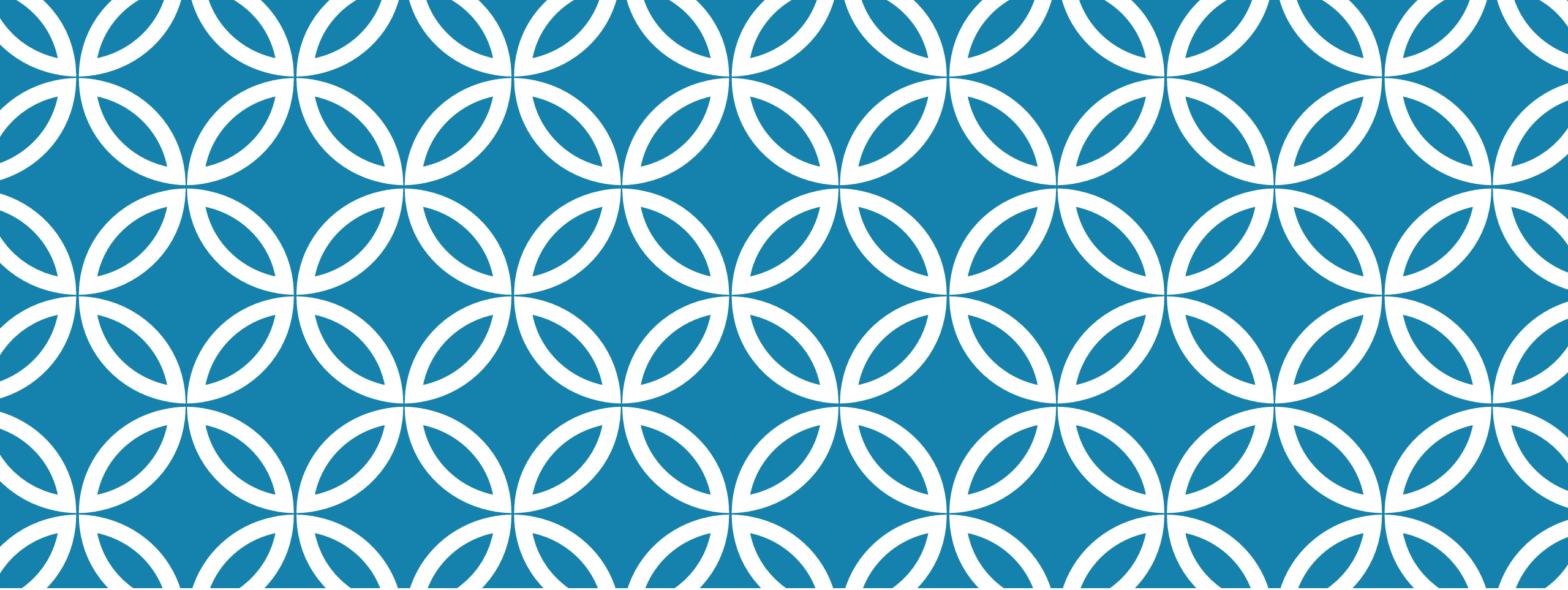


CHAPTER – 7
SYSTEM INTEGRATION TESTING

OUTLINE OF THE CHAPTER

- The Concept of Integration Testing
- Different Types of Interfaces
- Different Types of Interface Errors
- Granularity of System Integration Testing
- System Integration Techniques: Incremental, Top-down, Bottom-up, and Sandwich and Big-bang
- Test Plan for System Integration
- Off-the-shelf Component Testing

THE CONCEPT OF INTEGRATION TESTING

- A software module is a self-contained element of a system
- Modules are individually tested commonly known as *unit testing*
- Next major task is to put the modules, i.e., pieces together to construct the complete system
- Construction of a working system from the pieces is not a straightforward task because of numerous *interface errors*
- The objective of *system integration testing* (SIT) is to build a "working" version of the system
  - Putting modules together in an incremental manner
  - Ensuring that the additional modules work as expected without disturbing the functionalities of the modules already put together
- *Integration testing* is said to be complete when
  - The system is fully integrated together
  - All the test cases have been executed
  - All the severe and moderated defects found have been fixed

THE CONCEPT OF INTEGRATION TESTING

- Interface errors cannot be detected by performing unit testing on modules since unit testing causes computation to happen within a module, whereas interactions are required to happen between modules for interface errors to be detected.

The major advantages of conducting SIT are as follows:

- Defects are detected early
- It is easier to fix defects detected earlier
- We get earlier feedback on the health and acceptability of the individual modules and on the overall system
- Scheduling of defect fixes is flexible, and it can overlap with the development

DIFFERENT TYPES OF INTERFACES

- Modularization is an important principle in software design, and modules are interfaced with other modules to realize the system's functional requirements.
- An interface between two modules allows one module to access the service provided by the other.
- It implements a mechanism for passing control and data between modules.
- Three common paradigms for interfacing modules:
  - Procedure call interface
  - Shared memory interface
  - Message passing interface
- The problem arises when we "put modules together" because of interface errors
- **Interface errors**
  - Interface errors are those that are associated with structures existing outside the local environment of a module, but which the module uses

DIFFERENT TYPES OF INTERFACE ERRORS

1. Construction
2. Inadequate functionality
3. Location of functionality
4. Changes in functionality
5. Added functionality
6. Misuse of interface
7. Misunderstanding of interface
8. Data structure alteration
9. Inadequate error processing
10. Additions to error processing
11. Inadequate post-processing
12. Inadequate interface support
13. Initialization/value errors
14. Validation of data constraints
15. Timing/performance problems
16. Coordination changes
17. Hardware/software interfaces

GRANULARITY OF SYSTEM INTEGRATION TESTING

- System Integration testing is performed at different levels of granularity
- Intra-system testing
  - This form of testing constitutes low-level integration testing with the objective of combining the modules together to build a cohesive system
- Inter-system testing
  - It is a high-level testing phase that requires interfacing independently tested systems
  - For example, Integrating a call control system and a billing system in a telephone network is another example of intersystem testing
- Pairwise testing
  - In pairwise integration, only two interconnected systems in an overall system are tested at a time
  - The purpose of pairwise testing is to ensure that two systems under consideration can function together, assuming that the other systems within the overall environment behave as expected

SYSTEM INTEGRATION TECHNIQUES

- One of the objectives of integration testing is to combine the software modules into a working system so that system-level tests can be performed on the complete system.
- Integration testing need not wait until all the modules of a system are coded and unit tested. Instead, it can begin as soon as the relevant modules are available.
- Common approaches to perform system integration testing
  - Incremental
  - Top-down
  - Bottom-up
  - Sandwich
  - Big-bang

INCREMENTAL

- Integration testing is conducted in an incremental manner as a series of test cycles
- In each test cycle, a few more modules are integrated with an existing and tested build to generated larger builds
- The complete system is built, cycle by cycle until the whole system is operational for system-level testing.
- The number of SIT cycles and the total integration time are determined by the following parameters:
  - Number of modules in the system
  - Relative complexity of the module (cyclomatic complexity)
  - Relative complexity of the interfaces between the modules
  - Number of modules needed to be clustered together in each test cycle
  - Whether the modules to be integrated have boon adequately tested before
  - Turnaround time for each test-debug-fix cycle

INCREMENTAL

- A software image is a compiled software binary
- A build is an interim software image for internal testing within an organization
- Constructing a build is a process by which individual modules are integrated to form an interim software image.
- The final build is a candidate for system testing
- Constructing a software image involves the following activities
  - Gathering the latest unit tested, authorized versions of modules
  - Compiling the source code of those modules
  - Checking in the compiled code to the repository
  - Linking the compiled modules into subassemblies
  - Verifying that the subassemblies are correct
  - Exercising version control

INCREMENTAL

- Creating a daily build is very popular among many organization
- It facilitates to a faster delivery of the system
- It puts emphasis on small incremental testing
- It steadily increases number of test cases
- The system is tested using automated, re-usable test cases
- An effort is made to fix the defects that were found within 24 hours
- Prior version of the build are retained for references and rollback
- A typical practice is to retain the past 7-10 builds

INCREMENTAL

A release note containing the following information accompanies a build.

- What has changed since the last build?
- What outstanding defects have been fixed?
- What are the outstanding defects in the build?
- What new modules, or features, have been added?
- What existing modules, or features, have been enhanced, modified, or deleted?
- Are there any areas where unknown changes may have occurred?

A test strategy is created for each new build and the following issues are addressed while planning a test strategy

- What test cases need to be selected from the SIT test plan?
- What previously failed test cases should now be re-executed in order to test the fixes in the new build?
- How to determine the scope of a partial regression tests?
- What are the estimated time, resource demand, and cost to test this build?

TOP-DOWN

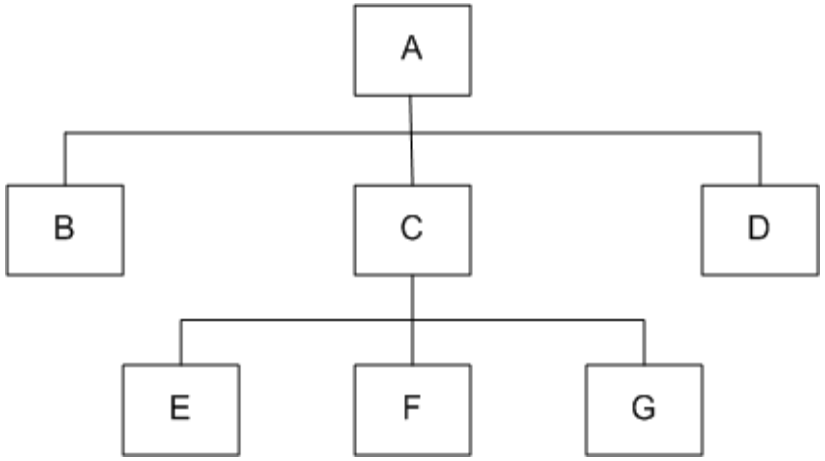


Figure 7.1: A module hierarchy with three levels and seven modules

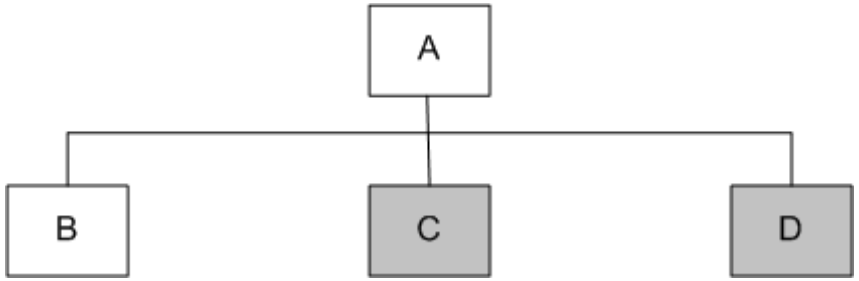


Figure 7.2: Top-down integration of modules A and B

Module A has been decomposed into modules B, C, and D

Modules B, D, E, F, and G are terminal modules

First integrate modules A and B using stubs C` and D` (represented by grey boxes)

Next stub D` has been replaced with its actual instance D

Two kinds of tests are performed:

- Test the interface between A and D
- Regression tests to look for interface defects between A and B in the presence of module D

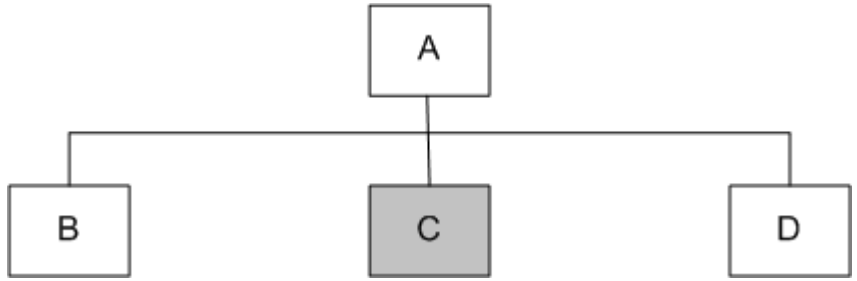


Figure 7.3: Top-down integration of modules A, B and D

TOP-DOWN

Stub C` has been replaced with the actual module C, and new stubs E`, F`, and G`

Perform tests as follows:

- first, test the interface between A and C;
- second, test the combined modules A, B, and D in the presence of C

The rest of the process depicted in the right hand side figures.

