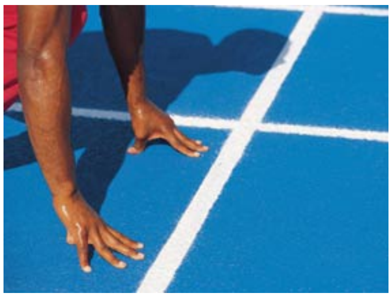
maXbox



maXbox Starter 43

Work with Code Metrics V2

1.1 Measure with Metrics

Software quality consists of both external and internal quality.
Continuous Inspection is a holistic, fully-realized process designed to make internal code quality an integral part of the software development life cycle process.
Today we step through optimize and review your code with maXbox and some style guide. You cannot improve what you don't measure and what you don't measure, you cannot prove.
A tool can be great for code quality but also provides a mechanism for extending your functions and quality with checks and tests.

One of a first metric is lines of code which you can define as a macro in maXbox and many other tools with #locs.
A good style is to set a header in your code file to document your purpose. You can set the macros like #locs: and #host: in your header or elsewhere in a comment line, but not two or more on the same line when it expands with content:

Try: `369_macro_demo.txt`

```
{*****************************************************************
 * Project   : Metrics Macro Demo
 * App Name  : #file:369_macro_demo.txt
 * Purpose   : Demonstrates function macros in header
 * Date      : 21/09/2010  - 14:56 - #date:01.06.2015 16:38:20
 *             #path E:\maxbox\maxbox3\examples\
 *             #file 369_macro_demo.txt
 *             #perf-50:0:4.484
 * History   : translate/implement June 2013, #name@max
 *           : Sys demo for mX3, enhanced macros, #locs:149
 ****************************************************************}
```

As you may know there's no simple solution to put a macro because the step of the preprocessor has to be the first to expand. So a macro is not

part of the source code and the best way is to set a macro always as a one liner or somewhere else in comment.
The ratio of code to comment is also a well known metric which you can predefine with a few procedures.

```
 writeln(CommentLinesWithSlashes('this is a //comment for a
moment'))
```

```
Function CommentLinesWithSlashes(const S: String): String');
Function UncommentLinesWithSlashes(const S: String): String');
```

All macros and metrics are marked with yellow. One of my favour as I said is #locs: means lines of code metric and you get always the certainty if something has changed by the numbers of line. So the editor has a programmatic macro system which allows the pre compiler to be extended by user code I would say user tags. You find more macros at
All Functions List: `maxbox_functions_all.pdf`

Some macros produce simple combinations of one liner tags but at least they replace the content by reference in contrary to templates which just copy a content by value.
After the `End.` of the code section you can set a macro without comment signs so you can enlarge your documentation with version and time information like:

#tech:perf: 0:0:1.151 threads: 4 192.168.56.1 12:06:17 4.2.2.95

Or you put a macro behind a code-line, so this is not the end of the line, a second time test is

```
 maxcalcF('400000078669 / 2000123')
 //#perf>0:0:1.163
```

and we get as float: 199987.740088485

maXbox4 `568_U_BigFloatTestscript2.pas` Compiled done: 6/18/2015

Generally you can categorize metrics in the following 4 groups:

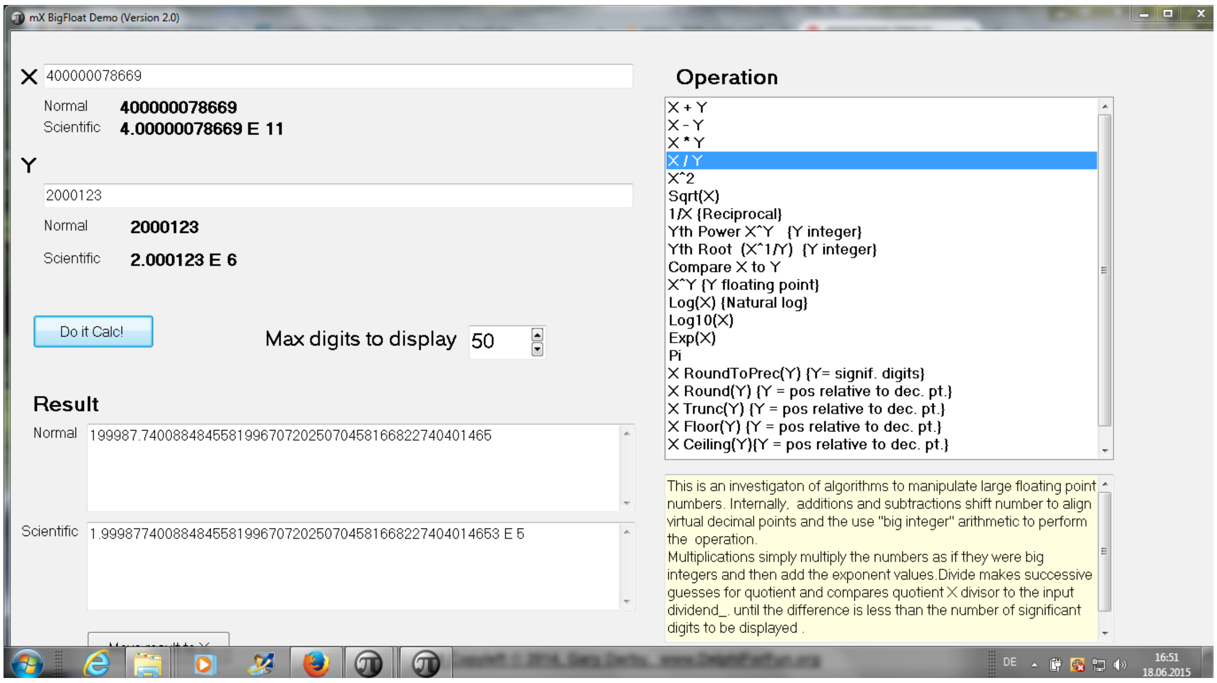
Maintainability 29
- Readability 25
- Understandability 4
- Modularity 3

