Software Quality Management

Software Testing Techniques

1

Fundamentals of Testing

- Testability
  - Make the software testable (ease to test)
- Test characteristics
  - Make the tests able to discover errors

2

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.

Software testability

- Operability—it operates cleanly
  - Program has good initial quality.
- Observability—the results of each test case are readily observed
  - Success of a test can be judged by observable behaviors.
- Controllability—the degree to which testing can be automated and optimized
- Decomposability—testing can be targeted
- Simplicity—reduce complex architecture and logic to simplify tests
- Stability—few changes are requested during testing
- Understandability—of the design
  - A good understanding of the design helps improve testing effectiveness.

3

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.

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.

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.

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.

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).

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.

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

- A good test has a high probability of finding an error
- A good test is not redundant.
- A good test should be “best of breed”
  - Select test cases from a pool of candidate test cases based on its likelihood of finding bugs.
- A good test should be neither too simple nor too complex
  - When it is too complex, it may be hard to locate a bug because of interaction between different factors.

4

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).

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.

Testing: internal/external views

- External (black-box)
  - demonstrate each function is fully operational while at the same time searching for errors in each function
- Internal (white-box)
  - internal operations are performed according to specifications and all internal components have been adequately exercised

5

Any engineered product (and most other things) can be tested in one of two ways:
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;

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 adequately exercised.

Black-box testing alludes to tests that are conducted at the software interface. A black-box test examines some fundamental aspect of a system with little regard for the internal logical structure of the software. *White-box testing* of software is

predicated

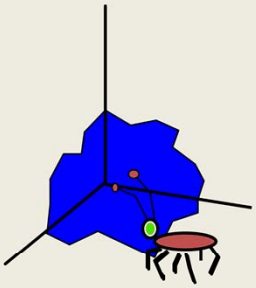
on close examination of procedural detail.

The terms *functional testing* and *structural testing* are sometimes used in place of black-box and

white-box testing, respectively.

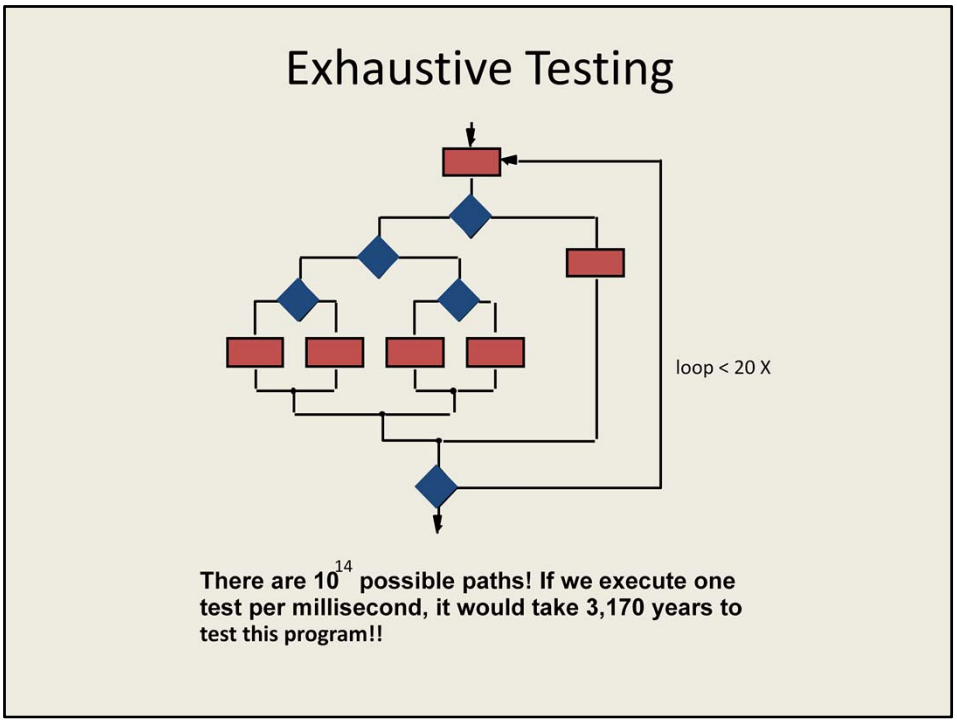
Test Case Design

"Bugs lurk in corners and congregate at boundaries ..."



OBJECTIVE	to uncover errors
CRITERIA	in a (relatively) complete manner
CONSTRAINT	with a minimum of effort and time

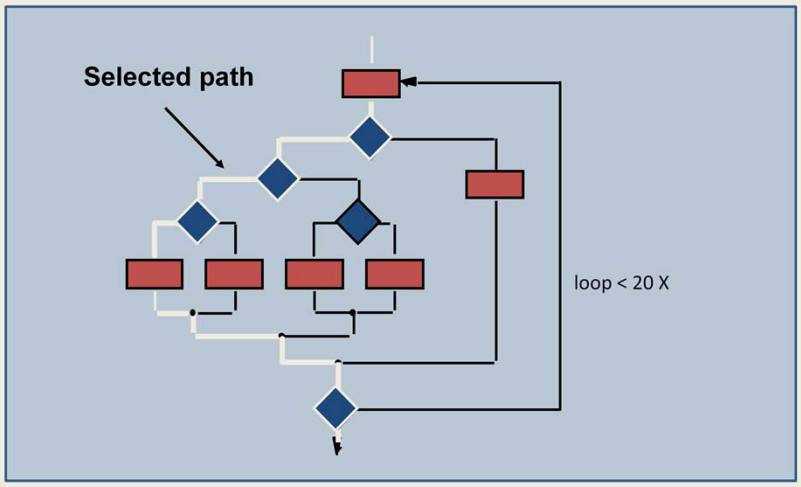
6



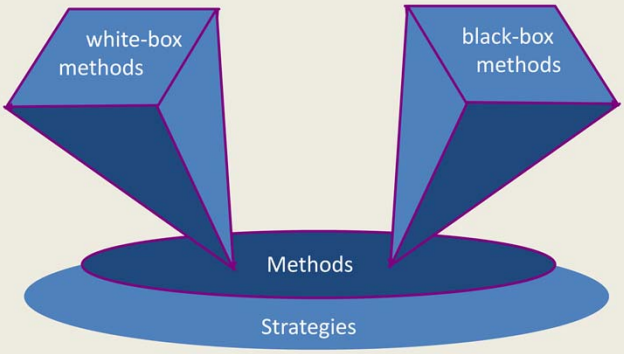
Unfortunately, exhaustive testing presents certain logistical problems. For even small programs, the number of possible logical