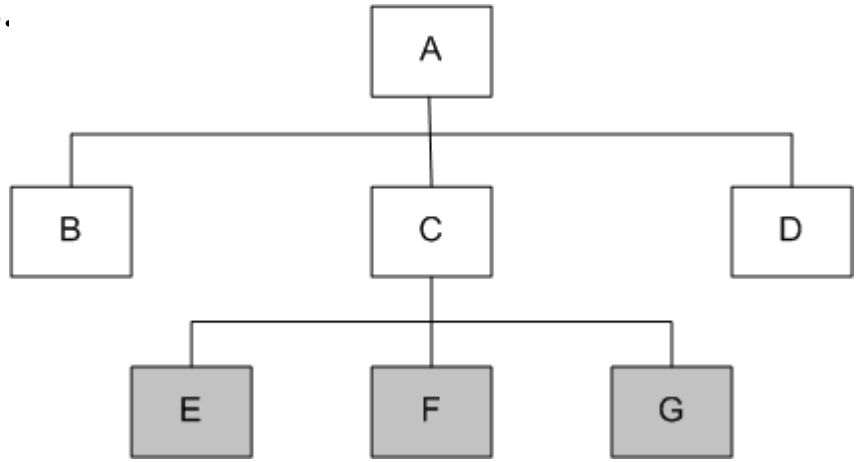


Figure 7.4: Top-down integration of modules A, B, D and C

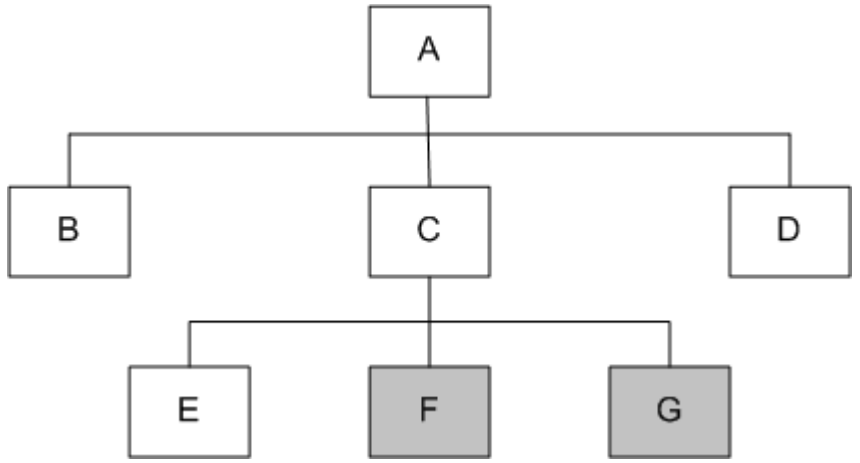


Figure 7.5: Top-down integration of modules A, B, C, D and E

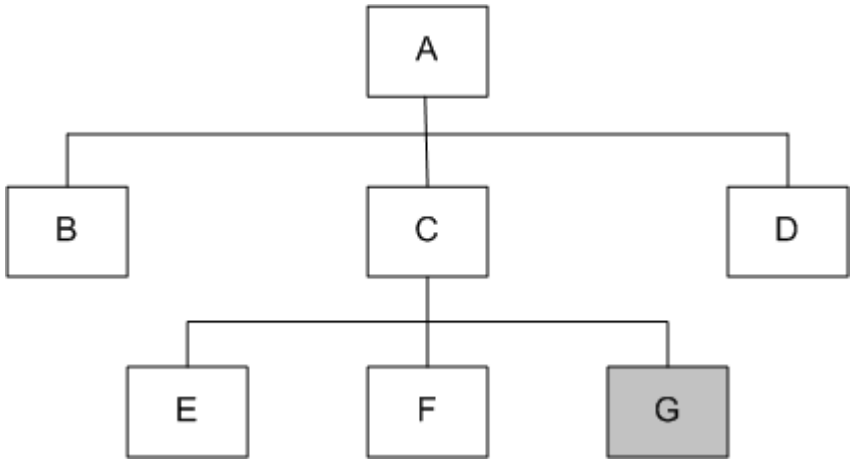


Figure 7.6: Top-down integration of modules A, B, C, D, E and F

TOP-DOWN

- **Advantages**
  - The SIT engineers continually observe system-level functions as the integration process continue
  - Isolation of interface errors becomes easier because of the incremental nature of the top-down integration
  - Test cases designed to test the integration of a module M are reused during the regression tests performed after integrating other modules
- **Disadvantages**
  - It may not be possible to observe meaningful system functions because of the absence of lower-level modules and the presence of stubs.
  - Test case selection and stub design become increasingly difficult when stubs lie far away from the top-level module.

BOTTOM-UP

- We design a test driver to integrate lowest-level modules E, F, and G
- Return values generated by one module are likely to be used in another module
- The test driver mimics module C to integrate E, F, and G in a limited way.
- The test driver is replaced with actual module , i.e., C.
- A new test driver is used
- At this moment, more modules such as B and D are integrated
- The new test driver mimics the behavior of module A
- Finally, the test driver is replaced with module A and further test are performed

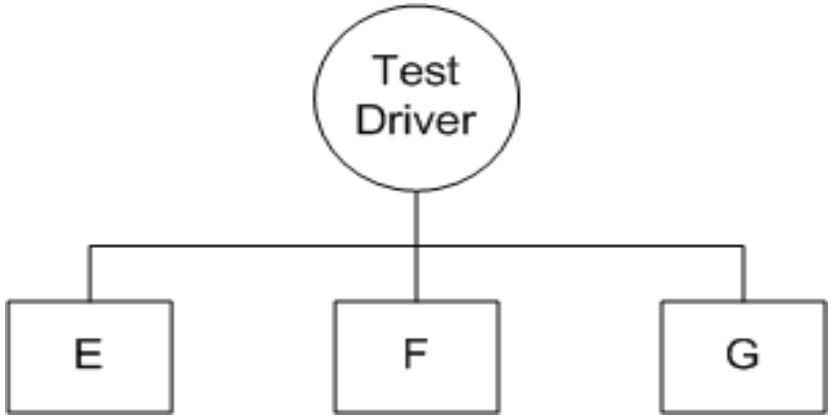


Figure 7.8: Bottom-up integration of module E, F, and G

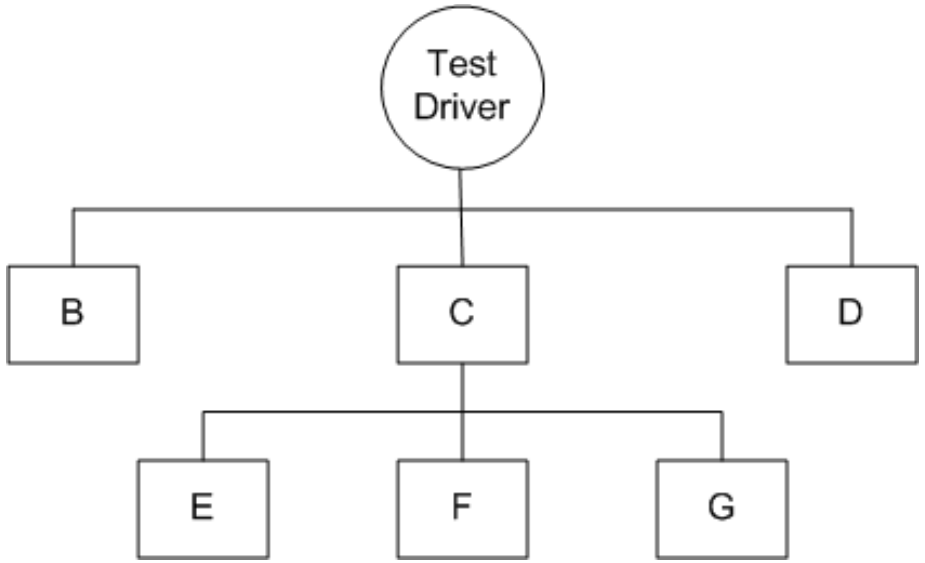


Figure 7.9: Bottom-up integration of module B, C, and D with F, F, and G

BOTTOM-UP

- **Advantages**
  - One designs the behavior of a test driver by simplifying the behavior of the actual module
  - If the low-level modules and their combined functions are often invoked by other modules, then it is more useful to test them first so that meaningful effective integration of other modules can be done
- **Disadvantages**
  - Discovery of major faults are detected towards the end of the integration process, because major design decisions are embodied in the top-level modules
  - Test engineers can not observe system-level functions from a partly integrated system. In fact, they can not observe system-level functions until the top-level test driver is in place

BIG-BANG AND SANDWICH

- **Big-bang Approach**
  - First all the modules are individually tested
  - Next all those modules are put together to construct the entire system which is tested as a whole
- **Sandwich Approach**
  - In this approach a system is integrated using a mix of top-down, bottom-up, and big-bang approaches
  - A hierarchical system is viewed as consisting of three layers
  - The bottom-up approach is applied to integrate the modules in the bottom-layer
  - The top layer modules are integrated by using top-down approach
  - The middle layer is integrated by using the big-bang approach after the top and the bottom layers have been integrated

TEST PLAN FOR SYSTEM INTEGRATION

1. Scope of Testing
2. Structure of the Integration Levels a. Integration test phases b. Modules or subsystems to be integrated in each phase c. Building process and schedule in each phase d. Environment to be set up and resources required in each phase
3. Criteria for Each Integration Test Phase n a. Entry criteria b. Exit criteria c. Integration Techniques to be used d. Test configuration set-up
4. Test Specification for Each Integration Test Phase a. Test case id# b. Input data c. Initial condition d. Expected results e. Test procedure How to execute this test? How to capture and interpret the results?
5. Actual Test Results for Each Integration Test Phase
6. References
7. Appendix

Table 7.3: A framework for System Integration Test (SIT) Plan.

TEST PLAN FOR SYSTEM INTEGRATION

Categories of System Integration Tests:

Interface integrity
- Internal and external interfaces are tested as each module is integrated

Functional validity
- Tests to uncover functional errors in each module after it is integrated

End-to-end validity
- Tests are designed to ensure that a completely integrated system works together from end-to-end

Pairwise validity
- Tests are designed to ensure that any two systems work properly when connected by a network

Interface stress
- Tests are designed to ensure that the interfaces can sustain the load

System endurance
- Tests are designed to ensure that the integrated system stay up for weeks without any crashes

OFF-THE-SELF COMPONENT INTEGRATION

Organization occasionally purchase off-the-self (OTS) components from vendors and integrate them with their own components

Useful set of components that assists in integrating actual components:

Wrapper: It is a piece of code that one builds to isolate the underlying components from other components of the system.

Glue: A glue component provides the functionality to combine different components

Tailoring: Components tailoring refers to the ability to enhance the functionality of a component

- Tailoring is done by adding some elements to a component to enrich it with functionality not provided by the vendor
- Tailoring does not involve modifying the source code of the component
- For example, executing some “script” when some event occurs.

OFF-THE-SHELF COMPONENT TESTING

OTS components produced by the vendor organizations are known as **commercial off-the-shelf** (COTS) components

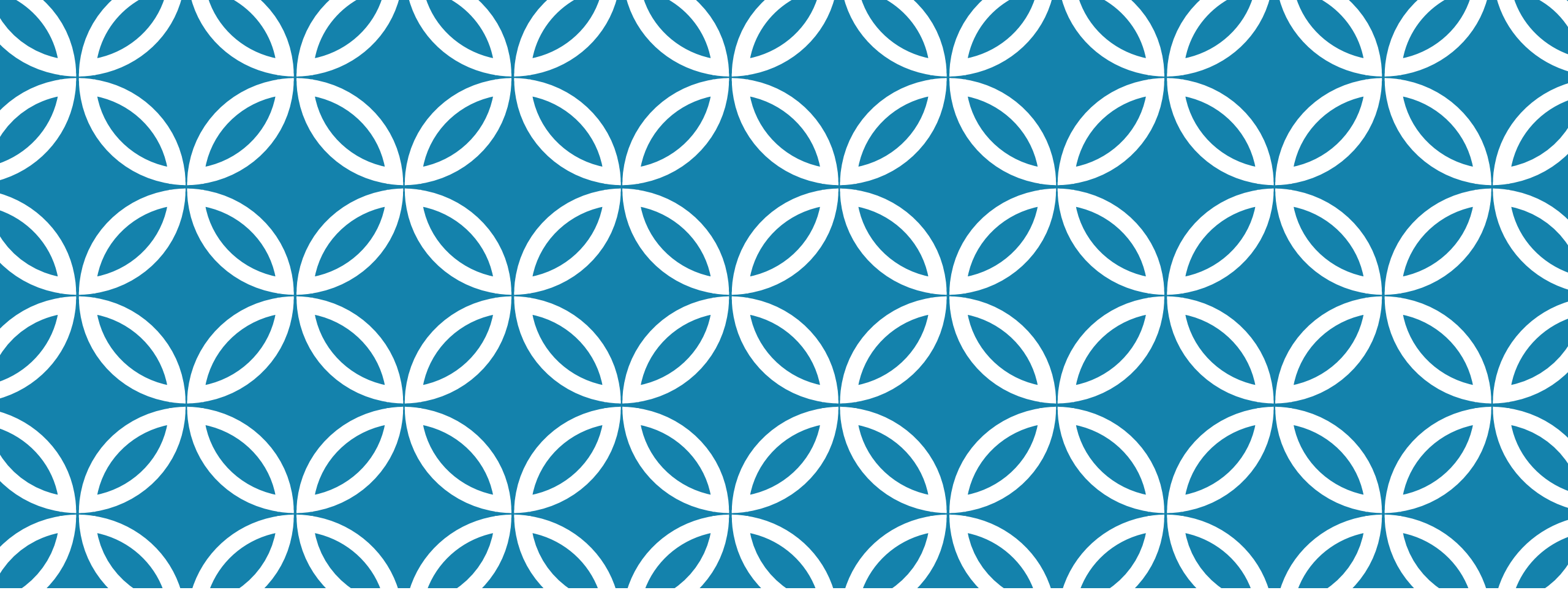
A COTS component is defined as: "*A unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*"

