UTC

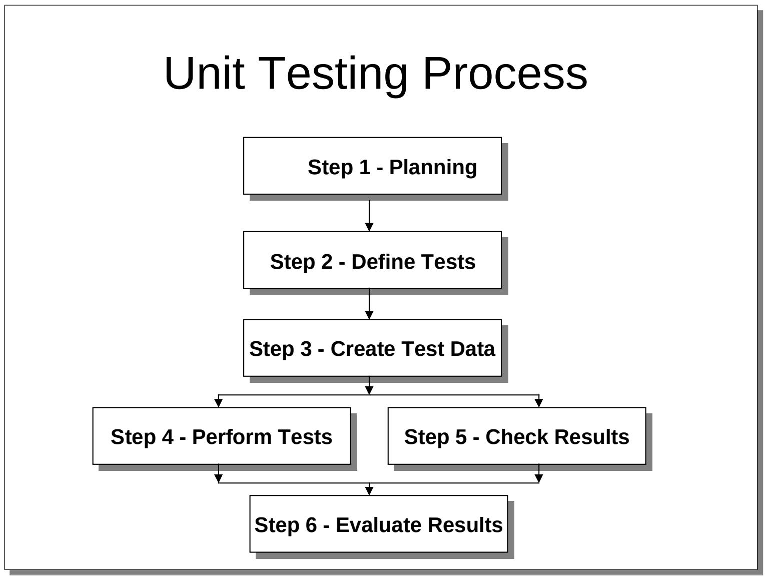
Unit Test Process

Objectives

- Learn a complete process for unit testing
- Learn how to create unit test cases
- Learn how to perform unit testing
- Understand the levels of code coverage
- Learn how to design a structural test
- Understand the role of regression testing and how to perform it
- Learn how to create and maintain test data

Synopsis

Learn a complete process for unit testing.

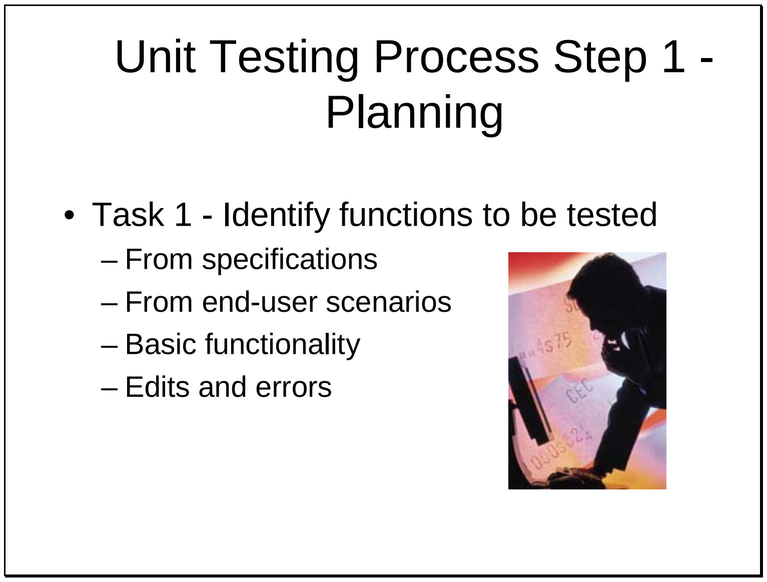


Unit Testing Process

The unit testing process has six major steps:

1. Planning
2. Define Tests
3. Create Test Data
4. Perform Test
5. Check Results
6. Evaluate Results

Each of these steps will be described in detail in this module.



Unit Testing Process Step 1 - Planning

- Task 1 - Identify functions to be tested
  - From specifications
  - From end-user scenarios
  - Basic functionality
  - Edits and errors

Unit Testing Process Step 1 – Planning

Task 1 - Identify functions to be tested

The functions to be tested can be derived from:

- **From specifications**

Examples: program specs, requirements, business rules

- **From end-user scenarios**

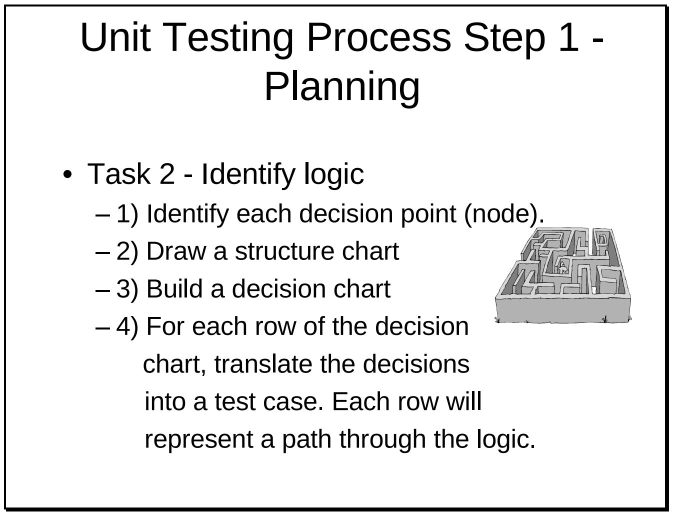
Examples: business processes, use cases, business events

