

maXbox



maXbox Starter 36

Start a Function Test V2

1.1 Measure Correctness and Stability

Today we step through the realm of testing and bug-finding and his visual representation in metrics and audits.

The traditional approach to quality control and testing has been to apply it after the development process completes. This approach is very limited as uncovering defects or errors at a late stage can produce long delays with high costs while the bugs are corrected, or can result in the publishing of a low-quality software product.

Lets start with a first function and how to improve and test it:

```
function CountPos(const subtxt: string; Text: string): Integer;  
begin  
  if (Length(subtxt)= 0) or (Length(Text)= 0) or (Pos(subtxt,Text)= 0) then  
    result:= 0  
  else  
    result:= (Length(Text)- Length(StringReplace(Text,subtxt,"  
    [rfReplaceAll]))) div Length(subtxt);  
end;
```

This function counts all sub-strings in a text or string. The point is that the function uses another functions like Pos() and StringReplace(). Do we have to test it also? No, but we need a reference value:

```
PrintF('CountPos: %d',[CountPos('max','this is max of maXbox a max numbermax')])
```

The result is 3, so we better extract our test-string in a constant:

```
Const TESTTEXT = 'this is max of maXbox a max numbermax';
```

```
PrintF('CountPos: %d',[CountPos('max',TESTTEXT)])
```

And with `Assert` we do automate that reference value (more on p. 5):
with procedure `Assert(expr: Boolean; const msg: string);` you set
an expression value which you expect to be true:

```
Assert (CountPos ('max', TESTTEXT)=3, 'count pos assert')
```



Find the position and count of all strings in another is the function to
test but is the function case sensitive or not?

Sometimes it is helpful to track your program's execution as your program
runs. Or maybe you want to see a variable's value without stopping
program execution at a breakpoint to test.

The `OutputDebugString()` function enables you to do exactly that. This
function is a convenient debugging tool that many programmers overlook,
primarily because of a general lack of discussion on the subject.

```
OutputDebugString (inttoStr (CountPos ('max', TESTTEXT)) +  
                    'CountPos runs..');
```

That's all you have to do. Because Delphi, Lazarus or Free Pascal is
installed as the system debugger, any strings sent using `OutputDebugString`
will show up in the Event Log. You can place calls to `OutputDebugString`
anywhere you want in your code.

To view the value of a variable, you must format a string and send that
string to `OutputDebugString`, for example:

```
procedure GetOutputDebugString;  
var  
  x : Integer;  
  S : string;  
begin  
  { Some code here...}  
  x:= CountPos('max',TESTTEXT);  
  S:= Format('X := %d', [X]);  
  OutputDebugString(PChar(S));  
  writeln(S)  
end;  
//OutputDebugString(PChar(Format('[%s][%s] %s',[aCaption, GetFormatDT(StartDT), aText])));
```

Also the Event Viewer is a Microsoft Management Console (MMC) snap-in
that enables you to browse and manage event logs. It is an indispensable
tool for monitoring the health of systems and troubleshooting issues when
they arise. Look at the last entry in the Event Log shown in the next
picture. That entry was generated using this code:

```
WriteToOSEventLog ('mX3', 'maXbox', 'Details3', TESTTEXT);
```

When you use Event Viewer to troubleshoot or track a problem, you need to locate events related to the problem, regardless of which event log they appear in. Event Viewer enables you to filter for specific events across multiple logs as of maXbox. We need values in

```
WriteToOSEventLog(const logName, logCaption,  
                  logDetails: String; const logRawData: Str);
```

a logName, logCaption, logDetails and some RawData:

