Software Engineering-BSCE-301L

Module 5:

Validation and Verification

Dr . Saurabh Agrawal

Faculty Id: 20165

School of Computer Science and Engineering

VIT, Vellore-632014

Tamil Nadu, India

Outline

- Strategic Approach to Software Testing
- Testing Fundamentals
- Test Plan
- Test Design
- Test Execution
- Reviews
- Inspection and Auditing
- Regression Testing
- Mutation Testing
- Object oriented Testing
- Testing Web based System
- Mobile App Testing
- Mobile Automation Test and Tools
- DevOps Testing
- Cloud and Big Data Testing

Strategic Approach to Software Testing

❑Testing is a set of activities that can be planned in advance and conducted systematically.

❑For this reason a template for software testing a set of steps into which you can place specific test case design techniques and testing methods should be defined for the software process.

❑A number of software testing strategies have been proposed and all have the following generic characteristics:

- To perform effective testing, you should conduct effective technical reviews. By doing this, many errors will be eliminated before testing commences.
- Testing begins at the component level and works “outward” toward the integration of the entire computer-based system.
- Different testing techniques are appropriate for different software engineering approaches and at different points in time.
- Testing is conducted by the developer of the software and (for large projects) an independent test group.
- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.

Strategic Approach to Software Testing

- A strategy for software testing must accommodate low-level tests that are necessary to verify that a small source code segment has been correctly implemented as well as high-level tests that validate major system functions against customer requirements.
- A strategy should provide guidance for the practitioner and a set of milestones for the manager.
- Because the steps of the test strategy occur at a time when deadline pressure begins to rise, progress must be measurable and problems should surface as early as possible.

Strategic Approach to Software Testing

❑Verification and Validation (V&V)

❑Software testing is one element of a broader topic that is often referred to as verification and validation (V&V).

❑*Verification refers to the set of tasks that ensure that* software correctly implements a specific function.

❑*Validation refers to a different set* of tasks that ensure that the software that has been built is traceable to customer requirements.

❑Boehm [Boe81] states this another way:

- Verification: “Are we building the product right?”
- Validation: “Are we building the right product?”

Strategic Approach to Software Testing

❑Verification and Validation (V&V)

❑Verification and validation includes a wide array of Software Quality Assurance (SQA) activities: technical reviews, quality and configuration audits, performance monitoring, simulation, feasibility study, documentation review, database review, algorithm analysis, development testing, usability testing, qualification testing, acceptance testing, and installation testing.

❑Although testing plays an extremely important role in V&V, many other activities are also necessary.

❑Miller [Mil77] relates software testing to quality assurance by stating that “the underlying motivation of program testing is to affirm software quality with methods that can be economically and effectively applied to both large-scale and small-scale systems.”

Strategic Approach to Software Testing

❑Software Testing Strategy: The Big Picture

❑The software process may be viewed as the spiral illustrated in Figure 17.1. Initially, system engineering defines the role of software and leads to software requirements analysis, where the information domain, function, behavior, performance, constraints, and validation criteria for software are established.

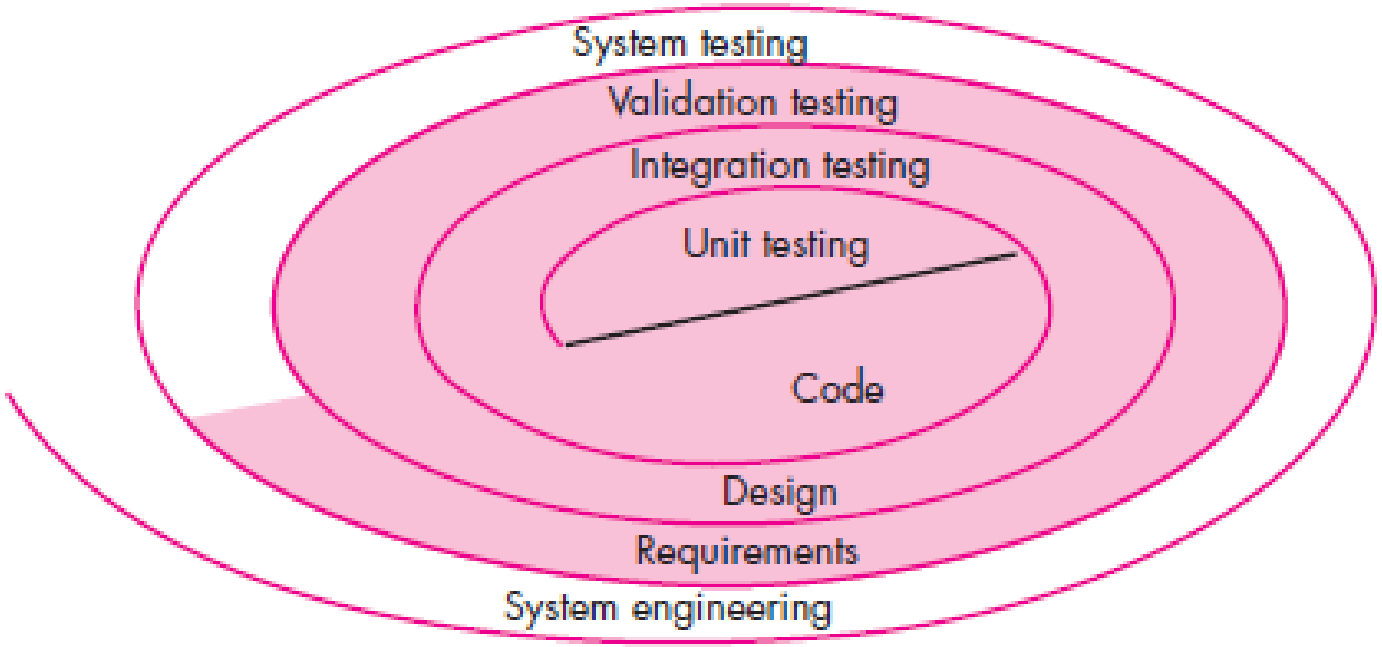
❑Moving inward along the spiral, you come to design and finally to coding.

❑To develop computer software, you spiral inward (counterclockwise) along streamlines that decrease the level of abstraction on each turn.

Strategic Approach to Software Testing

FIGURE 17.1

Testing strategy



Strategic Approach to Software Testing

❑Software Testing Strategy: The Big Picture

❑A strategy for software testing may also be viewed in the context of the spiral (Figure 17.1).

❑**Unit testing** begins at the vortex of the spiral and concentrates on each unit (e.g., component, class, or WebApp content object) of the software as implemented in source code.

❑Testing progresses by moving outward along the spiral to **integration testing**, where the focus is on design and the construction of the software architecture.

❑Taking another turn outward on the spiral, you encounter **validation testing**, where requirements established as part of requirements modeling are validated against the software that has been constructed.

❑Finally, you arrive at **system testing**, where the software and other system elements are tested as a whole.

❑To test computer software, you spiral out in a clockwise direction along streamlines that broaden the scope of testing with each turn.

Strategic Approach to Software Testing

❑Software Testing Strategy: The Big Picture

❑Considering the process from a procedural point of view, testing within the context of software engineering is actually a series of four steps that are implemented sequentially.

❑The steps are shown in Figure 17.2. Initially, tests focus on each component individually, ensuring that it functions properly as a unit.

❑Hence, the name unit testing.

❑**Unit testing** makes heavy use of testing techniques that exercise specific paths in a component's control structure to ensure complete coverage and maximum error detection.

❑Next, components must be assembled or integrated to form the complete software package.

❑**Integration testing** addresses the issues associated with the dual problems of verification and program construction.

Strategic Approach to Software Testing

❑Software Testing Strategy: The Big Picture

❑Test case design techniques that focus on inputs and outputs are more prevalent during integration, although techniques that exercise specific program paths may be used to ensure coverage of major control paths.

❑After the software has been integrated (constructed), a set of high-order tests is conducted.

❑Validation criteria (established during requirements analysis) must be evaluated.

❑**Validation testing** provides final assurance that software meets all informational, functional, behavioral, and performance requirements.

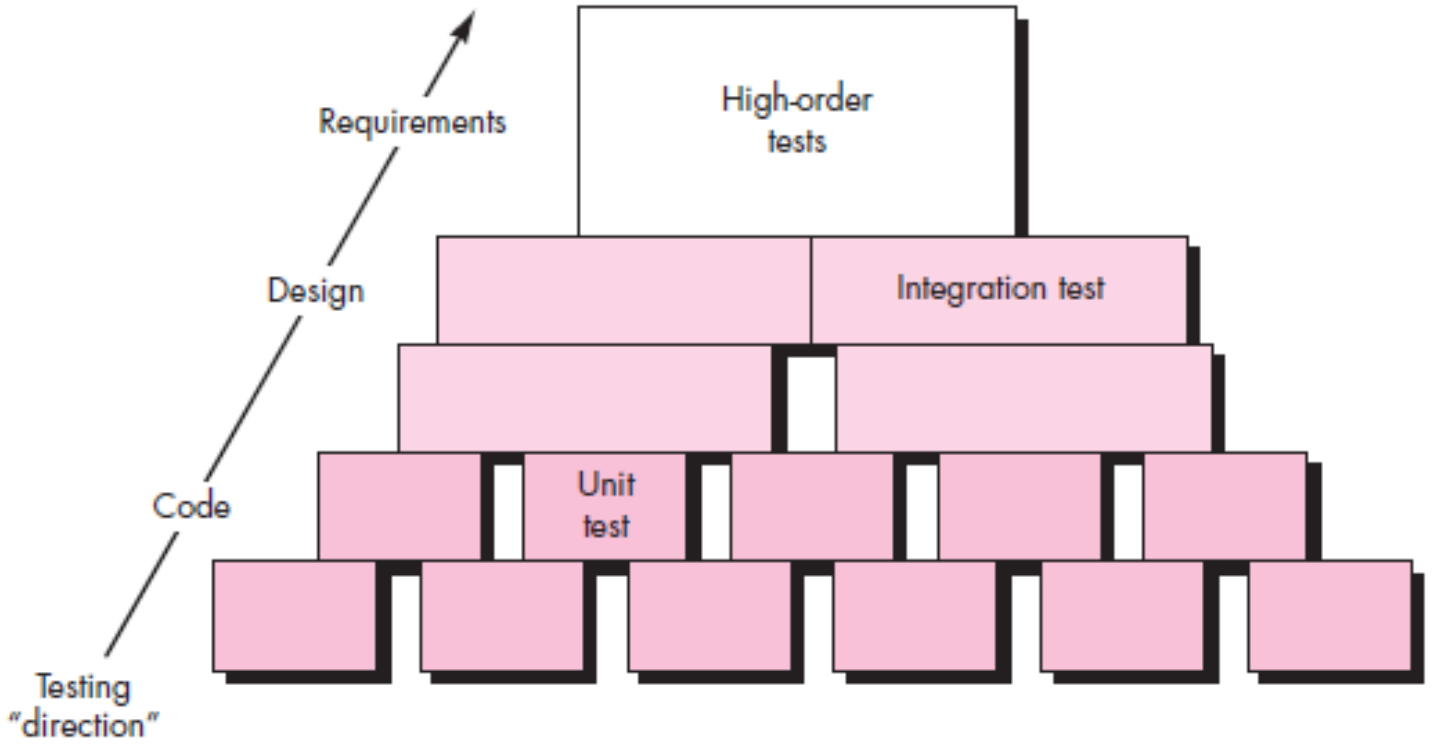
❑The last high-order testing step falls outside the boundary of software engineering and into the broader context of computer system engineering.

❑Software, once validated, must be combined with other system elements (e.g., hardware, people, databases).

❑System testing verifies that all elements mesh properly and that overall system performance is achieved.