- **Basic functionality**

Examples: menus, buttons, navigation, calculations

- **Edits and errors**

Examples: data entry, file handling



Unit Testing Process Step 1 - Planning

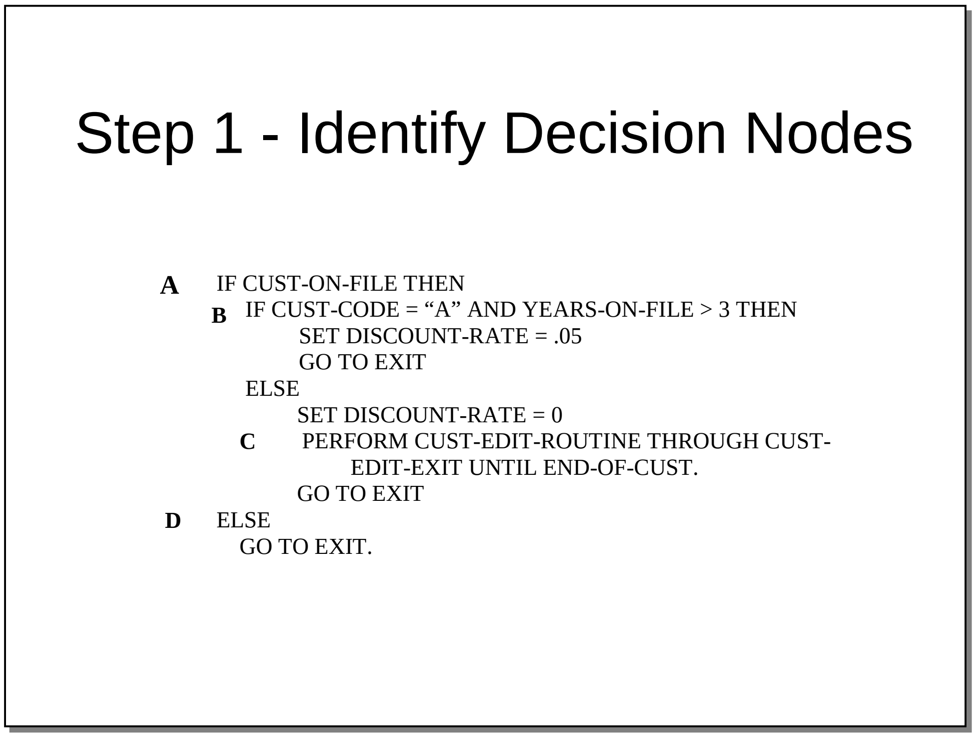
- Task 2 - Identify logic
  - 1) Identify each decision point (node).
  - 2) Draw a structure chart
  - 3) Build a decision chart
  - 4) For each row of the decision chart, translate the decisions into a test case. Each row will represent a path through the logic.

Task 2 - Identify Logic to be Tested

This will form the basis for structural testing. In many cases, only isolated sections of code will be tested at this level, depending on the risk.

To design structural tests:

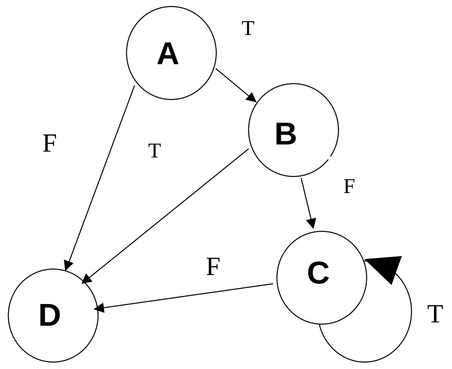
1. Identify each decision point (node).
2. Draw a structure chart
3. Build a decision chart
4. For each row of the decision chart, translate the decisions into a test case. Each row represents a path through the logic.

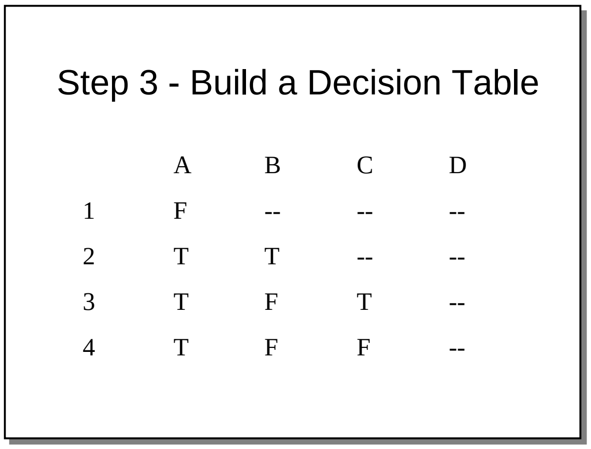


Step 1 - Identify Decision Nodes

```
A    IF CUST-ON-FILE THEN
   B   IF CUST-CODE = "A" AND YEARS-ON-FILE > 3 THEN
              SET DISCOUNT-RATE = .05
              GO TO EXIT
       ELSE
              SET DISCOUNT-RATE = 0
     C        PERFORM CUST-EDIT-ROUTINE THROUGH CUST-
                    EDIT-EXIT UNTIL END-OF-CUST.
              GO TO EXIT
D    ELSE
        GO TO EXIT.
```

