Software Testing Techniques

Slide Set - 15

Organized & Presented By:
Software Engineering Team CSED
TIET, Patiala

Chapter 14
Software Testing Techniques

- Testing fundamentals
- White-box testing
- Black-box testing
- Object-oriented testing methods

(Source: Pressman, R. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 2005)

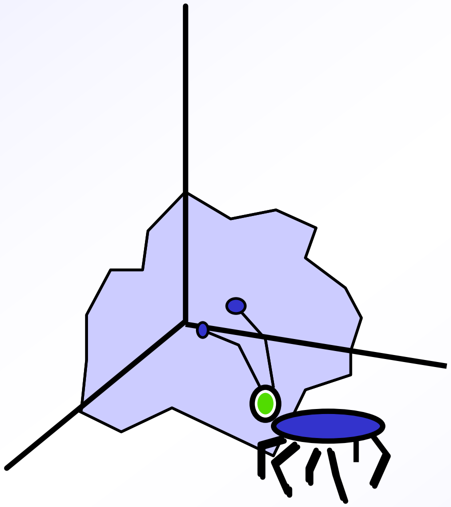
What is a “Good” Test?

- 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”
- A good test should be neither too simple nor too complex

Test Case Design

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

Boris Beizer

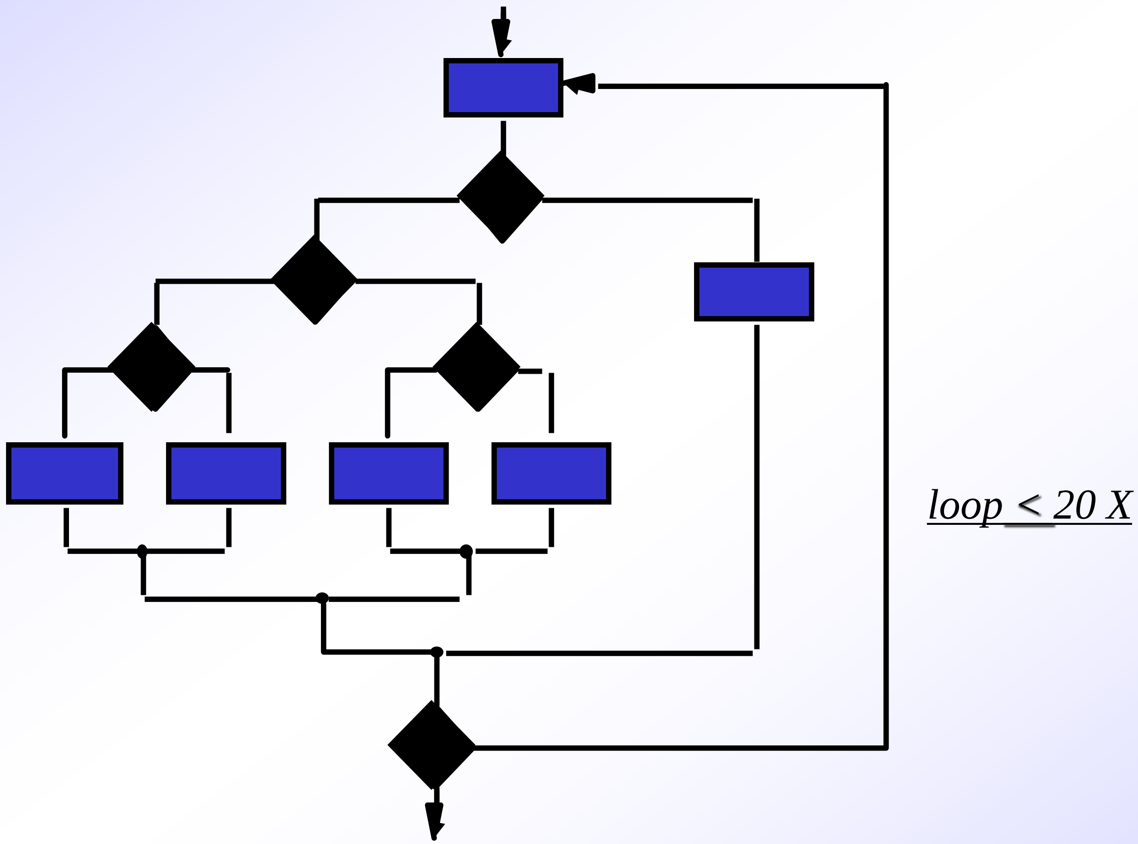


OBJECTIVE *to uncover errors*

CRITERIA *in a complete manner*

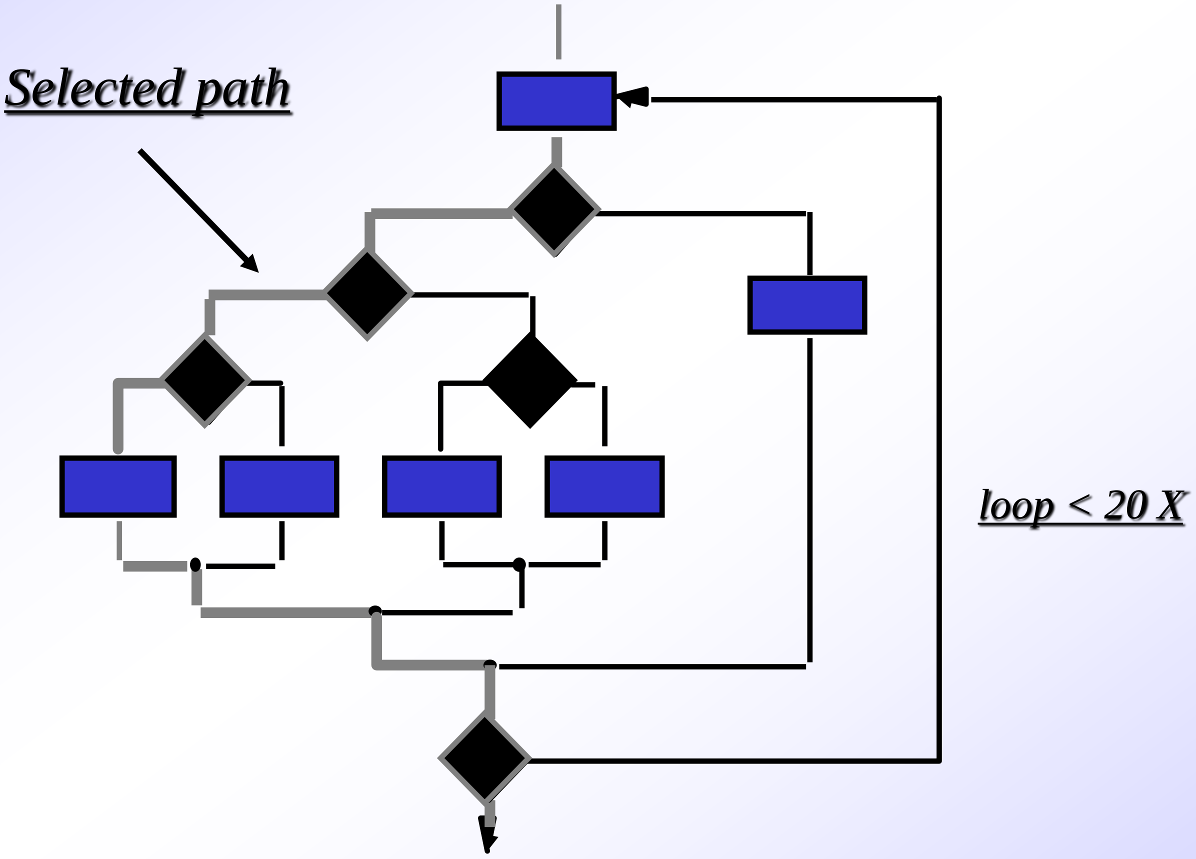
CONSTRAINT *with a minimum of effort and time*

