In 1947, computers were big, room-sized machines operating on mechanical relays and glowing vacuum tubes. The state of the art at the time was the Mark II, a behemoth being built at Harvard University. Technicians were running the new computer through its paces when it suddenly stopped working. They scrambled to figure out why and discovered, stuck between a set of relay contacts deep in the bowels of the computer, a moth. It had apparently flown into the system, attracted by the light and heat, and was zapped by the high voltage when it landed on the relay.

The computer bug was born. Well, okay, it died, but you get the point.

Welcome to the first chapter of *Software Testing*. In this chapter, you'll learn about the history of software bugs and software testing.

Highlights of this chapter include

- How software bugs impact our lives
- What bugs are and why they occur
- Who software testers are and what they do

Infamous Software Error Case Studies

It's easy to take software for granted and not really appreciate how much it has infiltrated our daily lives. Back in 1947, the Mark II computer required legions of programmers to constantly maintain it. The average person never conceived of someday having his own computer in his home. Now there's free software CD-ROMs attached to cereal boxes and more software in our kids' video games than on the space shuttle. What once were techie gadgets, such as pagers and cell phones, have become commonplace. Most of us now can't go a day without logging on to the Internet and checking our email. We rely on overnight packages, long-distance phone service, and cutting-edge medical treatments.

Software is everywhere. However, it's written by people—so it's not perfect, as the following examples show.

Disney's *Lion King*, 1994–1995

In the fall of 1994, the Disney company released its first multimedia CD-ROM game for children, *The Lion King Animated Storybook*. Although many other companies had been marketing children's programs for years, this was Disney's first venture into the market and it was highly promoted and advertised. Sales were huge. It was "the game to buy" for children that holiday season. What happened, however, was a huge debacle. On December 26, the day after Christmas, Disney's customer support phones began to ring, and ring, and ring. Soon the phone support technicians were swamped with calls from angry parents with crying children who couldn't get the software to work. Numerous stories appeared in newspapers and on TV news.

It turns out that Disney failed to properly test the software on the many different PC models available on the market. The software worked on a few systems—likely the ones that the Disney programmers used to create the game—but not on the most common systems that the general public had.

Intel Pentium Floating-Point Division Bug, 1994

Enter the following equation into your PC's calculator:

```
(4195835 / 3145727) * 3145727 - 4195835
```

If the answer is zero, your computer is just fine. If you get anything else, you have an old Intel Pentium CPU with a floating-point division bug—a software bug burned into a computer chip and reproduced over and over in the manufacturing process.

On October 30, 1994, Dr. Thomas R. Nicely of Lynchburg (Virginia) College traced an unexpected result from one of his experiments to an incorrect answer by a division problem solved on his Pentium PC. He posted his find on the Internet and soon afterward a firestorm erupted as numerous other people duplicated his problem and found additional situations that resulted in wrong answers. Fortunately, these cases were rare and resulted in wrong answers only for extremely math-intensive, scientific, and engineering calculations. Most people would never encounter them doing their taxes or running their businesses.

What makes this story notable isn't the bug, but the way Intel handled the situation:

- Their software test engineers had found the problem while performing their own tests before the chip was released. Intel's management decided that the problem wasn't severe enough or likely enough to warrant fixing it, or even publicizing it.
- Once the bug was found, Intel attempted to diminish its perceived severity through press releases and public statements.
- When pressured, Intel offered to replace the faulty chips, but only if a user could prove that he was affected by the bug.

There was a public outcry. Internet newsgroups were jammed with irate customers demanding that Intel fix the problem. News stories painted the company as uncaring and incredulous. In the end, Intel apologized for the way it handled the bug and took a charge of over $400 million to cover the costs of replacing bad chips. Intel now reports known problems on its Web site and carefully monitors customer feedback on Internet newsgroups.

NOTE

On August 28th, 2000, shortly before this book went to press, Intel announced a recall of all the 1.13MHz Pentium III processors it had shipped after the chip had been in production for a month. A problem was discovered with the execution of certain instructions that could cause running applications to freeze. Computer manufacturers were creating plans for recalling the PCs already in customers' hands and calculating the costs of replacing the defective chips. As the baseball legend Yogi Berra once said, "This is like *déjà vu* all over again."

NASA Mars Polar Lander, 1999

On December 3, 1999, NASA's Mars Polar Lander disappeared during its landing attempt on the Martian surface. A Failure Review Board investigated the failure and determined that the most likely reason for the malfunction was the unexpected setting of a single data bit. Most alarming was why the problem wasn't caught by internal tests.

In theory, the plan for landing was this: As the lander fell to the surface, it was to deploy a parachute to slow its descent. A few seconds after the chute deployed, the probe's three legs were to snap open and latch into position for landing. When the probe was about 1,800 meters from the surface, it was to release the parachute and ignite its landing thrusters to gently lower it the remaining distance to the ground.

To save money, NASA simplified the mechanism for determining when to shut off the thrusters. In lieu of costly radar used on other spacecraft, they put an inexpensive contact switch on the leg's foot that set a bit in the computer commanding it to shut off the fuel. Simply, the engines would burn until the legs "touched down."

Unfortunately, the Failure Review Board discovered in their tests that in most cases when the legs snapped open for landing, a mechanical vibration also tripped the touch-down switch, setting the fatal bit. It's very probable that, thinking it had landed, the computer turned off the thrusters and the Mars Polar Lander smashed to pieces after falling 1,800 meters to the surface.

The result was catastrophic, but the reason behind it was simple. The lander was tested by multiple teams. One team tested the leg fold-down procedure and another the landing process from that point on. The first team never looked to see if the touch-down bit was set—it wasn't their area; the second team always reset the computer, clearing the bit, before it started its testing. Both pieces worked perfectly individually, but not when put together.

Patriot Missile Defense System, 1991

The U.S. Patriot missile defense system is a scaled-back version of the Strategic Defense Initiative ("Star Wars") program proposed by President Ronald Reagan. It was first put to use in the Gulf War as a defense for Iraqi Scud missiles. Although there were many news stories touting the success of the system, it did fail to defend against several missiles, including one that killed 28 U.S. soldiers in Dhahran, Saudi Arabia. Analysis found that a software bug was the problem. A small timing error in the system's clock accumulated to the point that after 14 hours, the tracking system was no longer accurate. In the Dhahran attack, the system had been operating for more than 100 hours.

The Y2K (Year 2000) Bug, circa 1974

Sometime in the early 1970s a computer programmer—let's suppose his name was Dave—was working on a payroll system for his company. The computer he was using had very little memory for storage, forcing him to conserve every last byte he could. Dave was proud that he could pack his programs more tightly than any of his peers. One method he used was to shorten dates from their 4-digit format, such as 1973, to a 2-digit format, such as 73. Because his payroll program relied heavily on date processing, Dave could save lots of expensive memory space. He briefly considered the problems that might occur when the current year hit 2000 and his program began doing computations on years such as 00 and 01. He knew there would be problems but decided that his program would surely be replaced or updated in 25 years and his immediate tasks were more important than planning for something that far out in time. After all, he had a deadline to meet. In 1995, Dave's program was still being used, Dave was retired, and no one knew how to get into the program to check if it was Y2K compliant, let alone how to fix it.

It's estimated that several hundred billion dollars were spent, worldwide, to replace or update computer programs such as Dave's, to fix potential Year 2000 failures.

What Is a Bug?

You've just read examples of what happens when software fails. It can be inconvenient, as when a computer game doesn't work properly, or it can be catastrophic, resulting in the loss of life. In these instances, it was obvious that the software didn't operate as intended. As a software tester you'll discover that most failures are hardly ever this obvious. Most are simple, subtle failures, with many being so small that it's not always clear which ones are true failures, and which ones aren't.

Terms for Software Failures

Depending on where you're employed as a software tester, you will use different terms to describe what happens when software fails. Here are a few:

Defect	Variance
Fault	Failure
Problem	Inconsistency
Error	Feature
Incident	Bug
Anomaly	

(There's also a list of unmentionable terms, but they're most often used privately among programmers.)

You might be amazed that so many names could be used to describe a software failure. Why so many? It's all really based on the company's culture and the process the company uses to develop its software. If you look up these words in the dictionary, you'll find that they all have slightly different meanings. They also have inferred meanings by how they're used in day-to-day conversation.

For example, *fault*, *failure*, and *defect* tend to imply a condition that's really severe, maybe even dangerous. It doesn't sound right to call an incorrectly colored icon a *fault*. These words also tend to imply blame: "It's his fault that the software failed."

Anomaly, *incident*, and *variance* don't sound quite so negative and infer more unintended operation than an all-out failure. "The president stated that it was a software anomaly that caused the missile to go off course."

Problem, *error*, and *bug* are probably the most generic terms used.

Just Call It What It Is and Get On with It

It's interesting that some companies and product teams will spend hours and hours of precious development time arguing and debating which term to use. A well-known computer company spent weeks in discussion with its engineers before deciding to rename Product Anomaly Reports (PARs) to Product Incident Reports (PIRs). Countless dollars were spent in the process of deciding which term was better. Once the decision was made, all the paperwork, software, forms, and so on had to be updated to reflect the new term. It's unknown if it made any difference to the programmer's or tester's productivity.

So, why bring this topic up? It's important as a software tester to understand the personality behind the product development team you're working with. How they refer to their software problems is a tell-tale sign of how they approach their overall development process. Are they cautious, careful, direct, or just plain blunt?

In this book, all software problems will be called *bugs*. It doesn't matter if it's big, small, intended, unintended, or someone's feelings will be hurt because they create one. There's no reason to dice words. A bug's a bug's a bug.

Software Bug: A Formal Definition

Calling any and all software problems *bugs* may sound simple enough, but doing so hasn't really addressed the issue. Now the word *problem* needs to be defined. To keep from running in circular definitions, there needs to be a definitive description of what a bug is.

First, you need a supporting term: *product specification*. A product specification, sometimes referred to as simply a *spec* or *product spec*, is an agreement among the software development team. It defines the product they are creating, detailing what it will be, how it will act, what it will do, and what it won't do. This agreement can range in form from a simple verbal understanding to a formalized written document. In Chapter 2, "The Software Development Process," you will learn more about software specifications and the development process, but for now, this definition is sufficient.

For the purposes of this book and much of the software industry, a *software bug* occurs when one or more of the following five rules is true:

1. The software doesn't do something that the product specification says it should do.
2. The software does something that the product specification says it shouldn't do.
3. The software does something that the product specification doesn't mention.
4. The software doesn't do something that the product specification doesn't mention but should.
5. The software is difficult to understand, hard to use, slow, or—in the software tester's eyes—will be viewed by the end user as just plain not right.

To better understand each rule, try the following example of applying them to a calculator.

The specification for a calculator probably states that it will perform correct addition, subtraction, multiplication, and division. If you, as the tester, receive the calculator, press the + key, and nothing happens, that's a bug because of Rule #1. If you get the wrong answer, that's also a bug because of Rule #1.

The product spec might state that the calculator should never crash, lock up, or freeze. If you pound on the keys and get the calculator to stop responding to your input, that's a bug because of Rule #2.

Suppose that you receive the calculator for testing and find that besides addition, subtraction, multiplication, and division, it also performs square roots. Nowhere was this ever specified. An ambitious programmer just threw it in because he felt it would be a great feature. This isn't a feature—it's really a bug because of Rule #3.

The fourth rule may read a bit strange with its double negatives, but its purpose is to catch things that were forgotten in the specification. You start testing the calculator and discover when the battery gets weak that you no longer receive correct answers to your calculations. No one ever considered how the calculator should react in this mode. A bad assumption was made that the batteries would always be fully charged. You expected it to keep working until the batteries were completely dead, or at least notify you in some way that they were weak. Correct calculations didn't happen with weak batteries and it wasn't specified what should happen. Rule #4 makes this a bug.

Rule #5 is the catch-all. As a tester you are the first person to really use the software. If you weren't there, it would be the customer using the product for the first time. If you find something that you don't feel is right, for whatever reason, it's a bug. In the case of the calculator, maybe you found that the buttons were too small. Maybe the placement of the = key made it hard to use. Maybe the display was difficult to read under bright lights. All of these are bugs because of Rule #5.

NOTE

Every person who uses a piece of software will have different expectations and opinions as to how it should work. It would be impossible to write software that every user thought was perfect. As a software tester, you should keep this in mind when you apply Rule #5 to your testing. Be thorough, use your best judgment, and be reasonable.

These are greatly simplified examples, so think about how the rules apply to software that you use every day. What is expected, what is unexpected? What do you think was specified and what was forgotten? And, what do you just plain dislike about the software?

This definition of a bug covers a lot of ground but it assures that all problems are identified.

Why Do Bugs Occur?

Now that you know what bugs are, you might be wondering why they occur. What you'll be surprised to find out is that most of them aren't caused by programming errors. Numerous studies have been performed on very small to extremely large projects and the results are always the same. The number one cause of software bugs is the specification (see Figure 1.1).

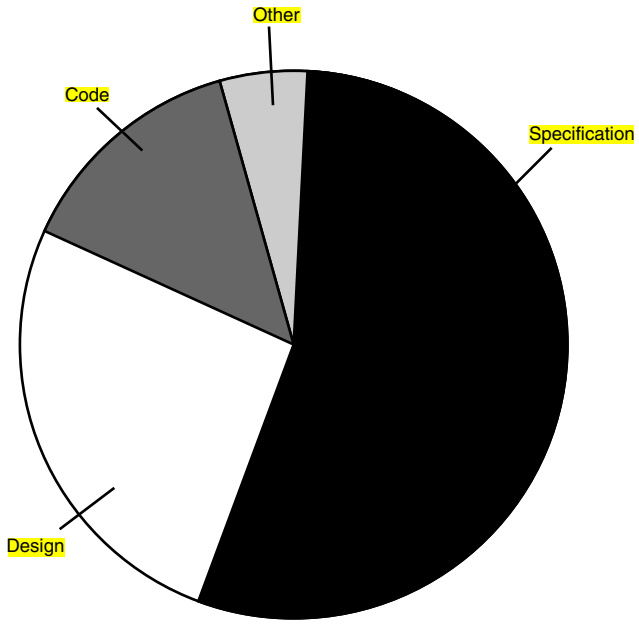


FIGURE 1.1
Bugs are caused for numerous reasons, but the main cause can be traced to the specification.

There are several reasons specifications are the largest bug producer. In many instances a spec simply isn't written. Other reasons may be that the spec isn't thorough enough, it's constantly changing, or it's not communicated well to the entire development team. Planning software is vitally important. If it's not done correctly, bugs will be created.

The next largest source of bugs is the design. This is where the programmers lay out their plan for the software. Compare it to an architect creating the blueprints for a building. Bugs occur here for the same reason they occur in the specification. It's rushed, changed, or not well communicated.

NOTE

There's an old saying, "If you can't say it, you can't do it." This applies perfectly to software development and testing.

Coding errors may be more familiar to you if you're a programmer. Typically, these can be traced to the software's complexity, poor documentation (especially in code that's being

updated or revised), schedule pressure, or just plain dumb mistakes. It's important to note that many bugs that appear on the surface to be programming errors can really be traced to specification and design errors. It's quite common to hear a programmer say, "Oh, so that's what it's supposed to do. If somebody had just told me that I wouldn't have written the code that way."

The other category is the catch-all for what's left. Some bugs can be blamed on false positives, conditions that were thought to be bugs but really weren't. There may be duplicate bugs, multiple ones that resulted from the same root cause. Some bugs can also be traced to testing errors. In the end, these bugs usually make up such a small percentage of all bugs found that they aren't worth worrying about.

The Cost of Bugs

As you will learn in Chapter 2, software doesn't just magically appear—there's usually a planned, methodical development process used to create it. From its inception, through the planning, programming, and testing, to its use by the public, there's the potential for bugs to be found. Figure 1.2 shows how the cost of fixing these bugs grows over time.

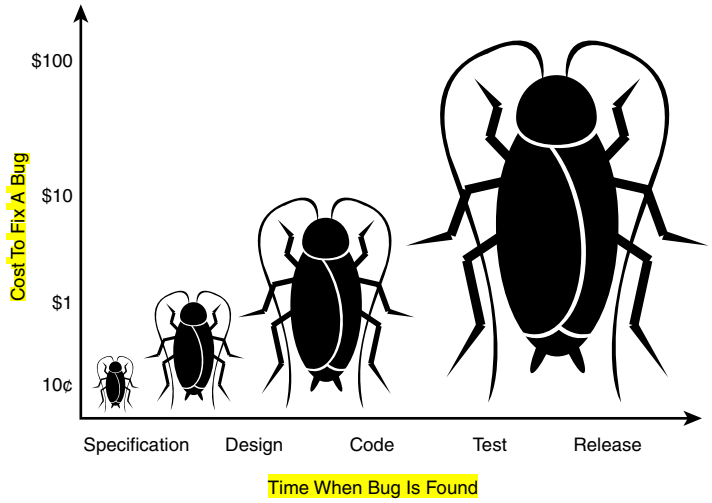


FIGURE 1.2
The cost to fix bugs increases dramatically over time.

The costs are logarithmic—that is, they increase tenfold as time increases. A bug found and fixed during the early stages when the specification is being written might cost next to nothing, or 10 cents in our example. The same bug, if not found until the software is coded and tested, might cost $1 to $10. If a customer finds it, the cost could easily top $100.

As an example of how this works, consider the Disney *Lion King* case discussed earlier. The root cause of the problem was that the software wouldn't work on a very popular PC platform. If, in the early specification stage, someone had researched what PCs were popular and specified that the software needed to be designed and tested to work on those configurations, the cost of that effort would have been almost nothing. If that didn't occur, a backup would have been for the software testers to collect samples of the popular PCs and verify the software on them. They would have found the bug, but it would have been more expensive to fix because the software would have to be debugged, fixed, and retested. The development team could have also sent out a preliminary version of the software to a small group of customers in what's called a *beta test*. Those customers, chosen to represent the larger market, would have likely discovered the problem. As it turned out, however, the bug was completely missed until many thousands of CD-ROMs were created and purchased. Disney ended up paying for telephone customer support, product returns, replacement CD-ROMs, as well as another debug, fix, and test cycle. It's very easy to burn up your entire product's profit if serious bugs make it to the customer.

What Exactly Does a Software Tester Do?

You've now seen examples of really nasty bugs, you know what the definition of a bug is, and you know how costly they can be. By now it should be pretty evident what a tester's goal is:

> *The goal of a software tester is to find bugs.*

You may run across product teams who want their testers to simply confirm that the software works, not to find bugs. Reread the case study about the Mars Polar Lander, and you'll see why this is the wrong approach. If you're only testing things that should work and setting up your tests so they'll pass, you will miss the things that don't work. You will miss the bugs.

If you're missing bugs, you're costing your project and your company money. As a software tester you shouldn't be content at just finding bugs—you should think about how to find them sooner in the development process, thus making them cheaper to fix.

> *The goal of a software tester is to find bugs, and find them as early as possible.*

But, finding bugs, even finding them early, isn't enough. Remember the definition of a bug. You, the software tester, are the customer's eyes, the first one to see the software. You speak for the customer and must demand perfection.

> *The goal of a software tester is to find bugs, find them as early as possible, and make sure they get fixed.*

This final definition is very important. Commit it to memory and refer back to it as you learn the testing techniques discussed throughout the rest of this book.

What Makes a Good Software Tester?

In the movie *Star Trek II: The Wrath of Khan*, Spock says, "As a matter of cosmic history, it has always been easier to destroy than to create." At first glance, it may appear that a software tester's job would be easier than a programmer's. Breaking code and finding bugs must surely be easier than writing the code in the first place. Surprisingly, it's not. The methodical and disciplined approach to software testing that you'll learn in this book requires the same hard work and dedication that programming does. It involves very similar skills, and although a software tester doesn't necessarily need to be a full-fledged programmer, having that knowledge is a great benefit.

