Software Testing

Testing

- Software engineer attempts to build software from an abstract concept to a tangible product. Next is Testing.
- The engineer creates a series of test cases that are intended to "demolish" the software that has been built.
- In fact, testing is the one step in the software process that could be viewed as destructive rather than constructive.

Testing Objective

- Primary Objective
    - Testing is a process of executing a program with the intent of finding an error.
    - A good test case is one that has a high probability of finding an as-yet undiscovered error.
    - A successful test is one that uncovers an as-yet-undiscovered error
- Objective is to design tests that systematically uncover different classes of errors and to do so with a minimum amount of time and effort.
- Testing demonstrates that software functions appear to be working according to specification, that behavioral and performance requirements appear to have been met.
- Data collected as testing is conducted provide a good indication of software reliability and some indication of software quality as a whole.
- But testing cannot show the absence of errors and defects, it can show only that software errors and defects are present.

Testing Principles

- **All tests should be traceable to customer requirements.**
- **Tests should be planned long before testing begins.**
- **The Pareto principle applies to software testing.**
- **Testing should begin “in the small” and progress toward testing “in the large.”**
- **Exhaustive testing is not possible.**
- **To be most effective, testing should be conducted by an independent third party.**

TEST CASE DESIGN

- Objectives of testing, we must design tests that have the highest likelihood of finding the most errors with a minimum amount of time and effort.

Any system can be tested on two ways:

1. Knowing the specified function that a product has been designed to perform; tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function (Black Box)
2. knowing the internal workings of a product, tests can be conducted to ensure that "all gears mesh," that is, internal operations are performed according to specifications and all internal components have been effectively exercised. (White box testing)

White box testing

- *White-box testing* of software is predicated on close examination of procedural detail.
- Logical paths through the software are tested by providing test cases that exercise specific sets of conditions and/or loops.
- The "status of the program" may be examined at various points.
- White-box testing, sometimes called *glass-box testing,* is a test case design method that uses the control structure of the procedural design to derive test cases.

White box testing

Using this method, SE 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.

Basis path testing

- *Basis path testing* is a white-box testing technique
- To derive a logical complexity measure of a procedural design.
- Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time.

Methods:

1. Flow graph notation
2. Independent program paths or Cyclomatic complexity
3. Deriving test cases
4. Graph Matrices

Flow Graph Notation

- Start with simple notation for the representation of *control flow (called flow graph).* It represent logical control flow.