Three types of testing techniques are used to determine the suitability of a COTS component:

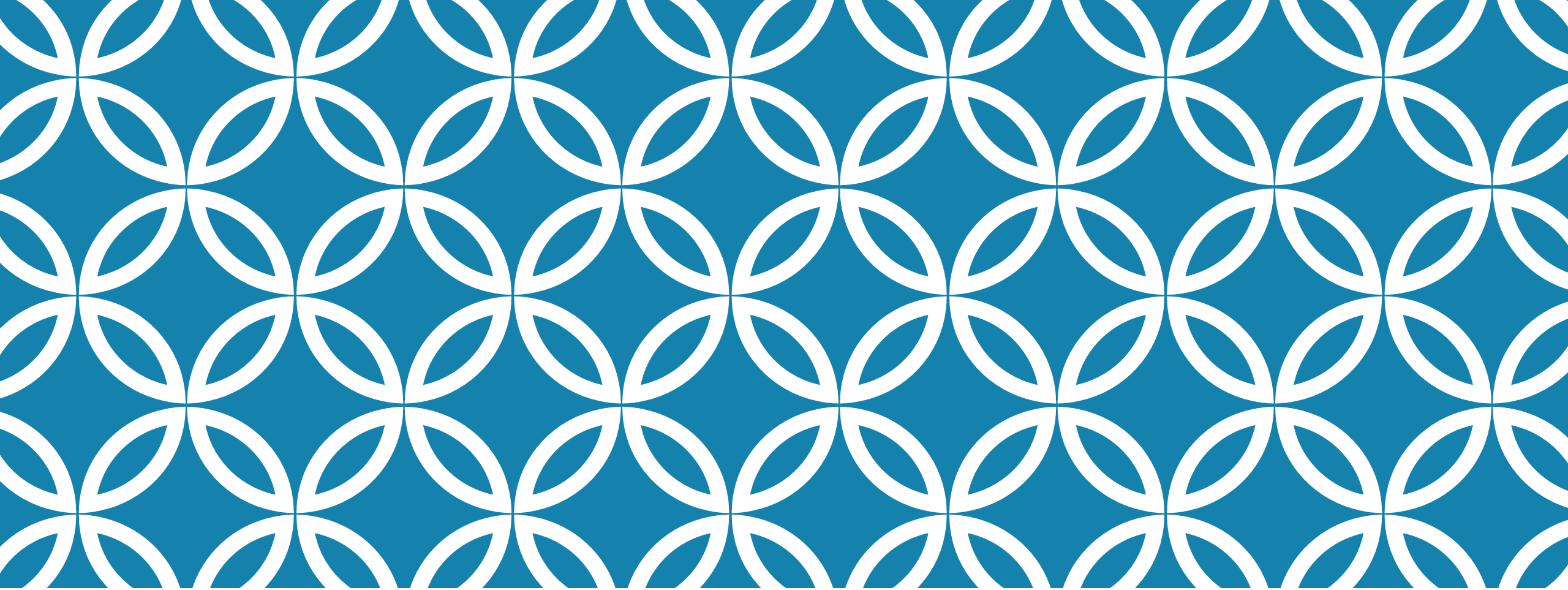
Black-box component testing: This is used to determine the quality of the component

System-level fault injection testing: This is used to determine how well a system will tolerate a failing component

Operational system testing: These kinds of tests are used to determine the tolerance of a software system when the COTS component is functioning correctly to determine if the OTS component is a good match for the system.



THANK YOU!!



CHAPTER – 8
SYSTEM TEST CATEGORIES

OUTLINE OF THE CHAPTER

- Taxonomy of System Tests
- Basic Tests
- Functionality Tests
- Robustness Tests
- Interoperability Tests
- Performance Tests
- Scalability Tests
- Stress Tests
- Load and Stability Tests
- Regression Tests
- Documentation Tests
- Regulatory Tests

TAXONOMY OF SYSTEM TESTS

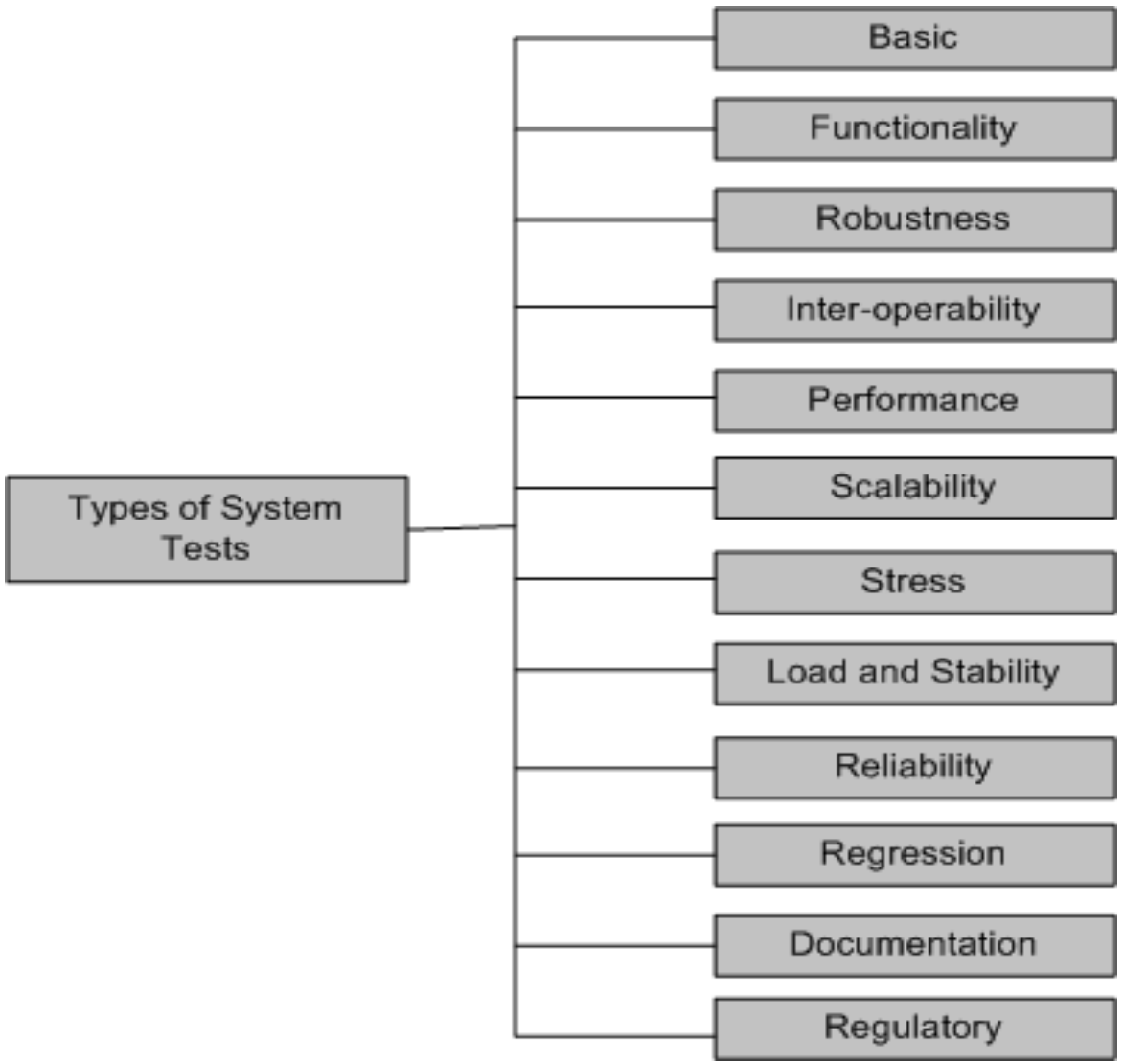


Figure 8.1: Types of system tests

TAXONOMY OF SYSTEM TESTS

- **Basic tests** provide an evidence that the system can be installed, configured and be brought to an operational state
- **Functionality tests** provide comprehensive testing over the full range of the requirements, within the capabilities of the system
- **Robustness tests** determine how well the system recovers from various input errors and other failure situations
- **Inter-operability tests** determine whether the system can inter-operate with other third party products
- **Performance tests** measure the performance characteristics of the system, e.g., throughput and response time, under various conditions

TAXONOMY OF SYSTEM TESTS

- **Scalability tests** determine the scaling limits of the system, in terms of user scaling, geographic scaling, and resource scaling
- **Stress tests** put a system under stress in order to determine the limitations of a system and, when it fails, to determine the manner in which the failure occurs
- **Load and Stability** tests provide evidence that the system remains stable for a long period of time under full load
- **Reliability tests** measure the ability of the system to keep operating for a long time without developing failures
- **Regression tests** determine that the system remains stable as it cycles through the integration of other subsystems and through maintenance tasks
- **Documentation tests** ensure that the system's user guides are accurate and usable

BASIC TESTS

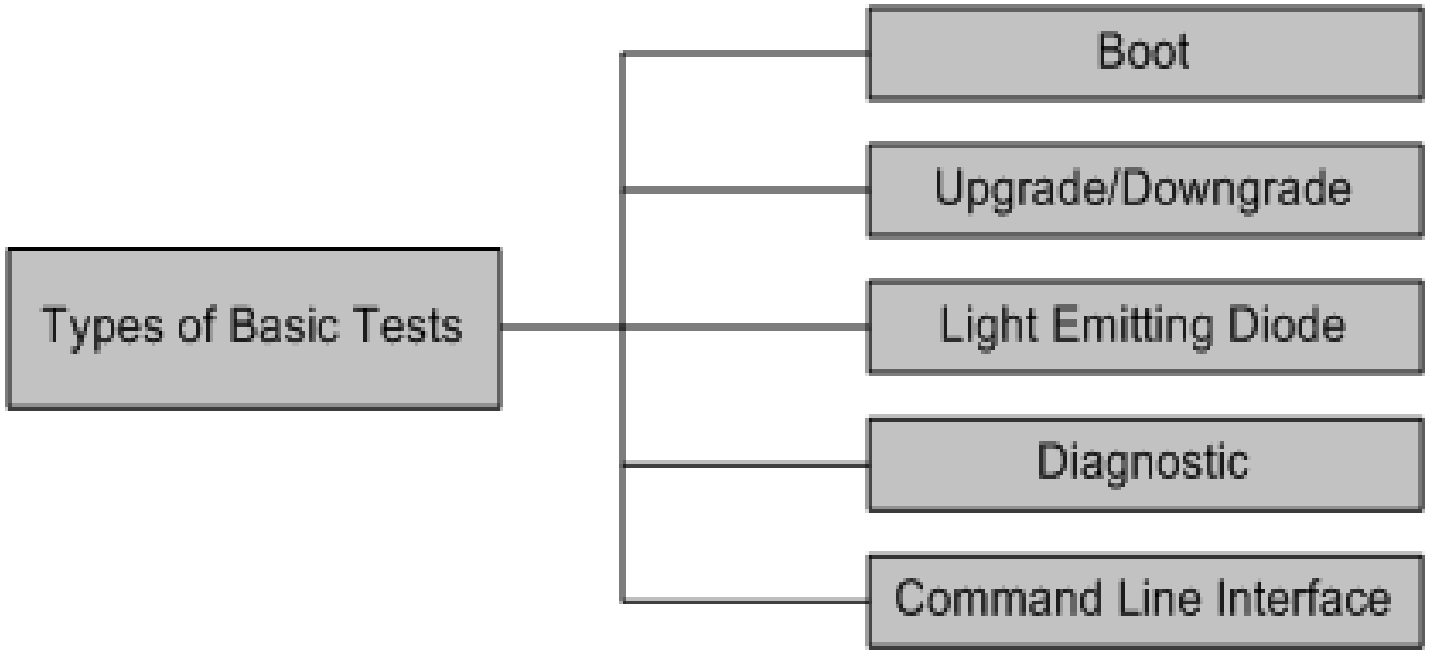


Figure 8.2: Types of basic tests

Boot: Boot tests are designed to verify that the system can boot up its software image (or, build) from the supported boot options