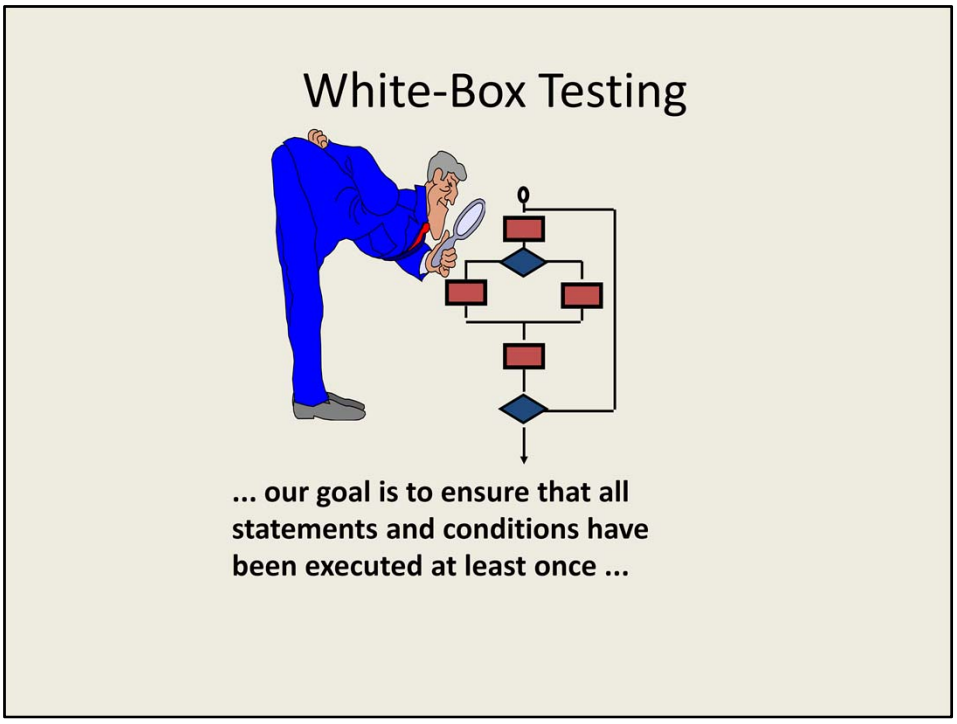
paths can be very large. White-box testing should not, however, be dismissed as impractical. A limited number of important logical paths can be selected and exercised. Important data structures can be probed for validity.

Selective Testing



Software 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

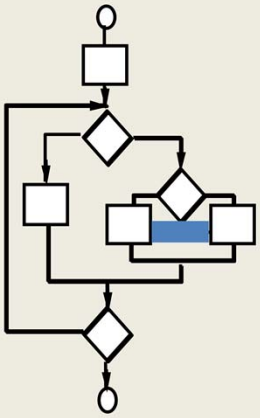
all loops at their boundaries and within their operational bounds, and (4) exercise

internal data structures to ensure their validity.

Why Cover?

- logic errors and incorrect assumptions are inversely proportional to a path's execution probability
- we often believe that a path is not likely to be executed; in fact, reality is often counter intuitive
- typographical errors are random; it's likely that untested paths will contain some

Basis Path Testing



First, we compute the cyclomatic complexity:

number of simple decisions + 1

or

number of enclosed areas + 1

In this case, V(G) = 4

Basis path testing is a white-box testing technique.

The basis path method enables the test-case designer to derive a logical

complexity measure of a procedural design and use this measure as a guide for defining

a basis set of execution paths. Test cases derived to exercise the basis set are guaranteed

to execute every statement in the program at least one time during testing.

Before we consider the basis path method, a simple notation for the representation

of control flow, called a ***flow graph*** (or *program graph*) must be introduced.3 The flow

graph depicts logical control flow using the notation illustrated in Figure 18.1

Areas bounded by edges and nodes are called

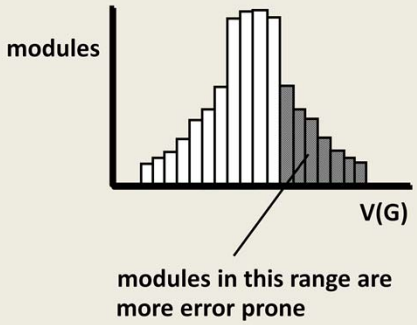
regions. When counting regions, we include the area outside the graph as a region

When stated in terms of a flow graph, an **independent path** must move along at least one edge that has not been

traversed before the path is defined; Note that each new path introduces a new edge

Cyclomatic Complexity

A number of industry studies have indicated that the higher V(G), the higher the probability or errors.



Cyclomatic Complexity	Risk Evaluation
1-10	a simple program, without much risk
11-20	more complex, moderate risk
21-50	complex, high risk program
greater than 50	untestable program (very high risk)

Cyclomatic complexity is a software metric that provides

a quantitative measure of the logical complexity of a program. When used in

the context of the basis path testing method, the value computed for cyclomatic complexity

defines the number of independent paths in the basis set of a program and

provides you with an upper bound for the number of tests that must be conducted to

ensure that all statements have been executed at least once.

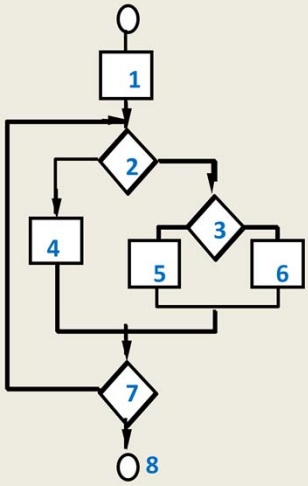
More important, the value for *V*(*G*) provides you with an upper bound for the number

of independent paths that form the basis set and, by implication, an upper bound

on the number of tests that must be designed and executed to guarantee coverage

of all program statements

Basis Path Testing



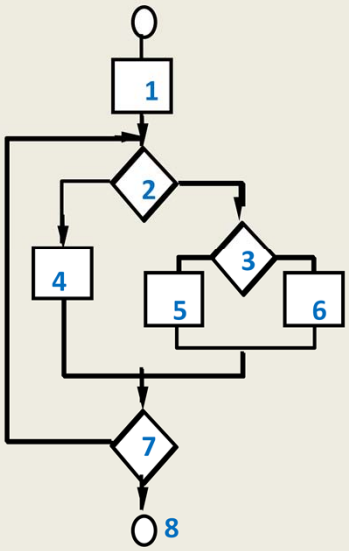
Next, we derive the independent paths:

Since V(G) = 4,
there are four paths

Path 1: 1,2,3,6,7,8
Path 2: 1,2,3,5,7,8
Path 3: 1,2,4,7,8
Path 4: 1,2,4,7,2,4,...7,8

Finally, we derive test cases to exercise these paths.

Basis Path Testing Notes



- you don't need a flow chart, but the picture will help when you trace program paths
- Test a base set of linearly Independent paths.
- basis path testing should be applied to critical modules

A *critical module* has one or more of the following characteristics: (1) addresses several

software requirements, (2) has a high level of control (resides relatively high in the

program structure), (3) is complex or error prone, or (4) has definite performance

requirements. Critical modules should be tested as early as is possible. In addition,

regression tests should focus on critical module function.

Deriving Test Cases

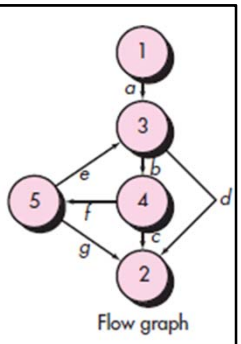
- *Summarizing:*
  - Using the design or code as a foundation, draw a corresponding flow graph.
  - Determine the cyclomatic complexity of the resultant flow graph.
  - Determine a basis set of linearly independent paths.
  - Prepare test cases that will force execution of each path in the basis set.

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.

It is important to note that some independent paths (e.g., path 1 in our example) cannot be tested in stand-alone fashion. That is, the combination of data required to traverse the path cannot be achieved in the normal flow of the program. In such cases, these paths are tested as part of another path test.