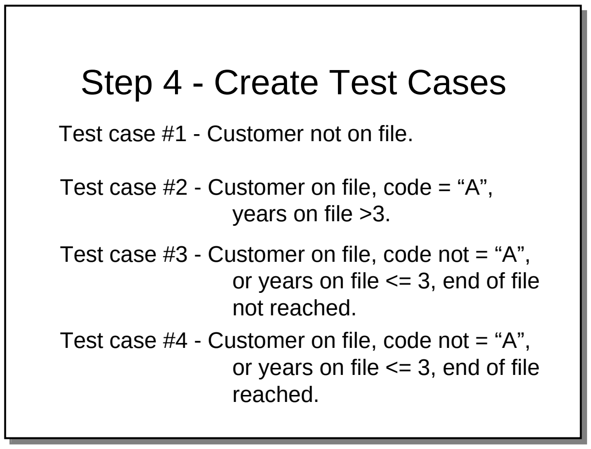
Step 2 - Draw a Structure Chart





Step 3 - Build a Decision Table

	A	B	C	D
1	F	--	--	--
2	T	T	--	--
3	T	F	T	--
4	T	F	F	--



Step 4 - Create Test Cases

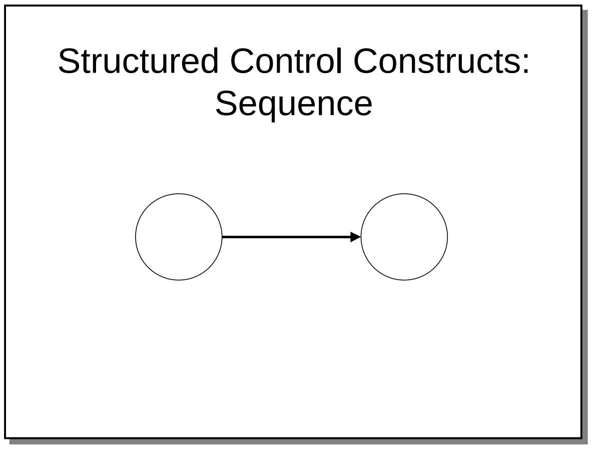
Test case #1 - Customer not on file.

Test case #2 - Customer on file, code = “A”, years on file >3.

Test case #3 - Customer on file, code not = “A”, or years on file <= 3, end of file not reached.

Test case #4 - Customer on file, code not = “A”, or years on file <= 3, end of file reached.

#	Condition	Expected Result	Procedure
1.	Customer not on file.	No discount applied. Exit to end of routine.	1. Enter customer that is not on file and press F4.
2.	Customer on file, customer code = “A”, years on file > 3.	5% discount applied. Exit to end of routine.	1. Enter customer that is on file with code = “A” and press F4.
3.	Customer on file, code not = “A”, end of file not reached.	No discount applied. Keep performing edit routine until last record read.	1. Enter customer that is on file with code not = “A” and multiple customer records to update. 2. Press F4.
4.	Customer on file, code not = “A”, end of file reached.	No discount applied. Exit to end of routine.	1. Enter customer that is on file with code not = “A” and one customer record to update. 2. Press F4.



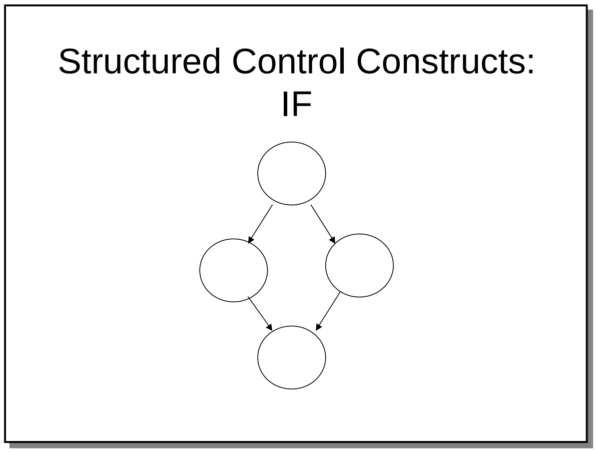
Structured Control Constructs: Sequence

Structured Control Constructs

In drawing structure charts, it is helpful to understand the basic set of constructs that can be combined to graph a module or section of code.

Sequence

The simplest of all structural constructs is the sequence, in which one logical node leads to another. In this construct, there are no branches, conditions or loops.

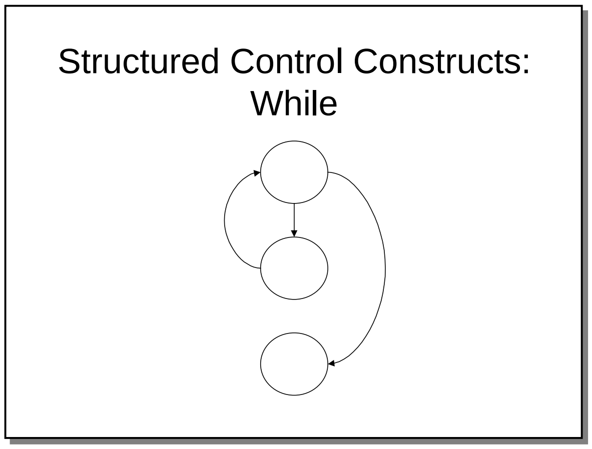


Structured Control Constructs: IF

IF

The IF construct is a simple IF…Else structure:

```
IF CUST-STATUS = “A” THEN
          SET CUST-DISC-RATE TO .05
ELSE
          SET CUST-DISC-RATE TO 0.
```