Strategic Approach to Software Testing

FIGURE 17.2
Software testing steps



Strategic Approach to Software Testing

❑Unit Testing

❑Unit testing focuses verification effort on the smallest unit of software design the software component or module.

❑Using the component-level design description as a guide, important control paths are tested to uncover errors within the boundary of the module.

❑The relative complexity of tests and the errors those tests uncover is limited by the constrained scope established for unit testing.

❑The unit test focuses on the internal processing logic and data structures within the boundaries of a component.

❑This type of testing can be conducted in parallel for multiple components.

❑Unit-test considerations.

❑Unit tests are illustrated schematically in Figure 17.3.

❑The module interface is tested to ensure that information properly flows into and out of the program unit under test.

❑Local data structures are examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution.

❑All independent paths through the control structure are exercised to ensure that all statements in a module have been executed at least once.

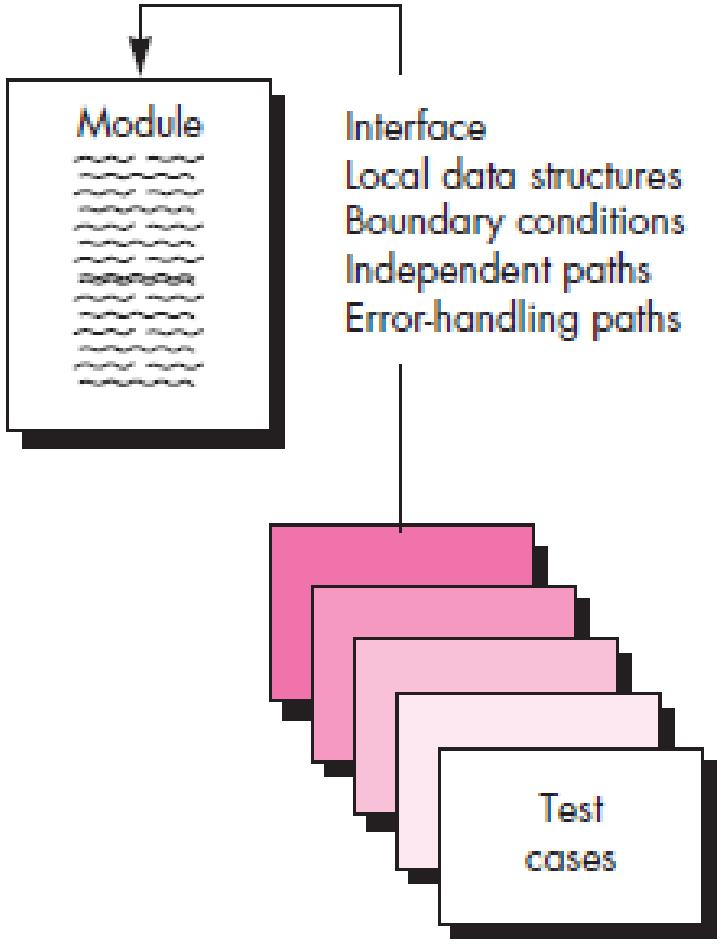
❑Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing.

❑ And finally, all error-handling paths are tested.

❑Unit-test considerations.

FIGURE 17.3

Unit test



Strategic Approach to Software Testing

❑Integration Testing

❑Integration testing is a systematic technique for constructing the software architecture while at the same time conducting tests to uncover errors associated with interfacing.

❑The objective is to take unit-tested components and build a program structure that has been dictated by design.

❑There is often a tendency to attempt nonincremental integration; that is, to construct the program using a “big bang” approach.

❑All components are combined in advance.

❑The entire program is tested as a whole.

❑And chaos usually results! A set of errors is encountered. Correction is difficult because isolation of causes is complicated by the vast expanse of the entire program.

❑Once these errors are corrected, new ones appear and the process continues in a seemingly endless loop.

Strategic Approach to Software Testing

❑Integration Testing

❑Incremental integration is the antithesis of the big bang approach.

❑The program is constructed and tested in small increments, where errors are easier to isolate and correct; interfaces are more likely to be tested completely; and a systematic test approach may be applied.

❑In the paragraphs that follow, a number of different incremental integration strategies are discussed.

❑Integration Testing: Top-down integration

❑Top-down integration testing is an incremental approach to construction of the software architecture.

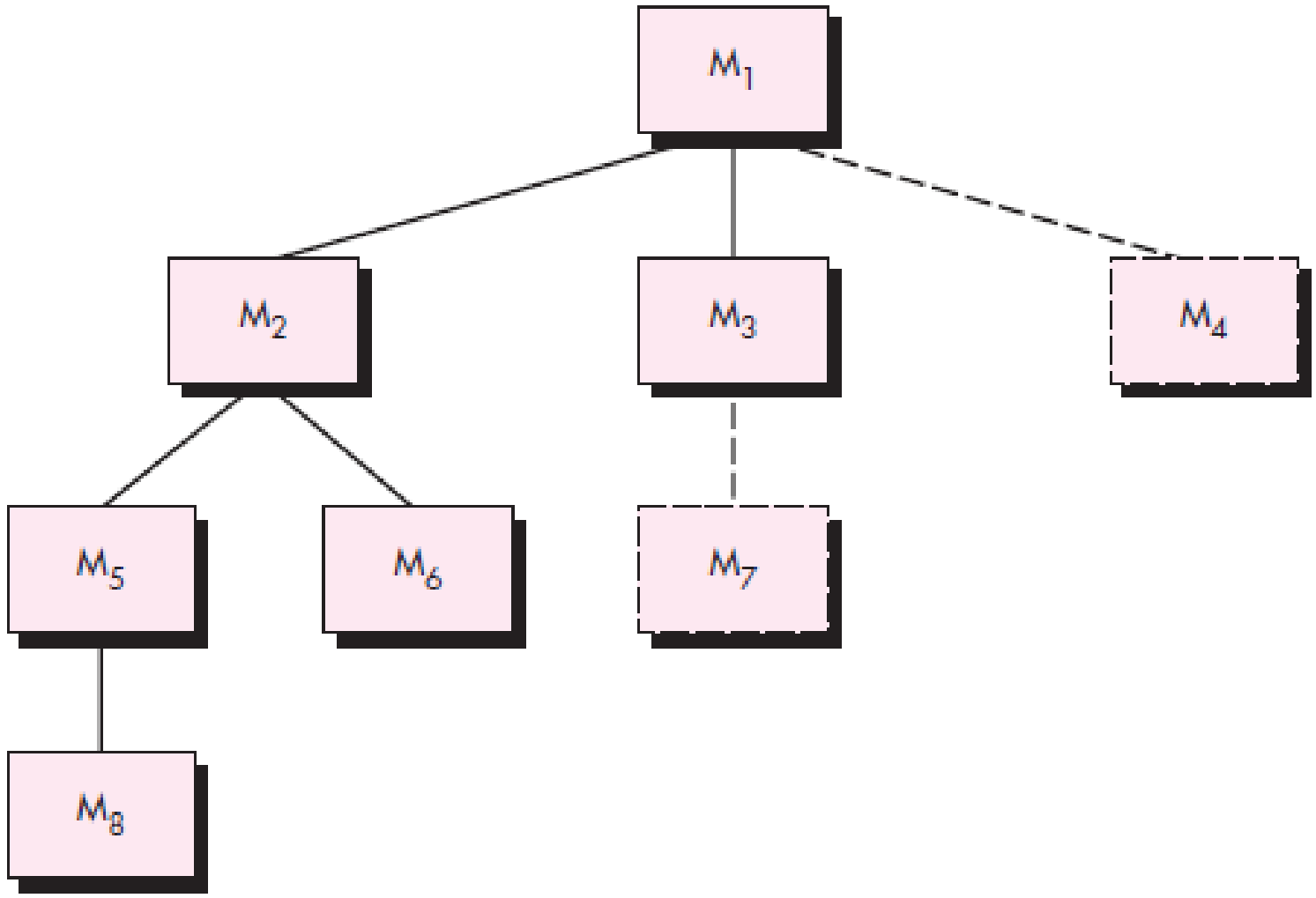
❑Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (main program).

❑Modules subordinate (and ultimately subordinate) to the main control module are incorporated into the structure in either a depth-first or breadth-first manner.

❑Integration Testing: Top-down integration

FIGURE 17.5

Top-down integration



Strategic Approach to Software Testing

❑**Integration Testing: Bottom-up integration.**

❑Bottom-up integration testing, as its name implies, begins construction and testing with atomic modules (i.e., components at the lowest levels in the program structure).

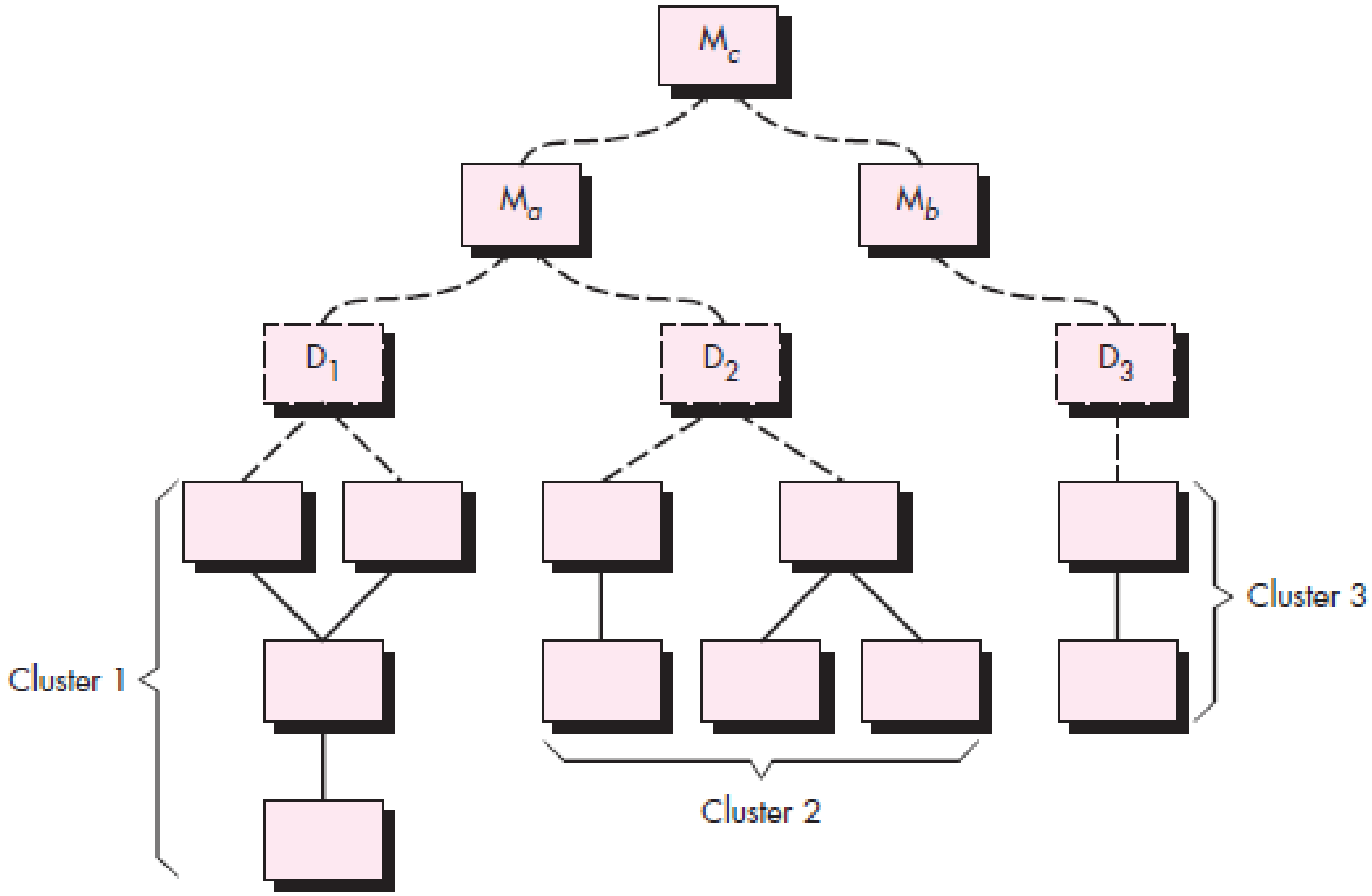
❑Because components are integrated from the bottom up, the functionality provided by components subordinate to a given level is always available and the need for stubs is eliminated.

