Unit-I

1. INTRODUCTION TO TESTING – WHY AND WHAT:

1.1 Why is testing necessary?
1.2 What is testing?
1.3 Role of Tester
1.4 Overview of STLC

2. SOFTWARE TESTING LIFE CYCLE - V MODEL:

2.1 V-Model
2.2 SDLC vs STLC
2.3 Different stages in STLC
2.4 Document templates generated in different phases of STLC
2.5 Different levels of testing
2.6 Different types of testing: Functional Testing
2.7 API Testing
2.8 Usability Testing
2.9 Exploratory Testing
2.10 Ad-hoc Testing
2.11 Static Testing**:**
Static techniques,
Reviews,
Walkthroughs.

3. BASICS OF TEST DESIGN TECHNIQUES:

3.1 Various test categories
3.2 Test design techniques for different categories of tests
3.3 Designing test cases using MS-Excel.

1. INTRODUCTION TO TESTING – WHY AND WHAT:

1.1 WHY IS TESTING NECESSARY?

1.1.1 Introduction

This section describes and illustrates how software defects or bugs can cause problems for people, the environment or a company. We'll explain why testing is necessary to find these defects, how testing promotes quality, and how testing fits into quality assurance. In this section, we will also introduce some fundamental principles of testing.

As we go through this section, watch for the Syllabus terms **bug, defect, error, failure, fault, mistake, quality, risk, software, testing** and **exhaustive testing.**

Why testing is needed? Testing is necessary because we all make mistakes. Some of those mistakes are unimportant, but some of them are expensive or dangerous. We need to check everything and anything we produce because things can always go wrong - humans make mistakes all the time - it is what we do best!

Because we should assume our work contains mistakes, we all need to check our own work. However, some mistakes come from bad assumptions and blind spots, so we might make the same mistakes when we check our own work as we made when we did it. So we may not notice the flaws in what we have done. Ideally, we should get someone else to check our work - another person is more likely to spot the flaws.

We know that in ordinary life, some of our mistakes do not matter, and some are very important. It is the same with software systems. We need to know whether a particular error is likely to cause problems. To help us think about this, we need to consider the context within which we use different software systems.

1.1.2 Software systems context

Testing Principle - *Testing is context dependent*

Testing is done differently in different contexts. For example, safety-critical software is tested differently from an e-commerce site.

These days, almost everyone is aware of **software** systems. We encounter them in our homes, at work, while shopping, and because of mass-communication systems. More and more, they are part of our lives. We use software in day-to-day business applications such as banking and in consumer products such as cars and washing machines. However, most people have had an experience with software that did not work as expected: an error on a bill, a delay when waiting for a credit card to process and a website that did not load correctly are common examples of problems that may happen because of software problems. Not all software systems carry the same level of **risk** and not all problems have the same impact when they occur. A risk is something that has not happened yet and it may never happen; it is a potential problem. We are concerned about these potential problems because, if one of them did happen, we'd feel a negative impact. When we

discuss risks, we need to consider how likely it is that the problem would occur and the impact if it happens. For example, whenever we cross the road, there is some risk that we'll be injured by a car. The likelihood depends on factors such as how much traffic is on the road, whether there is a safe crossing place, how well we can see, and how fast we can cross. The impact depends on how fast the car is going, whether we are wearing protective gear, our age and our health. The risk for a particular person can be worked out and therefore the best road-crossing strategy.

Some of the problems we encounter when using software are quite trivial, but others can be costly and damaging - with loss of money, time or business reputation and even may result in injury or death. For example, suppose a user interface has typographical defects. Does this matter? It may be trivial, but it could have a significant effect, depending on the website and the defect:

- If my personal family-tree website has my maternal grandmother's maiden name spelt wrong, my mother gets annoyed and I have to put up with some family teasing, but I can fix it easily and only the family see it (probably).
- If the company website has some spelling mistakes in the text, potential customers may be put off the company as it looks unprofessional.
- If a software program miscalculates pesticide application quantities, the effect could be very significant: suppose a decimal point is wrongly placed so that the application rate is 10 times too large. The farmer or gardener uses more pesticide than needed, which raises his costs, has environmental impacts on wildlife and water supplies and has health and safety impact for the farmer, gardener, family and workforce, livestock and pets. There may also be consequent loss of trust in and business for the company and possible legal costs and fines for causing the environmental and health problems.

1.1.3 Causes of software defects

Why it is that software system sometimes doesn’t work correctly? We know that people make mistakes - we are fallible.

If someone makes an **error** or mistake in using the software, this may lead directly to a problem - the software is used incorrectly and so does not behave as we expected. However, people also design and build the software and they can make mistakes during the design and build. These mistakes mean that there are flaws in the software itself. These are called **defects** or sometimes bugs or faults. Remember, the software is not just the code; check the definition of soft-ware again to remind yourself.

When the software code has been built, it is executed and then any defects may cause the system to fail to do what it should do (or do something it shouldn't), causing a **failure.** Not all defects result in failures; some stay dormant in the code and we may never notice them.

Do our mistakes matter?

Let's think about the consequences of mistakes. We agree that any human being, programmers and testers included, can make an error.

These errors may produce defects in the software code or system, or in a document. If a defect in code is executed, the system may experience a failure. So the mistakes we make matter partly because they have consequences for the products for which we are responsible.

Our mistakes are also important because software systems and projects are complicated. Many interim and final products are built during a project, and people will almost certainly make mistakes and errors in all the activities of the build. Some of these are found and removed by the authors of the work, but it is difficult for people to find their own mistakes while building a product. Defects in software, systems or documents may result in failures, but not all defects do cause failures. We could argue that if a mistake does not lead to a defect or a defect does not lead to a failure, then it is not of any importance -we may not even know we've made an error.

Our fallibility is compounded when we lack experience, don't have the right information, misunderstand, or if we are careless, tired or under time pressure. All these factors affect our ability to make sensible decisions - our brains either don't have the information or cannot process it quickly enough.

Additionally, we are more likely to make errors when dealing with perplexing technical or business problems, complex business processes, code or infra-structure, changing technologies, or many system interactions. This is because our brains can only deal with a reasonable amount of complexity or change when asked to deal with more our brains may not process the information we have correctly.

It is not just defects that give rise to failure. Failures can also be caused by environmental conditions as well: for example, a radiation burst, a strong magnetic field, electronic fields, or pollution could cause faults in hardware or firmware. Those faults might prevent or change the execution of software. Failures may also arise because of human error in interacting with the software, perhaps a wrong input value being entered or an output being misinterpreted. Finally, failures may also be caused by someone deliberately trying to cause a failure in a system malicious damage.

When we think about what might go wrong we have to consider defects and failures arising from:

- errors in the specification, design and implementation of the software and system;
- errors in use of the system;
- environmental conditions;
- intentional damage;
- potential consequences of earlier errors, intentional damage, defects and failures.

When do defects arise?

In Figure 1.1 we can see how defects may arise in four requirements for a product.

We can see that requirement 1 is implemented correctly - we understood the customer's requirement, designed correctly to meet that requirement, built correctly to meet the design, and so deliver that requirement with the right attributes: functionally, it does what it is supposed to do and it also has the right non-functional attributes, so it is fast enough, easy to understand and so on.

With the other requirements, errors have been made at different stages. Requirement 2 is fine until the software is coded, when we make some mistakes and introduce defects. Probably, these are easily spotted and corrected during testing, because we can see the product does not meet its design specification.
The defects introduced in requirement 3 are harder to deal with; we built exactly what we were told to but unfortunately the designer made some mistakes so there are defects in the design. Unless we check against the requirements definition, we will not spot those defects during testing. When we do notice them they will be hard to fix because design changes will be required.

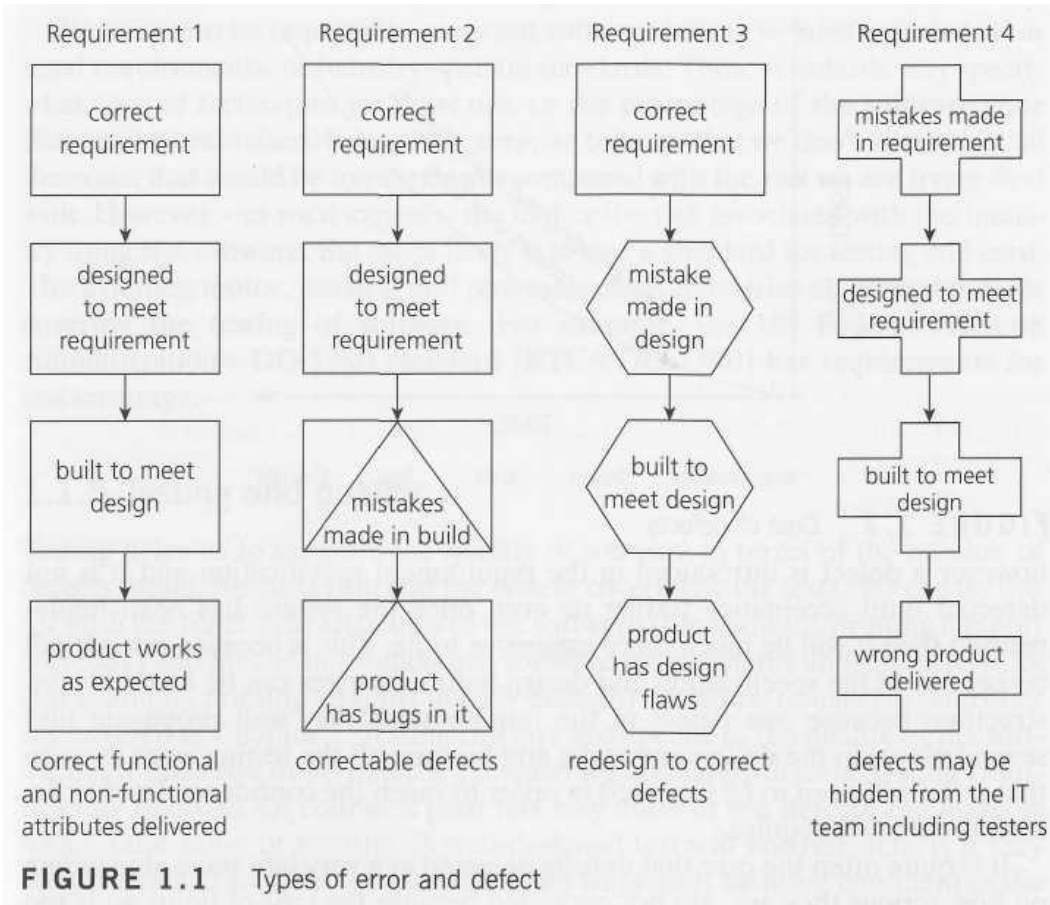


FIGURE 1.1 Types of error and defect

The defects in requirement 4 were introduced during the definition of the requirements; the product has been designed and built to meet that flawed requirements definition. If we test the product meets its requirements and design, it will pass its tests but may be rejected by the user or customer. Defects reported by the customer in acceptance test or live use can be very costly. Unfortunately, requirements and design defects are not rare; assessments of thousands of projects have shown that defects introduced during requirements and design make up close to half of the total number of defects.

What is the cost of defects?

As well as considering the impact of failures arising from defects we have not found, we need to consider the impact of when we find those defects. The cost of finding and fixing defects rises considerably across the life cycle; think of the old English proverb 'a stitch in time saves nine'. This means that if you mend a tear in your sleeve now while it is small, it's easy to mend, but if you leave it, it will get worse and need more stitches to mend it.