Graph Matrices

- A graph matrix is a square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on a flow graph
- Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes.
- By adding a *link weight* to each matrix entry, the graph matrix can become a powerful tool for evaluating program control structure during testing



Flow graph

Node \ Connected to node	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		

Graph matrix

17

the link weight provides additional information about control flow. In its simplest form, the link weight is 1 (a connection exists) or 0 (a connection does not exist). But

link weights can be assigned other, more interesting properties:

• The probability that a link (edge) will be execute.

• The processing time expended during traversal of a link

• The memory required during traversal of a link

• The resources required during traversal of a link.

Testing control structure

- Branch testing — a test case design method that exercises both true and false branch of each decision point contained in a program module. A.k.a, decision testing.
  - It is similar to, but not same as statement coverage.
- Condition testing — a test case design method that exercises both true and false outcomes of the logical conditions contained in a program module
- Data flow testing — selects test paths of a program according to the locations of definitions and uses of variables in the program

18

The basis path testing technique described in Section 18.4 is one of a number of techniques

for control structure testing. Although basis path testing is simple and highly

effective, it is not sufficient in itself. In this section, other variations on control structure

testing are discussed. These broaden testing coverage and improve the quality

of white-box testing.

```
1     # include "Hex_values.h"
2-11  // Description: ...
12    int cgi_decode(char *encoded, char *decoded) f
13        char *eptr = encoded;
14        char *dptr = decoded;
15        int ok= 0;
16        while (*eptr) f
17            char c;
18            c= *eptr;
19            /*Case 1: \+" maps to blank */
20            if (c= = '+') f
21                *dptr= ' ';
22            g else if (c= = '%') f
23                /*Case 2: '%xx' is hex for character xx */
24                int digit_high = Hex_Values[*(++eptr)];
25                int digit_low = Hex_Values[*(++eptr)];
26                /*Hex_Values maps illegal digits to -1 */
27                if (digit_high= = -1 jj digit_low = = -1) f
28                    /* *dptr = '?' */
29                    ok= 1; /* bad return state */
30                g else f
31                    "dptr = 16*digit_high+ digit_low;
32                g
33                /*Case 3: All other chracters map to themslves */
34            g else f
35                *dptr = *eptr;
36            g
37            ++dptr;
38            ++eptr;
39        g
40        *dptr = 'n 0';
41        return ok;
42    g
```

A case for a stronger adequacy criterion.

- Consider line 34-35 are missing from the code.
- T_3={ "+%0D+%4J", ""} still achieve full statement coverage.
  - T_3 fails to reveal the problem with handling "normal" characters.
  - The problem may be avoided if we excise both true and false branch for each control statement.

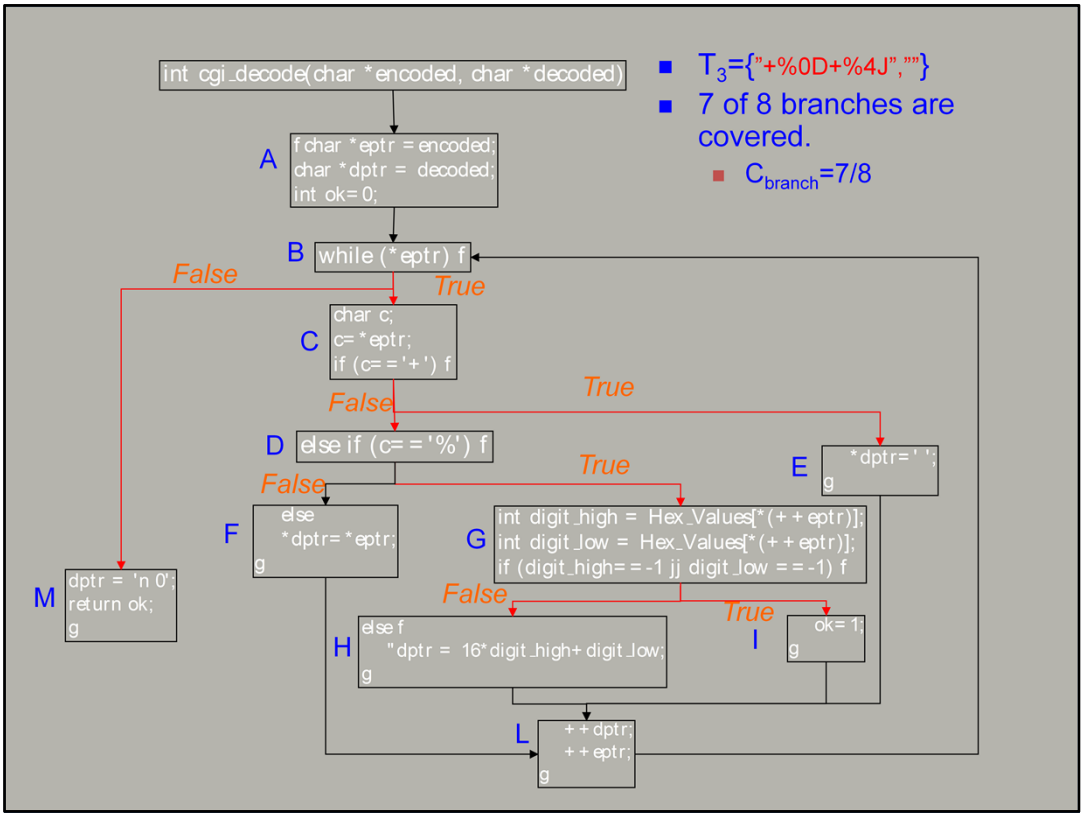
Demonstrate the insufficiency of basis path testing

Branch Testing

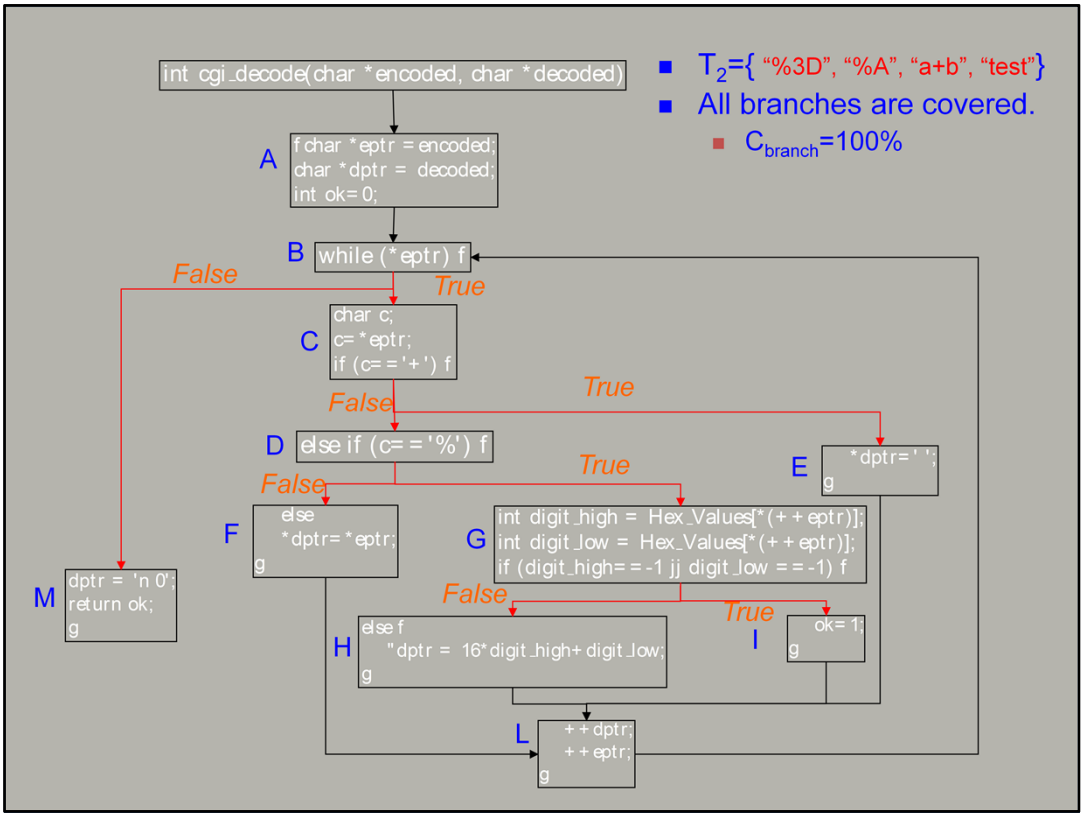
- Branch adequacy criterion requires each branch to be executed at least once.
  - Let T be a test suite for a program P. T satisfies the branch adequacy criterion iff for each branch B of P, there exists at least one test case in T that causes the execution of B.
  - (in control flow graph) Each edge needs to be visited at least once.
  - Also referred as decision coverage.
- Branch coverage criterion.