Exhaustive Testing



There are 10^{14} possible paths! If we execute one test per millisecond, it would take 3,170 years to test this program!!

Selective Testing



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*

Characteristics of Testable Software

- Operable
  - The better it works (i.e., better quality), the easier it is to test
- Observable
  - Incorrect output is easily identified; internal errors are automatically detected
- Controllable
  - The states and variables of the software can be controlled directly by the tester
- Decomposable
  - The software is built from independent modules that can be tested independently

(more on next slide)

Characteristics of Testable Software (continued)

- Simple
  - The program should exhibit functional, structural, and code simplicity
- Stable
  - Changes to the software during testing are infrequent and do not invalidate existing tests
- Understandable
  - The architectural design is well understood; documentation is available and organized

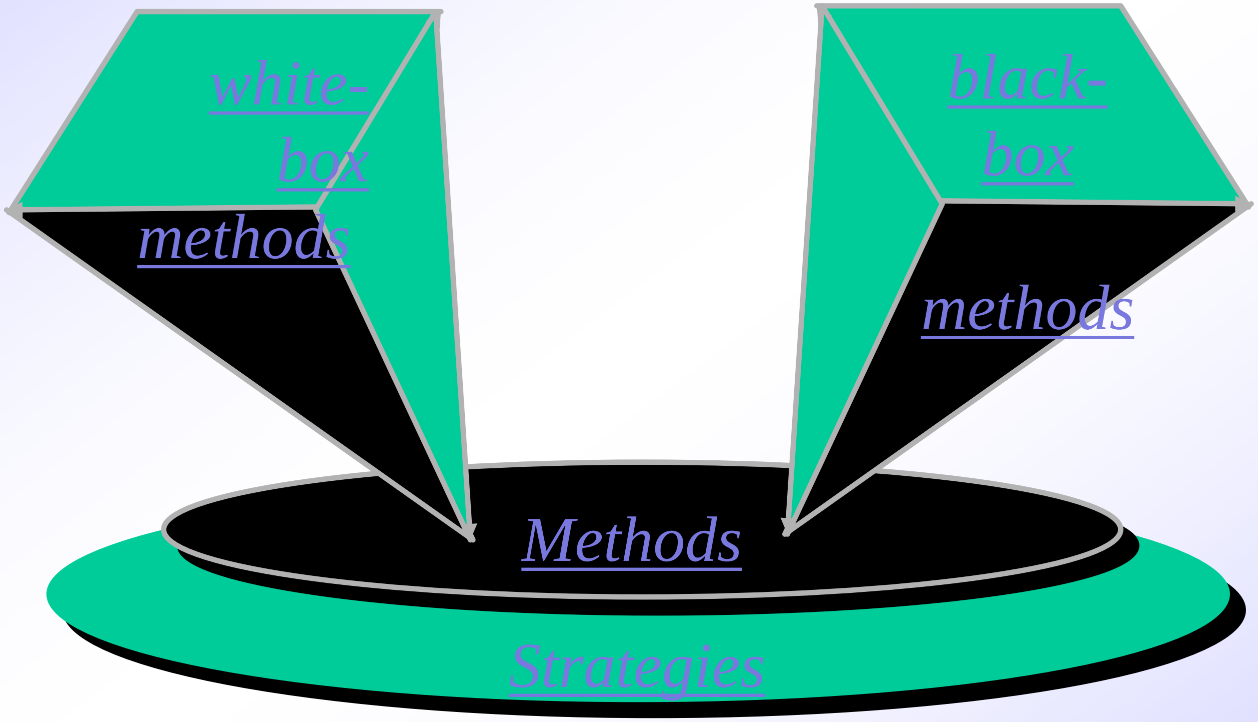
Test Characteristics

- A good test has a high probability of finding an error
  - The tester must understand the software and how it might fail
- A good test is not redundant
  - Testing time is limited; one test should not serve the same purpose as another test
- A good test should be “best of breed”
  - Tests that have the highest likelihood of uncovering a whole class of errors should be used
- A good test should be neither too simple nor too complex
  - Each test should be executed separately; combining a series of tests could cause side effects and mask certain errors

Two Unit Testing Techniques

- Black-box testing
  - Knowing the specified function that a product has been designed to perform, test to see if that function is fully operational and error free
  - Includes tests that are conducted at the software interface
  - Not concerned with internal logical structure of the software
- White-box testing
  - Knowing the internal workings of a product, test that all internal operations are performed according to specifications and all internal components have been exercised
  - Involves tests that concentrate on close examination of procedural detail
  - Logical paths through the software are tested
  - Test cases exercise specific sets of conditions and loops