If we relate the scenarios mentioned previously to Figure 1.2, we see that, if an error is made and the consequent defect is detected in the requirements at the specification stage, then it is relatively cheap to find and fix. The specification can be corrected and reissued. Similarly if an error is made and the consequent defect detected in the design at the design stage then the design can be corrected and reissued with relatively little expense. The same applies for construction. If however a defect is introduced in the requirement specification and it is not detected until acceptance testing or even once the system has been implemented then it will be much more expensive to fix. This is because rework will be needed in the specification and design before changes can be made in construction; because one defect in the requirements may well propagate into several places in the design and code; and because all the testing work done-to that point will need to be repeated in order to reach the confidence level in the software that we require.

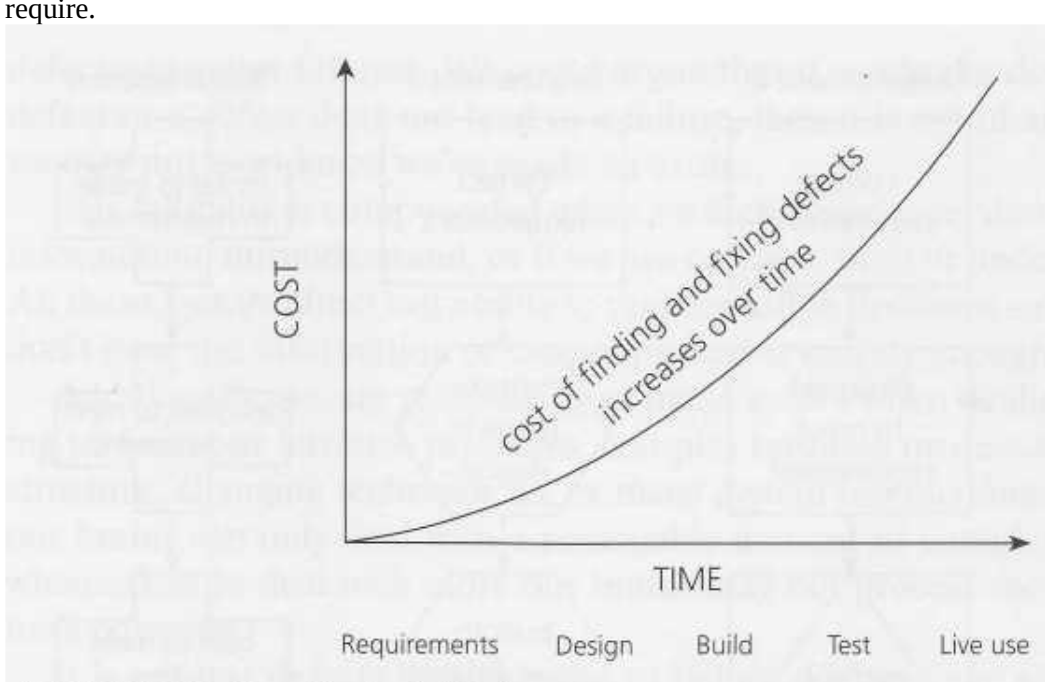


FIGURE 1.2 Cost of defects

It is quite often the case that defects detected at a very late stage, depending on how serious they are, are not corrected because the cost of doing so is too expensive. Also, if the software is delivered and meets an agreed specification, it sometimes still won't be

accepted if the specification was wrong. The project team may have delivered exactly what they were asked to deliver, but it is not what the users wanted. This can lead to users being unhappy with the system that is finally delivered. In some cases, where the defect is too serious, the system may have to be de-installed completely.

1.1.4 Role of testing in software development, maintenance and operations

We have seen that human errors can cause a defect or fault to be introduced at any stage within the software development life cycle and, depending upon the consequences of the mistake, the results can be trivial or catastrophic. Rigorous testing is necessary during development and maintenance to identify defects, in order to reduce failures in the operational environment and increase the quality of the operational system. This includes looking for places in the user interface where a user might make a mistake in input of data or in the interpretation of the output, and looking for potential weak points for intentional and malicious attack. Executing tests helps us move towards improved quality of product and service, but that is just one of the verification and validation methods applied to products. Processes are also checked, for example by audit. A variety of methods may be used to check work, some of which are done by the author of the work and some by others to get an independent view.

We may also be required to carry out software testing to meet contractual or legal requirements, or industry-specific standards. These standards may specify what type of techniques we must use, or the percentage of the software code that must be exercised. It may be a surprise to learn that we don't always test all the code; that would be too expensive compared with the risk we are trying deal with. However - as we'd expect the higher the risk associated with the industry using the software, the more likely it is that a standard for testing will exist. The avionics, motor, medical and pharmaceutical industries all have standards covering the testing of software. For example, the US Federal Aviation Administration's DO-178B standard [RTCA/DO-178B] has requirements for test coverage.

1.1.5 Testing and quality

Testing helps us to measure the **quality** of software in terms of the number of defects found, the tests run, and the system covered by the tests. We can do this for both the functional attributes of the software (for example, printing a report correctly) and for the non-functional software requirements and characteristics (for example, printing a report quickly enough). Testing can give confidence in the quality of the soft-ware if it finds few or no defects, provided we are happy that the testing is sufficiently rigorous. Of course, a poor test may uncover few defects and leave us with a false sense of security. A well-designed test will uncover defects if they are present and so, if such a test passes, we will rightly be more confident in the software and be able to assert that the overall level of risk of using the system has been reduced. When testing does find defects, the quality of the software system increases when those defects are fixed, provided the fixes are carried out properly.

What is quality?

Projects aim to deliver software to specification. For the project to deliver what the customer needs requires a correct specification. Additionally, the delivered system must meet the specification. This is known as validation ('is this the right specification?') and verification ('is the system correct to specification?'). Of course, as well as wanting the right software system built correctly, the customer wants the project to be within budget and timescale - it should arrive when they need it and not cost too much.

The ISTQB glossary definition covers not just the specified requirements but also user and customer needs and expectations. It is important that the project team, the customers and any other project stakeholders set and agree expectations. We need to understand what the customers understand by quality and what their expectations are. What we as software developers and testers may see as quality that the software meets its defined specification, is technically excellent and has few bugs in it may not provide a quality solution for our customers. Furthermore, if our customers find they have spent more money than they wanted or that the software doesn't help them carry out their tasks, they won't be impressed by the technical excellence of the solution.

If the customer wants a cheap car for a 'run about' and has a small budget then an expensive sports car or a military tank are not quality solutions, however well built they are.

To help you compare different people's expectations, Table 1.1 summarizes and explains quality viewpoints and expectations using 'producing and buying tomatoes' as an analogy for 'producing and buying software'. You'll see as you look through the table that the approach to testing would be quite different depending on which viewpoint we favor [Trienekens], [Evans].

In addition to understanding what quality feels and looks like to customers, users, and other stakeholders, it helps to have some quality attributes to measure quality against, particularly to aid the first, product based, viewpoint in the table. These attributes or characteristics can serve as a framework or checklists for areas to consider coverage. One such set of quality attributes can be found in the ISO 9126 standard.

Table 1.1 summarizes and explains quality

Viewpoint	Software	Tomatoes
Quality is measured by looking at the attributes of the product.	We will measure the attributes of the software, e.g. its reliability in terms of mean time between failures (MBTF), and release when they reach a specified level e.g. MTBF of 12 hours.	The tomatoes are the right size and shape for packing for the supermarket. The tomatoes have a good taste and color,
Quality is fitness for use. Quality can have subjective aspects and not just quantitative aspects.	We will ask the users whether they can carry out their tasks; if they are satisfied that they can we will release the software	The tomatoes are right for our recipe,

Quality is based on good manufacturing processes, and meeting defined requirements. It is measured by testing, inspection, and analysis of faults and failures.	We will use a recognized software development process. We will only release the software if there are fewer than five outstanding high-priority defects once the planned tests are complete.	The tomatoes are organically farmed. The tomatoes have no blemishes and no pest damage,
Expectation of value for money, affordability, and a value based trade-off between time, effort and cost aspects. We can afford to buy this software and we expect a return on investment.	We have time-boxed the testing to two weeks to stay in the project budget.	The tomatoes have a good shelf life. The tomatoes are cheap or good value for money,
Transcendent feelings - this is about the feelings of an individual or group of individuals towards a product or a supplier.	We like this software! It is fun and it's the latest thing! So what if it has a few problems? We want to use it anyway...We really enjoy working with this software team. So, there were a few problems they sorted them out really quickly we trust them.	We get our tomatoes from a small local farm and we get on so well with the growers,

What is root cause analysis?

When we detect failures, we might try to track them back to their root cause, the real reason that they happened. There are several ways of carrying out root cause analysis, often involving a group brainstorming ideas and discussing them, so you may see different techniques in different organizations. If you are interested in using root cause analysis in your work, you'll find simple techniques described in [Evans], [TQMI] and [Robson]. For example, suppose an organization has a problem with printing repeatedly failing. Some IT maintenance folk get together to examine the problem and they start by brainstorming all the possible causes of the failures. Then they group them into categories they have chosen, and see if there are common underlying or root causes. Some of the obvious causes they discover might be:

- Printer runs out of supplies (ink or paper).
- Printer driver software fails.
- Printer room is too hot for the printer and it seizes up.

These are the immediate causes. If we look at one of them - 'Printer runs out of supplies (ink or paper)' - it may happen because:

- No-one is responsible for checking the paper and ink levels in the printer; possible root cause: no process for checking printer ink/paper levels before use.
- Some staff doesn’t know how to change the ink cartridges; possible root cause: staff not trained or given instructions in looking after the printers.
- There is no supply of replacement cartridges or paper; possible root cause: no process for stock control and ordering.

If your testing is confined to software, you might look at these and say, 'These are not software problems, so they don't concern us!' So, as software testers we might confine ourselves to reporting the printer driver failure. However, our remit as testers may be beyond the software; we might have a remit to look at a whole system including hardware and firmware. Additionally, even if our remit is software, we might want to consider how software might help people prevent or resolve problems; we may look beyond this view. The software could provide a user interface which helps the user anticipate when paper or ink is getting low. It could provide simple step-by-step instructions to help the users change the cartridges or replenish paper. It could provide a high temperature warning so that the environment can be managed. As testers, we want not just to think and report on defects but, with the rest of the project team, think about any potential causes of failures.

We use testing to help us find faults and (potential) failures during software development, maintenance and operations. We do this to help reduce the risk of failures occurring in an operational environment - in other words once the system is being used and to contribute to the quality of the software system. However, whilst we need to think about and report on a wide variety of defects and failures, not all get fixed. Programmers and others may correct defects before we release the system for operational use, but it may be more sensible to work around the failure. Fixing a defect has some chance of introducing another defect or of being done incorrectly or incompletely. This is especially true if we are fixing a defect under pressure. For this reason, projects will take a view sometimes that they will defer fixing a fault. This does not mean that the tester who has found the problems has wasted time. It is useful to know that there is a problem and we can help the system users work around and avoid it.

The more rigorous our testing, the more defects we'll find. But you'll see in Chapters 3 and 4, when we look at techniques for testing, that rigorous testing does not necessarily mean more testing; what we want to do is testing that finds defects - a small number of well-placed, targeted tests may be more rigorous than a large number of badly focused tests.