❑A bottom-up integration strategy may be implemented with the following steps:

1. Low-level components are combined into clusters (sometimes called builds) that perform a specific software subfunction.
2. A driver (a control program for testing) is written to coordinate test case input and output.
3. The cluster is tested.
4. Drivers are removed and clusters are combined moving upward in the program structure.

❑Integration Testing: Bottom-up integration.

FIGURE 17.6
Bottom-up integration



Strategic Approach to Software Testing (Regression Testing)

- **Integration Testing: Regression testing.**
- Each time a new module is added as part of integration testing, the software changes.
- New data flow paths are established, new I/O may occur, and new control logic is invoked.
- These changes may cause problems with functions that previously worked flawlessly.
- In the context of an integration test strategy, regression testing is the reexecution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.
- In a broader context, successful tests (of any kind) result in the discovery of errors, and errors must be corrected.
- Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) is changed.
- Regression testing helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.

Strategic Approach to Software Testing (Regression Testing)

❑**Integration Testing: Regression testing.**

❑Regression testing may be conducted manually, by reexecuting a subset of all test cases or using automated capture/playback tools.

❑Capture/playback tools enable the software engineer to capture test cases and results for subsequent playback and comparison.

❑The regression test suite (the subset of tests to be executed) contains three different classes of test cases:

1. A representative sample of tests that will exercise all software functions.
2. Additional tests that focus on software functions that are likely to be affected by the change.
3. Tests that focus on the software components that have been changed.

Strategic Approach to Software Testing

❑**Integration Testing: Smoke testing.**

❑Smoke testing is an integration testing approach that is commonly used when product software is developed.

❑It is designed as a pacing mechanism for time-critical projects, allowing the software team to assess the project on a frequent basis.

❑In essence, the smoke-testing approach encompasses the following activities:

1. Software components that have been translated into code are integrated into a build. A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product Functions.
2. A series of tests is designed to expose errors that will keep the build from properly performing its function. The intent should be to uncover “showstopper” errors that have the highest likelihood of throwing the software project behind schedule.

Strategic Approach to Software Testing

❑Integration Testing: Smoke testing.

3. The build is integrated with other builds, and the entire product (in its current form) is smoke tested daily. The integration approach may be top down or bottom up.

❑Validation Testing:

❑Validation testing begins at the culmination of integration testing, when individual components have been exercised, the software is completely assembled as a package, and interfacing errors have been uncovered and corrected.

❑At the validation or system level, the distinction between conventional software, object-oriented software, and WebApps disappears.

❑Testing focuses on user-visible actions and user-recognizable output from the system.

❑Validation can be defined in many ways, but a simple (albeit harsh) definition is that validation succeeds when software functions in a manner that can be reasonably expected by the customer.

Strategic Approach to Software Testing

❑Validation Testing:

❑At this point a battle-hardened software developer might protest: “Who or what is the arbiter of reasonable expectations?”

❑ If a Software Requirements Specification has been developed, it describes all user-visible attributes of the software and contains a Validation Criteria section that forms the basis for a validation-testing approach.

❑Validation Testing Criterion

❑Software validation is achieved through a series of tests that demonstrate conformity with requirements.

❑A test plan outlines the classes of tests to be conducted, and a test procedure defines specific test cases that are designed to ensure that all functional requirements are satisfied, all behavioral characteristics are achieved, all content is accurate and properly presented, all performance requirements are attained, documentation is correct, and usability and other requirements are met (e.g., transportability, compatibility, error recovery, maintainability).

❑Validation Testing Criterion

❑After each validation test case has been conducted, one of two possible conditions exists:

1. The function or performance characteristic conforms to specification and is accepted
2. A deviation from specification is uncovered and a deficiency list is created.

❑Deviations or errors discovered at this stage in a project can rarely be corrected prior to scheduled delivery.

❑It is often necessary to negotiate with the customer to establish a method for resolving deficiencies.

❑Validation Testing Criterion

❑After each validation test case has been conducted, one of two possible conditions exists:

1. The function or performance characteristic conforms to specification and is accepted
2. A deviation from specification is uncovered and a deficiency list is created.

❑Deviations or errors discovered at this stage in a project can rarely be corrected prior to scheduled delivery.

❑It is often necessary to negotiate with the customer to establish a method for resolving deficiencies.

Strategic Approach to Software Testing

❑Alpha-Beta Testing

❑It is virtually impossible for a software developer to foresee how the customer will really use a program.

❑Instructions for use may be misinterpreted; strange combinations of data may be regularly used; output that seemed clear to the tester may be unintelligible to a user in the field.

❑When custom software is built for one customer, a series of acceptance tests are conducted to enable the customer to validate all requirements.

❑Conducted by the end user rather than software engineers, an acceptance test can range from an informal “test drive” to a planned and systematically executed series of tests.

❑In fact, acceptance testing can be conducted over a period of weeks or months, thereby uncovering cumulative errors that might degrade the system over time.

❑Alpha-Beta Testing

❑If software is developed as a product to be used by many customers, it is impractical to perform formal acceptance tests with each one.

❑Most software product builders use a process called alpha and beta testing to uncover errors that only the end user seems able to find.

❑The **alpha test** is conducted at the developer's site by a representative group of end users.

❑The software is used in a natural setting with the developer "looking over the shoulder" of the users and recording errors and usage problems.

❑Alpha tests are conducted in a controlled environment.

❑Alpha-Beta Testing

❑The **beta test** is conducted at one or more end-user sites. Unlike alpha testing, the developer generally is not present.

❑Therefore, the beta test is a “live” application of the software in an environment that cannot be controlled by the developer.

❑The customer records all problems (real or imagined) that are encountered during beta testing and reports these to the developer at regular intervals.

❑As a result of problems reported during beta tests, you make modifications and then prepare for release of the software product to the entire customer base.

❑ System Testing

❑System testing is actually a series of different tests whose primary purpose is to fully exercise the computer-based system.

❑Although each test has a different purpose, all work to verify that system elements have been properly integrated and perform allocated functions.

❑In the sections that follow, I discuss the types of system tests that are worthwhile for software-based systems.

❑ System Testing : Recovery Testing

❑Recovery testing is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed.

❑If recovery is automatic (performed by the system itself), reinitialization, checkpointing mechanisms, data recovery, and restart are evaluated for correctness.

❑If recovery requires human intervention, the mean-time-to-repair (MTTR) is evaluated to determine whether it is within acceptable limits.

❑System Testing : Security Testing

❑Security testing attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration.

❑To quote Beizer [Bei84]: “The system’s security must, of course, be tested for invulnerability from frontal attack—but must also be tested for invulnerability from flank or rear attack.”

Strategic Approach to Software Testing

❑System Testing : Stress Testing

❑Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency, or volume.

❑For example:

1. Special tests may be designed that generate ten interrupts per second, when one or two is the average rate.
2. Input data rates may be increased by an order of magnitude to determine how input functions will respond.
3. Test cases that require maximum memory or other resources are executed.
4. Test cases that may cause thrashing in a virtual operating system are designed.
5. Test cases that may cause excessive hunting for disk-resident data are created.

❑Essentially, the tester attempts to break the program.

❑System Testing : Performance Testing

❑Performance testing is designed to test the run-time performance of software within the context of an integrated system.

❑Performance testing occurs throughout all steps in the testing process.

❑Even at the unit level, the performance of an individual module may be assessed as tests are conducted.

❑However, it is not until all system elements are fully integrated that the true performance of a system can be ascertained.

❑System Testing : Deployment Testing

❑Deployment testing, sometimes called configuration testing, exercises the software in each environment in which it is to operate.

❑In addition, deployment testing examines all installation procedures and specialized installation software (e.g., “installers”) that will be used by customers, and all documentation that will be used to introduce the software to end users.

Testing Fundamentals

- The goal of testing is to find errors, and a good test is one that has a high probability of finding an error.
- Therefore, you should design and implement a computer based system or a product with "testability" in mind.
- At the same time, the tests themselves must exhibit a set of characteristics that achieve the goal of finding the most errors with a minimum of effort.

❑Testability

❑James Bach provides the following definition for testability: “Software testability is simply how easily [a computer program] can be tested.”

❑The following characteristics lead to testable software:

1. **Operability.** “The better it works, the more efficiently it can be tested.” If a system is designed and implemented with quality in mind, relatively few bugs will block the execution of tests, allowing testing to progress without fits and starts.
2. **Observability.** “What you see is what you test.” Inputs provided as part of testing produce distinct outputs. System states and variables are visible or queriable during execution. Incorrect output is easily identified. Internal errors are automatically detected and reported. Source code is accessible.

❑Testability

3. **Controllability.** “The better we can control the software, the more the testing can be automated and optimized.” All possible outputs can be generated through some combination of input, and I/O formats are consistent and structured. All code is executable through some combination of input. Software and hardware states and variables can be controlled directly by the test engineer. Tests can be conveniently specified, automated, and reproduced.
4. **Decomposability.** “By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting.” The software system is built from independent modules that can be tested independently.
5. **Simplicity.** “The less there is to test, the more quickly we can test it.” The program should exhibit functional simplicity (e.g., the feature set is the minimum necessary to meet requirements); structural simplicity (e.g., architecture is modularized to limit the propagation of faults), and code simplicity (e.g., a coding standard is adopted for ease of inspection and maintenance).

❑Testability

6. **Stability.** “The fewer the changes, the fewer the disruptions to testing.” Changes to the software are infrequent, controlled when they do occur, and do not invalidate existing tests. The software recovers well from failures.
7. **Understandability.** “The more information we have, the smarter we will test.” The architectural design and the dependencies between internal, external, and shared components are well understood. Technical documentation is instantly accessible, well organized, specific and detailed, and accurate. Changes to the design are communicated to testers.

❑Test Characteristics.

❑ And what about the tests themselves? Kaner, Falk, and Nguyen [Kan93] suggest the following attributes of a “good” test:

1. **A good test has a high probability of finding an error.** To achieve this goal, the tester must understand the software and attempt to develop a mental picture of how the software might fail. Ideally, the classes of failure are probed. For example, one class of potential failure in a graphical user interface is the failure to recognize proper mouse position. A set of tests would be designed to exercise the mouse in an attempt to demonstrate an error in mouse position recognition.
2. **A good test is not redundant.** Testing time and resources are limited. There is no point in conducting a test that has the same purpose as another test. Every test should have a different purpose (even if it is subtly different).

❑Test Characteristics.

3. **A good test should be "best of breed" [Kan93].** In a group of tests that have a similar intent, time and resource limitations may mitigate toward the execution of only a subset of these tests. In such cases, the test that has the highest likelihood of uncovering a whole class of errors should be used.
4. **A good test should be neither too simple nor too complex.** Although it is sometimes possible to combine a series of tests into one test case, the possible side effects associated with this approach may mask errors. In general, each test should be executed separately.

White Box Testing

❑**White-box testing**, sometimes called glass-box testing, is a test-case design philosophy that uses the control structure described as part of component-level design to derive test cases.