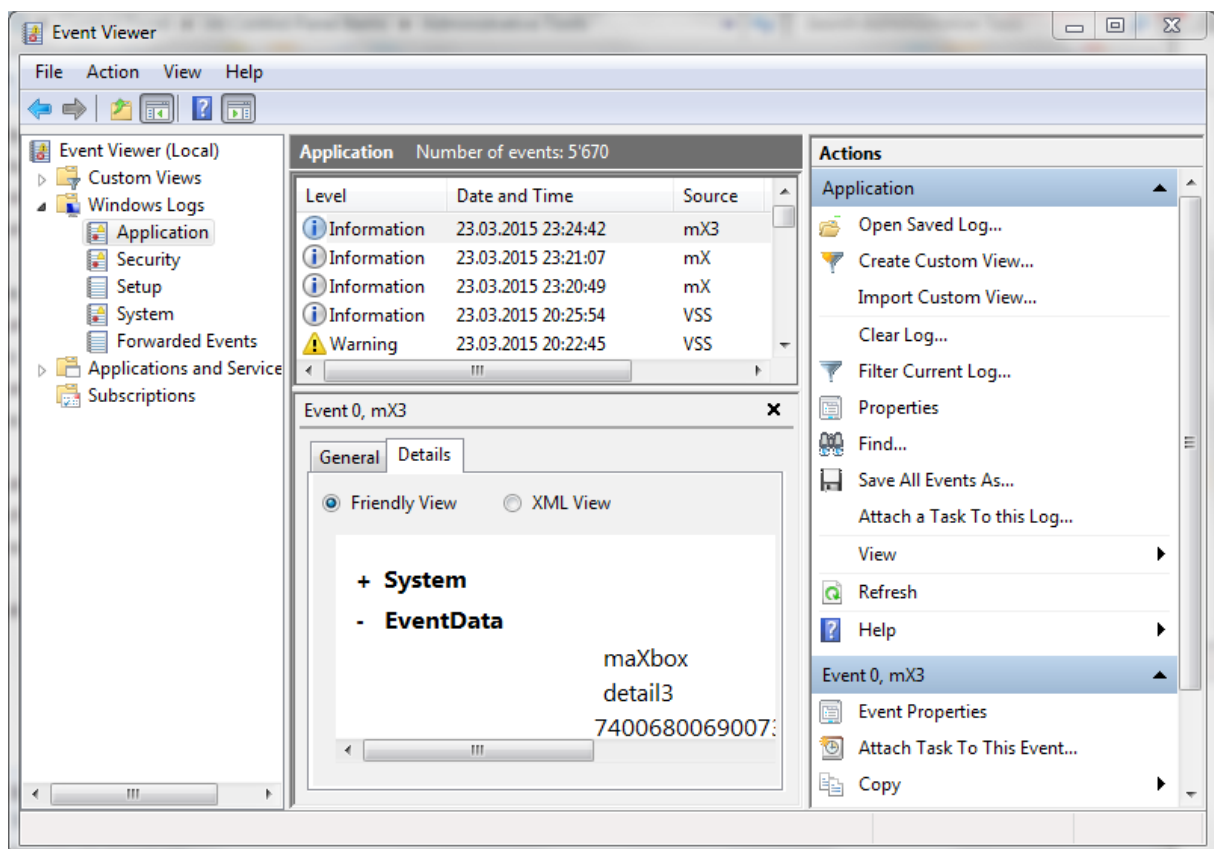
```
WriteToOSEventLog('mX3','maXbox','ID45:detail3',TESTTEXT);
```

The Application log (see picture) contains events logged by applications or programs. For example, a database program might record a file error in the application log. You as a developer decide which events to log.

To log a sound in a multimedia app for e.g. you have to set a song first:

```
playMP3(mp3path);  
closeMP3;  
DebugOutputString('closeMP3: '+ mp3path);
```

In a shortlink `%windir%\system32\eventvwr.msc /s` you start a log:



👉 Analytic or test events are published in high volume. They describe program operation and indicate problems that can't be handled by user tasks or intervention.

