Topic: 4.4.2 Testing

Like all human creations, computer programs are often less than perfect. Computer code may contain various types of bugs (errors).

Errors

Errors can be syntax errors, semantic errors, or logic errors.

The most obvious type of error is the syntax error, which occurs when you write code in a manner not allowed by the rules of the language. Syntax errors are almost always caught by the compiler or interpreter, which displays an error message informing you of the problem. In Visual Studio, these error messages appear in the Output window. These messages tell you the location of a syntax error (line number and file) and a short description of the problem. Finding the cause of a syntax error is normally a straightforward process once you learn to use and understand these descriptions.

Semantic errors are a more subtle type of error. A semantic error occurs when the syntax of your code is correct, but the semantics or meaning are not what you intended. Because the construction obeys the language rules, semantic errors are not caught by the compiler or interpreter. Compilers and interpreters concern themselves only with the structure of the code you write, not the meaning. A semantic error can cause your program to terminate abnormally, with or without an error message. In colloquial terms, it can cause your program to crash or hang.

Not all semantic errors manifest themselves in such an obvious fashion, however. A program can continue running after some semantic errors, but the internal state of the program will not be what you intended. Variables may not contain the correct data, or the program may continue down a path that is not what you intended. The eventual result will be incorrect output. These errors are called logic errors, because while the program does not crash, the logic that it executes is in error.

Testing

The only way to detect logic errors is by testing your program, manually or automatically, and verifying that the output is what you expected. Testing should be an integral part of your software development process. Unfortunately, while testing can show you that the output of your program is incorrect, it usually leaves you without a clue as to what part of your code actually caused the problem. This is where debugging comes in.

Once we have created our solution we need to test that the whole system functions effectively. To do this should be easy, as all we need to do is compare the finished product next to the objectives that we set out in the Analysis. There are several ways of testing a system; you need to know them all and the types of data that might be used.

Topic: 4.4.2 Testing

Objectives of Testing
Testing is essential because of:

- Software reliability
- Software quality
- System assurance
- Optimum performance and capacity utilization
- Price of non-conformance

Test Plan

When a system is designed it is important that some consideration is given to making sure that no mistakes have been made. A schedule should be drawn up which contains a test for every type of input that could be made and methods of testing that the program actually does what it was meant to do. This schedule is known as the test plan. Note that it is produced before the system is produced.

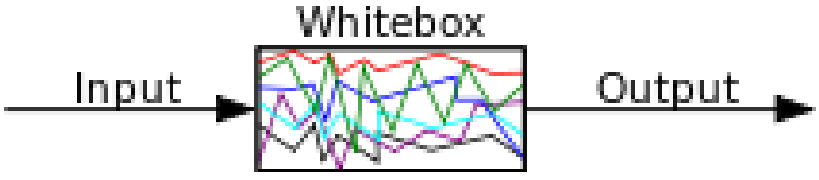
There are a number of ways of testing a program.

Black Box Testing

Black Box testing model

Consider the box to contain the program source code, you don't have access to it and you don't have to be aware of how it works. All you do is input data and test to see if the output is as expected. The internal workings are unknown; they are in a black box. Examples of Black Box testing would be if you were working as a games tester for a new console game. You wouldn't have been involved in the design or coding of the system, and all you will be asked to do is to input commands to see if the desired results are output.

White Box Testing



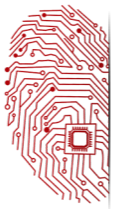
White Box testing model showing various routes through the code being put to test

With white box testing you understand the coding structure that makes up the program. All the tests that you perform will exercise the different routes through the program, checking to see that the correct results are output.

Topic: 4.4.2 Testing

White Box and Black Box testing techniques

White Box Testing	Black Box Testing
Complete Path Testing	Equivalence Partitioning
Branch or Decision	Boundary Value Analysis
Condition Testing	Cause Effect Graphing
Data Flow Testing	Syntax Testing
Loop Testing	



Topic: 4.4.2 Testing

Dry run testing

A dry run is a mental run of a computer program, where the computer programmer examines the source code one step at a time and determines what it will do when run. In theoretical computer science, a dry run is a mental run of an algorithm, sometimes expressed in pseudocode, where the computer scientist examines the algorithm's procedures one step at a time. In both uses, the dry run is frequently assisted by a trace table. And whilst we are here we might as well get some more practice in:

Exercise: Dry run testing

Complete the trace table for the following code:

```
var a <- 5
var b <- 4
var count <- 0
while count < 4
  a <- a + a
  count <- count + 1
end while
```

a	b	count
5	4	

Answer:

a	b	count
5	4	0
10	4	1
20	4	2
40	4	3
80	4	4

Complete the trace table for the following code:

```
array nums() <- {4,2,5,2,1}
var z <- 4
var high <- nums(0)
while z > 0
  if high < nums(z)
    high <- nums(z)
  end if
  z <- z - 1
end while
```

z	high
4	4

Answer:

z	high
4	4
3	4
2	4
1	5
0	5

Topic: 4.4.2 Testing

What is Walkthrough in software testing?

- It is not a formal process/review
- It is led by the authors
- Author guide the participants through the document according to his or her thought process to achieve a common understanding and to gather feedback.
- Useful for the people if they are not from the software discipline, who are not used to or cannot easily understand software development process.
- Is especially useful for higher level documents like requirement specification, etc.

