda Model

Moranda and Jelinski [86] have developed the first software reliability model that is widely known and extensively used. It forms the basis on which several other models have been developed. This model, which is a binomial-type model, assumes that: (1) The existence of an unknown fixed number of independent failures at the start of testing. (2) The faults are equally likely to cause a failure during testing. (3) Every discovered or detected fault is removed immediately without introducing any new failures. These assumptions lead to a constant hazard rate per fault that is proportional to the remaining faults in the software. In this case the failure intensity decays is exponentially in terms of cumulative failure time (Figure 3.1)

<table>
<thead>
<tr>
<th>Table 3.1: Summary of reliability functions for Jelinski-Moranda, Musa and Schneidewind Models</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Reliability Functions</strong></td>
</tr>
<tr>
<td>F(t)</td>
</tr>
<tr>
<td>f(t)</td>
</tr>
<tr>
<td>z(t)</td>
</tr>
<tr>
<td>$\mu(t)$</td>
</tr>
<tr>
<td>$\lambda(t)$</td>
</tr>
</tbody>
</table>
The model's parameters (total number of failure, N, and the hazard rate) can be estimated using maximum likelihood or least squares methods. The results of the maximum likelihood estimate are:

\[
\hat{\phi} = \frac{\eta}{\bar{N}(\sum_{i=1}^{\eta} X_i) - \sum_{i=1}^{\eta} (i-1)X_i} \quad \text{and} \quad \sum_{i=1}^{\eta} \frac{1}{\bar{N} - (i-1)} = \frac{\eta}{\bar{N}(1/\sum_{i=1}^{\eta} X_i)(\sum_{i=1}^{\eta} (i-1)X_i)}
\]

(3.1)

Although the Jelinski-Moranda model is one of the first and most used software reliability models, it has two main limitations: (1) It is not the case in real software that the remaining faults contribute with the equal amount to the failure rate. (2) During testing one can uncover imperfect fixes and that introduced new faults in the software.

### 3.5.1.2 Schneidewind's Model

As testing progresses, the error-detection process changes, so the current or recent failure counts are usually better than a past history of faults for predicting the future faults. It proposes a Poisson-type model that suggests the use of the current failure rate rather than the past failure rates, to predict the future behavior of the software under test. The numbers of faults detected are independent and the fault correction rate is proportional to the number of corrected faults. The failure intensity function is an exponentially decreasing function of time. The total number of faults \( \alpha \) is
proportional to the remaining number of faults \(\beta\), assuming that all the time intervals have the same length and the data used are failure counts, author proposes three models along with their maximum likelihood estimates:

**Model 1:** treats all data points (faults) the same

\[
\hat{\alpha} = \frac{\hat{\beta} F_\eta}{1 - \exp(-\hat{\beta} \eta)} \quad \text{and} \quad \frac{1}{\exp(\hat{\beta})} - \frac{\eta}{\exp(\hat{\beta} \eta) - 1} = \sum_{k=0}^{n-1} k \frac{f^k + 1}{F_\eta}
\]

(3.2)

**Model 2:** considers only the data points (faults) from \(s - 1\) to \(n\)

\[
\hat{\alpha} = \frac{\hat{\beta} F_{s,n}}{1 - \exp(-\hat{\beta}(\eta - s + 1))} \quad \text{and} \quad \frac{1}{\exp(\hat{\beta})} - \frac{\eta - s + 1}{\exp(\hat{\beta}(\eta - s + 1)) - 1} = \sum_{k=0}^{n-1} k \frac{f^s + \kappa}{F_s, \eta}
\]

(3.3)

**Model 3:** uses the cumulative fault counts from the intervals \(1\) to \(s - 1\) as the first data point and each fault counts for the period from \(s\) to \(n\) as the additional data points.

\[
\hat{\alpha} = \frac{\hat{\beta} F_\eta}{1 - \exp(-\hat{\beta} \eta)} \quad \text{and} \quad \frac{(s-1)F_{s-1}}{\exp(\hat{\beta}(s-1)) - 1} + \frac{F_{s,n}}{\exp(\hat{\beta}) - 1} - \frac{\eta F_\eta}{\exp(\hat{\beta} \eta) - 1} = \sum_{k=0}^{n-1} (s + \kappa - 1)f_{s+k}
\]

(3.4)

This model overcomes the first limitation of the Jelinski-Moranda model by putting more weight on the current data contribution to the failure rate. Later in the nineties, Schneidewind (1993a,1993b,1993c) identifies the optimal value of \(s\) using a variety of methods such as weighted least squares, mean square error to the next failure, and the mean square error for cumulative failures. This model has been used by IBM flight control software for the space shuttle [87] and was one of the chosen four models by AIAA's Recommended Practice for Software Reliability [88].

**3.5.1.3 Musa's Model**

Musa et al. [89] develops a model known as the basic execution time model which is the first model that uses the execution time (CPU) instead of calendar time. Musa's model is a Poisson-type model of the exponential class. The model has two components: (1) The basic execution time component: The concept of the fault reduction factor is introduce in Musa's model, which assumes the fault correction rate is proportional to the hazard rate. In addition to the parameter \(\beta_0\), the total number of expected faults, this concept of fault correction rate adds another parameter \(\beta_1 = \beta^0\). The model assumes an exponential distribution for the time to failure of each fault and
a constant per-fault hazard rate. The maximum likelihood estimates of the models' parameters \( \beta_0, \beta_1 \) are:

\[
\hat{\beta}_0 = \frac{\eta}{1 - \exp\left(-\beta_1 (t_n + x)\right)} \quad \text{and} \quad \hat{\beta}_1 = \frac{\eta t_n}{\exp\left[\beta_1 (t_n + x)\right] - 1} - \sum_{i=1}^{n} t_i = 0
\] (3.5)

(2) The basic calendar component: contains features to convert the execution time results to calendar time. Multiple measurements of the resources and personnel have to be collected and used to convert the execution time used in the first component of the model to calendar time [89].

### 3.5.2 Growth Models

Software reliability is the probability that software will not cause the failure of a product for specified time under specified conditions. The term “probability” is a function of the inputs to and use of the product, as well as a function of the existence of faults in the software. The inputs to the product will determine whether an existing fault is encountered or not. Many software reliability growth models have been developed over the years. For a detailed description of most models refer to [6]. Within these models one can distinguish two main categories: predictive models, assessment models. Predictive models typically address the reliability of the software early in the life cycle at the requirements or at the preliminary design level or even at the detailed design level in a waterfall life cycle process or in the first spiral of a spiral software development process. Predictive models could be used to assess the risk of developing software under a given set of requirements and for specified personnel before the project truly starts.

- **Assessment models** evaluate present and project future software reliability from failure data gathered when the integration of the software starts. Predictive software reliability models are few in number; most models can be categorized in the assessment category.

