DEFECT TESTING

Defect testing involves finding errors in a computer program by running it. On each run you enter input and study the program’s output to determine if it is correct. If not, you debug the program by finding and fixing the errors. Each program run is called a test run and its input the test data.

Since the goal of each test run is to uncover another error, a successful test run is one that causes your program to fail.

A program that runs correctly for one test datum may not be correct when given a different test datum. Therefore, good testing requires many test runs.

Example
This code performs clock arithmetic – converting from west coast to east coast time by adding 3 hours onto the hour. The code works for all values of westCoast except for 9.

```c
// declare data
int westCoast; // hour of the day Pacific time
int eastCoast; // hour of the day Eastern time

// calculate east coast time
eastCoast = (westCoast + 3) % 12;
```

Defect testing techniques are primarily concerned with developing an exhaustive set of test cases. A test case is a written description of what you are trying to test and how to do it. It includes:

- The program feature or requirement that the run is testing.
- The test data used for the run.
- How a correct program would respond to the test data.
- How your program responded to the test data and whether or not it is correct.

The entire set of test cases is sometimes called the test suite. An exhaustive test suite should run the computer program through as many combinations of data and processing steps as is practically possible. Generally, you can’t test all possible combinations because there’s too many of them.
Regression Testing
Upon finding an error in your program, you correct it by editing the program. When you do this, it is possible that you introduce a new error where there wasn’t one before. To help prevent this, you must practice regression testing – meaning that you repeat all prior test runs after fixing an error in your program.

To do regression testing you must keep good diary of your tests. Work out a test suite in advance and, during testing, record:

- Calendar date of each test run.
- If the program response is not correct, what you did to fix the error and the calendar date that you did it.

Such a diary can be a script for regression testing since you can easily repeat the recorded test runs.

Black-Box versus White-Box Testing
Black-box testing is so named because it relies strictly on the program requirements to select test data. You look at the program as a “black box” whose behavior is fathomed by its inputs and outputs. White-box (or structural) testing relies on knowledge about the program’s logic and code. White-box testing tends to be done early in the testing process, especially for unit testing. Because it focuses on the program’s user requirements and purposely ignores program code, black-box testing is applied during the later stages of testing.
Path Testing

Path testing is a white-box technique whose goal is to run every independent execution path at some time during defect testing. The sequence of statements that the computer executes during a run is called an execution path. Two execution paths are independent if each executes at least one statement not executed by the other. By running every independent execution path, you guarantee that every statement in the program must have been executed at least once and each simple Boolean expression must have been executed on both its true and false paths.

Example
This code has 1 execution path.

```java
1  costOfGoods = quantity * 13.50;
```

Example
This code has 2 independent paths: (1, 2, 5) and (1, 3, 4, 5).

```java
1  if ( quantity > 10 )
2   costOfGoods = quantity * 13.50;
3  else
4   costOfGoods = quantity * 15.00;
5  totalCharge = costOfGoods + handlingCharge;
```

Example
This code has 2 independent paths: (1, 2, 3, 4, 5, 6, 12, 13) and (1, 2, 3, 4, 5, 6, 8, 9, 10, 6, 12, 13).

```java
1  Scanner scanner = new Scanner( "10 15 20 5 25" );
2  int count;
3  double item, sum;
4  count = 0;
5  sum = 0.0;
6  while ( scanner.hasNext( ) )
7  {
8    item = scanner.nextDouble( );
9    sum += item;
10   count++;
11  }
12  System.out.println( sum + "/" + count + ":= " + sum/count );
13  scanner.close( );
```
You can determine the number of independent execution paths by adding one to the number of simple Boolean expressions within the program. `&&` and `||` combine simple expressions. This metric is known as the *cyclomatic complexity*.

**Example**
This code has 3 independent paths.

```java
1 if ( quantity > 100 && frequentCustomer == 'y')
2    price = 18;
3  else
4    price = 20;
5  totalCharge = price*quantity;
```

Path testing alone cannot select sufficient test data.

