T.Y.B.Sc. Computer Science

Semester V - Paper 2 – Software Testing

Unit 2 Notes:

Strategies of Software Testing :

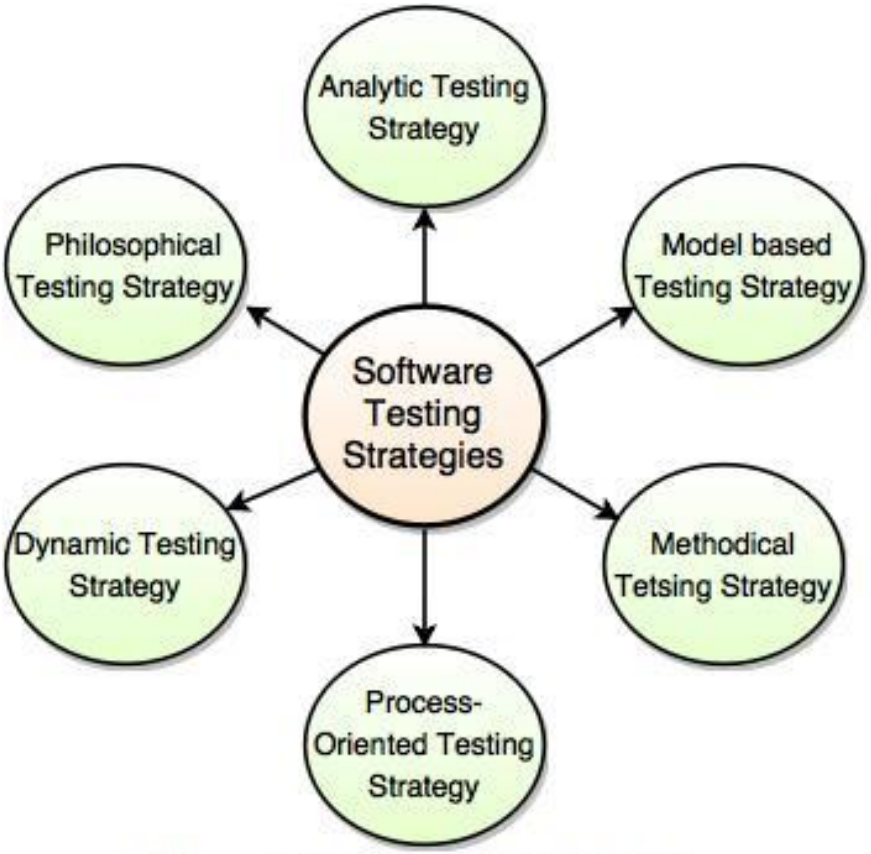


Fig. - Strategies of Software Testing

) Analytic testing strategy

- Uses formal and informal techniques to access and prioritize risks that might occur during software testing.
- Takes a full overview of requirements, design and implementation of objects to determine the aim of testing.
- Collects complete information regarding software, target that is achieved and the data needed for testing the software.

ii) Model-based testing strategy

- Tests the functionality of the software.

- Identifies the domain of data and selects suitable test cases as per the probability of errors in that domain.

iii) Methodical testing strategy

- Tests the function and status of software according to checklist, based on the user requirements.
- This strategy is used to test the functionality, reliability, usability and performance of the software.

iv) Process-oriented testing strategy

- Tests the software from the existing standards i.e IEEE standards.
- Ensures the functionality of the software by using automated testing tools.