- **Analytical models** - analytical modeling of software reliability involves four steps. The first step is to define the assumptions associated with a software test procedure, and the second step is to develop an analytical model based on the assumptions and the test procedure. The third step is to obtain parameters for the model using the data collected and the final step is to use the model for performance predictions. Two major types of analytical models can be identified.
They are dynamic and statistical models. In a dynamic software reliability model, the time dependent behaviors of the software failures are captured in the analytic model. However, in a statistic model, no reference is made to the time dependent behavior of software failures.

- **Empirical models** – in an empirical software reliability model, a relationship or a set of relationships between software reliability measures and appropriately defined software metrics are developed using empirical results available from past data. This model can then be applied to measure software reliability for which we have the required software metrics. This, in a way, is similar to the econometric models in forecasting theory. The major issues in this modeling technique are the identification of the appropriate software metrics and the development of the right type and form of relationships between these metrics and the reliability measures. The latter issue of developing the functional relationship is referred to as specifications. The accurate and most appropriate specification of an empirical model is a key step in the use of this technique for software reliability estimation.

For critical business applications, continuous availability is a requirement and software reliability is an important component of continuous application availability. Tandem customers expect continuous availability, and our process pair technology protects us from most transient software defects. However, rare kinds of single software defects can cause a system failure. To avoid these failures and to decrease software support costs, tandem needs to deliver reliable software. Developing reliable software is one of the most difficult problems facing the software industry. Schedule pressure, resource limitations, and unrealistic requirements can all negatively impact software reliability. Developing reliable software is especially hard when there is interdependence among the software modules as is the case with much of existing software. It is also a hard problem to know whether or not the software being delivered is reliable. After the software is shipped, its reliability is indicated by the customer feedback - problem reports, system outages, complaints or compliments, and so forth. However, by then it is too late; software vendors need to know whether their products are reliable before they are shipped to customers. Software reliability models attempt to provide that information.
There are essentially two types of software reliability models - those that attempt to predict software reliability from design parameters and those that attempt to predict software reliability from test data. The first type of models are usually called "defect density" models and use code characteristics such as lines of code, nesting of loops, external references, input/outputs, and so forth to estimate the number of defects in the software. The second type of models is usually called "software reliability growth" models. These models attempt to statistically correlate defect detection data with known functions such as an exponential function. If the correlation is good, the known function can be used to predict future behavior. Software reliability growth models are the focus of this report.

Most software reliability growth models have a parameter that relates to the total number of defects contained in a set of code. If we know this parameter and the current number of defects discovered, we know how many defects remain in the code (Figure 3.2). Knowing the number of residual defects helps us decide whether or not the code is ready to ship and how much more testing is required if we decide the code is not ready to ship. It gives us an estimation of the number of failures that our customers would encounter when operating the software. This estimate helps us to plan the appropriate levels of support that will be required for defect correction after the software has been shipped and determine the cost of supporting the software.

Software reliability growth models have been applied to portions of four software releases at Tandem over the past 4 years. This research, while still experimental, has provided a number of useful results and insights into software reliability growth modeling.
Reliability is usually defined as the probability that a system will operate without failure for a specified time period under specified operating conditions. Reliability is concerned with the time between failures or its reciprocal, the failure rate. In this report we are considering data from a test environment, so we report defect detection rate rather than failure rate. Defect detection is usually a failure during a test, but test software may also detect a defect even though the test continues to operate. Defects can also be detected during design reviews or code inspections, but we do not consider those sorts of activities in this report. Time in a test environment is a synonym for amount of testing, which can be measured in several ways. Defect detection data consists of a time for each defect or group of defects and can be plotted as shown in Figure 3.3. We can derive defect detection rates from this data.

![Fig. 3.3: Example Defect Detection Data](image)

A cumulative plot of defects vs. amount of testing such as Figure 3.3 should show that the defect discovery rate decreases as the amount of testing increases. The theory is that each defect is fixed as it is discovered. This decreases the number of defects in the code, so the defect discovery rate should decrease (the length of time between defect discoveries should increase). When the defect discovery rate reaches an acceptably low value, the software is deemed suitable to ship. However, it is difficult to extrapolate from defect discovery rate in a test environment to failure rate during system operation, primarily because it is hard to extrapolate from test time to system operation time. Instead, we look at the expected quantity of remaining defects in the code. These residual defects provide an upper limit on the number of unique failures our customers could encounter in field use.
Software reliability growth models have been grouped into two classes of models: concave and S-shaped. These two model types are shown in Figure 3.4. The most important thing about both models is that they have the same asymptotic behavior, i.e., the defect detection rate decreases as the number of defects detected (and repaired) increases, and the total number of defects detected asymptotically approaches a finite value. The theory for this asymptotic behavior is that:

1. A finite amount of code should have a finite number of defects. Repair and new functionality may introduce new defects, which increases the original finite number of defects. Some models explicitly account for new defect introduction during test while others assume they are negligible or handled by the statistical fit of the software reliability growth model to the data.

2. It is assumed that the defect detection rate is proportional to the number of defects in the code. Each time a defect is repaired; there are fewer total defects in the code, so the defect detection rate decreases as the number of defects detected (and repaired) increases. The concave model strictly follows this pattern. In the S-shaped model, it is assumed that the early testing is not as efficient as later testing, so there is a ramp-up period during which the defect detection rate increases. This could be a good assumption if the first QA tests are simply repeating tests that developers have already run or if early QA tests uncover defects in other products that prevent QA from finding defects in the product being tested. For example, an application test may uncover as defects that need to be corrected before the application can be run. Application test hours are accumulated, but defect data is minimal because as defects don't count as part of the application test data. After the as defects are corrected, the remainder of the application test data (after the inflection point in the S-shaped curve) looks like the concave model.

Fig. 3.4: Concave and S-Shaped Models
There are many different representations of software reliability models. This paper uses the model representation shown in Figure 3.4. This representation shows the expected number of defects at time t and is denoted $\mu(t)$, where t can be calendar time, execution time, or number of tests executed. An example equation for $\mu(t)$ is the Goel-Okumoto (G-O) model:

$$\mu(t) = a(1-e^{-bt})$$

(3.6)

Where

- $a = \text{expected total number of defects in the code}$
- $b = \text{shape factor = the rate at which the failure rate decreases, i.e., the rate at which we approach the total number of defects}$.

The Goel-Okumoto model is a concave model, and the parameter "a" would be plotted as the total number of defects in Figure 3.4. The Goel-Okumoto model has 2 parameters; other models can have 3 or more parameters. For most models, $\mu(t) = aF(t)$, where a is the expected total number of defects in the code and F(t) is a cumulative distribution function.

Note that $F(0) = 0$, so no defects are discovered before the test starts, and $F(\infty) = 1$, so $\mu(\infty) = a$ and a is the total number of defects discovered after an infinite amount of testing. Table 3.2 provides a list of the models that were evaluated as part of this effort. A derivation of the properties of most of these models can be found in [86] [90].

