10.5.2 Perform Causal Analysis

The Pareto chart helps identify the main types of defects that have been found in the project so far, and are likely to be found in the rest of the project unless some action is taken. These can be treated as "effects" which we would like to minimize in future. For reducing these defects, we have to find the main causes for these defects and then try to eliminate these causes. Cause-effect (CE) diagram is a technique that can be used to determine the causes of the observed effects [119, 139]. The understanding of the causes helps identify solutions to eliminate them.

The building of a CE diagram starts with identifying an effect whose causes we wish to understand. In the example above, the effect could be "too many GUI errors." To identify the causes, first some major categories of causes are established. For manufacturing, these major causes often are manpower, machines, methods, materials, measurement, and environment. One possible standard set of major causes in software can be process, people, technology, and training (this is used in an organization [97]). With the effect and major causes, the main structure of the diagram is made—effect as a box on the right connected by a straight horizontal line, and an angular line for each major cause connecting to the main line.

For analyzing the causes, the key is to continuously ask the question "Why does this cause produce this effect?" This is done for each of the major causes. The answers to these questions become the sub-causes and are represented as short horizontal lines joining the line for the major cause. Then the same question is asked for the causes identified. This "Why-Why-Why" process is repeated till all the root causes have been identified, i.e. the causes for which asking a "Why" does not make sense. When all the causes are marked in the diagram, the final picture looks like a fish-bone structure and hence the cause-effect diagram is also called the fish-bone diagram, or Ishikawa diagram after the name of its inventor.

The main steps in drawing a cause-effect diagram are as follows[139]:

1. Clearly define the problem (i.e., the effect) that is to be studied. For defect prevention, it typically will be "too many defects of type X".

2. Draw an arrow from left to right with a box containing the effect drawn at the head. This is the backbone of this diagram.

3. Decide the major categories of causes. These could be the standard categories or some variation of it to suit the problem.

4. Write these major categories in boxes and connect them with diagonal arrows to the backbone. These form the major bones of the diagram.

5. Brainstorm for the sub-causes to these major causes by asking repeatedly, for each major cause, the question, "Why does this major cause produce the effect?"

6. Add the sub-causes to the diagram clustered around the bone of the major cause. Further sub-divide these causes, if necessary. Stop when no worthwhile answer to the question can be found.

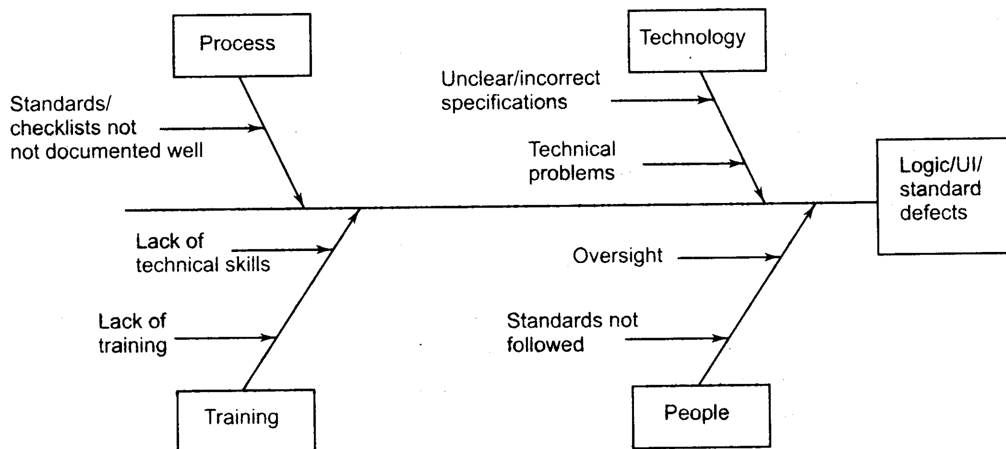

Figure 10.13: Cause-effect diagram for the example.

Once the fishbone diagram is finished, we have identified all the causes for the effect under study. However, most likely the initial fishbone diagram will have too many causes. Clearly, some of the causes have a larger impact than others. Hence, before completing the root cause analysis, the top few causes are identified. This is done largely through discussion. For defect prevention, this whole exercise can be done for the top one or two categories of defects found in the Pareto analysis.

The fish bone diagram for this example is shown in Figure 10 . In this analysis, causes of all the three major types of defects were discussed together. Hence, our effect is "too many logic/GUI/standards defects." When we asked the question "why do people and training cause too many logic or GUI or standards defects," some of the (almost obvious) reasons came out—lack of training, oversight, lack of technical skills. Similarly, when we asked the question "why do processes cause too many logic/GUI/standards defects," the answer came out as "standards not comprehensively documented" and "people not aware of standards." Similarly, for technology the causes were "unclear specifications" and "technical problems of tools." The brainstorming sessions for the causal analysis, of course, threw up many more causes. But after listing all the suggestions made during the meeting, they were prioritized. Prioritization can be done easily by considering each of the defects and identifying the causes for that defect. The causes that show up most frequently are the ones that are high priority, and are shown in Figure 10.