Software Testing



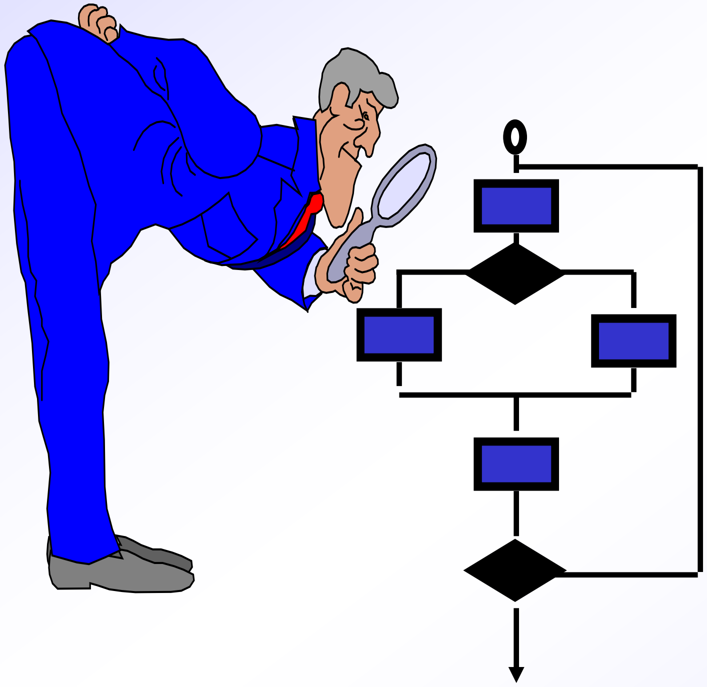
White-box Testing

White-box Testing

- Uses the control structure part of component-level design to derive the test cases
- These test cases
  - Guarantee that all independent paths within a module have been exercised at least once
  - Exercise all logical decisions on their true and false sides
  - Execute all loops at their boundaries and within their operational bounds
  - Exercise internal data structures to ensure their validity

"Bugs lurk in corners and congregate at boundaries"

White-Box Testing



... our goal is to ensure that all statements and conditions have been executed at least once ...

Basis Path Testing

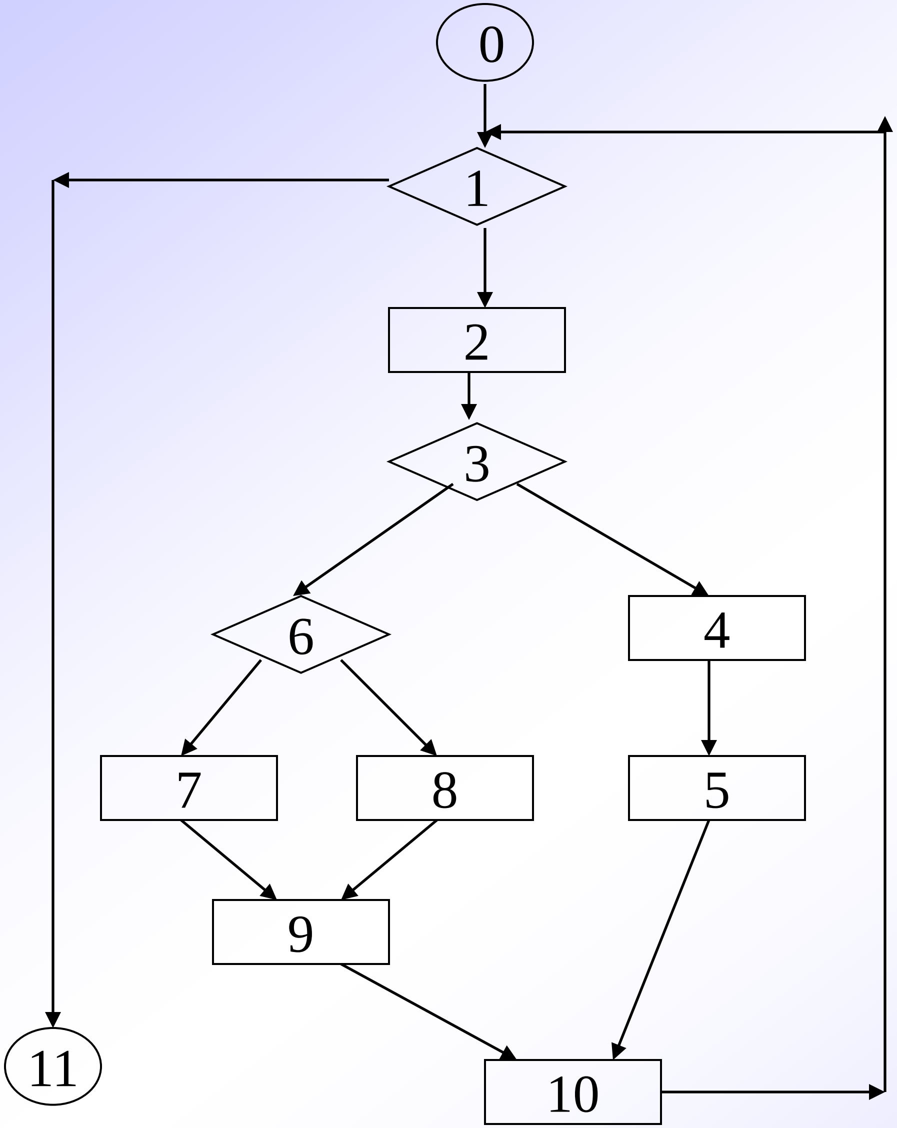
- White-box testing technique proposed by Tom McCabe
- Enables the test case designer to derive a logical complexity measure of a procedural design
- Uses this measure as a guide for defining a basis set of execution paths
- Test cases derived to exercise the basis set are guaranteed to execute <u>every statement</u> in the program <u>at least one time</u> during testing