We saw earlier that one strategy for dealing with errors, faults and failures is to try to prevent them, and we looked at identifying the causes of defects and failures. When we start a new project, it is worth learning from the problems encountered in previous projects or in the production software. Understanding the root causes of defect is an important aspect of quality assurance activities, and testing contributes by helping us to identify defects as early as possible before the software is in use. As testers, we are also interested in looking at defects found in other projects, so that we can improve our processes. Process improvements should prevent those defects recurring and, as a consequence, improve the quality of future systems. Organizations should consider testing as part of a larger quality assurance strategy, which includes other activities (e.g., development standards, training and root cause analysis).

1.1.6 How much testing is enough?

Testing Principle - Exhaustive testing is impossible

Testing everything (all combinations of inputs and preconditions) is not feasible except for trivial cases. Instead of exhaustive testing, we use risks and priorities to focus testing efforts.

We've seen that testing helps us find defects and improve software quality. How much testing should we do? We have a choice: test everything, test nothing or test some of the software. Now, your immediate response to that may well be to say, 'Everything must be tested'. We don't want to use software that has not been completely tested, do we? This implies that we must exercise every aspect of a software system during testing. What we need to consider is whether we must, or even can, test completely.

Let's look at how much testing we'd need to do to be able to test exhaustively. How many tests would you need to do to completely test a one-digit numeric field? The immediate question is, 'What you mean by test completely?' There are 10 possible valid numeric values but as well as the valid values we need to ensure that all the invalid values are rejected. There are 26 uppercase alpha characters, 26 lower case, at least 6 special and punctuation characters as well as a blank value. So there would be at least 68 tests for this example of a one-digit field.

This problem just gets worse as we look at more realistic examples. In practice, systems have more than one input field with the fields being of varying sizes. These tests would be alongside others such as running the tests in different environments. If we take an example where one screen has 15 input fields, each having 5 possible values, then to test all of the valid input value combinations you would need 30 517 578 125 (5^{15}) tests! It is unlikely that the project timescales would allow for this number of tests.

Testing our one-digit field with values 2, 3 and 4 makes our tests more thorough, but it does not give us more information than if we had just tested with the value 3.

Pressures on a project include time and budget as well as pressure to deliver a technical solution that meets the customers' needs. Customers and project managers will want to spend an amount on testing that provides a return on investment for them. This return on investment includes prevent-ing failures after release that are costly. Testing completely - even if that is what customers and project managers ask for - is simply not what they can afford.

Instead we need a test approach which provides the right amount of testing for this project, these customers (and other stakeholders) and this software. We do this by aligning the testing we do with the risks for the customers, the stake-holders, the project and the software. Assessing and managing risk is one of the most important activities in any project, and is a key activity and reason for testing. Deciding how much testing is enough should take account of the level of risk, including technical and business risks related to the product and project constraints such as time and budget.

We carry out a risk assessment to decide how much testing to do. We can then vary the testing effort based on the level of risk in different areas. Additionally, testing should provide sufficient information to stakeholders to make informed decisions about the release of the software or system we're testing, for the next development step or handover

to customers. The effort put into the quality assurance and testing activities needs to be tailored to the risks and costs associated with the project. Because of the limits in the budget, the time, and in testing we need to decide how we will focus our testing, based on the risks.

1.2 WHAT IS TESTING?

In this section, we will review the common objectives of testing. We'll explain how testing helps us to find defects, provide confidence and information, and prevent defects. We will also introduce additional fundamental principles of testing.
As you read this section, you'll encounter the terms **code, debugging, development of software, requirement, review, test basis, test case, testing** and **test objective.**

1.2.1 The driving test - an analogy for software testing

We have spent some time describing why we need to test, but we have not discussed what testing is. What do we mean by the word testing? We use the words test and testing in everyday life and earlier we said testing could be described as 'checking the software is OK'. That is not a detailed enough definition to help us understand software testing. Let's use an analogy to help us: driving tests. In a driving test, the examiner critically assesses the candidate's driving, noting every mistake, large or small, made by the driver under test. The examiner takes the driver through a route which tests many possible driving activities, such as road junctions of different types, control and maneuvering of the car, ability to stop safely in an emergency, and awareness of the road, other road users and hazards. Some of the activities *must* be tested. For example, in the UK, an emergency stop test is always carried out, with the examiner simulating the moment of emergency by hitting the dashboard at which point the driver must stop the car quickly, safely and without loss of control. At the end of the test, the examiner makes a judgment about the driver's performance. Has the driver passed the test or failed? The examiner bases the judgment on the number and severity of the failures identified, and also whether the driver has been able to meet the driving requirements. A single severe fault is enough to fail the whole test, but a small number of minor faults might still mean the test is passed. Many minor faults would reduce the confidence of the examiner in the quality —of the driving to the point where the driver cannot pass. The format of the driving test and the conduct of the examiner are worth considering:

- The test is planned and prepared for. In advance of the test, the examiner has planned a series of routes which cover the key driving activities to allow a thorough assessment of the driver's performance. The drivers under test do not know what route they will be asked to take in advance, although they know the requirements of the test.
- The test has known goals - assessing whether the driver is sufficiently safe to be allowed to drive by themselves without an instructor, without endangering themselves or others. There are clear pass/fail criteria, based on the number and severity of faults, but the confidence of the examiner in the driving is also taken into account.

- The test is therefore carried out to show that the driver satisfies the requirements for driving and to demonstrate that they are fit to drive. The examiner looks for faults in the driving. The time for the test is limited, so it is not a complete test of the driver's abilities, but it is representative and allows the examiner to make a risk based decision about the driver. All the drivers are tested in an equivalent way and the examiner is neutral and objective. The examiner will log factual observations which enable a risk assessment to be made about the driving. Based on this, a driver who passes will be given a form enabling him to apply for a full driving license. A driver who fails will get a report with a list of faults and areas to improve before retaking the test.

As well as observing the driver actually driving, the examiner will ask questions or the driver will take a written exam to check their understanding of the rules of the road, road signs, and what to do in various traffic situations.

1.2.2 Defining software testing

With that analogy in mind, let's look at the ISTQB definition of software **testing.**

Let's break the definition down into parts; the definition has some key phrases to remember. The definition starts with a description of testing as a process and then lists some objectives of the test process. First, let's look at testing as a process:

- *Process* - Testing is a process rather than a single activity - there are a series of activities involved.
- *All life cycle activities* - We saw earlier that the later in the life cycle we find bugs, the more expensive they are to fix. If we can find and fix requirements defects at the requirements stage, that must make commercial sense. We'll build the right software, correctly and at a lower cost overall. So, the thought process of designing tests early in the life cycle can help to prevent defects from being introduced into **code.** We sometimes refer to this as 'verifying the **test basis** via the test design'. The test basis includes documents such as the **requirements** and design specifications.
- *Both static and dynamic* - We'll see in Chapter 3 that as well as tests where the software code is executed to demonstrate the results of running tests (often called dynamic testing) we can also test and find defects without executing code. This is called static testing. This testing includes **reviewing** of documents (including source code) and static analysis. This is a useful and cost effective way of testing.
- *Planning* - Activities take place before and after test execution. We need to manage the testing; for example, we plan what we want to do; we control the test activities; we report on testing progress and the status of the software under test; and we finalize or close testing when a phase completes.
- *Preparation* - We need to choose what testing we'll do, by selecting test conditions and designing **test cases.**
- *Evaluation* - As well as executing the tests, we must check the results and evaluate the software under test and the completion criteria, which help us decide whether we have finished testing and whether the software product has passed the tests.

- *Software products and related work products* - We don't just test code. We test the requirements and design specifications, and we test related documents such as operation, user and training material. Static and dynamic testing is both needed to cover the range of products we need to test.

The second part of the definition covers the some of the objectives for testing the reasons why we do it:

- Determine that (software products) satisfy specified requirements - Some of the testing we do is focused on checking products against the specification for the product; for example we review the design to see if it meets requirements, and then we might execute the code to check that it meets the design. If the product meets its specification, we can provide that information to help stakeholders judge the quality of the product and decide whether it is ready for use.
- Demonstrate that (software products) are fit for purpose - This is slightly different to the point above; after all the specified requirements might be wrong or incomplete. 'Fit for purpose' looks at whether the software does enough to help the users to carry out their tasks; we look at whether the soft ware does what the user might reasonably expect. For example, we might look at who might purchase or use the software, and check that it does do what they expect; this might lead us to add a review of the marketing material to our static tests, to check that expectations of the software are properly set. One way of judging the quality of a product is by how fit it is for its purpose.
- Detect defects - We most often think of software testing as a means of detecting faults or defects that in operational use will cause failures. Finding the defects helps us understand the risks associated with putting the software into operational use, and fixing the defects improves the quality of the products. However, identifying defects has another benefit. With root cause analysis, they also help us improve the development processes and make fewer mistakes in future work.

This is a suitable definition of testing for any level of testing, from component testing through to acceptance testing, provided that we remember to take the varying objectives of these different levels of testing into account.

We can clearly see now why the common perception of testing (that it only consists of running tests, i.e. executing the software) is not complete. This is one of the testing activities, but not all of the testing process.

1.2.3 Software test and driving test compared

We can see that the software test is very like a driving test in many ways, although of course it is not a perfect analogy! The driving examiner becomes the software tester. The driver being examined becomes the system or software under test, and you'll see as we go through this book that the same approach broadly holds.

- *Planning and preparation* - Both the examiner and the tester need a plan of action and need to prepare for the test, which is not exhaustive, but is representative and allows risk based decisions about the outcome.

- *Static and dynamic* - Both dynamic (driving the car or executing the soft ware) and static (questions to the driver or a review of the software) tests are useful.
- *Evaluation* - The examiner and the tester must make an objective evaluation, log the test outcome and report factual observations about the test.
- *Determine that they satisfy specified requirements* - The examiner and tester both check against requirements to carry out particular tasks successfully.
- *Demonstrate that they are fit for purpose* - The examiner and the tester are not evaluating for perfection but for meeting sufficient of the attributes required to pass the test.
- *Detect defects* - The examiner and tester both look for and log faults.

Let's think a little more about planning. Because time is limited, in order to make a representative route that would provide a sufficiently good test, both software testers and driving examiners decide in advance on the route they will take. It is not possible to carry out the driving test and make decisions about where to ask the driver to go next on the spur of moment. If the examiner did that, they might run out of time and have to return to the test center without having observed all the necessary maneuvers. The driver will still want a pass/fail report. In the same way, if we embark on testing a software system without a plan of action, we are very likely to run out of time before we know whether we have done enough testing. We'll see that good testers always have a plan of action. In some cases, we use a lightweight outline providing the goals and general direction of the test, allowing the testers to vary the test during execution. In other cases, we use detailed scripts showing the steps in the test route and documenting exactly what the tester should expect to happen as each step. Whichever approach the tester takes, there will be some plan of action. Similarly, just as the driving examiner makes a log and report, a good tester will objectively document defects found and the outcome of the test.

So, test activities exist before and after test execution, and we explain those activities in this book. As a tester or test manager, you will be involved in planning and control of the testing, choosing test conditions, designing test cases based on those test conditions, executing them and checking results, evaluating whether enough testing has been done by Examining completion (or exit) criteria, reporting on the testing process and system under test, and presenting test completion (or summary) reports.

1.2.4 When can we meet our test objectives?

Testing Principle - Early testing

Testing activities should start as early as possible in the software or system development life cycle and should be focused on defined objectives.