$$C_{branch} = \frac{\text{number of executed branches}}{\text{number of branches}}$$

  - A test suite T satisfies the branch adequacy criterion for a program P iff $C_{branch}=1$



We have four boolean expression, thus eight branches; the branch F is missed.



So we add tests to cover that missed branch

```
1     # include "Hex_values.h"
2-11  // Description: ...
12    int cgi_decode(char *encoded, char *decoded) f
13        char *eptr = encoded;
14        char *dptr = decoded;
15        int ok= 0;
16        while (*eptr) f
17            char c;
18            c= *eptr;
19            /*Case 1: \+" maps to blank */
20            if (c= = '+') f
21                *dptr= ' ';
22            g else if (c= = '%') f
23                /*Case 2: '%xx' is hex for character xx */
24                int digit_high = Hex_Values[*(++eptr)];
25                int digit_low = Hex_Values[*(++eptr)];
26                /*Hex_Values maps illegal digits to -1 */
27                if (digit_high= = -1 jj digit_low = = -1) f
28                    /* *dptr = '?' */
29                    ok= 1; /* bad return state */
30                g else f
31                    "dptr = 16*digit_high+ digit_low;
32                g
33                /*Case 3: All other chracters map to themslves */
34            g else f
35                *dptr = *eptr;
36            g
37            ++dptr;
38            ++eptr;
39        g
40        *dptr = 'n 0';
41        return ok;
42    g
```

A case for condition adequacy criterion.

- Consider the first condition in line 27 misses "-" sign.
- T_2={ "%3D", "%A", "a+b", "test"} still achieve full branch coverage.
  - T_2 doesn't excise the true outcome of the first condition "digit_high==-1".
  - The problem may be avoided if we excise both true and false outcome for each conditions in control statement.

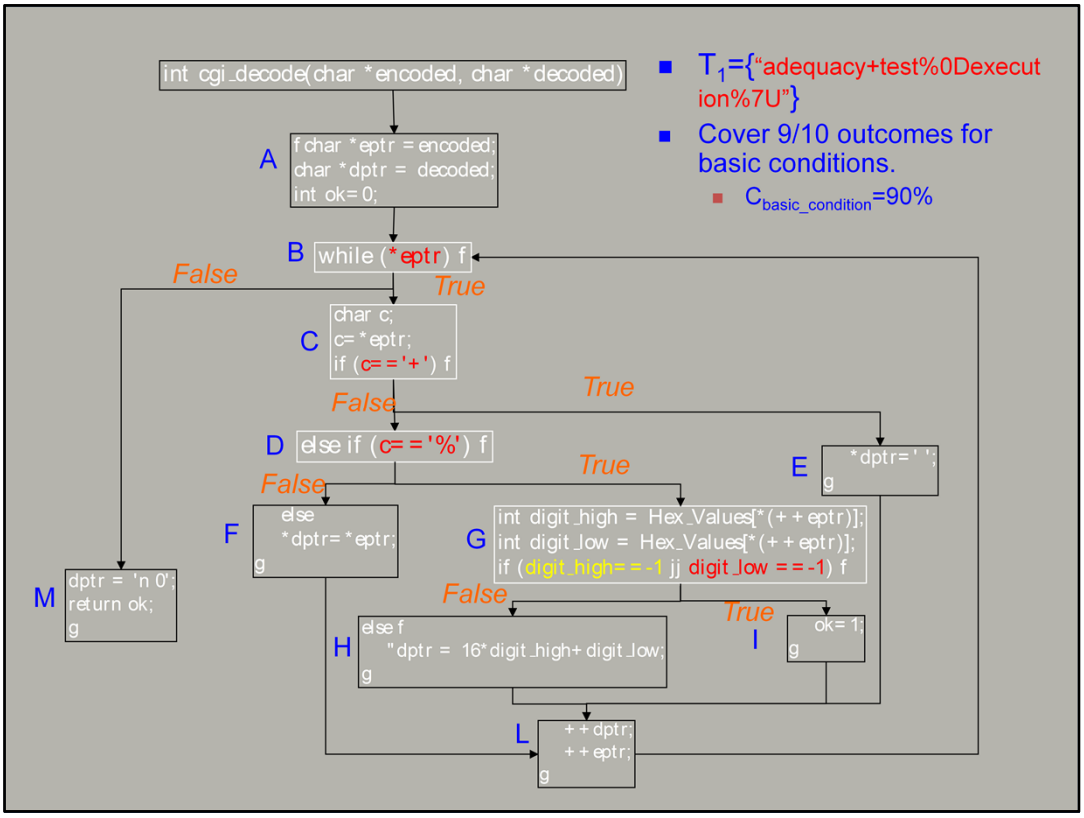
Demonstrate the need for condition testing