<table>
<thead>
<tr>
<th>Model Name</th>
<th>Model Type</th>
<th>$\mu(t)$</th>
<th>Reference</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goel-Okumoto (G-O)</td>
<td>Concave</td>
<td>$a(1-e^{-bt})$</td>
<td>Goel [79]</td>
<td>Also called Musa model or exponential model</td>
</tr>
<tr>
<td>G-OS- Shaped</td>
<td>S-Shaped</td>
<td>$a(1-(1+bt)e^{-bt})$</td>
<td>Yamada [83]</td>
<td>Modification of G-O model to make it S-shaped(Gamma function instead of exponential)</td>
</tr>
<tr>
<td>Hossain-Dahiya/ G-O</td>
<td>Concave</td>
<td>$a(1-e^{-bt})/(1+c^{-bt})$</td>
<td>Hossain [93]</td>
<td>Solves a technical condition with the G-O model. Becomes same as G-O as c approaches 0.</td>
</tr>
<tr>
<td>Model Name</td>
<td>Model Type</td>
<td>( \mu(t) )</td>
<td>Reference</td>
<td>Comments</td>
</tr>
<tr>
<td>------------</td>
<td>------------</td>
<td>----------------</td>
<td>-----------</td>
<td>----------</td>
</tr>
<tr>
<td>Gompertz</td>
<td>S-Shaped</td>
<td>( a(b^t) a \geq 0, \ 0 \leq b \leq 1, 0 &lt; c &lt; 1 )</td>
<td>Kececioglu [91]</td>
<td>Used by Fujitsu, Numazu Works</td>
</tr>
<tr>
<td>Pareto</td>
<td>Concave</td>
<td>( a(l-(l+t/\beta)^{-\alpha}) a \geq 0, \ \beta &gt; 0, \ 0 \leq \alpha \leq 1 )</td>
<td>Littlewood [81]</td>
<td>Assumes failures have different failure rates and failures with highest rates removed first</td>
</tr>
<tr>
<td>Weibull</td>
<td>Concave</td>
<td>( a(l-e^{-b^t}) a \geq 0, b &gt; 0, c &gt; 0 )</td>
<td>Musa [87]</td>
<td>Same as G-O for c=1</td>
</tr>
<tr>
<td>Yamada</td>
<td>Concave</td>
<td>( a(1-e^{-\alpha (1-e^{-\beta t})^n}) a \geq 0, r \alpha &gt; 0, \ \beta &gt; 0 )</td>
<td>Yamada [86]</td>
<td>Attempts to account for testing effort</td>
</tr>
<tr>
<td>Yamada</td>
<td>S-Shaped</td>
<td>( a(1-e^{-\alpha (1-e^{-\beta t})^n}) )</td>
<td>Yamada [86]</td>
<td>Attempts to account for testing effort</td>
</tr>
<tr>
<td>Log Poisson</td>
<td>Infinite Failure</td>
<td>( (l/c)ln(c \alpha l+l) )</td>
<td>Musa [87]</td>
<td>Failure rate decreases but does not approach 0</td>
</tr>
</tbody>
</table>

The Log Poisson model is a different type of model. This model assumes that the code has an infinite number of failures. Although this is not theoretically true, it may be essentially true in practice since all the defects are never found before the code is rewritten, and the model may provide a good fit for the useful life of the product. The models all make assumptions about testing and defect repair. Some of these assumptions seem very reasonable, but some are questionable. Table 3.3 contains a list and discussion of these assumptions.

**Table 3.3 Software Reliability Model Assumptions**

<table>
<thead>
<tr>
<th>Assumption</th>
<th>Reality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defects are repaired immediately when they are discovered</td>
<td>Defects are not repaired immediately, but this can be partially accommodated by not counting duplicates. Test time may be artificially accumulated if a non-repaired defect prevents other defects from being found.</td>
</tr>
<tr>
<td>Defect repair is perfect</td>
<td>Defect repair introduces new defects. The new defects are less likely to be discovered by test since the retest for the repaired code is not usually as comprehensive as the original testing.</td>
</tr>
<tr>
<td>No new code is introduced during QA test</td>
<td>New code is frequently introduced throughout the entire test period, both defect repair and new features. This is accounted for in parameter estimation since actual defect discoveries are used, but may change the shape of the curve, i.e., make it less concave.</td>
</tr>
<tr>
<td>Assumption</td>
<td>Reality</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
</tr>
<tr>
<td>Defects are only reported by the product testing group</td>
<td>Defects are reported by lots of groups because of parallel testing activity. If we add the test time for those groups, we have the problem of equivalency between an hour of QA test time and an hour of test time from a group that is testing a different product. This can be accommodated by restricting defects to those discovered by QA, but that eliminates important data. This problem means that defects do not correlate perfectly with test time.</td>
</tr>
<tr>
<td>Each unit of time (calendar, execution, number of test cases) is equivalent</td>
<td>This is certainly not true for calendar time or test cases as discussed earlier. For execution time, &quot;corner&quot; tests sometimes are more likely to find defects, so those tests create more stress on a per hour basis. When there is a section of code that has not been as thoroughly tested as other code, e.g., a product that is under schedule pressure, tests of that code will usually find more defects. Many tests are rerun to ensure defect repair has been done properly, and these reruns should be less likely to find new defects. However, as long as test sequences are reasonably consistent from release to release, this can be accounted for if necessary from lessons learned on previous releases.</td>
</tr>
<tr>
<td>Tests represent operational profile</td>
<td>Customers run so many different configurations and applications that it is difficult to define an appropriate operational profile. In some cases, the sheer size and transaction volume of the production system makes the operational environment impractical to replicate. The tests contained in the QA test library test basic functionality and operation, error recovery, and specific areas with which we have had problems in the past. Additional tests are continually being added, but the code also learns the old tests, i.e., the defects that the old tests would have uncovered have been repaired.</td>
</tr>
<tr>
<td>Failures are independent</td>
<td>Our experience is that this is reasonable except when there is a section of code that has not been as thoroughly tested as other code, e.g., a product behind schedule that was not thoroughly unit tested. Tests run against this section of code may find a disproportionate share of defects. [Musa, 87, P242] has a detailed discussion of the independence assumption.</td>
</tr>
</tbody>
</table>

It is difficult to determine how the violation of the model assumptions will affect the models. For instance, introducing new functionality may make the curve less concave, but test reruns could make it more concave. Removing defects discovered by other groups comes closer to satisfying the model assumptions but makes the model less useful because we are not including all the data (which may also make the results less
statistically valid). In general, small violations probably get lost in the noise while significant violations may force us to revise the models.