Flow Graph Notation

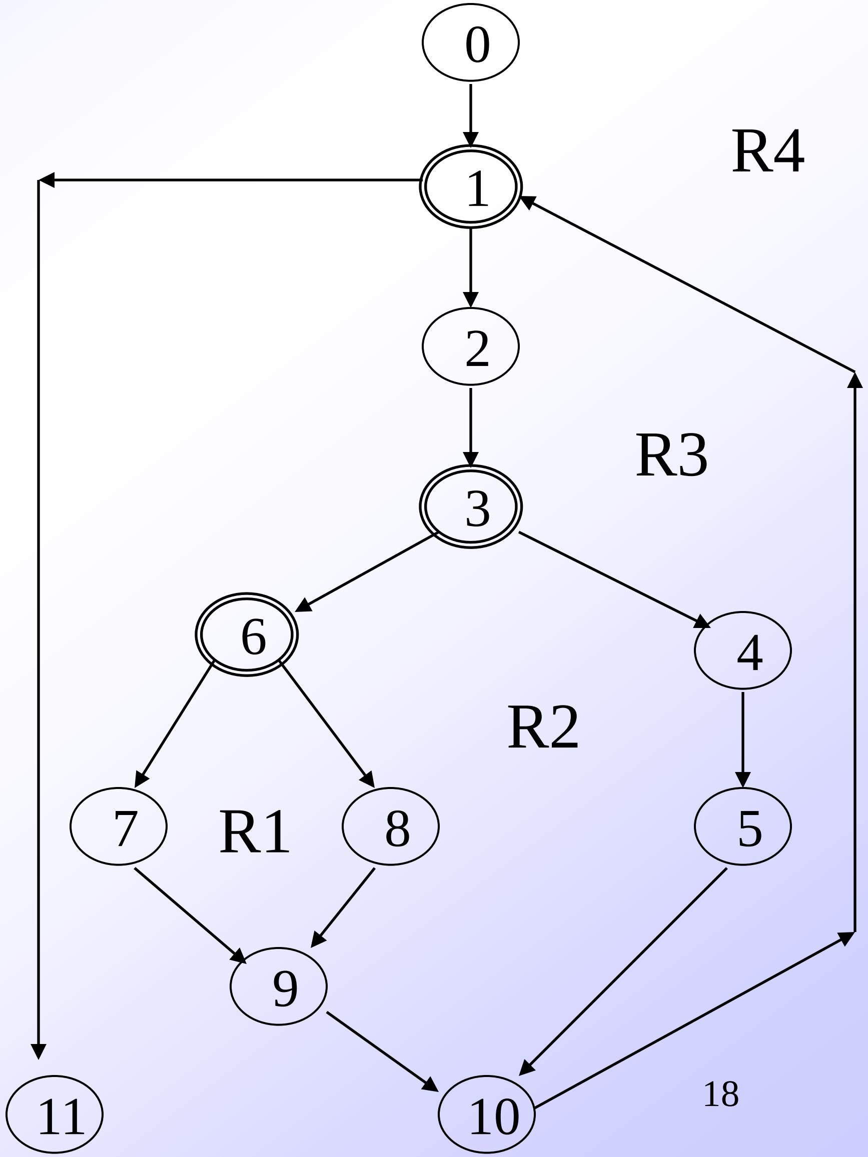
- A circle in a graph represents a <u>node</u>, which stands for a <u>sequence</u> of one or more procedural statements
- A node containing a simple conditional expression is referred to as a <u>predicate node</u>
  - Each <u>compound condition</u> in a conditional expression containing one or more Boolean operators (e.g., and, or) is represented by a separate predicate node
  - A predicate node has <u>two</u> edges leading out from it (True and False)
- An <u>edge</u>, or a link, is a an arrow representing flow of control in a specific direction
  - An edge must start and terminate at a node
  - An edge does not intersect or cross over another edge
- Areas bounded by a set of edges and nodes are called <u>regions</u>
- When counting regions, include the area outside the graph as a region, too

Flow Graph Example

FLOW CHART



FLOW GRAPH



Independent Program Paths

- Defined as a path through the program from the start node until the end node that introduces at least one new set of processing statements or a new condition (i.e., new nodes)
- Must move along <u>at least one</u> edge that has not been traversed before by a previous path
- Basis set for flow graph on previous slide
  - Path 1: 0-1-11
  - Path 2: 0-1-2-3-4-5-10-1-11
  - Path 3: 0-1-2-3-6-8-9-10-1-11
  - Path 4: 0-1-2-3-6-7-9-10-1-11
- The <u>number of paths</u> in the basis set is determined by the <u>cyclomatic complexity</u>

Cyclomatic Complexity

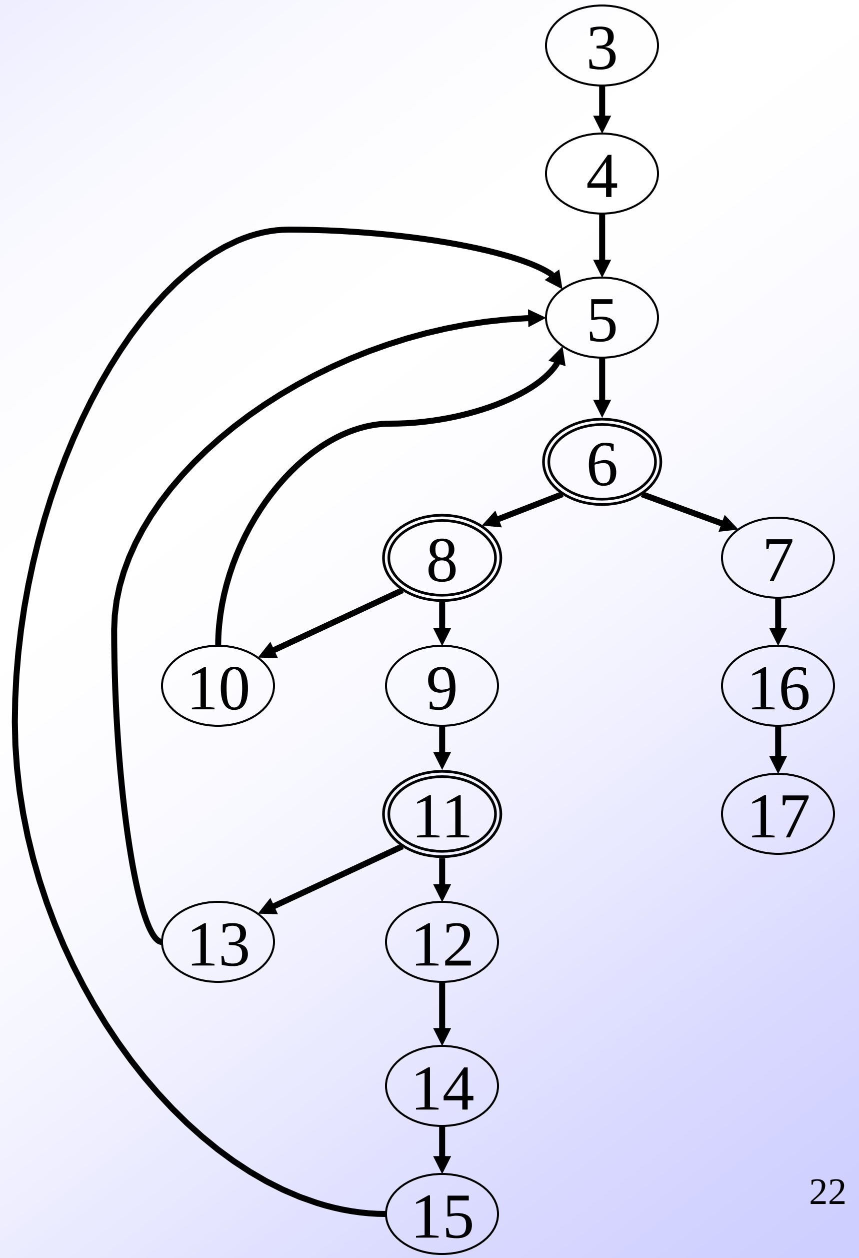
- Provides a quantitative measure of the logical complexity of a program
- Defines the number of independent paths in the basis set
- Provides an upper bound for the number of tests that must be conducted to ensure all statements have been executed at least once
- Can be computed three ways
  - The number of regions
  - V(G) = E – N + 2, where E is the number of edges and N is the number of nodes in graph G
  - V(G) = P + 1, where P is the number of predicate nodes in the flow graph G
- Results in the following equations for the example flow graph
  - Number of regions = 4
  - V(G) = 14 edges – 12 nodes + 2 = 4
  - V(G) = 3 predicate nodes + 1 = 4