Structured Control Constructs: While

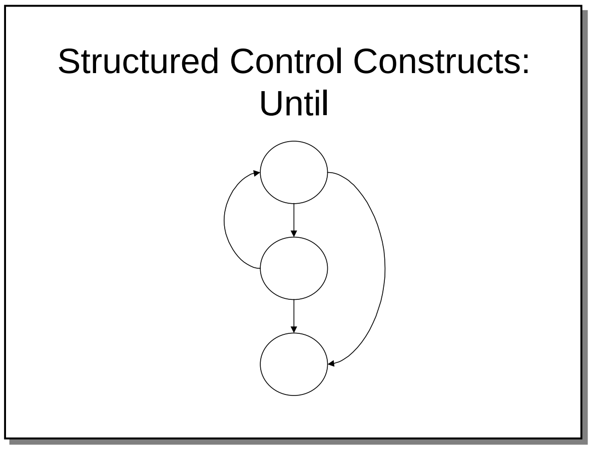
Structured Control Constructs

While

The While construct stays in a loop as long as a certain condition is true. Once the condition is false, a branch is taken to exit the loop.

Example:

PERFORM CUST-DISC-RTE WHILE CUST-STATUS = “A”.



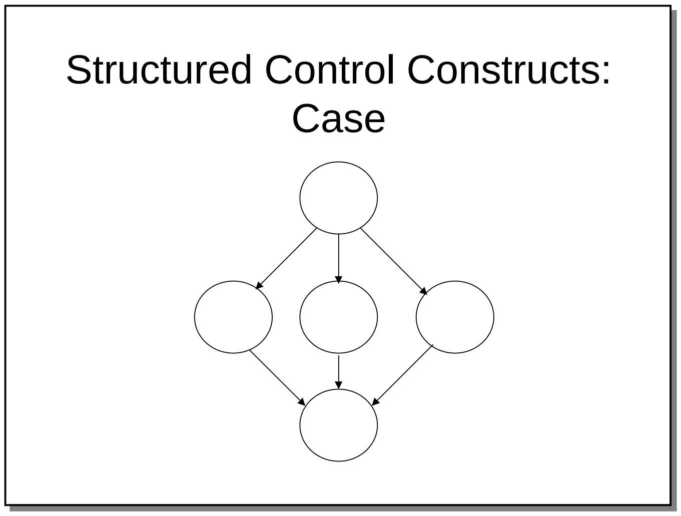
Structured Control Constructs: Until

Until

The Until construct, like the While construct, stays in a loop as long as a certain condition is true. Unlike the While construct, the Until construct branches out of the loop whenever the condition is reached, not necessarily at the initiation of the loop.

Example:

PERFORM CUST-DISC-RTE UNTIL CUST-STATUS = “A”.



Structured Control Constructs: Case

Structured Control Constructs

Case

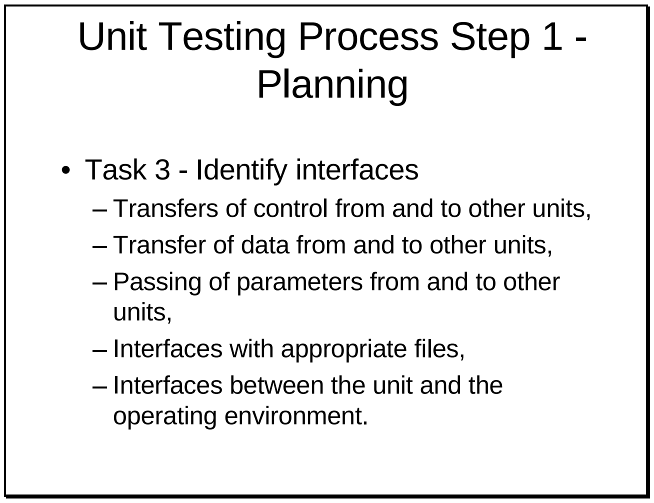
The Case construct branches to any number of nodes, based on a certain condition. Good coding practice terminates the control paths at a common ending point, such as an exit routine.