❑Using white-box testing methods, you can derive test cases that

1. Guarantee that all independent paths within a module have been exercised at least once
2. Exercise all logical decisions on their true and false sides
3. Execute all loops at their boundaries and within their operational bounds
4. Exercise internal data structures to ensure their validity.

Black Box Testing

- Black-box testing, also called behavioral testing, focuses on the functional requirements of the software.
- That is, black-box testing techniques enable you to derive sets of input conditions that will fully exercise all functional requirements for a program.
- Black-box testing is not an alternative to white-box techniques.
- Rather, it is a complementary approach that is likely to uncover a different class of errors than whitebox methods.
- Black-box testing attempts to find errors in the following categories:

1. Incorrect or missing functions
2. Interface errors
3. Errors in data structures or external database access
4. Behavior or performance errors
5. Initialization and termination errors.

Test Plan

- A test plan is a detailed document which describes software testing areas and activities.
- It outlines the test strategy, objectives, test schedule, required resources (human resources, software, and hardware), test estimation and test deliverables.
- The test plan is a base of every software's testing.
- It is the most crucial activity which ensures availability of all the lists of planned activities in an appropriate sequence.
- The test plan is a template for conducting software testing activities as a defined process that is fully monitored and controlled by the testing manager.
- The test plan is prepared by the Test Lead (60%), Test Manager(20%), and by the test engineer(20%).

Test Plan

❑**Types of Test Plan**: There are three types of the test plans

1. **Master Test Plan :** Master Test Plan is a type of test plan that has multiple levels of testing. It includes a complete test strategy.
2. **Phase Test Plan :** A phase test plan is a type of test plan that addresses any one phase of the testing strategy. For example, a list of tools, a list of test cases, etc.
3. **Specific Test Plans :** Specific test plan designed for major types of testing like security testing, load testing, performance testing, etc. In other words, a specific test plan designed for non-functional testing.

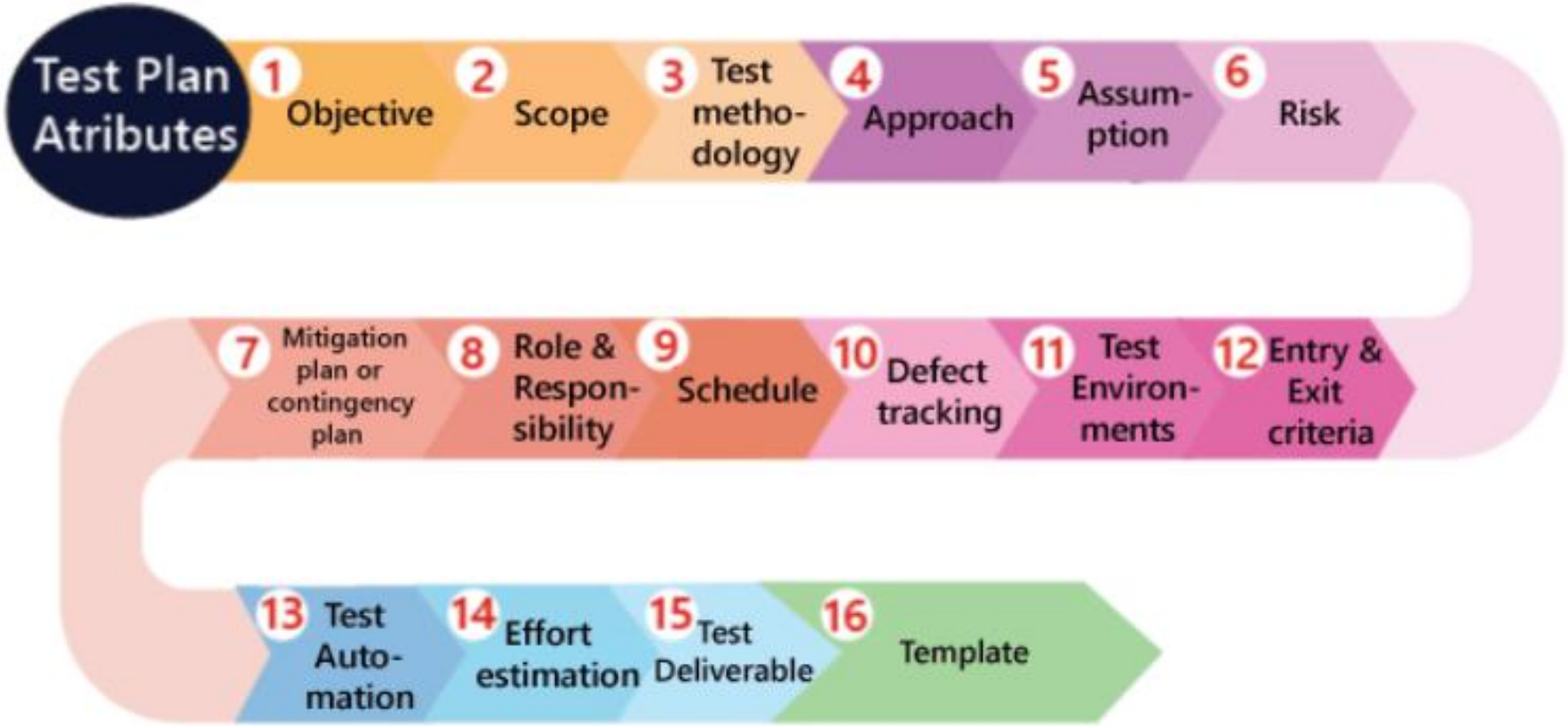
Test Plan

❑**How to write a Test Plan** : Making a test plan is the most crucial task of the test management process. According to IEEE 829, follow the following seven steps to prepare a test plan.

1. First, analyze product structure and architecture.
2. Now design the test strategy.
3. Define all the test objectives.
4. Define the testing area.
5. Define all the useable resources.
6. Schedule all activities in an appropriate manner.
7. Determine all the Test Deliverables.

Test Plan

❑**Test plan components or attributes :** The test plan consists of various parts, which help us to derive the entire testing activity.



Test Case Design

- Test case design techniques in software engineering refer to how we set up a test case which is a set of activities needed to test a specific function after a software development process.
- Using the right test case design methods will set a strong foundation for the project, increasing productivity and accuracy.
- Otherwise, you may fail to identify bugs and defects in the software testing process.
- As it is critical that your test cases are designed well, let's learn about the most popular test case design methods with examples in software testing.

Test Case Design

❑Categories of Software Testing Techniques: Following are five major test case design techniques:

1. **Boundary Value Analysis (BVA)**
2. **Equivalence Class Partitioning**
3. **Decision Table based testing**
4. **State Transition**
5. **Error Guessing**

Test Case Design

❑**Boundary value analysis** is a black-box testing technique to test the boundaries between partitions instead of testing multiple values in the equivalence region. This is because we often find a large number of errors at the boundaries rather than the center of the defined input values, and we suppose that if it is true for boundary values, it is true for the whole equivalence region. Also, BVA is considered an additional test case design type for equivalence classification.

❑**Equivalent Class Partitioning** (or Equivalent Partitioning) is a test case design method that divides the input domain data into various equivalence data classes, assuming that data in each group behaves the same. From that, we will design test cases for representative values of each class that can stand for the result of the whole class. The concept behind the Equivalent Partitioning testing technique is that the test case of a typical value is equal to the tests of the rest values in the same group. Hence, it helps reduce the number of test cases designed and executed.

Test Case Design

❑**Decision Table** is a software testing technique based on cause-effect relationships, also called a cause-effect table, used to test system behavior in which the output depends on a large combination of input. For instance, navigate a user to the homepage if all blanks/specific blanks in the log-in section are filled in. First and foremost, you need to identify the functionalities where the output responds to different input combinations. Then, for each function, divide the input set into possible smaller subsets that correspond to various outputs.

❑**State Transition** is another way to design test cases in black-box testing, in which changes in the input make changes to the state of the system and trigger different outputs. In this technique, testers execute valid and invalid cases belonging to a sequence of events to evaluate the system behavior.

Test Case Design

- In the **Error Guessing testing** technique, test cases are designed mostly based on experiences of the test analysts. He/she will try to guess and assume the possible errors or error-prone situations which can prevail in the code; hence the test designers must be skilled and experienced testers.

Test Case Design

❑**Which Test Case Design Techniques for You to Go With?:** We've gone through 5 important test case design techniques in black-box testing. Let's summarize them once again:

1. **Boundary Value Analysis (BVA)**: Test the boundaries between partitions with the assumption that they stand for the behavior of corresponding equivalence regions.
2. **Equivalence Class Partitioning**: Divide the input domains into smaller equivalence data classes and design test cases for each.
3. **Decision Table**: used when the output responds to varied combinations of input.
4. **State Transition**: used when there's a sequence of input events that can change the state of the system and produce different output.
5. **Error Guessing**: based on experience, knowledge, intuition to predict the error and defects which can prevail in the code, often used after another formal testing technique.

Test Execution

- The term Test Execution tells that the testing for the product or application needs to be executed in order to obtain the expected result.
- After the development phase, the testing phase will take place where the various levels of testing techniques will be carried out and the creation and execution of test cases will be taken place.
- Test Execution is the process of executing the tests written by the tester to check whether the developed code or functions or modules are providing the expected result as per the client requirement or business requirement.
- Test Execution comes under one of the phases of the Software Testing Life Cycle (STLC).

Test Execution

- In the test execution process, the tester will usually write or execute a certain number of test cases, and test scripts or do automated testing.
- If it creates any errors then it will be informed to the respective development team to correct the issues in the code.
- If the text execution process shows successful results then it will be ready for the deployment phase after the proper setup for the deployment environment.

Test Execution

❑Importance of Test Execution:

1. **The project runs efficiently:** Test execution ensures that the project runs smoothly and efficiently.
2. **Application competency:** It also helps to make sure the application's competency in the global market.
3. **Requirements are correctly collected:** Test executions make sure that the requirements are collected correctly and incorporated correctly in design and architecture.
4. **Application built in accordance with requirements:** It also checks whether the software application is built in accordance with the requirements or not.

Test Execution

❑Five Main Activities for Test Execution

1. **Defect Finding and Reporting:** Defect finding is the process of identifying the bugs or errors raised while executing the test cases on the developed code or modules. If any error appears or any of the test cases failed then it will be recorded and the same will be reported to the respective development team. Sometimes, during the user acceptance testing also end users may find the error and report it to the team. All the recorded details will be reported to the respective team and they will work on the recorded errors or bugs.
2. **Defect Mapping:** After the error has been detected and reported to the development team, the development team will work on those errors and fix them as per the requirement. Once the development team has done its job, the tester team will again map the test cases or test scripts to that developed module or code to run the entire tests to ensure the correct output.

Test Execution

❑Five Main Activities for Test Execution

3. **Re-Testing:** From the name itself, we can easily understand that Re-Testing is the process of testing the modules or entire product again to ensure the smooth release of the module or product. In some cases, the new module or functionality will be developed after the product release. In this case, all the modules will be re-tested for a smooth release. So that it cannot cause any other defects after the release of the product or application.
4. **Regression Testing:** Regression Testing is software testing that ensures that the newly made changes to the code or newly developed modules or functions should not affect the normal processing of the application or product.
5. **System Integration Testing:** System Integration Testing is a type of testing technique that will be used to check the entire component or modules of the system in a single run. It ensures that the whole system will be checked in a single test environment instead of checking each module or function separately.

Test Execution

❑Test Execution Process :

❑The test Execution technique consists of **three different phases** which will be carried out to process the test result and ensure the correctness of the required results.

❑In each phase, various activities or work will be carried out by various team members.