Lets go back to our function to test and build a test function too.

```
function CountPos(const subtxt: string; Text: string): Integer;
begin
  if (Length(subtxt)= 0) or (Length(Text)= 0) or (Pos(subtxt,Text)= 0) then
    result:= 0
  else
    result:= (Length(Text)- Length(StringReplace(Text,subtxt,"",
      [rfReplaceAll]))) div Length(subtxt);
end;
```

Such a small function like above contains another 3 functions (yellow) and before testing we should also check the understanding and the design of the function. For example `StringReplace()` which replace one or more substrings found within a string does have more parameters to test like the `[rfReplaceAll]`.

On the other side we can trust in a stable function of the runtime library otherwise we loose time not to mention the frustration caused by a bug that is difficult to find.

I asked a few lines above if the function to test is case-sensitive or not, do you remember? Yes it is and we do have to consider that in our test-string and test-function with `Uppercase` too.

-

```
function CountPosTest: boolean;
begin
  result:= false;
  if CountPos(Uppercase('max'),Uppercase(TESTTEXT))=4 then
    result:= true;
end;
```

Each time we test (or change) our function we call the test-function with a test-string and check the reference value which must be 4.

Normally you test a function by creating a test application or unit and if something goes wrong or crashes you will run the test application under the debugger and watching the values.

A debugger helps you find bugs or errors in your program at runtime. But the debugging process isn't just for finding and fixing bugs - it's a development and test tool as well.

Another great way to test a function is to compare the result with a already well known standard function, for example `CountStr()`:

```
Function CountStr(const ASearchFor, ASearchIn: string): Integer;
or check with a subroutine which both of them uses:
Function Unit SysUtils
StringReplace: Replace one or more substrings found within a string
```

```

function CountStr(const ASearchFor, ASearchIn: string): Integer;
var
    Start: Integer;
begin
    Result:= 0;
    Start:= Pos(ASearchFor, ASearchIn);
    while Start > 0 do begin
        Inc(Result);
        Start:= PosEx(ASearchFor, ASearchIn, Start+ 1);
    end;
end;

```

Or you will look at the web and ask to a search machine: How do I find and count the total amount of times that a particular word appears?¹

Often, as programmers gain experience in building applications, they develop a **sixth sense** for locating the cause of in-correctness or access violations.

With asserts you can support that feeling what could crash. An **assert** function is used to make sure that certain conditions which are assumed to be true are never violated. Assert provides an opportunity to intercept an unexpected condition and halt a program rather than allow execution to continue under unanticipated conditions.



If you encounter problems using any feature of Event Viewer, first ensure that the Windows Event Log service is running. If you are working with event subscriptions, ensure that the Windows Event Collector service is running.

With this call in maXbox you can run the event log from the script:

```
ExecuteCommand('cmd','/c %windir%\system32\eventvwr.msc /s')
```

Next we go to the topic of Unit Testing but not to generate units (this is another story) but consuming as a test-function or protocol e.g. in maXbox I do have some unit tests to check conditions before running a script!:

```
// test routines with assert or assert2 builds
```

```

SelftestPEM;
SelfTestCFundamentUtils;
SelfTestCFileUtils;
SelfTestCDateTime;
SelfTestCTimer;
SelfTestCRandom;

```

```

Writeln(' 6 Units Tested with success ')
ShowMessageBig(' 6 Units Tests with success!');

```

¹ <http://stackoverflow.com/questions/7283531/find-and-count-words-in-a-string-in-delphi>

A unit is the smallest testable part of software. It usually has one or a few inputs and usually a single output. In procedural programming a unit may be an individual program, function, routine, procedure, etc. In object-oriented programming, the smallest unit is a method, which may belong to a base/ super class, abstract class or derived/ child class².