10.5.3 Develop and Implement Solutions

Root Cause	Preventive Actions
Standards not followed	Do a group reading of the standards. Ensure that standards are followed in mock projects.
Oversight	Effective self review Rigorous code reviews
Unclear/Incorrect Specifications	Conduct specification reviews
Lack of training	Every new entrant will do a mock project. A detailed specification and test plan will be made for the same.
Lack of techinical Skills	Develop tutorials for the key technologies. Have members do mock projects.

Table 10.4: Root causes and proposed solutions.

So far we have discussed how to identify the types of defects that are occurring frequently, and what are the root causes for the major defect categories. But no action has yet been taken to reduce the occurrence of defects. This is done in this phase.

Once the root causes are known, then the next natural step is to think of what can be done to attack the root causes, such that their manifestation in the form of defects is lessened. Some common prevention actions are are building/improving checklists, training programs, reviews, use of some specific tool. The solutions are developed through a brainstorming session. The cause-effect analysis also is done through brainstorming. Hence, frequently, these two steps might be done in the same session. There is one brainstorming session in which the cause-effect analysis is done and the preventive solutions are identified. The root causes and the preventive actions for the example are also shown in Table 10. The preventive actions proposed are self-explanatory.

The preventive solutions are action items which someone has to perform. Hence, the implementation of the solutions is the key. Unless the solutions are implemented, they are of no use at all. One way to ensure this is to treat these as project activities, assign them to project members, and include them in the detailed project schedule.

An important part of implementing the solutions is to see if it is having the desired effect, that is, reducing the injection of defects and thereby reducing the rework effort expended in removing the defects. Analysis of defects some time after the solutions have been implemented can given some insight into this question. Generally, the next analysis for defect prevention can be used for this purpose. Besides tracking the impact, such follow-up analysis has tremendous reinforcing value—seeing the benefits convinces people like nothing else. Hence, besides implementation, the impact of implementation should also be analyzed.

10.6 Metrics—Reliability Estimation

After the testing is done and the software is delivered, the development is considered over. It will clearly be very desirable to know, in quantifiable terms, the reliability of the software being delivered. As testing directly impacts the reliability and most reliability models use data obtained during testing to predict reliability, reliability estimation is the main product metrics of interest at the end of the testing phase. We will focus our attention on this metric in this section.

Before we discuss the reliability modeling and estimation, let us briefly discuss a few main metrics that can be used for process evaluation at the end of the project.

Once the project is finished, one can look at the overall productivity achieved by the programmers during the project. As discussed earlier, productivity can be measured as lines of code (or function points) per person-month.

Another process metric of interest is *defect removal efficiency.* The *defect removal efficiency* of a defect removing process is defined as the percentage reduction of the defects that are present before the start of the process [104]. The *cumulative defect removal efficiency* of a series of defect removal processes is the percentage of defects that have been removed by this series. The defect removal efficiency cannot be determined exactly as the defects remaining in the system are not known. However, at the end of testing, as most defects have been uncovered, removal efficiencies can be estimated.

Let us now return to our main topic—software reliability modeling and assessment. Reliability of software often depends considerably on the quality of testing. Hence, by assessing reliability we can also judge the quality of testing. Alternatively, reliability estimation can be used to decide whether enough testing has been done. Hence, besides characterizing an important quality property of the product being delivered, reliability estimation has a direct role in project management—the reliability models being used by the project manager to decide when to stop testing.

Many models have been proposed for software reliability assessment, and a survey of many of the models is given in [71, 120, 61]. A discussion of the assumptions and consequent limitations on the models is given in [71]. Here we will discuss Musa's basic model, as it is one of the simplest models. The discussion of the model is based largely

on the book [120]. It should, however, be pointed out that reliability models are not in widespread use and are used mostly in special situations.

10.6.1 Basic Concepts and Definitions

Reliability of a product specifies the probability of failure-free operation of that product for a given time duration. As we discussed earlier in this chapter, unreliability of any product comes due to failures or presence of faults in the system. As software does not "wear out" or "age" as a mechanical or an electronic system does, the unreliability of software is primarily due to bugs or design faults in the software. It is widely believed that with the current level of technology it is impossible to detect and remove all the faults in a large software system (particularly before delivery). Consequently, a software system is expected to have some faults in it.

Reliability is a probabilistic measure that assumes that the occurrence of failure of software is a random phenomenon. That is, if we define the life of a software system as a variable, this is a random variable that may assume different values in different invocations of the software. This randomness of the failure occurrences is necessary for reliability modeling. Here, by *randomness* all that is meant is that the failure cannot be predicted accurately. This assumption will generally hold for larger systems, but may not hold for small programs that have bugs (in which case one might be able to predict the failures). Hence, reliability modeling is more meaningful for larger systems (In [120] it is suggested that it should be applied to systems larger than 5000 LOC, as such systems will provide enough data points to do statistical analysis.)