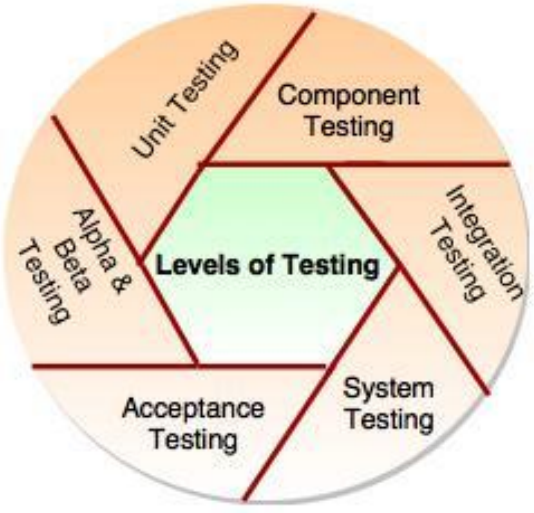
v) Dynamic testing strategy

- Tests the software after having common decision of the testing team.
- Gives information regarding the software, for e.g., the test cases used for testing the errors present in it.

vi) Philosophical testing strategy

- Tests the software by assuming that any component of the software terminates the functioning anytime.
- For testing the software it takes help from software developers, users and system analysis.

Levels of Software Testing



1) Unit testing:

A Unit is a smallest testable portion of system or application which can be compiled, liked, loaded, and executed. This kind of testing helps to test each module separately.

The aim is to test each part of the software by separating it. It checks that component are fulfilling functionalities or not. This kind of testing is performed by developers.

2) Integration testing:

Integration means combining. In this testing phase, different software modules are combined and tested as a group to make sure that integrated system is ready for system testing.

Integrating testing checks the data flow from one module to other modules. This kind of testing is performed by testers.

3) System testing:

System testing is performed on a complete, integrated system. It allows checking system's compliance as per the requirements. It tests the overall interaction of components. It involves load, performance, reliability and security testing.

System testing most often the final test to verify that the system meets the specification. It evaluates both functional and non-functional need for the testing.

4) Acceptance testing:

Acceptance testing is a test conducted to find if the requirements of a specification or contract are met as per its delivery. Acceptance testing is basically done by the user or customer. However, other stockholders can be involved in this process.

Other Types of Testing:

REGRESSION TESTING is a type of software testing that intends to ensure that changes (enhancements or defect fixes) to the software have not adversely affected it.

- The likelihood of any code change impacting functionalities that are not directly associated with the code is always there and it is essential that regression testing is conducted to make sure that fixing one thing has not broken another thing.
- During regression testing, new test cases are not created but previously created test cases are re-executed.
- Some tend to include Regression Testing as a separate level of software testing but that is a misconception. Regression Testing is, in fact, just a type of testing that can be performed at any of the four main levels.

Alpha Testing - Alpha testing is one of the most common software testing strategy used in software development. Its specially used by product development organizations.

- This test takes place at the developer's site. Developers observe the users and note problems.
- Alpha testing is testing of an application when development is about to complete. Minor design changes can still be made as a result of alpha testing.
- Alpha testing is typically performed by a group that is independent of the design team, but still within the company, e.g. in-house software test engineers, or software QA engineers.
- Alpha testing is final testing before the software is released to the general public. It has two phases:
    - In the first phase of alpha testing, the software is tested by in-house developers. They use either debugger software, or hardware-assisted debuggers. The goal is to catch bugs quickly.
    - In the second phase of alpha testing, the software is handed over to the software QA staff, for additional testing in an environment that is similar to the intended use.

- Alpha testing is simulated or actual operational testing by potential users/customers or an independent test team at the developers' site. Alpha testing is often employed for off-the-shelf software as a form of internal acceptance testing, before the software goes to beta testing.

Beta Testing - Beta Testing is also known as field testing. It takes place at customer's site. It sends the system/software to users who install it and use it under real-world working conditions.

- A beta test is the second phase of software testing in which a sampling of the intended audience tries the product out. (Beta is the second letter of the Greek alphabet.) Originally, the term alpha testing meant the first phase of testing in a software development process. The first phase includes unit testing, component testing, and system testing. Beta testing can be considered "pre-release testing.
- The goal of beta testing is to place your application in the hands of real users outside of your own engineering team to discover any flaws or issues from the user's perspective that you would not want to have in your final, released version of the application. Example: Microsoft and many other organizations release beta versions of their products to be tested by users.