3.5.3 Multi-Stage Models
One of the assumptions made by all the models is that the set of code being testing is unchanged throughout the test period. Clearly, defect repair invalidates that assumption, but it is assumed that the effects of defect repair are minimal so that the model is still a good approximation. If a significant amount of new code is added during the test period, there is a technique that allows us to translate the data to account for the increased code change. Theoretically, the problem is that adding a significant amount of changed code should increase the defect detection rate. Therefore, the overall curve will look something like Figure 3.5, where $D_1$ defects are found in $T_1$ time prior to the addition of the new code and an additional $D_2-D_1$ defects are found in $T_2-T_1$ time after that code addition. The problem is to translate the data to a model $\mu(t)$ that would have been obtained if the new code had been part of the software at the beginning of the test. Let $\mu_1(t)$ model the defect data prior to the addition of the new code, and let $\mu_2(t)$ model the defect data after that code addition. The model $\mu(t)$ is created by appropriately modifying the failure times from $\mu_1(t)$ and $\mu_2(t)$. This section describes how to perform the translation assuming $\mu(t)$, $\mu_1(t)$, and $\mu_2(t)$ are all G-O models. In theory, this technique could be applied to any of the models in Table 3-1, including the S-shaped models.

![Fig. 3.5: Two Stage Model Transformation](image)
3.5.4 Duane Reliability Growth Model

Over the years many mathematical models for reliability growth have been developed. In 1964, J.T. Duane [91] was first person to report the most commonly accepted pattern for reliability growth. The Duane model basically is a graphical approach to perform analysis of reliability growth data and is simple and straightforward to understand. Nonetheless, the two important benefits of this approach are as follows:

- The straight line used by the Duane plot can be fitted by eye to the data points.
- Various facts can be depicted by the Duane plot which otherwise could be hidden by statistical analysis. For example, even though the application of a goodness-of-fit test may conclude the rejections of a certain reliability growth model, it will not provide any possible reasons for the rejection. On the other hand, a plot of the same data might provide some possible reasons for the problem.

In contrast, the drawbacks of this model are the reliability parameters cannot be estimated as well in comparison to a statistical model and no interval estimates can be calculated.

3.5.5 Army Material System Analysis Activity (AMSAA) Model

This is another model that can be used to track reliability growth within test phases. The model allows, for the purpose of reliability growth tracking, the development of rigorous statistical procedures. The following two assumptions are associated with this model:

- Reliability growth can be modeled as a non-homogeneous Poisson process (NHPP) within a test phase.
- On the basis of failures and test time within a test phase, the cumulative failure rate is linear on log-log scale.

3.5.6 Schick-Wolverton Model [144]

George Schick and Ray Wolverton have developed a model that assumes a Raleigh distribution for the time between error occurrences, as against a proportion distribution. The Raleigh distribution proposed which error rate can be comparative

---

8 http://en.wikipedia.org/wiki/United_States_Army_Materiel_Systems_Analysis_Activity
for the number of errors enduring and addition that errors have been instantly removed when it will be discovered.

\[
MTTF (t) = \sqrt{\frac{\pi}{2K(ETF - P)}}
\]

\[
R(t) = \exp\left[-\frac{k(ETF - p)t^2}{2}\right]
\]

(3.8)

Two unknowns of the Schicks-Wolverton model are solved for as follows:

\[
k = \frac{2ED(t)}{i=1} \sum \left[ETF - (i - 1)\right]^2
\]

\[
ED(t) \sum_{i=1}^{ED(t)} \frac{1}{ETF - (i - 1)} = \frac{K \sum_{i=1}^{ED(t)} t^2}{2}
\]

(3.9)

This model assumes that \( p = ED(t) = EC(t) \); therefore, there is only one error occurrence per interval and \( t \) is the cumulative time to detect \( ED(t) = p = EC(t) \), the number of errors.

### 3.5.7 Leone Test Coverage Model

We have presented a replacement methodology of estimating the reliability of software primarily based on the completion and coverage of bound development and take a look at tasks. This software reliability model, the Leone test coverage model, is relatively simple to implement, lending itself to be a useful management, development, or testing tool, as well as a reliability measure.

Every input for this model we collected throughout testing and therefore ought to be readily accessible if effective software engineering practices are in place. If they are not, this model can aid in mentioning the specified areas of improvement with respect to software progress as well as engineering. This quantity model can be microscopic; it will take within consideration the meticulous construction of the software, even as

---

9 Ann Marie Neufelder, Ensuring Software Reliability,
critical macroscopic models, which have been reflected on solely the errors, have been detected.

There are four important inputs models are as following:
1. The proportion of total executable lines of code that’s tested and validated.
2. The proportion of open source programs are analyzed and validated.
3. The proportion of function/requirements that are tested and validated.
4. The proportion of hazard takes a look at cases that are tested and validated. These are take looks at cases proving that the software does not perform an undesirable perform. Undesirable and sudden perform are information corruption, failure of the software to detect erroneous conditions, software detecting incorrect conditions that is not erroneous, software perceiving erroneous conditions that is not erroneous and additional data on constructing these varieties of take a look at cases.

There are four essential inputs for these models are lines of executable code validated, freelance check ways validated, functions/requirements, and hazard check cases validated. Four weighting parameters are used. Every of the four inputs are determined from numerous sources, as well as automated tools and software reviews. The chance of the software performing successfully as per the software needs at some purpose in time is expressed as:

\[
F_1 = W_1 \times (\text{percentage of total executable lines of code that have been tasted during any integration/systems/acceptance test and validated})
\]

\[
F_2 = W_2 \times (\text{percentage of total independent test paths that have been tasted during any unit/integration/systems/acceptance test and validated})
\]

\[
F_3 = W_3 \times (\text{percentage of total function/requirements that have been tested during any unit/integration/systems/acceptance test and validate})
\]

\[
F_4 = W_4 \times (\text{percentage of total hazard test paths that have been tested during any unit/integration/systems/acceptance test and validate})
\]

\[
R = (F_1 + F_2 + F_3 + F_4) / 4
\]

\[
W_1 + W_2 + W_3 + W_4 = 4
\]
Note the definition “any unit/integration/systems/acceptance test.” This implies that some executable line of code can be tested in the same test case as some independent path and as some function/requirement. The ideal situation would be to have the minimum number of test cases covering the most amount of source lines of code, test paths, and functions.

Inspection of the model has shown that inefficient check coverage can in all probability not manufacture fascinating results. It is necessary to recollect that solely those tests that are validated as performing properly are counted. Any check case that didn’t perform properly would have to be compelled to be corrected and revalidated to be counted within the model.

The greatest technique to estimate the proportion of lines of code tested will go to get or develop an automatic tool that may count all of the overall lines of code within the supply models. This is often fairly simple to develop and will be used anyway for software management. Throughout development of check cases the lines of code executed by that check case can be used for software management. Throughout development of check cases the lines of code executed by that check case are often outlined. Some check tools can even count the lines of code that are tested.