Let X be the random variable that represents the life of a system. Reliability of a system is the probability that the system has not failed by time t. In other words,

$$R(t) = P(X > t).$$

The reliability of a system can also be specified as the *mean time to failure (MTTF)*. MTTF represents the expected lifetime of the system. From the reliability function, it can be obtained as [140]:

$$MTTF = \int_0^\infty R(x)dx.$$

Note that one can obtain the MTTF from the reliability function but the reverse is not always true. The reliability function can, however, be obtained from the MTTF if the failure process is assumed to be *Poisson*, that is, the life time has an *exponential distribution* [140]. With exponential distribution, if the failure rate of the system is known as λ, the MTTF is equal to $1/\lambda$.

Reliability can also be defined in terms of the number of failures experienced by the system by time t. Clearly, this number will also be random as failures are random. With this random variable, we define the *failure intensity* $\lambda(t)$ of the system as the number of

expected failures per unit time at time t. With failure intensity, the number of failures that will occur between t and $t + \Delta t$ can be approximated as $\lambda(t)\Delta t$.

Let us define what is meant by time in these reliability models. There are three common definitions of time for software reliability models [120]: execution time, calendar time, and clock time. *Execution time* is the actual CPU time the software takes during its execution. *Calendar time* is the regular time we use, and *clock time* is the actual clock time that elapses while the software is executing (i.e., it includes the time the software waits in the system). Different models have used different time definitions, though the most commonly used are execution time and calendar time. It is now believed that execution time models are better and more accurate than calendar time models, as they more accurately capture the "stress" on the software due to execution.

Though faults are the cause of failures, the failure of software also depends critically on the environment in which it is executing [120]. It is well known that software frequently fails only if some types of inputs are given. In other words, if software has faults, only some types of input will exercise that fault to affect failures. Hence, how often these inputs cause failures during execution will decide how often the software fails. The *operational profile* of software captures the relative probability of different types of inputs being given to the software during its execution. As the definition of reliability is based on failures, which in turn depends on the nature of inputs, reliability is clearly dependent on the operational profile of the software. Hence, when we say that the reliability of software is $R(t)$, it assumes that this is for some operational profile. If the operational profile changes dramatically, then we will need to either recompute $R(t)$ or recalibrate it.

10.6.2 A Reliability Model

Let us now discuss one particular reliability model—Musa's basic execution time model. The description given here of the model is based on [120]. This is an execution time model, that is, the time taken during modeling is the actual CPU execution time of the software being modeled. The model is simple to understand and apply.

The model focuses on failure intensity while modeling reliability. It assumes that the failure intensity decreases with time, that is, as (execution) time increases, the failure intensity decreases. This assumption is generally true as the following is assumed about the software testing activity, during which data is being collected: during testing, if a failure is observed, the fault that caused that failure is detected and the fault is removed. Consequently, the failure intensity decreases. Most other models make similar assumption which is consistent with actual observations.

In this model, it is assumed that each failure causes the same amount of decrement in the failure intensity. That is, the failure intensity decreases with a constant rate with the number of failures. That is, the failure intensity (number of failures per unit time)

as a function of the number of failures is given as

$$\lambda(\mu) = \lambda_0(1 - \frac{\mu}{\nu_0}),$$

where λ_0 is the initial failure intensity at the start of execution (i.e., at time $t = 0$), μ is the expected number of failures by the given time t, and ν_0 is the total number of failures that would occur in infinite time. The total number of failures in infinite time is finite as it is assumed that on each failure, the fault in the software is removed. As the total number of faults in a given software whose reliability is being modeled is finite, this implies that the number of failures is finite. The failure intensity, as a function of the total number of failures experienced, is shown in Figure 10 [120].

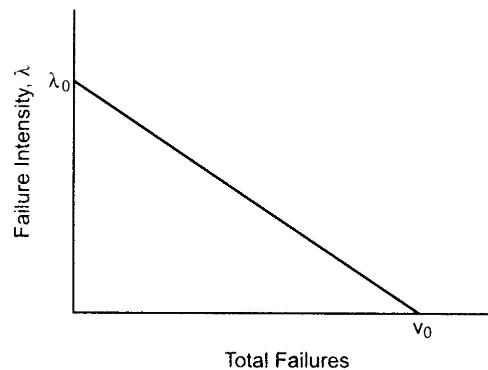

Figure 10.14: Failure intensity function.

The linear decrease in failure intensity as the number of failures observed increases is an assumption that is likely to hold for software for which the operational profile is uniform. That is, for software where the operational profile is such that any valid input is more or less equally likely, the assumption that the failure intensity decreases linearly generally holds. The intuitive rationale is that if the operational profile is uniform, any failure can occur at any time and all failures will have the same impact in failure intensity reduction. If the operational profile is not uniform, the failure intensity curves are ones whose slope decreases with the number of failures (i.e., each additional failure contributes less to the reduction in failure intensity). In such a situation the logarithmic model is better suited.

Note that the failure intensity decreases due to the nature of the software development process, in particular system testing, the activity in which reliability modeling is applied. Specifically, when a failure is detected during testing, the fault that caused the failure is identified and removed. It is removal of the fault that reduces the failure intensity. However, if the faults are not removed, as would be the situation if the software

was already deployed in the field (when the failures are logged or reported but the faults are not removed), then the failure intensity would stay constant. In this situation, the value of λ would stay the same as at the last failure that resulted in fault removal, and the reliability will be given by $R(t) = e^{-\lambda\tau}$, where τ is the execution time.