Upgrade/Downgrade: Upgrade/downgrade tests are designed to verify that the system software can be upgraded or downgraded (rollback) in a graceful manner

BASIC TESTS

Light Emitting Diode: The LED (Light Emitting Diode) tests are designed to verify that the system LED status indicators functioning as desired

Diagnostic: Diagnostic tests are designed to verify that the hardware components (or, modules) of the system are functioning as desired

- **Power-On Self Test**
- **Ethernet Loop Back Test**
- **Bit Error Test**

Command line Interface: Command Line Interface (CLI) tests are designed to verify that the system can be configured

FUNCTIONALITY TESTS

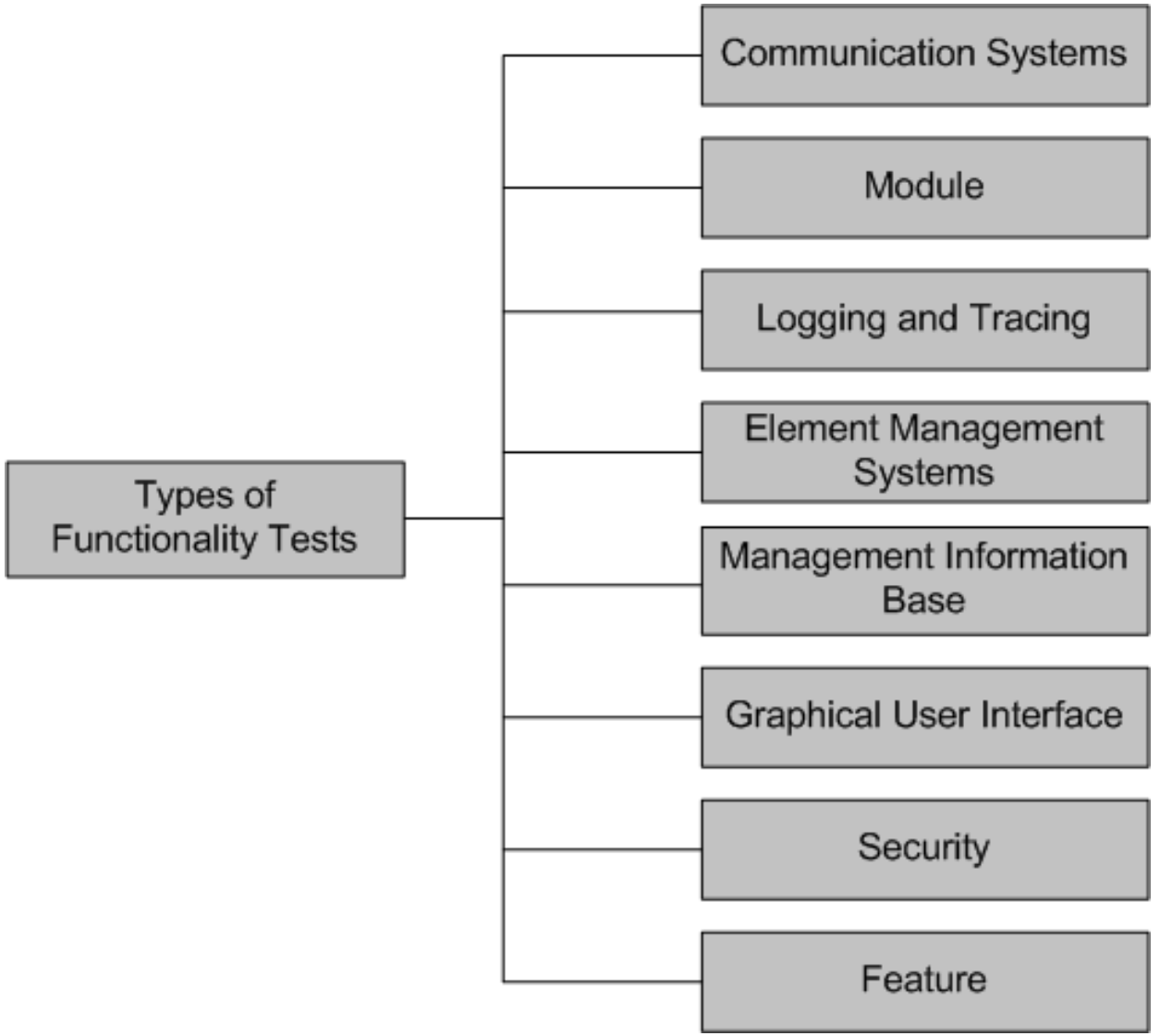


Figure 8.3: Types of functionality tests

FUNCTIONALITY TESTS

Communication Systems Tests

- These tests are designed to verify the implementation of the communication systems as specified in the customer requirements specification
- Four types of communication systems tests are recommended
  - Basis interconnection tests
  - Capability tests
  - Behavior tests
  - System resolution tests

Module Tests

- Module Tests are designed to verify that all the modules function individually as desired within the systems
- The idea here is to ensure that individual modules function correctly within the whole system.
  - For example, an Internet router contains modules such as line cards, system controller, power supply, and fan tray. Tests are designed to verify each of the functionalities

FUNCTIONALITY TESTS

Logging and Tracing Tests

- Logging and Tracing Tests are designed to verify the configurations and operations of logging and tracing
- This also includes verification of “flight data recorder: non-volatile Flash memory” logs when the system crashes

Element Management Systems (EMS) Tests

- EMS tests verifies the main functionalities, which are to manage, monitor and upgrade the communication systems network elements
- It includes both EMS client and EMS servers functionalities

Management Information Base (MIB) Tests

- MIB tests are designed to verify
  - Standard MIBs including MIB II
  - Enterprise MIBs specific to the system

FUNCTIONALITY TESTS

Graphical User Interface Tests

- Tests are designed to look-and-feel the interface to the users of an application system
- Tests are designed to verify different components such as icons, menu bars, dialog boxes, scroll bars, list boxes, and radio buttons
- The GUI can be utilized to test the functionality behind the interface, such as accurate response to database queries
- Tests the usefulness of the on-line help, error messages, tutorials, and user manuals
- The usability characteristics of the GUI is tested, which includes the following
  - ***Accessibility:*** Can users enter, navigate, and exit with relative ease?
  - ***Responsiveness:*** Can users do what they want and when they want in a way that is clear?
  - ***Efficiency:*** Can users do what they want to with minimum number of steps and time?
  - ***Comprehensibility:*** Do users understand the product structure with a minimum amount of effort?

FUNCTIONALITY TESTS

Security Tests

- Security tests are designed to verify that the system meets the security requirements
  - Confidentiality
    - It is the requirement that data and the processes be protected from unauthorized disclosure
  - Integrity
    - It is the requirement that data and process be protected from unauthorized modification
  - Availability
    - It is the requirement that data and processes be protected form the denial of service to authorized users
- Security test scenarios should include negative scenarios such as misuse and abuse of the software system

FUNCTIONALITY TESTS

Security Tests (cont'd) : useful types of security tests includes the following:

- Verify that only authorized accesses to the system are permitted
- Verify the correctness of both encryption and decryption algorithms for systems where data/messages are encoded.
- Verify that illegal reading of files, to which the perpetrator is not authorized, is not allowed
- Ensure that virus checkers prevent or curtail entry of viruses into the system
- Ensure that the system is available to authorized users when a zero-day attack occurs
- Try to identify any "backdoors" in the system usually left open by the software developers

FUNCTIONALITY TESTS

Feature Tests

- These tests are designed to verify any additional functionalities which are defined in requirement specification but not covered in the functional category discussed
- Examples
  - Data conversion testing
  - Cross-functionality testing

ROBUSTNESS TESTS

Robustness means how much sensitive a system is to erroneous input and changes its operational environment

Tests in this category are designed to verify how gracefully the system behaves in error situations and ina a changed operational environment

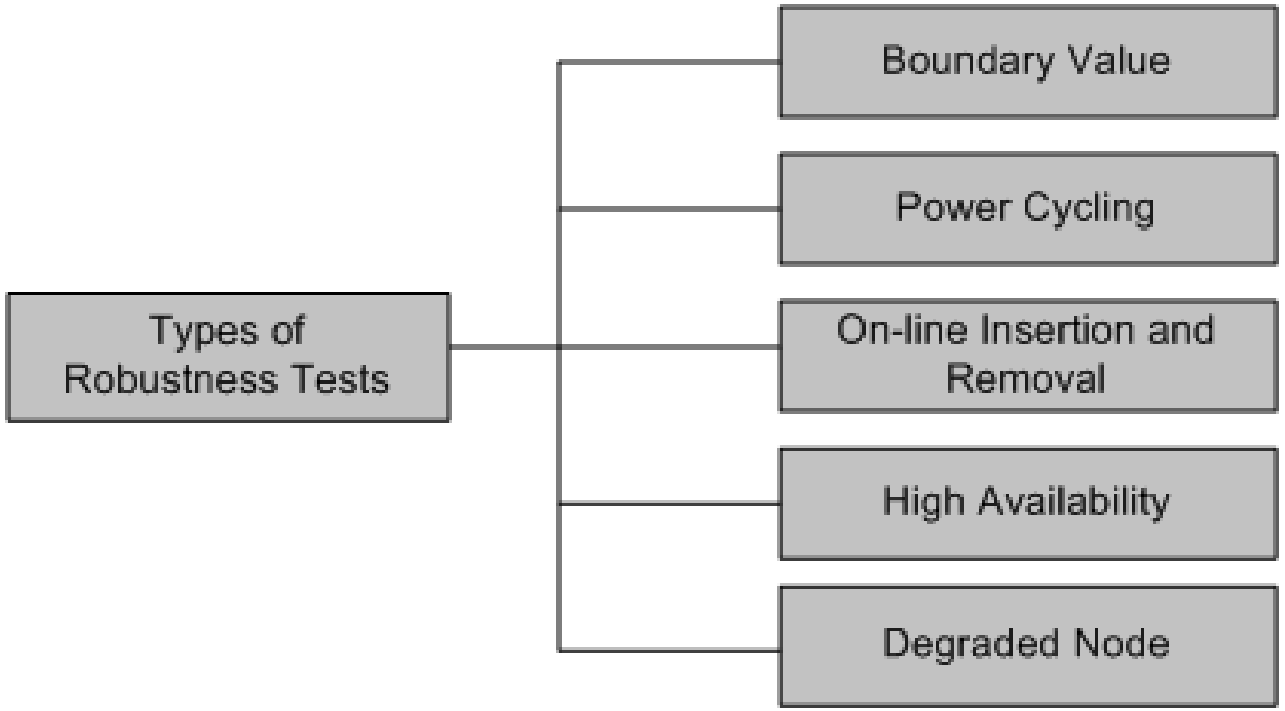


Figure 8.4: Types of robustness tests

ROBUSTNESS TESTS

Boundary value

- Boundary value tests are designed to cover boundary conditions, special values, and system defaults
- The tests include providing invalid input data to the system and observing how the system reacts to the invalid input.

Power cycling

- Power cycling tests are executed to ensure that, when there is a power glitch in a deployment environment, the system can recover from the glitch to be back in normal operation after power is restored

On-line insertion and removal

- On-line Insertion and Removal (OIR) tests are designed to ensure that on-line insertion and removal of modules, incurred during both idle and heavy load operations, are gracefully handled and recovered

ROBUSTNESS TESTS

High Availability

- The concept of high availability is also known as **fault tolerance**
- High availability tests are designed to verify the redundancy of individual modules, including the software that controls these modules.
- The goal is to verify that the system gracefully and quickly recovers from hardware and software failures without adversely impacting the operation of the system
- High availability is realized by means of proactive methods to maximize service up-time, and to minimize the downtime

Degraded Node

- Degraded node (also known as failure containment) tests verify the operation of a system after a portion of the system becomes non-operational
- It is a useful test for all mission-critical applications.

INTEROPERABILITY TESTS

Tests are designed to verify the ability of the system to inter-operate with third party products

The re-configuration activities during interoperability tests is known as configuration testing

Another kind of inter-operability tests is called (backward) compatibility tests

- Compatibility tests verify that the system works the same way across different platforms, operating systems, data base management systems
- Backward compatibility tests verify that the current software build flawlessly works with older version of platforms

PERFORMANCE TESTS

Tests are designed to determine the performance of the actual system compared to the expected one

Tests are designed to verify response time, execution time, throughput, resource utilization and traffic rate

One needs to be clear about the specific data to be captured in order to evaluate performance metrics.