Example:

CASE

F2: CALC-CUST-RATE
F3: END-RTE
F4: SEARCH

ENDCASE;



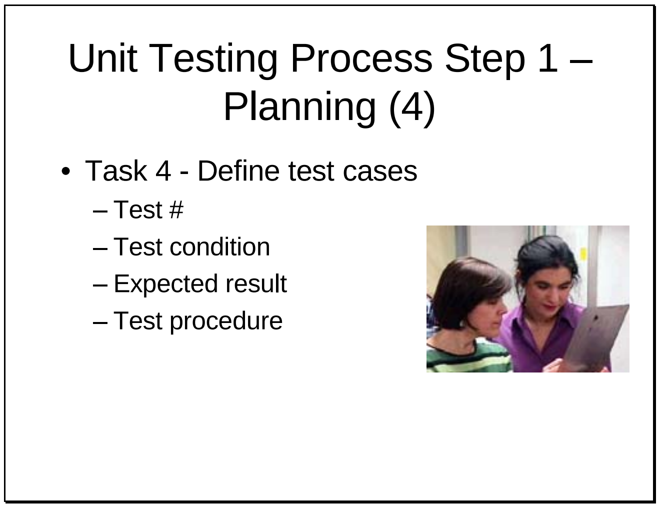
Unit Testing Process Step 1 - Planning

- Task 3 - Identify interfaces
  - Transfers of control from and to other units,
  - Transfer of data from and to other units,
  - Passing of parameters from and to other units,
  - Interfaces with appropriate files,
  - Interfaces between the unit and the operating environment.

Task 3 - Identify Interfaces

Unit testing also can extend outside of the unit to:

- Transfers of control from and to other units,
- Transfer of data from and to other units,
- Passing of parameters from and to other units,
- Interfaces with appropriate files,
- Interfaces between the unit and the operating environment.



Unit Testing Process Step 1 – Planning (4)

- Task 4 - Define test cases
  - Test #
  - Test condition
  - Expected result
  - Test procedure

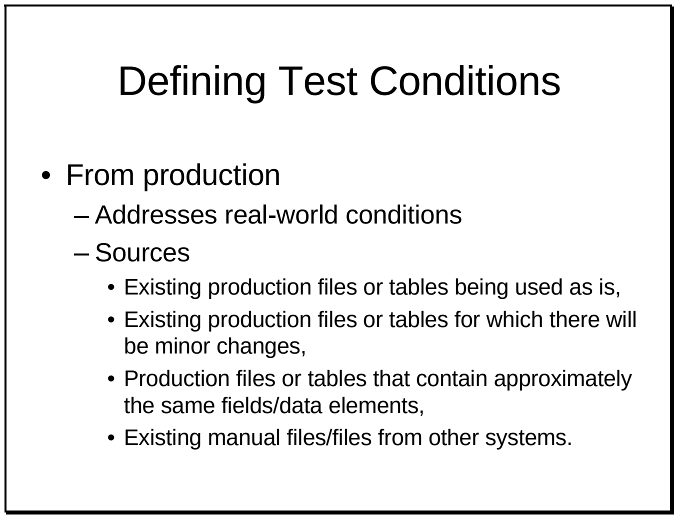
Task 4 – Define Test Cases

A test case can be defined as a combination of:

- **Test condition** – a triggering event or action

- **Expected result** – pre-defined observable outcome

- **Test procedure** – how to set up and perform the test case



Defining Test Conditions

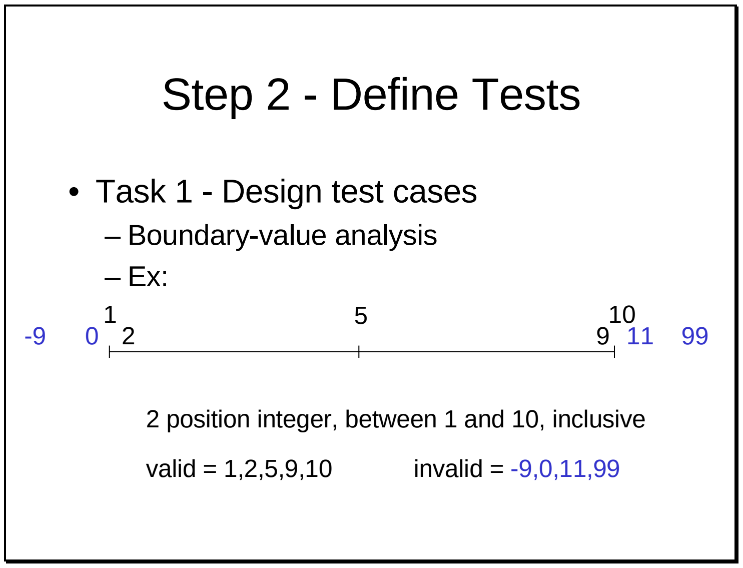
- From production
  - Addresses real-world conditions
  - Sources
    - Existing production files or tables being used as is,
    - Existing production files or tables for which there will be minor changes,
    - Production files or tables that contain approximately the same fields/data elements,
    - Existing manual files/files from other systems.

Defining Test Conditions

There are many sources of test conditions, including the production environment. This source of test conditions addresses some of the real-world cases that a developer or tester may not think of. Some common sources of production test cases include:

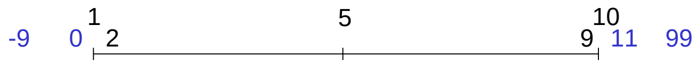
- Existing production files or tables being used as is,
- Existing production files or tables for which there will be minor changes,
- Production files or tables that contain approximately the same fields/data elements,
- Existing manual files/files from other systems.

However, as we will see later in this module, the production environment should not be the only source of test conditions for a variety of reasons.