❑The three main phases of test execution are the creation of test cases, test case execution, and validation of test results. Let us discuss each phase.

Test Execution

❑Test Execution Process :

1. **Creation of Test Cases:** The first phase is to create suitable test cases for each module or function. Here, the tester with good domain knowledge must be required to create suitable test cases. It is always preferable to create simple test cases and the creation of test cases should not be delayed else it will cause excess time to release the product. The created test cases should not be repeated again. It should cover all the possible scenarios raised in the application.
2. **Test Cases Execution:** After test cases have been created, execution of test cases will take place. Here, the Quality Analyst team will either do automated or manual testing depending upon the test case scenario. It is always preferable to do both automated as well as manual testing to have 100% assurance of correctness. The selection of testing tools is also important to execute the test cases.
3. **Validating Test Results:** After executing the test cases, note down the results of each test case in a separate file or report. Check whether the executed test cases achieved the expected result and record the time required to complete each test case i.e., measure the performance of each test case. If any of the test cases is failed or not satisfied the condition then report it to the development team for validating the code.

Test Execution

❑**Ways to Perform Test Execution :** Testers can choose from the below list of preferred methods to carry out test execution:

1. **Run test cases:** It is a simple and easiest approach to run test cases on the local machine and it can be coupled with other artifacts like test plans, test suites, test environments, etc.
2. **Run test suites:** A test suite is a collection of manual and automated test cases and the test cases can be executed sequentially or in parallel. Sequential execution is useful in cases where the result of the last test case depends on the success of the current test case.
3. **Run test case execution and test suite execution records:** Recording test case execution and test suite execution is a key activity in the test process and helps to reduce errors, making the testing process more efficient.
4. **Generate test results without execution:** Generating test results from non-executed test cases can be helpful in achieving comprehensive test coverage.
5. **Modify execution variables:** Execution variables can be modified in the test scripts for particular test runs.

Test Execution

❑**Ways to Perform Test Execution :** Testers can choose from the below list of preferred methods to carry out test execution:

6. **Run automated and manual tests:** Test execution can be done manually or can be automated.
7. **Schedule test artifacts:** Test artifacts include video, screenshots, data reports, etc. These are very helpful as they document the results of the past test execution and provide information about what needs to be done in future test execution.
8. **Defect tracking:** Without defect tracking test execution is not possible, as during testing one should be able to track the defects and identify what when wrong and where.

Test Execution

❑**Test Execution States:** The tester or the Quality Analyst team reports or notices the result of each test case and records it in their documentation or file. There are various results raised when executing the test cases. They are

1. **Pass:** It tells that the test cases executed for the module or function are successful.
2. **Fail:** It tells that the test cases executed for the module or function are not successful and resulted in different outputs.
3. **Not Run:** It tells that the test cases are yet to be executed.
4. **Partially Executed:** It tells that only a certain number of test cases are passed and others aren't met the given requirement.
5. **Inconclusive:** It tells that the test cases are executed but it requires further analysis before the final submission.
6. **In Progress:** It tells that the test cases are currently executed.
7. **Unexpected Result:** It tells that all the test cases are executed successfully but provide different unexpected results.

Test Execution

❑**Test Execution Cycle:** A test execution cycle is an iterative approach that will be helpful in detecting errors. The test execution cycle includes various processes. These are:

1. **Requirement Analysis:** In which, the QA team will gather all the necessary requirements needed for test execution. For example, how many testers are needed, what automation test tools are needed, what testing covers under the given budget, etc., the QA team will also plan depending upon the client or business requirement.
2. **Test Planning:** In this phase, the QA team will plan when to start and complete the testing. Choosing of correct automation test tool, and testers needed for executing the test plan. They further plan who should develop the test cases for which module/function, who should execute the test cases, how many test cases needed to be executed, etc.,
3. **Test Cases Development:** This is the phase in which the QA team assigned a group of testers to write or generate the test cases for each module. A tester with good domain knowledge will easily write the best test cases or test scripts. Prioritizing the developed test cases is also the main factor.

Test Execution

❑Test Execution Cycle:

4. **Test Environment Setup:** Test Environment Setup usually differs from project to project. In some cases, it is created by the team itself and it is also created by clients or customers. Test Environment Setup is nothing but testing the entire developed product with suitable software or hardware components or with both by executing all the tests on it. It is essential and it is sometimes carried out along with the test case development process.
5. **Test Execution:** This stage involves test execution by the team and all the detected bugs are recorded and reported for remediation and rectification.
6. **Test Closure:** This is the final stage and here it records the entire details of the test execution process. It also contains the end-users testing details. It again modifies the testing process if any defects are found during the testing. Hence, it is a repetitive process.

Reviews

❑**Software Review** is a systematic inspection of software by one or more individuals who work together to find and resolve errors and defects in the software during the early stages of the Software Development Life Cycle (SDLC).

❑A software review is an essential part of the Software Development Life Cycle (SDLC) that helps software engineers in validating the quality, functionality, and other vital features and components of the software.

❑It is a whole process that includes testing the software product and it makes sure that it meets the requirements stated by the client.

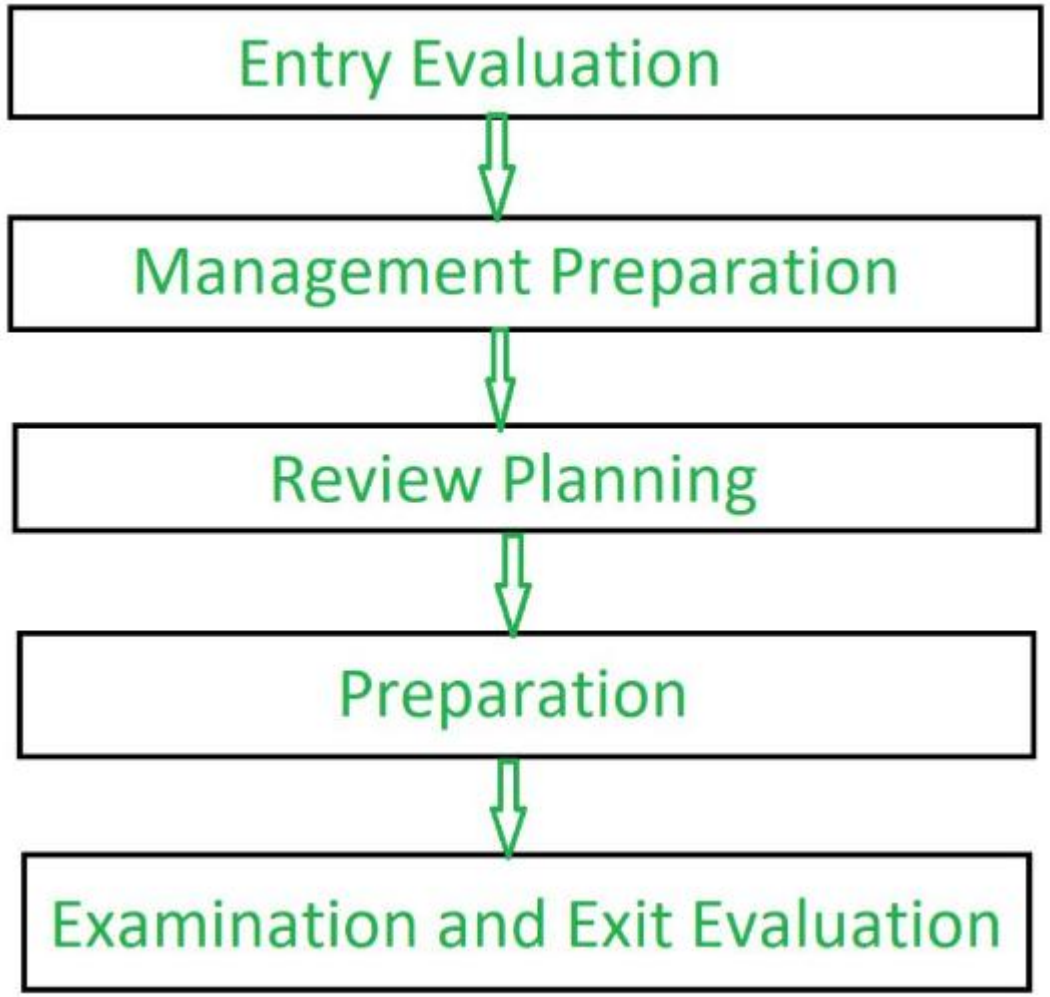
❑Usually performed manually, software review is used to verify various documents like requirements, system designs, codes, test plans, and test cases.

Reviews

- Following are the main objective of the software review:

1. To improve the productivity of the development team.
2. To make the testing process time and cost-effective.
3. To make the final software with fewer defects.
4. To eliminate the inadequacies.

❑Process of Software Review



❑Process of Software Review

1. **Entry Evaluation:** By confirming documentation, fulfilling entry requirements and assessing stakeholder and team preparation, you can determine the software's availability.
2. **Management Preparation:** To get ready for the review process, assign roles, gather resources and provide brief management.
3. **Review Planning:** Establish the review's goals and scope, invite relevant parties and set a time for the meeting.
4. **Preparation:** Distribute appropriate resources, give reviewers time to get familiar and promote issue identification to help them prepare.
5. **Examination and Exit Evaluation:** Reviewers should collaborate to examine the results, record concerns, and encourage candid communication in meetings. It assess the results, make remedial plans based on flaws that have been reported and assess the process's overall efficacy.

Reviews

- **There are mainly 3 types of software reviews:**

1. Software Peer Review
2. Software Management Review
3. Software Audit Review

Reviews

❑Software Peer Review

❑Peer review is the process of assessing the technical content and quality of the product and it is usually conducted by the author of the work product along with some other developers.

❑Peer review is performed in order to examine or resolve the defects in the software, whose quality is also checked by other members of the team.

❑Peer Review has following types:

1. **Code Review:** Computer source code is examined in a systematic way.
2. **Pair Programming:** It is a code review where two developers develop code together at the same platform.
3. **Walkthrough:** Members of the development team is guided by author and other interested parties and the participants ask questions and make comments about defects.
4. **Technical Review:** A team of highly qualified individuals examines the software product for its client's use and identifies technical defects from specifications and standards.
5. **Inspection:** In inspection the reviewers follow a well-defined process to find defects.

Reviews

❑**Software Management Review:** Software Management Review evaluates the work status. In this section decisions regarding downstream activities are taken.

❑**Software Audit Review:** Software Audit Review is a type of external review in which one or more critics, who are not a part of the development team, organize an independent inspection of the software product and its processes to assess their compliance with stated specifications and standards. This is done by managerial level people.

Inspection and Auditing

❑Audit:

❑A software audit is a thorough review of a software product to check its quality, progress, standards and regulations.

❑It basically checks the health of a product and ensures that everything is going as planned.

❑It can be done by an internal team or any external independent auditors.

❑If the audit is performed by an external auditor, it can add some financial strain to the company and disrupt development to accommodate such audits.

❑It is recommended to perform regular internal audits on the software to ensure that proper regulations are followed and all licenses are up-to-date as it can also save the company from unnecessary legal issues.

❑Audit: Benefits of Audit in Software Testing

1. Helps in validating the testing process and identifies ways to optimize the existing process
2. It checks for any mismatches between the requirements and the delivered features. There can be miscommunication between business and technical teams which can cause such mismatches. Audits helps in capturing such issues
3. Ensures the development progress is as expected with compliance to regulations and best practices. It can also catch any potential risks to the product and help mitigate it
4. Incase any issues are noticed, proper suggestions are gives to improve upon the process or the product

Inspection and Auditing

❑Audit: Types of Audit in Software Testing

1. **Internal audit:** These audits are done within the organization
2. **External audit:** These are done by independent contractors or external agencies
3. **Compliance audit:** This audit checks if the process is within the given standards. If the testing process has certain standards to adhere to, this audit ensures that it's followed
4. **Process improvement:** If there are any changes needed for the existing process, this audit helps in identifying them. This is done by evaluating the various steps in the process and identifying any problems, and eliminating them.
5. **Root cause analysis:** This audit helps to find the root cause for a problem using different testing processes. It is done for specific problems that require some attention and needs to be resolved.