**Example**
This code fragment, which has only one execution path, works for all input except the value 7.

```java
1 int hour, later;
2 System.out.println( "What hour is it?" );
3 hour = input.getInt( );
4 later = (hour + 5) % 12;
5 System.out.printf( "5 hrs. later is %d o'clock\n", later );
```
Modularization Simplifies Path Testing

Breaking a program into methods significantly reduces the number of test cases needed for path testing. Suppose you have two code fragments $F$ and $G$ where the number of execution paths through $F$ is $f$ and through $G$ is $g$. $F$ and $G$ implemented and tested as separate methods require $f + g$ test cases whereas in a single method they require $f \times g$ test cases.

**Example**

Path testing of method `three` requires 3 test cases; method two requires two test cases. The total number of tests is $3 + 2 = 5$. Path testing of method `combined`, where the code of three and two have been combined into one method, requires $3 \times 2 = 6$ test cases. The 6 independent paths in `combined` are:

- 4, 5, 11, 12
- 4, 5, 11, 13, 14
- 4, 6, 7, 11, 12
- 4, 6, 7, 11, 13, 14
- 4, 6, 8, 9, 11, 12
- 4, 6, 8, 9, 11, 13, 14

```cpp
1 void three( )
2 {
3     // following fragment has 3 execution paths
4     if ( . . . )
5         . . .
6     else if ( . . . )
7         . . .
8     else
9         . . .
10 }
```

```cpp
1 void two( )
2 {
3     // following fragment has 2 execution paths
4     if ( . . . )
5         . . .
6     else
7         . . .
8 }
```
```c
text

void combined()
{
    // following fragment has 3 execution paths
    if ( . . . )
        ...
    else if ( . . . )
        ...
    else
        ...
    // following fragment has 2 execution paths
    if ( . . . )
        ...
    else
        ...
}
```

Equivalence Partitioning

*Equivalence partitioning* is a black-box technique that involves identifying the various classes of data that must be handled by the program and selecting test data within each class. It is based on the idea that program data falls into different classes, such as negative versus positive numbers, strings with embedded blanks versus those without, etc. We expect a program to behave equivalently for all data in the same class, which we call an *equivalence partition*.

**Example**

A program inputs an employee’s gross pay and computes his or her withholding tax. The tax is 8% on the first $500 of pay, 16% on any pay over $500 up to $1000 and 24% on pay over $1000. Gross pay that is zero or negative is not valid.

<table>
<thead>
<tr>
<th>#</th>
<th>Equivalence Class</th>
<th>Expected Program Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Gross pay of zero or less</td>
<td>Error</td>
</tr>
<tr>
<td>2</td>
<td>Gross pay in range $1 to $500</td>
<td>Compute tax = 0.08*pay</td>
</tr>
<tr>
<td>3</td>
<td>Gross pay in range $501 to $1,000</td>
<td>Compute tax = 40 + 0.16 *(pay over 500)</td>
</tr>
<tr>
<td>4</td>
<td>Gross pay over $1,000</td>
<td>Compute tax = 120 + 0.24 *(pay over 1000)</td>
</tr>
</tbody>
</table>
**Example**
A program calculates the two real-valued roots of the quadratic equation $ax^2 + bx + c$ by using the quadratic formula $-b \pm \sqrt{b^2 - 4ac} \over 2a$.

<table>
<thead>
<tr>
<th>#</th>
<th>Equivalence Class</th>
<th>Expected Program Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Values of $a$, $b$ and $c$ for which two real-valued roots exist</td>
<td>Calculate both real roots</td>
</tr>
<tr>
<td>2</td>
<td>Values of $a$, $b$ and $c$ for which only one real-valued root exists</td>
<td>Calculate the one real root</td>
</tr>
<tr>
<td>3</td>
<td>Values of $a$, $b$ and $c$ for which no real-valued roots exist</td>
<td>Error</td>
</tr>
</tbody>
</table>
Example
A program determines if an applicant qualifies for a loan. To qualify, applicants must have more than $20,000 either in savings or as equity in their home. In addition, they must have over $20,000 annual income.