The expected number of failures as a function of execution time τ (i.e., expected number of failures by time τ), $\mu(\tau)$, in the model is assumed to have an exponential distribution. That is,

$$\mu(\tau) = \nu_0(1 - e^{-\lambda_0/\nu_0 * \tau}).$$

By substituting this value in the equation for λ given earlier, we get the failure intensity as a function of time:

$$\lambda(\tau) = \lambda_0 * e^{-\lambda_0/\nu_0 * \tau}.$$

A typical shape of the failure intensity as it varies with time is shown in Figure 10 [120].

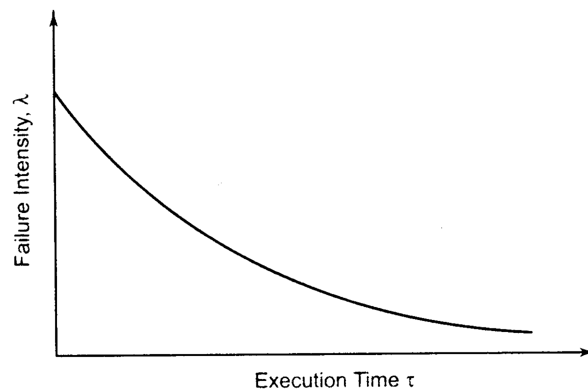

Figure 10.15: Failure intensity with time.

This reliability model has two parameters whose values are needed to predict the reliability of given software. These are the initial failure intensity λ_0 and the total number of failures ν_0. Unless the value of these are known, the model cannot be applied to predict the reliability of software. Most software reliability models are like this; they frequently will have a few parameters whose values are needed to apply the model.

It would be very convenient if these parameters had constant values for all software systems or if they varied in a manner that their values for a particular software can be determined easily based on some clearly identified and easily obtained characteristic of the software (e.g., size or complexity). Some speculations have been made regarding how these parameters may depend on software characteristics. However, no such simple method is currently available that is dependable. The method that is currently used for all software reliability models is to *estimate* the value of these parameters for the

particular software being modeled through the failure data for that software itself. In other words, the failure data of the software being modeled is used to obtain the value of these parameters. Some statistical methods are used for this, which we will discuss shortly.

The consequence of this fact is that, in general, for reliability modeling, the behavior of the software system is carefully observed during system testing and data of failures observed during testing is collected up to some time τ. Then statistical methods are applied to this collected data to obtain the value of these parameters. Once the values of the parameters are known, the reliability (in terms of failure intensity) of the software can be predicted. As statistical methods require that "enough" data points be available before accurate estimation of the parameters can be done, this implies that reliability can be estimated only after sufficient data has been collected. The requirement that there be a reasonably large failure data set before the parameters can be estimated is another reason reliability models cannot effectively be applied to software that is small in size (as it will not provide enough failure data points). Another consequence of this approach is that we can never determine the values of the parameters precisely. They will only be estimates, and there will always be some uncertainty with the values we compute. This uncertainty results in corresponding uncertainty in the reliability estimates computed using the models.

Let us assume that the failure data collection begins with system testing (as is usually the case). That is, time $\tau = 0$ is taken to be the commencement of system testing. The selection of the start of time is somewhat arbitrary. However, selecting the start of time where the assumptions about randomness and operational profile may not hold will cause the model to give incorrect estimates. This is why data of unit testing or integration testing, where the whole system is not being tested, is not considered. System testing, in which the entire system is being tested, is really the earliest point from where the data can be collected.

This model can be applied to compute some other values of interest that can help decide if enough testing has been done or how much more testing needs to be done to achieve a target reliability. Suppose the target reliability is specified in terms of desired failure intensity, λ_F. Let the present failure intensity be λ_P. Then the number of failures that we can expect to observe before the software achieves the desired reliability can be computed by computing $\lambda_F - \lambda_P$, which gives,

$$\Delta\mu = \frac{\nu_0}{\lambda_0}(\lambda_P - \lambda_F).$$

In other words, at any time we can now clearly say how many more failures we need to observe (and correct) before the software will achieve the target reliability. Similarly, we can compute the additional time that needs to be spent before the target reliability is achieved. This is given by

$$\Delta t = \frac{\nu_0}{\lambda_0} ln \frac{\lambda_P}{\lambda_F}.$$

That is, we can expect that the software needs to be executed for Δt more time before we will observe enough failures (and remove the faults corresponding to them) to reach the target reliability. This time can be converted to calendar time, which is what is used in projects, by incorporating some parameters about the software development environment. This issue will be discussed later.

10.6.3 Failure Data and Parameter Estimation

To apply the reliability model for a particular software, we need to obtain the value of the two parameters: λ_0 and ν_0. These parameters are not the same for all software and have to be estimated for the software being modeled using statistical techniques.