Condition Testing

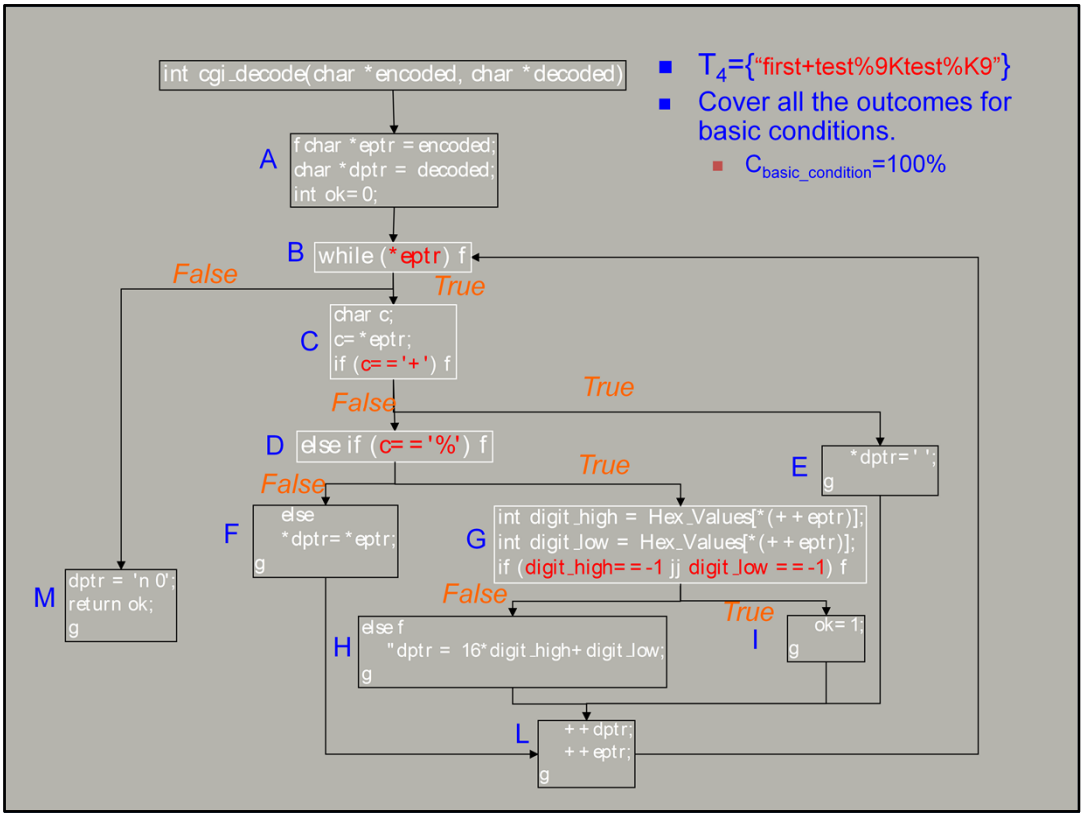
- Condition adequacy criterion requires each basic condition in control statements to be covered for true and false at least once.
  - Let T be a test suite for a program P. T satisfies the branch adequacy criterion iff for each basic condition in P has a true outcome in at least one test cases in T and a false outcome in at least one test case in T.
  - In contrast, branch (decision) coverage only covers the true and false branches of entire Boolean expression in control statements.
- Condition coverage criterion.

$$C_{basic\_conditions} = \frac{\text{total number of truth values assumed by all basic conditions}}{2 \pounds \text{ number of basic conditions}}$$

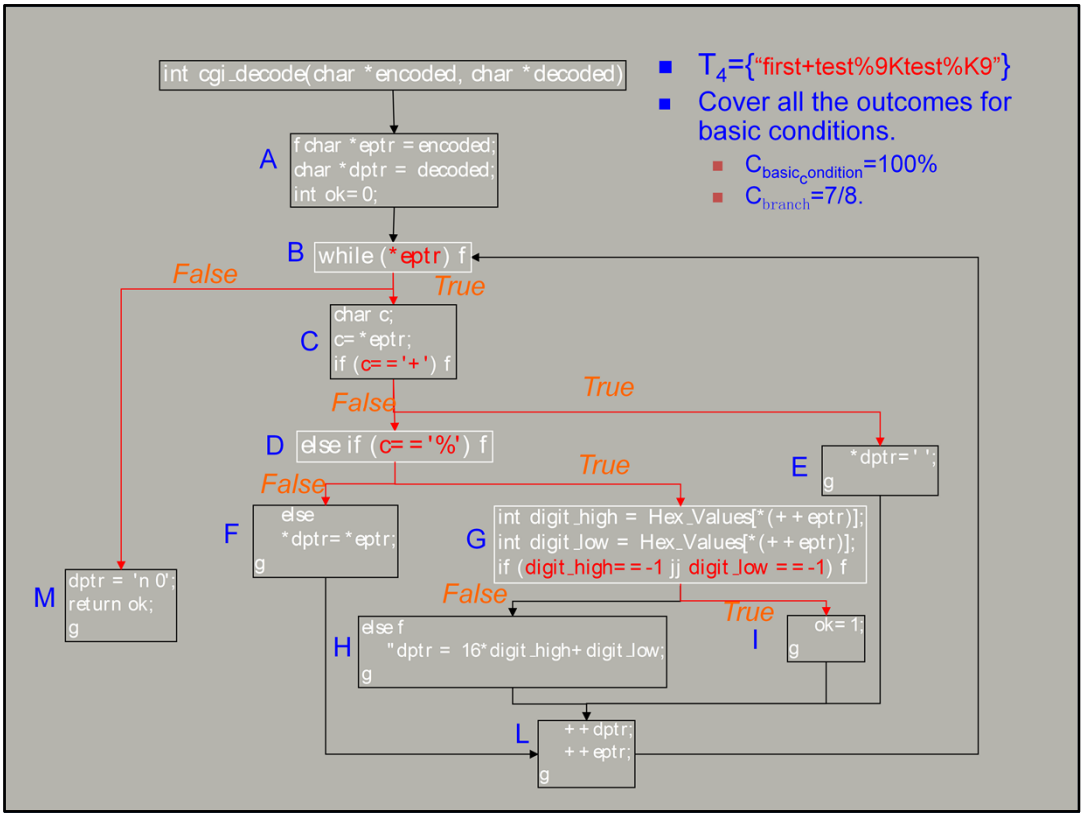
  - A test suite T satisfies condition adequacy criterion for a program P iff $C_{basic\_conditions}=1$
  - Condition Testing doesn't subsume branch testing.



We have five logic conditions, thus 10 outcomes. The ‘digit_high==-1’ condition being true is missed



We changed the test to cover the 'true' outcome of that condition



But with this single test, branch H is not covered. So branch coverage is not sufficient, thus may need another test.

Data-flow testing

- DEF(*S*) {*X* | statement *S* contains a definition of *X*}
- USE(*S*) {*X* | statement *S* contains a use of *X*}
- A *definition-use (DU) chain* of variable X is of the form [*X, S, S'*], where *S* and *S'* are statement numbers, *X* is in DEF(*S*) and USE(*S'*), and the definition of *X* in statement *S* is live at statement *S'*
- *DU Testing*
  - require that every DU chain be covered at least once

28

One simple data flow testing strategy is to require that every DU chain be covered

at least once. We refer to this strategy as the DU testing strategy. It has been shown