Lets say you have a program comprising of two units and the only test you perform is system testing between Win and Linux:

SelfTestCFileUtils;

SelfTestCDateTime; as an extract in test-cases:

```
Assert2(FilePath('C', '.\X\Y', 'A\B', '\') = 'A\X\Y\C', 'FilePath');
Assert2(PathCanonical('A\B\..\C\D\..\..\..\.', '\') = '\', 'PathCanonical');
Assert2(UnixPathToWinPath('/c/d.f') = 'c\d.f', 'UnixPathToWinPath');
Assert2(WinPathToUnixPath('c\d.f') = '/c/d.f', 'WinPathToUnixPath');
A:= EncodeDateTime(2001, 09, 02, 12, 11, 10, 0);
Assert2(Month(A) = 9, 'EncodeDateTime');
S:= GMTTimeToRFC1123TimeA(A, True);
Assert2(S = '12:11:10 GMT', 'GMT');
S:= GMTDateTimeToRFC1123DateTimeA(A, True);
Assert2(S = 'Sun, 02 Sep 2001 12:11:10 GMT', 'GMTDateTimeToRFC1123DateTime');
Assert2(TickDeltaW($FFFFFFF6, 0) = 10, 'TrickDelta');
Assert2(CPUClockFrequency > 0, 'RTC Prepare');
Assert2(Length(RandomPassword(0, 0, True, True, True)) = 0, 'random passwd');
Assert2(Length(RandomPassword(1, 1, True, True, True)) = 1, 'random passwd');
CompareDateTime(DateOf(Now), DateOf(Now)+1)
```



For me the important thing is to isolate the development environment from the test environment and make sure you are using a version control system to keep track of your test scripts and constants.

```
V1.2.1 SelftestPEM;
V1.7.0 SelfTestCFundamentUtils;
V1.2.0 SelfTestCFileUtils;
V1.1.1 SelfTestCDateTime;
V1.2.5 SelfTestCTimer;
V0.9.2 SelfTestCRandom;
```

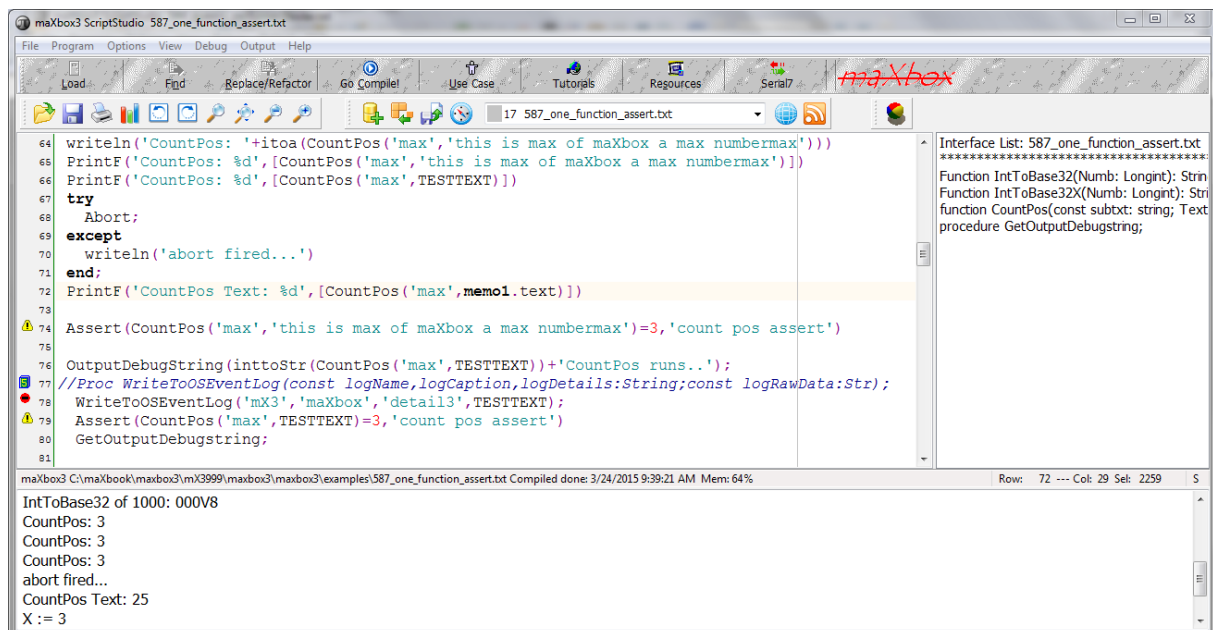
```
SingleCompareDelta = 1.0E-34;
DoubleCompareDelta = 1.0E-280;
{$IFDEF CLR}
ExtendedCompareDelta = DoubleCompareDelta;
{$ELSE}
ExtendedCompareDelta = 1.0E-4400;
{$ENDIF}
```