For statistical approaches to parameter estimation, data has to be collected about the failures of the software being modeled. Generally, the earliest point to start collecting data for reliability estimation is the start of system testing (a later point can also be taken, though it will reduce the number of failures that can be observed). The data can be collected in two different forms. The first form is to record the failure times (in execution time) of the failures observed during execution. This data will essentially be a sequence of (execution) times representing the first, second, and so on failures that are observed. The second form of data is to record the number of failures observed during execution in different time intervals (called *grouped failure data*). This form might sometimes be easier to collect if the unit is a clearly identified unit, like a day. In this form, the data will be in the form of a table, where the duration of the interval (in execution time) and the number of failures observed during that interval are given. We will only discuss the parameter estimation with the first form. For further details on parameter estimation, the reader is referred to [120].

There are many ways in which the model can be "fitted" to the data points to obtain the parameters or coefficients. One common method is the least squares approach in which the goal is to select the parameters for the model so that the square of the difference between the observed value and the one predicted by the model is minimized. This approach works well when the size of the data set is not very large.

For applying the least squares approach, we will consider the equation for the failure intensity as a function of the mean number of failures (i.e., $\lambda(\mu) = \lambda_0(1 - \mu/\nu_0)$). To determine parameters for this equation, we need a set of observed data points, each containing the value of the dependent variable and the value of the independent variable. In this case, this means that we need data points, each of which gives the failure intensity and the number of failures.

The data collected, as specified earlier, may be in the form of failure times or grouped failure data. The first thing that needs to be done is to convert the data to the desired form by determining the failure intensity for each failure. If the data about failure times is available, this conversion is done as follows [120]. Let the observation interval be $(0, t_e]$ (t_e is the time when the observations are stopped; it will generally be greater than the time of the last failure). We partition this observation interval at every kth failure

occurrence. That is, this time interval is partitioned into sub-intervals, each (except the last one) containing k distinct failures. If the total number of failures observed until t_e is m_e, then the number of subintervals is p, where $p = \lceil m_e/k \rceil$. The observed failure intensity for an interval can now be computed by dividing the number of failures in that interval by the duration of the interval. That is, for an interval l, the observed failure intensity r_l is given by

$$r_l = \frac{k}{t_{kl} - t_{k(l-1)}}, l = 1, \ldots, p-1.$$

For the last interval, the failure intensity is

$$r_p = \frac{m_e - k(p-1)}{t_e - t_{k(l-1)}}.$$

These failure intensities are *independent* of each other as the different time intervals are disjoint. The estimate for the mean value for the lth interval, m_l, can be obtained by

$$m_l = k(l-1).$$

(This takes the start value for the interval but has been found to be better than taking the average or midpoint value [120].) In this method, if k is chosen to be too small, large variations will occur in failure intensity. If the value of k is very large, too much smoothing may occur. A value of about five (i.e., $k = 5$) gives reasonable results [120].

Obtaining data in this form from grouped data is even easier. For each time interval for which failures were counted, dividing the number of failures by the duration of the interval will give the failure intensity of that interval. The total number of failures for an interval is the sum of all the failures of all the intervals before this interval.

In this manner, we can get from the collected data a set of p data points, each giving a failure intensity and the total number of failures observed. As the relationship between them is linear, a regression line can be fit in these data points. From the coefficients of the line, model parameters can be determined easily.

However, the approach of simple linear regression minimizes the sum of absolute errors (between the predicted value by the model and the actual value observed). This approach gives a higher weight to the data points with larger failure intensity. In other words, the coefficients will be influenced more by data points with larger failure intensity. A better approach is to consider *relative error*, which is absolute error divided by the value given by the model. The least squares approach here will be to minimize the sum of all the relative errors. With relative errors, each data point is given the same weight. However, with this, linear regression cannot be used, and closed-form equations for determining the coefficients are not available. For this approach, numerical methods must be used to determine the coefficients. The approach will be to obtain the derivatives of the equation for least squares (with relative error) with the two coefficients to be determined, set these to 0, and then solve these two simultaneous equations

through some standard numerical technique like the Newton-Raphson method. For further discussion on this, the reader is referred to [120] or any numerical analysis text.

Once the parameters are known, we can also predict the number of faults in the delivered software using the reliability model (which can be used to predict faults per KLOC). As we don't otherwise know how many faults remain in software, generally, this data is available for a project only after the software has been in operation for a few years and most of its faults have been identified. By using the reliability model, we can predict this with some confidence.

The total failures experienced in infinity time by a software is related to the total faults in the system, as we are assuming that faults are generally removed after a failure is detected. However, the fault removal process may not be perfect and may introduce errors. In addition, each failure may not actually result in removing of a fault, as the information obtained on failure may not be sufficient for fault detection. If the total number of faults in the software is ω_0, we can get ν_0 from this by using the *fault reduction factor, B*:

$$\nu_0 = \frac{\omega_0}{B}$$

The fault reduction factor, B, is the ratio of the *net* fault reduction to the total number of failures experienced. If each failure resulted in exactly one fault being removed, then B would be 1. However, sometimes a failure is not sufficient to locate a fault or a fault removal adds some faults. Due to these, the fault reduction factor is not always 1. Currently available data suggests that B is close to 1, with an average value of about 0.95 [120]. This value can be used to predict the number of faults that remain in the software. Alternatively, the value of B can be computed from the data collected (additional data about fault correction will have to be compiled).