Step 2 - Define Tests

- Task 1 - Design test cases
  - Boundary-value analysis
  - Ex:



2 position integer, between 1 and 10, inclusive

valid = 1,2,5,9,10 invalid = -9,0,11,99

Step 2 – Define Tests

Boundary Value Analysis

The main idea in this step is to design tests that complete the test conditions identified in the previous step. In addition to the test condition, test cases include pre-defined results and procedures for performing the test.

One technique for designing tests in boundary/value analysis. When a range of values is validated, write test cases that explore the inside and outside "edge" or boundaries of the range.

This type of test case design is performed where thresholds are involved. It is important to realize that the only category of defect this kind of test finds is that of incorrect operators, such as >= being used instead of >. For example a specification may state that customers with a code > 10 are to get a special discount. However, the rule might get coded as customers with a code > or = to 10 get the discount. These tests are designed to be performed around the boundaries of a business rule, therefore other test cases must be designed using other techniques.

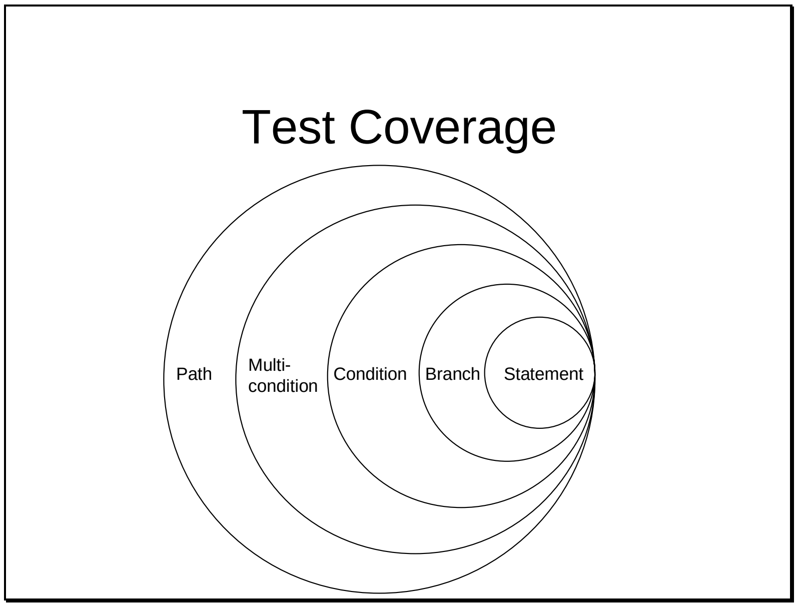
The concept of boundary value analysis is to test just below, right on, and just above the condition. In the above example test cases would be customer codes of 0, 1, 2, 5, 9, 10 and 11. The degree of precision determines the values to be used.

A major downside with this technique is that it gives an illusion of rigorous testing, but doesn’t deliver. You also test more conditions than necessary because you are testing below, on and above each condition. If you knew what the code looked like, you could eliminate some of the cases.

Warning: If you try to test each boundary value case in conjunction with all the other boundary values, the complexity gets astronomical in a hurry.