We can use both dynamic testing and static testing as a means for achieving similar **test objectives.** Both provide information to improve both the system to be tested, and the

development and testing processes. We mentioned above that testing can have different goals and objectives, which often include:

• finding defects;
• gaining confidence in and providing information about the level of quality;
• preventing defects.

Many types of review and testing activities-take place at different stages in the life cycle, as we'll see in Chapter 2. These have different objectives. Early testing - such as early test design and review activities - finds defects early on when they are cheap to find and fix. Once the code is written, programmers and testers often run a set of tests so that they can identify and fix defects in the software. In this 'development testing' (which includes component, integration and system testing), the main objective may be to cause as many failures as possible so that defects in the software are identified and can be fixed. Following that testing, the users of the software may carry out acceptance testing to confirm that the system works as expected and to gain confidence that it has met the requirements.

Fixing the defects may not always be the test objective or the desired outcome. Sometimes we simply want to gather information and measure the software. This can take the form of attribute measures such as mean time between failures to assess reliability, or an assessment of the defect density in the software to assess and understand the risk of releasing it.

When maintaining software by enhancing it or fixing bugs, we are changing software that is already being used. In that case an objective of testing may be to ensure that we have not made errors and introduced defects when we changed the software. This is called regression testing - testing to ensure nothing has changed that should not have changed.

We may continue to test the system once it is in operational use. In this case, the main objective may be to assess system characteristics such as reliability or availability.

Testing Principle - Defect clustering

A small number of modules contain most of the defects discovered during pre-release testing or show the most operational failures.

1.2.5 Focusing on defects can help us plan our tests

Reviewing defects and failures in order to improve processes allows us to improve our testing and our requirements, design and development processes. One phenomenon that many testers have observed is that defects tend to cluster. This can happen because an area of the code is particularly complex and tricky, or because changing software and other products tends to cause knock-on defects.

Testers will often use this information when making their risk assessment for planning the tests, and will focus on known 'hot spots'.

A main focus of reviews and other static tests is to carry out testing as early as possible, finding and fixing defects more cheaply and preventing defects from appearing at later stages of this project. These activities help us find out about defects earlier and identify potential clusters. Additionally, an important outcome of all testing is information that assists in risk assessment; these reviews will contribute to the planning for the tests executed later in the software development life cycle. We might also carry out root cause analysis to prevent defects and failures happening again and perhaps to identify the cause of clusters and potential future clusters.

1.2.6 The defect clusters change over time

Testing Principle - Pesticide paradox
If the same tests are repeated over and over again, eventually the same set of test cases will no longer find any new bugs. To overcome this 'pesticide paradox', the test cases need to be regularly reviewed and revised, and new and different tests need to be written to exercise different parts of the software or system to potentially find more defects.

Over time, as we improve our whole software development life cycle and the early static testing, we may well find that dynamic test levels find fewer defects. A typical test improvement initiative will initially find more defects as the testing improves and then, as the defect prevention kicks in, we see the defect numbers dropping, as shown in Figure 1.3. The first part of the improvement enables us to reduce failures in operation; later the improvements are making us more efficient and effective in producing the software with fewer defects in it.

As the 'hot spots' for bugs get cleaned up we need to move our focus else-where, to the next set of risks. Over time, our focus may change from finding coding bugs, to looking at the requirements and design documents for defects, and to looking for process improvements so that we prevent defects in the product. Referring to Figure 1.3, by releases 9 and 10, we would expect that the overall cost and effort associated with reviews and testing is much lower than in releases 4 or 7.

1.2.7 Debugging removes defects

When a test finds a defect that must be fixed, a programmer must do some work to locate the defect in the code and make the fix. In this process, called **debugging,** a programmer will examine the code for the immediate cause of the problem, repair the code and check that the code now executes as expected. The fix is often then tested separately (e.g. by an independent tester) to confirm the fix. Notice that testing and debugging are different activities. Developers may test their own fixes, in which case the very tight cycle of identifying faults,

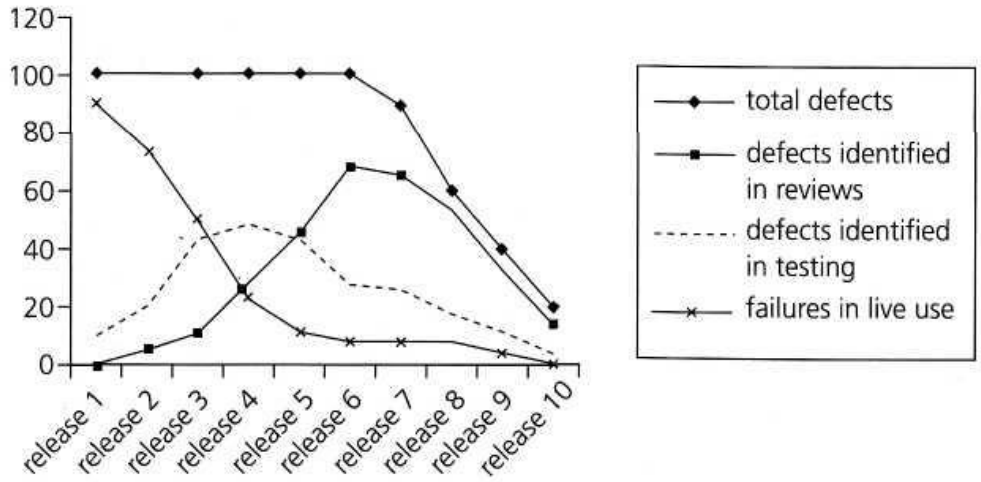


FIGURE 1.3 Changes in defect numbers during process improvement

debugging, and retesting is often loosely referred to as debugging. However, often following the debugging cycle the fixed code is tested independently both to retest the fix itself and to apply regression testing to the surrounding unchanged software.

1.2.8 Is the software defect free?

Testing Principle - Testing shows presence of defects

Testing can show that defects are present, but cannot prove that there are no defects. Testing reduces the probability of undiscovered defects remaining in the software but, even if no defects are found, it is not a proof of correctness.

This principle arises from the theory of the process of scientific experimentation and has been adopted by testers; you'll see the idea in many testing books. While you are not expected to read the scientific theory [Popper] the analogy used in science is useful; however many white swans we see, we cannot say 'All swans are white'. However, as soon as we see one black swan we can say 'Not all swans are white'. In the same way, however many tests we execute without finding a bug, we have not shown 'There are no bugs'. As soon as we find a bug, we have shown 'This code is not bug-free'.

1.2.9 If we don't find defects does that mean the users will accept the software?

Testing Principle - Absence of errors fallacy

Finding and fixing defects does not help if the system built is unusable and does not fulfill the users' needs and expectations.

There is another important principle we must consider; the customers for software the people and organizations who buy and use it to aid in their day-to-day tasks are not interested in defects or numbers of defects, except when they are directly affected by the instability of the software. The people using soft-ware are more interested in the software

supporting them in completing tasks efficiently and effectively. The software must meet their needs.

1.3 Role of Tester:

A Software Tester is required from early stages of SDLC:
In requirements phase – tester does requirement analysis.
In design phase – tester does use-case analysis and starts drafting a Test Plan.
In development phase – tester start developing test cases and test scripts and finalize the test plan.
In testing phase – conduct various types of tests, maintain logs and test summary reports.
In deployment phase – prepare training documentation, lessons learned etc.
In support phase – test the production issues.

1.4 Overview of STLC:

Software Testing Lifecycle is a standard procedure divided into different phases, followed by the QA Team to complete all testing activities. This is a brief tutorial that introduces the readers to the various phases of Software Testing Life Cycle.

STLC stands for Software Testing Life Cycle. STLC is a sequence of different activities performed by the testing team to ensure the quality of the software or the product.

- STLC is an integral part of Software Development Life Cycle (SDLC). But, STLC deals only with the testing phases.
- STLC starts as soon as requirements are defined or SRD (Software Requirement Document) is shared by stakeholders.
- STLC provides a step-by-step process to ensure quality software.
- In the early stage of STLC, while the software or the product is developing, the tester can analyze and define the scope of testing, entry and exit criteria and also the Test Cases. It helps to reduce the test cycle time along with better quality.
- As soon as the development phase is over, the testers are ready with test cases and start with execution. This helps to find bugs in the initial phase.

2. SOFTWARE TESTING LIFE CYCLE

2.1 STLC V-Model

What is STLC V-Model?

One of the major handicaps of waterfall STLC model was that, defects were found at a very later state of the development process, since testing was done at the end of the development cycle. It became very challenging and costly to fix the defects since it were found at a very later stage. To overcome this problem, a new development model was introduced called the “V Model”

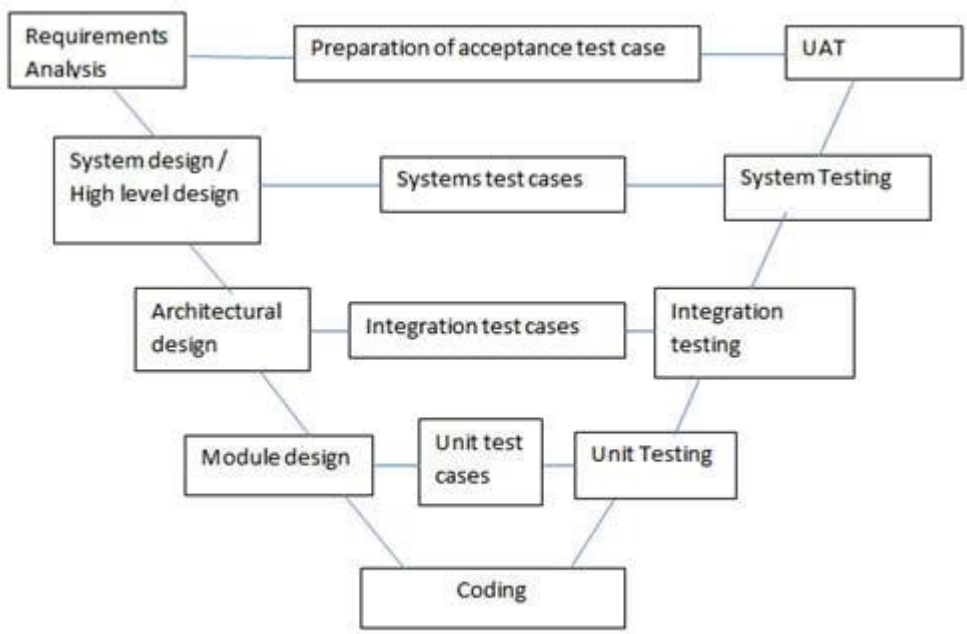
V model is now one of the most widely used software development process. Introduction of V model has actually proved the implementation of testing right from the requirement phase. V model is also called as verification and validation model.

To understand the V model, let's first understand what is verification and validation in software.

***Verification*:** Verification is a static analysis technique. In this technique testing is done without executing the code. Examples include – Reviews, Inspection and walkthrough.

***Validation*:** Validation is a dynamic analysis technique where testing is done by executing the code. Examples include functional and non-functional testing techniques.

In V model, the development and QA activities are done simultaneously. There is no discrete phase called Testing, rather testing starts right from the requirement phase. The verification and validation activities go hand in hand. To understand the V model, let's look at the figure below:



STLC - V Model - © www.SoftwareTestingHelp.com

In a typical development process, the left hand side shows the development activities and right hand side shows the testing activities. I should not be wrong if I say that in the development phase both verification and validation are performed along with the actual development activities. Now let's understand the figure:

Left Hand side:

As said earlier, left hand side activities are the development activities. Normally we feel, ***what testing can we do in development phase,*** but this is the beauty of this model which demonstrates that testing can be done in all phase of development activities as well.