Software Metrics

- A measure of some property of a piece of software or its specifications
- They are all measurable, that is they can be quantified.

Some common software metrics are:-

1. LOC
2. Cyclomatic complexity, is used to measure code complexity.
3. Function point analysis (FPA), is used to measure the size (functions) of software.
4. Bugs per lines of code.
5. Code coverage - Code lines that are executed for a given set of software tests.
6. Cohesion - how well the source code in a given module work together to provide a single function.
7. Coupling - how well two software components are *data* related, i.e. how independent they are.

The first application of project metrics on most software projects occurs during estimation. Metrics collected from past projects are used as a basis from which effort and time estimates are made for current software work. As a project proceeds, measures of effort and calendar time expended are compared to original estimates (and the project schedule). The project manager uses these data to monitor and control progress.

As technical work commences, other project metrics begin to have significance. Production rates represented in terms of pages of documentation, review hours, function points, and delivered source lines are measured. In addition, errors uncovered during each software engineering task are tracked. As the software evolves from specification into design, technical metrics (Chapters 19 and 24) are collected to assess design quality and to provide indicators that will influence the approach taken to code generation and testing.

The intent of project metrics is twofold. First, these metrics are used to minimize the development schedule by making the adjustments necessary to avoid delays and mitigate potential problems and risks. Second, project metrics are used to assess product quality on an ongoing basis and, when necessary, modify the technical approach to improve quality.

As quality improves, defects are minimized, and as the defect count goes down, the amount of rework required during the project is also reduced. This leads to a reduction in overall project cost. Another model of software project metrics [HET93] suggests that every project should measure:

• *Inputs*—measures of the resources (e.g., people, environment) required to do the work.

• *Outputs*—measures of the deliverables or work products created during the software engineering process.

• *Results*—measures that indicate the effectiveness of the deliverables.

Size-Oriented Metrics

Size-oriented software metrics are derived by normalizing quality and/or productivity measures by considering the *size* of the software that has been produced. If a software organization maintains simple records, a table of size-oriented measures, such as the one shown in Figure can be created. The table lists each software development project that has been completed over the past few years and corresponding measures for that project. Referring to the table entry for project alpha:

12,100 lines of code were developed with 24 person-months of effort at a cost of $168,000. It should be noted that the effort and cost recorded in the table represent all software engineering activities (analysis, design, code, and test), not just coding. Further information for project alpha indicates that 365 pages

of documentation were developed, 134 errors were recorded before the software was released, and 29 defects

Project	LOC	Effort	$(000)	Pp. doc.	Errors	Defects	People
alpha	12,100	24	168	365	134	29	3
beta	27,200	62	440	1224	321	86	5
gamma	20,200	43	314	1050	256	64	6
•	•	•	•	•	•		
•	•	•	•	•	•		
•	•	•	•	•	•		

were encountered after release to the customer within the first year of operation. Three people worked on the development of software for project alpha. In order to develop metrics that can be assimilated with similar metrics from other projects, we choose lines of code as our normalization value. From the rudimentary data contained in the table, a set of simple size-oriented metrics can be developed for each project:

- Errors per KLOC (thousand lines of code).
- Defects4 per KLOC.
- $ per LOC.
- Page of documentation per KLOC.

In addition, other interesting metrics can be computed:

- Errors per person-month.
- LOC per person-month.
- $ per page of documentation.