For example, if the objective is to evaluate the response time, then one needs to capture

- End-to-end response time (as seen by external user)
- CPU time
- Network connection time
- Database access time
- Network connection time
- Waiting time

SCALABILITY TESTS

Tests are designed to verify that the system can scale up to its engineering limits

Scaling tests are conducted to ensure that the system response time remains the same, or increases by a small amount, as the number of users are increased.

There are three major causes of these limitations:

- data storage limitations
- network bandwidth limitations
- speed limit

Extrapolation is often used to predict the limit of scalability

STRESS TESTS

The goal of stress testing is to evaluate and determine the behavior of a software component while the offered load is in excess of its designed capacity

The system is deliberately stressed by pushing it to and beyond its specified limits

It ensures that the system can perform acceptably under worst-case conditions, under an expected peak load. If the limit is exceeded and the system does fail, then the recovery mechanism should be invoked

Stress tests are targeted to bring out the problems associated with one or more of the following:

- Memory leak
- Buffer allocation and memory carving

LOAD AND STABILITY TESTS

Tests are designed to ensure that the system remains stable for a long period of time under full load

When a large number of users are introduced and applications that run for months without restarting, a number of problems are likely to occur:

- the system slows down
- the system encounters functionality problems
- the system crashes altogether

Load and stability testing typically involves exercising the system with virtual users and measuring the performance to verify whether the system can support the anticipated load

This kind of testing help one to understand the ways the system will fare in real-life situations

RELIABILITY TESTS

Reliability tests are designed to measure the ability of the system to remain operational for long periods of time.

The reliability of a system is typically expressed in terms of mean time to failure (MTTF)

The average of all the time intervals between successive failures is called the MTTF

After a failure is observed, the developers analyze and fix the defects, which consumes some time – let us call this interval the repair time.

The average of all the repair times is known as the mean time to repair (MTTR)

Now we can calculate a value called mean time between failure (MTBF) as MTBF = MTTF + MTTR

The random testing technique discussed in Chapter 9 is used for reliability measurement

The software reliability modeling and testing is discussed in Chapter 15 in detail

REGRESSION TESTS

In this category, new tests are not designed, instead, test cases are selected from the existing pool and executed

The main idea in regression testing is to verify that no defect has been introduced into the unchanged portion of a system due to changes made elsewhere in the system

During system testing, many defects are revealed and the code is modified to fix those defects

One of four different scenarios can occur for each fix:

- The reported defect is fixed
- The reported defect could not be fixed inspite of making an effort
- The reported defect has been fixed, but something that used to work before has been failing
- The reported defect could not be fixed inspite of an effort, and something that used to work before has been failing

REGRESSION TESTS

One possibility is to re-execute every test case from version n − 1 to version n before testing anything new

A full test of a system may be prohibitively expensive.

A subset of the test cases is carefully selected from the existing test suite to

- maximize the likelihood of uncovering new defects
- reduce the cost of testing

DOCUMENTATION TESTS

Documentation testing means verifying the technical accuracy and readability of the user manuals, tutorials and the on-line help

Documentation testing is performed at three levels:

- ***Read test:*** In this test a documentation is reviewed for clarity, organization, flow, and accuracy without executing the documented instructions on the system
- ***Hands-on test:*** Exercise the on-line help and verify the error messages to evaluate their accuracy and usefulness.
- ***Functional test:*** Follow the instructions embodied in the documentation to verify that the system works as it has been documented.

REGULATORY TESTS

- In this category, the final system is shipped to the regulatory bodies in those countries where the product is expected to be marketed
- The idea is to obtain compliance marks on the product from various countries
- Most of these regulatory bodies issue safety and EMC (electromagnetic compatibility)/ EMI (electromagnetic interference) compliance certificates (emission and immunity)
- The regulatory agencies are interested in identifying flaws in software that have potential safety consequences
- The safety requirements are primarily based on their own published standards

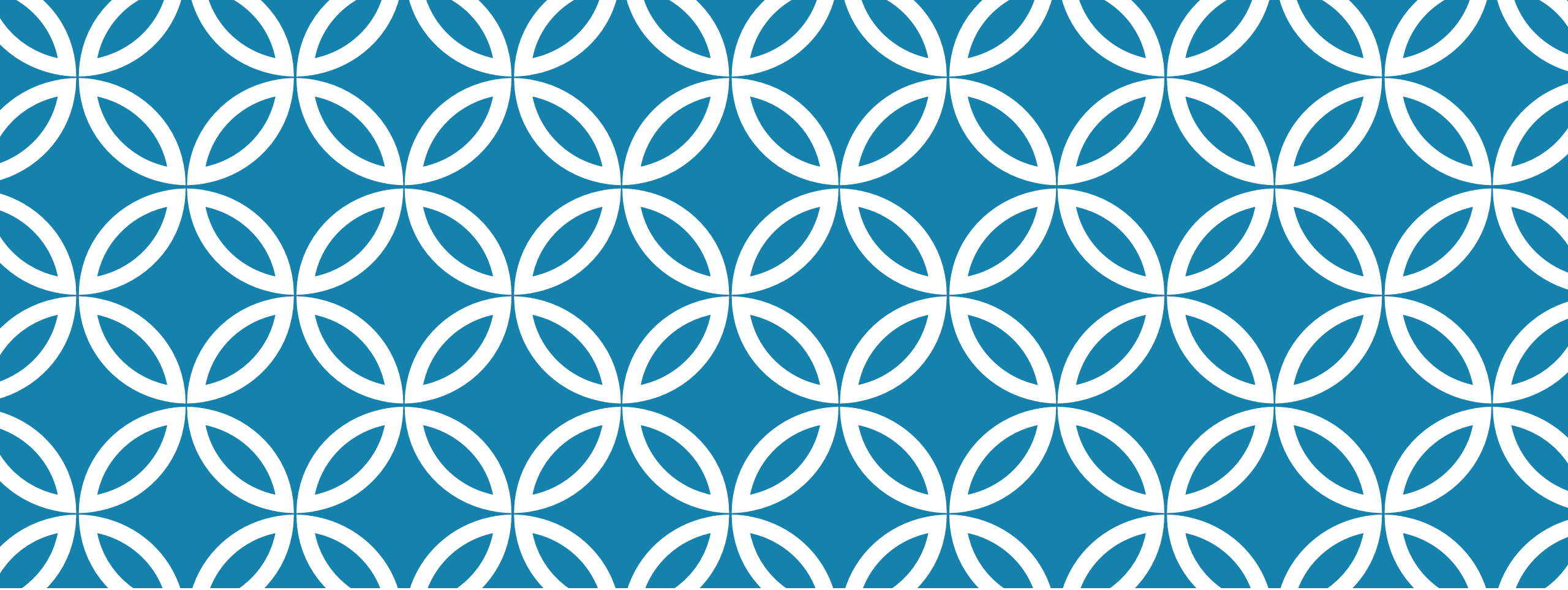
SOFTWARE SAFETY

- A *hazard* is a state of a system or a physical situation which when combined with certain environmental conditions, could lead to an *accident* or *mishap*
- An *accident* or *mishap* is an unintended event or series of events that results in death, injury, illness, damage or loss of property, or harm to the environment
- Software *safety* is defined in terms of hazards
- A software in isolation cannot do physical damage. However, a software in the context of a system and an embedding environment could be vulnerable
- Examples:
  - A software module in a database application is not hazardous by itself, but when it is embedded in a missile navigation system, it could be hazardous
  - If a missile takes a U-turn because of a software error in the navigation system, and destroys the submarine that launched it, then it is not a safe software

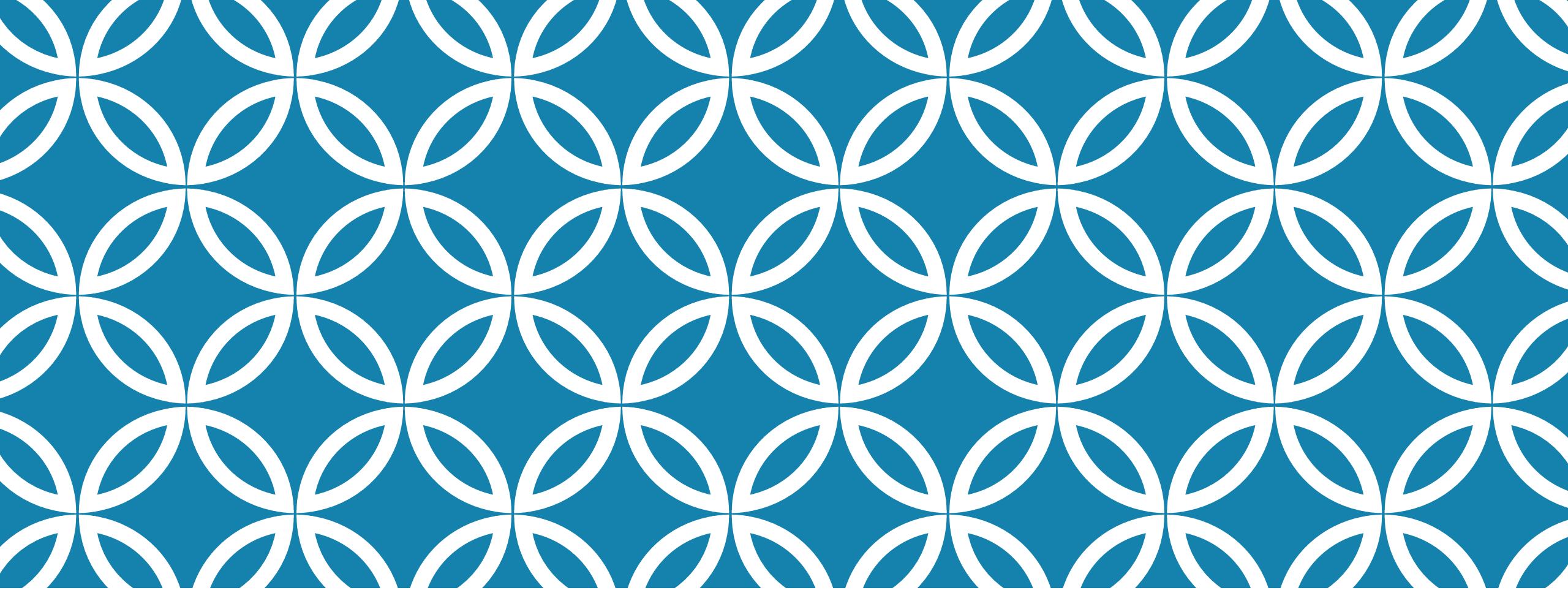
SAFETY ASSURANCE

There are two basic tasks performed by a **safety assurance** engineering team:

- Provide methods for identifying, tracking, evaluating, and eliminating hazards associated with a system
- Ensure that safety is embedded into the design and implementation in a timely and cost-effective manner, such that the risk created by the user/operator error is minimized



THANK YOU!!



CHAPTER – 9
FUNCTIONAL TESTING

OUTLINE OF THE CHAPTER

- Functional Testing Concepts of Howden
- Different Types of Variables
- Test Vectors
- Howden's Functional Testing Summary
- Pairwise Testing
- Orthogonal Array
- In Parameter Order
- Equivalence Class Partitioning
- Guidelines for Equivalence Class Partitioning
- Identification of Test Cases
- Advantages of Equivalence Class Partitioning
- Boundary Value Analysis (BVA)
- Guidelines for Boundary Value Analysis
- Decision Tables
- Random Testing
- Adaptive Random Testing
- Error Guessing
- Category Partition

FUNCTIONAL TESTING CONCEPTS

The four key concepts in functional testing are:

Precisely identify the domain of each input and each output variable

Select values from the data domain of each variable having important properties

Consider combinations of special values from different input domains to design test cases

Consider input values such that the program under test produces special values from the domains of the output variables

DIFFERENT TYPES OF VARIABLES

Numeric Variables

- A set of discrete values
- A few contiguous segments of values

Arrays

- An array holds values of the same type, such as integer and real. Individual elements of an array are accessed by using one or more indices.

Substructures

- A structure means a data type that can hold multiple data elements. In the field of numerical analysis, matrix structure is commonly used.