10.6.4 Translating to Calendar Time

The model discussed here is an execution-time model: all the times are the CPU execution time of the software. However, software development and project planning works in calendar time—hours, days, months, etc. Hence, we would like to convert the estimates to calendar time, particularly when we are trying to predict the amount of time still needed to achieve the desired reliability. In this case, it is clearly desirable to specify the time in calendar time, so that the project plan can be modified appropriately, if needed.

As reliability modeling is performed from system testing onward, the execution time can be related to the effort for testing, debugging, etc. The simplest way to do this is to determine an average ratio of the amount of effort to execution time and then to use this effort to estimate the calendar time. Alternatively, instead of giving one ratio, two ratios can be specified—one for the CPU time expended and one for the failures detected. These ratios can then be used to determine the total amount of effort.

Let us explain this approach with a simple example. Generally, the main resource during testing is the test team effort. For now, we consider this as the only resource of interest for modeling calendar time. Suppose the test team runs the software for 10 CPU hours, during which it detects 25 failures. Suppose that for each hour of CPU execution time, an average of 8 person-hours of the test team are consumed (ratio of effort to CPU time), and that on an average 4 person-hours is needed on each failure to analyze it (ratio of effort to failures). Hence, the total effort required for this is

$$10 * 8 + 25 * 4 = 180 \text{ person-hours}.$$

If the quantity of test team resources (i.e., the number of members in the test team) is three persons, this means that the calendar time for this is 60 hours. As the number of failures experienced is a function of time according to the Basic model, one overall ratio could also have been given with CPU time (or with number of failures). In this example, the overall ratio will be 18 person-hours per CPU hour.

10.6.5 An Example

Let us illustrate the use of the reliability model discussed earlier through the use of an example. In [120], times for more than 130 failures for a real system called T1 are given. For illustration purposes, we select about 50 data points from it, starting from after about 2000 CPU sec have elapsed (from the 21st failure). We define $\tau = 0$ after the first 2000 sec of [120] to illustrate that the choice of $\tau = 0$ is up to the reliability estimator and to eliminate the first few data points, which are likely to show a wider variation, as they probably represent the start of testing. The times of failures with this $\tau = 0$ are given in Table 10 [120].

As we can see, this is the failure times data. From this, using $k = 5$, we obtain the failure intensities and the cumulative failures as discussed earlier. The data points we get are:

(0.0045, 0), (0.0026, 5), (0.0182, 10), (0.0047, 15), (0.0040, 20),
(0.0020, 25), (0.0056, 30), (0.0032, 35), (0.0023, 40), (0.0035, 45)

For the purposes of this example, we will try to fit a regression line to this data using the regular least squares approach, for which parameter determination can be done in a simple manner. As discussed earlier, this method is likely to give poorer results compared to minimizing the square of relative errors. Using the regular regression line fitting approach, we get $\lambda_0 = 0.0074$ failure/CPU sec and $\nu_0 \approx 70$ failures. (If the complete data from [120] is used, then ν_0 comes out to about 136 failures. Because we are not counting the first 20, this means that by fitting a line on the complete data using the relative error approach, ν_0 would come out to be around 110. This error in our estimate is coming due to the smaller sample and the use of absolute error for determining the coefficients.) By the reliability model, the current reliability of the software (after 50 failures have been observed) is about 0.002 failure per CPU second.

Time of Failure (in CPU sec)				
311	3089	5922	10,559	14,358
366	3565	6738	10,559	15,168
608	3623	8089	10,791	
676	4080	8237	11,121	
1098	4380	8258	11,486	
1278	4477	8491	12,708	
1288	4740	8625	13,251	
2434	5192	8982	13,261	
3034	5447	9175	13,277	
3049	5644	9411	13,806	
3085	5837	9442	14,185	
3089	5843	9811	14,229	

Table 10.5: Failure data for a real system.

We can see that the total number of estimated faults in the system at the start of the time is 70. Out of this, 50 faults have been removed (after observing the 50 failures). Hence, there are still 20 faults left in the software. Suppose the size of the final software was 20,000 LOC. If the failure data given earlier is until the end of system testing (i.e., the software is to be delivered after this) and this software development project is a typical project for the process that was followed, we can say that the capability of this process is to deliver software with a fault density of 1.0 per KLOC.

Now let's suppose the current failure intensity after 50 failures is not acceptable to the client. The desired failure intensity is 0.001 failure per CPU second. Using the model, we can say that to achieve this reliability, further testing needs to be done and the amount of CPU time that will be consumed in this extra testing can be estimated to be

$$70/0.0074 * ln(0.002/0.001) = 6,527\ CPU - sec.$$

That is, approximately 1.81 CPU hours of testing needs to be performed to achieve the target reliability. Suppose the limiting resource is only the testing personnel, there is one person assigned to test this software, and on an average 20 person-hours of testing personnel effort is spent for each hour of CPU time. In this case, we can say that more than 36 person-hours of testing need to be done. In other words, the calendar time needed to achieve the target reliability is about a week.