Size-oriented metrics are not universally accepted as the best way to measure the process of software development [JON86]. Most of the controversy swirls around the use of lines of code as a key measure. Proponents of the LOC measure claim that LOC is an "artifact" of all software development projects that can be easily counted, that many existing software estimation models use LOC

or KLOC as a key input, and that a large body of literature and data predicated on LOC already exists. On the other hand, opponents argue that LOC measures are programming language dependent, that they penalize well-designed but shorter programs, that they cannot easily accommodate nonprocedural languages, and that their use in estimation requires a level of detail that may be difficult to achieve

Function-Oriented Metrics

Function-oriented software metrics use a measure of the functionality delivered by the application as a normalization value. Since 'functionality' cannot be measured directly, it must be derived indirectly using other direct measures. Function points are derived using an empirical relationship based on countable (direct) measures of software's information domain and assessments of software complexity. Function points are computed [IFP94] by completing the table shown in Figure. Five information domain characteristics are determined and counts are provided in the appropriate table location. Information domain values are defined in the following manner:

			Weighting factor			
Measurement parameter	**Count**		**Simple**	**Average**	**Complex**	
Number of user inputs		×	3	4	6	=
Number of user outputs		×	4	5	7	=
Number of user inquiries		×	3	4	6	=
Number of files		×	7	10	15	=
Number of external interfaces		×	5	7	10	=
Count total	→					

Number of user inputs. Each user input that provides distinct application oriented data to the software is counted. Inputs should be distinguished from inquiries, which are counted separately.

Number of user outputs. Each user output that provides application oriented information to the user is counted. In this context output refers to reports, screens, error messages, etc. Individual data items within a report are not counted separately.

Number of user inquiries. An inquiry is defined as an on-line input that results in the generation of some immediate software response in the form of an on-line output. Each distinct inquiry is counted.

Number of files. Each logical master file (i.e., a logical grouping of data that may be one part of a large database or a separate file) is counted.

Number of external interfaces. All machine readable interfaces (e.g., data files on storage media) that are used to transmit information to another system are counted.

Once these data have been collected, a complexity value is associated with each count. Organizations that use function point methods develop criteria for determining whether a particular entry is simple, average, or complex. Nonetheless, the determination of complexity is somewhat subjective.

To compute function points (FP), the following relationship is used:

FP = count total * (0.65 + 0.01 * sum of (*Fi*))

where count total is the sum of all FP entries

The *Fi* (*i* = 1 to 14) are "complexity adjustment values" based on responses to the following questions :

1. Does the system require reliable backup and recovery?
2. Are data communications required?
3. Are there distributed processing functions?
4. Is performance critical?
5. Will the system run in an existing, heavily utilized operational environment?
6. Does the system require on-line data entry?
7. Does the on-line data entry require the input transaction to be built over multiple
screens or operations?
8. Are the master files updated on-line?
9. Are the inputs, outputs, files, or inquiries complex?
10. Is the internal processing complex?
11. Is the code designed to be reusable?
12. Are conversion and installation included in the design?
13. Is the system designed for multiple installations in different organizations?
14. Is the application designed to facilitate change and ease of use by the user?

Each of these questions is answered using a scale that ranges from 0 (not important or applicable) to 5 (absolutely essential). Once function points have been calculated, they are used in a manner analogous to LOC as a way to normalize measures for software productivity, quality, and other attributes:

- Errors per FP.
- Defects per FP.
- $ per FP.
- Pages of documentation per FP.
- FP per person-month.

RECONCILING DIFFERENT METRICS APPROACHES

The relationship between lines of code and function points depends upon the programming language that is used to implement the software and the quality of the design. A number of studies have attempted to relate FP and LOC measures. To quote Albrecht and Gaffney [ALB83]:

The thesis of this work is that the amount of function to be provided by the application (program) can be estimated from the itemization of the major components8 of data to be used or provided by it. Furthermore, this estimate of function should be correlated to both the amount of LOC to be developed and the development effort needed. The following table [JON98] provides rough estimates of the average number of lines of code required to build one function point in various programming languages:

Programming Language LOC/FP (average)

Assembly language 320
C 128
COBOL 106
FORTRAN 106
Pascal 90
C++ 64
Ada95 53
Visual Basic 32
Smalltalk 22
Powerbuilder (code generator) 16
SQL 12

A review of these data indicates that one LOC of C++ provides approximately 1.6 times the "functionality" (on average) as one LOC of FORTRAN. Furthermore, one LOC of a Visual Basic provides more than three times the functionality of a LOC for a conventional programming language. More detailed data on the relationship between FP A good engineer insists on maintaining a complete understanding and control over every aspect of the project. The more difficult the project the more firmly the insistence on simplicity – without it no one can understand what is going on. Software designers and programmers have sometimes been accused of exhibiting the exact opposite characteristic; they deliberately avoid simple solutions and gain satisfaction from the complexities of their designs.