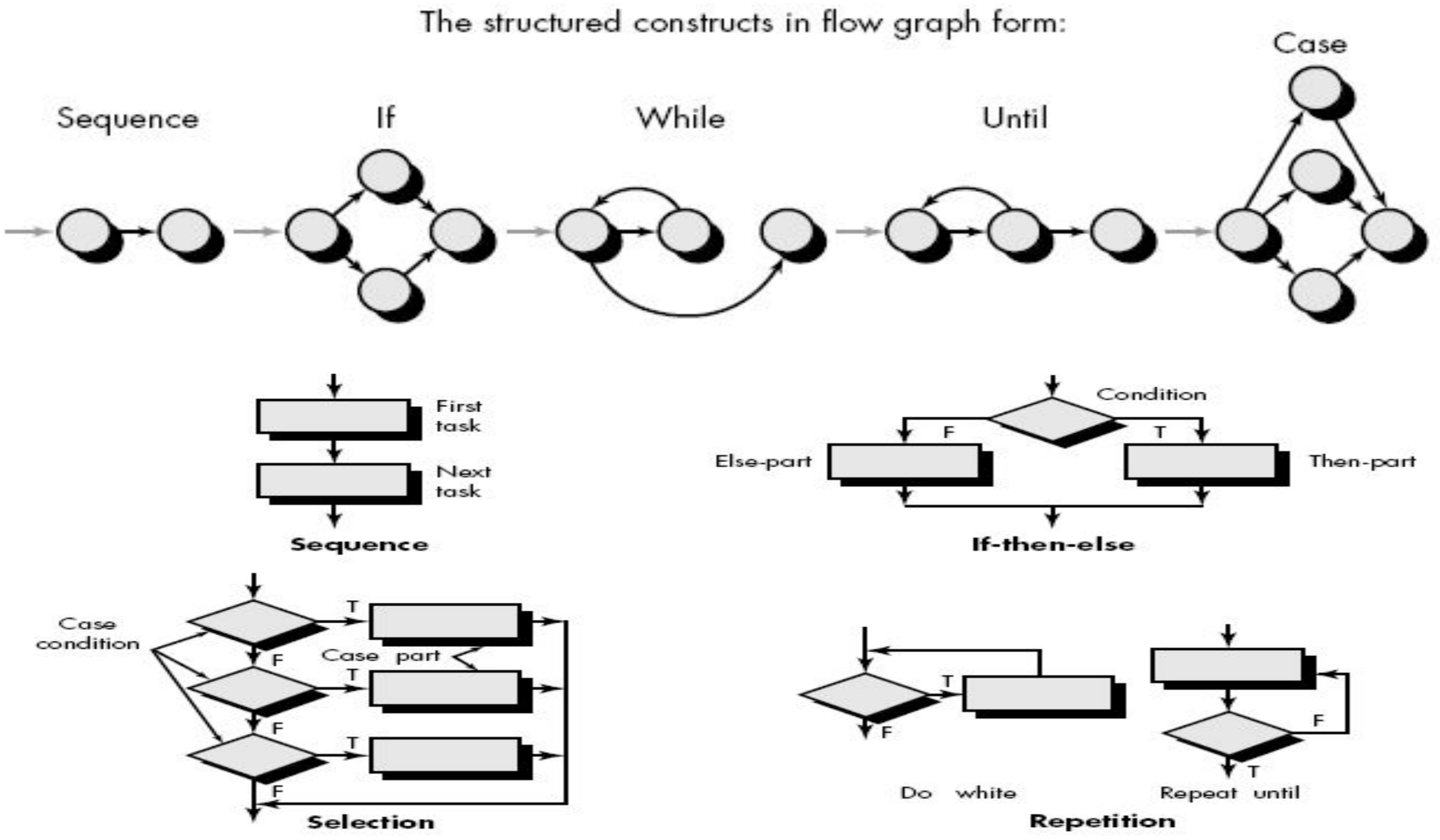
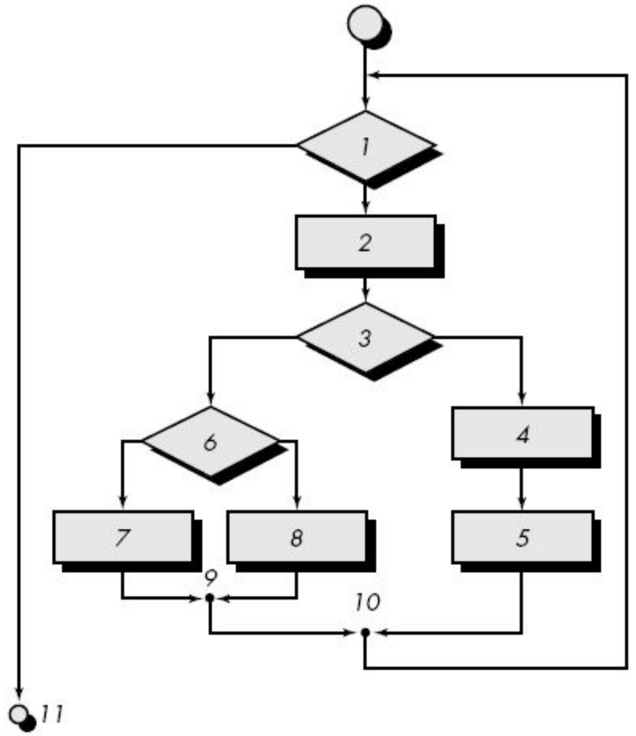
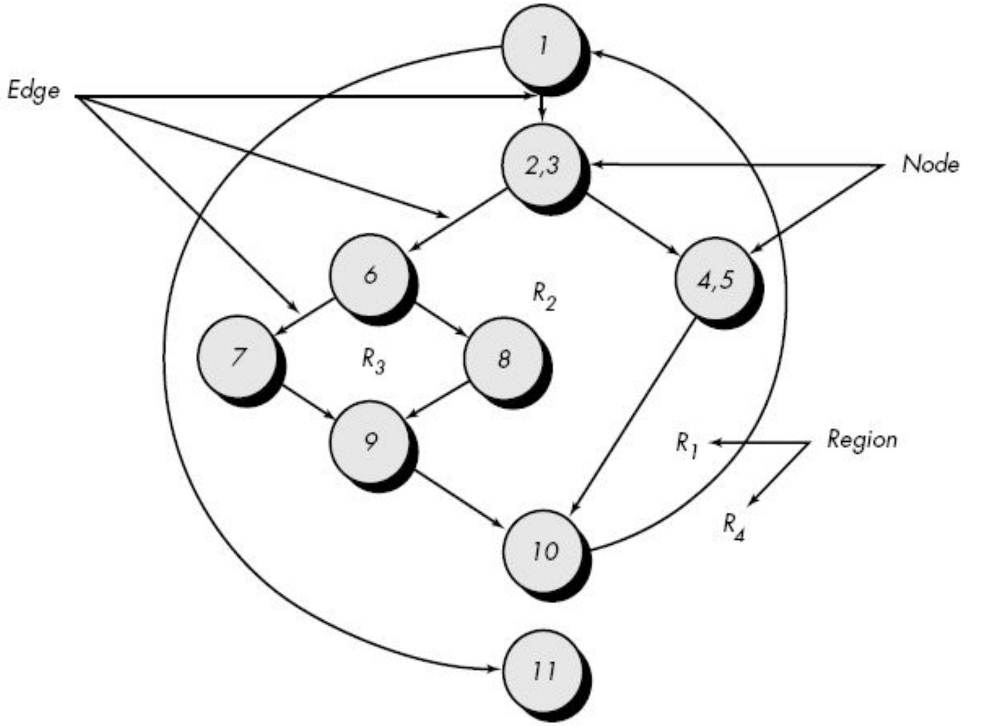