² <http://softwaretestingfundamentals.com/unit-testing/>

*// Default CompareDelta is set to SingleCompareDelta. This allows any type
 // of floating-point value to be compared with any other.
 // DefaultCompareDelta = SingleCompareDelta;*

Const

```
Bytes1KB = 1024;
Bytes1MB = 1024 * Bytes1KB;
Bytes1GB = 1024 * Bytes1MB;
Bytes64KB = 64 * Bytes1KB;
Bytes64MB = 64 * Bytes1MB;
Bytes2GB = 2 * LongWord(Bytes1GB);
```



You see the screen with those test scripts notion and some calls with external dependencies.

One of a question often arises is the following:

Why do I continue getting error messages even after I have written an exception handler?

Answer: In its default settings, the Delphi IDE notifies you whenever an exception occurs in your program.

What's important to realize is that at that point, none of your program's exception-handling code has run yet. It's all Delphi itself; its special status as a debugger allows it to get first notification of any exception in your program, even before your program knows about it.

Simply put, debugging when using exception handling can be a bit of a pain. Each runtime an exception is raised, the debugger pauses program execution at the except block just as if a breakpoint were set on that line. If the except block is in your code, the execution point is displayed as it would be if you had stopped at a breakpoint.

You can or must restart the program again by clicking the Run button, or you can step through your code.



You might not be able to continue execution of your program after an exception is raised. Whether you can continue debugging depends on what went wrong in your program.

If an exception passes through the tryblocks in the application code, the application automatically calls the `HandleException` method. Unless the exception object is `EAbort`, `HandleException` calls the `OnException` handler, if one exists. Otherwise, it calls `ShowException` to display a message box indicating an error occurred.

1.1.1 Performance Testing

A bit weird but also important is performance and tuning. Sometimes you will need to call a function or message handler for a particular measure of the time consuming:

```
with TStopWatch.Create() do begin
  try
    Start;
    //TimeOutThisFunction() to test performance
    Sleep(2);    //2ms = 0.002 secs
    Stop;
  finally
    Writeln('watchTime: ' + floatToStr(GetValueMSec/1000));
    Writeln('watchTime: ' + GetTimeString);
    Free; //TStopWatch Class
  end;
```

Time has long been a major subject of philosophy, art, poetry and science. Many fields use an operational definition in which the units of time are defined. Scholars disagree on whether time itself can be measured or is itself part of the measuring system³.

Finally, if a high-resolution performance counter exists on the system, you can use the `QueryPerformanceFrequency` Win API function to express the frequency, in counts per second. The value of the count is processor dependent!