<table>
<thead>
<tr>
<th>#</th>
<th>Income</th>
<th>Equity</th>
<th>Savings</th>
<th>Expected Program Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>≤ $20,000</td>
<td>≤ $20,000</td>
<td>≤ $20,000</td>
<td>Reject</td>
</tr>
<tr>
<td>2</td>
<td>≤ $20,000</td>
<td>≤ $20,000</td>
<td>&gt; $20,000</td>
<td>Reject</td>
</tr>
<tr>
<td>3</td>
<td>≤ $20,000</td>
<td>&gt; $20,000</td>
<td>≤ $20,000</td>
<td>Reject</td>
</tr>
<tr>
<td>4</td>
<td>≤ $20,000</td>
<td>&gt; $20,000</td>
<td>&gt; $20,000</td>
<td>Reject</td>
</tr>
<tr>
<td>5</td>
<td>&gt; $20,000</td>
<td>≤ $20,000</td>
<td>≤ $20,000</td>
<td>Reject</td>
</tr>
<tr>
<td>6</td>
<td>&gt; $20,000</td>
<td>≤ $20,000</td>
<td>&gt; $20,000</td>
<td>Accept</td>
</tr>
<tr>
<td>7</td>
<td>&gt; $20,000</td>
<td>&gt; $20,000</td>
<td>≤ $20,000</td>
<td>Accept</td>
</tr>
<tr>
<td>8</td>
<td>&gt; $20,000</td>
<td>&gt; $20,000</td>
<td>&gt; $20,000</td>
<td>Accept</td>
</tr>
</tbody>
</table>
**Boundary Value Analysis**

*Boundary value analysis* is a black-box technique that involves choosing test data that lie on both sides of the boundaries between equivalence partitions. It is based on the observation that programmers often make errors at the boundary between partitions.

**Example**

This code fragment incorrectly determines if the user is of legal age. It works for all ages except on the boundary of 18.

```java
if ( age > 18 )
    legalAge = true;
else
    legalAge = false;
```

**Example**

A program inputs an employee’s gross pay and computes his or her withholding tax. The tax is 8% on the first $500 of pay, 16% on any pay over $500 up to $1000 and 24% on pay over $1000. Gross pay that is zero or negative is not valid.

<table>
<thead>
<tr>
<th>#</th>
<th>Equivalence Class</th>
<th>Test Data</th>
<th>Expected Program Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Gross pay of zero or less</td>
<td>0</td>
<td>Error</td>
</tr>
<tr>
<td>2</td>
<td>Gross pay in range $1 to $500</td>
<td>1</td>
<td>tax = 0.08</td>
</tr>
<tr>
<td></td>
<td></td>
<td>500</td>
<td>tax = 40.00</td>
</tr>
<tr>
<td>3</td>
<td>Gross pay in range $501 to $1,000</td>
<td>501</td>
<td>tax = 40.16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1000</td>
<td>tax = 120.00</td>
</tr>
<tr>
<td>4</td>
<td>Gross pay over $1,000</td>
<td>1001</td>
<td>tax = 120.24</td>
</tr>
</tbody>
</table>
**Transaction Flow Testing**  

*Transaction flow testing* is a black-box technique that studies each step and decision point in a transaction carried out by the program so that every possible sequence of steps through the transaction is tested.

**Example**  
This diagram shows the steps to withdraw money from an ATM. From this diagram, you would develop test cases that execute four transaction paths:

<table>
<thead>
<tr>
<th>#</th>
<th>Transaction Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Welcome, Cancel</td>
</tr>
<tr>
<td>2</td>
<td>Welcome, PIN Request, Cancel</td>
</tr>
<tr>
<td>3</td>
<td>Welcome, PIN Request, Amount Request, Accept ATM Charge, Cancel</td>
</tr>
<tr>
<td>4</td>
<td>Welcome, PIN Request, Amount Request, Accept ATM Charge, Dispense Money &amp; Receipt</td>
</tr>
</tbody>
</table>

![Diagram of ATM transaction flow](chart.png)
**Expert Observations**

Here are some observations about defect testing made by software engineering experts.