Fig. A represent program control structure and fig. B maps the flowchart into a corresponding flow graph.



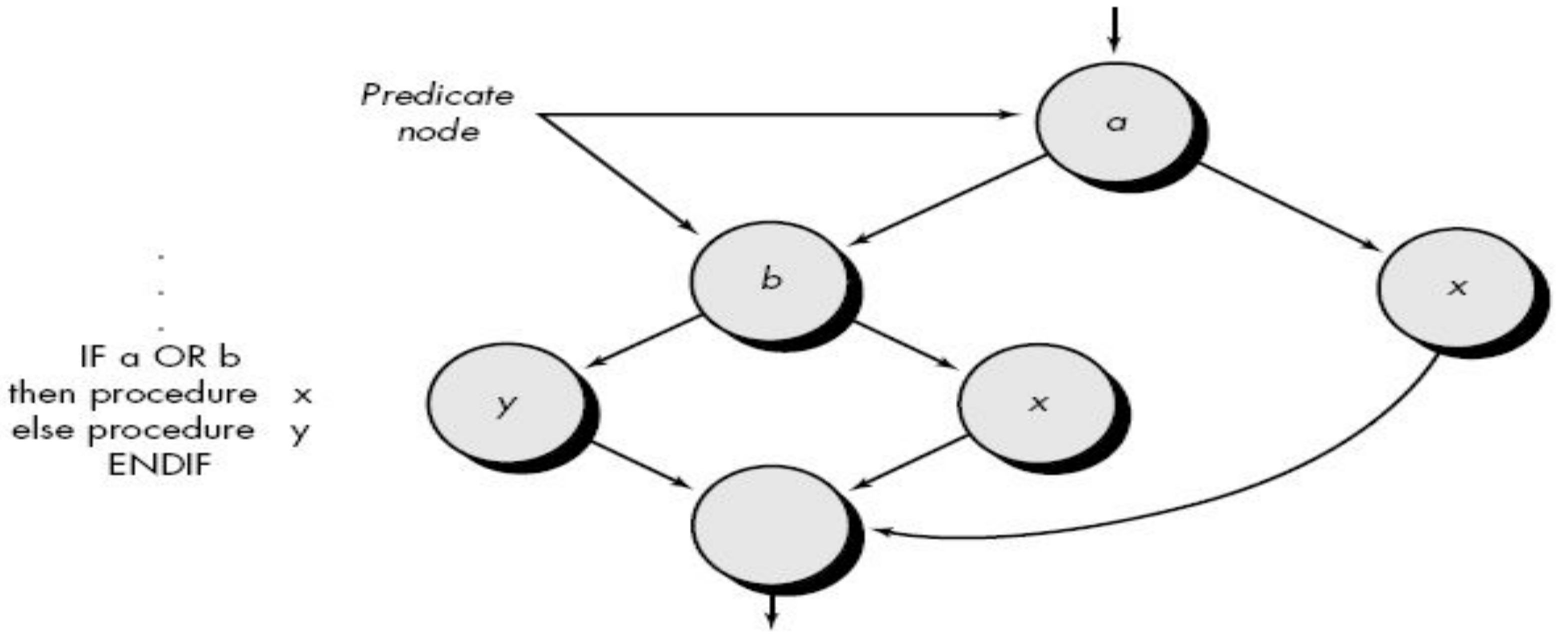
(A)



(B)

In fig. B each circle, called flow graph node, represent one or more procedural statement.