Requirement analysis: In this phase the requirements are collected, analyzed and studied. Here how the system is implemented, is not important but, what the system is supposed to do, is important. Brain storming sessions/walkthrough, interviews are done to have the objectives clear.

- *Verification activities*: Requirements reviews.
- *Validation activities*: Creation of UAT (User acceptance test) test cases
- *Artifacts produced*: Requirements understanding document, UAT test cases.

System requirements / High level design: In this phase a high level design of the software is build. The team studies and investigates on how the requirements could be implemented. The technical feasibility of the requirements is also studied. The team also comes up with the modules that would be created/ dependencies, Hardware / software needs

- *Verification activities*: Design reviews
- *Validation activities*: Creation of System test plan and cases, Creation of traceability metrics
- *Artifacts produced*: System test cases, Feasibility reports, System test plan, Hardware software requirements, and modules to be created etc.

Architectural design: In this phase, based on the high level design**,** software architecture is created. The modules, their relationships and dependencies, architectural diagrams, database tables, technology details are all finalized in this phase.

- *Verification activities*: Design reviews
- *Validation activities*: Integration test plan and test cases.
- *Artifacts produced*: Design documents, Integration test plan and test cases, Database table designs etc.

Module design/ Low level Design: In this phase each and every module or the software components are designed individually. Methods, classes, interfaces, data types etc are all finalized in this phase.

- *Verification activities*: Design reviews
- *Validation activities*: Creation and review of unit test cases.
- *Artifacts produced*: Unit test cases,

Implementation / Code: In this phase, actual coding is done.

- *Verification activities*: Code review, test cases review
- *Validation activities*: Creation of functional test cases.
- *Artifacts produced*: test cases, review checklist.

Right Hand Side:

Right hand side demonstrates the testing activities or the Validation Phase. We will start from bottom.

Unit Testing: In this phase all the unit test case, created in the Low level design phase are executed.

Unit testing is a white box testing technique, where a piece of code is written which invokes a method (or any other piece of code) to test whether the code snippet is giving the expected output or not. This testing is basically performed by the development team. In case of any anomaly, defects are logged and tracked.

Artifacts produced: Unit test execution results

Integration Testing: In this phase the integration test cases are executed which were created in the Architectural design phase. In case of any anomalies, defects are logged and tracked.

*Integration Testing: Integration testing is a technique where the unit tested modules are integrated and tested whether the integrated modules are rendering the expected results. In simpler words, It validates whether the components of the application work together as expected.

Artifacts produced: Integration test results.

Systems testing: In this phase all the system test cases, functional test cases and nonfunctional test cases are executed. In other words, the actual and full fledge testing of the application takes place here. Defects are logged and tracked for its

closure. Progress reporting is also a major part in this phase. The traceability metrics are updated to check the coverage and risk mitigated.

Artifacts produced: Test results, Test logs, defect report, test summary report and updated traceability matrices.

User acceptance Testing: Acceptance testing is basically related to the business requirements testing. Here testing is done to validate that the business requirements are met in the user environment. Compatibility testing and sometimes nonfunctional testing (Load, stress and volume) testing are also done in this phase.

Artifacts produced: UAT results, Updated Business coverage matrices.

When to use V Model?

V model is applicable when:

- Requirement is well defined and not ambiguous
- Acceptance criteria are well defined.
- Project is short to medium in size.
- Technology and tools used are not dynamic.

Pros and Cons of using V model:

PROS	CONS
-Development and progress is very organized and systematic -Works well for smaller to medium sized projects. -Testing starts from beginning so ambiguities are identified from the beginning. -Easy to manage as each phase has well defined objectives and goals.	-Not suitable for bigger and complex projects -Not suitable if the requirements are not consistent. -No working software is produced in the intermediate stage. -No provision for doing risk analysis so uncertainty and risks are there.

2.2 SDLC vs STLC

- STLC is part of SDLC. It can be said that STLC is a subset of the SDLC set.
- STLC is limited to the testing phase where quality of software or product ensures. SDLC has vast and vital role in complete development of a software or product.
- However, STLC is a very important phase of SDLC and the final product or the software cannot be released without passing through the STLC process.
- STLC is also a part of the post-release/ update cycle, the maintenance phase of SDLC where known defects get fixed or a new functionality is added to the software.

The following table lists down the factors of comparison between SDLC and STLC based on their phases −

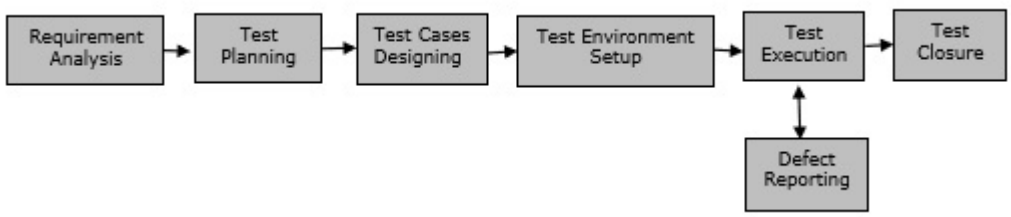
Phase	**SDLC**	**STLC**
Requirement Gathering	• Business Analyst gathers requirements. • Development team analyzes the requirements. • After high level, the development team starts analyzing from the architecture and the design perspective.	• Testing team reviews and analyzes the SRD document. • Identifies the testing requirements - Scope, Verification and Validation key points. • Reviews the requirements for logical and functional relationship among various modules. This helps in the identification of gaps at an early stage.
Design	• The architecture of SDLC helps you develop a high-level and low-level design of the software based on the requirements. • Business Analyst	• In STLC, either the Test Architect or a Test Lead usually plan the test strategy. • Identifies the testing points. • Resource allocation and timelines are

	works on the mocker of UI design. • Once the design is completed, it is signed off by the stakeholders.	finalized here.
Development	• Development team starts developing the software. • Integrate with different systems. • Once all integration is done, a ready to test software or product is provided.	• Testing team writes the test scenarios to validate the quality of the product. • Detailed test cases are written for all modules along with expected behaviour. • The prerequisites and the entry and exit criteria of a test module are identified here.
Environment Set up	• Development team sets up a test environment with developed product to validate.	• The Test team confirms the environment set up based on the prerequisites. • Performs smoke testing to make sure the environment is stable for the product to be tested.
Testing	• The actual testing is carried out in this phase. It includes unit testing, integration testing, system testing, defect retesting, regression testing, etc. • The Development team fixes the bug	• System Integration testing starts based on the test cases. • Defects reported, if any, get retested and fixed. • Regression testing is performed here and the product is signed off once it meets the exit

	reported, if any and sends it back to the tester for retesting. • UAT testing performs here after getting sign off from SIT testing.	criteria.
Deployment/ Product Release	• Once sign-off is received from various testing team, application is deployed in prod environment for real end users.	• Smoke and sanity testing in production environment is completed here as soon as product is deployed. • Test reports and matrix preparation are done by testing team to analyze the product.
Maintenance	• It covers the post deployment supports, enhancement and updates, if any.	• In this phase, the maintaining of test cases, regression suits and automation scripts take place based on the enhancement and updates.

2.3 STLC Phases (Different stages in STLC)

STLC has the following different phases but it is not mandatory to follow all phases. Phases are dependent on the nature of the software or the product, time and resources allocated for the testing and the model of SDLC that is to be followed.



There are 6 major phases of STLC −

- **Requirement Analysis** − When the SRD is ready and shared with the stakeholders, the testing team starts high level analysis concerning the AUT (Application under Test).
- **Test Planning** − Test Team plans the strategy and approach.
- **Test Case Designing** − Develop the test cases based on scope and criteria's.
- **Test Environment Setup** − When integrated environment is ready to validate the product.
- **Test Execution** − Real-time validation of product and finding bugs.
- **Test Closure** − Once testing is completed, matrix, reports, results are documented.

2.4 Document templates generated in different phases of STLC

Requirement Analysis − The Matrix is created at the very beginning of a project as it forms the basis of the project's scope and deliverables that will be produced.

- **Test Planning** − The basic entry criteria of this phase is provision of Test plan/strategy document .
- **Test Case Designing** – Test cases/scripts
- **Test Environment Setup** –
- **Test Execution** –
- **Test Closure** –

2.5 TEST LEVELS

The V-model for testing was introduced in Section 2.1. This section looks in more detail at the various test levels. The key characteristics for each test level are discussed and defined to be able to more clearly separate the various test levels. A thorough understanding and definition of the various test levels will identify missing areas and prevent overlap and repetition. Sometimes we may wish to introduce deliberate overlap to address specific risks. Understanding whether we want overlaps and removing the gaps will make the test levels more complementary thus leading to more effective and efficient testing.

2.5.1 Component testing

Component testing, also known as unit, module and program testing, searches for defects in, and verifies the functioning of software (e.g. modules, programs, objects, classes, etc.) that are separately testable.

Component testing may be done in isolation from the rest of the system depending on the context of the development life cycle and the system. Most often **stubs** and **drivers** are used to replace the missing software and simulate the interface between the software components in a simple manner. A stub is called from the software component to be tested; a driver calls a component to be tested (see Figure 2.5).

Component testing may include testing of functionality and specific non-functional characteristics such as resource-behavior (e.g. memory leaks), performance or **robustness testing,** as well as structural testing (e.g. decision coverage). Test cases are derived from work products such as the software design or the data model.

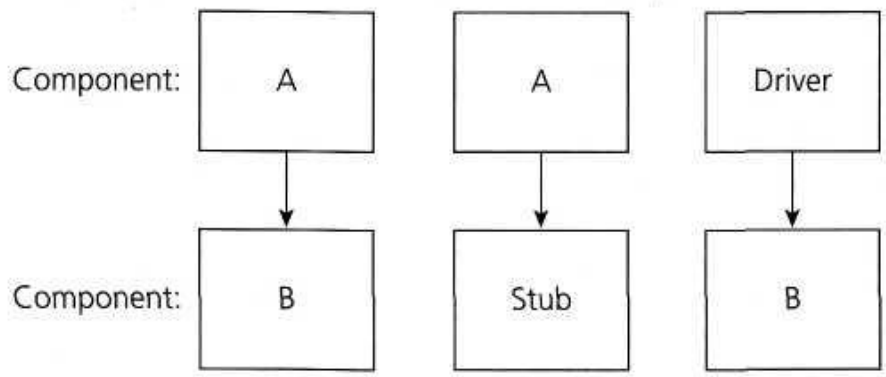


FIGURE 2.5 Stubs and drivers

with the support of the development environment, such as a unit test frame-work or debugging tool, and in practice usually involves the programmer who wrote the code. Sometimes, depending on the applicable level of risk, component testing is carried out by a different programmer thereby introducing independence. Defects are typically fixed as soon as they are found, without formally recording the incidents found.

One approach in component testing, used in Extreme Programming (XP), is to prepare and automate test cases before coding. This is called a test-first approach or **test-driven development.** This approach is highly iterative and is based on cycles of developing test cases, then building and integrating small pieces of code, and executing the component tests until they pass.

2.5.2 Integration testing

Integration testing tests interfaces between components, interactions to different parts of a system such as an operating system, file system and hard-ware or interfaces between systems. Note that integration testing should be differentiated from other integration activities. Integration testing is often carried out by the integrator, but preferably by a specific integration tester or test team.