*Testing typical situations is more important than testing boundary value cases.*

---

Nathan Petschenik

---

*Black-box testing is more effective in discovering errors than white-box testing.*

---

Victor Basili

---


Exercises

1. State the cyclomatic complexity of this code fragment list test data that execute every independent execution path through the fragment.

```java
int c = 0;
while ( c < n )
{
    c++;
    System.out.println( "c = " + c );
}
```

2. State the cyclomatic complexity of this code fragment list test data that execute every independent execution path through the fragment.

```java
System.out.print( "Enter temperature and scale (e.g. 75 F): " );
temp = scanner.nextDouble( );
scale = scanner.next( );
if ( scale.equals( "F" ) )  // Fahrenheit
{
    // convert Fahrenheit to Celsius
    f = temp;
    c = (5.0 / 9.0) * (f - 32);
}
else // Celsius
{
    // convert Celsius to Fahrenheit
    c = temp;
    f = 1.8 * c + 32;
}
System.out.println( f + "\u00B0F = " + c + "\u00B0C" );
```
3. State the cyclomatic complexity of this code fragment list test data that execute every independent execution path through the fragment.

```java
System.out.print( "Enter a number: " );
x = scanner.nextDouble();
c = 0;
do {
x /= 2.0;
c++;
System.out.println( c + ": " + x );
} while ( x > 1.0 );
System.out.println( "Done" );
```

4. State the cyclomatic complexity of this code fragment list test data that execute every independent execution path through the fragment.

```java
System.out.print( "You " );
if ( age >= 21 )
{
    System.out.print( "can " );
}
else
{
    System.out.print( "can't " );
}
System.out.println( "purchase liquor!" );
```
5. State the cyclomatic complexity of this code fragment list test data that execute every independent execution path through the fragment.

```java
import java.util.Scanner;

public class TimesDrill {

    public static void main(String args[]) {
        int a, b;  // times operands
        int ansR;  // right answer
        int ansU;  // user's answer
        int score; // user's score
        String s;  // output string

        Scanner scanner = new Scanner(System.in);
        System.out.println("Practice multiplication");
        System.out.println("To quit, enter -1");
        score = 0; // initialize score

        do
            {
                // generate question
                a = (int)(Math.random()*11);
                b = (int)(Math.random()*11);
                s = a + " X " + b + " = ";
                System.out.print(s + "? ");
                ansR = a * b; // compute right answer
                ansU = scanner.nextInt(); // get user's answer
                if (ansU == -1) // user wants to quit
                    {
                        System.out.println("Goodbye");
                        break; // quit
                    }
                else if (ansU != ansR) // user answered wrong
                    {
                        System.out.print("Sorry, " + s + ansR);
                        break; // quit
                    }
                // user entered a correct answer so add one to
                // score and encourage continuing
                score++;
                System.out.println(score+" right. Keep going!");
            } while (true);
    }
}
```
6. State the cyclomatic complexity of this code fragment list test data that execute every independent execution path through the fragment.

```java
int totalDays = 0;
switch ( month )
{
    case 12:
        totalDays += 31;
    case 11:
        totalDays += 30;
    case 10:
        totalDays += 31;
    case 9:
        totalDays += 30;
    case 8:
        totalDays += 31;
    case 7:
        totalDays += 31;
    case 6:
        totalDays += 30;
    case 5:
        totalDays += 31;
    case 4:
        totalDays += 30;
    case 3:
        totalDays += 31;
    case 2:
        totalDays += 28;
    case 1:
        totalDays += 31;
}
System.out.println( "days = " + totalDays );
```
7. State the cyclomatic complexity of this code fragment list test data that execute every independent execution path through the fragment.

```
1 switch ( grade )
2 {
3     case "A+" :
4         System.out.println("Highest honors");
5         break;
6     case "A" :
7     case "A-":
8         System.out.println("Honors");
9         break;
10    case "B+":
11    case "B" :
12        System.out.println("Favorable Mention");
13        break;
14 }
15 . . .
```