However, many software designers and programmers today strive to make their software as clear and simple as possible. A programmer finishes a program and is satisfied both that it works correctly and that it is clearly written. But how do we know that it is clear? Is a shorter program necessarily simpler than a longer

one (that achieves the same end), or is a heavily nested program simpler than an equivalent program without nesting?

We cannot establish a measure of complexity – for example, the number of statements in a program – without investigating how such a measure corresponds with programmers' perceptions and experiences. We now describe one attempt to establish a meaningful measure of complexity. One aim of such work is to guide programmers in selecting clear program structures and rejecting unclear structures, either during design or afterwards. The approach taken is to hypothesize about what factors affect program complexity. For example, we might conjecture that program length, the number of alternative paths through the program and the number of references to data might all affect complexity. We could perhaps invent a formula that allows us to calculate the overall complexity of a program from these constituent factors. The next step is to verify the hypothesis.

Amongst several attempts to measure complexity is McCabe's cyclomatic complexity. McCabe suggests that complexity does not depend on the number of statements. Instead it depends only on the decision structure of the program – the number of if, while and similar statements. To calculate the cyclomatic complexity of a program, count the number of conditions and add one. For example, the program fragment:

```
x = y;
if (a == b)
        c = d;
else
        e = f;
p = q
```

has a complexity of 2, because there are two independent paths through the program. Similarly a while and a repeat each count one towards the complexity count. Compound conditions like: if a > b and c > d then
count two because this if statement could be rewritten as two, nested if statements. Note that a program that consists only of a sequence of statements, has a cyclomatic complexity of 1, however long it is. Thus the smallest value of this metric is 1.

How this metric is useful for software testing?

Basis Path testing is one of White box technique and it guarantees to execute atleast one statement during testing. It checks each linearly independent path

through the program, which **means number test cases, will be equivalent to the cyclomatic complexity of the program.**

This metric is useful because of properties of Cyclomatic complexity (M) -

M can be number of test cases to achieve branch coverage (Upper Bound)

Consider this example -

```
If (Condition 1)

Statement 1

Else

Statement 2

If (Condition 2)

Statement 3

Else

Statement 4
```

Cyclomatic Complexity for this program will be 9-7+2=4.

As complexity has calculated as 4, four test cases are necessary to the complete path coverage for the above example.

Cyclomatic complexity is a useful attempt to quantify complexity, and it is claimed that it has been successfully applied. It is, however, open to several criticisms as follows.

1. The figure 10 for the maximum allowed complexity is somewhat arbitrary and unscientific.
2. The measure makes no allowance for the sheer length of a module, so that a one-page module (with no decisions) is rated as equally complex as a thousand-page module (with no decisions).
3. The measure depends only on control flow, ignoring, for example, references to data. One program might only act upon a few items of data, while another might involve operations on a variety of

complex objects. (Indirect references to data, say via pointers, are an extreme case.)

What is Defect?

A Defect, in simple terms, is a flaw or an error in an application that is restricting the normal flow of an application by mismatching the expected behavior of an application with the actual one.

The defect occurs when any mistake is made by a developer during the designing or building of an application and when this flaw is found by a tester, it is termed as a defect.

It is the responsibility of a tester to do a thorough testing of an application with an intention to find as many defects as possible so as to ensure that a quality product will reach the customer.