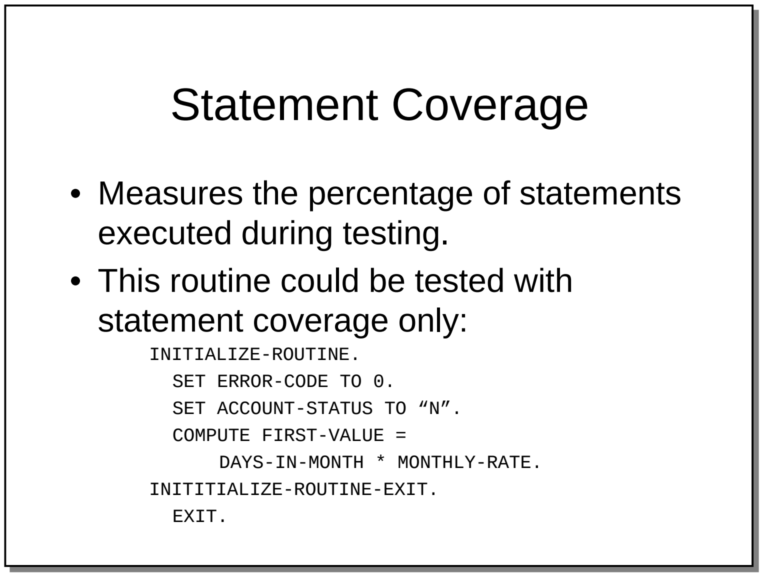
SAMPLE TEST CASE DESCRIPTIONS

#	Test Condition	Expected Result	Test Procedure
1.	At home page, click on “products” button.	Products page should be displayed correctly.	1) Access home page 2) Click on “products” button
2.	At home page, click on “products” text link	Products page should be displayed correctly.	1) Access home page 2) Click on “products” text link
3.	At products page, perform a search by clicking on the pull-down menu each of the following product categories: a) book b) CD c) Video d) DVD	For each type of product selected, the correct product category page should be displayed correctly.	From the Products page: 1) Click on the down arrow on the search pull-down menu 2) Select product category
4.	At the book category product page, enter a book title that is on file in the search field and click “Go” button.	All matching titles should be displayed correctly.	From the Book Product page: 1) Type a book title that is known to be on file 2) Click on the “Go” button.
5.	At the book category product page, enter a book title that is not on file in the search field and click “Go” button.	No matching books displayed.	From the Book Product page: 1) Type a book title that is known not to be on file 2) Click on the “Go” button.
6.	At the book category product page, enter a partial book title that is on file in the search field and click “Go” button.	All matching titles should be displayed correctly.	From the Book Product page: 1) Type a partial book title that is known to be on file 2) Click on the “Go” button.
7.	At the book category product page, enter a book author that is on file in the search field and click “Go” button.	All books by matching authors should be displayed correctly.	From the Book Product page: 1) Type an author that is known to be on file 2) Click on the “Go” button.
8.	At the book category product page, enter a book author that is not on file in the search field and click “Go” button.	No books by matching authors displayed.	From the Book Product page: 1) Type an author that is known not to be on file 2) Click on the “Go” button.
9.	At the book category product page, enter a partial book author that is on file in the search field and click “Go” button.	All books by matching authors should be displayed correctly.	From the Book Product page: 1) Type a partial author that is known to be on file 2) Click on the “Go” button.


Test Coverage

Structural Test Coverage Levels

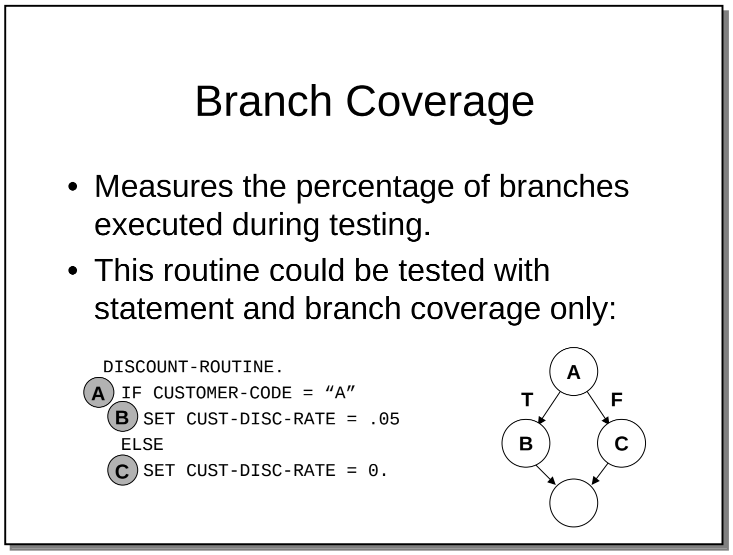
Test coverage is a way to measure the completeness of a test structurally and can be measured in a number of ways, as shown above.


Statement Coverage

- Measures the percentage of statements executed during testing.
- This routine could be tested with statement coverage only:

```
INITIALIZE-ROUTINE.
  SET ERROR-CODE TO 0.
  SET ACCOUNT-STATUS TO “N”.
  COMPUTE FIRST-VALUE =
      DAYS-IN-MONTH * MONTHLY-RATE.
INITITIALIZE-ROUTINE-EXIT.
  EXIT.
```

- **Statement coverage** is the lowest level of coverage and indicates that a line of code has been executed at least once.



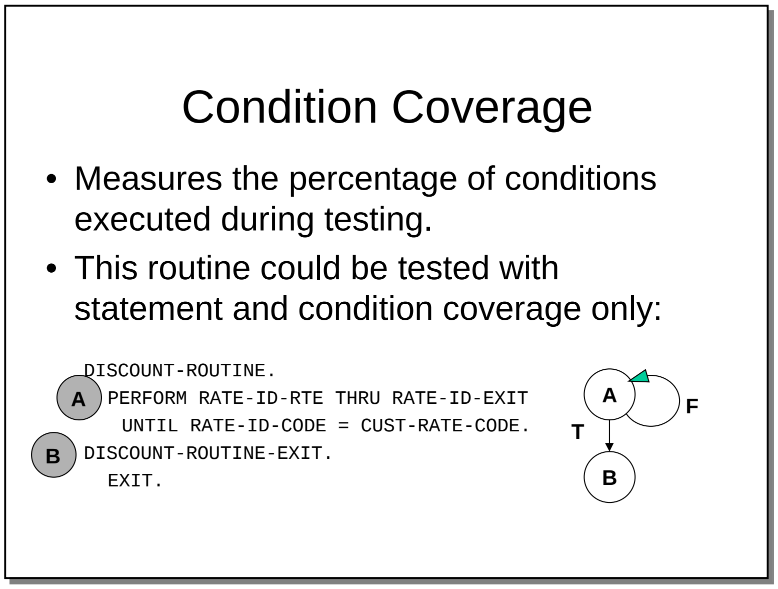
Branch Coverage

- Measures the percentage of branches executed during testing.
- This routine could be tested with statement and branch coverage only:

```
  DISCOUNT-ROUTINE.
A   IF CUSTOMER-CODE = “A”
B     SET CUST-DISC-RATE = .05
    ELSE
C     SET CUST-DISC-RATE = 0.
```

Branch Coverage

- **Branch coverage** indicates that both “True” and “False” branches have been covered in code logic.