There may be more than one level of integration testing and it may be carried out on test objects of varying size. For example:

- Component integration testing tests the interactions between software components and is done after component testing;
- System integration testing tests the interactions between different systems and may be done after system testing. In this case, the developing organization may control only one side of the interface, so changes may be destabilizing. Business processes implemented as workflows may involve a series of systems that can even run on different platforms.

The greater the scope of integration, the more difficult it becomes to isolate failures to a specific interface, which may lead to an increased risk. This leads to varying approaches to integration testing. One extreme is that all components or systems are integrated simultaneously, after which everything is tested as a whole. This is called 'big-bang' integration testing. Big-bang testing has the advantage that everything is finished before integration testing starts. There is no need to simulate (as yet unfinished) parts. The major disadvantage is that in general it is time-consuming and difficult to trace the cause of failures with this late integration. So big-bang integration may seem like a good idea when planning the project, being optimistic and expecting to find no problems. If one thinks integration testing will find defects, it is a good practice to consider whether time might be saved by breaking the down the integration test process. Another extreme is that all programs are integrated one by one, and a test is carried out after each step (incremental testing). Between these two extremes, there is a range of variants. The incremental approach has the advantage that the defects are found early in a smaller assembly when it is relatively easy to detect the cause. A disadvantage is that it can be time-consuming since stubs and drivers have to be developed and used in the test. Within incremental integration testing a range of possibilities exist, partly depending on the system architecture:

- Top-down: testing takes place from top to bottom, following the control flow or architectural structure (e.g. starting from the GUI or main menu). Components or systems are substituted by stubs.
- Bottom-up: testing takes place from the bottom of the control flow upwards. Components or systems are substituted by drivers.
- Functional incremental: integration and testing takes place on the basis of the functions or functionality, as documented in the functional specification.

The preferred integration sequence and the number of integration steps required depend on the location in the architecture of the high-risk interfaces. The best choice is to start integration with those interfaces that are expected to cause most problems. Doing so prevents major defects at the end of the integration test stage. In order to reduce the risk of late defect discovery, integration should normally be incremental rather than 'big-bang'. Ideally testers should understand the architecture and influence integration planning. If integration tests are planned before components or systems are built, they can be developed in the order required for most efficient testing.

At each stage of integration, testers concentrate solely on the integration itself. For example, if they are integrating component A with component B they are interested in testing the communication between the components, not the functionality of either one. Both functional and structural approaches may be used. Testing of specific non-functional characteristics (e.g. performance) may also be included in integration testing. Integration testing may be carried out by the developers, but can be done by a separate team of specialist integration testers, or by a specialist group of developers/integrators including non-functional specialists.

2.5.3 System testing

System testing is concerned with the behavior of the whole system/product as defined by the scope of a development project or product. It may include tests based on risks and/or requirements specification, business processes, use cases, or other high level descriptions of system behavior, interactions with the operating system, and system resources. System testing is most often the final test on behalf of development to verify that the system to be delivered meets the specification and its purpose may be to find as many defects as possible. Most often it is carried out by specialist testers that form a dedicated, and sometimes independent, test team within development, reporting to the development manager or project manager. In some organizations system testing is carried out by a third party team or by business analysts. Again the required level of independence is based on the applicable risk level and this will have a high influence on the way system testing is organized.

System testing should investigate both **functional** and **non-functional requirements** of the system. Typical non-functional tests include performance and reliability. Testers may also need to deal with incomplete or undocumented requirements. System testing of functional requirements starts by using the most appropriate specification-based (black-box) techniques for the aspect of the system to be tested. For example, a decision table may be created for combinations of effects described in business rules. Structure-based (white-box) techniques may also be used to assess the thoroughness of testing elements such as menu dialog structure or web page navigation (see Chapter 4 for more on the various types of technique).

System testing requires a controlled **test environment** with regard to, amongst other things, control of the software versions, testware and the test data (see Chapter 5 for more on configuration management). A system test is executed by the development organization in a (properly controlled) environment. The test environment should correspond to the final target or production environment as much as possible in order to minimize the risk of environment-specific failures not being found by testing.

2.5.4 Acceptance testing

When the development organization has performed its system test and has corrected all or most defects, the system will be delivered to the user or customer for **acceptance testing.** The acceptance test should answer questions such as: 'Can the system be released?', 'What, if any, are the outstanding (business) risks?' and 'Has development met

their obligations?'. Acceptance testing is most often the responsibility of the user or customer, although other stakeholders may be involved as well. The execution of the acceptance test requires a test environment that is for most aspects, representative of the production environment ('as-if production').

The goal of acceptance testing is to establish confidence in the system, part of the system or specific non-functional characteristics, e.g. usability, of the system. Acceptance testing is most often focused on a validation type of testing, whereby we are trying to determine whether the system is fit for purpose. Finding defects should not be the main focus in acceptance testing. Although it assesses the system's readiness for deployment and use, it is not necessarily the final level of testing. For example, a large-scale system integration test may come after the acceptance of a system.

Acceptance testing may occur at more than just a single level, for example:

- A Commercial Off The Shelf (COTS) software product may be acceptance tested when it is installed or integrated.
- Acceptance testing of the usability of a component may be done during component testing.
- Acceptance testing of a new functional enhancement may come before system testing.

Within the acceptance test for a business-supporting system, two main test types can be distinguished; as a result of their special character, they are usually prepared and executed separately. The user acceptance test focuses mainly on the functionality thereby validating the fitness-for-use of the system by the business user, while the **operational acceptance test** (also called production acceptance test) validates whether the system meets the requirements for operation. The user acceptance test is performed by the users and application managers. In terms of planning, the user acceptance test usually links tightly to the system test and will, in many cases, be organized partly overlapping in time. If the system to be tested consists of a number of more or less independent subsystems, the acceptance test for a subsystem that complies with the exit criteria of the system test can start while another subsystem may still be in the system test phase. In most organizations, system administration will perform the operational acceptance test shortly before the system is released. The operational acceptance test may include testing of backup/restore, disaster recovery, maintenance tasks and periodic check of security vulnerabilities.

Other types of acceptance testing that exist are contract acceptance testing and **compliance acceptance testing.** Contract acceptance testing is performed against a contract's acceptance criteria for producing custom-developed software. Acceptance should be formally defined when the contract is agreed. Compliance acceptance testing or regulation acceptance testing is performed against the regulations which must be adhered to, such as governmental, legal or safety regulations.

If the system has been developed for the mass market, e.g. commercial off-the-shelf software (COTS), then testing it for individual users or customers is not practical or even possible in some cases. Feedback is needed from potential or existing users in their market before the software product is put out for sale commercially. Very often this type of system undergoes two stages of acceptance test. The first is called **alpha testing.** This test takes place at the developer's site. A cross-section of potential users and members of the developer's organization are invited to use the system. Developers observe the users

and note problems. Alpha testing may also be carried out by an independent test team. **Beta testing,** or field testing, sends the system to a cross-section of users who install it and use it under real-world working conditions. The users send records of incidents with the system to the development organization where the defects are repaired.

Note that organizations may use other terms, such as factory acceptance testing and site acceptance testing for systems that are tested before and after being moved to a customer's site.

2.6 TEST TYPES: Functional testing

Test types are introduced as a means of clearly defining the objective of a certain test level for a programme or project. We need to think about different types of testing because testing the functionality of the component or system may not be sufficient at each level to meet the overall test objectives. Focusing the testing on a specific test objective and, therefore, selecting the appropriate type of test helps making and communicating decisions against test objectives easier.

1. Testing of function (functional testing)

The function of a system (or component) is 'what it does'. This is typically described in a requirements specification, a functional specification, or in use cases. There may be some functions that are 'assumed' to be provided that are not documented that are also part of the requirement for a system, though it is difficult to test against undocumented and implicit requirements. Functional tests are based on these functions, described in documents or understood by the testers and may be performed at all test levels (e.g. test for components may be based on a component specification).

Functional testing considers the specified behavior and is often also referred to as **black-box testing. Function** (or functionality) **testing** can, based upon ISO 9126, be done focusing on suitability, **interoperability, security,** accuracy and compliance. Security testing, for example, investigates the functions (e.g. a firewall) relating to detection of threats, such as viruses, from malicious outsiders.

Testing functionality can be done from two perspectives: requirements-based or business-process-based.

Requirements-based testing uses a specification of the functional requirements for the system as the basis for designing tests. A good way to start is to use the table of contents of the requirements specification as an initial test inventory or list of items to test (or not to test). We should also prioritize the requirements based on risk criteria (if this is not already done in the specification) and use this to prioritize the tests. This will ensure that the most important and most critical tests are included in the testing effort.

Business-process-based testing uses knowledge of the business processes. Business processes describe the scenarios involved in the day-to-day business use of the system. For example, a personnel and payroll system may have a business process along the lines of: someone joins the company, he or she is paid on a regular basis, and he or she finally leaves the company. Use cases originate from object-oriented development, but are

nowadays popular in many development life cycles. They also take the business processes as a starting point, although they start from tasks to be performed by users. Use cases are a very useful basis for test cases from a business perspective.

The techniques used for functional testing are often **specification-based,** but experienced-based techniques can also be used (see Chapter 4 for more on test techniques). Test conditions and test cases are derived from the functionality of the component or system. As part of test designing, a model may be developed, such as a process model, state transition model or a plain-language specification.

2.7 API Testing

What is an API?

API stands for **A**pplication **P**rogramming **I**nterface, which specifies how one component should interact with the other. It consists of a set of routines, protocols and tools for building the software applications.

What is an API Testing?

The API Testing is performed for the system, which has a collection of API that ought to be tested. During Testing, a test of following things is looked at.

- Exploring boundary conditions and ensuring that the test harness varies parameters of the API calls in ways that verify functionality and expose failures.
- Generating more value added parameter combinations to verify the calls with two or more parameters.
- Verifying the behaviour of the API which is considering the external environment conditions such as files, peripheral devices, and so forth.
- Verifying the Sequence of API calls and check if the API's produce useful results from successive calls.

Common Tests performed on API's

- Return Value based on input condition - The return value from the API's are checked based on the input condition.
- Verify if the API's does not return anything.
- Verify if the API triggers some other event or calls another API. The Events output should be tracked and verified.
- Verify if the API is updating any data structure.

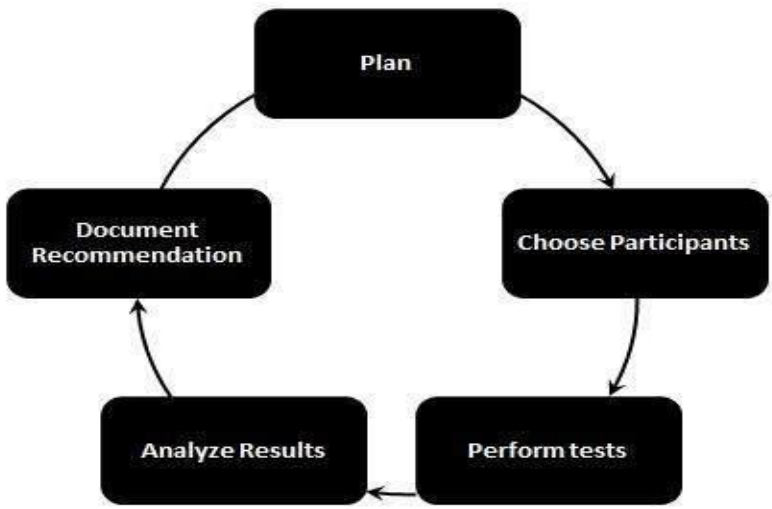
2.8 Usability Testing

What is Usability Testing ?