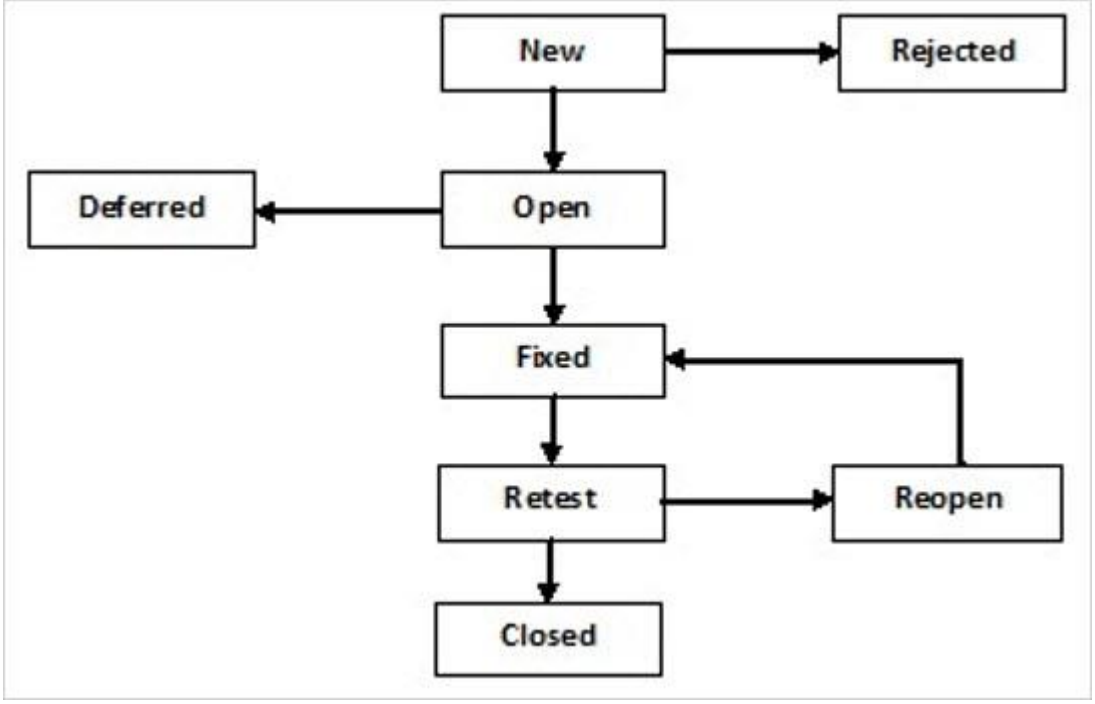
It is important to understand about defect life cycle before moving to the workflow and different states of the defect.

Defect Life Cycle in Detail

A Defect life cycle, also known as a Bug life cycle, is a cycle of a defect from which it goes through covering the different states in its entire life. This starts as soon as any new defect is found by a tester and comes to an end when a tester closes that defect assuring that it won't get reproduced again.

Defect Workflow:

It is now time to understand the actual workflow of a defect life cycle with the help of a simple diagram as shown below.



Defect States:

#1) New: This is the first state of a defect in the defect life cycle. When any new defect is found, it falls in a 'New' state and validations and testing are performed on this defect in the later stages of the defect life cycle.

#2) Assigned: In this stage, a newly created defect is assigned to the development team for working on the defect. This is assigned by the project lead or the manager of the testing team to a developer.

#3) Open: Here, the developer starts the process of analyzing the defect and works on fixing it, if required. If the developer feels that the defect is not appropriate then it may get transferred to any of the below four states namely **Duplicate, Deferred, Rejected or Not a Bug**-based upon the specific reason.

I will discuss these four states in a while.

#4) Fixed: When the developer finishes the task of fixing a defect by making the required changes then he can mark the status of the defect as 'Fixed'.

#5) Pending Retest: After fixing the defect, the developer assigns the defect to the tester for retesting the defect at their end and till the tester works on retesting the defect, the state of the defect remains in 'Pending Retest'.