In brief summarized, the number of freelance check methods is equal to the amount of call points within the logic of the code and one. The choice points in logic are that if, case, repeat, while, loop, and alternative statements that cause the logic of the software to branch in one or additional directions. A check case should be developed for every branch in logic. Keep in mind that the amount of freelance check cases isn’t equal to the amount of all potential methods. The excellence should be clear, since for a medium- or large-sized software program the whole range of all potential methods would be, for all sensible functions, immeasurable in terms of testing. These freelance check cases would already be developed if structured style, code, test, and maintenance are used.

Determine the number of functions tested might not be as straight-forward as measuring the opposite 2 parameters necessary for the model. We suggest a check case for each written demand and for each practical check case.
The hazards take a look at cases are determined as of a mistake tree analysis or a failure modes effect and criticality analysis (FMECA).

The influence of parameters is used when the contribution of every parameter to the reliability of the software is thought to be not equal. If you are doing not recognize the relative relationships between every of the 3 parameters and their impact on software reliability, then assume every weight equals one. On the opposite hand, if you recognize from past expertise or historical information that one or a lot of those factors contribute to the reliability of the software quite the others, and then set the weights accordingly.

Software that are being developed for the major time or is new technology can most likely have the function/requirements parameter weighted additional heavily than the others. Software that’s intensely user interactive may have that weight be heavier. However, software that’s clearly outlined however structurally (not functionally) advanced would most likely have the check path parameter weighted to be heavier. Software that’s safety or mission vital could have the fourth parameter weighted additional heavily.

For any reasonably application testing, each line of supply code can in all probability be necessary with respect to those 3 parameters. The Leone check coverage model is comparatively easy to implement and may and will be implemented with alternative style and check methodologies and tools, also as getting used as a reliability measurement. Knowledge information used as inputs to the model are data that will be readily accessible, notably if the software is being designed and tested in step with structured ways.

If structured technique does not seem to be used, this model can facilitate to isolate areas for improvement. This model differs from a number of the opposite existing software reliability models in that it makes an attempt to isolate how software is or becomes reliable, as against estimating the software’s reliability primarily based on error counts that don’t take under consideration how or why the error occurred.
3.5.8 Error Seeding Models

Seeding was performed the inserting errors in software so as to estimate the whole variety of inherent errors within the software. We have powerfully suggested that seeding doest not be executed. However, seeding has mentioned as a result of there are different models that were tailored from the seeding concept.

Oddly enough, the error seeding concept originated with fish tagging. So as to see what percentage fish were in an exceedingly body of water, several fish be tagged. The ratio of the fish will be caught that might be tagged to the complete variety of tagged fish will be unspecified to be stable because the relationship among the whole fish caught and therefore the total variety of fish within the pond.

Error seeding models need that errors be inserted into the software throughout the testing section. The errors seeding should be representative of real errors that might be found throughout traditional testing. Seeded errors should be randomly chosen and should cowl the supply code uniformly. This can be a contradiction terms while random involve the portion of contribution code where the error can be included be unknown. Blocking could be exploited to declare that errors are accidental and does cover a cross part for the supply code. Blocking suggests that dividing the supply code into sections and then randomly inserting errors into every section.

These are the parameters of the seeding model are as follows:

\[
V = \text{total number of seeded errors} \\
v = \text{number of seeded errors that were detected by the testers} \\
ED(t) = \text{nonseeded errors detected by the testers} \\
ETV = \text{total number of inherent nonseeded errors} \\
\frac{ED(t)}{ETV} = \frac{v}{V}
\]

The connection between seeded errors found and seeded errors known to exist within the software is assumed to be equal to the link of variety of nonseeded errors found to the entire number of errors that exist. The parameter N is that the solely unknown during this model. If the testers have found solely a little portion of the entire seeded errors that were inserted, then the model assumes that solely a little portion of all of the inherent errors has been found.
Seeded errors should not be obvious to the testers or the users or the seeding model is useless. The seeded errors should even be documented and removed as soon as attainable in order that no new errors are generated by these seeded errors. This presents a much bigger drawback than could also be imagined.

Assume that the error is randomly and discretely inserted into the software and also the checkers test using this version of software. Currently allow us to say that at the top of testing we tend to should assure that each one the seeded errors are removed. It is going to appear that the plain resolution is to easily edit the supply code and take away the errors. However, it is probable that in this point changes are created to different components of the code which will have an impression on the code within which the error was inserted. It is conjointly potential that these seeded errors prevented some real errors from being detected. Therefore, we’d be forced to make a decision between recalling the software in its original kind before the error insertion and creating all the error corrections that were created throughout testing to the present version, or making an attempt to get rid of the errors from the tested version and taking an opportunity on missing some or adversely affecting code that was already tested. In either case we’d be doing additional work than necessary or fascinating and would be taking a giant risk of truly adversely affecting the reliability of the software when removing the errors from it.

Management additionally because the client might not just like the plan of errors being inserted into software. The testers themselves might not be overjoyed if they understand they are testing software with unnatural errors in it. For all of those reasons, seeding isn’t instructed by this author as a method of estimating software reliability. However, there’s a model based mostly on the seeding concept which will be instructed as an attainable one to be implemented.

### 3.5.9 Dual Test Group Model

The dual check cluster model is applied for several years as a statistical tool before being applied to software. The twin check cluster model assumes that the particular seeding method simply outlined is replaced with 2 index pendent simultaneous check teams. Rather than relating the seeded errors found to the non-seeded errors found, we are going to compare the errors found by one check cluster with errors found by another equally competent check cluster.
The first assumptions of the model will that the 2 check teams are regarding equal in expertise and range. It conjointly assumes that every check cluster is testing identical version of software. The check teams should be freelance in that they are not tuned in to what errors are detected by the opposite check cluster or maybe of errors or rate of errors found by the opposite test group.

If these assumptions hold true, then one in every of the subsequent is assumed to occur:
1. Each team can uncover equivalent errors if there aren’t several residual errors left within the software.
2. Each team can uncover completely different errors if there are several residual errors within the software.

We should track the quantity of errors that are commonly found by each take a look at cluster one and take a look at cluster 2; see Figure 3.6 for a graphical illustration of the model.

The parameters of the twin take a look at cluster are outlined as:

\[ ETV = \frac{ED_{12}}{E_1 + E_2} \]  

(3.10)

where
- \( ED_1 \)=the number of errors exposed solely by test group 1
- \( ED_2 \)=the number of errors exposed solely by test group 2
- \( ED_{12} \)=the number of errors originate by both test group 1 and 2
- \( E_1 = ED_{12}/ED_2 \)
- \( ED_2 = ED_{12}/ED_1 \)

One downside to the present model is that the case when every take a look at cluster inputs fully totally take a look at cases testing different parts of the code. Imagine take a look at cluster one take a look testing half the subprograms and test cluster two testing the opposite 0.5. There would in all probability not be several common errors found, notably if the planning were structured to be modular, because it ought to be. During this case the model would be pessimistic since each take a look at teams combined might have found several errors however the quantity found commonly is tiny (Figure 3.7).
Fig. 3.6: Double test group model-basic assumptions of the model.