Today, most mature companies treat software testing as a technical engineering profession. They recognize that having trained software testers on their project teams and allowing them to apply their trade early in the development process allows them to build better quality software.

Here's a list of traits that most software testers should have:

- **They are explorers.** Software testers aren't afraid to venture into unknown situations. They love to get a new piece of software, install it on their PC, and see what happens.
- **They are troubleshooters.** Software testers are good at figuring out why something doesn't work. They love puzzles.
- **They are relentless.** Software testers keep trying. They may see a bug that quickly vanishes or is difficult to re-create. Rather than dismiss it as a fluke, they will try every way possible to find it.
- **They are creative.** Testing the obvious isn't sufficient for software testers. Their job is to think up creative and even off-the-wall approaches to find bugs.
- **They are (mellowed) perfectionists.** They strive for perfection, but they know when it becomes unattainable and they're OK with getting as close as they can.
- **They exercise good judgment.** Software testers need to make decisions about what they will test, how long it will take, and if the problem they're looking at is really a bug.
- **They are tactful and diplomatic.** Software testers are always the bearers of bad news. They have to tell the programmers that their baby is ugly. Good software testers know how to do so tactfully and professionally and know how to work with programmers who aren't always tactful and diplomatic.
- **They are persuasive.** Bugs that testers find won't always be viewed as severe enough to be fixed. Testers need to be good at making their points clear, demonstrating why the bug does indeed need to be fixed, and following through on making it happen.

Software Testing Is Fun!

A fundamental trait of software testers is that they simply like to break things. They live to find those elusive system crashes. They take great satisfaction in laying to waste the most complex programs. They're often seen jumping up and down in glee, giving each other high-fives, and doing a little dance when they bring a system to its knees. It's the simple joys of life that matter the most.

In addition to these traits, having some education in software programming is a big plus. As you'll see in Chapter 6, "Examining the Code," knowing how software is written can give you a different view of where bugs are found, thus making you a more efficient and effective tester. It can also help you develop the testing tools discussed in Chapter 14, "Automated Testing and Test Tools."

Lastly, if you're an expert in some non-computer field, your knowledge can be invaluable to a software team creating a new product. Software is being written to do just about everything today. Your knowledge of teaching, cooking, airplanes, carpentry, medicine, or whatever would be a tremendous help finding bugs in software for those areas.

Summary

Software testing is a critical job. With the size and complexity of today's software, it's imperative that software testing be performed professionally and effectively. Too much is at risk. We don't need more defective computer chips or lost Mars landers.

In the following chapters of Part I, you'll learn more about the big picture of software development and how software testing fits in. This knowledge is critical to helping you apply the specific test techniques covered in the remainder of this book.

Quiz

These quiz questions are provided for your further understanding. See Appendix A, "Answers to Quiz Questions," for the answers—but don't peek!

1. In the Year 2000 bug example, did Dave do anything wrong?
2. **True or False:** It's important what term your company or team calls a problem in its software.
3. What's wrong with just testing that a program works as expected?
4. How much more does it cost to fix a bug found after the product is released than it does from the very start of the project?

You're now down to the final chapter of software testing. Well, okay, maybe the final chapter of the book *Software Testing*, but definitely not of the job. Your work in that area has only just begun.

You probably began reading this book with little knowledge of what software testing is all about. You've likely experienced the minor little annoyances and the occasional crashes of the software you use on your computer at home or at work. You've seen and heard news stories of major software bugs and you know about the infamous Y2K bug, which, after all the last-minute testing and preparation, didn't bite as hard as expected.

Hopefully you've now been enlightened and understand why these bugs can still happen despite the best efforts of the people behind the software. You've learned about the test planning process, where to look for bugs, and how to report them. You now understand the difficult decision-making process that's involved in determining which bugs to fix and which ones to defer, and you've seen the graphs that show a product that's ready to release and one that still has a long way to go.

Above all else, you should now understand that software testing is a complex and difficult job. To be successful at it requires discipline, training, and experience. Simply sitting down, pounding the keys, and shouting over the wall to the programmer when you see something odd won't cut it. Software is too important. Businesses have failed, careers have been ruined, and people have died because of software bugs. Your job as a software tester is to find those bugs, efficiently and professionally, before they make it out the door.

This final chapter will give you pointers to more information about software testing, explain a few of the possible career options, and leave you with an important message about software quality. Highlights of this chapter include

- The career path options available for software testers
- Where to look for a testing job
- How to get more hands-on experience at finding bugs
- Where to learn more about software testing
- The Computer User's Bill of Rights

Your Job as a Software Tester

One serious misconception about software testing is that it's only an entry-level position in the software industry. This erroneous belief persists because of the ignorance of what software testing is and what it involves—mainly due to the number of companies still developing software without any real process. They don't yet know that they need software testers of all skill levels to create great software. But, as more emphasis is put on creating software of higher and higher quality, the value of software testing as a career is becoming understood.

Because of this increased awareness, the opportunities are there for the taking. Software testers with just a couple years of experience are highly sought after. Testers who can also program and perform white-box testing or develop automated tests are even more in demand. And, if you've been through a few product development cycles and can lead a small team of other testers, you're in a highly marketable position. It's truly a job-hunter's market for software testers.

Here's a breakout of various software testing positions and their descriptions. Keep in mind, as you learned in Chapter 20, "Software Quality Assurance," the names vary and may not mean exactly what the job really is, but ultimately most software testing jobs fall into these categories.

- **Software test technician.** This is often a true entry-level test position. You would be responsible for setting up test hardware and software configurations, running simple test scripts or test automation, and possibly working with beta sites to isolate and reproduce bugs. Some work can become mundane and repetitive, but being a test technician is a good way to become introduced to software testing.
- **Software tester or software test engineer.** Most companies have several levels of software testers based on experience and expertise. An entry-level tester may perform the duties of technician, working their way up to running more advanced and complex tests. As you progress, you'll write your own test cases and test procedures and might attend design and specification reviews. You'll perform testing and isolate, reproduce, and report the bugs you find. If you have programming abilities, you'll write test automation or testing tools and work closely with the programmers as you perform white-box testing.
- **Software test lead.** A test lead is responsible for the testing of a major portion of a software project or sometimes an entire small project. They often generate the test plan for their areas and oversee the testing performed by other testers. They're frequently involved in collecting metrics for their products and presenting them to management. They usually also perform the duties of a software tester.
- **Software test manager.** A test manager oversees the testing for an entire project or multiple projects. The test leads report to them. They work with the project managers and development managers to set schedules, priorities, and goals. They're responsible for providing the appropriate testing resources—people, equipment, space, and so on—for their projects. They set the tone and strategy for the testing their teams perform.

Finding a Software Testing Position

So where do you look for a software testing job? The answer is the same places you would look for a programming job—with any business or company that develops software.

- **Use the Internet.** A quick search done using several job search engines just before this book went to print found more than 1,000 open software testing positions at companies all around the country. Many of these positions were for entry-level testers. There were

1.1 Why is Testing Necessary (K2) — *20 minutes*

Terms
Bug, defect, error, failure, fault, mistake, quality, risk

1.1.1 Software Systems Context (K1)

Software systems are an integral part of life, from business applications (e.g., banking) to consumer products (e.g., cars). Most people have had an experience with software that did not work as expected. Software that does not work correctly can lead to many problems, including loss of money, time or business reputation, and could even cause injury or death.

1.1.2 Causes of Software Defects (K2)

A human being can make an error (mistake), which produces a defect (fault, bug) in the program code, or in a document. If a defect in code is executed, the system may fail to do what it should do (or do something it shouldn't), causing a failure. Defects in software, systems or documents may result in failures, but not all defects do so.

Defects occur because human beings are fallible and because there is time pressure, complex code, complexity of infrastructure, changing technologies, and/or many system interactions.

Failures can be caused by environmental conditions as well. For example, radiation, magnetism, electronic fields, and pollution can cause faults in firmware or influence the execution of software by changing the hardware conditions.

1.1.3 Role of Testing in Software Development, Maintenance and Operations (K2)

Rigorous testing of systems and documentation can help to reduce the risk of problems occurring during operation and contribute to the quality of the software system, if the defects found are corrected before the system is released for operational use.

Software testing may also be required to meet contractual or legal requirements, or industry-specific standards.

1.1.4 Testing and Quality (K2)

With the help of testing, it is possible to measure the quality of software in terms of defects found, for both functional and non-functional software requirements and characteristics (e.g., reliability, usability, efficiency, maintainability and portability). For more information on non-functional testing see Chapter 2; for more information on software characteristics see 'Software Engineering – Software Product Quality' (ISO 9126).

Testing can give confidence in the quality of the software if it finds few or no defects. A properly designed test that passes reduces the overall level of risk in a system. When testing does find defects, the quality of the software system increases when those defects are fixed.

Lessons should be learned from previous projects. By understanding the root causes of defects found in other projects, processes can be improved, which in turn should prevent those defects from reoccurring and, as a consequence, improve the quality of future systems. This is an aspect of quality assurance.

Testing should be integrated as one of the quality assurance activities (i.e., alongside development standards, training and defect analysis).

1.1.5 How Much Testing is Enough? (K2)

Deciding how much testing is enough should take account of the level of risk, including technical, safety, and business risks, and project constraints such as time and budget. Risk is discussed further in Chapter 5.

Testing should provide sufficient information to stakeholders to make informed decisions about the release of the software or system being tested, for the next development step or handover to customers.

1.2 What is Testing? (K2) — *30 minutes*

Terms
Debugging, requirement, review, test case, testing, test objective

Background
A common perception of testing is that it only consists of running tests, i.e., executing the software. This is part of testing, but not all of the testing activities.

Test activities exist before and after test execution. These activities include planning and control, choosing test conditions, designing and executing test cases, checking results, evaluating exit criteria, reporting on the testing process and system under test, and finalizing or completing closure activities after a test phase has been completed. Testing also includes reviewing documents (including source code) and conducting static analysis.

Both dynamic testing and static testing can be used as a means for achieving similar objectives, and will provide information that can be used to improve both the system being tested and the development and testing processes.

Testing can have the following objectives:
- Finding defects
- Gaining confidence about the level of quality
- Providing information for decision-making
- Preventing defects

The thought process and activities involved in designing tests early in the life cycle (verifying the test basis via test design) can help to prevent defects from being introduced into code. Reviews of documents (e.g., requirements) and the identification and resolution of issues also help to prevent defects appearing in the code.

Different viewpoints in testing take different objectives into account. For example, in development testing (e.g., 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. In acceptance testing, the main objective may be to confirm that the system works as expected, to gain confidence that it has met the requirements. In some cases the main objective of testing may be to assess the quality of the software (with no intention of fixing defects), to give information to stakeholders of the risk of releasing the system at a given time. Maintenance testing often includes testing that no new defects have been introduced during development of the changes. During operational testing, the main objective may be to assess system characteristics such as reliability or availability.

Debugging and testing are different. Dynamic testing can show failures that are caused by defects. Debugging is the development activity that finds, analyzes and removes the cause of the failure. Subsequent re-testing by a tester ensures that the fix does indeed resolve the failure. The responsibility for these activities is usually testers test and developers debug.

The process of testing and the testing activities are explained in Section 1.4.

1.3 Seven Testing Principles (K2) — *35 minutes*

Terms
Exhaustive testing

Principles
A number of testing principles have been suggested over the past 40 years and offer general guidelines common for all testing.

Principle 1 – 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.

Principle 2 – Exhaustive testing is impossible
Testing everything (all combinations of inputs and preconditions) is not feasible except for trivial cases. Instead of exhaustive testing, risk analysis and priorities should be used to focus testing efforts.

Principle 3 – Early testing
To find defects early, testing activities shall be started as early as possible in the software or system development life cycle, and shall be focused on defined objectives.

Principle 4 – Defect clustering
Testing effort shall be focused proportionally to the expected and later observed defect density of modules. A small number of modules usually contains most of the defects discovered during pre-release testing, or is responsible for most of the operational failures.

Principle 5 – 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 defects. To overcome this “pesticide paradox”, 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 find potentially more defects.

Principle 6 – Testing is context dependent
Testing is done differently in different contexts. For example, safety-critical software is tested differently from an e-commerce site.

Principle 7 – 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.

1.4 Fundamental Test Process (K1) — *35 minutes*

Terms
Confirmation testing, re-testing, exit criteria, incident, regression testing, test basis, test condition, test coverage, test data, test execution, test log, test plan, test procedure, test policy, test suite, test summary report, testware

Background
The most visible part of testing is test execution. But to be effective and efficient, test plans should also include time to be spent on planning the tests, designing test cases, preparing for execution and evaluating results.

The fundamental test process consists of the following main activities:
- Test planning and control
- Test analysis and design
- Test implementation and execution
- Evaluating exit criteria and reporting
- Test closure activities

Although logically sequential, the activities in the process may overlap or take place concurrently. Tailoring these main activities within the context of the system and the project is usually required.

1.4.1 Test Planning and Control (K1)
Test planning is the activity of defining the objectives of testing and the specification of test activities in order to meet the objectives and mission.

Test control is the ongoing activity of comparing actual progress against the plan, and reporting the status, including deviations from the plan. It involves taking actions necessary to meet the mission and objectives of the project. In order to control testing, the testing activities should be monitored throughout the project. Test planning takes into account the feedback from monitoring and control activities.

Test planning and control tasks are defined in Chapter 5 of this syllabus.

1.4.2 Test Analysis and Design (K1)
Test analysis and design is the activity during which general testing objectives are transformed into tangible test conditions and test cases.

The test analysis and design activity has the following major tasks:
- Reviewing the test basis (such as requirements, software integrity level[1] (risk level), risk analysis reports, architecture, design, interface specifications)
- Evaluating testability of the test basis and test objects
- Identifying and prioritizing test conditions based on analysis of test items, the specification, behavior and structure of the software
- Designing and prioritizing high level test cases
- Identifying necessary test data to support the test conditions and test cases
- Designing the test environment setup and identifying any required infrastructure and tools
- Creating bi-directional traceability between test basis and test cases

[1] The degree to which software complies or must comply with a set of stakeholder-selected software and/or software-based system characteristics (e.g., software complexity, risk assessment, safety level, security level, desired performance, reliability, or cost) which are defined to reflect the importance of the software to its stakeholders.

1.4.3 Test Implementation and Execution (K1)

Test implementation and execution is the activity where test procedures or scripts are specified by combining the test cases in a particular order and including any other information needed for test execution, the environment is set up and the tests are run.

Test implementation and execution has the following major tasks:
- Finalizing, implementing and prioritizing test cases (including the identification of test data)
- Developing and prioritizing test procedures, creating test data and, optionally, preparing test harnesses and writing automated test scripts
- Creating test suites from the test procedures for efficient test execution
- Verifying that the test environment has been set up correctly
- Verifying and updating bi-directional traceability between the test basis and test cases
- Executing test procedures either manually or by using test execution tools, according to the planned sequence
- Logging the outcome of test execution and recording the identities and versions of the software under test, test tools and testware
- Comparing actual results with expected results
- Reporting discrepancies as incidents and analyzing them in order to establish their cause (e.g., a defect in the code, in specified test data, in the test document, or a mistake in the way the test was executed)
- Repeating test activities as a result of action taken for each discrepancy, for example, re-execution of a test that previously failed in order to confirm a fix (confirmation testing), execution of a corrected test and/or execution of tests in order to ensure that defects have not been introduced in unchanged areas of the software or that defect fixing did not uncover other defects (regression testing)

1.4.4 Evaluating Exit Criteria and Reporting (K1)

Evaluating exit criteria is the activity where test execution is assessed against the defined objectives. This should be done for each test level (see Section 2.2).

Evaluating exit criteria has the following major tasks:
- Checking test logs against the exit criteria specified in test planning
- Assessing if more tests are needed or if the exit criteria specified should be changed
- Writing a test summary report for stakeholders

1.4.5 Test Closure Activities (K1)

Test closure activities collect data from completed test activities to consolidate experience, testware, facts and numbers. Test closure activities occur at project milestones such as when a software system is released, a test project is completed (or cancelled), a milestone has been achieved, or a maintenance release has been completed.

Test closure activities include the following major tasks:
- Checking which planned deliverables have been delivered
- Closing incident reports or raising change records for any that remain open
- Documenting the acceptance of the system
- Finalizing and archiving testware, the test environment and the test infrastructure for later reuse
- Handing over the testware to the maintenance organization
- Analyzing lessons learned to determine changes needed for future releases and projects
- Using the information gathered to improve test maturity

1.5 The Psychology of Testing (K2) — *25 minutes*

Terms
Error guessing, independence

Background
The mindset to be used while testing and reviewing is different from that used while developing software. With the right mindset developers are able to test their own code, but separation of this responsibility to a tester is typically done to help focus effort and provide additional benefits, such as an independent view by trained and professional testing resources. Independent testing may be carried out at any level of testing.

A certain degree of independence (avoiding the author bias) often makes the tester more effective at finding defects and failures. Independence is not, however, a replacement for familiarity, and developers can efficiently find many defects in their own code. Several levels of independence can be defined as shown here from low to high:
- Tests designed by the person(s) who wrote the software under test (low level of independence)
- Tests designed by another person(s) (e.g., from the development team)
- Tests designed by a person(s) from a different organizational group (e.g., an independent test team) or test specialists (e.g., usability or performance test specialists)
- Tests designed by a person(s) from a different organization or company (i.e., outsourcing or certification by an external body)

People and projects are driven by objectives. People tend to align their plans with the objectives set by management and other stakeholders, for example, to find defects or to confirm that software meets its objectives. Therefore, it is important to clearly state the objectives of testing.

Identifying failures during testing may be perceived as criticism against the product and against the author. As a result, testing is often seen as a destructive activity, even though it is very constructive in the management of product risks. Looking for failures in a system requires curiosity, professional pessimism, a critical eye, attention to detail, good communication with development peers, and experience on which to base error guessing.

If errors, defects or failures are communicated in a constructive way, bad feelings between the testers and the analysts, designers and developers can be avoided. This applies to defects found during reviews as well as in testing.

The tester and test leader need good interpersonal skills to communicate factual information about defects, progress and risks in a constructive way. For the author of the software or document, defect information can help them improve their skills. Defects found and fixed during testing will save time and money later, and reduce risks.

Communication problems may occur, particularly if testers are seen only as messengers of unwanted news about defects. However, there are several ways to improve communication and relationships between testers and others:

- Start with collaboration rather than battles – remind everyone of the common goal of better quality systems
- Communicate findings on the product in a neutral, fact-focused way without criticizing the person who created it, for example, write objective and factual incident reports and review findings
- Try to understand how the other person feels and why they react as they do
- Confirm that the other person has understood what you have said and vice versa

1.6 Code of Ethics — *10 minutes*

Involvement in software testing enables individuals to learn confidential and privileged information. A code of ethics is necessary, among other reasons to ensure that the information is not put to inappropriate use. Recognizing the ACM and IEEE code of ethics for engineers, the ISTQB states the following code of ethics:

PUBLIC - Certified software testers shall act consistently with the public interest

CLIENT AND EMPLOYER - Certified software testers shall act in a manner that is in the best interests of their client and employer, consistent with the public interest

PRODUCT - Certified software testers shall ensure that the deliverables they provide (on the products and systems they test) meet the highest professional standards possible

JUDGMENT- Certified software testers shall maintain integrity and independence in their professional judgment

MANAGEMENT - Certified software test managers and leaders shall subscribe to and promote an ethical approach to the management of software testing

PROFESSION - Certified software testers shall advance the integrity and reputation of the profession consistent with the public interest

COLLEAGUES - Certified software testers shall be fair to and supportive of their colleagues, and promote cooperation with software developers

SELF - Certified software testers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession

References

1.1.5 Black, 2001, Kaner, 2002
1.2 Beizer, 1990, Black, 2001, Myers, 1979
1.3 Beizer, 1990, Hetzel, 1988, Myers, 1979
1.4 Hetzel, 1988
1.4.5 Black, 2001, Craig, 2002
1.5 Black, 2001, Hetzel, 1988

Chapter 3 : Verification and Validation

Prepared by Roshan Chitrakar

The terms '**Verification**' and '**Validation**' are frequently used in the software testing world but the meaning of these terms are mostly vague and debatable. Verification and validation are often used interchangeably but have different definitions. Generally speaking, Verification is the process confirming that something—software—meets its specification; and Validation is the process confirming that it meets the user's requirements.