Subroutine Arguments

- Some programs accept input variables whose values are the names of functions. Such programs are found in numerical analysis and statistical applications

TEST VECTORS

A test vector is an instance of an input to a program, a.k.a. test data

If a program has n input variables, each of which can take on k special values, then there are k^n possible combinations of test vectors

We have more than one million test vectors even for $k = 3$ and $n = 20$

There is a need to identify a method for reducing the number of test vectors

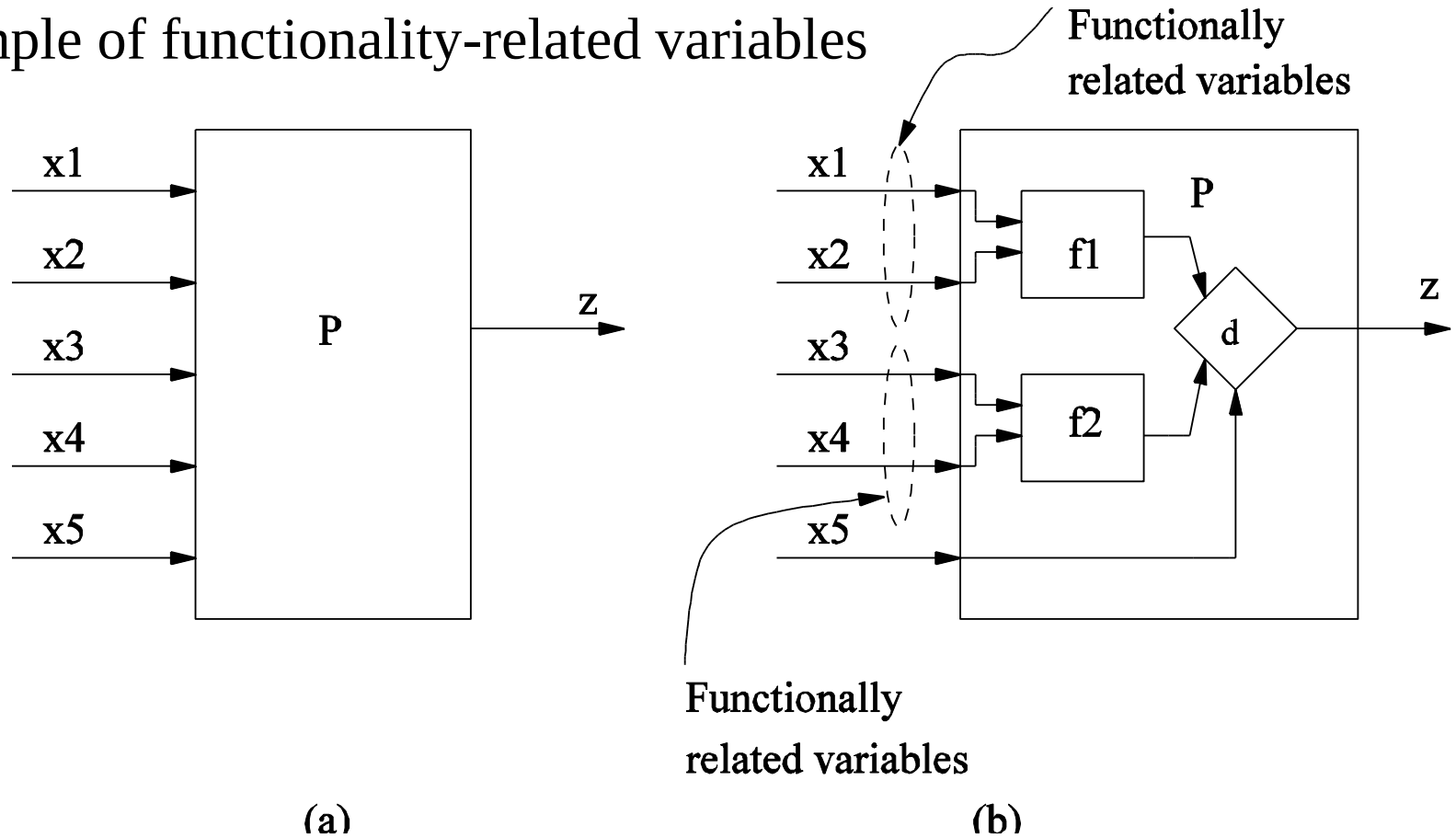
It is suggested that there is no need of combining values of all input variables to design a test vector, if the variables are not *functionally related*

It is difficult to give a formal definition of the idea of functionally related variables, but it is easy to identify them

- Variables appearing in the same assignment statement are functionally related
- Variables appearing in the same branch predicate – the condition part of an if statement, for example – are functionally related

TEST VECTORS

- Example of functionality-related variables



Functionality related variables

HOWDEN'S FUNCTIONAL TESTING SUMMARY

Let us summarize the main points in functional testing:

Identify the input and the output variables of the program and their data domains

Compute the expected outcomes as illustrated in Figure 9.5(a), for selected input values

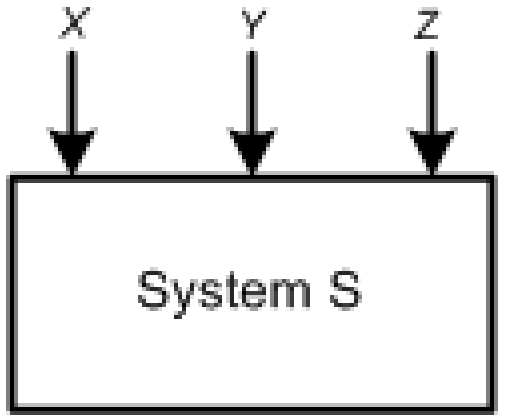
Determine the input values that will cause the program to produce selected outputs as illustrated in Figure 9.5(b).

PAIRWISE TESTING

Pairwise testing means that each possible combination of values for every pair of input variables is covered by at least one test case

Consider the system S in the figure, which has three input variables X, Y, and Z.

For the three given variables X, Y, and Z, their value sets are as follows: D(X) = {True, False}, D(Y) = {0, 5}, and D(Z) = {Q, R}



System S with three input variables.

PAIRWISE TESTING

The total number of all-combination test cases is $2 \times 2 \times 2 = 8$

However, a subset of four test cases, as shown in Table 9.5, covers all pairwise combinations

Test Case Id	Input X	Input Y	Input Z
TC_1	$True$	0	Q
TC_2	$True$	5	R
TC_3	$False$	0	Q
TC_4	$False$	5	R

Table 9.5: Pairwise test cases for system S.

ORTHOGONAL ARRAY

Consider the two-dimensional array of integers shown in Table 9.6

This is an example of $L_4(2^3)$ orthogonal array

The "4" indicates that the array has 4 rows, also known as runs

The "2^3" part indicates that the array has 3 columns, known as factors, and each cell in the array contains 2 different values, known as levels.

Levels mean the maximum number of values that a single factor can take on

Orthogonal arrays are generally denoted by the pattern $L_{Runs}(Levels^{Factors})$

Runs	Factors		
	1	2	3
1	1	1	1
2	1	2	2
3	2	1	2
4	2	2	1

Table 9.6: $L_4(2^3)$ orthogonal array.

ORTHOGONAL ARRAY

Let us consider our previous example of the system S.

The system S which has three input variables X, Y, and Z.

For the three given variables X, Y , and Z, their value sets are as follows: D(X) = {True, False}, D(Y) = {0, 5}, and D(Z) = {Q,R}.

Map the variables to the factors and values to the levels onto the $L_4(2^3)$ orthogonal array (Table 9.6) with the resultant in Table 9.5.

In the first column, let 1 = True, 2 = False.

In the second column, let 1 = 0, 2 = 5.

In the third column, let 1 = Q, 2 = R.

Note that, not all combinations of all variables have been selected

Instead combinations of all pairs of input variables have been covered with four test cases

ORTHOGONAL ARRAY

Orthogonal arrays provide a technique for selecting a subset of test cases with the following properties:

- It guarantees testing the pairwise combinations of all the selected variables
- It generates fewer test cases than a all-combination approach.
- It generates a test suite that has even distribution of all pairwise combinations
- It can be automated

ORTHOGONAL ARRAY

In the following, the steps of a technique to generate orthogonal arrays are presented. The steps are further explained by means of a detailed example.

Step 1: Identify the maximum number of independent input variables with which a system will be tested. This will map to the *factors* of the array—each input variable maps to a different factor.

Step 2: Identify the maximum number of values that each independent variable will take. This will map to the levels of the array.

Step 3: Find a suitable orthogonal array with the smallest number of runs $L_{\text{Runs}}(X^Y)$, where X is the number of levels and Y is the number of factors. A suitable array is one that has at least as many factors as needed from step 1 and has at least as many levels for each of those factors as identified in step 2.

Step 4: Map the variables to the factors and values of each variable to the levels on the array.

Step 5: Check for any "left-over" levels in the array that have not been mapped. Choose arbitrary valid values for those left-over levels.

Step 6: Transcribe the runs into test cases.

ORTHOGONAL ARRAY

Web Example. Consider a website that is viewed on a number of browsers with various plug-ins and operating systems (OSs) and through different connections as shown in Table 9.6. The table shows the variables and their values that are used as elements of the orthogonal array. We need to test the system with different combinations of the input values.

TABLE 9.6 Various Values That Need to Be Tested in Combinations

Variables	Values
Browser	Netscape, Internet Explorer (IE), Mozilla
Plug-in	Realplayer, Mediaplayer
OS	Windows, Linux, Macintosh
Connection	LAN, PPP, ISDN

Note: LAN, local-area network; PPP, Point-to-Point Protocal; ISDN, Integrated Services Digital Network.

ORTHOGONAL ARRAY

Following the steps laid out previously, let us design an orthogonal array to create a set of test cases for pairwise testing:

Step 1: There are four independent variables, namely, Browser, Plug-in, OS, and Connection.

Step 2: Each variable can take at most three values.

Step 3: An orthogonal array $L_9(3^4)$ as shown in Table 9.7 is good enough for the purpose. The array has nine rows, three levels for the values, and four factors for the variables.

Step 4: Map the variables to the factors and values to the levels of the array: the factor 1 to Browser, the factor 2 to Plug-in, the factor 3 to OS, and the factor 4 to Connection. Let 1 = Netscape, 2 = IE, and 3 = Mozilla in the Browser column. In the Plug-in column, let 1 = Realplayer and 3 = Mediaplayer. Let 1 = Windows, 2 = Linux, and 3 = Macintosh in the OS column. Let 1 = LAN, 2 = PPP, and 3 = ISDN in the Connection column. The mapping of the variables and the values onto the orthogonal array is given in Table 9.8.

ORTHOGONAL ARRAY

TABLE 9.7 $L_9(3^4)$ Orthogonal Array

	Factors			
Runs	1	2	3	4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

ORTHOGONAL ARRAY

Step 5: There are left-over levels in the array that are not being mapped. The factor 2 has three levels specified in the original array, but there are only two possible values for this variable. This has caused a level (2) to be left over for variable Plug-in after mapping the factors. One must provide a value in the cell. The choice of this value can be arbitrary, but to have a coverage, start at the top of the Plug-in column and cycle through the possible values when filling in the left-over levels. Table 9.9 shows the mapping after filling in the remaining levels using the cycling technique mentioned.

Step 6: We generate nine test cases taking the test case values from each run. Now let us examine the result:

- Each Browser is tested with every Plug-in, with every OS, and with every Connection.
- Each Plug-in is tested with every Browser, with every OS, and with every Connection.
- Each OS is tested with every Browser, with every Plug-in, and with every Connection.
- Each Connection is tested with every Browser, with every Plug-in, and with every OS.

ORTHOGONAL ARRAY

TABLE 9.8 $L_9(3^4)$ Orthogonal Array after Mapping Factors

Test Case ID	Browser	Plug-in	OS	Connection
TC_1	Netscape	Realplayer	Windows	LAN
TC_2	Netscape	2	Linux	PPP
TC_3	Netscape	Mediaplayer	Macintosh	ISDN
TC_4	IE	Realplayer	Linux	ISDN
TC_5	IE	2	Macintosh	LAN
TC_6	IE	Mediaplayer	Windows	PPP
TC_7	Mozilla	Realplayer	Macintosh	PPP
TC_8	Mozilla	2	Windows	ISDN
TC_9	Mozilla	Mediaplayer	Linux	LAN

ORTHOGONAL ARRAY

TABLE 9.9 Generated Test Cases after Mapping Left-Over Levels

Test Case ID	Browser	Plug-in	OS	Connection
TC_1	Netscape	Realplayer	Windows	LAN
TC_2	Netscape	Realplayer	Linux	PPP
TC_3	Netscape	Mediaplayer	Macintosh	ISDN
TC_4	IE	Realplayer	Linux	ISDN
TC_5	IE	Mediaplayer	Macintosh	LAN
TC_6	IE	Mediaplayer	Windows	PPP
TC_7	Mozilla	Realplayer	Macintosh	PPP
TC_8	Mozilla	Realplayer	Windows	ISDN
TC_9	Mozilla	Mediaplayer	Linux	LAN

IN PARAMETER ORDER

Tai and Lei have given an algorithm called In Parameter Order (IPO) to generate a test suite for pairwise coverage of input variables

The algorithm runs in three phases, namely, *initialization*, *horizontal growth*, and *vertical growth*, in that order

In the *initialization phase*, test cases are generated to cover two input variables

In the *horizontal growth* phase, the existing test cases are extended with the values of the other input variables.