❑Audit: Metrics for Software Audit

1. Project Metrics:

Percentage of test case execution: It analyses how many of the test cases are executed in the testing process:

Percent of Test Case Execution = (Number of Passed Tests + Number of Failed Tests + Number of Blocked Tests) / Number of Test Cases

2. Product Metrics

Critical defects: This helps in understanding the current quality of a product

Total Percentage of Critical Defects = (Critical Defects / Total Defects Reported) x 100

Defect distribution across components: There may be some components in a product that may have significantly higher defects than others, and it is important to identify them. This metric helps in analyzing the problematic areas and focuses on these issues.

Defect Distribution Across Components = Total Number of Defects / Functional area(s)

Defect priority distribution: This helps in gauging the effectiveness of the testing process. Based on the priority of the defects within a component, it helps to decide which component requires more attention over the others.

❑Audit: Metrics for Software Audit

3. People Metrics

Issues per reporter: This keeps track of how many issues were reported by each reporter. It gives an idea of which defects the tester is working on, i.e., regression testing or identifying bugs

Tests reported by each team member: This metric helps the management gauge each team member's performance

Inspection and Auditing

❑**Audit: There are some simple steps to follow while performing an audit:**

1. Identify the purpose of the audit and what it hopes to find. By being specific, helps in getting optimum results and eliminates the problems efficiently
2. Examine the testing processes being done and verify the current processes against the planned and defined procedures and guidelines which were documented as a part of the testing manual prior to the testing phase
3. Once the testing process is verified, each of the test cases, test suites, test logs, defect reports, test coverage and traceability matrix are thoroughly reviewed
4. Interviewing the individuals involved at different stages in the testing process to get a better idea of the current progress

Inspection and Auditing

❑Inspection:

❑In general the term Inspection means the process of evaluating or examining things.

❑The output of inspection is compared with the set standard or specified requirement to check whether an item that is being developed is as per the requirement or not.

❑It is the non-destructive type of testing and it doesn't harm the product under evaluation.

❑Inspection is a formal review type that is led by trained and expert moderators.

❑During the inspection the documents are prepare and checked thoroughly by the reviewers before the meeting. It involves peers to examine the product.

❑Moreover, a separate preparation is carried out in Inspection, during which the product is examined and the defects are found.

❑These defects found are documented in a logging list or issue log, after which a formal follow-up is carried out by the moderator applying exit criteria.

❑Inspection: Goals of Inspection in Software Testing

❑Inspection is a type of appraisal execution that is frequently used in software applications.

❑The objective of inspection is to enable the observer to achieve agreement on an exertion system and endorse it for employing it in the development of the software application.

❑Usually inspected job systems consist of software necessities definition and test designs.

❑Other goals of Inspection are:

1. It helps the author to improve the quality of the document under inspection.
2. Removes defects efficiently, as soon as possible.
3. Helps in improving product quality.
4. Enables common understanding by exchanging information.
5. One can learn from defects found and prevent the occurrence of similar defects.

Inspection and Auditing

❑Inspection: Following are the Characteristics of Inspection

1. The process of inspection is usually led by a trained moderator, who is not the author. Moderator's role is to do a peer examination of a document.
2. It is extremely formal and is driven by checklists and rules.
3. This review process makes use of **entry and exit criteria**.
4. It is essential to have a pre-meeting preparation.
5. Inspection report is prepared and shared with the author for appropriate actions.
6. Post Inspection, a formal follow-up process is used to ensure a timely and a prompt corrective action.
7. Aim of Inspection is not only to identify defects, but also to bring process improvement.

Mutation Testing

- Mutation testing is a white box method in software testing where we insert errors purposely into a program (under test) to verify whether the existing test case can detect the error or not.
- In this testing, the mutant of the program is created by making some modifications to the original program.
- The primary objective of mutation testing is to check whether each mutant created an output, which means that it is different from the output of the original program.
- We will make slight modifications in the mutant program because if we change it on a massive scale than it will affect the overall plan.
- When we detected the number of errors, it implies that either the program is correct or the test case is inefficient to identify the fault.
- Mutation testing purposes is to evaluate the quality of the case that should be able to fail the mutant code hence this method is also known as Fault-based testing as it used to produce an error in the program and that why we can say that the mutation testing is performed to check the efficiency of the test cases.

Mutation Testing

❑What is mutation?

❑The mutation is a small modification in a program; these minor modifications are planned to typical low-level errors which are happened at the time of coding process.

❑Generally, we deliberate the mutation operators in the form of rules which match the data and also generate some efficient environment to produce the mutant.

❑**Types of mutation testing**: Mutation testing can be classified into three parts, which are as follows:

1. Decision mutations
2. value mutations
3. Statement mutations

Mutation Testing

- **Types of mutation testing**: Decision mutations
- In this type of mutation testing, we will check the design errors.
- And here, we will do the modification in arithmetic and logical operator to detect the errors in the program.
- Like if we do the following changes in arithmetic operators:
  - plus(+)→ minus(-)
  - asterisk(*)→ double asterisk(**)
  - plus(+)→incremental operator(i++)
  - Like if we do the following changes in logical operators
  - Exchange P **>** → P**<**, OR P**>=**

Mutation Testing

❑**Types of mutation testing**: Decision mutations

Now, let see one example for our better understanding:

Original Code	Modified Code

if(p >q)
r= 5;
else
r= 15;
	if(p < q)
	r = 5;
	else
	r = 15;

❑**Types of mutation testing**: Value mutations

❑In this, the values will modify to identify the errors in the program, and generally, we will change the following:

❑Small value à higher value

❑Higher value à Small value.

Example:

Original Code	Modified Code

int add =9000008;
int p = 65432;
int q =12345;
int r = (p+ q);
	int mod = 9008;
	int p = 65432;
	int q =12345;
	int r= (p + q);

❑Types of mutation testing: Statement mutations

❑Statement mutations means that we can do the modifications into the statements by removing or replacing the line as we see in the below example:

Original Code	Modified Code

if(p * q)
r = 15;
else
r = 25;
	if(p* q)
	s = 15;
	else
	s = 25;

❑In the above case, we have replaced the statement r=15 by s=15, and r=25 by s=25.

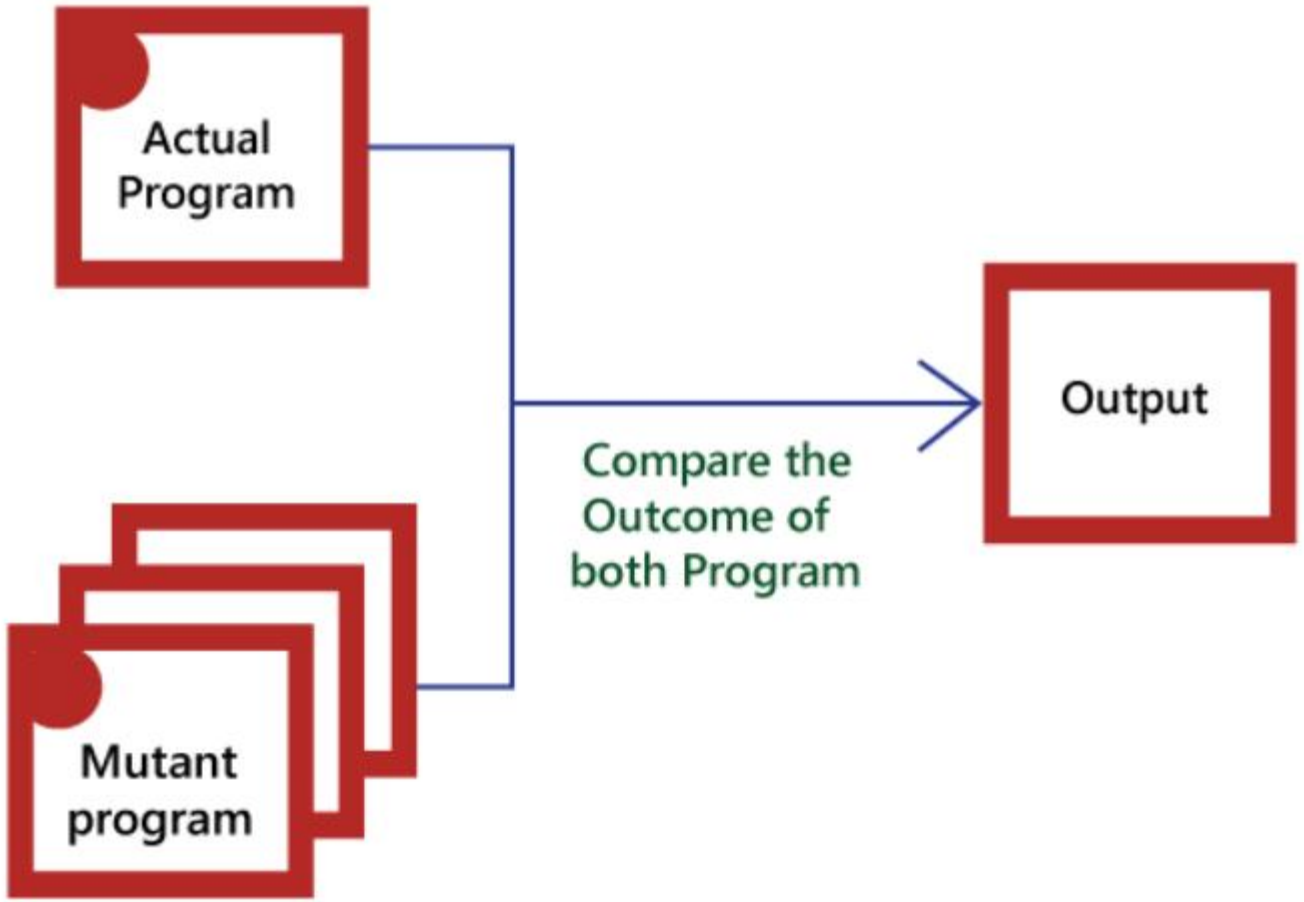
Mutation Testing

❑Steps of Mutation Testing:

1. In this, firstly, we will add the errors into the source code of the program by producing various versions, which are known mutants. Here every mutant having the one error, which leads the mutant kinds unsuccessful and also validates the efficiency of the test cases.
2. After that, we will take the help of the test cases in the mutant program and the actual application will find the errors in the code.
3. Once we identify the faults, we will match the output of the actual code and mutant code.
4. After comparing the output of both actual and mutant programs, if the results are not matched, then the mutant is executed by the test cases. Therefore the test case has to be sufficient for identifying the modification between the actual program and the mutant program.
5. And if the actual program and the mutant program produced the exact result, then the mutant is saved. And those cases are more active test cases because it helps us to execute all the mutants.

Mutation Testing

❑Steps of Mutation Testing:



Object Oriented Testing

- Classical software testing strategy begins with “testing in the small” and works outward toward “testing in the large.”
- Stated in the jargon of software testing, you begin with unit testing, then progress toward integration testing, and culminate with validation and system testing.
- In conventional applications, unit testing focuses on the smallest compliable program unit the subprogram (e.g., component, module, subroutine, procedure).
- Once each of these units has been testing individually, it is integrated into a program structure while a series of regression tests are run to uncover errors due to interfacing the modules and side effects that are caused by the addition of new units.
- Finally, the system as a whole is tested to ensure that errors in requirements are uncovered.