Some distinctions between the two are given below: -

Criteria	Verification	Validation
Definition	The process of evaluating work-products (not the actual final product) of a development phase to determine whether they meet the specified requirements for that phase.	The process of evaluating software during or at the end of the development process to determine whether it satisfies specified business requirements.
Objective	To ensure that the product is being built according to the requirements and design specifications. In other words, to ensure that work products meet their specified requirements.	To ensure that the product actually meets the user's needs, and that the specifications were correct in the first place. In other words, to demonstrate that the product fulfills its intended use when placed in its intended environment.
Question	Are we building the product *right*?	Are we building the *right* product?
Evaluation Items	Plans, Requirement Specs, Design Specs, Code, Test Cases	The actual product/software.
Activities	• Reviews • Walkthroughs • Inspections	• Testing

Software Walkthrough

The software walkthrough is organized to serve the needs of the producer or author of the software artifact in acquiring superior knowledge of all aspects of the software artifact. It is a learning experience. A desirable side effect of the software walkthrough is the forging of a shared vision among the reviewers and consensus among participants on the approaches taken, product and engineering practices applied, completeness and correctness of capabilities and features, and rules of construction for the domain product. Since the software walkthrough caters to the needs of the author, it is the author who initiates the session. Consequently, there may be several walkthroughs in each life-cycle activity. Software walkthroughs yield open issues and action items. While these issues and action items may be tracked to closure, the only measurement taken is a count of the software walkthroughs held.

Software Inspection

The software inspection is structured to serve the needs of quality management in verifying that the software artifact complies with the standard of excellence for software engineering artifacts. The focus is one of verification, on doing the job right. The software inspection is a formal review held at the

conclusion of a life-cycle activity and serves as a quality gate with an exit criteria for moving on to subsequent activities.

The software inspection utilizes a structured review process of planning, preparation, entry criteria, conduct, exit criteria, report out, and follow-up. It ensures that a close and strict examination of the product artifact is conducted according to the standard of excellence criteria, which spans completeness, correctness, style, rules of construction, and multiple views and may also include technology and metrics. This close and strict examination results in the early detection of defects. The software inspection is led by a moderator and assisted by other role players including recorder, reviewer, reader, and producer. The software inspection is initiated as an exit criteria for each activity in the life cycle. Product and process measurements are recorded during the software inspection session and recorded on specially formatted forms and reports. These issues and defects are tracked to closure.

Reviews

The activities of the structured review process are organized for software inspections. Software walkthroughs may employ variations for planning, conduct, and follow-up.

Steps in Review Process are : -

1. Planning
2. Preparation
3. Conduct
4. Reporting
5. Follow-Up

Elements of Review and Walkthrough (Inspection)

Elements	*Software Inspection*	*Software Walkthrough*
Structured review process	Planning	Planning: optional
	Preparation	
	Conduct	Conduct
	Report out	
	Follow up	Follow up
Standard of excellence	Completeness	Completeness
	Correctness	Correctness
	Style	
	Rules of construction	Rules of construction
	Multiple views	
		Product and engineering practice
Defined roles of participants	Moderator	Moderator: optional
	Recorder	
	Producer	Producer
	Reviewer	Reviewer
	Reader	
Forms and reports	Inspection record	
	Inspection reporting form	
	Report summary form	
		Open issues
		Action items

Sources:

- Don O'Neill, Inspection as an Up-Front Quality Technique, *Handbook of Software Quality Assurance*, Fourth Edition
- Verification vs Validation *downloaded from http://softwaretestingfundamentals.com/verification-vs-validation/*

2.1 Software Development Models (K2) — *20 minutes*

Terms
Commercial Off-The-Shelf (COTS), iterative-incremental development model, validation, verification, V-model

Background
Testing does not exist in isolation; test activities are related to software development activities. Different development life cycle models need different approaches to testing.

2.1.1 V-model (Sequential Development Model) (K2)
Although variants of the V-model exist, a common type of V-model uses four test levels, corresponding to the four development levels.

The four levels used in this syllabus are:
- Component (unit) testing
- Integration testing
- System testing
- Acceptance testing

In practice, a V-model may have more, fewer or different levels of development and testing, depending on the project and the software product. For example, there may be component integration testing after component testing, and system integration testing after system testing.

Software work products (such as business scenarios or use cases, requirements specifications, design documents and code) produced during development are often the basis of testing in one or more test levels. References for generic work products include Capability Maturity Model Integration (CMMI) or 'Software life cycle processes' (IEEE/IEC 12207). Verification and validation (and early test design) can be carried out during the development of the software work products.

2.1.2 Iterative-incremental Development Models (K2)
Iterative-incremental development is the process of establishing requirements, designing, building and testing a system in a series of short development cycles. Examples are: prototyping, Rapid Application Development (RAD), Rational Unified Process (RUP) and agile development models. A system that is produced using these models may be tested at several test levels during each iteration. An increment, added to others developed previously, forms a growing partial system, which should also be tested. Regression testing is increasingly important on all iterations after the first one. Verification and validation can be carried out on each increment.

2.1.3 Testing within a Life Cycle Model (K2)
In any life cycle model, there are several characteristics of good testing:
- For every development activity there is a corresponding testing activity
- Each test level has test objectives specific to that level
- The analysis and design of tests for a given test level should begin during the corresponding development activity
- Testers should be involved in reviewing documents as soon as drafts are available in the development life cycle

Test levels can be combined or reorganized depending on the nature of the project or the system architecture. For example, for the integration of a Commercial Off-The-Shelf (COTS) software product into a system, the purchaser may perform integration testing at the system level (e.g.,

integration to the infrastructure and other systems, or system deployment) and acceptance testing (functional and/or non-functional, and user and/or operational testing).

2.2 Test Levels (K2) — *40 minutes*

Terms
Alpha testing, beta testing, component testing, driver, field testing, functional requirement, integration, integration testing, non-functional requirement, robustness testing, stub, system testing, test environment, test level, test-driven development, user acceptance testing

Background
For each of the test levels, the following can be identified: the generic objectives, the work product(s) being referenced for deriving test cases (i.e., the test basis), the test object (i.e., what is being tested), typical defects and failures to be found, test harness requirements and tool support, and specific approaches and responsibilities.

Testing a system's configuration data shall be considered during test planning,

2.2.1 Component Testing (K2)
Test basis:
- Component requirements
- Detailed design
- Code

Typical test objects:
- Components
- Programs
- Data conversion / migration programs
- Database modules

Component testing (also known as unit, module or program testing) searches for defects in, and verifies the functioning of, software modules, programs, objects, classes, etc., that are separately testable. It may be done in isolation from the rest of the system, depending on the context of the development life cycle and the system. Stubs, drivers and simulators may be used.

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

Typically, component testing occurs with access to the code being tested and with the support of a development environment, such as a unit test framework or debugging tool. In practice, component testing usually involves the programmer who wrote the code. Defects are typically fixed as soon as they are found, without formally managing these defects.

One approach to component testing 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 correcting any issues and iterating until they pass.

2.2.2 Integration Testing (K2)

Test basis:
- Software and system design
- Architecture
- Workflows
- Use cases

Typical test objects:
- Subsystems
- Database implementation
- Infrastructure
- Interfaces
- System configuration and configuration data

Integration testing tests interfaces between components, interactions with different parts of a system, such as the operating system, file system and hardware, and interfaces between systems.

There may be more than one level of integration testing and it may be carried out on test objects of varying size as follows:
1. Component integration testing tests the interactions between software components and is done after component testing
2. System integration testing tests the interactions between different systems or between hardware and software and may be done after system testing. In this case, the developing organization may control only one side of the interface. This might be considered as a risk. Business processes implemented as workflows may involve a series of systems. Cross-platform issues may be significant.

The greater the scope of integration, the more difficult it becomes to isolate defects to a specific component or system, which may lead to increased risk and additional time for troubleshooting.

Systematic integration strategies may be based on the system architecture (such as top-down and bottom-up), functional tasks, transaction processing sequences, or some other aspect of the system or components. In order to ease fault isolation and detect defects early, integration should normally be incremental rather than “big bang”.

Testing of specific non-functional characteristics (e.g., performance) may be included in integration testing as well as functional testing.

At each stage of integration, testers concentrate solely on the integration itself. For example, if they are integrating module A with module B they are interested in testing the communication between the modules, not the functionality of the individual module as that was done during component testing. Both functional and structural approaches may be used.

Ideally, testers should understand the architecture and influence integration planning. If integration tests are planned before components or systems are built, those components can be built in the order required for most efficient testing.

2.2.3 System Testing (K2)

Test basis:
- o System and software requirement specification
- o Use cases
- o Functional specification
- o Risk analysis reports

Typical test objects:
- o System, user and operation manuals
- o System configuration and configuration data

System testing is concerned with the behavior of a whole system/product. The testing scope shall be clearly addressed in the Master and/or Level Test Plan for that test level.

In system testing, 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 in testing.

System testing may include tests based on risks and/or on requirements specifications, business processes, use cases, or other high level text descriptions or models of system behavior, interactions with the operating system, and system resources.

System testing should investigate functional and non-functional requirements of the system, and data quality characteristics. Testers 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 techniques (white-box) may then be used to assess the thoroughness of the testing with respect to a structural element, such as menu structure or web page navigation (see Chapter 4).

An independent test team often carries out system testing.

2.2.4 Acceptance Testing (K2)

Test basis:
- o User requirements
- o System requirements
- o Use cases
- o Business processes
- o Risk analysis reports

Typical test objects:
- o Business processes on fully integrated system
- o Operational and maintenance processes
- o User procedures
- o Forms
- o Reports
- o Configuration data

Acceptance testing is often the responsibility of the customers or users of a system; other stakeholders may be involved as well.

The goal in acceptance testing is to establish confidence in the system, parts of the system or specific non-functional characteristics of the system. Finding defects is not the main focus in acceptance testing. Acceptance testing may assess the system's readiness for deployment and

use, although it is not necessarily the final level of testing. For example, a large-scale system integration test may come after the acceptance test for a system.

Acceptance testing may occur at various times in the life cycle, for example:
- A 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

Typical forms of acceptance testing include the following:

User acceptance testing
Typically verifies the fitness for use of the system by business users.

Operational (acceptance) testing
The acceptance of the system by the system administrators, including:
- Testing of backup/restore
- Disaster recovery
- User management
- Maintenance tasks
- Data load and migration tasks
- Periodic checks of security vulnerabilities

Contract and regulation acceptance testing
Contract acceptance testing is performed against a contract's acceptance criteria for producing custom-developed software. Acceptance criteria should be defined when the parties agree to the contract. Regulation acceptance testing is performed against any regulations that must be adhered to, such as government, legal or safety regulations.

Alpha and beta (or field) testing
Developers of market, or COTS, software often want to get feedback from potential or existing customers in their market before the software product is put up for sale commercially. Alpha testing is performed at the developing organization's site but not by the developing team. Beta testing, or field-testing, is performed by customers or potential customers at their own locations.

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

2.3 Test Types (K2) — *40 minutes*

Terms
Black-box testing, code coverage, functional testing, interoperability testing, load testing, maintainability testing, performance testing, portability testing, reliability testing, security testing, stress testing, structural testing, usability testing, white-box testing

Background
A group of test activities can be aimed at verifying the software system (or a part of a system) based on a specific reason or target for testing.

A test type is focused on a particular test objective, which could be any of the following:
- A function to be performed by the software
- A non-functional quality characteristic, such as reliability or usability
- The structure or architecture of the software or system
- Change related, i.e., confirming that defects have been fixed (confirmation testing) and looking for unintended changes (regression testing)

A model of the software may be developed and/or used in structural testing (e.g., a control flow model or menu structure model), non-functional testing (e.g., performance model, usability model security threat modeling), and functional testing (e.g., a process flow model, a state transition model or a plain language specification).

2.3.1 Testing of Function (Functional Testing) (K2)
The functions that a system, subsystem or component are to perform may be described in work products such as a requirements specification, use cases, or a functional specification, or they may be undocumented. The functions are “what” the system does.

Functional tests are based on functions and features (described in documents or understood by the testers) and their interoperability with specific systems, and may be performed at all test levels (e.g., tests for components may be based on a component specification).

Specification-based techniques may be used to derive test conditions and test cases from the functionality of the software or system (see Chapter 4). Functional testing considers the external behavior of the software (black-box testing).

A type of functional testing, security testing, investigates the functions (e.g., a firewall) relating to detection of threats, such as viruses, from malicious outsiders. Another type of functional testing, interoperability testing, evaluates the capability of the software product to interact with one or more specified components or systems.

2.3.2 Testing of Non-functional Software Characteristics (Non-functional Testing) (K2)
Non-functional testing includes, but is not limited to, performance testing, load testing, stress testing, usability testing, maintainability testing, reliability testing and portability testing. It is the testing of “how” the system works.

Non-functional testing may be performed at all test levels. The term non-functional testing describes the tests required to measure characteristics of systems and software that can be quantified on a varying scale, such as response times for performance testing. These tests can be referenced to a quality model such as the one defined in ‘Software Engineering – Software Product Quality’ (ISO

9126). Non-functional testing considers the external behavior of the software and in most cases uses black-box test design techniques to accomplish that.

2.3.3 Testing of Software Structure/Architecture (Structural Testing) (K2)

Structural (white-box) testing may be performed at all test levels. Structural techniques are best used after specification-based techniques, in order to help measure the thoroughness of testing through assessment of coverage of a type of structure.

Coverage is the extent that a structure has been exercised by a test suite, expressed as a percentage of the items being covered. If coverage is not 100%, then more tests may be designed to test those items that were missed to increase coverage. Coverage techniques are covered in Chapter 4.

At all test levels, but especially in component testing and component integration testing, tools can be used to measure the code coverage of elements, such as statements or decisions. Structural testing may be based on the architecture of the system, such as a calling hierarchy.

Structural testing approaches can also be applied at system, system integration or acceptance testing levels (e.g., to business models or menu structures).

2.3.4 Testing Related to Changes: Re-testing and Regression Testing (K2)

After a defect is detected and fixed, the software should be re-tested to confirm that the original defect has been successfully removed. This is called confirmation. Debugging (locating and fixing a defect) is a development activity, not a testing activity.

Regression testing is the repeated testing of an already tested program, after modification, to discover any defects introduced or uncovered as a result of the change(s). These defects may be either in the software being tested, or in another related or unrelated software component. It is performed when the software, or its environment, is changed. The extent of regression testing is based on the risk of not finding defects in software that was working previously.

Tests should be repeatable if they are to be used for confirmation testing and to assist regression testing.

Regression testing may be performed at all test levels, and includes functional, non-functional and structural testing. Regression test suites are run many times and generally evolve slowly, so regression testing is a strong candidate for automation.

Software Testing, Verification, Validation and QA

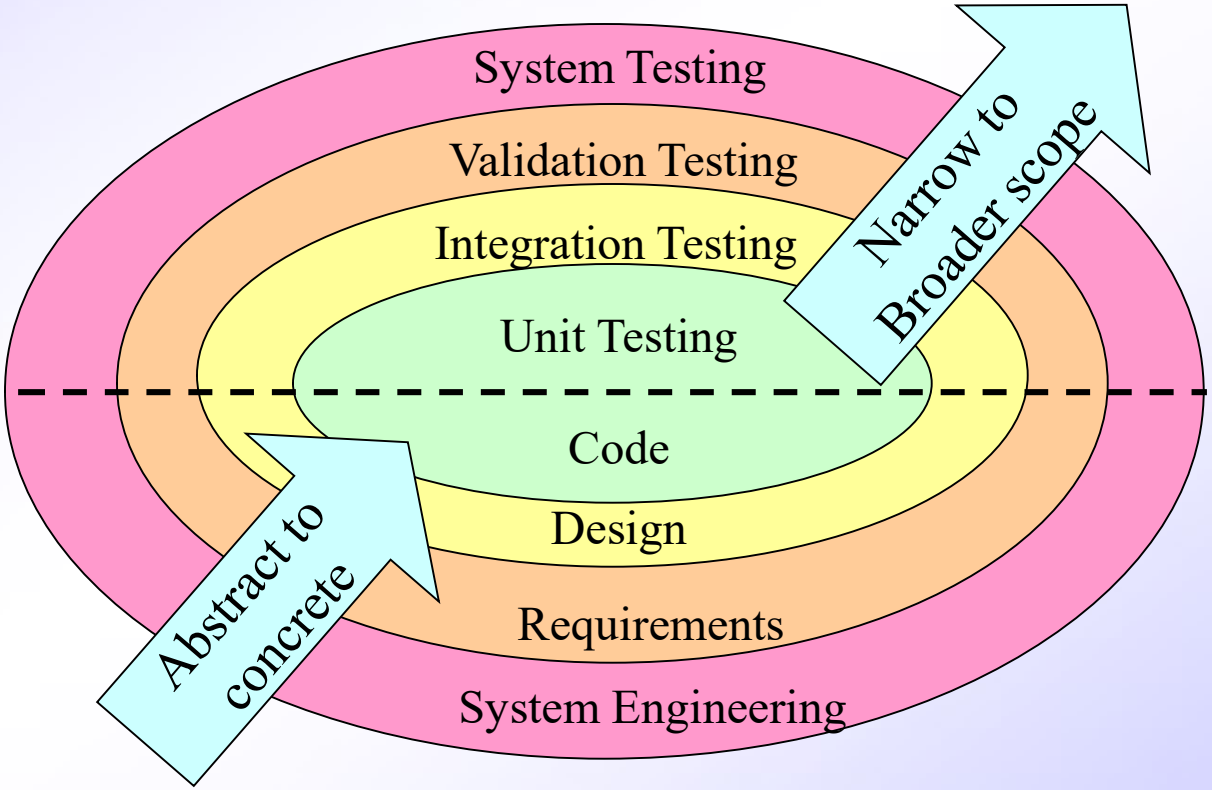
Supplementary to

Unit 4 : Testing throughout the Software Life Cycle

Compiled by
Roshan Chitrakar, PhD
roshanchi@gmail.com

(Source: Pressman, R. *Software Engineering: A Practitioner's Approach*.