- A sequence of process boxes and decision diamond can map into a single node.
- The arrows on the flow graph, called edges or links, represent flow of control and are parallel to flowchart arrows.
- An edge must terminate at a node, even if the node does not represent any procedural statement.
- Areas bounded by edges and nodes are called <u>*regions*</u>. When counting regions, we include the are outside the graph as a region.
- When compound condition are encountered in procedural design, flow graph becomes slightly more complicated.



- When we translating PDL segment into flow graph, separate node is created for each condition.
- Each node that contains a condition is called *predicate node* and is characterized by two or more edges comes from it.

Independent program paths or Cyclomatic complexity

- An *independent path* is any path through the program that introduces at least one *new set of processing statement* or *new condition.*
- For example, a set of independent paths for flow graph:
  - Path 1: 1-11
  - Path 2: 1-2-3-4-5-10-1-11
  - Path 3: 1-2-3-6-8-9-1-11
  - Path 4: 1-2-3-6-7-9-1-11

Basis Set

- Note that each new path introduces a new edge.
- The path 1-2-3-4-5-10-1-2-3-6-8-9-1-11 is not considered to e an independent path because it is simply a combination of already specified paths and does not traverse any new edges.
- Test cases should be designed to force execution of these paths (basis set).
- Every statement in the program should be executed at least once and every condition will have been executed on its true and false.

- How do we know how many paths to looks for ?
- *Cyclomatic complexity* is a software metrics that provides a quantitative measure of the logical complexity of a program.
- It defines no. of independent paths in the basis set and also provides number of test that must be conducted.
- One of three ways to compute cyclomatic complexity:
  1. The *no. of regions* corresponds to the cyclomatic complexity.
  2. Cyclomatic complexity, V(G), for a flow graph, G, is defined as

     V(G) = E - N + 2

     where E is the number of flow graph edges, N is the number of flow graph nodes.
  3. Cyclomatic complexity, V(G), for a flow graph, G, is also defined as

     V(G) = P + 1

     where P is the number of predicate nodes edges

So the value of V(G) provides us with upper bound of test cases.

Deriving Test Cases

- It is a series of steps method.
- The *procedure average* depicted in PDL.
- *Average*, an extremely simple algorithm, contains compound conditions and loops.

To derive basis set, follow the steps.

1. **Using the design or code as a foundation, draw a corresponding flow graph.**
    - A flow graph is created by numbering those PDL statements that will be mapped into corresponding flow graph nodes.