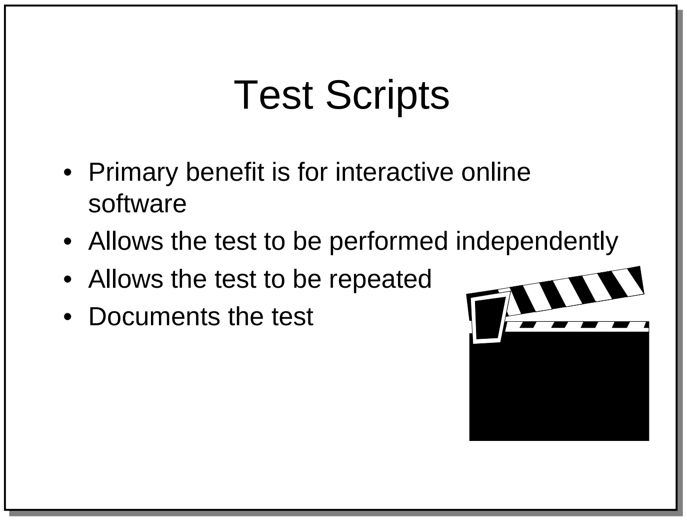
Condition Coverage

- Measures the percentage of conditions executed during testing.
- This routine could be tested with statement and condition coverage only:

```
  DISCOUNT-ROUTINE.
A   PERFORM RATE-ID-RTE THRU RATE-ID-EXIT
      UNTIL RATE-ID-CODE = CUST-RATE-CODE.
B DISCOUNT-ROUTINE-EXIT.
    EXIT.
```

Condition Coverage

- **Condition coverage** indicates that all non-binary decisions have been covered, such as “Do/While” loops, case constructs, etc.



Test Scripts

- Primary benefit is for interactive online software
- Allows the test to be performed independently
- Allows the test to be repeated
- Documents the test

Test Scripts

Test scripts are a very helpful tool for testing interactive online software, such as CICS transactions. Test scripts add a great degree of rigor to your testing effort, but they do require a significant effort to write manually. The choice of whether or not to use test scripts depends on amount of test planning time and resources available.

If the test is to be performed by someone other that the designer of the test, scripts are almost a necessity to convey exactly what should be done in performing the test. In addition, test scripts allow the test to be repeated. This is needed if a defect is found and the fix needs to be re-tested.

Another major benefit of test scripts is that they document the test. If there is ever any doubt after the test about which functions were tested, test scripts are a good reference.

The sample test script shown on the following page shows both good and bad practice.

Good practice

You will notice that specific data values are not indicated on the test script. The reason is that embedding specific data in a test script requires a separate test script for each instance on test data. So, if there are 200 test data items, 200 test scripts will be required. Each item of test documentation will likely require maintenance. For example, if the process to be tested changes, or if the data changes, then every piece of test documentation must be changed. Therefore, keeping specific data out of the test scripts and test cases will reduce the test documentation maintenance burden.

Bad practice

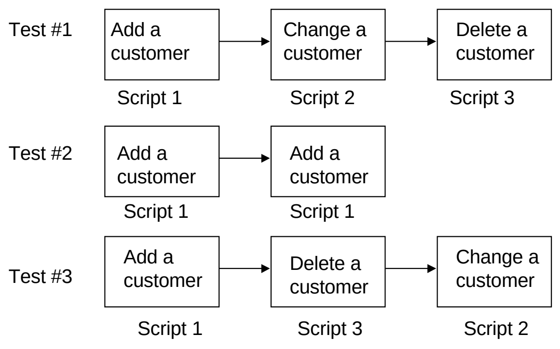
You will notice that the expected result accommodates two situations – one that accounts for a new employee and one that accounts for an existing one. This introduces complexity to the script – plus a need to handle the next action in a clean way. The better practice would be to break the script into two scripts – one for the new employee and one for the existing employee.

Sample Test Script

System: Payroll System **Performed By:** ____________________________ **Date:** ______________

Step #	Module ID	Action	Expected Result	Observed Result	Pass/ Fail	Defect #
1.	PY001	Type employee name and ID in employee entry screen.	If employee is on file, basic information is displayed correctly. If employee is not on file, a message is displayed "Employee not on file. Do you wish to add?" Type "Y" and press ENTER key. The employee is added and message is displayed "Employee successfully added."			
2.	PY001	Enter a pay rate effective in two pay periods.	Rate accepted.			
3.	PY005B	Run batch payroll job for the next pay period.	Payroll calculated correctly. The pay rate entered in step 2 is not applied.			

Test Script Modularity and Reusability



Script Modularity

By keeping scripts modular, you can combine them to construct many different test scenarios. This also reduces the overall number of scripts to maintain. In addition, if you are currently writing manual test scripts, modular scripts are much easier to convert to an automated tool format.

Script modularity requires that you start and stop scripts at a common point, such as a empty or blank entry screen.

EX: CUSTOMER ENTRY SCRIPTS

SCRIPT #1 - Enter a customer
SCRIPT #2 - Change a customer
SCRIPT #3 - Delete (Cancel) a customer

One test might be to enter, change, and then cancel a customer. Another test might be to enter a customer and then enter the same customer again. Both of these tests could be accomplished with just three scripts. There are many more possible combinations.