Deriving the Basis Set and Test Cases

1) Using the design or code as a foundation, draw a corresponding flow graph
2) Determine the cyclomatic complexity of the resultant flow graph
3) Determine a basis set of linearly independent paths
4) Prepare test cases that will force execution of each path in the basis set

A Second Flow Graph Example

```
 1   int functionY(void)
 2   {
 3      int x = 0;
 4      int y = 19;

 5   A: x++;
 6      if (x > 999)
 7         goto D;
 8      if (x % 11 == 0)
 9         goto B;
10      else goto A;

11   B: if (x % y == 0)
12         goto C;
13      else goto A;

14   C: printf("%d\n", x);
15      goto A;

16   D: printf("End of list\n");
17      return 0;
18   }
```

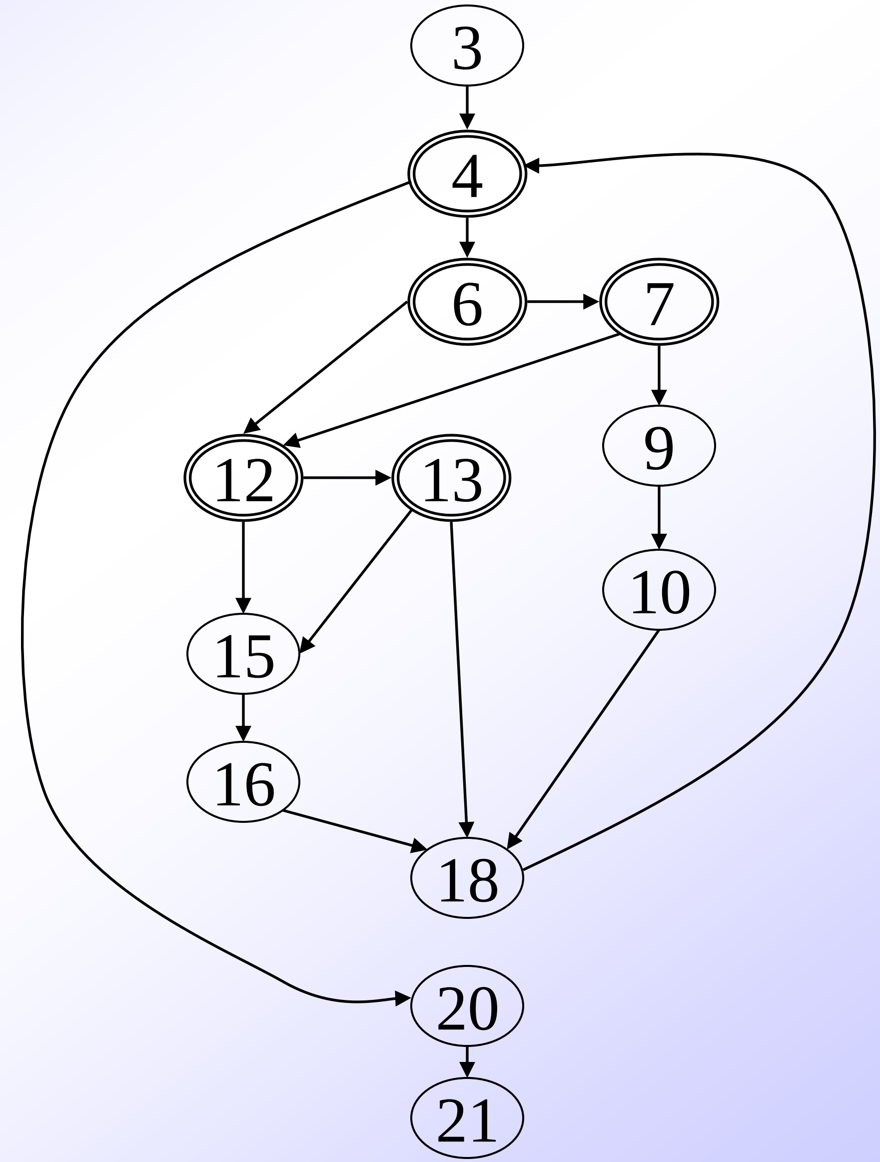


A Sample Function to Diagram and Analyze

```
int functionZ(int y)
{
int x = 0;

while (x <= (y * y))
   {
   if ((x % 11 == 0) &&
        (x % y == 0))
      {
      printf(“%d”, x);
      x++;
      } // End if
   else if ((x % 7 == 0) ||
              (x % y == 1))
      {
      printf(“%d”, y);
      x = x + 2;
      } // End else
   printf(“\n”);
   } // End while

printf("End of list\n");
return 0;
} // End functionZ
```

A Sample Function to Diagram and Analyze

```
 1  int functionZ(int y)
 2  {
 3  int x = 0;

 4  while (x <= (y * y))
 5      {
 6      if ((x % 11 == 0) &&
 7            (x % y == 0))
 8          {
 9          printf("%d", x);
10          x++;
11          } // End if
12      else if ((x % 7 == 0) ||
13                  (x % y == 1))
14          {
15          printf("%d", y);
16          x = x + 2;
17          } // End else
18      printf("\n");
19      } // End while

20  printf("End of list\n");
21  return 0;
22  } // End functionZ
```