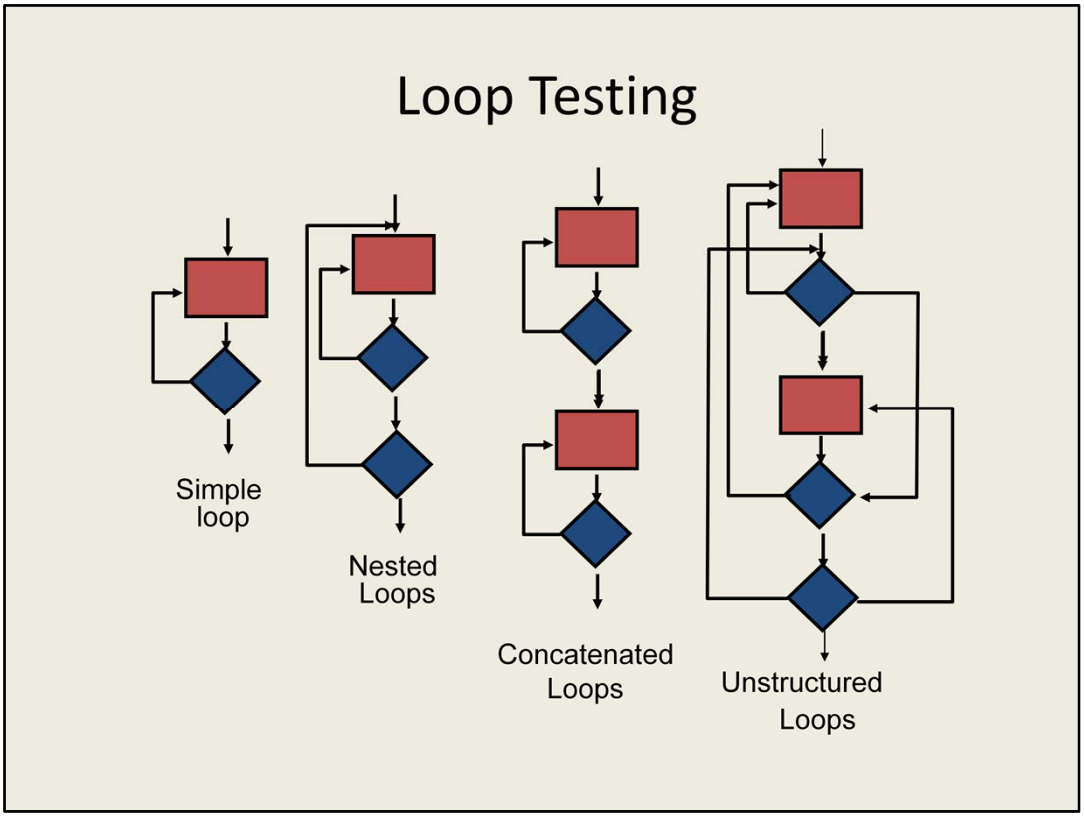
that DU testing does not guarantee the coverage of all branches of a program. However,

a branch is not guaranteed to be covered by DU testing only in rare situations

such as if-then-else constructs in which the *then part* has no definition of any variable

and the *else part* does not exist. In this situation, the else branch of the *if* statement

is not necessarily covered by DU testing.



Loop testing is a white-box testing technique that focuses exclusively on the validity of loop constructs. Four different classes of loops [Bei90] can be defined: simple

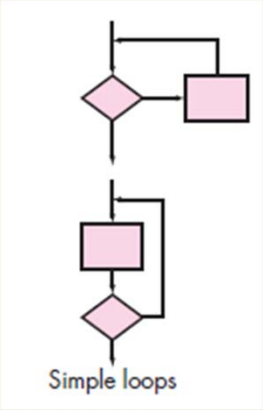
loops, concatenated loops, nested loops, and unstructured loops (Figure 18.7).

Loop Testing: Simple Loops

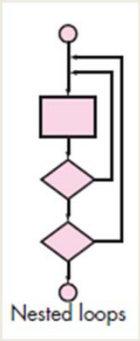
Minimum conditions—Simple Loops

1. **skip the loop entirely**
2. **only one pass through the loop**
3. **two passes through the loop**
4. **m passes through the loop where m < n**
5. **(n-1), n, and (n+1) passes through the loop**

where n is the maximum number of allowable passes



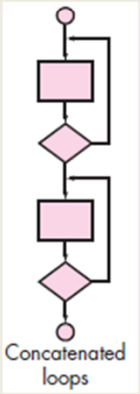
The following set of tests can be applied to simple loops, where *n* is the maximum number of allowable passes through the loop.



Loop Testing: Nested Loops

Nested Loops

1. Start at the innermost loop. Set all outer loops to their **minimum iteration parameter values.**
2. Test the min+1, typical, max-1 and max for the innermost loop, while holding the outer loops at their **minimum values.**
3. Move out one loop and set it up as in step 2, holding all other loops at typical values. Continue this step until the outermost loop has been tested.



Concatenated Loops

If the loops are independent of one another
then treat each as a simple loop
else* treat as nested loops
endif*

for example, the final loop counter value of loop 1 is used to initialize loop 2.

If we were to extend the test approach for simple loops to nested

loops, the number of possible tests would grow geometrically as the level of nesting

increases. This would result in an impractical number of tests.

Concatenated loops can be tested using the approach

defined for simple loops, if each of the loops is independent of the other. However,

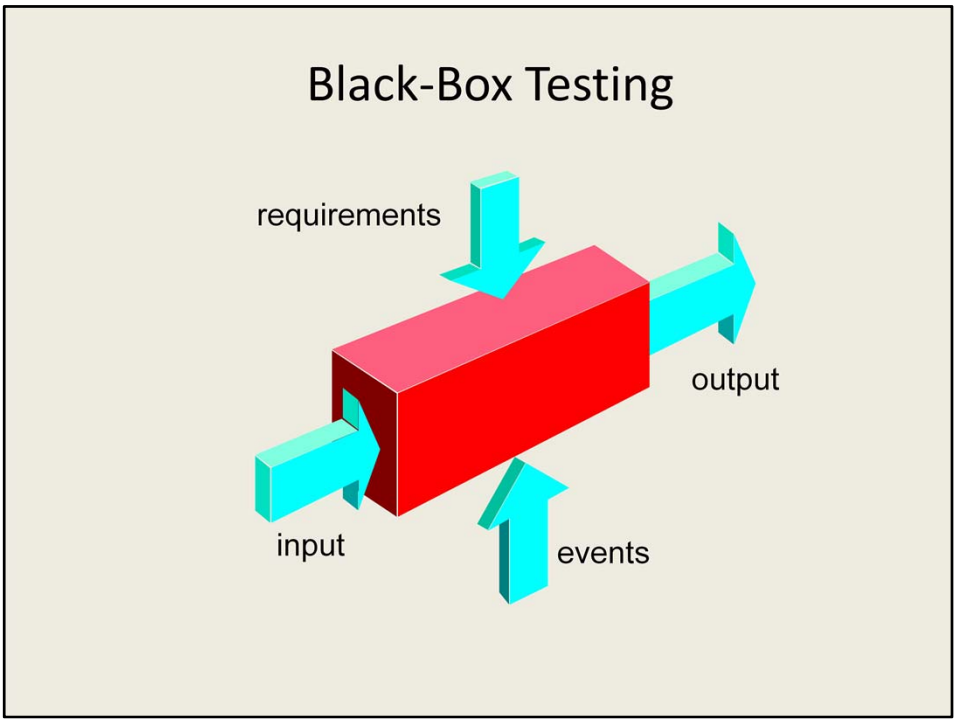
if two loops are concatenated and the loop counter for loop 1 is used as the initial

value for loop 2, then the loops are not independent. When the loops are not independent,

the approach applied to nested loops is recommended.

Unstructured loops. Whenever possible, this class of loops should be redesigned

to reflect the use of the structured programming constructs



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

- How is functional validity tested?
- How is system behavior and performance tested?
- What classes of input will make good test cases?
- Is the system particularly sensitive to certain input values?
- How are the boundaries of a data class isolated?
- What data rates and data volume can the system tolerate?
- What effect will specific combinations of data have on system operation?

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, and (5) initialization and termination errors.