10.7 Summary

Testing plays a critical role in quality assurance for software. Due to the limitations of the verification methods for the previous phases, design and requirement faults also appear in the code. Testing is used to detect these errors, in addition to the errors introduced during the coding phase.

Testing is a dynamic method for verification and validation, where the system to be tested is executed and the behavior of the system is observed. Due to this, testing observes the failures of the system, from which the presence of faults can be deduced. However, separate activities have to be performed to identify the faults (and then remove them).

There are two approaches to testing: black-box and white-box. In black-box testing, the internal logic of the system under testing is not considered and the test cases are decided from the specifications or the requirements. It is often called functional testing. Equivalence class partitioning, boundary value analysis, and cause-effect graphing are examples of methods for selecting test cases for black-box testing. State-based testing is another approach in which the system is modeled as a state machine and then this model is used to select test cases using some transition or path based coverage criteria. State-based testing can also be viewed as grey-box testing in that it often requires more information than just the requirements.

In white-box testing, the test cases are decided entirely on the internal logic of the program or module being tested. The external specifications are not considered. Often a criterion is specified, but the procedure for selecting test cases is left to the tester. The most common control flow-based criteria are statement coverage and branch coverage, and the common data flow-based criteria are all-defs and all-uses. Mutation testing is another approach for white-box testing that creates mutants of the original program by changing the original program. The testing criterion is to kill all the mutants by having the mutant generate a different output from the original program.

As the goal of testing is to detect any errors in the programs, different levels of testing are often used. Unit testing is used to test a module or a small collection of modules and the focus is on detecting coding errors in modules. During integration testing, modules are combined into subsystems, which are then tested. The goal here is to test the system design. In system testing and acceptance testing, the entire system is tested. The goal here is to test the system against the requirements, and to test the requirements themselves. White-box testing can be used for unit testing, while at higher levels mostly black-box testing is used.

The testing process usually commences with a test plan, which is the basic document guiding the entire testing of the software. It specifies the levels of testing and the units that need to be tested. For each of the different units, first the test cases are specified

and then they are reviewed. During the test case execution phase, the test cases are executed, and various reports are produced for evaluating testing. The main outputs of the execution phase are the test summary report and the error report.

The main metric of interest during testing is the reliability of the software under testing. Reliability of software depends on the faults in the software. To assess the reliability of software, reliability models are needed. To use a model for a given software system, data is needed about the software that can be used in the model to estimate the reliability of the software. Most reliability models are based on the data obtained during the system and acceptance testing. Data about time between failures observed during testing are used by these models to estimate the reliability of the software. We discussed one such reliability model in the chapter in some detail and have discussed how the reliability model can be used in a project and what the limitations of reliability models are.

Exercises

1. What are the different levels of testing and the goals of the different levels? For each level, specify which of the testing approaches is most suitable.
2. Testing, including debugging and fixing of bugs, is the most expensive task in a project. List the major activities in the entire testing process, and give your view on what % of the testing effort each consumes.
3. Suppose a software has three inputs, each having a defined valid range. How many test cases will you need to test all the boundary values?
4. For boundary value analysis, if the strategy for generating test cases is to consider all possible combinations for the different values, what will be the set of test cases for a software that has three inputs X, Y, and Z?
5. Take three variables A, B, and C, each having two values. Generate a set of test cases that will exercise all pairs.
6. Suppose a software has five different configuration variables that are set independently. If three of them are binary (have two possible values), and the rest have three values, how many test cases will be needed if pair-wise testing method is used?
7. Consider a vending machine that takes quarters and when it has received two quarters, gives a can of soda. Develop a state model of this system, and then generate sets of test cases for the various criteria.
8. Suppose you have to test a class for implementing a queue of integers. Using state-based approach (and one criteria for it), generate a set of test cases that you will use to test it. Assume standard operations like add, delete on the queue.

9. Consider a simple text formatter problem. Given a text consisting of words separated by blanks (BL) or newline (NL) characters, the text formatter has to covert it into lines, so that no line has more than MAXPOS characters, breaks between lines occurs at BL or NL, and the maximum possible number of words are in each line. The following program has been written for this text formatter [73]:

```
alarm := false;
bufpos := 0;
fill := 0;
repeat
  inchar(c);
  if (c = BL) or (c = NL) or (c = EOF)
  then
    if bufpos != 0
    then begin
      if (fill + bufpos < MAXPOS) and (fill != 0)
      then begin
        outchar(BL);
        fill := fill + 1; end
      else begin
        outchar(NL);
        fill := 0; end;
      for k:=1 to bufpos do
        outchar(buffer[k]);
      fill := fill + bufpos;
      bufpos := 0; end
    else
      if bufpos = MAXPOS
      then alarm := true
      else begin
        bufpos := bufpos + 1;
        buffer[bufpos] := c; end
until alarm or (c = EOF);
```

For this program, do the following:

(a) Select a set of test cases using the black-box testing approach. Use as many techniques as possible and select test cases for special cases using the "error guessing" method.