In the *vertical growth phase*, additional test cases are created such that the test suite satisfies pairwise coverage for the values of the new variables.

IN PARAMETER ORDER

Algorithm: In Parameter Order.

Input: Parameter p_i and its domain $D(p_i) = \{v_1, v_2, ..., v_q\}$, where $i = 1$ to n.

Output: A test suite T satisfying pairwise coverage.

Initialization

Step 1: For the first two parameters p_1 and p_2, generate test suite:
$T := \{(v_1, v_2) \mid v_1$ and v_2 are values of p_1 and p_2, respectively$\}$

Step 2: If $i = 2$ Stop. Otherwise, for $i = 3, 4, ..., n$ repeat **Step 3** and **Step 4**.

IN PARAMETER ORDER

Horizontal Growth Phase

Step 3: Let $D(p_i) = \{v_1, v_2, ..., v_q\}$;

Create a set $\pi_i :=$ { pairs between values of p_i and all values of $p_1, p_2, ..., p_{i-1}$};

If $|T| \leq q$, then

{ for $1 \leq j \leq |T|$, extend the jth test in T by adding values v_j and remove from π_i pairs covered by the extended test }

else

{ for $1 \leq j \leq q$, extend the jth test in T by adding value v_j and remove from π_i pairs covered by the extended test;

for $q < j \leq |T|$, extend the jth test in T by adding one value of p_i such that the resulting test covers the most numbers of pairs in π_i, and remove from π_i pairs covered by the extended test };

IN PARAMETER ORDER

Vertical Growth Phase

Step 4: Let $T' := \Phi$ (empty set) and $|\pi_i| > 0$;
for each pair in π_i (let the pairs contains value w of p_k, $1 \leq k < i$, and values u of p_i)
{
if (T' contains a test with "–" as the value of p_k and u as the value of p_i)
modify this test by replacing the "–" with w;
else
add a new test to T' that has w as the value of p_k, u as the value of p_i, and "–" as the value of every other parameter;
};
$T := T \cup T'$;

IN PARAMETER ORDER

Example:

- Consider the system *S* which has three input parameters *X*, *Y*, and *Z*. Assume that a set *D*, a set of input test data values, has been selected for each input variable such that *D(X)* = {True, False}, *D(Y)* = {0, 5}, and *D(Z)* = {*P,Q,R*}. The total number of possible test cases is 2 × 2 × 3 = 12, but the IPO algorithm generates six test cases. Let us apply step 1 of the algorithm.

IN PARAMETER ORDER

Initialization:

- **Step 1:** Generate a test suite consisting of four test cases with pairwise coverage for the first two parameters *X* and *Y*:

$$T = \begin{bmatrix} (\text{True}, & 0) \\ (\text{True}, & 5) \\ (\text{False}, & 0) \\ (\text{False}, & 5) \end{bmatrix}$$

- **Step 2:** $i = 3 > 2$; therefore, steps 2 and 3 must be executed.

IN PARAMETER ORDER

Horizontal Growth:

- **Step 1:** Generate a test suite consisting of four test cases with pairwise coverage for the first two parameters *X* and *Y*:

$$T = \begin{bmatrix} (\text{True}, & 0) \\ (\text{True}, & 5) \\ (\text{False}, & 0) \\ (\text{False}, & 5) \end{bmatrix}$$

- **Step 2:** $i = 3 > 2$; therefore, steps 2 and 3 must be executed.

EQUIVALENCE CLASS PARTITIONING

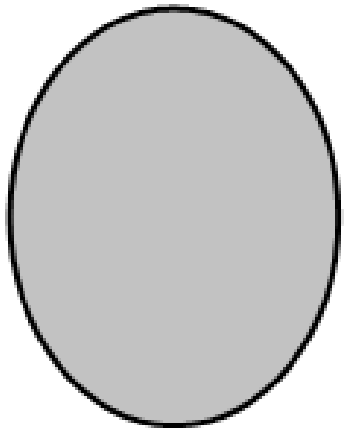
An input domain may be too large for all its elements to be used as test input Figure (a)

The input domain is partitioned into a finite number of subdomains

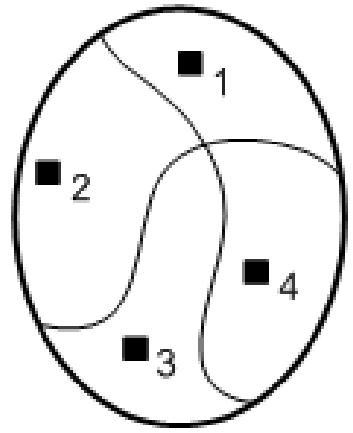
Each subdomain is known as an equivalence class, and it serves as a source of at least one test input Figure (b).

A valid input to a system is an element of the input domain that is expected to return a non error value

An invalid input is an input that is expected to return an error value.



(a) Input Domain



(b) Input Domain Partitioned into Four Sub-domains

(a) Too many test input;

(b) One input is selected from each of the subdomain

GUIDELINES FOR EQUIVALENCE CLASS PARTITIONING

An input condition specifies a range [a, b]

- one equivalence class for a < X < b, and
- two other classes for X < a and X > b to test the system with invalid inputs

An input condition specifies a set of values

- one equivalence class for each element of the set $\{M_1\}$, $\{M_2\}$, ..., $\{M_N\}$, and
- one equivalence class for elements outside the set $\{M_1, M_2, ..., M_N\}$

Input condition specifies for each individual value

- If the system handles each valid input differently then create one equivalence class for each valid input

An input condition specifies the number of valid values (Say N)

- Create one equivalence class for the correct number of inputs
- two equivalence classes for invalid inputs – one for zero values and one for more than N values

An input condition specifies a “must be” value

- Create one equivalence class for a “must be” value, and
- one equivalence class for something that is not a “must be” value

IDENTIFICATION OF TEST CASES

Test cases for each equivalence class can be identified by:

- Assign a unique number to each equivalence class
- For each equivalence class with valid input that has not been covered by test cases yet, write a new test case covering as many uncovered equivalence classes as possible
- For each equivalence class with invalid input that has not been covered by test cases, write a new test case that covers one and only one of the uncovered equivalence classes

ADVANTAGES OF EQUIVALENCE CLASS PARTITIONING

- A small number of test cases are needed to adequately cover a large input domain
- One gets a better idea about the input domain being covered with the selected test cases
- The probability of uncovering defects with the selected test cases based on equivalence class partitioning is higher than that with a randomly chosen test suite of the same size
- The equivalence class partitioning approach is not restricted to input conditions alone – the technique may also be used for output domains

BOUNDARY VALUE ANALYSIS (BVA)

- The central idea in Boundary Value Analysis (BVA) is to select test data near the boundary of a data domain so that data both within and outside an equivalence class are selected
- The BVA technique is an extension and refinement of the equivalence class partitioning technique
- In the BVA technique, the boundary conditions for each of the equivalence class are analyzed in order generate test cases

GUIDELINES FOR BOUNDARY VALUE ANALYSIS

The equivalence class specifies a range

- If an equivalence class specifies a range of values, then construct test cases by considering the boundary points of the range and points just beyond the boundaries of the range

The equivalence class specifies a number of values

- If an equivalence class specifies a number of values, then construct test cases for the minimum and the maximum value of the number
- In addition, select a value smaller than the minimum and a value larger than the maximum value.

The equivalence class specifies an ordered set

- If the equivalence class specifies an ordered set, such as a linear list, table, or a sequential file, then focus attention on the first and last elements of the set.

DECISION TABLES

The structure of a decision table has been shown in Table 9.13

It comprises a set of conditions (or, causes) and a set of effects (or, results) arranged in the form of a column on the left of the table

In the second column, next to each condition, we have its possible values: Yes (Y), No (N), and Don't Care ("-")

To the right of the "Values" column, we have a set of rules. For each combination of the three conditions {C1,C2,C3}, there exists a rule from the set {R1,R2, ..,R8}

Each rule comprises a Yes (Y), No (N), or Don't Care ("-") response, and contains an associated list of effects {E1,E2,E3}

For each relevant effect, an effect sequence number specifies the order in which the effect should be carried out, if the associated set of conditions are satisfied

The "Checksum" is used for verification of the combinations, the decision table represent

Each rule of a decision table represents a test case

DECISION TABLES

Conditions	Values	Rules or Combinations							
		R1	**R2**	**R3**	**R4**	**R5**	**R6**	**R7**	**R8**
C1	Y, N, -	Y	Y	Y	Y	N	N	N	N
C2	Y, N, -	Y	Y	N	N	Y	Y	N	N
C3	Y, N, -	Y	N	Y	N	Y	N	Y	N
Effects									
E1		1		2	1				
E2			2	1			2	1	
E3		2	1	3		1	1		
Checksum	8	1	1	1	1	1	1	1	1

Table 9.13: A decision table comprising a set of conditions and effects.

DECISION TABLES

The steps in developing test cases using decision table technique:

Step 1: The test designer needs to identify the conditions and the effects for each specification unit.

- A condition is a distinct input condition or an equivalence class of input conditions
- An effect is an output condition. Determine the logical relationship between the conditions and the effects

Step 2: List all the conditions and effects in the form of a decision table. Write down the values the condition can take

Step 3: Calculate the number of possible combinations. It is equal to the number of different values raised to the power of the number of conditions

DECISION TABLES

Step 4: Fill the columns with all possible combinations – each column corresponds to one combination of values. For each row (condition) do the following:

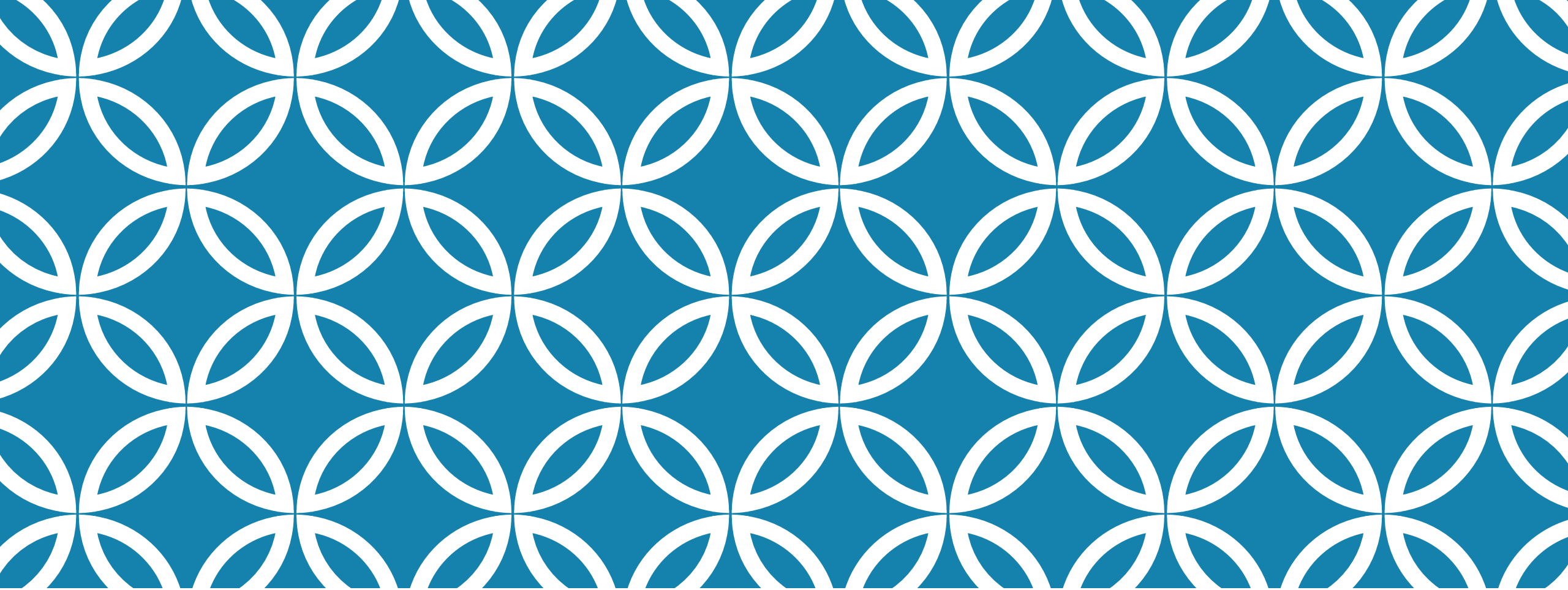
- Determine the Repeating Factor (RF): divide the remaining number of combinations by the number of possible values for that condition
- Write RF times the first value, then RF times the next and so forth, until row is full