8. State the cyclomatic complexity of this code fragment list test data that execute every independent execution path through the fragment.

```
1 switch ( month )
2 {
3     case 9:
4     case 4:
5     case 6:
6     case 11:
7         lastDate = 30;
8         break;
9     case 2:
10        if ( new GregorianCalendar().isLeapYear( year ) )
11            lastDate = 29;
12        else
13            lastDate = 28;
14        break;
15    default:
16        lastDate = 31;
17        break;
18 }
19 System.out.println("last date = " + lastDate);
```
For each of the following specifications, (a) state the equivalence classes, (b) give the boundary values and (c) list a set of test data based on a and b.

<table>
<thead>
<tr>
<th>Specification</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.</td>
<td>The program is supposed to impose a 2% handling charge and 5% shipping charge on sales less than $100. Sales of $100 or more have no extra charges.</td>
</tr>
<tr>
<td>10.</td>
<td>The program is supposed to calculate a salesperson's bonus. His or her employer gives a 5% bonus on the amount of sales that exceed $100,000.</td>
</tr>
<tr>
<td>11.</td>
<td>A mail-order catalog sells cargo pants for $27 a pair or two pairs for $50. For example, it charges $27 for one pair, $50 for two, $77 for three (i.e. $50 plus $27), $100 for four (i.e. $50 twice). The program is supposed to inputs the number of pants sold and prints the total charge.</td>
</tr>
<tr>
<td>12.</td>
<td>The program is supposed to calculate an employee’s pay. He or she earns $9 per hour for up to 40 hours of work and $13.50 per hour for hours over 40. For example, an employee working 50 hours receives 40·9 + 10·13.50 = $495.</td>
</tr>
<tr>
<td>13.</td>
<td>The program is supposed to calculate a family’s season membership fee to the municipal swimming pool. A season membership costs $90 for a family of 4. Additional family members can be added for a cost of $30 per member. For example, a family of 6 pays 90 + 2×30 = $150.</td>
</tr>
<tr>
<td>14.</td>
<td>The program is supposed to determine if a farmer can apply pesticide to a field. It’s OK to apply pesticide if the temperature is at least 60 degrees, the relative humidity is between 15 and 35 percent (inclusive) and the wind speed is at most 10 miles per hour.</td>
</tr>
<tr>
<td>15.</td>
<td>The program controls the automatic snow gun – a device that makes artificial snow when there’s not enough natural snow – at a ski resort. The gun makes snow when the temperature is from 30°F down to 20°F with a relative humidity of less than 30, or a temperature less than 20°F.</td>
</tr>
<tr>
<td>16.</td>
<td>The program controls the cash register at Pistol Pete’s all-you-can-eat buffet restaurant. Charges are based on the customer’s age and, if the customer is a child, his or her weight. Here’s the schedule of charges:</td>
</tr>
<tr>
<td></td>
<td>Age</td>
</tr>
<tr>
<td></td>
<td>under 6</td>
</tr>
<tr>
<td></td>
<td>6 to 15</td>
</tr>
<tr>
<td></td>
<td>16-59</td>
</tr>
<tr>
<td></td>
<td>60 and over</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>-----------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>17.</td>
<td>A credit company program is supposed to evaluate a loan application using the applicant’s income, assets and debt. To qualify for credit, this rule must apply: an applicant must have income at least $25,000 and, if debt exceeds $50,000 then assets must be at least $100,000.</td>
</tr>
<tr>
<td>18.</td>
<td>The program is supposed to identify each college student for whom the professor must complete a midterm grade report. Such a student must be a freshman (less than 30 credits) or a non-freshman whose GPA is between 1.8 and 2.5, inclusive.</td>
</tr>
</tbody>
</table>