❑Unit Testing in the OO Context

- ❑When object-oriented software is considered, the concept of the unit changes.
- ❑Encapsulation drives the definition of classes and objects.
- ❑This means that each class and each instance of a class (object) packages attributes (data) and the operations (also known as methods or services) that manipulate these data.
- ❑Rather than testing an individual module, the smallest testable unit is the encapsulated class.
- ❑Because a class can contain a number of different operations and a particular operation may exist as part of a number of different classes, the meaning of unit testing changes dramatically.

❑Integration Testing in the OO Context

❑Because object-oriented software does not have a hierarchical control structure, conventional top-down and bottom-up integration strategies have little meaning.

❑In addition, integrating operations one at a time into a class (the conventional incremental integration approach) is often impossible because of the “direct and indirect interactions of the components that make up the class” [Ber93].

❑Integration Testing in the OO Context

❑There are two different strategies for integration testing of OO systems [Bin94a].

❑The first, 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.

❑The second integration approach, use-based testing, begins the construction of the system by testing those classes (called independent classes) that use very few (if any) of server classes.

❑After the independent classes are tested, the next layer of classes, called dependent classes, that use the independent classes are tested.

❑This sequence of testing layers of dependent classes continues until the entire system is constructed.

❑Unlike conventional integration, the use of driver and stubs as replacement operations is to be avoided, when possible.

❑Integration Testing in the OO Context

❑Cluster testing is one step in the integration testing of OO software.

❑Here, a cluster of collaborating classes (determined by examining the CRC and object relationship model) is exercised by designing test cases that attempt to uncover errors in the collaborations.

❑Validation Testing in an OO Context

❑At the validation or system level, the details of class connections disappear.

❑Like conventional validation, the validation of OO software focuses on user-visible actions and user-recognizable outputs from the system.

❑To assist in the derivation of validation tests, the tester should draw upon use cases that are part of the requirements model.

❑The use case provides a scenario that has a high likelihood of uncovered errors in user-interaction requirements.

❑Conventional black-box testing methods can be used to drive validation tests. In addition, you may choose to derive test cases from the object behavior model and from an event flow diagram created as part of OOA.

❑OBJECT-ORIENTED TESTING METHODS

❑The architecture of object-oriented software results in a series of layered subsystems that encapsulate collaborating classes.

❑Each of these system elements (subsystems and classes) performs functions that help to achieve system requirements.

❑It is necessary to test an OO system at a variety of different levels in an effort to uncover errors that may occur as classes collaborate with one another and subsystems communicate across architectural layers.

❑Test-case design methods for object-oriented software continue to evolve.

❑OBJECT-ORIENTED TESTING METHODS

❑However, an overall approach to OO test-case design has been suggested by Berard [Ber93]:

1. Each test case should be uniquely identified and explicitly associated with the class to be tested.

2. The purpose of the test should be stated.

3. A list of testing steps should be developed for each test and should contain:

a. A list of specified states for the class that is to be tested

b. A list of messages and operations that will be exercised as a consequence of the test

c. A list of exceptions that may occur as the class is tested

d. A list of external conditions (i.e., changes in the environment external to the software that must exist in order to properly conduct the test)

e. Supplementary information that will aid in understanding or implementing the test

❑The Test-Case Design Implications of OO Concepts

❑As a class evolves through the requirements and design models, it becomes a target for test-case design.

❑Because attributes and operations are encapsulated, testing operations outside of the class is generally unproductive.

❑Although encapsulation is an essential design concept for OO, it can create a minor obstacle when testing.

❑As Binder [Bin94a] notes, “Testing requires reporting on the concrete and abstract state of an object.” Yet, encapsulation can make this information somewhat difficult to obtain.

❑Unless built-in operations are provided to report the values for class attributes, a snapshot of the state of an object may be difficult to acquire.

❑Applicability of Conventional Test-Case Design Methods

❑The white-box testing methods can be applied to the operations defined for a class.

❑Basis path, loop testing, or data flow techniques can help to ensure that every statement in an operation has been tested.

❑However, the concise structure of many class operations causes some to argue that the effort applied to white-box testing might be better redirected to tests at a class level.

❑Black-box testing methods are as appropriate for OO systems as they are for systems developed using conventional software engineering methods.

❑Use cases can provide useful input in the design of black-box and state based tests.

❑Fault-Based Testing

❑The object of fault-based testing within an OO system is to design tests that have a high likelihood of uncovering plausible faults.

❑Because the product or system must conform to customer requirements, preliminary planning required to perform fault based testing begins with the analysis model.

❑The tester looks for plausible faults (i.e., aspects of the implementation of the system that may result in defects).

❑To determine whether these faults exist, test cases are designed to exercise the design or code.

Testing Web Based System

- Testing is the process of exercising software with the intent of finding (and ultimately correcting) errors.
- This fundamental philosophy, does not change for WebApps.
- In fact, because Web-based systems and applications reside on a network and interoperate with many different operating systems, browsers (residing on a variety of devices), hardware platforms, communications protocols, and “backroom” applications, the search for errors represents a significant challenge.
- To understand the objectives of testing within a Web engineering context, you should consider the many dimensions of WebApp quality.
- In the context of this discussion, I consider quality dimensions that are particularly relevant in any discussion of WebApp testing.
- I also consider the nature of the errors that are encountered as a consequence of testing, and the testing strategy that is applied to uncover these errors.

❑Dimensions of Quality

❑Quality is incorporated into a Web application as a consequence of good design. It is evaluated by applying a series of technical reviews that assess various elements of the design model and by applying a testing process.

❑Both reviews and testing examine one or more of the following quality dimensions [Mil00a]:

1. **Content** is evaluated at both a syntactic and semantic level. At the syntactic level, spelling, punctuation, and grammar are assessed for text-based documents. At a semantic level, correctness (of information presented), consistency (across the entire content object and related objects), and lack of ambiguity are all assessed.
2. **Function** is tested to uncover errors that indicate lack of conformance to customer requirements. Each WebApp function is assessed for correctness, instability, and general conformance to appropriate implementation standards (e.g., Java or AJAX language standards).

❑Dimensions of Quality

3. **Structure** is assessed to ensure that it properly delivers WebApp content and function, that it is extensible, and that it can be supported as new content or functionality is added.
4. **Usability** is tested to ensure that each category of user is supported by the interface and can learn and apply all required navigation syntax and semantics.
5. **Navigability** is tested to ensure that all navigation syntax and semantics are exercised to uncover any navigation errors (e.g., dead links, improper links, erroneous links).
6. **Performance** is tested under a variety of operating conditions, configurations, and loading to ensure that the system is responsive to user interaction and handles extreme loading without unacceptable operational degradation.

❑Dimensions of Quality

7. **Compatibility** is tested by executing the WebApp in a variety of different host configurations on both the client and server sides. The intent is to find errors that are specific to a unique host configuration.
8. **Interoperability** is tested to ensure that the WebApp properly interfaces with other applications and/or databases.
9. **Security** is tested by assessing potential vulnerabilities and attempting to exploit each. Any successful penetration attempt is deemed a security failure.

Testing Web Based System

❑Testing Strategy

❑The strategy for WebApp testing adopts the basic principles for all software testing and applies a strategy and tactics that have been recommended for object-oriented systems.

❑The following steps summarize the approach:

1. The content model for the WebApp is reviewed to uncover errors.
2. The interface model is reviewed to ensure that all use cases can be accommodated.
3. The design model for the WebApp is reviewed to uncover navigation errors.
4. The user interface is tested to uncover errors in presentation and/or navigation mechanics.
5. Functional components are unit tested.

❑Testing Strategy

7. The WebApp is implemented in a variety of different environmental configurations and is tested for compatibility with each configuration.

8. Security tests are conducted in an attempt to exploit vulnerabilities in the WebApp or within its environment.

9. Performance tests are conducted.

10. The WebApp is tested by a controlled and monitored population of end users; the results of their interaction with the system are evaluated for content and navigation errors, usability concerns, compatibility concerns, and WebApp security, reliability, and performance.

Testing Web Based System

❑Testing Process

❑You begin the WebApp testing process with tests that exercise content and interface functionality that are immediately visible to end users.

❑As testing proceeds, aspects of the design architecture and navigation are exercised.

❑Finally, the focus shifts to tests that examine technological capabilities that are not always apparent to end users—WebApp infrastructure and installation/implementation issues.

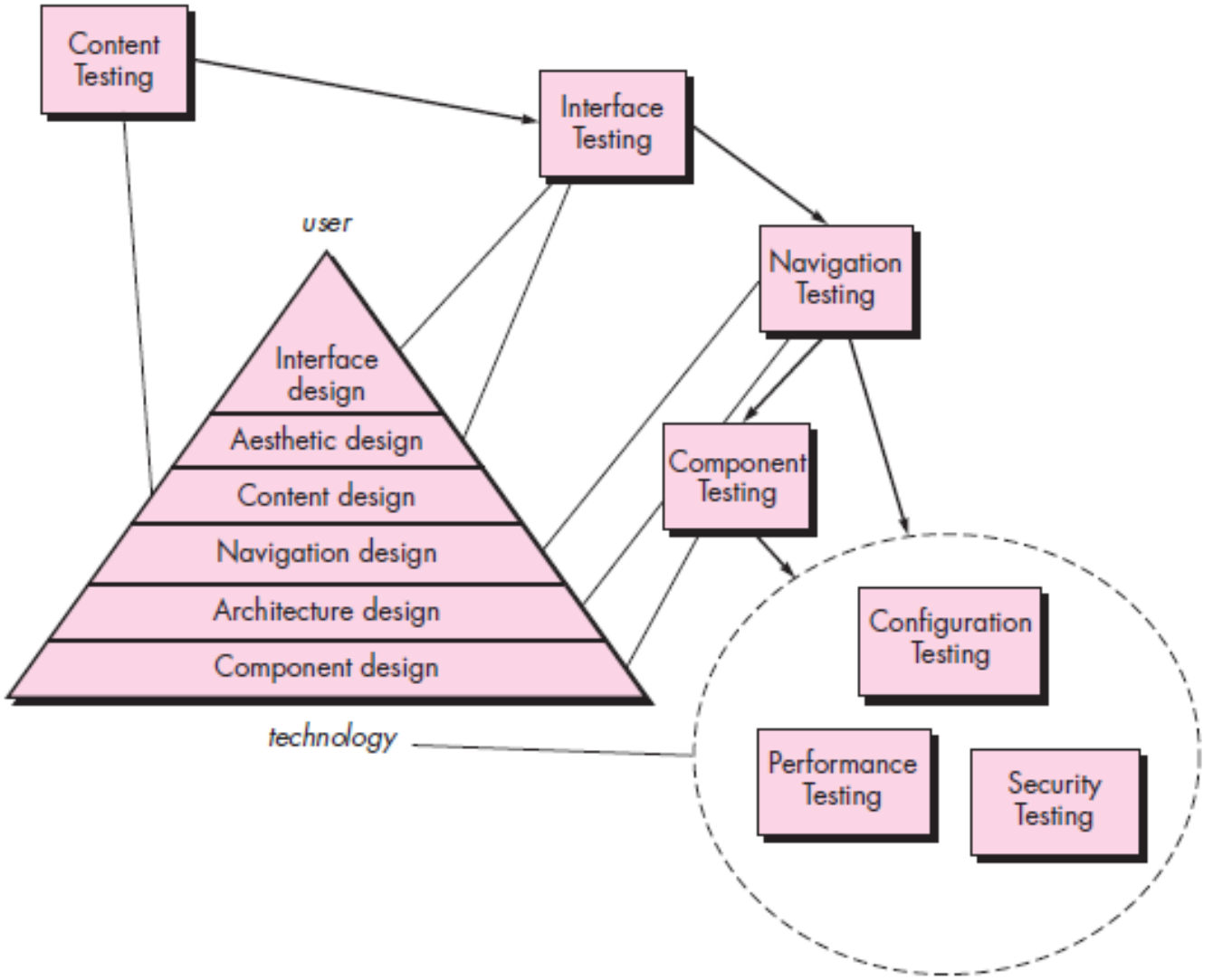
❑Figure 20.1 juxtaposes the WebApp testing process with the design pyramid for WebApps.