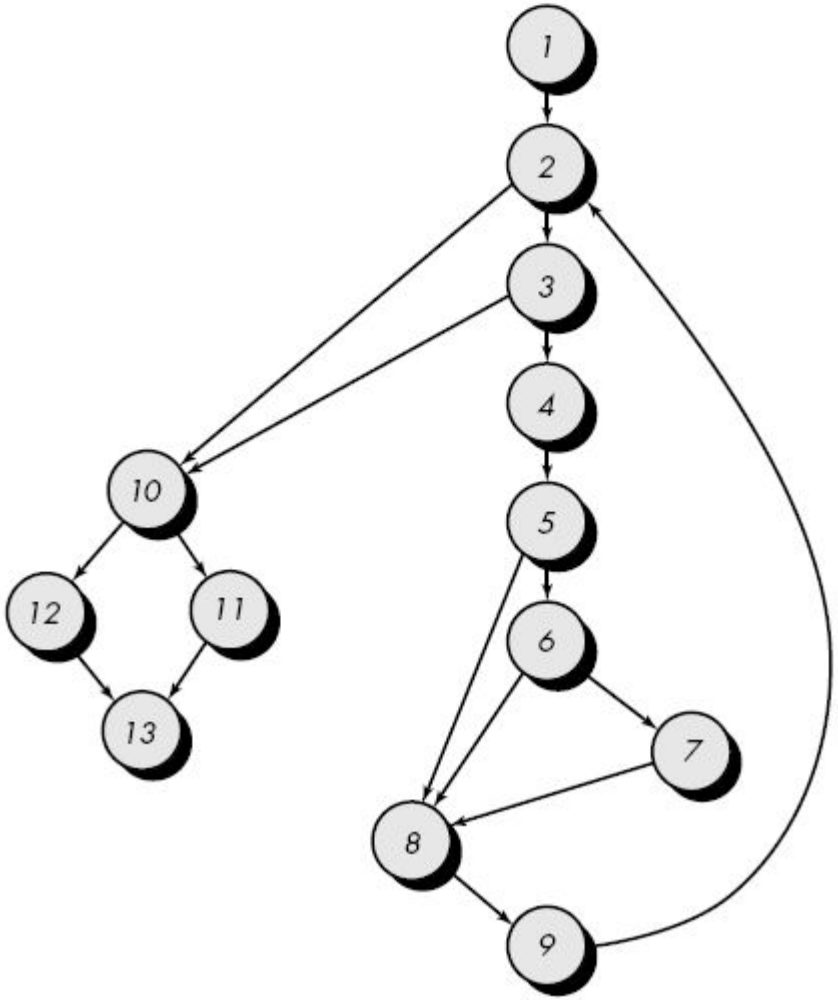
Deriving Test Cases

```
PROCEDURE average;

    *   This procedure computes the average of 100 or fewer
        numbers that lie between bounding values; it also computes the
        sum and the total number valid.

    INTERFACE RETURNS average, total.input, total.valid;
    INTERFACE ACCEPTS value, minimum, maximum;

    TYPE value[1:100] IS SCALAR ARRAY;
    TYPE average, total.input, total.valid;
        minimum, maximum, sum IS SCALAR;
    TYPE i IS INTEGER;
    i = 1;
    total.input = total.valid = 0;                2
1   sum = 0;
    DO WHILE value[i] <> -999 AND total.input < 100    3
  4     increment total.input by 1;
        IF value[i] > = minimum AND value[i] < = maximum    6
 5
      7     THEN increment total.valid by 1;
                  sum = s sum + value[i]
            ELSE skip
        ENDIF
 8      increment i by 1;
 9  ENDDO
    IF total.valid > 0    10
 11     THEN average = sum / total.valid;
12      ELSE average = -999;
13  ENDIF
END average
```



Flow graph for the procedure average

2. Determine the cyclomatic complexity of the resultant flow graph.

2. *V*(*G*) can be determined without developing a flow graph by counting all conditional statements in the PDL (for the procedure *average,* compound conditions count as two) and adding 1
3. *V*(*G*) = 6 regions
4. *V*(*G*) = 17 edges - 13 nodes + 2 = 6
5. *V*(*G*) = 5 predicate nodes + 1 = 6

3. Determine a basis set of linearly independent paths

2. The value of *V*(*G*) provides the number of linearly independent paths through the program control structure.
3. path 1: 1-2-10-11-13
4. path 2: 1-2-10-12-13
5. path 3: 1-2-3-10-11-13
6. path 4: 1-2-3-4-5-8-9-2-. . .
7. path 5: 1-2-3-4-5-6-8-9-2-. . .
8. path 6: 1-2-3-4-5-6-7-8-9-2-. . .
9. The ellipsis (. . .) following paths 4, 5, and 6 indicates that any path through the remainder of the control structure is acceptable.

4. **Prepare test cases that will force execution of each path in the basis set.**
   - Data should be chosen so that conditions at the predicate nodes are appropriately set as each path is tested.
   - Each test case is executed and compared to expected results.
   - Once all test cases have been completed, the tester can be sure that all statements in the program have been executed at least once.

Black box testing

- Also called *behavioral testing,* focuses on the functional requirements of the software.
- It enables the software engineer 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 but it is complementary approach.
- Black-box testing attempts to find errors in the following categories:
  - Incorrect or missing functions,
  - Interface errors,
  - Errors in data structures or external data base access.
  - Behavior or performance errors,
  - Initialization and termination errors.