Usability testing, a non-functional testing technique that is a measure of how easily the system can be used by end users. It is difficult to evaluate and measure but can be evaluated based on the below parameters:

- Level of Skill required to learn/use the software. It should maintain the balance for both novice and expert user.
- Time required to get used to in using the software.
- The measure of increase in user productivity if any.
- Assessment of a user's attitude towards using the software.

Usability Testing Process:



2.9. Exploratory Testing

What is an Exploratory Testing?

During testing phase where there is severe time pressure, Exploratory testing technique is adopted that combines the experience of testers along with a structured approach to testing.

Exploratory testing often performed as a black box testing technique, the tester learns things that together with experience and creativity generate new good tests to run.

Benefits:

Following are the benefits of Exploratory Testing:

- Exploratory testing takes less preparation.
- Critical defects are found very quickly.
- The testers can use reasoning based approach on the results of previous results to guide their future testing on the fly.

Drawbacks:

Following are the Drawbacks of Exploratory Testing:

- Tests cannot be reviewed.
- It is difficult to keep track of what tests have been tested.
- It is unlikely to be performed in exactly the same manner and to repeat specific details of the earlier tests.

2.10. Adhoc Testing

What is Adhoc Testing?

When a software testing performed without proper planning and documentation, it is said to be Adhoc Testing. Such kind of tests are executed only once unless we uncover the defects.

Adhoc Tests are done after formal testing is performed on the application. Adhoc methods are the least formal type of testing as it is NOT a structured approach. Hence, defects found using this method are hard to replicate as there are no test cases aligned for those scenarios.

Testing is carried out with the knowledge of the tester about the application and the tester tests randomly without following the specifications/requirements. Hence the success of Adhoc testing depends upon the capability of the tester, who carries out the test. The tester has to find defects without any proper planning and documentation, solely based on tester's intuition.

When to Execute Adhoc Testing ?

Adhoc testing can be performed when there is limited time to do exhaustive testing and usually performed after the formal test execution. Adhoc testing will be effective only if the tester has in-depth understanding about the System Under Test.

Forms of Adhoc Testing :

1. **Buddy Testing:** Two buddies, one from development team and one from test team mutually work on identifying defects in the same module. Buddy testing helps the testers develop better test cases while development team can also make design changes early. This kind of testing happens usually after completing the unit testing.
2. **Pair Testing:** Two testers are assigned the same modules and they share ideas and work on the same systems to find defects. One tester executes the tests while another tester records the notes on their findings.
3. **Monkey Testing:** Testing is performed randomly without any test cases in order to break the system.

Various ways to make Adhoc Testing More Effective

1. **Preparation:** By getting the defect details of a similar application, the probability of finding defects in the application is more.
2. **Creating a Rough Idea:** By creating a rough idea in place the tester will have a focussed approach. It is NOT required to document a detailed plan as what to test and how to test.
3. **Divide and Rule:** By testing the application part by part, we will have a better focus and better understanding of the problems if any.
4. **Targeting Critical Functionalities:** A tester should target those areas that are NOT covered while designing test cases.
5. **Using Tools:** Defects can also be brought to the lime light by using profilers, debuggers and even task monitors. Hence being proficient in using these tools one can uncover several defects.
6. **Documenting the findings:** Though testing is performed randomly, it is better to document the tests if time permits and note down the deviations if any. If defects are found, corresponding test cases are created so that it helps the testers to retest the scenario.

2.11. Static Testing

The definition of testing outlines objectives that relate to evaluation, revealing defects and quality. As indicated in the definition two approaches can be used to achieve these objectives, **static testing** and **dynamic testing.**

With dynamic testing methods, software is executed using a set of input values and its output is then examined and compared to what is expected. During static testing, software work products are examined manually, or with a set of tools, but not executed. As a consequence, dynamic testing can only be applied to software code.

What is Static Testing?

Static Testing, a software testing technique in which the software is tested without executing the code

Reviews:

A review is a systematic examination of a document by one or more people with the main aim of finding and removing errors early in the software development life cycle. Reviews are used to verify documents such as requirements, system designs, code, test plans and test cases.

Types of Review:

1.Walkthrough: A walkthrough is characterized by the author of the document under review guiding the participants through the document and his or her thought processes, to achieve a common understanding and to gather feedback. This is especially useful if people from outside the software discipline are present, who are not used to, or cannot easily understand software development documents. The content of the document is explained step by step by the author, to reach consensus on changes or to gather information.

The specific goals of a walkthrough depend on its role in the creation of the document. In general the following goals can be applicable:

- To present the document to stakeholders both within and outside the soft ware discipline, in order to gather information regarding the topic under documentation;
- To explain (knowledge transfer) and evaluate the contents of the docu ment;
- Tto establish a common understanding of the document;
- To examine and discuss the validity of proposed solutions and the viability of alternatives, establishing consensus.

Key characteristics of walkthroughs are:

- The meeting is led by the authors; often a separate scribe is present.
- Scenarios and dry runs may be used to validate the content.
- Separate pre-meeting preparation for reviewers is optional.

2. Technical review: Is a discussion meeting that focuses on achieving consensus about the technical content of a document.

The goals of a technical review are to:

- Aassess the value of technical concepts and alternatives in the product and project environment;
- Eestablish consistency in the use and representation of technical concepts;
- Ensure, at an early stage, that technical concepts are used correctly;
- Inform participants of the technical content of the document.

Key characteristics of a technical review are:

- It is a documented defect-detection process that involves peers and technical experts.
- It is often performed as a peer review without management partici pation.
- Ideally it is led by a trained moderator, but possibly also by a technical expert.
- A separate preparation is carried out during which the product is examined and the defects are found.
- More formal characteristics such as the use of checklists and a logging list or issue log are optional.

3. Inspection is the most formal review type. It is usually led by a trained moderator (certainly not by the author).The document under inspection is prepared and checked thoroughly by the reviewers before the meeting, comparing the work product with its sources and other referenced documents, and using rules and checklists.

The generally accepted goals of inspection are to:

- Help the author to improve the quality of the document under inspection;
- Remove defects efficiently, as early as possible;
- Improve product quality, by producing documents with a higher level of quality;
- Create a common understanding by exchanging information among the inspection participants;
- Train new employees in the organization's development process;
- Llearn from defects found and improve processes in order to prevent recur rence of similar defects;
- Sample a few pages or sections from a larger document in order to measure the typical quality of the document, leading to improved work by individuals in the future, and to process improvements.

Key characteristics of an inspection are:

- It is usually led by a trained moderator (certainly not by the author).
- It uses defined roles during the process.
- It involves peers to examine the product.
- Rules and checklists are used during the preparation phase.
- A separate preparation is carried out during which the product is examined and the defects are found.
- The defects found are documented in a logging list or issue log.
- A formal follow-up is carried out by the moderator applying exit criteria.
- Optionally, a causal analysis step is introduced to address process improve ment issues and learn from the defects found.
- Metrics are gathered and analyzed to optimize the process.

3. BASICS OF TEST DESIGN TECHNIQUES:

3.1 Various test categories and 3.2 Test design techniques for different categories of tests

In this section we will look at the different types of test design technique, how they are used and how they differ. The three types or categories are distin-guished by their primary source: a specification, the structure of the system or component, or a person's experience. All categories are useful and the three are complementary.

In this section, look for the definitions of the glossary terms: **black-box test design techniques, experience-based test design techniques, specification-based test design techniques, structure-based test design techniques** and **white-box test design techniques.**

Introduction

There are many different types of software testing technique, each with its own strengths and weaknesses. Each individual technique is good at finding particular types of defect and relatively poor at finding other types. For example, a technique that explores the upper and lower limits of a single input range is more likely to find boundary value defects than defects associated with combinations of inputs. Similarly, testing performed at different stages in the software development life cycle will find different types of defects; component testing is more likely to find coding logic defects than system design defects.

Each **testing technique** falls into one of a number **of different categories**. Broadly speaking there **are two main categories, static and dynamic**. Dynamic techniques are subdivided into three more categories: specification-based (black-box, also known as behavioral techniques), structure-based (white-box or structural techniques) and experience based. Specification-based techniques include both functional and non-functional techniques (i.e. quality characteristics). The techniques covered in the syllabus are summarized in Figure 4.1.

Static testing techniques:

Static testing techniques do not execute the code being examined and are generally used before any tests are executed on the software. They could be called non-execution techniques. Most static testing techniques can be used to 'test' any form of document including source code, design documents and models, functional specifications and requirement specifications. However, 'static analysis' is a tool-supported type of static testing that concentrates on testing formal languages and so is most often used to statically test source code.

- **data-flow analysis**: is a technique for gathering information about the possible set of values calculated at various locations in a computer program
- **control-flow analysis**: is a technique for determining the order of operations in a computer program.

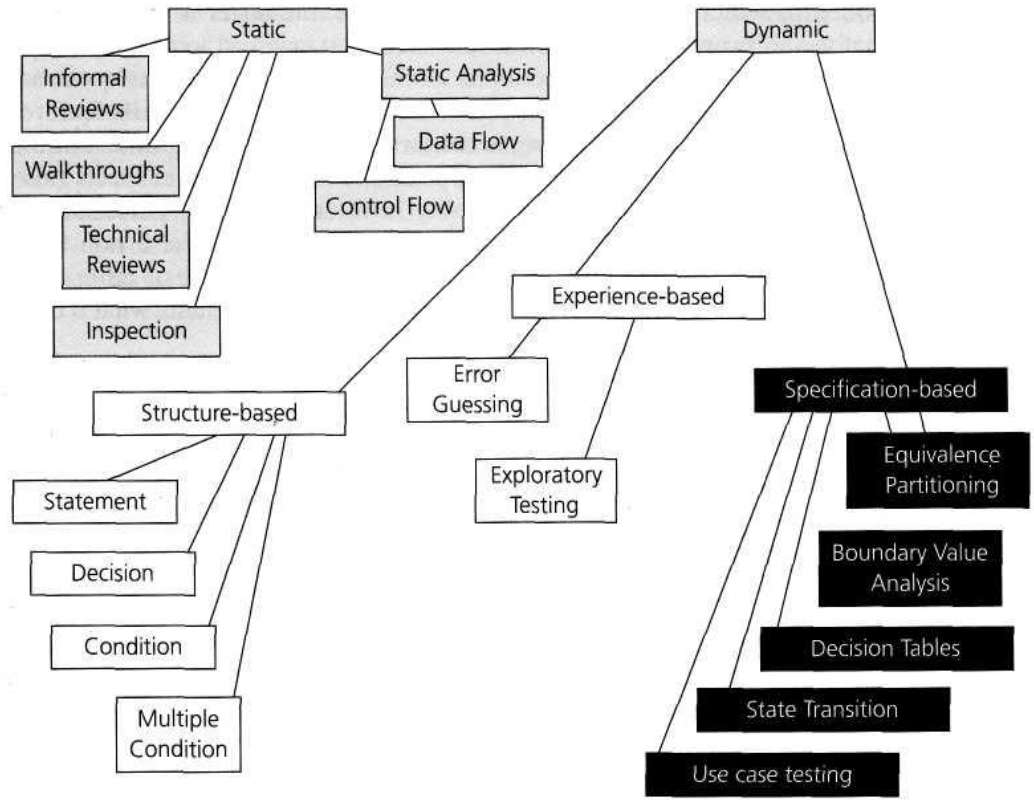


FIGURE 4.1 Testing techniques

Structure-based (white-box) testing techniques:

Structure-based testing techniques (which are also dynamic rather than static) use the internal structure of the software to derive test cases. They are commonly called **'white-box'** or 'glass-box' techniques (implying you can see into the system) since they require knowledge of how the software is implemented, that is, how it works. For example, a structural technique may be concerned with exercising loops in the software. Different test cases may be derived to exercise the loop once, twice, and many times. This may be done regardless of the functionality of the software.