(b) Select a set of test cases that will provide 100% branch coverage.

(c) Select a set of test cases that will satisfy the all-defs and the all-uses criteria (except the ones that are not feasible).

(d) Create a few mutants by simple transformations. Then select a set of test cases that will kill these mutants.

(e) Suppose that this program is written as a procedure. Write a driver for testing this procedure with the test cases selected in (a) and (b). Clearly specify the format of the test cases and how they are used by the driver.

10. Suppose three numbers A, B, and C are given in ascending order representing the lengths of the sides of a triangle. The problem is to determine the type of the triangle (whether it is isosceles, equilateral, right, obtuse, or acute). Consider the following program written for this problem:

```
read(a, b, c);
if (a < b) or (b < c) then
    print("Illegal inputs");
    return;
if (a=b) or (b=c) then
    if (a=b) and (b=c) then print("equilateral triangle")
    else print("isosceles triangle")
else begin
    a := a*a; b := b*b; c := c*c;
    d := b+c;
    if (a = d) then print("right triangle")
    else if (a<d) then print("acute triangle")
    else print("obtuse triangle");
end;
```

For this program, perform the same exercises as in the previous problem.

11. **What are the limitations of the reliability model discussed in the chapter for using it for estimating the reliability of a product?**

12. Suppose you want to predict the reliability of a product at the time of the release using the model discussed in the chapter. What data will you collect and when for this, and what changes (if any) will you make in your testing?

13. **Define some data flow criteria for testing an entire class (i.e., not just for testing the methods independently) (refer to [82]).**

14. In your next project, collect the defects in the last stages of testing. Perform the cause-effect analysis for these defects leading to some actions on how you should do things differently in the future for reducing the errors you make.

15. **Another method for evaluating software reliability is to use the Mill's seeding approach. In this method some faults are seeded in the program, and reliability is assessed based on how many of these seeded faults are detected during testing. Develop a simple reliability model based on this approach. Define your parameters, and give a formula for estimating the reliability and the number of faults remaining in the system. Also discuss the drawbacks and limitations of this model?**

Case Studies

Here we briefly discuss the test plans and strategy for the two case studies. The detailed test case specifications for system testing are available from the Web site.

Test Plan for Case Study 1 (Course Scheduling)

This document describes the plan for testing the course scheduling software. All major testing activities are specified here; additional testing may be scheduled later, if necessary.

1. Test Units

In this project we will perform two levels of testing: unit testing and system testing. Because the system is small, it is felt that there is no need for elaborate integration testing. The basic units to be tested are:

Modules to input file-1
Modules to input file-2
Modules for scheduling

In addition, some other units may be chosen for testing. The testing for these different units will be done independently.

2. Features to be Tested

All the functional features specified in the requirements document will be tested. No testing will be done for the performance, as the response time requirement is quite weak.

3. Approach for Testing

For unit testing, structural testing based on the branch coverage criterion will be used. The goal is to achieve branch coverage of more than 95%. The CCOV coverage analyzer tool will be used to determine the coverage. System testing will be largely functional in nature. The focus is on invalid and valid cases, boundary values, and special cases.

4. Test Deliverables

The following documents are required (besides this test plan):

- Unit test report for each unit
- Test case specification for system testing
- Test report for system testing
- Error report

The test case specification for system testing has to be submitted for review before system testing commences.

5. Test Case Specifications for System Testing
For test case specifications we specify all test cases that are used for system testing. First, the different conditions that need to be tested, along with the test cases used for testing those conditions and the expected outputs are given. Then the data files used for testing are given. The test cases are specified with respect to these data files. The test cases have been selected using the functional approach. The goal is to test the different functional requirements, as specified in the requirements document. Test cases have been selected for both valid and invalid inputs. The entire test case specifications is available from from the Web site.

Case Study 2—PIMS

The test plan for PIMS is similar to the previous one. It also follows a two level testing—unit and then system. Unit testing is performed by the programmers, and no unit test reports are mandated. As the overall plan is the same, we do not discuss it here. We just discuss some aspects of planning for system test cases.

System testing would begin with the development team releasing applications to the test team. The sequence of activities is:

- Development team does a unit testing of the application, fixes identified problems and hands over the environment to the Test team.
- Test team runs some quick checks (e.g., that the system installs, that it can take inputs) and some tests for critical functionality. If 80% of these tests pass, then the application is considered ready for system testing, otherwise it is returned to the developers.
- The test team runs the test cases.
- Testing will be suspended if during testing the test team encounters any critical defects, or a set of major defects which would prevent effective testing.
- The testing shall resume only when 100% of critical defects are fixed and at least 80% major defects are fixed.
- Testing shall end when all the test cases in the test plan have been executed.
- Defects identified will be notified to the development team regularly and all defect fixes received from the development team will be included for retesting.

For this case study, the system test plan was prepared with inputs from some software quality professionals from commercial organizations. So, in a sense, the test cases represent the type of testing that may be done by professionals. The test case specifications are available from the Web site.