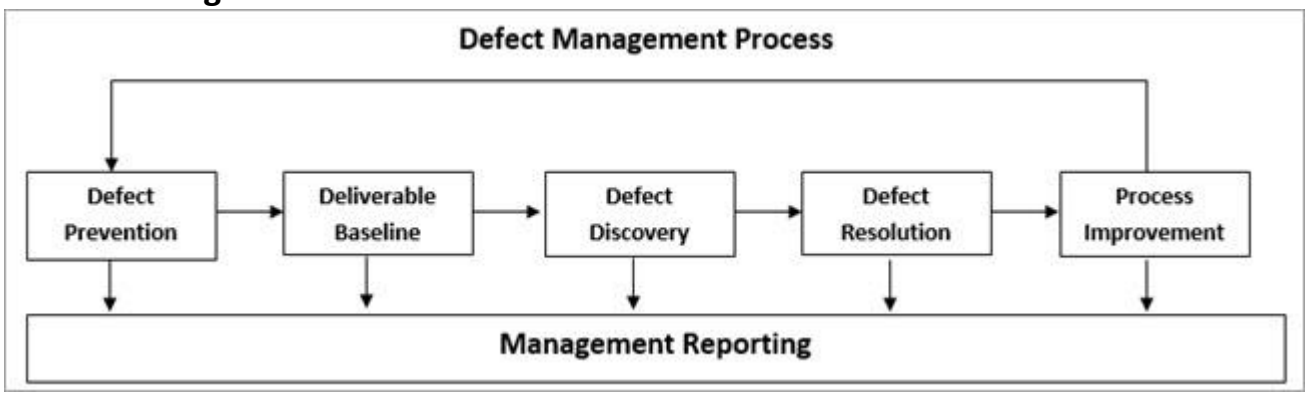
#6) Retest: At this point, the tester starts the task of working on the retesting of the defect to verify if the defect is fixed accurately by the developer as per the requirements or not.

#7) Reopen: If any issue still persists in the defect then it will be assigned to the developer again for testing and the status of the defect gets changed to 'Reopen'.

#8) Verified: If the tester does not find any issue in the defect after being assigned to the developer for retesting and he feels that if the defect has been fixed accurately then the status of the defect gets assigned to 'Verified'.
#9) Closed: When the defect does not exist any longer then the tester changes the status of the defect to 'Closed'.
Few More:

- **Rejected:** If the defect is not considered as a genuine defect by the developer then it is marked as 'Rejected' by the developer.
- **Duplicate:** If the developer finds the defect as same as any other defect or if the concept of the defect matches with any other defect then the status of the defect is changed to 'Duplicate' by the developer.
- **Deferred:** If the developer feels that the defect is not of very important priority and it can get fixed in the next releases or so in such a case, he can change the status of the defect as 'Deferred'.
- **Not a Bug:** If the defect does not have an impact on the functionality of the application then the status of the defect gets changed to 'Not a Bug'.

The **mandatory fields** when a tester logs any new bug are Build version, Submit On, Product, Module, Severity, Synopsis and Description to Reproduce
Defect Management Process



Defect management process is explained below in detail.
#1) Defect Prevention:
Defect Prevention is the best method to eliminate the defects in the early stage of testing instead of finding the defects in the later stage and then fixing it. This method is also cost effective as the cost required for fixing the defects found in the early stages of testing is very low.
However, it is not possible to remove all the defects but at least you can minimize the impact of the defect and cost to fix the same.

The major steps involved in Defect Prevention are as follow:

- **Identify Critical Risk**: Identify the critical risks in the system which will impact more if occurred during testing or in the later stage.

- **Estimate Expected Impact**: For each critical risk, calculate how much would be the financial impact if the risk actually encountered.
- **Minimize expected impact**: Once you identify all critical risks, take the topmost risks which may be harmful to the system if encountered and try to minimize or eliminate the risk. For risks which cannot be eliminated, it reduces the probability of occurrence and its financial impact.