❑Note that as the testing flow proceeds from left to right and top to bottom, user-visible elements of the WebApp design (top elements of the pyramid) are tested first, followed by infrastructure design elements.

Testing Web Based System

❑Testing Process

FIGURE 20.1
The testing process



Mobile App Testing

- Explosive growth in the use of mobile devices and the development of mobile devices makes testing an essential requirement for the successful and rapid delivery of high-quality mobile applications.
- We will follow two different approaches for the testing of mobile applications.
- Here we will follow two approaches to test the mobile application, and those are manual testing and automated testing.
- **Manual Testing**
- **Automation Testing**

Mobile App Testing

- **Manual Testing:** is a human process.
- The primary focus of manual testing is on the experience of the user.
- The Analysis and evaluation of the application's functionality can be done through the medium of the user in an explorative process.
- Manual Testing ensures that the application work on the standard of user-friendliness.
- Manual testing is generally the time-consuming process because the process is to find out the bugs will take time.
- Therefore, according to the thumb rule, 20% of applications at the time of releases should be tested with the alpha and beta testing.
- In the rest part of the application, automated testing should be performed.

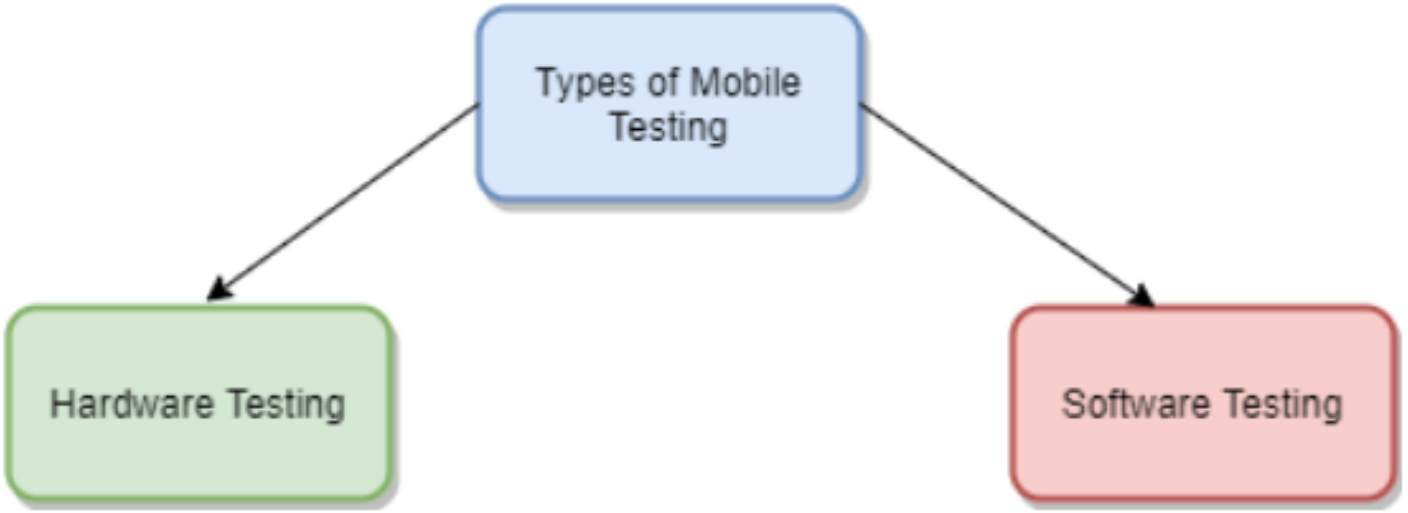
Mobile App Testing

- **Automation Testing:** is the second approach to test the mobile application.
- In this process, an array of test cases is set up.
- Automated Testing covers 80% of the testing process.
- The percentage is not set up, but this is the general guideline followed by the software industry.

Mobile App Testing

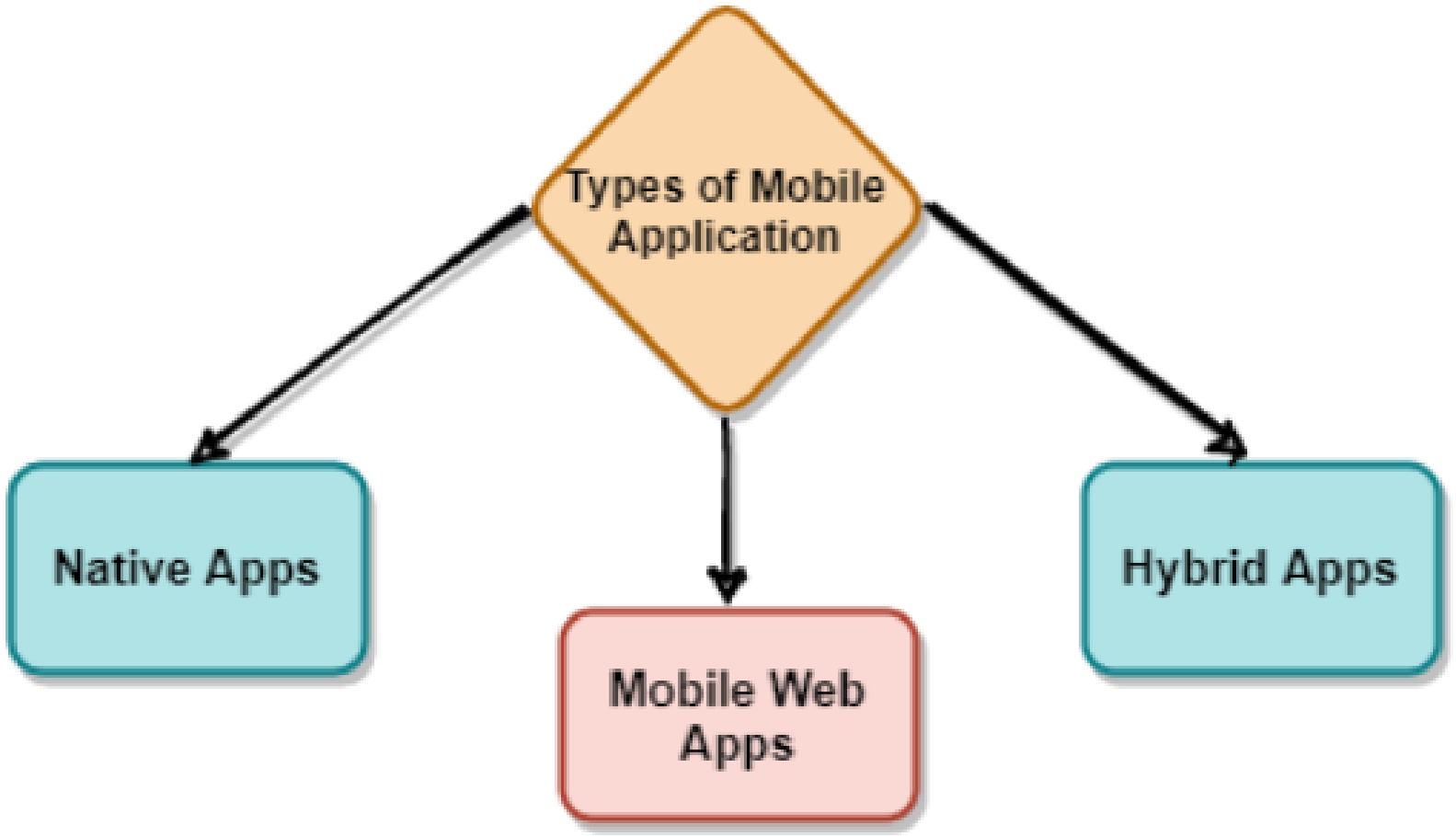
❑**Types of Mobile Testing:** Mobile testing has two types that we can perform on mobile devices.

1. **Hardware Testing:** Hardware testing is done on the internal processors, resolution, internal hardware, size of the screen, radio, space, camera, Bluetooth or WIFI, etc. This testing is known as simple "mobile testing.“
2. **Software Testing or the Application Testing:** The functionality of those applications should be tested, which works on mobile devices. We call this testing as the "Mobile Application Testing".



Mobile App Testing

❑**Types of Mobile Applications:** There are three types of applications.



Mobile App Testing

❑**Types of Mobile Applications:** There are three types of applications.

1. **Native Apps:** Native application is used on different platforms like mobile and tablets.
2. **Mobile Web Apps:** Mobile apps are the server-side apps. We can access the mobile web apps on mobile by using the different browsers like Chrome, Firefox, after connecting the mobile to the mobile network or wireless network like WIFI.
3. **Hybrid Apps:** Hybrid applications are a combination of both types of applications like native and web applications. Hybrid apps run offline on the devices. Hybrid apps are written by using the web technologies like HTML5 and CSS.

Mobile Automation Test and Tools

- Mobile automation testing uses automated testing tools and scripts to assess the functionality, performance, and usability of mobile applications on various device-platforms-OS combinations.
- The primary goal of mobile automation testing is to ensure the quality and reliability of mobile apps by detecting and preventing defects and regressions.
- It involves writing and executing test scripts that automate interactions with a mobile app.
- These scripts simulate user actions like tapping buttons, entering text, swiping, and scrolling.

Mobile Automation Test and Tools

❑Mobile Testing Tools:

1. **Appium:** is a popular open-source framework used for automated mobile app testing. It allows developers to automate the testing of native or hybrid iOS and Android applications. Appium doesn't work alone. It runs the test cases using the WebDriver interface.
2. **NightwatchJS :** is a Node.js based framework that is developed and maintained by BrowserStack. Nightwatch uses Appium under the hood to achieve mobile application automation on virtual simulators & real devices. Nightwatch also takes care of the entire installation with just a single command.
3. **Calabash:** is a mobile test automation framework that works with multiple languages. It supports Ruby, Java, Flex, and .NET. Testers can use APIs to enable native applications that run on touchscreen devices. This framework has libraries that allow test scripts to interact programmatically with native and hybrid apps.
4. **XCUITest:** is Apple's native automation framework for testing iOS applications. Among mobile testing tools, this one is best known for testing iOS apps. Launched by Apple in 2015, the XCUITest framework is meant to create and run UI tests on iOS apps using Swift / Objective C.

❑Mobile Testing Tools:

5. **EarlGrey:** Developed by Google, EarlGrey is a testing framework beneficial for creating UI and functional tests. EarlGrey 2.0 combines EarlGrey with XCUITest, thus allowing iOS testing along with Android. Google uses EarlGrey to test iOS versions of its apps such as Youtube, Gmail, etc.
6. **Selendroid:** is also known as Selenium for mobile apps for Android. Testers can do native and hybrid mobile application testing using Selendroid. Selendroid can execute parallel test cases on multiple devices, similar to Selenium for cross browser testing.
7. **Espresso:** is a mobile automation framework from Google that enables creating and deploying of UI tests for Android applications. Since testing the user interface of an application is essential before deploying it, app developers and app testers widely use Espresso. Developers can create a scenario and record how they interact with a device. Next, they can add assertions that verify the behavior of the UI elements of the app by capturing snapshots. The tool saves the recording and generates UI test cases that they can use to test their application.

Note for Students

- This power point presentation is for lecture, therefore it is suggested that also utilize the text books and lecture notes.