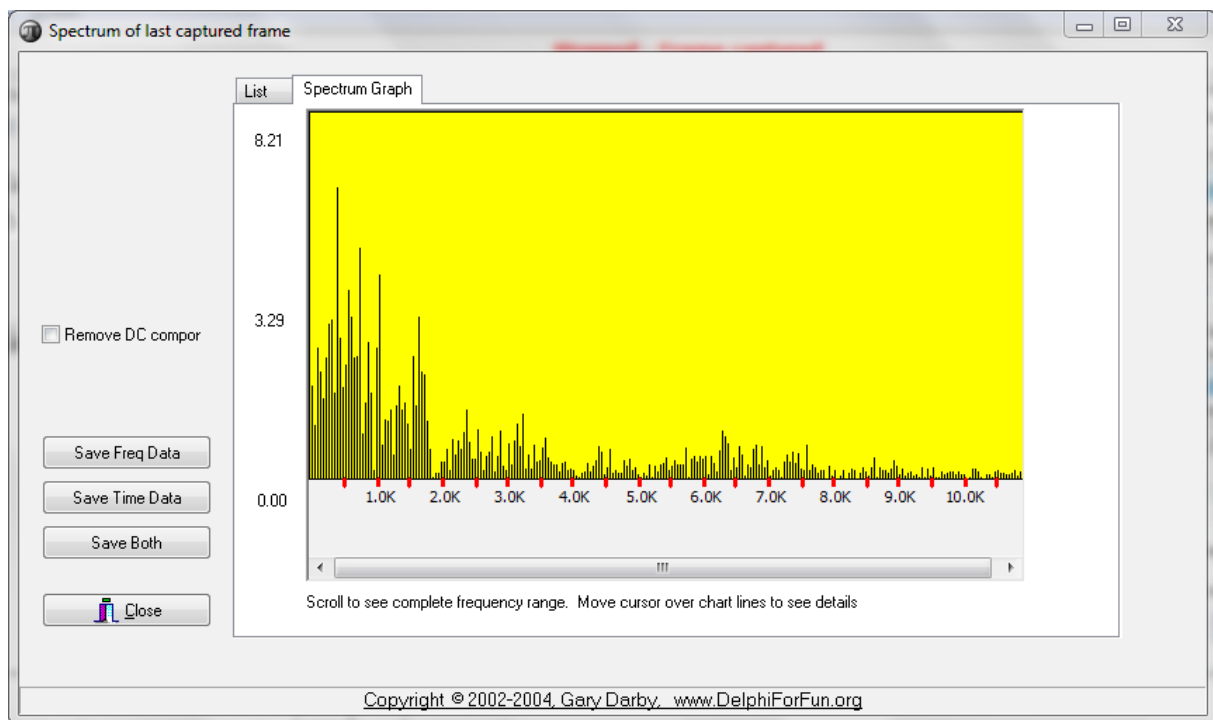
`QueryPerformanceFrequency` returns the number of "ticks" per seconds - it is the amount that the counter, `QueryPerformanceCounter`, will increment in a second.

What about when you need to process millions of tree brunch leaves or generate 10 quadrillions (10^{16}) of unique random numbers? In such scenarios it is important that your code executes as quickly as possible.

³ A clock shows just the position of the sun – we name that time!

The calculation is almost self defining: cycles per second = cycles / seconds, or, as reported, millions of cycles per second = cycles / microseconds. The program sets priorities to high values before performing a calculation and restores them after to increase the accuracy of the timing.

For example the voltage signals that an oscilloscope measures are generically called waves, which is any repeating pattern (sound waves, oceans waves, etc), and the signal displayed on the oscilloscope is called a waveform. Those waveforms can be split up in a frequency spectrum too:



```

if QueryPerformanceFrequency(Frequency) then
    QueryPerformanceCounter(Start1);
    //Sleep(2000); //2000 millisecs = 2 secs
    PerformanceDelayMS(2000000*10); //2000 micro secs = 2 milli secs
    //2000*1000 = 2 seconds

    QueryPerformanceCounter(Stop1);
    WriteLn('Delta: '+IntToStr(Stop1-Start1) + ' Freq: ' +
        IntToStr(Frequency) + ' ' +
        Format('%0.6f', [(Stop1-Start1)/Frequency]) + ' seconds')

```

☞ Just note that you have to check the boolean return value from `QueryPerformanceFrequency()` which returns false if performance counters are not implemented in your hardware.

```

PrintF('Delta: %d Freq: %d of %0.6f micro secs!', [Stop1-Start1, Frequency,
    (Stop1-Start1)/Frequency])