- Black-box testing purposely ignored control structure, attention is focused on the information domain. Tests are designed to answer the following questions:
  - How is functional validity tested?
  - How is system behavior and performance tested?
  - What classes of input will make good test cases?
- By applying black-box techniques, we derive a set of test cases that satisfy the following criteria
  - Test cases that reduce the number of additional test cases that must be designed to achieve reasonable testing (i.e minimize effort and time)
  - Test cases that tell us something about the presence or absence of classes of errors
- Black box testing methods
  - Syntax Based Testing Methods
  - Equivalence partitioning
  - Boundary value analysis (BVA)
  - Cause effect graphTesting

Equivalence Partitioning

- *Equivalence partitioning* is a black-box testing method that divides the input domain of a program into classes of data from which test cases can be derived.
- Test case design for equivalence partitioning is based on an evaluation of *equivalence classes* for an input condition.
- An *equivalence class* represents a set of valid or invalid states for input conditions.
- Typically, an input condition is either a specific numeric value, a range of values, a set of related values, or a Boolean condition.

To define equivalence classes follow the guideline

1. If an input condition specifies a *range,* one valid and two invalid equivalence classes are defined.
2. If an input condition requires a specific *value,* one valid and two invalid equivalence classes are defined.
3. If an input condition specifies a member of a *set,* one valid and one invalid equivalence class are defined.
4. If an input condition is *Boolean,* one valid and one invalid class are defined.

Example

- area code—blank or three-digit number
- prefix—three-digit number not beginning with 0 or 1
- suffix—four-digit number
- password—six digit alphanumeric string
- commands— check, deposit, bill pay, and the like

- area code:
  - Input condition, *Boolean*—the area code may or may not be present.
  - Input condition, *value*— three digit number
- prefix:
  - Input condition, *range*—values defined between 200 and 999, with specific exceptions.
- Suffix:
  - Input condition, value—four-digit length
- password:
  - Input condition, *Boolean*—a password may or may not be present.
  - Input condition, *value*—six-character string.
- command:
  - Input condition, *set*— check, deposit, bill pay.

Boundary Value Analysis (BVA)

- Boundary value analysis is a test case design technique that complements equivalence partitioning.
- Rather than selecting any element of an equivalence class, BVA leads to the selection of test cases at the "edges" of the class.
- In other word, Rather than focusing solely on input conditions, BVA derives test cases from the output domain as well.

Guidelines for BVA

1. If an input condition specifies a range bounded by values *a* and *b,* test cases should be designed with values *a* and *b* and just above and just below *a* and *b*.
2. If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested.
3. Apply guidelines 1 and 2 to output conditions.
4. If internal program data structures have prescribed boundaries be certain to design a test case to exercise the data structure at its boundary

Orthogonal Array Testing

- The number of input parameters is small and the values that each of the parameters may take are clearly bounded.
- When these numbers are very small (e.g., three input parameters taking on three discrete values each), it is possible to consider every input permutation .
- However, as the number of input values grows and the number of discrete values for each data item increases (exhaustive testing occurs)
- *Orthogonal array testing* can be applied to problems in which the input domain is relatively small but too large to accommodate exhaustive testing.
- *Orthogonal Array Testing can be used to reduce the number of combinations and provide <u>maximum coverage with a minimum number of test cases.</u>*

Example

- Consider the *send* function for a fax application.
- Four parameters, P1, P2, P3, and P4, are passed to the *send* function. Each takes on three discrete values.
- P1 takes on values:
    - P1 = 1, send it now
    - P1 = 2, send it one hour later
    - P1 = 3, send it after midnight
- P2, P3, and P4 would also take on values of 1, 2 and 3, signifying other *send* functions.
- OAT is an array of values in which each column represents a Parameter - value that can take a certain set of values called levels.
- Each row represents a test case.
- Parameters are combined pair-wise rather than representing all possible combinations of parameters and levels