Tests are designed to answer the following questions:

By applying black-box techniques, you derive a set of test cases that satisfy the following

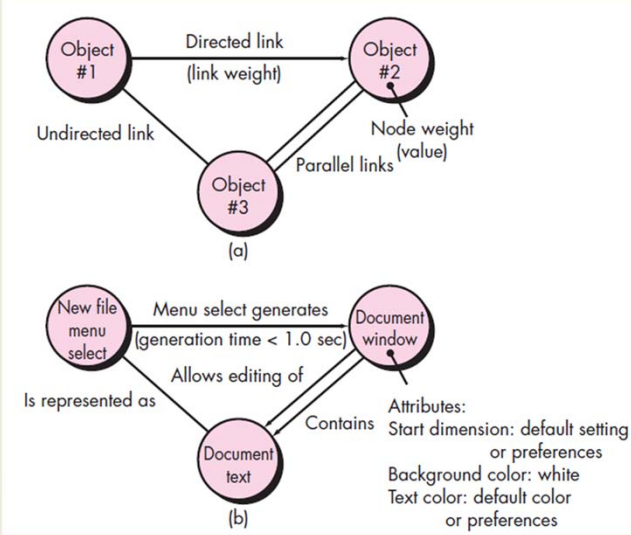
criteria [Mye79]: (1) test cases that reduce, by a count that is greater than one, the number of additional test cases that must be designed to achieve reasonable testing, and (2) test cases that tell you something about the presence or absence of

classes of errors, rather than an error associated only with the specific test at hand

Graph-Based Methods

To understand the objects that are modeled in software and the relationships that connect these objects

In this context, we consider the term "objects" in the broadest possible context. It encompasses data objects, traditional components (modules), and object-oriented elements of computer software.



The first step in black-box testing is to understand the objects5 that are modeled in

software and the relationships that connect these objects

Software testing begins by creating a graph of important objects and their relationships and then devising a series of tests that will cover the graph so that each object and relationship is exercised and errors are uncovered.

you begin by creating a *graph*—a collection of *nodes*

that represent objects, *links* that represent the relationships between objects, *node*

weights that describe the properties of a node (e.g., a specific data value or state

behavior), and *link weights* that describe some characteristic of a link.

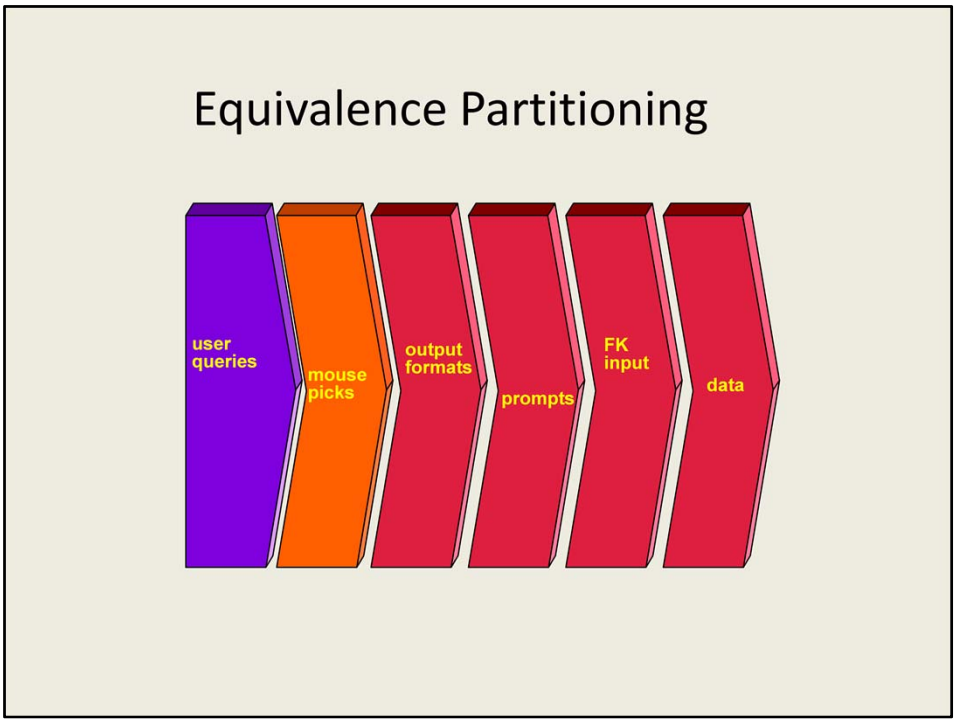
A *directed link* (represented by an arrow) indicates that a relationship moves in only

one direction. A *bidirectional link,* also called a *symmetric link,* implies that the relationship

applies in both directions. *Parallel links* are used when a number of different

relationships are established between graph nodes.

Referring to the figure, a menu select on **newFile** generates a document window. The node weight of **documentWindow** provides a list of the window attributes that are to be expected when the window is generated. The link weight indicates that the window must be generated in less than 1.0 second. An undirected link establishes a symmetric relationship between the **newFile** menu selection and **documentText,** and parallel links indicate relationships between **documentWindow** and **documentText.** In reality, a far more detailed graph would have to be generated as a precursor to test-case design. You can then derive test cases by traversing the graph and covering each of the relationships shown. These test cases are designed in an attempt to find errors in any of the relationships.



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.

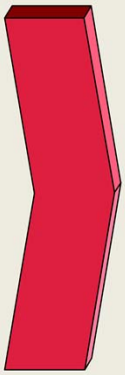
if a set of objects can be linked by relationships that are symmetric transitive, and reflexive, an equivalence class is present [Bei95]. An equivalence

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.

Sample Equivalence Classes



Valid data

- **user supplied commands**
- **responses to system prompts**
- **file names**
- **computational data**
  - **physical parameters**
  - **bounding values**
  - **initiation values**
- output data formatting
- **responses to error messages**
- graphical data (e.g., mouse picks)

Invalid data

- data outside bounds of the program
- **physically impossible data**
- proper value supplied in wrong place

Equivalence classes may be defined according to the
following guidelines:

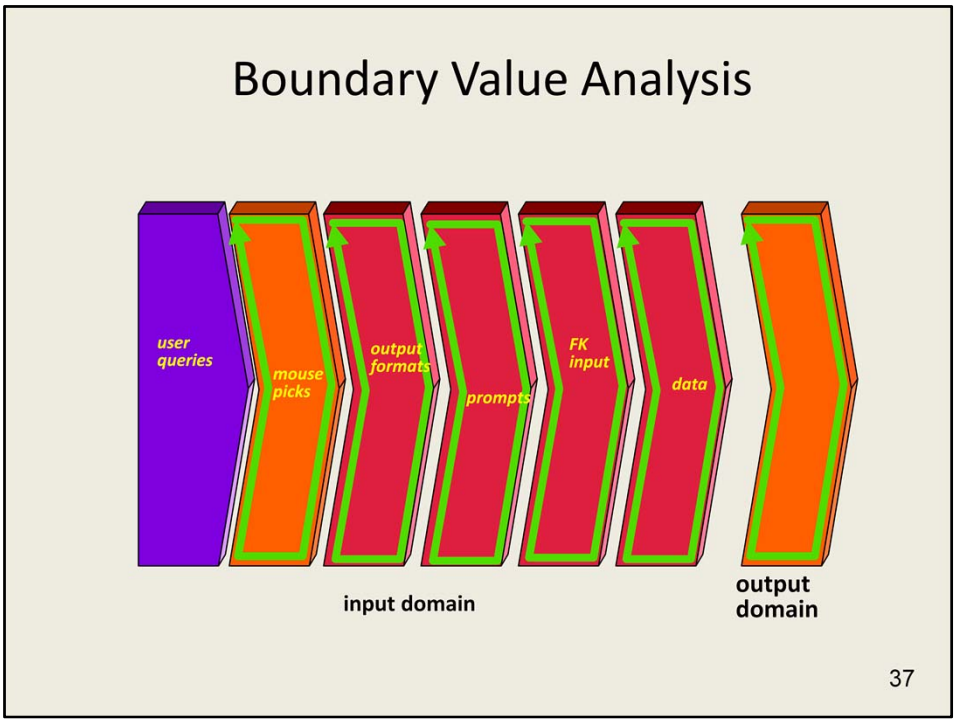
1. If an input condition specifies a range, one valid and two invalid equivalence classes are defined.