Levels of Testing



Testing applied to Sequential Models

- Unit testing
  - Exercises specific paths in a component's control structure to ensure complete coverage and maximum error detection
  - Components are then assembled and integrated
- Integration testing
  - Focuses on inputs and outputs, and how well the components fit together and work together
- Validation testing
  - Provides final assurance that the software meets all functional, behavioral, and performance requirements
- System testing
  - Verifies that all system elements (software, hardware, people, databases) mesh properly and that overall system function and performance is achieved

Unit Testing

- Focuses testing on the function or software module
- Concentrates on the internal processing logic and data structures
- Is simplified when a module is designed with high cohesion
  - Reduces the number of test cases
  - Allows errors to be more easily predicted and uncovered
- Concentrates on critical modules and those with high cyclomatic complexity when testing resources are limited

Targets for Unit Test Cases

- Module interface
  - Ensure that information flows properly into and out of the module
- Local data structures
  - Ensure that data stored temporarily maintains its integrity during all steps in an algorithm execution
- Boundary conditions
  - Ensure that the module operates properly at boundary values established to limit or restrict processing
- Independent paths (basis paths)
  - Paths are exercised to ensure that all statements in a module have been executed at least once
- Error handling paths
  - Ensure that the algorithms respond correctly to specific error conditions

Drivers and Stubs for Unit Testing

- Driver
  - A simple main program that accepts test case data, passes such data to the component being tested, and prints the returned results
- Stubs
  - Serve to replace modules that are subordinate to (called by) the component to be tested
  - It uses the module's exact interface, may do minimal data manipulation, provides verification of entry, and returns control to the module undergoing testing
- Drivers and stubs both represent overhead
  - Both must be written but don't constitute part of the installed software product

Integration Testing

- Defined as a systematic technique for constructing the software architecture
  - At the same time integration is occurring, conduct tests to uncover errors associated with interfaces
- Objective is to take unit tested modules and build a program structure based on the prescribed design
- Two Approaches
  - Non-incremental Integration Testing
  - Incremental Integration Testing

Non-incremental Integration Testing

- Commonly called the “Big Bang” approach
- All components are combined in advance
- The entire program is tested as a whole
- Chaos results
- Many seemingly-unrelated errors are encountered
- Correction is difficult because isolation of causes is complicated
- Once a set of errors are corrected, more errors occur, and testing appears to enter an endless loop

Incremental Integration Testing

- Three kinds
  - Top-down integration
  - Bottom-up integration
  - Sandwich integration
- The program is constructed and tested in small increments
- Errors are easier to isolate and correct
- Interfaces are more likely to be tested completely
- A systematic test approach is applied

Top-down Integration

- Modules are integrated by moving downward through the control hierarchy, beginning with the main module
- Subordinate modules are incorporated in either a depth-first or breadth-first fashion
  - DF: All modules on a major control path are integrated
  - BF: All modules directly subordinate at each level are integrated
- Advantages
  - This approach verifies major control or decision points early in the test process
- Disadvantages
  - Stubs need to be created to substitute for modules that have not been built or tested yet; this code is later discarded
  - Because stubs are used to replace lower level modules, no significant data flow can occur until much later in the integration/testing process

Bottom-up Integration

- Integration and testing starts with the most atomic modules in the control hierarchy
- Advantages
  - This approach verifies low-level data processing early in the testing process
  - Need for stubs is eliminated
- Disadvantages
  - Driver modules need to be built to test the lower-level modules; this code is later discarded or expanded into a full-featured version
  - Drivers inherently do not contain the complete algorithms that will eventually use the services of the lower-level modules; consequently, testing may be incomplete or more testing may be needed later when the upper level modules are available

Sandwich Integration

- Consists of a combination of both top-down and bottom-up integration
- Occurs both at the highest level modules and also at the lowest level modules
- Proceeds using functional groups of modules, with each group completed before the next
  - High and low-level modules are grouped based on the control and data processing they provide for a specific program feature
  - Integration within the group progresses in alternating steps between the high and low level modules of the group
  - When integration for a certain functional group is complete, integration and testing moves onto the next group
- Reaps the advantages of both types of integration while minimizing the need for drivers and stubs
- Requires a disciplined approach so that integration doesn't tend towards the "big bang" scenario

Regression Testing

- Each new addition or change to baselined software may cause problems with functions that previously worked flawlessly
- Regression testing re-executes a small subset of tests that have already been conducted
  - Ensures that changes have not propagated unintended side effects
  - Helps to ensure that changes do not introduce unintended behavior or additional errors
  - May be done manually or through the use of automated capture/playback tools
- Regression test suite contains three different classes of test cases
  - A representative sample of tests that will exercise all software functions
  - Additional tests that focus on software functions that are likely to be affected by the change
  - Tests that focus on the actual software components that have been changed

Smoke Testing

- Taken from the world of hardware
  - Power is applied and a technician checks for sparks, smoke, or other dramatic signs of fundamental failure
- Designed as a pacing mechanism for time-critical projects
  - Allows the software team to assess its project on a frequent basis
- Includes the following activities
  - The software is compiled and linked into a build
  - A series of breadth tests is designed to expose errors that will keep the build from properly performing its function
    - The goal is to uncover "show stopper" errors that have the highest likelihood of throwing the software project behind schedule
  - The build is integrated with other builds and the entire product is smoke tested daily
    - Daily testing gives managers and practitioners a realistic assessment of the progress of the integration testing
  - After a smoke test is completed, detailed test scripts are executed

Testing for OO Models

- With object-oriented software, you can no longer test a single operation in isolation (conventional thinking)
- Traditional top-down or bottom-up integration testing has little meaning
- Class testing for object-oriented software is the equivalent of unit testing for conventional software
  - Focuses on operations encapsulated by the class and the state behavior of the class
- Drivers can be used
  - To test operations at the lowest level and for testing whole groups of classes
  - To replace the user interface so that tests of system functionality can be conducted prior to implementation of the actual interface
- Stubs can be used
  - In situations in which collaboration between classes is required but one or more of the collaborating classes has not yet been fully implemented

Testing OO Models (contd.)

- Two different object-oriented testing strategies
  - Thread-based testing
    - Integrates the set of classes required to respond to one input or event for the system
    - Each thread is integrated and tested individually
    - Regression testing is applied to ensure that no side effects occur
  - Use-based testing
    - First tests the independent classes that use very few, if any, server classes
    - Then the next layer of classes, called dependent classes, are integrated
    - This sequence of testing layer of dependent classes continues until the entire system is constructed

Validation Testing

- Validation testing follows integration testing
- The distinction between conventional and object-oriented software disappears
- Focuses on user-visible actions and user-recognizable output from the system
- Demonstrates conformity with requirements
- Designed to ensure that
  - All functional requirements are satisfied
  - All behavioral characteristics are achieved
  - All performance requirements are attained
  - Documentation is correct
  - Usability and other requirements are met (e.g., transportability, compatibility, error recovery, maintainability)
- After each validation test
  - The function or performance characteristic conforms to specification and is accepted
  - A deviation from specification is uncovered and a deficiency list is created
- A configuration review or audit ensures that all elements of the software configuration have been properly developed, cataloged, and have the necessary detail for entering the support phase of the software life cycle

Alpha and Beta Testing

- Alpha testing
  - Conducted at the developer's site by end users
  - Software is used in a natural setting with developers watching intently
  - Testing is conducted in a controlled environment
- Beta testing
  - Conducted at end-user sites
  - Developer is generally not present
  - It serves as a live application of the software in an environment that cannot be controlled by the developer
  - The end-user records all problems that are encountered and reports these to the developers at regular intervals
- After beta testing is complete, software engineers make software modifications and prepare for release of the software product to the entire customer base

System Testing

- Recovery testing
  - Tests for recovery from system faults
  - Forces the software to fail in a variety of ways and verifies that recovery is properly performed
  - Tests reinitialization, checkpointing mechanisms, data recovery, and restart for correctness
- Security testing
  - Verifies that protection mechanisms built into a system will, in fact, protect it from improper access
- Stress testing
  - Executes a system in a manner that demands resources in abnormal quantity, frequency, or volume
- Performance testing
  - Tests the run-time performance of software within the context of an integrated system
  - Often coupled with stress testing and usually requires both hardware and software instrumentation
  - Can uncover situations that lead to degradation and possible system failure

Debugging

- Debugging occurs as a consequence of successful testing
- It is still very much an art rather than a science
- Good debugging ability may be an innate human trait
- Large variances in debugging ability exist
- The debugging process begins with the execution of a test case
- Results are assessed and the difference between expected and actual performance is encountered
- This difference is a symptom of an underlying cause that lies hidden
- The debugging process attempts to match symptom with cause, thereby leading to error correction

Debugging Strategies

- Objective of debugging is to find and correct the cause of a software error
- Bugs are found by a combination of systematic evaluation, intuition, and luck
- Debugging methods and tools are not a substitute for careful evaluation based on a complete design model and clear source code
- There are three main debugging strategies
  - Brute force
  - Backtracking
  - Cause elimination

Debugging Strategy: Brute Force

- Most commonly used and least efficient method
- Used when all else fails
- Involves the use of memory dumps, run-time traces, and output statements
- Leads many times to wasted effort and time

Debugging Strategy: Backtracking

- Can be used successfully in small programs
- The method starts at the location where a symptom has been uncovered
- The source code is then traced backward (manually) until the location of the cause is found
- In large programs, the number of potential backward paths may become unmanageably large

Debugging Strategy: Cause Elimination

- Involves the use of induction or deduction and introduces the concept of binary partitioning
  - Induction (specific to general): Prove that a specific starting value is true; then prove the general case is true
  - Deduction (general to specific): Show that a specific conclusion follows from a set of general premises
- Data related to the error occurrence are organized to isolate potential causes
- A cause hypothesis is devised, and the aforementioned data are used to prove or disprove the hypothesis
- Alternatively, a list of all possible causes is developed, and tests are conducted to eliminate each cause
- If initial tests indicate that a particular cause hypothesis shows promise, data are refined in an attempt to isolate the bug

3.1 Static Techniques and the Test Process (K2)	*15 minutes*

Terms
Dynamic testing, static testing

Background
Unlike dynamic testing, which requires the execution of software, static testing techniques rely on the manual examination (reviews) and automated analysis (static analysis) of the code or other project documentation without the execution of the code.

Reviews are a way of testing software work products (including code) and can be performed well before dynamic test execution. Defects detected during reviews early in the life cycle (e.g., defects found in requirements) are often much cheaper to remove than those detected by running tests on the executing code.

A review could be done entirely as a manual activity, but there is also tool support. The main manual activity is to examine a work product and make comments about it. Any software work product can be reviewed, including requirements specifications, design specifications, code, test plans, test specifications, test cases, test scripts, user guides or web pages.

Benefits of reviews include early defect detection and correction, development productivity improvements, reduced development timescales, reduced testing cost and time, lifetime cost reductions, fewer defects and improved communication. Reviews can find omissions, for example, in requirements, which are unlikely to be found in dynamic testing.

Reviews, static analysis and dynamic testing have the same objective – identifying defects. They are complementary; the different techniques can find different types of defects effectively and efficiently. Compared to dynamic testing, static techniques find causes of failures (defects) rather than the failures themselves.

Typical defects that are easier to find in reviews than in dynamic testing include: deviations from standards, requirement defects, design defects, insufficient maintainability and incorrect interface specifications.

3.2 Review Process (K2) — *25 minutes*

Terms
Entry criteria, formal review, informal review, inspection, metric, moderator, peer review, reviewer, scribe, technical review, walkthrough

Background
The different types of reviews vary from informal, characterized by no written instructions for reviewers, to systematic, characterized by team participation, documented results of the review, and documented procedures for conducting the review. The formality of a review process is related to factors such as the maturity of the development process, any legal or regulatory requirements or the need for an audit trail.

The way a review is carried out depends on the agreed objectives of the review (e.g., find defects, gain understanding, educate testers and new team members, or discussion and decision by consensus).

3.2.1 Activities of a Formal Review (K1)

A typical formal review has the following main activities:

1. Planning
   - Defining the review criteria
   - Selecting the personnel
   - Allocating roles
   - Defining the entry and exit criteria for more formal review types (e.g., inspections)
   - Selecting which parts of documents to review
   - Checking entry criteria (for more formal review types)
2. Kick-off
   - Distributing documents
   - Explaining the objectives, process and documents to the participants
3. Individual preparation
   - Preparing for the review meeting by reviewing the document(s)
   - Noting potential defects, questions and comments
4. Examination/evaluation/recording of results (review meeting)
   - Discussing or logging, with documented results or minutes (for more formal review types)
   - Noting defects, making recommendations regarding handling the defects, making decisions about the defects
   - Examining/evaluating and recording issues during any physical meetings or tracking any group electronic communications
5. Rework
   - Fixing defects found (typically done by the author)
   - Recording updated status of defects (in formal reviews)
6. Follow-up
   - Checking that defects have been addressed
   - Gathering metrics
   - Checking on exit criteria (for more formal review types)

3.2.2 Roles and Responsibilities (K1)

A typical formal review will include the roles below:
- Manager: decides on the execution of reviews, allocates time in project schedules and determines if the review objectives have been met.

- Moderator: the person who leads the review of the document or set of documents, including planning the review, running the meeting, and following-up after the meeting. If necessary, the moderator may mediate between the various points of view and is often the person upon whom the success of the review rests.
- Author: the writer or person with chief responsibility for the document(s) to be reviewed.
- Reviewers: individuals with a specific technical or business background (also called checkers or inspectors) who, after the necessary preparation, identify and describe findings (e.g., defects) in the product under review. Reviewers should be chosen to represent different perspectives and roles in the review process, and should take part in any review meetings.
- Scribe (or recorder): documents all the issues, problems and open points that were identified during the meeting.

Looking at software products or related work products from different perspectives and using checklists can make reviews more effective and efficient. For example, a checklist based on various perspectives such as user, maintainer, tester or operations, or a checklist of typical requirements problems may help to uncover previously undetected issues.

3.2.3 Types of Reviews (K2)

A single software product or related work product may be the subject of more than one review. If more than one type of review is used, the order may vary. For example, an informal review may be carried out before a technical review, or an inspection may be carried out on a requirements specification before a walkthrough with customers. The main characteristics, options and purposes of common review types are:

Informal Review

- No formal process
- May take the form of pair programming or a technical lead reviewing designs and code
- Results may be documented
- Varies in usefulness depending on the reviewers
- Main purpose: inexpensive way to get some benefit

Walkthrough

- Meeting led by author
- May take the form of scenarios, dry runs, peer group participation
- Open-ended sessions
  - Optional pre-meeting preparation of reviewers
  - Optional preparation of a review report including list of findings
- Optional scribe (who is not the author)
- May vary in practice from quite informal to very formal
- Main purposes: learning, gaining understanding, finding defects

Technical Review

- Documented, defined defect-detection process that includes peers and technical experts with optional management participation
- May be performed as a peer review without management participation
- Ideally led by trained moderator (not the author)
- Pre-meeting preparation by reviewers
- Optional use of checklists
- Preparation of a review report which includes the list of findings, the verdict whether the software product meets its requirements and, where appropriate, recommendations related to findings
- May vary in practice from quite informal to very formal
- Main purposes: discussing, making decisions, evaluating alternatives, finding defects, solving technical problems and checking conformance to specifications, plans, regulations, and standards

Inspection
- Led by trained moderator (not the author)
- Usually conducted as a peer examination
- Defined roles
- Includes metrics gathering
- Formal process based on rules and checklists
- Specified entry and exit criteria for acceptance of the software product
- Pre-meeting preparation
- Inspection report including list of findings
- Formal follow-up process (with optional process improvement components)
- Optional reader
- Main purpose: finding defects

Walkthroughs, technical reviews and inspections can be performed within a peer group, i.e., colleagues at the same organizational level. This type of review is called a “peer review”.

3.2.4 Success Factors for Reviews (K2)

Success factors for reviews include:
- Each review has clear predefined objectives
- The right people for the review objectives are involved
- Testers are valued reviewers who contribute to the review and also learn about the product which enables them to prepare tests earlier
- Defects found are welcomed and expressed objectively
- People issues and psychological aspects are dealt with (e.g., making it a positive experience for the author)
- The review is conducted in an atmosphere of trust; the outcome will not be used for the evaluation of the participants
- Review techniques are applied that are suitable to achieve the objectives and to the type and level of software work products and reviewers
- Checklists or roles are used if appropriate to increase effectiveness of defect identification
- Training is given in review techniques, especially the more formal techniques such as inspection
- Management supports a good review process (e.g., by incorporating adequate time for review activities in project schedules)
- There is an emphasis on learning and process improvement

3.3 Static Analysis by Tools (K2) — *20 minutes*

Terms
Compiler, complexity, control flow, data flow, static analysis

Background
The objective of static analysis is to find defects in software source code and software models. Static analysis is performed without actually executing the software being examined by the tool; dynamic testing does execute the software code. Static analysis can locate defects that are hard to find in dynamic testing. As with reviews, static analysis finds defects rather than failures. Static analysis tools analyze program code (e.g., control flow and data flow), as well as generated output such as HTML and XML.

The value of static analysis is:
- Early detection of defects prior to test execution
- Early warning about suspicious aspects of the code or design by the calculation of metrics, such as a high complexity measure
- Identification of defects not easily found by dynamic testing
- Detecting dependencies and inconsistencies in software models such as links
- Improved maintainability of code and design
- Prevention of defects, if lessons are learned in development

Typical defects discovered by static analysis tools include:
- Referencing a variable with an undefined value
- Inconsistent interfaces between modules and components
- Variables that are not used or are improperly declared
- Unreachable (dead) code
- Missing and erroneous logic (potentially infinite loops)
- Overly complicated constructs
- Programming standards violations
- Security vulnerabilities
- Syntax violations of code and software models

Static analysis tools are typically used by developers (checking against predefined rules or programming standards) before and during component and integration testing or when checking-in code to configuration management tools, and by designers during software modeling. Static analysis tools may produce a large number of warning messages, which need to be well-managed to allow the most effective use of the tool.

Compilers may offer some support for static analysis, including the calculation of metrics.

References
3.2 IEEE 1028
3.2.2 Gilb, 1993, van Veenendaal, 2004
3.2.4 Gilb, 1993, IEEE 1028
3.3 van Veenendaal, 2004

SQA : Chapter 5 Supplementary Note

- Compiled by Roshan Chitrakar

Difference Between Static Testing And Dynamic Testing

Static testing and dynamic testing are important testing methods available for developers and testers in Software Development lifecycle. These are *software testing techniques* which the organisation must choose carefully which to implement on the software application. In order to get the most out of each type of testing, and choose the right tools for a given situation, it's crucial to understand the benefits and limitations of each type of testing.

What is Static Testing?

Static Testing is type of testing in which the code is not executed. It can be done manually or by a set of tools. This type of testing checks the code, requirement documents and design documents and puts review comments on the work document. When the software is non–operational and inactive, we perform security testing to analyse the software in non-runtime environment. With static testing, we try to find out the errors, code flaws and potentially malicious code in the software application. It starts earlier in development life cycle and hence it is also called verification testing. Static testing can be done on work documents like requirement specifications, design documents, source code, test plans, test scripts and test cases, web page content.

The Static test techniques include:

•***Inspection:*** Here the main purpose is to find defects. Code walkthroughs are conducted by moderator. It is a formal type of review where a checklist is prepared to review the work documents.

•***Walkthrough***: In this type of technique a meeting is lead by author to explain the product. Participants can ask questions and a scribe is assigned to make notes.

•***Technical reviews***: In this type of static testing a technical round of review is conducted to check if the code is made according to technical specifications and standards. Generally the test plans, test strategy and test scripts are reviewed here.

•***Informal reviews***: Static testing technique in which the document is reviewed informally and informal comments are provided.

What is Dynamic Testing?

Dynamic testing is done when the code is in operation mode. Dynamic testing is performed in runtime environment. When the code being executed is input with a value, the result or the output of the code is checked and compared with the expected output. With this we can observe the functional behaviour of

the software, monitor the system memory, CPU response time, performance of the system. Dynamic testing is also known as validation testing , evaluating the finished product. Dynamic testing is of two types: Functional Testing and Non functional testing.

Types of Dynamic Testing techniques are as follows:

- ***Unit Testing***: Testing of individual modules by developers.. The source code is tested in it.
- ***Integration Testing***: Testing the interface between different modules then they are joined..
- ***System Testing***: Testing performed on the system as a whole.
- ***Acceptance Testing***: Testing done from user point of view at user's end.