Fig. 3.7: Double test group model-cases of efficient testers and test cases with no coverage overlap.

Another downside to the present model is that if each take a look at team’s input identical take a look at cases, then the model might not be correct since some tests could also be fully overlooked by each team. The model can in all probability be optimistic during this case (Fig. 3.8).

If the belief of equal expertise among take a look at teams isn’t valid, then its attainable the model are pessimistic. If take a look at cluster one is far additional capable of finding errors than take a look at cluster two, then the amount of commonly found errors might not be terribly high, inflicting the prediction of the full variety of inherent errors will be high.

3.5.10 Testing Success Model

The testing success model has been by so much the only model presented during this book. The model is described as follows:

$$R = \frac{S}{N}$$

(3.11)
where $S$ might be the total variety of successfully executed check cases and $N$ is that the total variety of check cases executed. This model makes the fundamental assumption that the chance of the software performing successfully is equal to the chance of implementing a successful check case. For this model to be comparatively correct needs that a hit or fail standing be collected for each check case executed.

If the software is fastened, retested, and take a look at cases that previously weren’t successful don’t seem to be successful, then $S$ would wish to be updated whereas $N$ remained constant, unless new take a look at cases were introduced.

The take a look at cases themselves would wish to mirror the profile of the top user and would need to utterly cowl the supply code. It is attainable to use blocking techniques that may make sure that the supply code is roofed to some extent. Blocking implies that the supply code is split into parts, like the first functions, which take a look at cases are drawn randomly from every of the parts. Therefore, the take a look at cases are randomly chosen however additionally cowl all of the software functions.

This model could offer an estimate of how reliable the software is at this time; but, it’ll not project the reliability of the software at some future time.

### 3.5.11 Weibull Model

Where $S$ is that the total variety of number successfully executed check cases and $N$ is that the total variety of check cases executed.

This model makes the essential assumption that the chance of the software performing successfully is equal to the chance of implementing a successful take a look at case. For this model to be comparatively correct needs that successful or fail standing be collected for each take a look at case executed. If the software is mounted, retested, and take a look at cases that previously weren’t successful don’t seem to be successful, then $S$ would want to be updated whereas $N$ remained constant, unless new take a look at cases were introduced.

The test cases themselves would wish to replicate the profile of the tip user and would need to utterly cowl the supply code. It is doable to use blocking techniques which will make sure that the supply code is roofed to some extent. Blocking implies that the
supply code is split into parts, like the first functions, which check cases are drawn randomly from every of the parts. Therefore, the check cases are randomly chosen however additionally cowl all of the software functions.

This model might provide an estimate of how reliable the software is at the present time; but, it’ll not project the reliability of the software at some future time.

Another reliability model is that the Weibull model. This model supposed a Weibull allotment of software faults. One advantage of using another

![Diagram](image)

**Fig. 3.8** Dual test group model-case of inefficient testers or incomplete test coverage.

![Diagram](image)

**Fig. 3.9** Dual test group model-case of test groups not equal in number or experience.

Model is that the amendment in fault density could also be positive, negative, or constant. Its parameters are:

\[
MTTF = \frac{b}{a} \Gamma \left( \frac{1}{a} \right)
\]

\[
R(t) = \exp \left[ \frac{t^c}{b} \right]
\]
where $a > 1$, or $a < 1$, or $a = 1$ and $b$ is a constant of proportionality. The terms $a$ and $b$ must be determined using graphical procedures similar to the procedure used to estimate $k$ and ETF or ETV. We have been currently exploring this model’s application to real software development environments.

### 3.6 Predictive Models

The models we exposed thus far are estimative models, as critical predictive. The predictive models use empirical as critical project knowledge to predict, before coding even begins, what the reliability is the Mil-Hdbk 217-F uses empirical knowledge to predict the reliability of electronic parts primarily based on some characteristics of the element. RADC has determined some relationships between software characteristics and reliability within the technical report TR-87–171.

In this research there are three areas that verify software reliability is the appliance sort, development surroundings, and characteristics of the software.

$$R = A \times D \times S$$

where

$$S = SA \times ST \times SQ \times SL \times SM \times SX \times SR$$

and where $R$ is in conditions of faults per executable code line, $A$ is decided by employing a look-up chart, and $D$ is decided by answering queries in a very checklist to see how structured the event organization is. The term $SA$ is a sign of how software anomalies are managed; $ST$ is that the necessities traceability indicator; and $SQ$ is that the quality indicator. Checklists are used to see these values. Further, $SL$ is that the language indicator, $SM$ is that the modularity indicator, $SX$ is that the size indicator, and $SR$ is that the review indicator. These values are determined by look-up charts and checklists.

As mentioned in previous, we have not found that errors per lines of code may be valid software reliability metric. Sadly, however, this prediction document is currently one in every of solely some prediction techniques usually out there. A corporation is also able to confirm its own empirical values, and thus its own predictive measurement, by collecting knowledge on its own programs and determining what relationship, if any, bound characteristics have with reliability. RADC collected
knowledge on the 9 characteristics shown of this model. However, there is also and possibly are a lot of characteristics which will be quantified. The advantage to using the RADC pointers is that RADC has collected knowledge on an outsized cross section of comes. The disadvantage is that those comes might not represent your software project.

There is some skepticism in business that it is going to not be doable to predict software reliability before it is developed. At now in time that argument is also valid. However, as additional information is collected on many sorts of software, and additional analysis is conducted not solely on software reliability however on software engineering, it is going to become doable to predict reliability for software with some confidence. The forecast methods for electronic mechanism developed and it still developing during this manner.

3.7 Theoretical Comparison of Techniques

This section compares the three parameter estimation techniques from a theoretical perspective. It focuses on their ease of use, confidence interval shape, and parameter scalability. Since optimization packages are readily available, factors like maximum likelihood, classical least squares, and alternative least squares are all straightforward to solve. However, maximum likelihood only applies to the G-O model, and a new maximum likelihood technique must be derived for each software reliability growth model. These formulations can be difficult to derive, especially for the more complex models. Classical least squares apply to the exponential family of models that includes the G-O model. It is fairly easy to modify this equation for similar models. However, alternative least squares are the easiest to use since it applies to any software reliability growth model, so the alternative least squares method is the easiest to apply. Confidence intervals for all of the estimation techniques are based on assuming that estimation errors are normally distributed.