- **Statement testing**

Statement testing is a white box testing approach in which test scripts are designed to execute code statements. The statement coverage is the measure of the percentage of statements of code executed by the test scripts out of the total code statements in the application. The statement coverage is the least preferred metric for checking test coverage.

- **Decision testing/branch testing**

Decision testing or branch testing is a white box testing approach in which test coverage is measured by the percentage of decision points(e.g. if-else conditions) executed out of the total decision points in the application.

- **Condition testing**

Testing the condition outcomes(TRUE or FALSE). So, getting 100% condition coverage required exercising each condition for both TRUE and FALSE results using test scripts(For n conditions we will have 2n test scripts).

- **Multiple condition testing**

Testing the different combinations of condition outcomes. Hence for 100% coverage we will have 2^n test scripts. This is very exhaustive and very difficult to achieve 100% coverage.

Experience-based testing techniques :

In **experience-based techniques,** people's knowledge, skills and background are a prime contributor to the test conditions and test cases. The experience of both technical and business people is important, as they bring different perspectives to the test analysis and design process. Due to previous experience with similar systems, they may have insights into what could go wrong, which is very useful for testing.

- **Exploratory Testing:**

Usually this process will be carried out by domain experts. They perform testing just by exploring the functionalities of the application without having the knowledge of the requirements.

Whilst using this technique, testers could explore and learn the system. High severity bugs are found very quickly in this type of testing.

- **Error Guessing:**

Error guessing is one of the testing techniques used to find bugs in a software application based on tester's prior experience. In Error guessing we don't follow any specific rules.

Some of the examples are:

- Submitting a form without entering values.
- Entering invalid values such as entering alphabets in the numeric field.

Specification-based (black-box) testing techniques:
The first of the dynamic testing techniques we will look at are the specification-based testing techniques. These are also known as '**black-box**' or input/output-driven testing techniques because they view the software as a Black-box with inputs and outputs, but they have no knowledge of how the system or component is structured inside the box. In essence, the tester is concentrating on what the software does, not how it does it.
Notice that the definition mentions both functional and non-functional testing. Functional testing is concerned with what the system does, its features or functions. Non-functional testing is concerned with examining how well the system does something, rather than what it does. Non-functional aspects (also known as quality characteristics or quality attributes) include performance, usability, portability, maintainability, etc. Techniques to test these non-functional aspects are less procedural and less formalized than those of other categories as the actual tests are more dependent on the type of system, what it does and the resources available for the tests.

- **Equivalence Partitioning:**

It is also known as Equivalence Class Partitioning (ECP).

Using equivalence partitioning test design technique, we divide the test conditions into class (group). From each group we do test only one condition. Assumption is that all the conditions in one group works in the same manner. If a condition from a group works then all of the conditions from that group work and vice versa. It reduces lots of rework and also gives the good test coverage. We could save lots of time by reducing total number of test cases that must be developed.

For example: A field should accept numeric value. In this case, we split the test conditions as Enter numeric value, Enter alpha numeric value, Enter Alphabets, and so on. Instead of testing numeric values such as 0, 1, 2, 3, and so on.

- **Boundary Value Analysis:**

Using Boundary value analysis (BVA), we take the test conditions as partitions and design the test cases by getting the boundary values of the partition. The boundary

between two partitions is the place where the behavior of the application varies. The test conditions on either side of the boundary are called boundary values. In this we have to get both valid boundaries (from the valid partitions) and invalid boundaries (from the invalid partitions).

For example: If we want to test a field which should accept only amount more than 10 and less than 20 then we take the boundaries as 10-1, 10, 10+1, 20-1, 20, 20+1. Instead of using lots of test data, we just use 9, 10, 11, 19, 20 and 21.

- **Decision Table:**

Decision Table is aka Cause-Effect Table. This test technique is appropriate for functionalities which has logical relationships between inputs (if-else logic). In Decision table technique, we deal with combinations of inputs. To identify the test cases with decision table, we consider conditions and actions. We take conditions as inputs and actions as outputs.

- **State Transition Testing:**

Using state transition testing, we pick test cases from an application where we need to test different system transitions. We can apply this when an application gives a different output for the same input, depending on what has happened in the earlier state.

Some examples are Vending Machine, Traffic Lights.

Vending machine dispenses products when the proper combination of coins is deposited.

Traffic Lights will change sequence when cars are moving / waiting

Where to apply the different categories of techniques :

Specification-based techniques are appropriate at all levels of testing (component testing through to acceptance testing) where a specification exists. When performing system or acceptance testing, the requirements specification or functional specification may form the basis of the tests. When performing component or integration testing, a design document or low-level specification forms the basis of the tests.

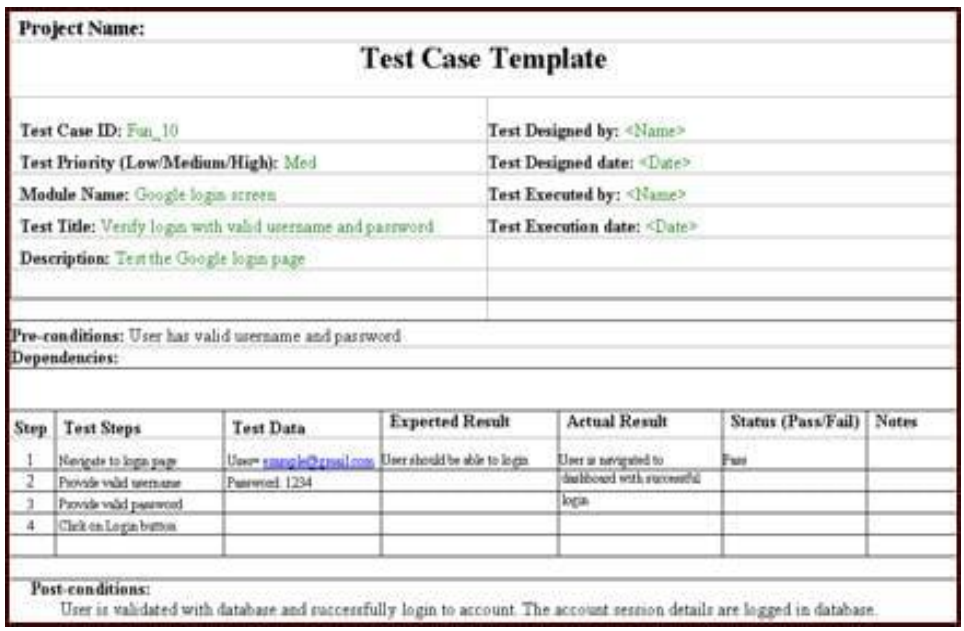
Structure-based techniques can also be used at all levels of testing. Developers use structure-based techniques in component testing and component integration testing, especially where there is good tool support for code coverage. Structure-based techniques are also used in system and acceptance
testing, but the structures are different. For example, the coverage of menu options or major business transactions could be the structural element in system or acceptance testing.

Experience-based techniques are used to complement specification-based and structure-based techniques, and are also used when there is no specification, or if the specification

is inadequate or out of date. This may be the only type of technique used for low-risk systems, but this approach may be particularly useful under extreme time pressure - in fact this is one of the factors leading to exploratory testing.

3.3 Designing test cases using MS-Excel.

Test case formats may vary from one organization to another. But using a standard test case format for writing test cases is one step closer to set up a testing process for your project.
It also minimizes ad-hoc testing that is done without proper test case documentation. But even if you use standard templates, you need to set up test cases writing, review & approve, test execution and most importantly test report preparation process, etc by using manual methods.
Also if you have a process to review the test cases by the business team, then you must format these test cases in a template that is agreed by both the parties.
Here is how to make this manual test case management process a bit easier with the help of simple testing templates.



Project Name:

Test Case Template

Test Case ID: Fun_10	**Test Designed by:** <Name>
Test Priority (Low/Medium/High): Med	**Test Designed date:** <Date>
Module Name: Google login screen	**Test Executed by:** <Name>
Test Title: Verify login with valid username and password	**Test Execution date:** <Date>
Description: Test the Google login page	

Pre-conditions: User has valid username and password
Dependencies:

Step	Test Steps	Test Data	Expected Result	Actual Result	Status (Pass/Fail)	Notes
1	Navigate to login page	User= example@gmail.com	User should be able to login	User is navigated to	Pass	
2	Provide valid username	Password: 1234		dashboard with successful		
3	Provide valid password			login		
4	Click on Login button					

Post-conditions:
User is validated with database and successfully login to account. The account session details are logged in database

Several standard fields of a sample Test Case template are listed below.

Test case ID: Unique ID is required for each test case. Follow some convention to indicate the types of the test. E.g. 'TC_UI_1' indicating 'user interface test case #1'.

Test priority (Low/Medium/High): This is very useful while test execution. Test priority for business rules and functional test cases can be medium or higher whereas minor user

interface cases can be of a low priority. Test priority should always be set by the reviewer.

Module Name: Mention the name of the main module or the sub-module.

Test Designed By: Name of the Tester.

Test Designed Date: Date when it was written.

Test Executed By: Name of the Tester who executed this test. To be filled only after test execution.

Test Execution Date: Date when the test was executed.

Test Title/Name: Test case title. E.g. verify login page with a valid username and password.

Test Summary/Description: Describe the test objective in brief.

Pre-conditions: Any prerequisite that must be fulfilled before the execution of this test case. List all the pre-conditions in order to execute this test case successfully.

Dependencies: Mention any dependencies on the other test cases or test requirement.

Test Steps: List all the test execution steps in detail. Write test steps in the order in which they should be executed. Make sure to provide as many details as you can. Tip – In order to manage a test case efficiently with a lesser number of fields use this field to describe the test conditions, test data and user roles for running the test.

Test Data: Use of test data as an input for this test case. You can provide different data sets with exact values to be used as an input.

Expected Result: What should be the system output after test execution? Describe the expected result in detail including message/error that should be displayed on the screen.

Post-condition: What should be the state of the system after executing this test case?

Actual result: Actual test result should be filled after test execution. Describe the system behavior after test execution.

Status (Pass/Fail): If actual result is not as per the expected result, then mark this test as failed. Otherwise, update it as passed.

Notes/Comments/Questions: If there are some special conditions to support the above fields, which can't be described above or if there are any questions related to expected or actual results then mention them here.

Add the following fields if necessary:

Defect ID/Link: If the test status is failed, then include the link to the defect log or mention the defect number.

Test Type/Keywords: This field can be used to classify the tests based on test types.

E.g. functional, usability, business rules etc.

Requirements: Requirements for which this test case is being written for. Preferably the exact section number of the requirement doc.

Attachments/References: This field is useful for complex test scenario in order to explain the test steps or expected result using a Visio diagram as a reference. Provide the link or location to the actual path of the diagram or document.

Automation? (Yes/No): Whether this test case is automated or not. It is useful to track the automation status when test cases are automated.