However, both Static Testing and Dynamic Testing are important for the software application. There are number of strengths and weaknesses associated with both types of testing which should be considered while implementing these testing on code:

Difference between Static Testing and Dynamic Testing

Static Testing	Dynamic Testing
1. Static Testing is white box testing which is done at early stage if development life cycle. It is more cost effective than dynamic testing	1. Dynamic Testing on the other hand is done at the later stage of development lifecycle.
2. Static testing has more statement coverage than dynamic testing in shorter time	2. Dynamic Testing has less statement stage because it is covers limited area of code
3. It is done before code deployment	3. It is done after code deployment
4. It is performed in Verification Stage	4. It is done in Validation Stage
5. This type of testing is done without the execution of code.	5. This type of execution is done with the execution of code.
6. Static testing gives assessment of code as well as documentation.	6. Dynamic Testing gives bottlenecks of the software system.
7. In Static Testing techniques a checklist is prepared for testing process	7. In Dynamic Testing technique the test cases are executed.
8. Static Testing Methods include Walkthroughs, code review.	8. Dynamic testing involves functional and nonfunctional testing

Example : Software Application: *Online Shopping Cart*

Static Test Techniques:

1.Review the requirement documents, design documents initially

2.Checking the GUI of the application

3. Checking the database structure of the application.

Dynamic Testing Techniques:

1.Testing the functionality of the different page.

2.Checking the checkout process and payment methods.

3.Testing the interfaces between different pages.

SQA Chapter 6 Supplementary note

Compiled by Roshan Chitrakar

Source: Ron Patton, Software Testing

Testing Specifications

- Highlights of this chapter include
  - What is black-box and white-box testing
  - How static and dynamic testing differ
  - What high-level techniques can be used for reviewing a product specification

A tester uses the specification document to find bugs before the first line of code is written

Black-Box and White-Box Testing

- In black-box testing, the tester only knows what the software is supposed to do—he can’t look in the box to see how it operates.
- In white-box testing (sometimes called clear-box testing), the software tester has access to the program’s code and can examine it for clues to help him with his testing—he can see inside the box.
- Based on what he sees, the tester may determine that certain numbers are more or less likely to fail and can tailor his testing based on that information.

Static and Dynamic Testing

- Static testing refers to testing something that’s not running—examining and reviewing it.
- Dynamic testing is what you would normally think of as testing—running and using the software

Static Black-Box Testing: Testing the Specification

- Testing the specification is static black-box testing.
- You can take that document, perform static black-box testing, and carefully examine it for bugs.
- You can even test an unwritten specification by questioning the people who are designing and writing the software.

High-Level Review of the Specification

- The first step is to stand back and view it from a high level. Examine the spec for large fundamental problems, oversights, and omissions.
- Pretend to Be the Customer
- Research Existing Standards and Guidelines
- Review and Test Similar Software

Research Existing Standards and Guidelines

- ***Corporate Terminology and Conventions.*** If this software is tailored for a specific company, it should adopt the common terms and conventions used by the employees of

that company.

- ***Industry Requirements.*** The medical, pharmaceutical, industrial, and financial industries have very strict standards that their software must follow.
- ***Government Standards.*** The government, especially the military, has strict standards.
- ***Graphical User Interface (GUI).*** If your software runs under Microsoft Windows or Apple Macintosh operating systems, there are published standards and guidelines for how the software should look and feel to a user.
- ***Hardware and Networking Standards.*** Low-level software and hardware interface standards must be adhered to, to assure compatibility across systems.

Review and Test Similar Software

- ***Scale***. Will your software be smaller or larger? Will that size make a difference in your testing?
- ***Complexity***. Will your software be more or less complex? Will this impact your testing?
- ***Testability***. Will you have the resources, time, and expertise to test software such as this?
- ***Quality/Reliability***. Is this software representative of the overall quality planned for your software? Will your software be more or less reliable?

Low-Level Specification Test Techniques

- Specification Attributes Checklist
- Specification Terminology Checklist

Specification Attributes Checklist

- ***Complete***. Is anything missing or forgotten? Is it thorough? Does it include everything necessary to make it stand alone?
- ***Accurate***. Is the proposed solution correct? Does it properly define the goal? Are there any errors?
- ***Precise, Unambiguous, and Clear***. Is the description exact and not vague? Is there a single interpretation? Is it easy to read and understandable?
- ***Consistent***. Is the description of the feature written so that it doesn't conflict with itself or other items in the specification?
- ***Relevant***. Is the statement necessary to specify the feature? Is it extra information that should be left out? Is the feature traceable to an original customer need?
- ***Feasible***. Can the feature be implemented with the available personnel, tools, and resources within the specified budget and schedule?
- ***Code-free***. Does the specification stick with defining the product and not the underlying software design, architecture, and code?
- ***Testable***. Can the feature be tested? Is enough information provided that a tester could create tests to verify its operation?

Specification Terminology Checklist

- ***Always, Every, All, None, Never.*** If you see words such as these that denote something as certain or absolute, make sure that it is, indeed, certain. Put on your tester's hat and think of cases that violate them.
- ***Certainly, Therefore, Clearly, Obviously, Evidently***. These words tend to persuade you into accepting something as a given. Don't fall into the trap.
- ***Some, Sometimes, Often, Usually, Ordinarily, Customarily, Most, Mostly***. These words are too vague. It's impossible to test a feature that operates "sometimes."
- ***Etc., And So Forth, And So On, Such As***. Lists that finish with words such as these aren't testable. Lists need to be absolute or explained so that there's no confusion as to how the series is generated and what appears next in the list.
- ***Good, Fast, Cheap, Efficient, Small, Stable***. These are unquantifiable terms. They aren't testable. If they appear in a specification, they must be further defined to explain exactly what they mean.
- ***Handled, Processed, Rejected, Skipped, Eliminated***. These terms can hide large amounts of functionality that need to be specified.
- ***If…Then…(but missing Else)***. Look for statements that have "If…Then" clauses but don't have a matching "Else." Ask yourself what will happen if the "if" doesn't happen.

Dynamic White-Box Testing

Dynamic white-box testing is using information you gain from seeing what the code does and how it works to determine.

Also called “Structural Testing”

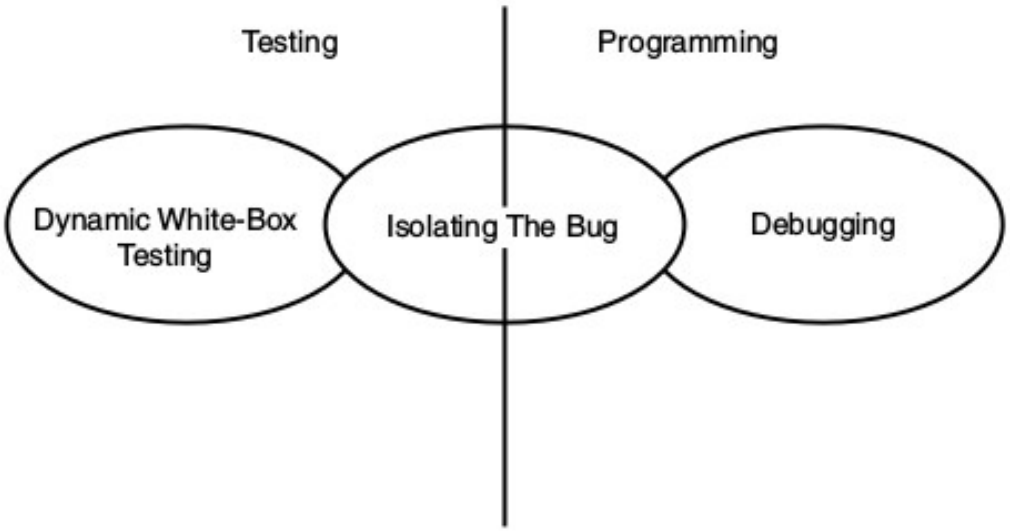
Dynamic white-box testing isn’t limited just to seeing what the code does. It also can involve directly testing and controlling the software viz.

1. Directly testing low-level functions, procedures, subroutines, or libraries
2. Adjusting your test cases based on what you know
3. Gaining access to read variables and state information
4. Measuring how much of the code and specifically what code you “hit”

Dynamic White-Box Testing versus Debugging

The goal of dynamic white-box testing is to find bugs. The goal of debugging is to fix them.

If it's white-box testing, that could even include information about what lines of code look suspicious. The programmer who does the debugging picks the process up from there, determines exactly what is causing the bug, and attempts to fix it.



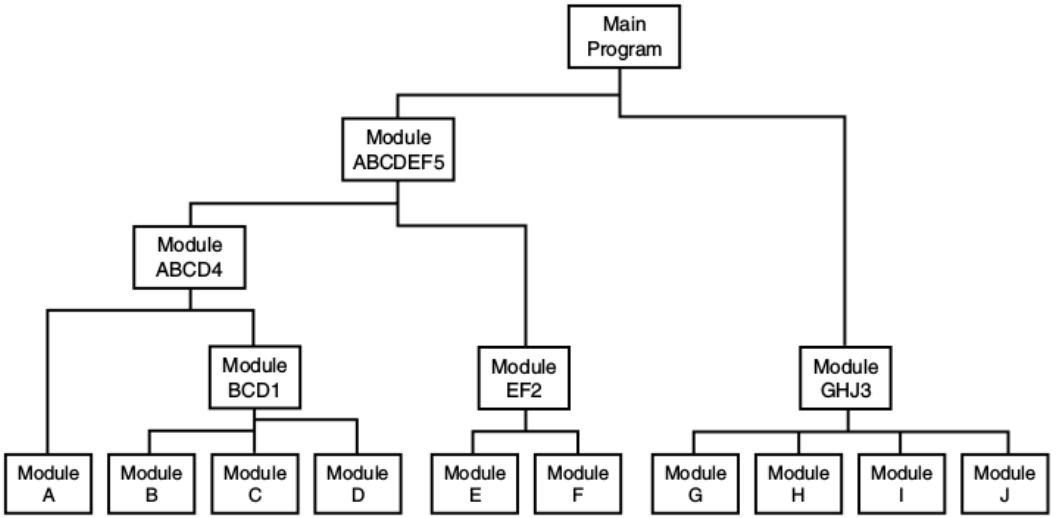
- You will use many of the same tools that programmers use.
- You can use the same compiler but possibly with different settings to enable better error detection.
- You may also write your own programs to test separate code modules

Unit and Integration Testing

Testing that occurs at the lowest level is called unit testing or module testing.

Integration testing is performed against groups of modules until the entire product is tested at once in a process called system testing.

When a problem is found at the unit level, the problem must be in that unit. If a bug is found when multiple units are integrated, it must be related to how the modules interact.

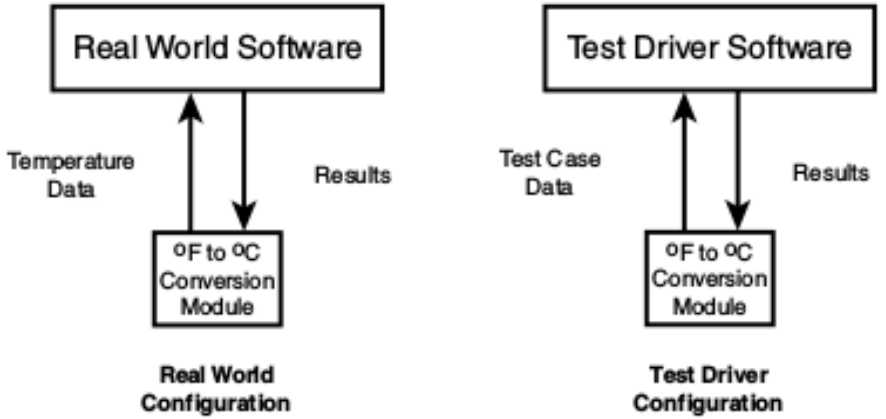


Top-down and Bottom-up testing

There are two approaches to incremental testing: bottom-up and top-down.

In bottom-up testing, you write your own modules, called test drivers.

These drivers send test-case data to the modules under test, read back the results, and verify that they're correct.

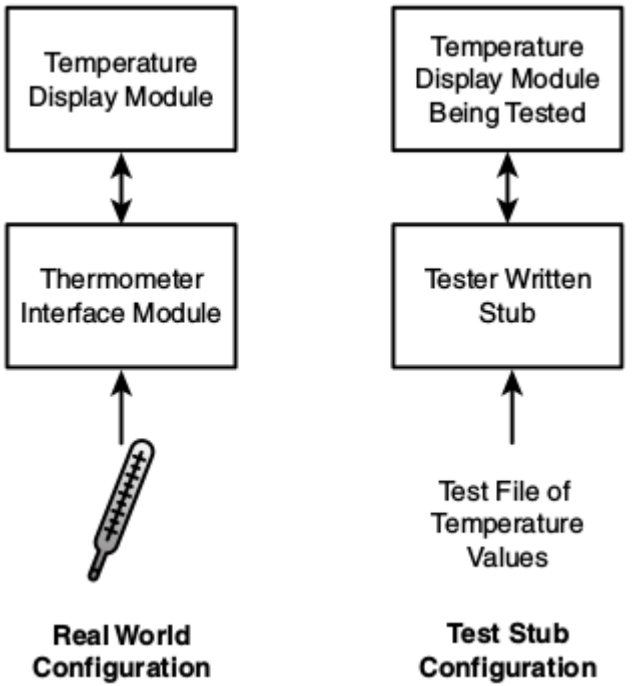


Top-down and Bottom-up testing
(contd.)

Top-down testing may sound like big-bang testing on a smaller scale.

you could write a small piece of code called a stub that acts just like the interface module.

With this test stub configuration, you could quickly run through numerous test values.



Data Coverage

Divide the code just as you did in black-box testing —into its data and its states (or program flow)

you can easily map the white-box information you gain to the black-box cases you've already written.

Consider the data first. Data includes all the variables, constants, arrays, data structures, keyboard and mouse input, files and screen input and output, and I/O to other devices such as modems, networks, and so on.

Data Flow

Data flow coverage involves tracking a piece of data completely through the software.

At the unit test level, this would just be through an individual module or function. The same tracking could be done through several integrated modules.

you would use a debugger and watch variables to view the data as the program runs.

With dynamic white-box testing, you could also check intermediate values during program execution.

Sub-Boundaries

Every piece of software will have its own unique sub-boundaries e.g.

- A module that computes taxes might switch from using a data table to using a formula at a certain financial cut-off point.
- An operating system running low on RAM may start moving data to temporary storage on the hard drive.
- A complex numerical analysis program may switch to a different equation for solving the problem depending on the size of the number.

If you perform white-box testing, you need to examine the code carefully to look for sub-boundary conditions.

Formulas and Equations

A good black-box tester would hopefully choose a test case of n=0, but a white-box tester, after seeing the formula in the code, would know to try n=0 because that would cause the formula to blow up with a divide-by-zero error.

But, what if n was the result of another computation?

Error Forcing

If you’re running the software that you’re testing in a debugger, you don’t just have the ability to watch variables and see what values they hold —you can also force them to specific values.

You could use your debugger to force it to zero.

If you take care in selecting your error forcing scenarios, error forcing can be an effective tool.

Code Coverage

You must test the program's states and the program's flow among them; must attempt to enter and exit every module, execute every line of code, and follow every logic and decision path. Examining the software at this level of detail is called code-coverage analysis.

For very small programs or individual modules, using a debugger is often sufficient. However, performing code coverage on most software requires a specialized tool known as a code coverage analyzer.

Code-coverage analyzers hook into the software you're testing and run transparently in the background while you run your test cases.

You can then obtain statistics that identify which portions of the software were executed and which portions weren't.

You will also have a general feel for the quality of the software.

Program Statement and Line Coverage

Your goal is to make sure that you execute every statement in the program at least once.

You could run your tests and add test cases until every statement in the program is touched.

It can tell you if every statement is executed, but it can't tell you if you've taken all the paths through the software.

LISTING 7.1 It's Very Easy to Test Every Line of This Simple Program

```
PRINT "Hello World"
PRINT "The date is: "; Date$
PRINT "The time is: "; Time$
END
```

Branch Coverage

The simplest form of path testing is called branch coverage testing.

In the example below, ensuring 100 percent statement coverage requires only a single test case with the Date$ variable set to January 1, 2000.

But you still need to try a test case for a date that's not January 1, 2000, which would execute the other path through the program.

LISTING 7.2 The IF Statement Creates Another Branch Through the Code

```
PRINT "Hello World"
IF Date$ = "01-01-2000" THEN
    PRINT "Happy New Year"
    END IF
PRING "The date is: "; Date$
PRINT "The time is: "; Time$
END
```

Condition Coverage

Condition coverage testing takes the extra conditions on the branch statements into account.

```
PRINT "Hello World"
IF Date$ = "01-01-2000" AND Time$ = "00:00:00" THEN
    PRINT "Happy New Year"
    END IF
PRINT "The date is: "; Date$
PRINT "The time is: "; Time$
END
```

The following cases assure that each possibility in the IF statement are covered. If you were concerned only with branch coverage, the first three conditions would be redundant and could be equivalence partitioned into a single test case. But, with condition coverage testing, all four cases are important because they exercise different conditions of the IF statement in line 4.

Date$	*Time$*	*Line # Execution*
01-01-0000	11:11:11	1,2,5,6,7
01-01-0000	00:00:00	1,2,5,6,7
01-01-2000	11:11:11	1,2,5,6,7
01-01-2000	00:00:00	1,2,3,4,5,6,7

4.1 The Test Development Process (K3) — *15 minutes*

Terms
Test case specification, test design, test execution schedule, test procedure specification, test script, traceability

Background
The test development process described in this section can be done in different ways, from very informal with little or no documentation, to very formal (as it is described below). The level of formality depends on the context of the testing, including the maturity of testing and development processes, time constraints, safety or regulatory requirements, and the people involved.

During test analysis, the test basis documentation is analyzed in order to determine what to test, i.e., to identify the test conditions. A test condition is defined as an item or event that could be verified by one or more test cases (e.g., a function, transaction, quality characteristic or structural element).

Establishing traceability from test conditions back to the specifications and requirements enables both effective impact analysis when requirements change, and determining requirements coverage for a set of tests. During test analysis the detailed test approach is implemented to select the test design techniques to use based on, among other considerations, the identified risks (see Chapter 5 for more on risk analysis).

During test design the test cases and test data are created and specified. A test case consists of a set of input values, execution preconditions, expected results and execution postconditions, defined to cover a certain test objective(s) or test condition(s). The ‘Standard for Software Test Documentation’ (IEEE STD 829-1998) describes the content of test design specifications (containing test conditions) and test case specifications.

Expected results should be produced as part of the specification of a test case and include outputs, changes to data and states, and any other consequences of the test. If expected results have not been defined, then a plausible, but erroneous, result may be interpreted as the correct one. Expected results should ideally be defined prior to test execution.

During test implementation the test cases are developed, implemented, prioritized and organized in the test procedure specification (IEEE STD 829-1998). The test procedure specifies the sequence of actions for the execution of a test. If tests are run using a test execution tool, the sequence of actions is specified in a test script (which is an automated test procedure).

The various test procedures and automated test scripts are subsequently formed into a test execution schedule that defines the order in which the various test procedures, and possibly automated test scripts, are executed. The test execution schedule will take into account such factors as regression tests, prioritization, and technical and logical dependencies.

4.2 Categories of Test Design Techniques (K2) — *15 minutes*

Terms
Black-box test design technique, experience-based test design technique, test design technique, white-box test design technique

Background
The purpose of a test design technique is to identify test conditions, test cases, and test data.

It is a classic distinction to denote test techniques as black-box or white-box. Black-box test design techniques (also called specification-based techniques) are a way to derive and select test conditions, test cases, or test data based on an analysis of the test basis documentation. This includes both functional and non-functional testing. Black-box testing, by definition, does not use any information regarding the internal structure of the component or system to be tested. White-box test design techniques (also called structural or structure-based techniques) are based on an analysis of the structure of the component or system. Black-box and white-box testing may also be combined with experience-based techniques to leverage the experience of developers, testers and users to determine what should be tested.