For the maximum likelihood technique, this assumption is good for large sample sizes because of the asymptotically normal properties of this estimator. However, it is not as good for the smaller samples that we typically have. Nevertheless, the maximum likelihood technique provides the best confidence intervals because it requires less normality assumptions and because it provides asymmetric confidence intervals for
the total defect parameter. The lower confidence limit is larger than the number of experienced defects, and the upper confidence limit is farther from the point estimate than the lower confidence limit to represent the possibility that there could be many defects that have gone undetected by testing. Conversely, for the least squares techniques, the lower confidence limit can be less than the number of experienced defects (which is obviously impossible), and the confidence interval is symmetric. Also, additional assumptions pertaining to the normality of the parameters is necessary to derive confidence intervals for the least squares techniques. The transformation technique consists of multiplying the test time by an arbitrary (but convenient) constant and multiplying the number of defects observed by a different arbitrary constant. For this technique to work, the predicted number of total defects must be unaffected by the test time scaling and must scale by the same amount as the defect data. For example, we may experience 50 total defects during test and want to scale that to 100 for confidentiality or ease of reporting. To do that transformation, the number of defects reported each week must be multiplied by 2. If 75 total defects were predicted by a model based on the unscaled data, then the total defects predicted from the scaled data should be 150.

Reliability growth models are categorized as hardware models and software models.

- **Hardware models** – using for electronics systems, functions blocks, electronics components without connections to software. Basic terms are a reliability operating state and a failure. There are two basic types of failures during the development phase – random failures, early failures.

- **Software models** – using for software.

- **Reliability growth models** are generally categorized as probabilistic models and statistical models.

- **Probabilistic reliability growth models** – because of no unknown parameters associated with these models, the data obtained during the program cannot be incorporated.

- **Statistical reliability growth models** – unknown parameters are associated with these models. In addition, these parameters are estimated throughout the development of the product in question.

- **Time independent reliability growth models** – number of failures or repairs in definite time interval are not depended on time.
• **Time dependent reliability growth models** – a reliability growth model is function of time.

• **Continuous reliability growth models** – these are time models.

• **Discrete reliability growth models** – these are useful for unrecoverable objects, there are two discrete states – a reliability operating state or a failure.

• **Classically reliability growth models** (A brief survey of reliability growth models, [92]) – mathematical equipment is theory of probability, Weiss RGM (1956), Lloyd-Lipow RGM (1962), Chernoff and Woods RGM (1962), Wolmano RGM (1963), Barlow-Scheuer RGM (1966).


• **Unconventional reliability growth models** – there are all reliability growth models, for which is no existing possibility to arrange in classification categories.

### 3.8 Model Comparison

This chapter presented a range of reliability models that are employed in trade. The models were additionally evaluated for easy implementation, practicality, and accuracy. An outline chart of those models is presented in Table 3.4. The info that has got to be collected are shown for every of those models in Table 3.5.

<table>
<thead>
<tr>
<th>Model Name</th>
<th>Assumptions</th>
<th>Unknowns</th>
<th>Cycle</th>
<th>Ease</th>
</tr>
</thead>
<tbody>
<tr>
<td>Musa Basic</td>
<td>1. Finite number of inherent errors&lt;br&gt;2. Constant error rate over time&lt;br&gt;3. Exponential</td>
<td>ET</td>
<td>After integration</td>
<td>E-F</td>
</tr>
<tr>
<td>Model Name</td>
<td>Assumptions</td>
<td>Unknowns</td>
<td>Cycle</td>
<td>Ease</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>----------</td>
<td>------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Musa Logarithmic</td>
<td>1. Infinite number of inherent errors</td>
<td></td>
<td>Unit to system test</td>
<td>E-F</td>
</tr>
<tr>
<td></td>
<td>2. Logarithmic</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. Changing error rate over time</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shooman</td>
<td>1. Finite and constant number of inherent errors</td>
<td>ET</td>
<td>After integration</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td>2. Errors corrected as soon as detected</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. Constant error rate over time</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4. Binomial exponential</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jelinski-Moranda</td>
<td>Same as Shooman except when: Error rate is related to errors corrected</td>
<td>ET</td>
<td>After integration</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td>1. ET is not fixed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2. Errors are not assumed to be corrected as soon as discovered</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lipow</td>
<td>Same as Shooman except that:</td>
<td>ET</td>
<td>After integration</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td>1. ET is not fixed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2. Errors are not assumed to be corrected as soon as discovered</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Goel-Okumoto</td>
<td>1. Nonhomogeneous error distribution</td>
<td>a,b</td>
<td>After integration</td>
<td>F-M</td>
</tr>
<tr>
<td></td>
<td>2. Errors may be generated due to maintenance</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. Exponential Poisson</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jelinski-Moranda</td>
<td>1. Error data are in periodic form</td>
<td>D, k</td>
<td>Any phase</td>
<td>F-M</td>
</tr>
<tr>
<td>geometric</td>
<td>2. Exponential</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Duane Growth</td>
<td>1. Exponential</td>
<td>Growth rate</td>
<td>After integration</td>
<td>F</td>
</tr>
<tr>
<td>Schick Wolverton</td>
<td>1. Raleigh distribution</td>
<td>ET</td>
<td>After integration</td>
<td>M</td>
</tr>
<tr>
<td></td>
<td>2. Used in noise or communications</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Seeding</td>
<td>1. Errors inserted in code randomly</td>
<td>ET</td>
<td>After integration</td>
<td>D</td>
</tr>
<tr>
<td></td>
<td>2. Relationship of seeded bugs to non-seeded bugs is same as between bugs found and total bugs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. Do not use this</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dual test</td>
<td>1. Two independent concurrent test groups testing identical code but not identical test cases</td>
<td>ET</td>
<td>After integration</td>
<td>M</td>
</tr>
<tr>
<td>Model Name</td>
<td>Assumptions</td>
<td>Unknowns</td>
<td>Cycle</td>
<td>Ease</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>----------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>Weibull</td>
<td>Weibull distribution</td>
<td>a,b</td>
<td>After code</td>
<td>F</td>
</tr>
<tr>
<td>Predictions</td>
<td>Prediction of reliability based on collection data</td>
<td>Various parameters</td>
<td>Any phase</td>
<td>E-D</td>
</tr>
<tr>
<td>Testing success</td>
<td>1. Reliability is directly related to ratio of successful test runs to total test runs  2. Test runs must be cover code to be accurate</td>
<td>Test data</td>
<td>Demo test</td>
<td>E</td>
</tr>
<tr>
<td>Test coverage</td>
<td>Validation of test paths, lines of codes, and functionality has a direct relationship to reliability</td>
<td>Test data</td>
<td>Unit to field test</td>
<td>F</td>
</tr>
</tbody>
</table>

**Key (Unknowns):** ET-Error Tracking; a-Constant; b-Errors per time  
**Key (Ease):** E-easy; F-fair; M-moderate; D-difficult