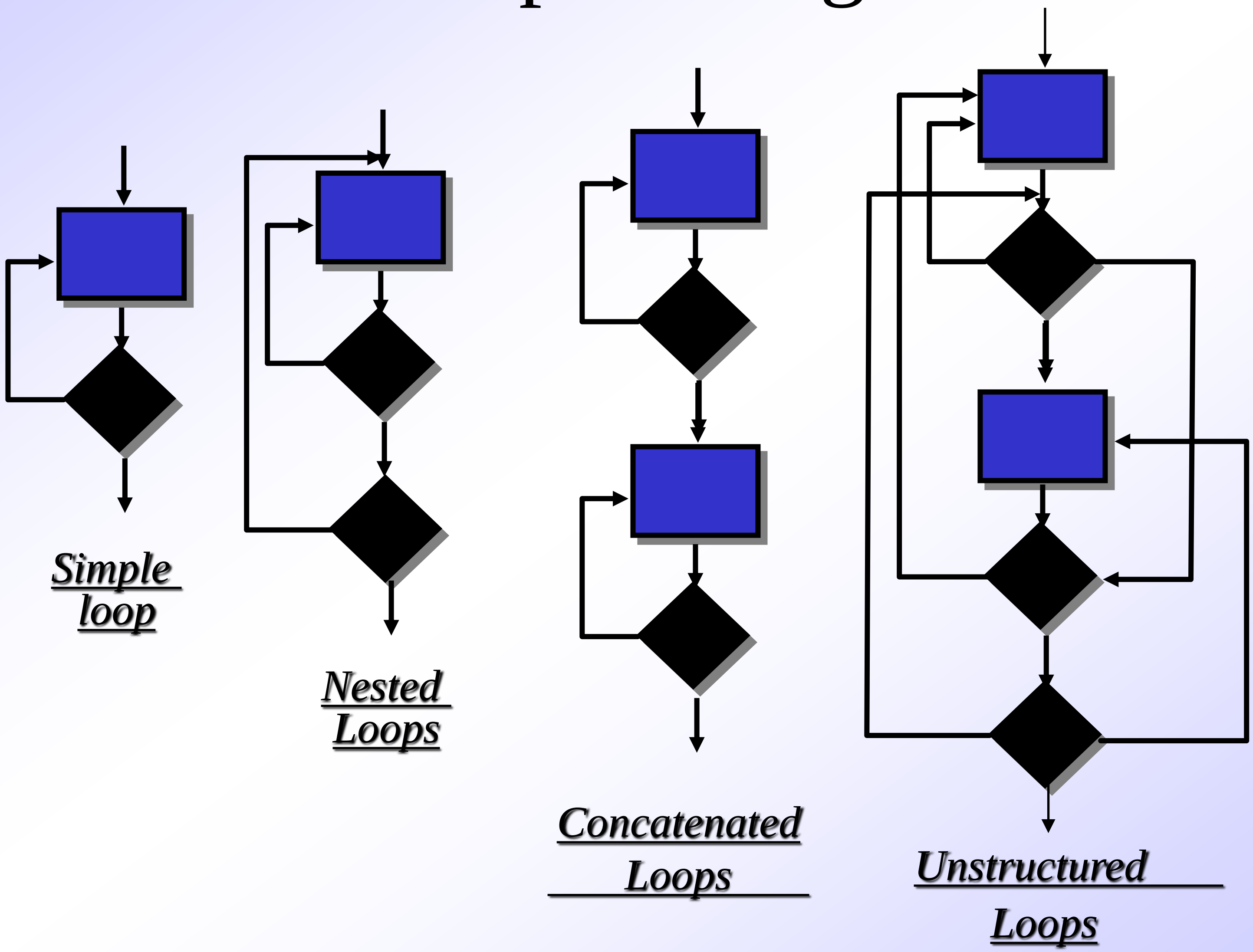


Loop Testing - General

- A white-box testing technique that focuses exclusively on the validity of loop constructs
- Four different classes of loops exist
  - Simple loops
  - Nested loops
  - Concatenated loops
  - Unstructured loops
- Testing occurs by varying the loop boundary values
  - Examples:

```
for (i = 0; i < MAX_INDEX; i++)

while (currentTemp >= MINIMUM_TEMPERATURE)
```

Loop Testing



Testing of 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, $n + 1$ passes through the loop

'n' is the maximum number of allowable passes through the loop

Testing of Nested Loops

1) Start at the innermost loop; set all other loops to minimum values
2) Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter values; add other tests for out-of-range or excluded values
3) Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to "typical" values
4) Continue until all loops have been tested

Testing of Concatenated Loops

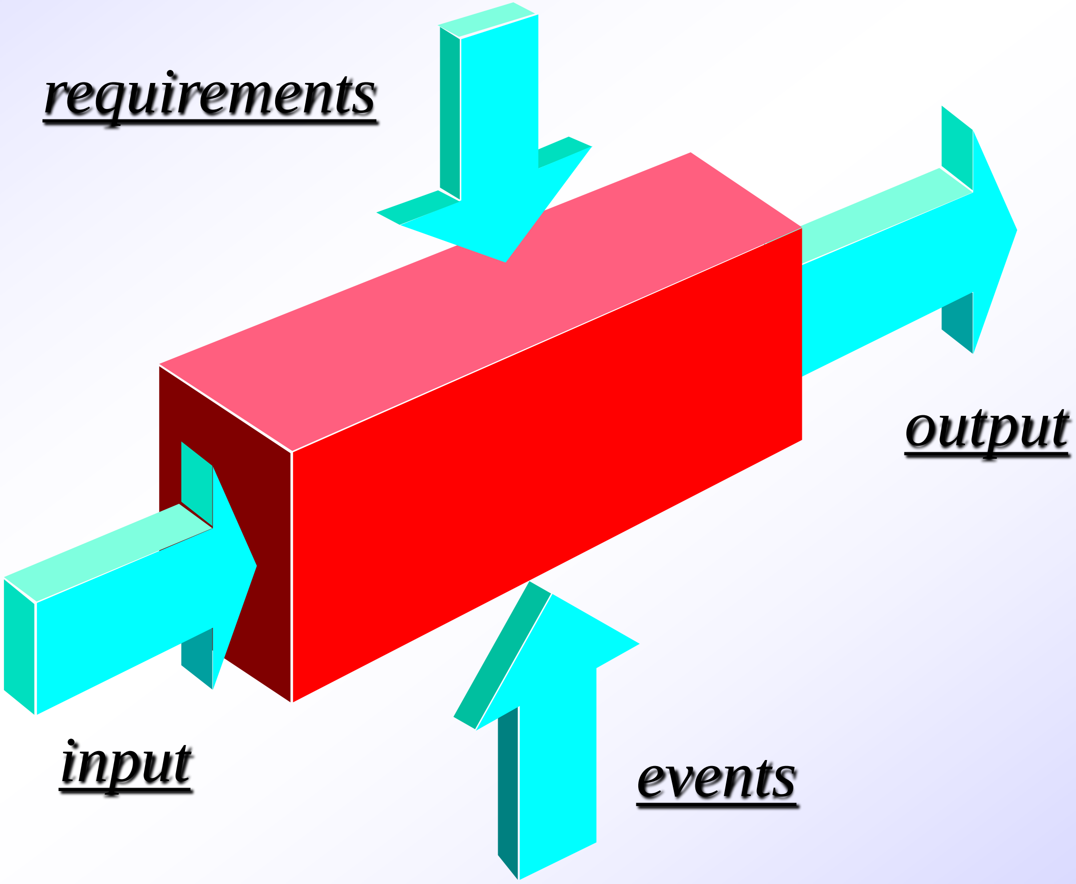
- For independent loops, use the same approach as for simple loops
- Otherwise, use the approach applied for nested loops