Some techniques fall clearly into a single category; others have elements of more than one category.

This syllabus refers to specification-based test design techniques as black-box techniques and structure-based test design techniques as white-box techniques. In addition experience-based test design techniques are covered.

Common characteristics of specification-based test design techniques include:
- Models, either formal or informal, are used for the specification of the problem to be solved, the software or its components
- Test cases can be derived systematically from these models

Common characteristics of structure-based test design techniques include:
- Information about how the software is constructed is used to derive the test cases (e.g., code and detailed design information)
- The extent of coverage of the software can be measured for existing test cases, and further test cases can be derived systematically to increase coverage

Common characteristics of experience-based test design techniques include:
- The knowledge and experience of people are used to derive the test cases
- The knowledge of testers, developers, users and other stakeholders about the software, its usage and its environment is one source of information
- Knowledge about likely defects and their distribution is another source of information

4.3 Specification-based or Black-box Techniques (K3) — *150 minutes*

Terms
Boundary value analysis, decision table testing, equivalence partitioning, state transition testing, use case testing

4.3.1 Equivalence Partitioning (K3)

In equivalence partitioning, inputs to the software or system are divided into groups that are expected to exhibit similar behavior, so they are likely to be processed in the same way. Equivalence partitions (or classes) can be found for both valid data, i.e., values that should be accepted and invalid data, i.e., values that should be rejected. Partitions can also be identified for outputs, internal values, time-related values (e.g., before or after an event) and for interface parameters (e.g., integrated components being tested during integration testing). Tests can be designed to cover all valid and invalid partitions. Equivalence partitioning is applicable at all levels of testing.

Equivalence partitioning can be used to achieve input and output coverage goals. It can be applied to human input, input via interfaces to a system, or interface parameters in integration testing.

4.3.2 Boundary Value Analysis (K3)

Behavior at the edge of each equivalence partition is more likely to be incorrect than behavior within the partition, so boundaries are an area where testing is likely to yield defects. The maximum and minimum values of a partition are its boundary values. A boundary value for a valid partition is a valid boundary value; the boundary of an invalid partition is an invalid boundary value. Tests can be designed to cover both valid and invalid boundary values. When designing test cases, a test for each boundary value is chosen.

Boundary value analysis can be applied at all test levels. It is relatively easy to apply and its defect-finding capability is high. Detailed specifications are helpful in determining the interesting boundaries.

This technique is often considered as an extension of equivalence partitioning or other black-box test design techniques. It can be used on equivalence classes for user input on screen as well as, for example, on time ranges (e.g., time out, transactional speed requirements) or table ranges (e.g., table size is 256*256).

4.3.3 Decision Table Testing (K3)

Decision tables are a good way to capture system requirements that contain logical conditions, and to document internal system design. They may be used to record complex business rules that a system is to implement. When creating decision tables, the specification is analyzed, and conditions and actions of the system are identified. The input conditions and actions are most often stated in such a way that they must be true or false (Boolean). The decision table contains the triggering conditions, often combinations of true and false for all input conditions, and the resulting actions for each combination of conditions. Each column of the table corresponds to a business rule that defines a unique combination of conditions and which result in the execution of the actions associated with that rule. The coverage standard commonly used with decision table testing is to have at least one test per column in the table, which typically involves covering all combinations of triggering conditions.

The strength of decision table testing is that it creates combinations of conditions that otherwise might not have been exercised during testing. It may be applied to all situations when the action of the software depends on several logical decisions.

4.3.4 State Transition Testing (K3)

A system may exhibit a different response depending on current conditions or previous history (its state). In this case, that aspect of the system can be shown with a state transition diagram. It allows the tester to view the software in terms of its states, transitions between states, the inputs or events that trigger state changes (transitions) and the actions which may result from those transitions. The states of the system or object under test are separate, identifiable and finite in number.

A state table shows the relationship between the states and inputs, and can highlight possible transitions that are invalid.

Tests can be designed to cover a typical sequence of states, to cover every state, to exercise every transition, to exercise specific sequences of transitions or to test invalid transitions.

State transition testing is much used within the embedded software industry and technical automation in general. However, the technique is also suitable for modeling a business object having specific states or testing screen-dialogue flows (e.g., for Internet applications or business scenarios).

4.3.5 Use Case Testing (K2)

Tests can be derived from use cases. A use case describes interactions between actors (users or systems), which produce a result of value to a system user or the customer. Use cases may be described at the abstract level (business use case, technology-free, business process level) or at the system level (system use case on the system functionality level). Each use case has preconditions which need to be met for the use case to work successfully. Each use case terminates with postconditions which are the observable results and final state of the system after the use case has been completed. A use case usually has a mainstream (i.e., most likely) scenario and alternative scenarios.

Use cases describe the “process flows” through a system based on its actual likely use, so the test cases derived from use cases are most useful in uncovering defects in the process flows during real-world use of the system. Use cases are very useful for designing acceptance tests with customer/user participation. They also help uncover integration defects caused by the interaction and interference of different components, which individual component testing would not see. Designing test cases from use cases may be combined with other specification-based test techniques.

4.4 Structure-based or White-box Techniques (K4)	*60 minutes*

Terms
Code coverage, decision coverage, statement coverage, structure-based testing

Background
Structure-based or white-box testing is based on an identified structure of the software or the system, as seen in the following examples:
- Component level: the structure of a software component, i.e., statements, decisions, branches or even distinct paths
- Integration level: the structure may be a call tree (a diagram in which modules call other modules)
- System level: the structure may be a menu structure, business process or web page structure

In this section, three code-related structural test design techniques for code coverage, based on statements, branches and decisions, are discussed. For decision testing, a control flow diagram may be used to visualize the alternatives for each decision.

4.4.1 Statement Testing and Coverage (K4)
In component testing, statement coverage is the assessment of the percentage of executable statements that have been exercised by a test case suite. The statement testing technique derives test cases to execute specific statements, normally to increase statement coverage.

Statement coverage is determined by the number of executable statements covered by (designed or executed) test cases divided by the number of all executable statements in the code under test.

4.4.2 Decision Testing and Coverage (K4)
Decision coverage, related to branch testing, is the assessment of the percentage of decision outcomes (e.g., the True and False options of an IF statement) that have been exercised by a test case suite. The decision testing technique derives test cases to execute specific decision outcomes. Branches originate from decision points in the code and show the transfer of control to different locations in the code.

Decision coverage is determined by the number of all decision outcomes covered by (designed or executed) test cases divided by the number of all possible decision outcomes in the code under test.

Decision testing is a form of control flow testing as it follows a specific flow of control through the decision points. Decision coverage is stronger than statement coverage; 100% decision coverage guarantees 100% statement coverage, but not vice versa.

4.4.3 Other Structure-based Techniques (K1)
There are stronger levels of structural coverage beyond decision coverage, for example, condition coverage and multiple condition coverage.

The concept of coverage can also be applied at other test levels For example, at the integration level the percentage of modules, components or classes that have been exercised by a test case suite could be expressed as module, component or class coverage.

Tool support is useful for the structural testing of code.

4.5 Experience-based Techniques (K2) — *30 minutes*

Terms
Exploratory testing, (fault) attack

Background
Experience-based testing is where tests are derived from the tester's skill and intuition and their experience with similar applications and technologies. When used to augment systematic techniques, these techniques can be useful in identifying special tests not easily captured by formal techniques, especially when applied after more formal approaches. However, this technique may yield widely varying degrees of effectiveness, depending on the testers' experience.

A commonly used experience-based technique is error guessing. Generally testers anticipate defects based on experience. A structured approach to the error guessing technique is to enumerate a list of possible defects and to design tests that attack these defects. This systematic approach is called fault attack. These defect and failure lists can be built based on experience, available defect and failure data, and from common knowledge about why software fails.

Exploratory testing is concurrent test design, test execution, test logging and learning, based on a test charter containing test objectives, and carried out within time-boxes. It is an approach that is most useful where there are few or inadequate specifications and severe time pressure, or in order to augment or complement other, more formal testing. It can serve as a check on the test process, to help ensure that the most serious defects are found.

4.6 Choosing Test Techniques (K2) — *15 minutes*

Terms
No specific terms.

Background
The choice of which test techniques to use depends on a number of factors, including the type of system, regulatory standards, customer or contractual requirements, level of risk, type of risk, test objective, documentation available, knowledge of the testers, time and budget, development life cycle, use case models and previous experience with types of defects found.

Some techniques are more applicable to certain situations and test levels; others are applicable to all test levels.

When creating test cases, testers generally use a combination of test techniques including process, rule and data-driven techniques to ensure adequate coverage of the object under test.

References
4.1 Craig, 2002, Hetzel, 1988, IEEE STD 829-1998
4.2 Beizer, 1990, Copeland, 2004
4.3.1 Copeland, 2004, Myers, 1979
4.3.2 Copeland, 2004, Myers, 1979
4.3.3 Beizer, 1990, Copeland, 2004
4.3.4 Beizer, 1990, Copeland, 2004
4.3.5 Copeland, 2004
4.4.3 Beizer, 1990, Copeland, 2004
4.5 Kaner, 2002
4.6 Beizer, 1990, Copeland, 2004

5.1 Test Organization (K2) — *30 minutes*

Terms
Tester, test leader, test manager

5.1.1 Test Organization and Independence (K2)

The effectiveness of finding defects by testing and reviews can be improved by using independent testers. Options for independence include the following:
- No independent testers; developers test their own code
- Independent testers within the development teams
- Independent test team or group within the organization, reporting to project management or executive management
- Independent testers from the business organization or user community
- Independent test specialists for specific test types such as usability testers, security testers or certification testers (who certify a software product against standards and regulations)
- Independent testers outsourced or external to the organization

For large, complex or safety critical projects, it is usually best to have multiple levels of testing, with some or all of the levels done by independent testers. Development staff may participate in testing, especially at the lower levels, but their lack of objectivity often limits their effectiveness. The independent testers may have the authority to require and define test processes and rules, but testers should take on such process-related roles only in the presence of a clear management mandate to do so.

The benefits of independence include:
- Independent testers see other and different defects, and are unbiased
- An independent tester can verify assumptions people made during specification and implementation of the system

Drawbacks include:
- Isolation from the development team (if treated as totally independent)
- Developers may lose a sense of responsibility for quality
- Independent testers may be seen as a bottleneck or blamed for delays in release

Testing tasks may be done by people in a specific testing role, or may be done by someone in another role, such as a project manager, quality manager, developer, business and domain expert, infrastructure or IT operations.

5.1.2 Tasks of the Test Leader and Tester (K1)

In this syllabus two test positions are covered, test leader and tester. The activities and tasks performed by people in these two roles depend on the project and product context, the people in the roles, and the organization.

Sometimes the test leader is called a test manager or test coordinator. The role of the test leader may be performed by a project manager, a development manager, a quality assurance manager or the manager of a test group. In larger projects two positions may exist: test leader and test manager. Typically the test leader plans, monitors and controls the testing activities and tasks as defined in Section 1.4.

Typical test leader tasks may include:
- Coordinate the test strategy and plan with project managers and others
- Write or review a test strategy for the project, and test policy for the organization

- Contribute the testing perspective to other project activities, such as integration planning
- Plan the tests – considering the context and understanding the test objectives and risks – including selecting test approaches, estimating the time, effort and cost of testing, acquiring resources, defining test levels, cycles, and planning incident management
- Initiate the specification, preparation, implementation and execution of tests, monitor the test results and check the exit criteria
- Adapt planning based on test results and progress (sometimes documented in status reports) and take any action necessary to compensate for problems
- Set up adequate configuration management of testware for traceability
- Introduce suitable metrics for measuring test progress and evaluating the quality of the testing and the product
- Decide what should be automated, to what degree, and how
- Select tools to support testing and organize any training in tool use for testers
- Decide about the implementation of the test environment
- Write test summary reports based on the information gathered during testing

Typical tester tasks may include:

- Review and contribute to test plans
- Analyze, review and assess user requirements, specifications and models for testability
- Create test specifications
- Set up the test environment (often coordinating with system administration and network management)
- Prepare and acquire test data
- Implement tests on all test levels, execute and log the tests, evaluate the results and document the deviations from expected results
- Use test administration or management tools and test monitoring tools as required
- Automate tests (may be supported by a developer or a test automation expert)
- Measure performance of components and systems (if applicable)
- Review tests developed by others

People who work on test analysis, test design, specific test types or test automation may be specialists in these roles. Depending on the test level and the risks related to the product and the project, different people may take over the role of tester, keeping some degree of independence. Typically testers at the component and integration level would be developers, testers at the acceptance test level would be business experts and users, and testers for operational acceptance testing would be operators.

5.2 Test Planning and Estimation (K3) — *40 minutes*

Terms
Test approach, test strategy

5.2.1 Test Planning (K2)

This section covers the purpose of test planning within development and implementation projects, and for maintenance activities. Planning may be documented in a master test plan and in separate test plans for test levels such as system testing and acceptance testing. The outline of a test-planning document is covered by the 'Standard for Software Test Documentation' (IEEE Std 829-1998).

Planning is influenced by the test policy of the organization, the scope of testing, objectives, risks, constraints, criticality, testability and the availability of resources. As the project and test planning progress, more information becomes available and more detail can be included in the plan.

Test planning is a continuous activity and is performed in all life cycle processes and activities. Feedback from test activities is used to recognize changing risks so that planning can be adjusted.

5.2.2 Test Planning Activities (K3)

Test planning activities for an entire system or part of a system may include:
- Determining the scope and risks and identifying the objectives of testing
- Defining the overall approach of testing, including the definition of the test levels and entry and exit criteria
- Integrating and coordinating the testing activities into the software life cycle activities (acquisition, supply, development, operation and maintenance)
- Making decisions about what to test, what roles will perform the test activities, how the test activities should be done, and how the test results will be evaluated
- Scheduling test analysis and design activities
- Scheduling test implementation, execution and evaluation
- Assigning resources for the different activities defined
- Defining the amount, level of detail, structure and templates for the test documentation
- Selecting metrics for monitoring and controlling test preparation and execution, defect resolution and risk issues
- Setting the level of detail for test procedures in order to provide enough information to support reproducible test preparation and execution

5.2.3 Entry Criteria (K2)

Entry criteria define when to start testing such as at the beginning of a test level or when a set of tests is ready for execution.

Typically entry criteria may cover the following:
- Test environment availability and readiness
- Test tool readiness in the test environment
- Testable code availability
- Test data availability

5.2.4 Exit Criteria (K2)

Exit criteria define when to stop testing such as at the end of a test level or when a set of tests has achieved specific goal.

Typically exit criteria may cover the following:
- Thoroughness measures, such as coverage of code, functionality or risk
- Estimates of defect density or reliability measures
- Cost
- Residual risks, such as defects not fixed or lack of test coverage in certain areas
- Schedules such as those based on time to market

5.2.5 Test Estimation (K2)

Two approaches for the estimation of test effort are:
- The metrics-based approach: estimating the testing effort based on metrics of former or similar projects or based on typical values
- The expert-based approach: estimating the tasks based on estimates made by the owner of the tasks or by experts

Once the test effort is estimated, resources can be identified and a schedule can be drawn up.

The testing effort may depend on a number of factors, including:
- Characteristics of the product: the quality of the specification and other information used for test models (i.e., the test basis), the size of the product, the complexity of the problem domain, the requirements for reliability and security, and the requirements for documentation
- Characteristics of the development process: the stability of the organization, tools used, test process, skills of the people involved, and time pressure
- The outcome of testing: the number of defects and the amount of rework required

5.2.6 Test Strategy, Test Approach (K2)

The test approach is the implementation of the test strategy for a specific project. The test approach is defined and refined in the test plans and test designs. It typically includes the decisions made based on the (test) project's goal and risk assessment. It is the starting point for planning the test process, for selecting the test design techniques and test types to be applied, and for defining the entry and exit criteria.

The selected approach depends on the context and may consider risks, hazards and safety, available resources and skills, the technology, the nature of the system (e.g., custom built vs. COTS), test objectives, and regulations.

Typical approaches include:
- Analytical approaches, such as risk-based testing where testing is directed to areas of greatest risk
- Model-based approaches, such as stochastic testing using statistical information about failure rates (such as reliability growth models) or usage (such as operational profiles)
- Methodical approaches, such as failure-based (including error guessing and fault attacks), experience-based, checklist-based, and quality characteristic-based
- Process- or standard-compliant approaches, such as those specified by industry-specific standards or the various agile methodologies
- Dynamic and heuristic approaches, such as exploratory testing where testing is more reactive to events than pre-planned, and where execution and evaluation are concurrent tasks
- Consultative approaches, such as those in which test coverage is driven primarily by the advice and guidance of technology and/or business domain experts outside the test team
- Regression-averse approaches, such as those that include reuse of existing test material, extensive automation of functional regression tests, and standard test suites

Different approaches may be combined, for example, a risk-based dynamic approach.

5.3 Test Progress Monitoring and Control (K2)	*20 minutes*

Terms
Defect density, failure rate, test control, test monitoring, test summary report

5.3.1 Test Progress Monitoring (K1)

The purpose of test monitoring is to provide feedback and visibility about test activities. Information to be monitored may be collected manually or automatically and may be used to measure exit criteria, such as coverage. Metrics may also be used to assess progress against the planned schedule and budget. Common test metrics include:

- Percentage of work done in test case preparation (or percentage of planned test cases prepared)
- Percentage of work done in test environment preparation
- Test case execution (e.g., number of test cases run/not run, and test cases passed/failed)
- Defect information (e.g., defect density, defects found and fixed, failure rate, and re-test results)
- Test coverage of requirements, risks or code
- Subjective confidence of testers in the product
- Dates of test milestones
- Testing costs, including the cost compared to the benefit of finding the next defect or to run the next test

5.3.2 Test Reporting (K2)

Test reporting is concerned with summarizing information about the testing endeavor, including:

- What happened during a period of testing, such as dates when exit criteria were met
- Analyzed information and metrics to support recommendations and decisions about future actions, such as an assessment of defects remaining, the economic benefit of continued testing, outstanding risks, and the level of confidence in the tested software

The outline of a test summary report is given in 'Standard for Software Test Documentation' (IEEE Std 829-1998).

Metrics should be collected during and at the end of a test level in order to assess:

- The adequacy of the test objectives for that test level
- The adequacy of the test approaches taken
- The effectiveness of the testing with respect to the objectives

5.3.3 Test Control (K2)

Test control describes any guiding or corrective actions taken as a result of information and metrics gathered and reported. Actions may cover any test activity and may affect any other software life cycle activity or task.

Examples of test control actions include:

- Making decisions based on information from test monitoring
- Re-prioritizing tests when an identified risk occurs (e.g., software delivered late)
- Changing the test schedule due to availability or unavailability of a test environment
- Setting an entry criterion requiring fixes to have been re-tested (confirmation tested) by a developer before accepting them into a build

5.4 Configuration Management (K2) — *10 minutes*

Terms
Configuration management, version control

Background
The purpose of configuration management is to establish and maintain the integrity of the products (components, data and documentation) of the software or system through the project and product life cycle.

For testing, configuration management may involve ensuring the following:
- All items of testware are identified, version controlled, tracked for changes, related to each other and related to development items (test objects) so that traceability can be maintained throughout the test process
- All identified documents and software items are referenced unambiguously in test documentation

For the tester, configuration management helps to uniquely identify (and to reproduce) the tested item, test documents, the tests and the test harness(es).

During test planning, the configuration management procedures and infrastructure (tools) should be chosen, documented and implemented.