**Table 3.5 Required Data for Each Software Reliability Model**

<table>
<thead>
<tr>
<th>Model name</th>
<th>Data to be collected for model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Musa basic</td>
<td>Error count, time of error detection</td>
</tr>
<tr>
<td>Musa logarithmic</td>
<td>Error count, time of error detection</td>
</tr>
<tr>
<td>Shooman</td>
<td>Error count, time of error detection</td>
</tr>
<tr>
<td>Jelinski-Moranda</td>
<td>Error count, time of error detection</td>
</tr>
<tr>
<td>Lipow</td>
<td>Error count, time of error detection, intervals</td>
</tr>
<tr>
<td>Goel – Okumoto</td>
<td>Error count, time of error detection, intervals</td>
</tr>
<tr>
<td>Jelinski-Moranda geometric</td>
<td>Error count, during some intervals</td>
</tr>
<tr>
<td>Duane growth</td>
<td>MTTF or failure rate over time</td>
</tr>
<tr>
<td>Schick – Wolverton</td>
<td>Error count, time of error detection</td>
</tr>
<tr>
<td>Dual test</td>
<td>Common error count, error count from both groups</td>
</tr>
<tr>
<td>Weibull</td>
<td>Error count, time of error detection</td>
</tr>
<tr>
<td>Predictions</td>
<td>Many forms of empirical data</td>
</tr>
<tr>
<td>Testing success</td>
<td>Number of test runs successful, total number of runs</td>
</tr>
<tr>
<td>Test coverage</td>
<td>Percent functions tested, % paths tested, % source tested</td>
</tr>
</tbody>
</table>
3.9 Open Source Software

Open Source Software (OSS) in general refers to any software whose source code is freely available for distribution. The success and benefits of OSS can be attributed to many factors such as code modification by any party as the needs arise, promotion of software reliability and quality due to peer review and collaboration among many volunteer programmers from different organizations, and the fact that the knowledge-base is not bound to a particular organization, which allows for faster development and the likelihood of the software to be available for different platforms. White-box and black-box models are two approaches for predication of software reliability. The white-box models attempt to measure the quality of a software system based on its structure that is normally architected during the specification and design of the product [145].

3.10 About Apache OFBiz:

Apache Open for Business (OFBiz) is an open source enterprise automation system providing, among many other features, ERP functionality. The system attempts to provide reusable modules for common business functions and a variety of paths to develop custom business logic and connect with external systems. Similar to Seam applications, OFBiz has a three-layer architecture. The presentation layer is based on the Model-View-Controller pattern and makes extensive use of the Decorator pattern for reuse of design elements. OFBiz integrates with many presentation-tier technologies, including Tomcat or Jetty as the Web server, the Freemaker template engine, the JasperReports report engine, the JavaPOS point-of-sale device API, the XML User Interface (XUI) rich client platform, and the BeanShell Java scripting language. The business logic layer utilizes the Service Oriented Architecture pattern in which application developers organize the business logic as a set of reusable services. Services are implemented as scripts using an OFBiz-specific XML scripting language, Java via BeanShell, or other languages such as Python via the Bean Scripting Framework (BSF). Services can be easily exported as SOAP Web service or Java RMI endpoints. The persistence layer provides a database-independent entity persistence engine to the other layers using an XML specification of the database mapping and an API following the Table Data Gateway pattern.
By open source enterprise automation we mean: Open Source ERP, Open Source CRM, Open Source E-Business / E-Commerce, Open Source SCM, Open Source MRP, Open Source CMMS/EAM, and so on. Apache OFBiz is a foundation and starting point for reliable, secure and scalable enterprise solutions. Use it out-of-the-box (OOTB) or customize to suit even your most challenging business needs.

Architecture is really just a fancy word for the organization and composition of application components. There are many different "tools" available as part of Java, J2EE and the OFBiz Core Framework that can be used together to efficiently and effectively organize data and business logic, to provide interfaces to other systems, and to create user interfaces for humans to interact with the system. With OFBiz in place, you can get started right away and then grow your operations as your business grows, without the huge deployment and maintenance costs of traditional enterprise automation systems. The OFBiz Framework goes one step further and provides “best practices” tools for each of these needs. These best practices tools follow somewhat of an 80/20 intent in that they should be applicable about 80% of the time and when they do apply should require about 20% of the work that alternatives would require. In addition to the primary best practices tools the OFBiz Framework also has recommendations for various secondary tools such as Java Methods, FreeMarker Templates, and BeanShell scripts that can be used when more flexibility or control are needed. The framework has touch points that allow the use of other tools in any place, and combined with the best practice and secondary tools. As a general rule for OFBiz (and a nice principle for life in general) the less code the better and minor compromises in functionality are okay for the sake of less code (easier and cheaper to maintain, customize, etc, etc). Often these compromises in functionality are really a good thing for the sake of consistency; it is easier for both end-users and customizers to understand the system as a whole.
Table 3.6 OSS Opportunity and Threat

<table>
<thead>
<tr>
<th>Factor</th>
<th>IS Client</th>
<th>SW Producer</th>
<th>SW Distributor</th>
<th>IT Consultant</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• It is possible to continuous development by the client</td>
<td>• Possibility of using open source versions</td>
<td></td>
<td>• More complex technology</td>
</tr>
<tr>
<td></td>
<td>• Free</td>
<td>• In marketing strategy</td>
<td></td>
<td>• Corresponding to entry barrier</td>
</tr>
<tr>
<td></td>
<td>• Public Source Code</td>
<td>• Possible of using</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Possibility of depending not from just one (or small number) of technology suppliers</td>
<td>• Open source versions in development strategies</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Many disperse solutions</td>
<td>• Possibility of losing control over the software</td>
<td>• Distributors whose business is supported in commissions based in license had to change their business.</td>
<td>• Widely used technology</td>
</tr>
<tr>
<td></td>
<td>• Less expensive commercial ERP Software</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• No guaranties supplied by SW producers</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Dependency from technical workers</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
3.11 Characteristics of OSS

OSS has many characteristics. The important ones are:

- It is generally acquired freely
- Manufacturer or developer has no right to claim royalties on the distribution or use
- Source code is accessible to the user and distributed with the software
- No denial to an individual or to a group to access source code of the software
- It has provision of modifications and derivations under the programme’s original name
- Rights of facilities attached to the programme must not depend on the programme’s being part of a particular software distribution
- Licensed software cannot place restriction on other software that is distributed with it
- Distribution of License should not be specific to a product and License should be technology neutral, etc.
Important points should be taken into consideration while choosing an OSS are:

- Reputation of the software
- Monitor ongoing efforts and local usability
- Support for Standards and Interoperability
- User support
- Discussion Forums
- Check versions, documentation available for the software
- Skills of the workers
- Availability and conditions of the license and the hidden cost involve
- Commercial support for operability, etc.