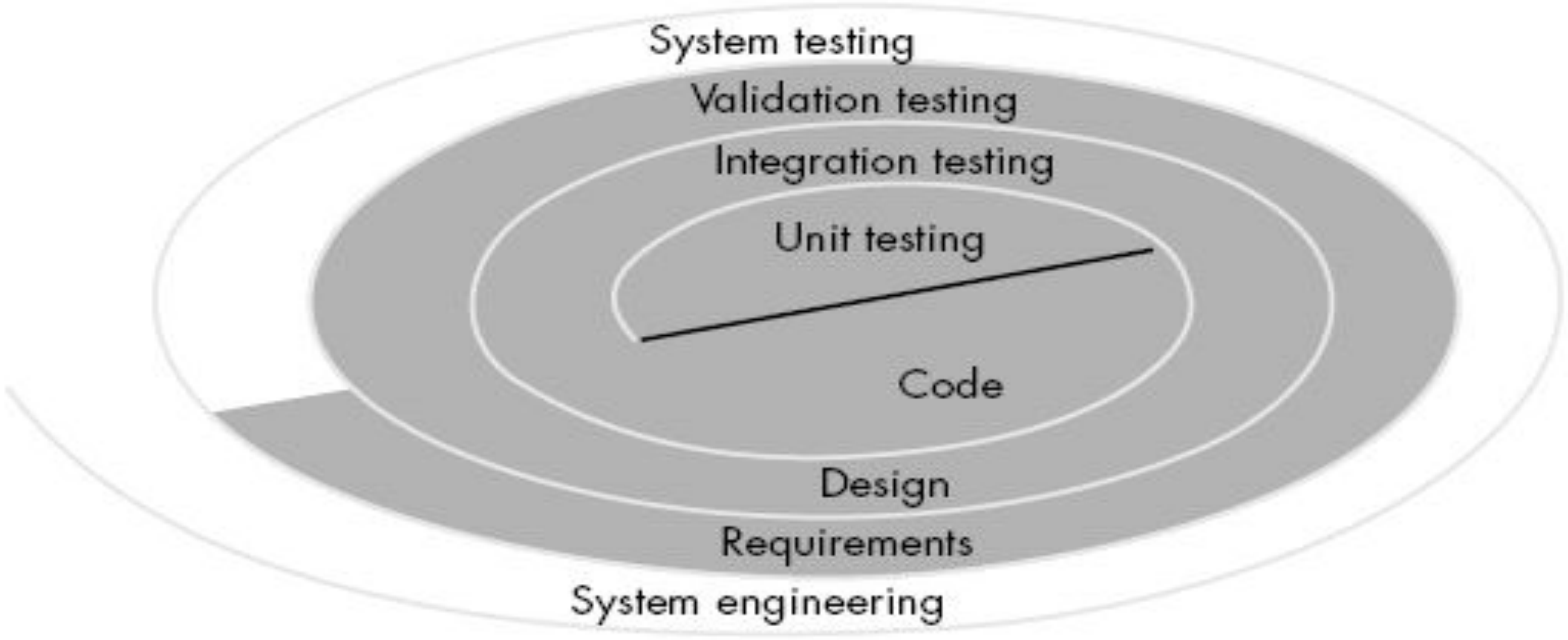
Test case	Test parameters			
	P_1	P_2	P_3	P_4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