The goals of a walkthrough:

- To present the documents both within and outside the software discipline in order to gather the information regarding the topic under documentation.
- To explain or do the knowledge transfer and evaluate the contents of the document
- To achieve a common understanding and to gather feedback.
- To examine and discuss the validity of the proposed solutions

Integration Testing

Integration testing is a logical extension of unit testing. In its simplest form, two units that have already been tested are combined into a component and the interface between them is tested. A component, in this sense, refers to an integrated aggregate of more than one unit. In a realistic scenario, many units are combined into components, which are in turn aggregated into even larger parts of the program. The idea is to test combinations of pieces and eventually expand the process to test your modules with those of other groups. Eventually all the modules making up a process are tested together. Beyond that, if the program is composed of more than one process, they should be tested in pairs rather than all at once.

Integration testing identifies problems that occur when units are combined. By using a test plan that requires you to test each unit and ensure the viability of each before combining units, you know that any errors discovered when combining units are likely related to the interface between units. This method reduces the number of possibilities to a far simpler level of analysis.

You can do integration testing in a variety of ways but the following are three common strategies:

- The top-down approach to integration testing requires the highest-level modules be test and integrated first. This allows high-level logic and data flow to be tested early in the process and it tends to minimize the need for drivers. However, the need for stubs complicates test management and low-level utilities are tested relatively late in the development cycle. Another disadvantage of top-down integration testing is its poor support for early release of limited functionality.

Topic: 4.4.2 Testing

- The bottom-up approach requires the lowest-level units be tested and integrated first. These units are frequently referred to as utility modules. By using this approach, utility modules are tested early in the development process and the need for stubs is minimized. The downside, however, is that the need for drivers complicates test management and high-level logic and data flow are tested late. Like the top-down approach, the bottom-up approach also provides poor support for early release of limited functionality.

- The third approach, sometimes referred to as the umbrella approach, requires testing along functional data and control-flow paths. First, the inputs for functions are integrated in the bottom-up pattern discussed above. The outputs for each function are then integrated in the top-down manner. The primary advantage of this approach is the degree of support for early release of limited functionality. It also helps minimize the need for stubs and drivers. The potential weaknesses of this approach are significant, however, in that it can be less systematic than the other two approaches, leading to the need for more regression testing.

Alpha and Beta testing. When you have written a program and you sit down to test it, you have a certain advantage because you know what to expect. After all, you wrote the program. This can be extended to the whole software company, as the employees are all computer-minded people. Testing carried out by people like this is known as alpha testing. Eventually, the company will want ordinary users to test the program because they are likely to find errors that the software specialists did not find. Testing carried out by the users of the program is called beta testing. If you continue the course next year, in order to turn your AS grade into an A level, you will have to write a project. This involves using a computer to solve a problem for someone. When you have finished your project you will be expected to test whether or not it works, this is alpha testing. You will also get marks if you persuade the person whose problem you are solving to test it for you, this is the beta testing.

User Acceptance Testing is often the final step before rolling out the application. Usually the end users who will be using the applications test the application before 'accepting' the application. This type of testing gives the end users the confidence that the application being delivered to them meets their requirements. This testing also helps nail bugs related to usability of the application.

Topic: 4.4.2 Testing

Selection of test data

Normal, Abnormal, Extreme

There are three types of test data we can use. What are they? The answer lies mostly in their name; let's take a look at this example where someone has created a secondary school registration system which lets students register themselves. We don't want people who are too young attending, and we don't want students who are too old. In fact we are looking for students between 11 and 16 years old.

(A Normal Student will be 12, 13, 14 or 15)

(An Abnormal (or wrong) aged student will be 45, 6 or any age outside those allowed.)

(An Extreme (or boundary) aged student has just started or is just about to leave, they will be 11 or 16)

If you are testing data that has Normal, Abnormal and Extreme data, it is best to show tests for all three. Some tests might only have Normal and abnormal, for example entering a correct password might only have a Normal and an abnormal value. Some things might only have Normal testing, such as if a button to a next page works or not, or if a calculation is correct.

Example: Electronic Crafts Test Data

Imagine that the following objectives were set:

The maximum number of yellow cards a player can receive in a game is 2
Normal : 0,1,2 (most likely to list 1)
Abnormal: 3, -6
Extreme: 0,2

There should be no more than 15 minutes extra time in a game
Normal : 0,1m45,9m23
Abnormal: -6, 15m01
Extreme: 0,15m00

The name of a team should be no longer than 20 characters
Normal : Monster United
Abnormal: Monster and Dagrington League of Gentlefolk
Extreme: Moincicestier United (20 characters!)

Topic: 4.4.2 Testing

Exercise: Test Data

- List the normal, abnormal and extreme data for the following:

  The number of cigarettes currently in a 20 cigarette carton:

Answer:

Normal: 5, 16,18
Abnormal: 21, -7
Extreme: 0 or 20

- *The username for a system that must be of the form "<letter><letter><number><number>":*

Answer:

Normal: GH24
Abnormal: G678
Extreme: AA00, ZZ99

- *The age of a college teacher:*

Answer:

Normal: 28, 56, 32
Abnormal: 16, 86
Extreme: 21, 68

- *The date for someone's birthday:*

Answer :
Normal : 12/07/1987
Abnormal: 31/09/1987, 31/02/1987
Extreme: 31/12/1999, 01/01/2001
(this is harder to specify, it might depend on the age range you allow)

Topic: 4.4.2 Testing

- *Someone's hair color:*

Answer:

Normal: brown, red, black
Abnormal: bicycle
Extreme: N/A (we're not here to judge people!)

- *Does the following calculation work: 14 * 2*

Answer :

Normal: 28
Abnormal: N/A
Extreme: N/A

- *Number of pages in a book*

Answer:

Normal: 24, 500
Abnormal: -9
Extreme: 1
(notice no upper end; this is assuming that it wouldn't be a book without any pages!)