Testing of Unstructured Loops

- Redesign the code to reflect the use of structured programming practices
- Depending on the resultant design, apply testing for simple loops, nested loops, or concatenated loops

Black-box Testing

Black-Box Testing



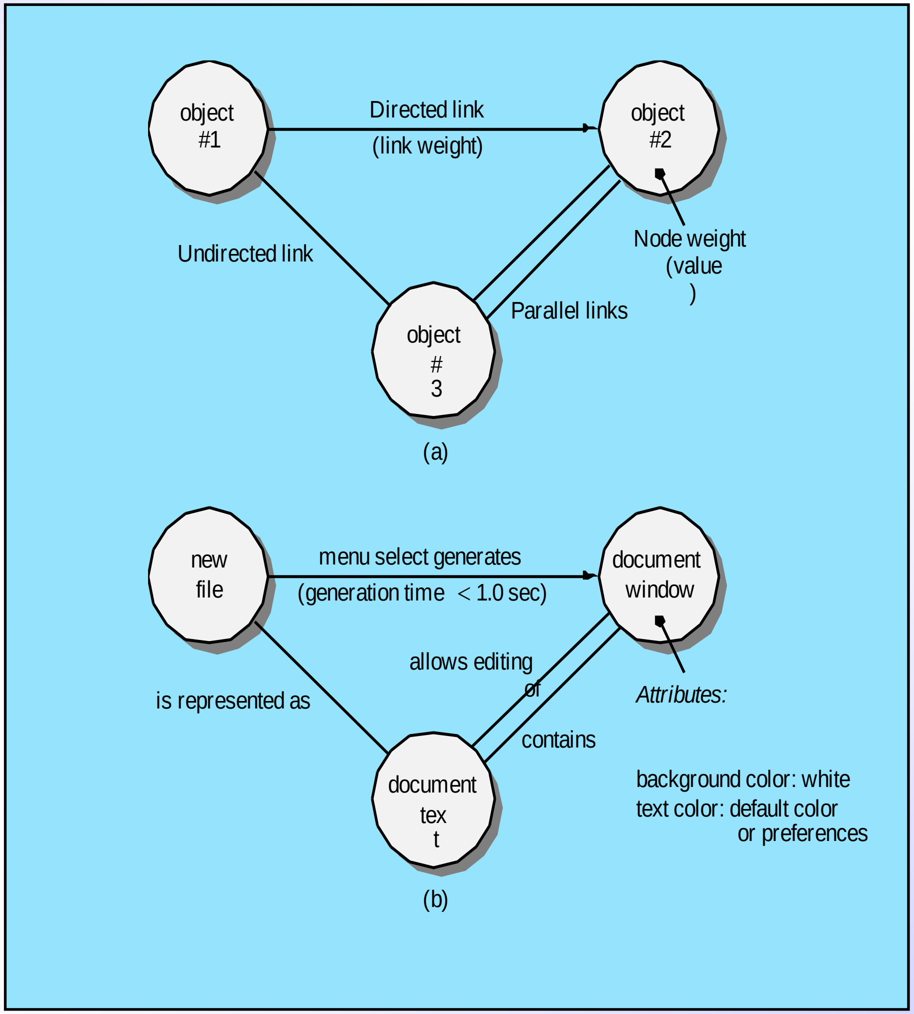
Black-Box Testing

- 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?

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.



Black-box Testing

- Complements white-box testing by uncovering different classes of errors
- Focuses on the functional requirements and the information domain of the software
- Used during the later stages of testing after white box testing has been performed
- The tester identifies a set of input conditions that will fully exercise all functional requirements for a program
- The test cases satisfy the following:
  - Reduce, by a count greater than one, the number of additional test cases that must be designed to achieve reasonable testing
  - Tell us something about the presence or absence of classes of errors, rather than an error associated only with the specific task at hand

Black-box Testing Categories

- Incorrect or missing functions
- Interface errors
- Errors in data structures or external data base access
- Behavior or performance errors
- Initialization and termination errors

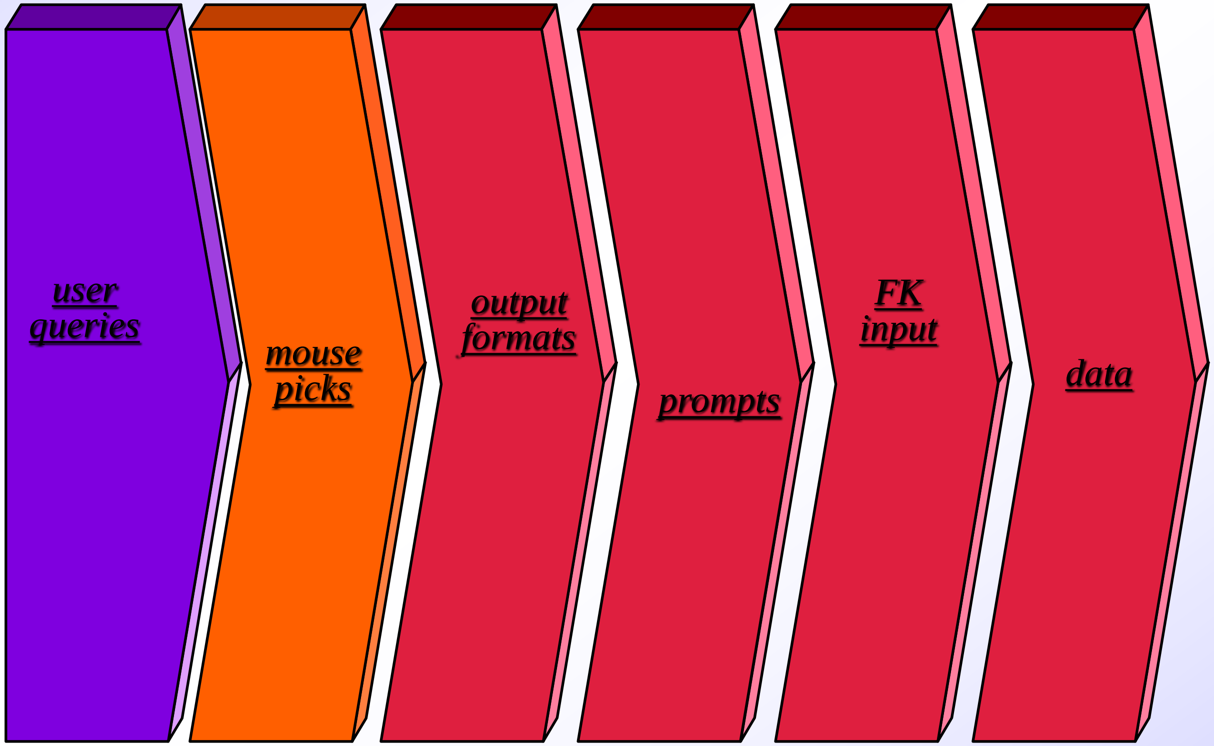
Questions answered by Black-box Testing

- How is functional validity tested?
- How are 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 boundary values 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?