5.5 Risk and Testing (K2) — *30 minutes*

Terms
Product risk, project risk, risk, risk-based testing

Background
Risk can be defined as the chance of an event, hazard, threat or situation occurring and resulting in undesirable consequences or a potential problem. The level of risk will be determined by the likelihood of an adverse event happening and the impact (the harm resulting from that event).

5.5.1 Project Risks (K2)

Project risks are the risks that surround the project's capability to deliver its objectives, such as:

- Organizational factors:
  - Skill, training and staff shortages
  - Personnel issues
  - Political issues, such as:
    - Problems with testers communicating their needs and test results
    - Failure by the team to follow up on information found in testing and reviews (e.g., not improving development and testing practices)
  - Improper attitude toward or expectations of testing (e.g., not appreciating the value of finding defects during testing)
- Technical issues:
  - Problems in defining the right requirements
  - The extent to which requirements cannot be met given existing constraints
  - Test environment not ready on time
  - Late data conversion, migration planning and development and testing data conversion/migration tools
  - Low quality of the design, code, configuration data, test data and tests
- Supplier issues:
  - Failure of a third party
  - Contractual issues

When analyzing, managing and mitigating these risks, the test manager is following well-established project management principles. The 'Standard for Software Test Documentation' (IEEE Std 829-1998) outline for test plans requires risks and contingencies to be stated.

5.5.2 Product Risks (K2)

Potential failure areas (adverse future events or hazards) in the software or system are known as product risks, as they are a risk to the quality of the product. These include:

- Failure-prone software delivered
- The potential that the software/hardware could cause harm to an individual or company
- Poor software characteristics (e.g., functionality, reliability, usability and performance)
- Poor data integrity and quality (e.g., data migration issues, data conversion problems, data transport problems, violation of data standards)
- Software that does not perform its intended functions

Risks are used to decide where to start testing and where to test more; testing is used to reduce the risk of an adverse effect occurring, or to reduce the impact of an adverse effect.

Product risks are a special type of risk to the success of a project. Testing as a risk-control activity provides feedback about the residual risk by measuring the effectiveness of critical defect removal and of contingency plans.

A risk-based approach to testing provides proactive opportunities to reduce the levels of product risk, starting in the initial stages of a project. It involves the identification of product risks and their use in guiding test planning and control, specification, preparation and execution of tests. In a risk-based approach the risks identified may be used to:

- Determine the test techniques to be employed
- Determine the extent of testing to be carried out
- Prioritize testing in an attempt to find the critical defects as early as possible
- Determine whether any non-testing activities could be employed to reduce risk (e.g., providing training to inexperienced designers)

Risk-based testing draws on the collective knowledge and insight of the project stakeholders to determine the risks and the levels of testing required to address those risks.

To ensure that the chance of a product failure is minimized, risk management activities provide a disciplined approach to:

- Assess (and reassess on a regular basis) what can go wrong (risks)
- Determine what risks are important to deal with
- Implement actions to deal with those risks

In addition, testing may support the identification of new risks, may help to determine what risks should be reduced, and may lower uncertainty about risks.

5.6 Incident Management (K3) — *40 minutes*

Terms
Incident logging, incident management, incident report

Background
Since one of the objectives of testing is to find defects, the discrepancies between actual and expected outcomes need to be logged as incidents. An incident must be investigated and may turn out to be a defect. Appropriate actions to dispose incidents and defects should be defined. Incidents and defects should be tracked from discovery and classification to correction and confirmation of the solution. In order to manage all incidents to completion, an organization should establish an incident management process and rules for classification.

Incidents may be raised during development, review, testing or use of a software product. They may be raised for issues in code or the working system, or in any type of documentation including requirements, development documents, test documents, and user information such as “Help” or installation guides.

Incident reports have the following objectives:
- Provide developers and other parties with feedback about the problem to enable identification, isolation and correction as necessary
- Provide test leaders a means of tracking the quality of the system under test and the progress of the testing
- Provide ideas for test process improvement

Details of the incident report may include:
- Date of issue, issuing organization, and author
- Expected and actual results
- Identification of the test item (configuration item) and environment
- Software or system life cycle process in which the incident was observed
- Description of the incident to enable reproduction and resolution, including logs, database dumps or screenshots
- Scope or degree of impact on stakeholder(s) interests
- Severity of the impact on the system
- Urgency/priority to fix
- Status of the incident (e.g., open, deferred, duplicate, waiting to be fixed, fixed awaiting re-test, closed)
- Conclusions, recommendations and approvals
- Global issues, such as other areas that may be affected by a change resulting from the incident
- Change history, such as the sequence of actions taken by project team members with respect to the incident to isolate, repair, and confirm it as fixed
- References, including the identity of the test case specification that revealed the problem

The structure of an incident report is also covered in the ‘Standard for Software Test Documentation’ (IEEE Std 829-1998).

6.1 Types of Test Tools (K2) — *45 minutes*

Terms
Configuration management tool, coverage tool, debugging tool, dynamic analysis tool, incident management tool, load testing tool, modeling tool, monitoring tool, performance testing tool, probe effect, requirements management tool, review tool, security tool, static analysis tool, stress testing tool, test comparator, test data preparation tool, test design tool, test harness, test execution tool, test management tool, unit test framework tool

6.1.1 Tool Support for Testing (K2)

Test tools can be used for one or more activities that support testing. These include:
1. Tools that are directly used in testing such as test execution tools, test data generation tools and result comparison tools
2. Tools that help in managing the testing process such as those used to manage tests, test results, data, requirements, incidents, defects, etc., and for reporting and monitoring test execution
3. Tools that are used in reconnaissance, or, in simple terms: exploration (e.g., tools that monitor file activity for an application)
4. Any tool that aids in testing (a spreadsheet is also a test tool in this meaning)

Tool support for testing can have one or more of the following purposes depending on the context:
- Improve the efficiency of test activities by automating repetitive tasks or supporting manual test activities like test planning, test design, test reporting and monitoring
- Automate activities that require significant resources when done manually (e.g., static testing)
- Automate activities that cannot be executed manually (e.g., large scale performance testing of client-server applications)
- Increase reliability of testing (e.g., by automating large data comparisons or simulating behavior)

The term “test frameworks” is also frequently used in the industry, in at least three meanings:
- Reusable and extensible testing libraries that can be used to build testing tools (called test harnesses as well)
- A type of design of test automation (e.g., data-driven, keyword-driven)
- Overall process of execution of testing

For the purpose of this syllabus, the term “test frameworks” is used in its first two meanings as described in Section 6.1.6.

6.1.2 Test Tool Classification (K2)

There are a number of tools that support different aspects of testing. Tools can be classified based on several criteria such as purpose, commercial / free / open-source / shareware, technology used and so forth. Tools are classified in this syllabus according to the testing activities that they support.

Some tools clearly support one activity; others may support more than one activity, but are classified under the activity with which they are most closely associated. Tools from a single provider, especially those that have been designed to work together, may be bundled into one package.

Some types of test tools can be intrusive, which means that they can affect the actual outcome of the test. For example, the actual timing may be different due to the extra instructions that are executed by the tool, or you may get a different measure of code coverage. The consequence of intrusive tools is called the probe effect.

Some tools offer support more appropriate for developers (e.g., tools that are used during component and component integration testing). Such tools are marked with "(D)" in the list below.

6.1.3 Tool Support for Management of Testing and Tests (K1)

Management tools apply to all test activities over the entire software life cycle.

Test Management Tools
These tools provide interfaces for executing tests, tracking defects and managing requirements, along with support for quantitative analysis and reporting of the test objects. They also support tracing the test objects to requirement specifications and might have an independent version control capability or an interface to an external one.

Requirements Management Tools
These tools store requirement statements, store the attributes for the requirements (including priority), provide unique identifiers and support tracing the requirements to individual tests. These tools may also help with identifying inconsistent or missing requirements.

Incident Management Tools (Defect Tracking Tools)
These tools store and manage incident reports, i.e., defects, failures, change requests or perceived problems and anomalies, and help in managing the life cycle of incidents, optionally with support for statistical analysis.

Configuration Management Tools
Although not strictly test tools, these are necessary for storage and version management of testware and related software especially when configuring more than one hardware/software environment in terms of operating system versions, compilers, browsers, etc.

6.1.4 Tool Support for Static Testing (K1)

Static testing tools provide a cost effective way of finding more defects at an earlier stage in the development process.

Review Tools
These tools assist with review processes, checklists, review guidelines and are used to store and communicate review comments and report on defects and effort. They can be of further help by providing aid for online reviews for large or geographically dispersed teams.

Static Analysis Tools (D)
These tools help developers and testers find defects prior to dynamic testing by providing support for enforcing coding standards (including secure coding), analysis of structures and dependencies. They can also help in planning or risk analysis by providing metrics for the code (e.g., complexity).

Modeling Tools (D)
These tools are used to validate software models (e.g., physical data model (PDM) for a relational database), by enumerating inconsistencies and finding defects. These tools can often aid in generating some test cases based on the model.

6.1.5 Tool Support for Test Specification (K1)

Test Design Tools
These tools are used to generate test inputs or executable tests and/or test oracles from requirements, graphical user interfaces, design models (state, data or object) or code.

Test Data Preparation Tools
Test data preparation tools manipulate databases, files or data transmissions to set up test data to be used during the execution of tests to ensure security through data anonymity.

6.1.6 Tool Support for Test Execution and Logging (K1)

Test Execution Tools
These tools enable tests to be executed automatically, or semi-automatically, using stored inputs and expected outcomes, through the use of a scripting language and usually provide a test log for each test run. They can also be used to record tests, and usually support scripting languages or GUI-based configuration for parameterization of data and other customization in the tests.

Test Harness/Unit Test Framework Tools (D)
A unit test harness or framework facilitates the testing of components or parts of a system by simulating the environment in which that test object will run, through the provision of mock objects as stubs or drivers.

Test Comparators
Test comparators determine differences between files, databases or test results. Test execution tools typically include dynamic comparators, but post-execution comparison may be done by a separate comparison tool. A test comparator may use a test oracle, especially if it is automated.

Coverage Measurement Tools (D)
These tools, through intrusive or non-intrusive means, measure the percentage of specific types of code structures that have been exercised (e.g., statements, branches or decisions, and module or function calls) by a set of tests.

Security Testing Tools
These tools are used to evaluate the security characteristics of software. This includes evaluating the ability of the software to protect data confidentiality, integrity, authentication, authorization, availability, and non-repudiation. Security tools are mostly focused on a particular technology, platform, and purpose.

6.1.7 Tool Support for Performance and Monitoring (K1)

Dynamic Analysis Tools (D)
Dynamic analysis tools find defects that are evident only when software is executing, such as time dependencies or memory leaks. They are typically used in component and component integration testing, and when testing middleware.

Performance Testing/Load Testing/Stress Testing Tools
Performance testing tools monitor and report on how a system behaves under a variety of simulated usage conditions in terms of number of concurrent users, their ramp-up pattern, frequency and relative percentage of transactions. The simulation of load is achieved by means of creating virtual users carrying out a selected set of transactions, spread across various test machines commonly known as load generators.

Monitoring Tools
Monitoring tools continuously analyze, verify and report on usage of specific system resources, and give warnings of possible service problems.

6.1.8 Tool Support for Specific Testing Needs (K1)

Data Quality Assessment
Data is at the center of some projects such as data conversion/migration projects and applications like data warehouses and its attributes can vary in terms of criticality and volume. In such contexts, tools need to be employed for data quality assessment to review and verify the data conversion and

migration rules to ensure that the processed data is correct, complete and complies with a pre-defined context-specific standard.

Other testing tools exist for usability testing.

6.2 Effective Use of Tools: Potential Benefits and Risks (K2) — *20 minutes*

Terms
Data-driven testing, keyword-driven testing, scripting language

6.2.1 Potential Benefits and Risks of Tool Support for Testing (for all tools) (K2)

Simply purchasing or leasing a tool does not guarantee success with that tool. Each type of tool may require additional effort to achieve real and lasting benefits. There are potential benefits and opportunities with the use of tools in testing, but there are also risks.

Potential benefits of using tools include:
- Repetitive work is reduced (e.g., running regression tests, re-entering the same test data, and checking against coding standards)
- Greater consistency and repeatability (e.g., tests executed by a tool in the same order with the same frequency, and tests derived from requirements)
- Objective assessment (e.g., static measures, coverage)
- Ease of access to information about tests or testing (e.g., statistics and graphs about test progress, incident rates and performance)

Risks of using tools include:
- Unrealistic expectations for the tool (including functionality and ease of use)
- Underestimating the time, cost and effort for the initial introduction of a tool (including training and external expertise)
- Underestimating the time and effort needed to achieve significant and continuing benefits from the tool (including the need for changes in the testing process and continuous improvement of the way the tool is used)
- Underestimating the effort required to maintain the test assets generated by the tool
- Over-reliance on the tool (replacement for test design or use of automated testing where manual testing would be better)
- Neglecting version control of test assets within the tool
- Neglecting relationships and interoperability issues between critical tools, such as requirements management tools, version control tools, incident management tools, defect tracking tools and tools from multiple vendors
- Risk of tool vendor going out of business, retiring the tool, or selling the tool to a different vendor
- Poor response from vendor for support, upgrades, and defect fixes
- Risk of suspension of open-source / free tool project
- Unforeseen, such as the inability to support a new platform

6.2.2 Special Considerations for Some Types of Tools (K1)

Test Execution Tools
Test execution tools execute test objects using automated test scripts. This type of tool often requires significant effort in order to achieve significant benefits.

Capturing tests by recording the actions of a manual tester seems attractive, but this approach does not scale to large numbers of automated test scripts. A captured script is a linear representation with specific data and actions as part of each script. This type of script may be unstable when unexpected events occur.

A data-driven testing approach separates out the test inputs (the data), usually into a spreadsheet, and uses a more generic test script that can read the input data and execute the same test script with different data. Testers who are not familiar with the scripting language can then create the test data for these predefined scripts.

There are other techniques employed in data-driven techniques, where instead of hard-coded data combinations placed in a spreadsheet, data is generated using algorithms based on configurable parameters at run time and supplied to the application. For example, a tool may use an algorithm, which generates a random user ID, and for repeatability in pattern, a seed is employed for controlling randomness.

In a keyword-driven testing approach, the spreadsheet contains keywords describing the actions to be taken (also called action words), and test data. Testers (even if they are not familiar with the scripting language) can then define tests using the keywords, which can be tailored to the application being tested.

Technical expertise in the scripting language is needed for all approaches (either by testers or by specialists in test automation).

Regardless of the scripting technique used, the expected results for each test need to be stored for later comparison.

Static Analysis Tools
Static analysis tools applied to source code can enforce coding standards, but if applied to existing code may generate a large quantity of messages. Warning messages do not stop the code from being translated into an executable program, but ideally should be addressed so that maintenance of the code is easier in the future. A gradual implementation of the analysis tool with initial filters to exclude some messages is an effective approach.

Test Management Tools
Test management tools need to interface with other tools or spreadsheets in order to produce useful information in a format that fits the needs of the organization.

6.3 Introducing a Tool into an Organization (K1)	*15 minutes*

Terms
No specific terms.

Background
The main considerations in selecting a tool for an organization include:
- Assessment of organizational maturity, strengths and weaknesses and identification of opportunities for an improved test process supported by tools
- Evaluation against clear requirements and objective criteria
- A proof-of-concept, by using a test tool during the evaluation phase to establish whether it performs effectively with the software under test and within the current infrastructure or to identify changes needed to that infrastructure to effectively use the tool
- Evaluation of the vendor (including training, support and commercial aspects) or service support suppliers in case of non-commercial tools
- Identification of internal requirements for coaching and mentoring in the use of the tool
- Evaluation of training needs considering the current test team's test automation skills
- Estimation of a cost-benefit ratio based on a concrete business case

Introducing the selected tool into an organization starts with a pilot project, which has the following objectives:
- Learn more detail about the tool
- Evaluate how the tool fits with existing processes and practices, and determine what would need to change
- Decide on standard ways of using, managing, storing and maintaining the tool and the test assets (e.g., deciding on naming conventions for files and tests, creating libraries and defining the modularity of test suites)
- Assess whether the benefits will be achieved at reasonable cost

Success factors for the deployment of the tool within an organization include:
- Rolling out the tool to the rest of the organization incrementally
- Adapting and improving processes to fit with the use of the tool
- Providing training and coaching/mentoring for new users
- Defining usage guidelines
- Implementing a way to gather usage information from the actual use
- Monitoring tool use and benefits
- Providing support for the test team for a given tool
- Gathering lessons learned from all teams

References
6.2.2 Buwalda, 2001, Fewster, 1999
6.3 Fewster, 1999

Chapter 9 : Testing Web and Mobile Application

Written by Roshan Chitrakar

Types of Web Application

1. Static Web Application : HTML, CSS, jQuery, Ajax
   banners, GIFs, videos etc.

&

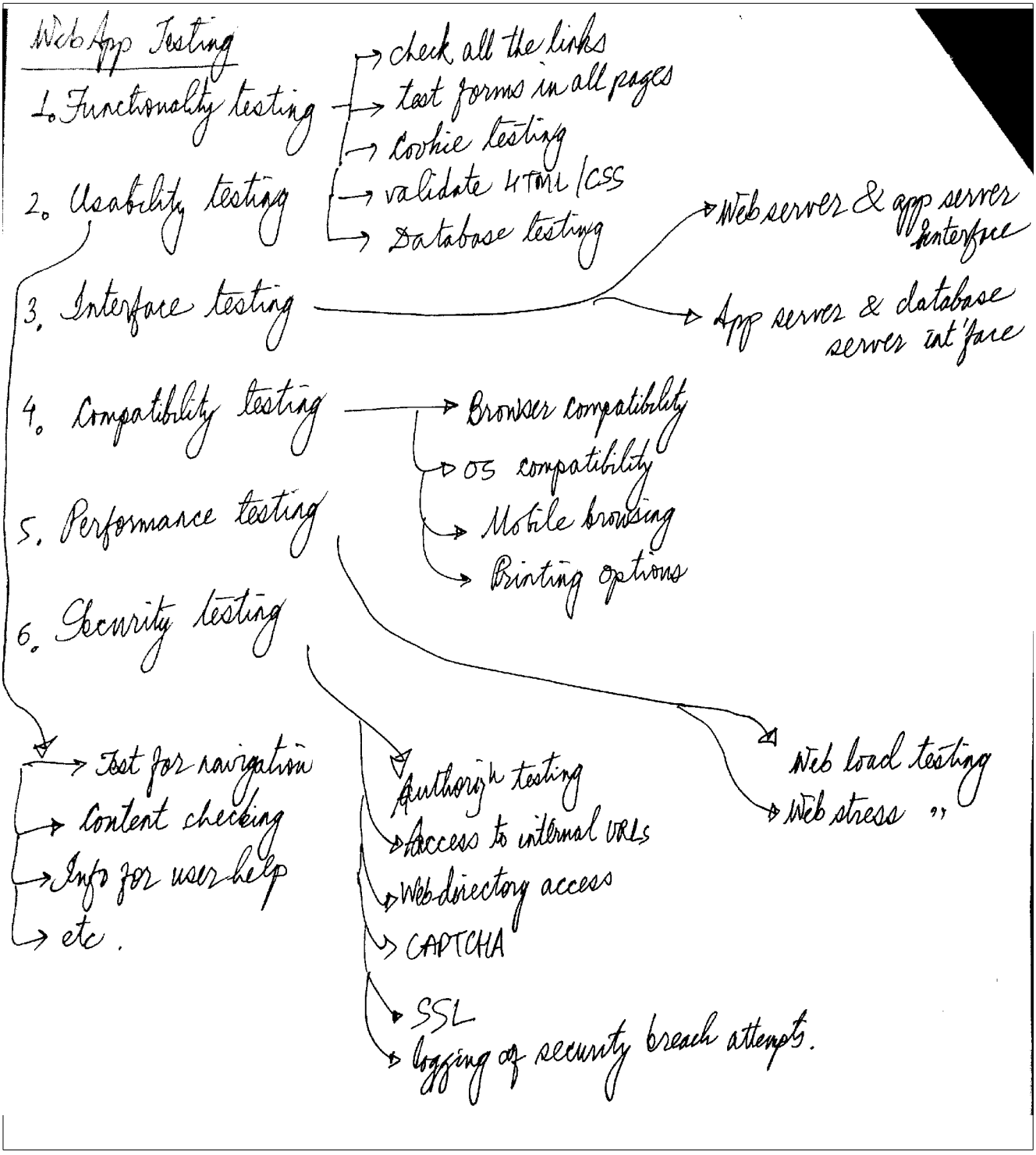
2. Dynamic Web Application : Databases, CMS (admin panel), PHP, ASP
3. Shop-online / E-Commerce : m-commerce, e-payment
4. Portal Web App : access various services / categories through home page.
5. Animated Web App : Flash, modern design
6. Web App with CMS : WordPress, Joomla, Drupal etc.