Verification and Validation

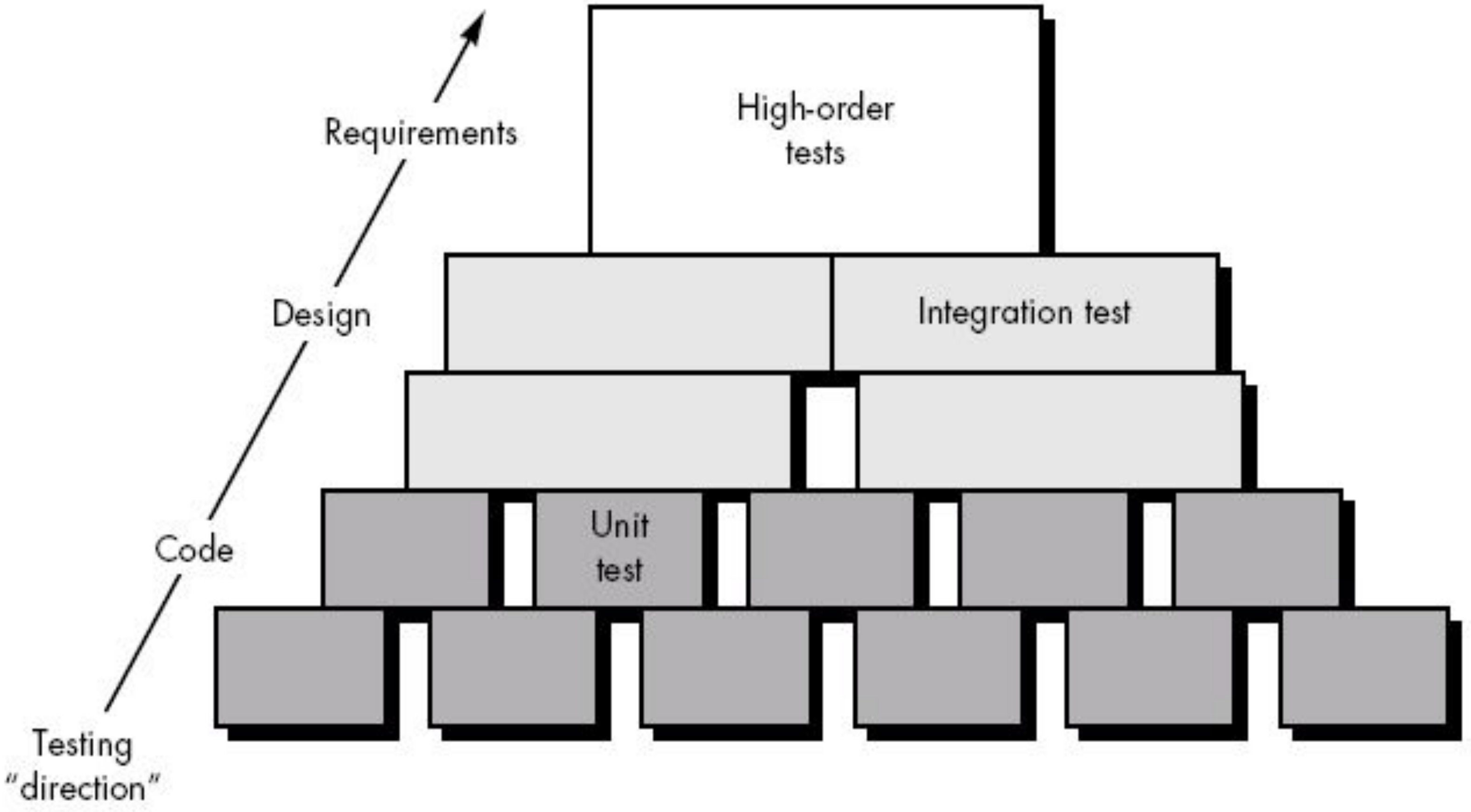
- Testing is one element of a broader topic that is often referred to as *verification and validation (V&V).*
- *Verification* refers to the set of activities that ensure that software correctly implements a specific function.
- *Validation* refers to a different set of activities that ensure that the software that has been built is traceable to customer requirements.
- State another way:
  - *Verification:* "Are we building the product right?"
  - *Validation:* "Are we building the right product?“
- The definition of V&V encompasses many of the activities that are similar to *software quality assurance* (SQA).

- V&V encompasses a wide array of SQA activities that include
    - Formal technical reviews,
    - quality and configuration audits,
    - performance monitoring,
    - simulation,
    - feasibility study,
    - documentation review,
    - database review,
    - algorithm analysis,
    - development testing,
    - qualification testing, and installation testing
- Testing does provide the last bastion from which quality can be assessed and, more pragmatically, errors can be uncovered.
- Quality is not measure only by no. of error but it is also measure on application methods, process model, tool, formal technical review, etc will lead to quality, that is confirmed during testing.

Software Testing Strategy for conventional software architecture



- A *Software process* & *strategy for software testing* may also be viewed in the context of the *spiral.*
- *Unit testing* begins at the vortex of the spiral and concentrates on each unit (i.e., component) of the software.
- Testing progresses by moving outward along the spiral to *integration testing,* where the focus is on design and the construction.
- Another turn outward on the spiral, we encounter *validation testing,* where requirements established as part of software requirements analysis are validated against the software.
- Finally, we arrive at *system testing,* where the software and other system elements are tested as a whole.



- Software process from a procedural point of view; a series of four steps that are implemented sequentially.

- Initially, tests focus on each component individually, ensuring that it functions properly as a unit.
- Unit testing makes heavy use of white-box testing techniques, exercising specific paths in a module's control structure.
- Integration testing addresses the issues associated with the dual problems of verification and program construction.
- Black-box test case design techniques are the most prevalent during integration.
- Now, validation testing provides final assurance that software meets all functional, behavioral, and performance requirements.
- Black-box testing techniques are used exclusively during validation.
- 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 function / performance is achieved.

Software Quality