Equivalence Partitioning

- A black-box testing method that divides the input domain of a program into classes of data from which test cases are derived
- An ideal test case single-handedly uncovers a complete class of errors, thereby reducing the total number of test cases that must be developed
- Test case design 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
- From each equivalence class, test cases are selected so that the largest number of attributes of an equivalence class are exercise at once

Equivalence Partitioning



Guidelines for Defining Equivalence Classes

- If an input condition specifies a range, one valid and two invalid equivalence classes are defined
  - Input range: 1 – 10 Eq classes: {1..10}, {x < 1}, {x > 10}
- If an input condition requires a specific value, one valid and two invalid equivalence classes are defined
  - Input value: 250 Eq classes: {250}, {x < 250}, {x > 250}
- If an input condition specifies a member of a set, one valid and one invalid equivalence class are defined
  - Input set: {-2.5, 7.3, 8.4} Eq classes: {-2.5, 7.3, 8.4}, {any other x}
- If an input condition is a Boolean value, one valid and one invalid class are define
  - Input: {true condition} Eq classes: {true condition}, {false 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)

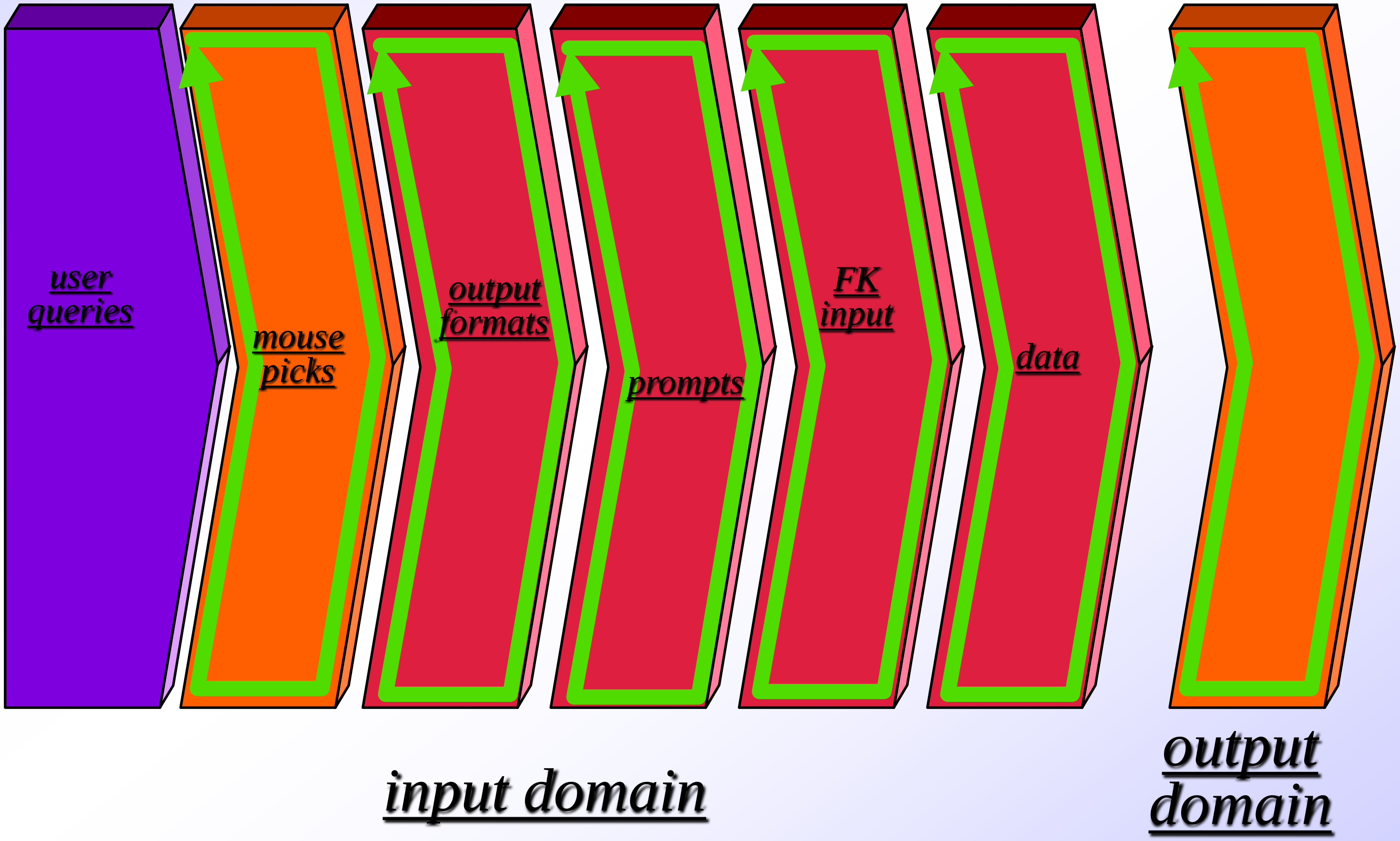
Invalid data

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

Boundary Value Analysis

- A greater number of errors occur at the boundaries of the input domain rather than in the "center"
- Boundary value analysis is a test case design method that complements equivalence partitioning
  - It selects test cases at the edges of a class
  - It derives test cases from both the input domain and output domain

Boundary Value Analysis



Guidelines for Boundary Value Analysis

- 1. If an input condition specifies a range bounded by values *a* and *b*, test cases should be designed with values *a* and *b* as well as values just above and just below *a* and *b*
- 2. If an input condition specifies a number of values, test case should be developed that exercise the minimum and maximum numbers. Values just above and just below the minimum and maximum are also tested
- Apply guidelines 1 and 2 to output conditions; produce output that reflects the minimum and the maximum values expected; also test the values just below and just above
- If internal program data structures have prescribed boundaries (e.g., an array), design a test case to exercise the data structure at its minimum and maximum boundaries