```

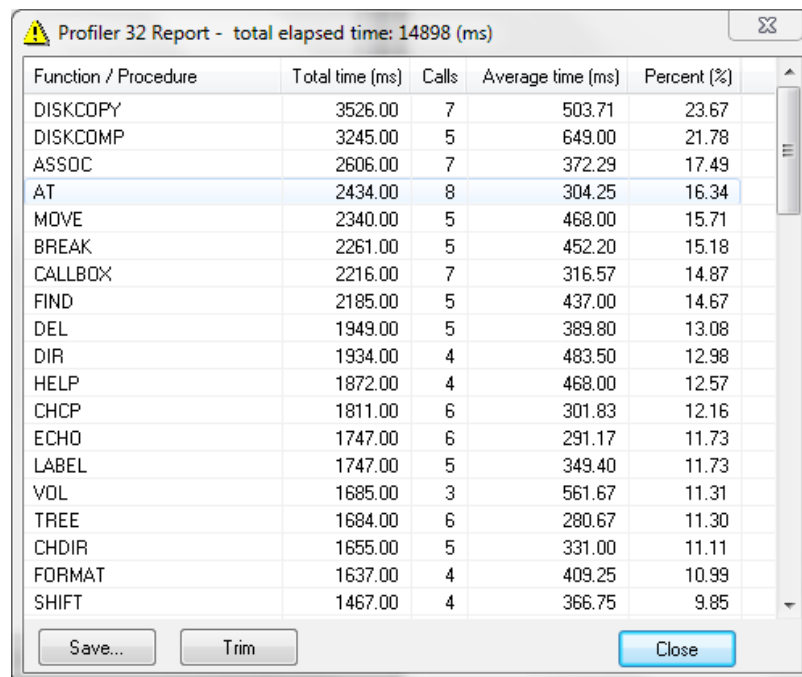
By calling this function at the beginning and end of a section of code, an application uses the counter as a high-resolution timer.

```
Time1:= Time;
PerformanceDelayMS(2000000*10);    //2000 micro secs = 2 milli secs
Printf('%d %s',[Trunc((Time-Time1)*24),
FormatDateTime("'"h runtime:" nn:ss:zzz',Time-Time1)])
```

Important is to find any hot spots of an application and then measure time or test the consuming cycles. Most of the time, you will be using the event handler for notification that a particular event occurred. But this time is user dependent, you can't accelerate a mouse move or something control driven.

Take the `OnMouseDown` event, for example. If you handle the `OnMouseDown` event, you are simply asking to be notified when the user clicks the component with the mouse (remember that forms or dialogues are components, too).

This picture below shows a profiler app in maXbox.



Profiler 32 Report - total elapsed time: 14898 (ms)

Function / Procedure	Total time (ms)	Calls	Average time (ms)	Percent (%)
DISKCOPY	3526.00	7	503.71	23.67
DISKCOMP	3245.00	5	649.00	21.78
ASSOC	2606.00	7	372.29	17.49
AT	2434.00	8	304.25	16.34
MOVE	2340.00	5	468.00	15.71
BREAK	2261.00	5	452.20	15.18
CALLBOX	2216.00	7	316.57	14.87
FIND	2185.00	5	437.00	14.67
DEL	1949.00	5	389.80	13.08
DIR	1934.00	4	483.50	12.98
HELP	1872.00	4	468.00	12.57
CHCP	1811.00	6	301.83	12.16
ECHO	1747.00	6	291.17	11.73
LABEL	1747.00	5	349.40	11.73
VOL	1685.00	3	561.67	11.31
TREE	1684.00	6	280.67	11.30
CHDIR	1655.00	5	331.00	11.11
FORMAT	1637.00	4	409.25	10.99
SHIFT	1467.00	4	366.75	9.85

Buttons: Save... Trim Close

1.1.2 Code Review Integration

Code-coverage use.

So we have seen testing goes with automation to test. There are tools which automate your tests driven by software metrics. With these tools at hand, quality managers, developers and technical architects have a complete workbench to rate application quality, perform quantitative analysis and define corrective action. Those tools measure the amount of test-cases and the metric is called code-coverage.



The cost of fixing a defect detected during unit testing or code metrics is lesser in comparison to that of defects detected at higher levels. Unit Testing is the first level of testing and is performed and logged prior to Integration Testing.