#2) Deliverable Baseline:

When a deliverable (system, product or document) reaches its pre-defined milestone then you can say a deliverable is a baseline. In this process, the product or the deliverable moves from one stage to another and as the deliverable moves from one stage to another, the existing defects in the system also gets carried forward to the next milestone or stage.

For Example, consider a scenario of coding, unit testing and then system testing. If a developer performs coding and unit testing then system testing is carried out by the testing team. Here coding and Unit Testing is one milestone and System Testing is another milestone.

So during unit testing, if the developer finds some issues then it is not called as a defect as these issues are identified before the meeting of the milestone deadline. Once the coding and unit testing have been completed, the developer hand-overs the code for system testing and then you can say that the code is **"baselined"** and ready for next milestone, here, in this case, it is "system testing".

Now, if the issues are identified during testing then it is called as the defect as it is identified after the completion of the earlier milestone i.e. coding and unit testing.

Basically, the deliverables are baselined when the changes in the deliverables are finalized and all possible defects are identified and fixed. Then the same deliverable passes on to the next group who will work on it.

#3) Defect Discovery:

It is almost impossible to remove all the defects from the system and make a system as a defect-free one. But you can identify the defects early before they become costlier to the project. We can say that the defect discovered means it is formally brought to the attention of the development team and after analysis of that the defect development team also accepted it as a defect.

Steps involved in Defect Discovery are as follows:

- **Find a Defect**: Identify defects before they become a major problem to the system.
- **Report Defect**: As soon as the testing team finds a defect, their responsibility is to make the development team aware that there is an issue identified which needs to be analyzed and fixed.
- **Acknowledge Defect**: Once the testing team assigns the defect to the development team, its the development team's responsibility to acknowledge the defect and continue further to fix it if it is a valid defect.

#4) Defect Resolution:

In the above process, the testing team has identified the defect and reported to the development team. Now here the development team needs to proceed for the resolution of the defect.

The steps involved in the defect resolution are as follows:

- **Prioritize the risk**: Development team analyzes the defect and prioritizes the fixing of the defect. If a defect has more impact on the system then they make the fixing of the defect on a high priority.
- **Fix the defect**: Based on the priority, the development team fixes the defect, higher priority defects are resolved first and lower priority defects are fixed at the end.
- **Report the Resolution**: Its the development team's responsibility to ensure that the testing team is aware when the defects are going for a fix and how the defect has been fixed i.e. by changing one of the configuration files or making some code changes. This will be helpful for the testing team to understand the cause of the defect.

#5) Process Improvement:

Though in the defect resolution process the defects are prioritized and fixed, from a process perspective, it does not mean that lower priority defects are not important and are not impacting much to the system. From process improvement point of view, all defects identified are same as a critical defect.

Even these minor defects give an opportunity to learn how to improve the process and prevent the occurrences of any defect which may impact system failure in the future. Identification of a defect having a lower impact on the system may not be a big deal but the occurrences of such defect in the system itself is a big deal.

For process improvement, everyone in the project needs to look back and check from where the defect was originated. Based on that you can make changes in

the validation process, base-lining document, review process which may catch the defects early in the process which are less expensive.

Metrics related to defects :

- Failure index (failure cost/IT budget)
- Defect removal efficiency
- Mean time to failure of critical systems
- Defect arrival rate for critical development projects

The following areas should be viewed as prerequisites to process improvement:

- ***Documented Process***: The process used to develop and maintain software should be documented.

- ***Deliverable Definitions***: The process must define the required content of deliverables, the point in the development process where each deliverable is produced, and the point in the process where each deliverable is baselined. This step is required to define and identify defects.

- ***Defect Reporting System***: Procedures should be established to formally and concisely report defects and gather the data required to compute the critical metrics. The bulk of this work should be the responsibility of individual project teams. A quality assurance function or other group should be responsible for consolidating and reporting information across projects.

Once the above areas are in place, defect information should be collected and analyzed. This information should be used to guide the ongoing process improvement activities.