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.

Test cases are selected
so that the largest number of attributes of an equivalence class are exercised at once.



A greater number of errors occurs at the boundaries of the input domain rather than

in the “center.” It is for this reason that *boundary value analysis* (BVA) has been developed

as a testing technique. Boundary value analysis leads to a selection of test

cases that exercise bounding values.

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. Rather than focusing

solely on input conditions, BVA derives test cases from the output domain as well

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. For example, assume that a temperature versus pressure table is required as output from an engineering analysis program. *Test cases should be designed to create an output report that produces the maximum (and minimum) allowable number* of table entries.

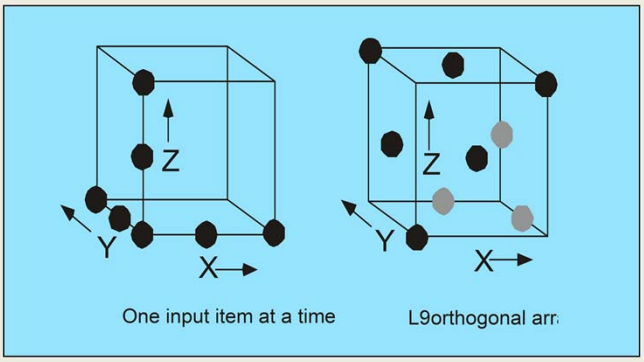
4. If internal program data structures have prescribed boundaries (e.g., a table has a defined limit of 100 entries), be certain to design a test case to exercise the data structure at its boundary.

Comparison Testing

- Used only in situations in which the reliability of software is absolutely critical (e.g., human-rated systems)
  - Separate software engineering teams develop independent versions of an application using the same specification
  - Each version can be tested with the same test data to ensure that all provide identical output
  - Then all versions are executed in parallel with real-time comparison of results to ensure consistency

Orthogonal Array Testing

- Used when the number of input parameters is small and the values that each of the parameters may take are clearly bounded



P499, second paragraph;

There are many applications in which the input domain is relatively limited. That is,

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 and exhaustively test the input domain. However, as the number

of input values grows and the number of discrete values for each data item increases,

exhaustive testing becomes impractical or impossible.

Orthogonal array testing can be applied to problems in which the input domain is

relatively small but too large to accommodate exhaustive testing. The orthogonal

array testing method is particularly useful in finding *region faults*—an error category

associated with faulty logic within a software component.

To illustrate the difference between orthogonal array testing and more conventional

"one input item at a time" approaches, consider a system that has three input

items, *X, Y,* and *Z.* Each of these input items has three discrete values associated with

it. There are 33 27 possible test cases. Phadke [Pha97] suggests a geometric view

of the possible test cases associated with X, Y, and Z illustrated in Figure 18.9.

Referring to the figure, one input item at a time may be varied in sequence along each

input axis. This results in relatively limited coverage of the input domain (represented

by the left-hand cube in the figure).

When orthogonal array testing occurs, an L9 *orthogonal array* of test cases is

created. The L9 orthogonal array has a "balancing property" [Pha97]. That is, test

cases (represented by dark dots in the figure) are "dispersed uniformly throughout

the test domain," as illustrated in the right-hand cube in Figure 18.9. Test coverage

across the input domain is more complete.

If a "one input item at a time" testing strategy were chosen, the following

sequence of tests (P1, P2, P3, P4) would be specified: (1, 1, 1, 1), (2, 1, 1, 1), (3, 1, 1, 1),

(1, 2, 1, 1), (1, 3, 1, 1), (1, 1, 2, 1), (1, 1, 3, 1), (1, 1, 1, 2), and (1, 1, 1, 3).

1,1,1,1

1,2,2,2

1,3,3,3

2,1,2,3

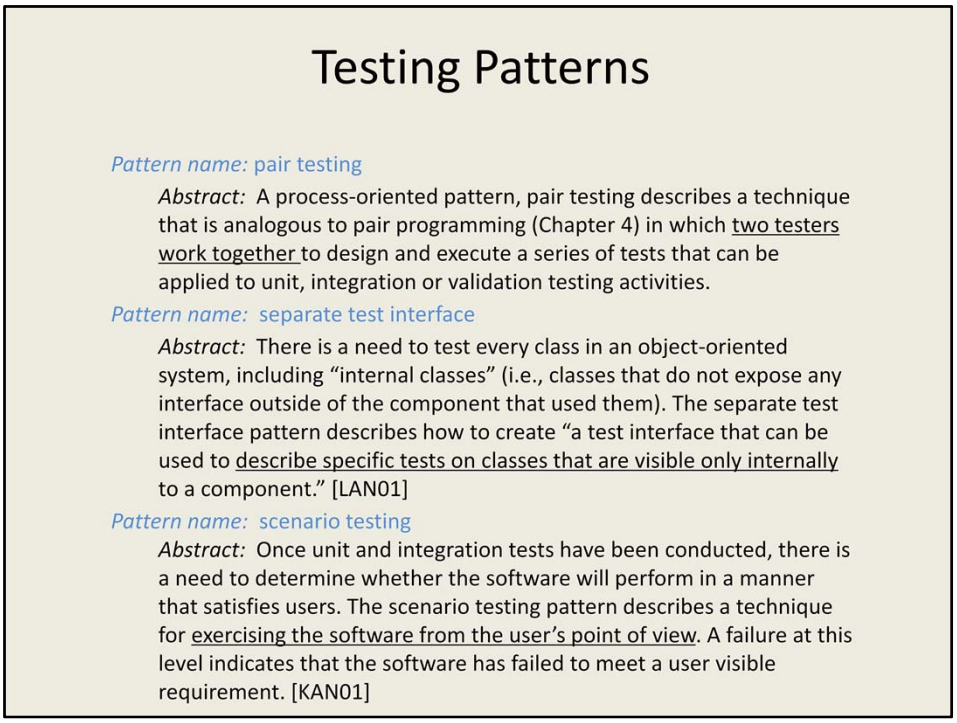
2,2,3,1

2,3,1,2

3,1,3,2

3,2,1,3

3,3,2,1



The use of patterns as a mechanism for describing solutions to specific design problems

was discussed in Chapter 12. But patterns can also be used to propose solutions to other

software engineering situations—in this case, software testing. *Testing patterns* describe

common testing problems and solutions that can assist you in dealing with them.

Testing patterns are described in much the same way as design patterns

(Chapter 12). Dozens of testing patterns have been proposed in the literature (e.g.,

[Mar02]). The following three testing patterns (presented in abstract form only)

provide representative examples:

Summary

- Testability
- White-box testing
  - Basis path testing
  - Branch testing
  - Condition testing
  - Data-flow testing
- Black-box testing
  - Partitioning
  - Boundary analysis