Development viewpoint

Web App categories

1. Document centric (static homepage, web radio, company website)
2. Interactive (virtual exhibition, news site, travel planning)
3. Transactional (online banking, shopping, booking system)
4. Workflow based (e-government, B2B solution)
5. Collaborative (chat room, E-learning platform, P2P-services)
6. Portal oriented (community portal, online shopping mall etc.)
7. Ubiquitous (customized services, location aware services, multi-platform delivery)
8. Semantic (knowledge mgmt, syndication, recommender)
9. Social web (web logs, collaborative filtering, virtual shared workplace)

Activities / functions perspective


Web App Testing
1. Functionality testing
check all the links
test forms in all pages
Cookie testing
validate HTML/CSS
Database testing
2. Usability testing
3. Interface testing
Web server & app server interface
App server & database server int'face
4. Compatibility testing
Browser compatibility
OS compatibility
Mobile browsing
Printing options
5. Performance testing
Web load testing
Web stress "
6. Security testing
Authorization testing
Access to internal URLs
Web directory access
CAPTCHA
SSL
logging of security breach attempts.
Test for navigation
Content checking
Info for user help
etc.

Mobile App testing (Mobile App. Testing)

Types in general:

1. Hardwar testing (mobile testing)
2. SW " (mobile app ")
   - Native app
   - Mobile Web app
   - Hybrid app

Categories of testing

1. Usability testing
2. Performance "
3. Installation "
4. Security "

Testing strategy

1. Selection of devices
2. Emulators → Device emulators eg. iPhone Tester, Mobilephone emulatorste.
   - Browser "
   - OS "
   - etc.
3. physical devices (testing on)
4. cloud computing based testing
5. Automation vs. Manual " → object based automation tool, image based "
6. Network configuration

Test cases

1. Battery usage
2. Speed of applic"
3. Data requirements
4. Memory "
5. Functionality of applic"

Chapter 10 : Quality Management

- Compliled by Roshan Chitrakar

Software Quality Definition

The Software Quality conforms to some explicit characteristics as well as implicit characteristics.

These are:-

- **Explicit Characteristics:**

  The software should explicitly conform to the stated functional and performance requirements. The software should conform explicitly as per the documented development standards

- **Implicit Characteristics:**

  The implicit characteristics of software quality are those expected from all professionally developed software.

This above definition of software quality emphasizes the following three points.

1. Software requirements are the basic foundations from which quality of software is measured. Lack of conformance to the requirement is lack of quality.
2. Software Development criteria comprise of a set of specified standards that guide the manner in which the software is engineered. If the criteria are not followed, software quality will definitely be lost.
3. There is a set of implicit requirements like scope and desire for good maintainability, even if not mentioned, matter a lot towards the software quality.

   If software does not conform to the implicit requirements, no matter how strongly it meets the explicit requirements, the quality of the software will be suspected.

Quality Concepts

Variation control is the heart of quality control (software engineers strive to control the process applied, resources expended, and end product quality attributes). The tape duplication example.

Quality of design - refers to characteristics designers specify for the end product to be constructed. The grade of materials, tolerances, and performance specifications all contribute to the quality of design.

Quality of conformance - degree to which design specifications are followed in manufacturing the product. If the implementation follows the design and the resulting system meets its requirements and performance goals, conformance quality is high.

Quality control - series of inspections, reviews, and tests used to ensure conformance of a work product to its specifications. Quality control includes a feedback loop to the process that created the work product.

Quality assurance - consists of the auditing and reporting procedures used to provide management with data needed to make proactive (practical and positive) decisions.

Cost of Quality – It includes all the costs incurred in the pursuit of quality or in performing quality related activities. It provides a baseline for the current cost of quality, identify opportunities for reducing the cost of quality, and provide a normalized basis of comparison. Cost of quality includes:

- **Prevention costs** - Includes quality planning, formal technical reviews, test equipment, training.
- **Appraisal costs** – Includes in-process and inter-process inspection, equipment calibration and maintenance, testing. (Appraisal means evaluation or assessment)
- **Failure costs** – Includes rework, repair, and failure mode analysis.
- **External failure costs** – Includes complaint resolution, product return and replacement, help line support, warranty work.

Software Quality Assurance

Conformance to software requirements is the foundation from which software quality is measured. Specified standards are used to define the development criteria that are used to guide the manner in which software is engineered. Software must conform to implicit requirements (ease of use, maintainability, reliability, etc.) as well as its explicit requirements.

The three important point of concern for SQA are:

- Software requirements are the foundation from which quality is measured. Lack of conformance to requirements is lack of quality.
- Specific standards define a set of development criteria that guide the manner in which software is engineered. If the criteria are not followed, lack of quality will almost surely result.

- A set of implicit requirements often goes unmentioned e.g., the desire for ease of use and good maintainability. If the software conforms to its explicit requirements but fails to meet implicit requirements, software quality is suspected.

SQA Group Activities

- Prepare SQA plan for the project.
- Participate in the development of the project's software process description.
- Review software engineering activities to verify compliance with the defined software process.
- Audit designated software work products to verify compliance with those defined as part of the software process.
- Ensure that any deviations in software or work products are documented and handled according to a documented procedure.
- Record any evidence of noncompliance and reports them to management.

Software Reviews

The purpose is to find defects (errors) before they are passed on to another software engineering activity or released to the customer. Software engineers (and others) conduct formal technical reviews (FTR) for software engineers. Using formal technical reviews (walkthroughs or inspections) is an effective means for improving software quality.

Formal Technical Reviews

- Involves 3 to 5 people (including reviewers)
- Advance preparation (no more than 2 hours per person) required
- Duration of review meeting should be less than 2 hours
- Focus of review is on a discrete work product
- Review leader organizes the review meeting at the producer's request
- Reviewers ask questions that enable the producer to discover his or her own error (the product is under review not the producer)
- Producer of the work product walks the reviewers through the product
- Recorder writes down any significant issues raised during the review

- Reviewers decide to accept or reject the work product and whether to require additional reviews of product or not

For software industry to be more quantitative about quality, the statistical quality assurance implies the following steps:

1. Information about the software defects is collected and categorized.
2. An attempt is made to trace each defect to its underlying cause.

   [e.g. non-conformance to specification, design error, validation of standards, poor communication with customer, etc.]
3. Using the Pareto principle, isolate the 20% of the most vital errors.
4. Now, move to correct the problems that have caused the defects.

In view of the above steps, during software development cycle, various errors encountered can be attached to one of more of the following causes: -

- IES = Incomplete or Erroneous specification
- MCC= Misinterpretation of Customer Communication.
- IDS = Intentional Deviation from Specification.
- VPS = Violation of Programming Standards.
- EDR= Error in Data Representation.
- IMI= Inconsistent Module Interface
- EDL = Error in Design Logic
- IET = Incomplete or Erroneous Testing
- IID = Inaccurate or Incomplete Documentation
- PLT=error in Programming Language Translation of design
- HCI = ambiguous or inconsistent Human-Computer Interface.
- MIS= Miscellaneous

It is seen that the causes IES, MCC, EDR and IET account for most of the major errors. Once the vital few causes are located, the software development origination can plan for necessary corrective actions.

In addition to the collection of defect information, software developer can calculate Defect Index (DI) for each major step in the software development process. After analysis, design, coding, testing and release, the following data are gathered: -

Di = total number of defects encountered during i^{th} please of the software engineering process.

Si = number of serious defects

Mi = number of moderate defects

Ti= number of minor (trivial) defects.

PS= Size of the product (LOC, design statements, pages of documentation) at i^{th} step.

wi = Weighting factor for serious, moderate, and trivial defects, where j= 1 to 3

At each step of the software development t process, a phase index PI_i is competed, as follows: -

PI = w1(si/Di) + w2(Mi/Di) + w3(Ti/Di)

Finally, the defect index DI_i is calculated by computing the cumulative effect of each PI_i, giving higher weights to the errors encountered later in the software engineering process than those encountered earlier.

$DI= \sum(i*PI_i)/PS$

$= (PI_1+2PI_2+3PI_3+.....+ iPI_i)/PS$

Software Reliability

The reliability of a computer program is an important element of its overall quality. If a program repeatedly and frequently fails to perform, it matters little whether other software quality factors are accepted.

Software reliability can be measured directly and estimated using historical and developmental data. Software reliability is defined in statistical terms as "The probability of failure free operation of a computer program in a specified environment for a specified time."

Software Reliability Index

In context of a computer-based system, a simple measure of the software reliability is Mean Time Between Failure (MTBF),

MTBF= MTTF + MTTR

Where, MTTF = Mean Time To Failure,

MTTR = Mean Time to Recover/Repair

In addition to the reliability measure, a measure of the availability is also important. Software reliability is defined as “The probability that the program is operating according to requirements at a given point of time"

Software Availability is given by:-

Availability = MTTF/(MTTF+MTTR)*100%

Six Sigma for Software Engineering

Six Sigma is the most widely used strategy for statistical quality assurance in industry today. Originally popularized by Motorola in the 1980s, the Six Sigma strategy “is a rigorous and disciplined methodology that uses data and statistical analysis to measure and improve a company’s operational performance by identifying and eliminating defects’ in manufacturing and service-related processes”. The term Six Sigma is derived from six standard deviations—3.4 instances (defects) per million occurrences—implying an extremely high quality standard. The Six Sigma methodology defines three core steps:

1. Define customer requirements and deliverables and project goals via well-defined methods of customer communication.
2. Measure the existing process and its output to determine current quality performance (collect defect metrics).
3. Analyze defect metrics and determine the vital few causes.

If an existing software process is in place, but improvement is required, Six Sigma suggests two additional steps:

- Improve the process by eliminating the root causes of defects.
- Control the process to ensure that future work does not reintroduce the causes of defects.

These core and additional steps are sometimes referred to as the DMAIC (define, measure, analyze, improve, and control) method. If an organization is developing a software process (rather than improving an existing process), the core steps are augmented as follows:

- Design the process to (1) avoid the root causes of defects and (2) to meet customer requirements.
- Verify that the process model will, in fact, avoid defects and meet customer requirements.

This variation is sometimes called the DMADV (define, measure, analyze, design, and verify) method.

Measuring Software Quality

Software Quality is a complex mix of functions that vary according to the following: -

- Applications requirements, and
- Customer's request of requirements

The following two categories of measurable factors affect the software quality:

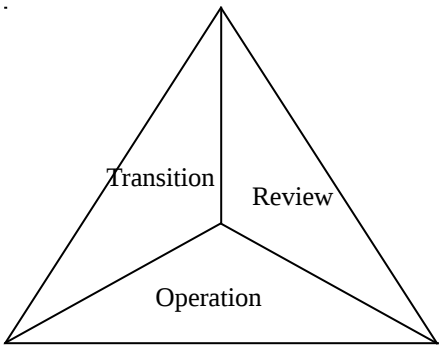
- Factors that can be directly measured: -
    1. Errors
    2. KLOC
    3. Unit time
- Factors that can only be measured indirectly: -
    1. Usability
    2. Maintainability

However, in each case, measurement must be done. The software (documents, programs, and data) must be compared against some information to arrive at an indicator of the quality.

McCall Categorization of Software Quality

As per McCall, the software quality emphasizes the following three major aspects of software product. These are: -

1) Product operations - The operational characteristics of the software
2) Product transition - the ability to undergo changes
3) Product revision - the scope of adaptability to new environments



Above three aspects are all equally important. The three aspects provide the following descriptions as per McCall.

Product Operation comprises of the following descriptions: -

- Correctness - (Does the software do what the customer need?)
- Reliability - (Does the software perform the task accurately at all time?)
- Efficiency - (Does the software run on user’s hardware efficiently?)

- Integrity - (Is the functionality of the software secured in respect of unauthorized user's access?)
- Usability - (Is the software developed keeping user's convenience in view e.g. to learn, to operate, etc.?)

Product Revision aspect comprises of the following descriptions: -

- Maintainability - If any error is occurred, can the user locate it and fix it?
- Flexibility - Can the user make necessary changes to incorporate his changing requirement?
- Testability - Can the user make necessary test to ascertain the functionality?

Product Transition aspect comprises of the following descriptions: -

- Portability – Whether the user can use the software on another hardware setup?
- Reusability - Will the user be able to reuse full or part of the software?
- Inter operability - Will the user be able to interface the software with another system?

However, in most of the cases, it is impossible to develop direct measurement of the above quality factors as proposed by McCall. Therefore, a set of metrics are defined and used to develop expressions for each of the factors according to the following relationship:

$$F_q = c_1 \times m_1 + c_2 \times m_2 + \ldots\ldots\ldots\ldots\ldots + c_n \times m_n$$

where,

F_q = Software quality factor

c_n = The regression factor, and

m_n = metrics that affect the quality factor

However, the metrics defined above are not practicable to measure; rather they can only be measured subjectively. The metrics may be kept in the form of check list that is used to grade specific attributes of the software, of the basis of 10 point grading scale.

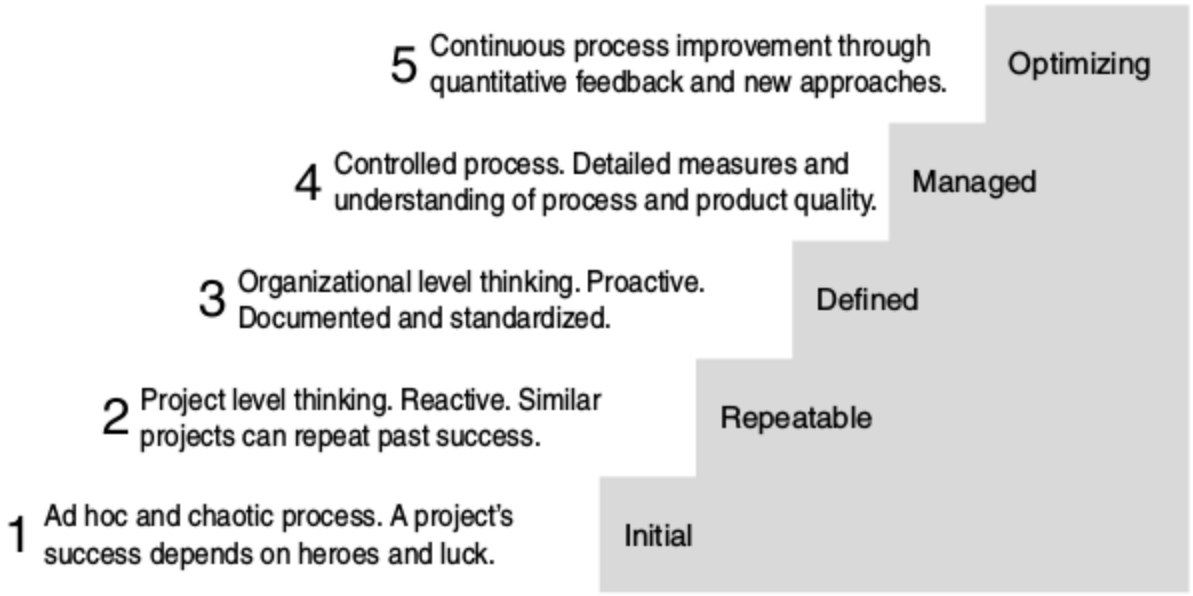
Software Quality Standards

There are different standards related to software quality. CMM and ISO are described in this section.

Capability Maturity Model (CMM)

The Capability Maturity Model for Software (CMM or SW-CMM) is an industry-standard model for defining and measuring the maturity of a software company’s development process and for providing direction on what they can do to improve their software quality. It was developed by the software

development community along with the Software Engineering Institute (SEI) and Carnegie Mellon University, under direction of the U.S. Department of Defense.



Here are descriptions of the five CMM Maturity Levels:• Level 1: Initial. The software development processes at this level are ad hoc and often chaotic. The project's success depends on heroes and luck. There are no general practices for planning, monitoring, or controlling the process. It's impossible to predict the time and cost to develop the software. The test process is just as ad hoc as the rest of the process.

• Level 2: Repeatable. This maturity level is best described as project-level thinking. Basic project management processes are in place to track the cost, schedule, functionality, and quality of the project. Lessons learned from previous similar projects are applied. There's a sense of discipline. Basic software testing practices, such as test plans and test cases, are used.

• Level 3: Defined. Organizational, not just project specific, thinking comes into play at this level. Common management and engineering activities are standardized and documented. These standards are adapted and approved for use on different projects. The rules aren't thrown out when things get stressful. Test documents and plans are reviewed and approved before testing begins. The test group is independent from the developers. The test results are used to determine when the software is ready.

• Level 4: Managed. At this maturity level, the organization's process is under statistical control. Product quality is specified quantitatively beforehand (for example, this product won't release until it has fewer than 0.5 defects per 1,000 lines of code) and the software isn't released until that goal is met. Details of the development process and the software's quality are collected over the project's development, and adjustments are made to correct deviations and to keep the project on plan.

• Level 5: Optimizing. This level is called Optimizing (not "optimized") because it's continually improving from Level 4. New technologies and processes are attempted, the results are measured, and both incremental and revolutionary changes are instituted to achieve even better quality levels. Just when everyone thinks the best has been obtained, the crank is turned one more time, and the next level of improvement is obtained.

ISO 9000

ISO is an international standards organization that sets standards for everything from nuts and bolts to, in the case of ISO 9000, quality management and quality assurance.

ISO 9000 is a family of standards on quality management and quality assurance that defines a basic set of good practices that will help a company consistently deliver products (or services) that meet their customer's quality requirements.

It's impossible to detail all the ISO 9000 requirements for software in this chapter, but the following list will give you an idea of what types of criteria the standard contains.

- Develop detailed quality plans and procedures to control configuration management, product verification and validation (testing), nonconformance (bugs), and corrective actions (fixes).
- Prepare and receive approval for a software development plan that includes a definition of the project, a list of the project's objectives, a project schedule, a product specification, a description of how the project is organized, a discussion of risks and assumptions, and strategies for controlling it.
- Communicate the specification in terms that make it easy for the customer to understand
- and to validate during testing.

- Plan, develop, document, and perform software design review procedures.
- Develop procedures that control software design changes made over the product’s life cycle.
- Develop and document software test plans.
- Develop methods to test whether the software meets the customer’s requirements.
- Perform software validation and acceptance tests.
- Maintain records of the test results.
- Control how software bugs are investigated and resolved.
- Prove that the product is ready before it’s released.
- Develop procedures to control the software’s release process.
- Identify and define what quality information should be collected.
- Use statistical techniques to analyze the software development process.
- Use statistical techniques to evaluate product quality.