But most of the time we forget to check our code twice. To avoid this problem, metric products provide support for the so called Continuous Inspection paradigm by enabling real-time notifications when code quality defects are introduced in an application.

This transparency ensures that internal quality assurance is an integral part of the software development life-cycle, rather than an afterthought.

These features enable full support of a Continuous Inspection process and produce three immediate benefits:

- They encourage developer adoption by enabling developers to quickly identify code improvements and quick wins.
- They improve quality of code produced by increasing developer knowledge and understanding of code quality issues.
- They reduce maintenance cost through early identification of quality issues.


Every step in your development work-flow that can be automated should be. You need this automation to be able to focus all your time on building your product and providing value to your customers. Testing is one of those steps where automation is key.

Test the script with **F9** / F2 or press Compile. So far now we'll open the test-function example: `587_one_function_assert.txt`

http://www.softwareschule.ch/examples/587_one_function_assert.txt

Hope you did already work with the Starter 24 on Clean Code topics:

http://www.softwareschule.ch/download/maxbox_starter24.pdf

 **Conclusion** Stepping through code is one of the most basic debugging operations, yet it still needs to be mentioned here. Sometimes you fail to see the forest for the trees. (Just as sometimes authors of programming books fail to include the obvious!) Reviewing the basics from time to time can reveal something you were not previously aware of.

Test functions return a boolean that you can test for conditional parts of the functionality (correctness).

It is important to understand that when you catch an exception, code execution continues after executing the except block. One of the advantages to catching exceptions is that you can recover from the exception and continue program execution.

Notice the `Exit` statement in a code snippet. In this case, you don't want to continue code execution following the exception, so you exit the procedure after handling the exception.

“If the brain were so simple we could understand it, we would be so simple we couldn't.” Lyall Watson



Pic of script: 123_graphics_atom_save2.txt

Feedback @ max@kleiner.com

Literature: Kleiner et al., Patterns konkret, 2003, Software & Support

http://www.softwareschule.ch/download/codesign_2015.pdf

<http://softwaretestingfundamentals.com/unit-testing/>

<http://www.sonarsource.com/products/features/continuous-inspection/>

https://www.academia.edu/5129685/Software_Quality_with_SONAR

<https://github.com/maxkleiner/maXbox3/releases>

1.2 Appendix Assert Study with Assembler

```
PROCEDURE Assert(Cond: boolean; const Msg: string);  
var  
  progSeg,  
  progOfs: word;  
begin  
  asm  
    mov ax, [bp+04]  
    mov es, ax  
    mov ax, word ptr es:0  
    mov progSeg, ax  
    mov ax, [bp+02]  
    mov progOfs, ax  
  end;  
  if (Cond = FALSE) then begin  
    mmDebug.Lines.Add(Msg + ' at location ' +  
      IntToHex(progSeg, 4) + ':' + IntToHex(progOfs, 4) );  
    ShowModal;  
  end;  
end;  
  
// WriteToOSEventLog//  
procedure WriteToOSEventLog(const logName, logCaption, logDetails : UnicodeString;  
  const logRawData : String = "");  
  
var  
  eventSource : THandle;  
  detailsPtr : array [0..1] of PWideChar;  
begin  
  if logName<>" then  
    eventSource:=RegisterEventSourceW(nil, PWideChar(logName))  
  else eventSource:=RegisterEventSourceW(nil,  
  PWideChar(ChangeFileExt(ExtractFileName(ParamStr(0)), "")));  
  if eventSource>0 then begin  
    try  
      detailsPtr[0]:=PWideChar(logCaption);  
      detailsPtr[1]:=PWideChar(logDetails);  
      ReportEventW(eventSource, EVENTLOG_INFORMATION_TYPE, 0, 0, nil,  
        2, Length(logRawData),  
        @detailsPtr, Pointer(logRawData));  
    finally  
      DeregisterEventSource(eventSource);  
    end;  
  end;  
end;
```

Check the script! Script Example: maxbox3\examples\318_excel_export3