Step 5: Reduce combinations (rules). Find indifferent combinations - place a "-" and join column where columns are identical. While doing this, ensure that effects are the same

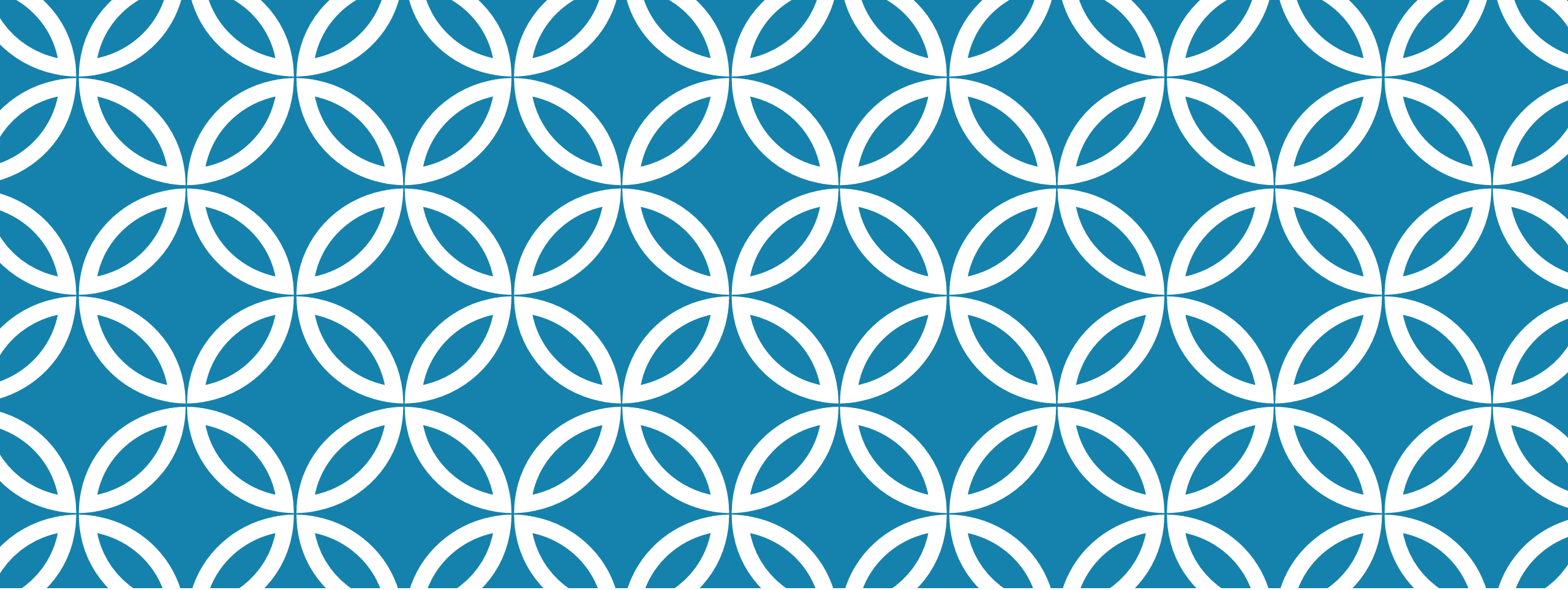
Step 6: Check covered combinations (rules). For each column calculate the combinations it represents. A "-" represents as many combinations as the condition has. Multiply for each "-" down the column. Add up total and compare with step 3. It should be the same

Step 7: Add effects to the column of the decision table. Read column by column and determine the effects. If more than one effect can occur in a single combinations, then assign a sequence number to the effects, thereby specifying the order in which the effects should be performed. Check the consistency of the decision table

Step 8: The columns in the decision table are transformed into test cases



THANK YOU!!



CHAPTER – 14
ACCEPTANCE TESTING

OUTLINE OF THE CHAPTER

- Types of Acceptance Testing
- Acceptance Criteria
- Selection of Acceptance Criteria
- Acceptance Test Plan
- Acceptance Test Execution
- Acceptance Test Report
- Acceptance Testing in eXtreme Programming

TYPES OF ACCEPTANCE TESTING

Acceptance testing is a formal testing conducted to determine whether a system satisfies its acceptance criteria

There are two categories of acceptance testing:

- User Acceptance Testing (UAT)
  - It is conducted by the customer to ensure that system satisfies the contractual acceptance criteria before being signed-off as meeting user needs.
  - This involves verifying if the user's specific requirements have been met.
- Business Acceptance Testing (BAT)
  - It is undertaken within the development organization of the supplier to ensure that the system will eventually pass the user acceptance testing.
  - Here you are assessing whether the product meets the business goals set out in the design.

TYPES OF ACCEPTANCE TESTING

Three major objectives of acceptance testing:

Confirm that the system meets the agreed upon criteria

Identify and resolve discrepancies, if there is any

Determine the readiness of the system for cut-over to live operations

ACCEPTANCE CRITERIA

The acceptance criteria are defined on the basis of the following attributes:

- Functional Correctness and Completeness
- Accuracy
- Data Integrity
- Data Conversion
- Backup and Recovery
- Competitive Edge
- Usability
- Performance
- Start-up Time
- Stress
- Reliability and Availability
- Maintainability and Serviceability
- Robustness
- Timeliness
- Confidentiality and Availability
- Compliance
- Installability and Upgradability
- Scalability
- Documentation

SELECTION OF ACCEPTANCE CRITERIA

The acceptance criteria discussed are too many and very general

The customer needs to select a subset of the quality attributes

The quality attributes are prioritize them to specific situation

IBM used the quality attribute list CUPRIMDS for their products

- Capability, Usability, Performance, Reliability, Installation, Maintenance, Documentation, and Service

Ultimately, the acceptance criteria must be related to the business goals of the customer's organization

ACCEPTANCE TEST PLAN

1. Introduction
2. Acceptance test category For each category of acceptance criteria (a) Operation environment (b) Test case specification (i) Test case Id# (ii) Test title (iii)Test objective (iv) Test procedure
3. Schedule
4. Human resources

Table 14.1: An outline of an acceptance test plan.

ACCEPTANCE TEST EXECUTION

The acceptance test cases are divided into two subgroups

- The first subgroup consists of basic test cases, and
- The second consists of test cases that are more complex to execute

The acceptance tests are executed in two phases

- In the first phase, the test cases from the basic test group are executed
- If the test results are satisfactory then the second phase, in which the complex test cases are executed, is taken up.
- In addition to the basic test cases, a subset of the system-level test cases are executed by the acceptance test engineers to independently confirm the test results

Acceptance test execution activity includes the following detailed actions:

- The developers train the customer on the usage of the system
- The developers and the customer co-ordinate the fixing of any problem discovered during acceptance testing
- The developers and the customer resolve the issues arising out of any acceptance criteria discrepancy

ACCEPTANCE TEST EXECUTION

The acceptance test engineer may create an Acceptance Criteria Change (ACC) document to communicate the deficiency in the acceptance criteria to the supplier

A representative format of an ACC document is shown in Table 14.2.

An ACC report is generally given to the supplier's marketing department through the on-site system test engineers

1. ACC Number:	A unique number
2. Acceptance Criteria Affected:	The existing acceptance criteria
3. Problem/Issue Description:	Brief description of the issue
4. Description of Change Required:	Description of the changes needed to be done to the original acceptance criterion
5. Secondary Technical Impacts:	Description of the impact it will have on the system
6. Customer Impacts:	What impact it will have on the end user
7. Change Recommended by:	Name of the acceptance test engineer(s)
8. Change Approved by:	Name of the approver(s) from both the parties

Table 14.2: Acceptance criteria change document information.

ACCEPTANCE TEST REPORT

The acceptance test activities are designed to reach at a conclusion:

- accept the system as delivered
- accept the system after the requested modifications have been made
- do not accept the system

Usually some useful intermediate decisions are made before making the final decision.

- A decision is made about the continuation of acceptance testing if the results of the first phase of acceptance testing is not promising
- If the test results are unsatisfactory, changes be made to the system before acceptance testing can proceed to the next phase

During the execution of acceptance tests, the acceptance team prepares a test report on a daily basis

A template of the test report is given in Table 14.3

At the end of the first and the second phases of acceptance testing an acceptance test report is generated which is outlined in Table 14.4

ACCEPTANCE TEST REPORT

1. Date:	Acceptance report date
2. Test case execution status:	Number of test cases executed today Number of test cases passing Number of test cases failing
3. Defect identifier:	Submitted defect number Brief description of the issue
4. ACC number(s):	Acceptance criteria change document number(s), if any
5. Cumulative test execution status:	Total number of test cases executed Total number of test cases passing Total number of test cases failing Total number of test cases not executed yet

Table 14.3: Structure of the acceptance test status report.

ACCEPTANCE TEST REPORT

1. Report identifier
2. Summary
3. Variances
4. Summary of results
5. Evaluation
6. Recommendations
7. Summary of activities
8. Approval

Table 14.4: Structure of the acceptance test summary report.

ACCEPTANCE TESTING IN EXTREME PROGRAMMING

In XP framework, the user stories are used as acceptance criteria

The user stories are written by the customer as things that the system needs to do for them

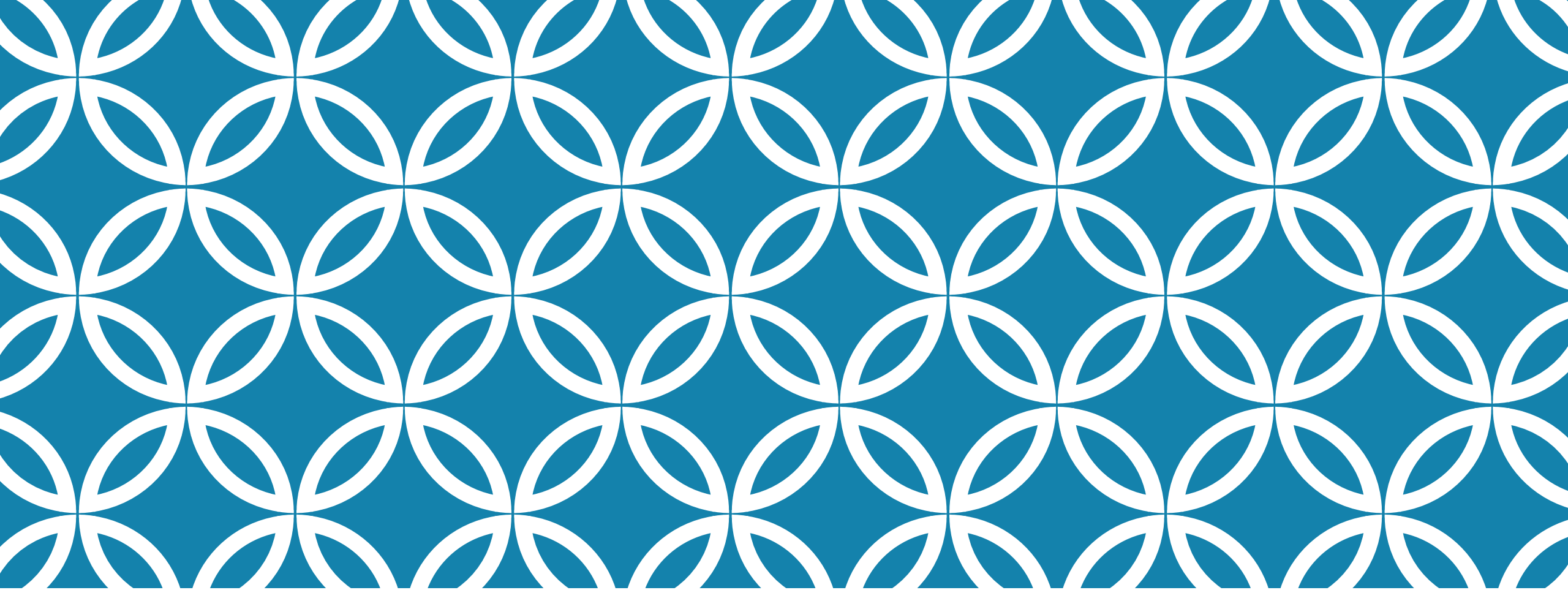
Several acceptance tests are created to verify the user story has been correctly implemented

The customer is responsible for verifying the correctness of the acceptance tests and reviewing the test results

A story is incomplete until it passes its associated acceptance tests

Ideally, acceptance tests should be automated, either using the unit testing framework, before coding

The acceptance tests take on the role of regression tests



THANK YOU!!