Changeability 2
- Data 1
- Logic 1

Reliability 10
- Architecture 1
- Data 1
- Exception handling 1

    Fault tolerance 1
    Instruction 5
    Logic 1
    Security 3
Testability 1
    Unit level 1
    Function level 2
    Uncharacterised 3

Lets go back to our performance metric example to close, which is one of a non static metric:


mX BigFloat Demo (Version 2.0)
Operation
X + Y
X - Y
X * Y
X / Y
X^2
Sqrt(X)
1/X {Reciprocal}
Yth Power X^Y {Y integer}
Yth Root (X^1/Y) {Y integer}
Compare X to Y
X^Y {Y floating point}
Log(X) {Natural log}
Log10(X)
Exp(X)
Pi
X RoundToPrec(Y) {Y= signif. digits}
X Round(Y) {Y = pos relative to dec. pt.}
X Trunc(Y) {Y = pos relative to dec. pt.}
X Floor(Y) {Y = pos relative to dec. pt.}
X Ceiling(Y){Y = pos relative to dec. pt.}
Do it Calc!
DE
16:51
18.06.2015

X 400000078669
Normal 400000078669
Scientific 4.00000078669 E 11

Y 2000123
Normal 2000123
Scientific 2.000123 E 6

Max digits to display 50

Result
Normal 199987.74008848455819967072025070458166822740401465
Scientific 1.9998774008848455819967072025070458166822740401 4653 E 5

This is an investigaton of algorithms to manipulate large floating point numbers. Internally, additions and subtractions shift number to align virtual decimal points and the use "big integer" arithmetic to perform the operation.
Multiplications simply multiply the numbers as if they were big integers and then add the exponent values.Divide makes successive guesses for quotient and compares quotient X divisor to the input dividend_. until the difference is less than the number of significant digits to be displayed .

29×37×127^(-1)×179×15749^(-1)×2082607

```
   maxcalcF('29*37*(127^-1)*179*(15749^-1)*2082607');

>> 199987.740088485

   maxcalcF('29*37*(127^-1)*179*(15749^-1)*2082607');

//#tech>perf: 0:0:1.190 threads: 6 192.168.56.1 12:49:55 4.2.2.95
//>>> 199987.740088485
```

1.2 Extend with Technical Quality

There is an ISO definition of software quality, called ISO 25010 [2]. This standard defines 8 main quality factors and a lot of attributes. The 8 main technical quality factors are:

1. Functional suitability.

The degree to which the product provides functions that meet stated and implied needs of a style guide when the product is used under specified conditions.

2. Reliability. The degree to which a system or component performs specified functions under specified conditions for a specified period of time.

3. Performance efficiency. The performance relative to the amount of resources used under stated conditions.

4. Operability. The degree to which the product has attributes that enable it to be understood, learned, used and attractive to the user, when used under specified conditions.

5. Security. Degree of protection of information and data so that unauthorized persons or systems cannot read or modify them and authorized persons or systems are not denied access to them.

6. Compatibility. Degree to which two or more systems or components can exchange information and/or perform their required functions while sharing the same hardware or software.

7. Maintainability. Degree of effectiveness and efficiency with which the product as a software can be modified.

8. Transferability. The degree to which a system or component can be effectively and efficiently transferred from one hardware, software or other operational or usage environment to another.

- http://www.tiobe.com/_userdata/files/TIOBEQualityIndicator_v3_8.pdf

Now we step with a traceable real world function through those 8 points.

For example a DLL is a library, short for Dynamic Link Library, a library of executable functions or data that can be used by a Windows or Linux application. This is how we declare a function we want to use from a DLL:

```
Function OpenProcess2(dwDesiredAccess: DWORD;
         bInheritHandle: BOOL; dwProcessId:DWORD): THandle;
               External 'OpenProcess@kernel32.dll stdcall';
```

Suppose you want to use the function `OpenProcess` of the ‘`kernel32.dll`’. All you have to do is to declare above statement and you get access to the kernel! With external you made these functions available to callers external to the DLL, so we must export them or at least say the function we use is from `External`.

1. Functional suitability

This means for e.g. also to use the modifier `stdcall` because this is a C convention. The function name `OpenProcess2` is different from the original name `OpenProcess`! This is an alias to prevent name conflicts or name it you like because you do have conventions you are free to rename or reuse a function in your script.

2. Reliability

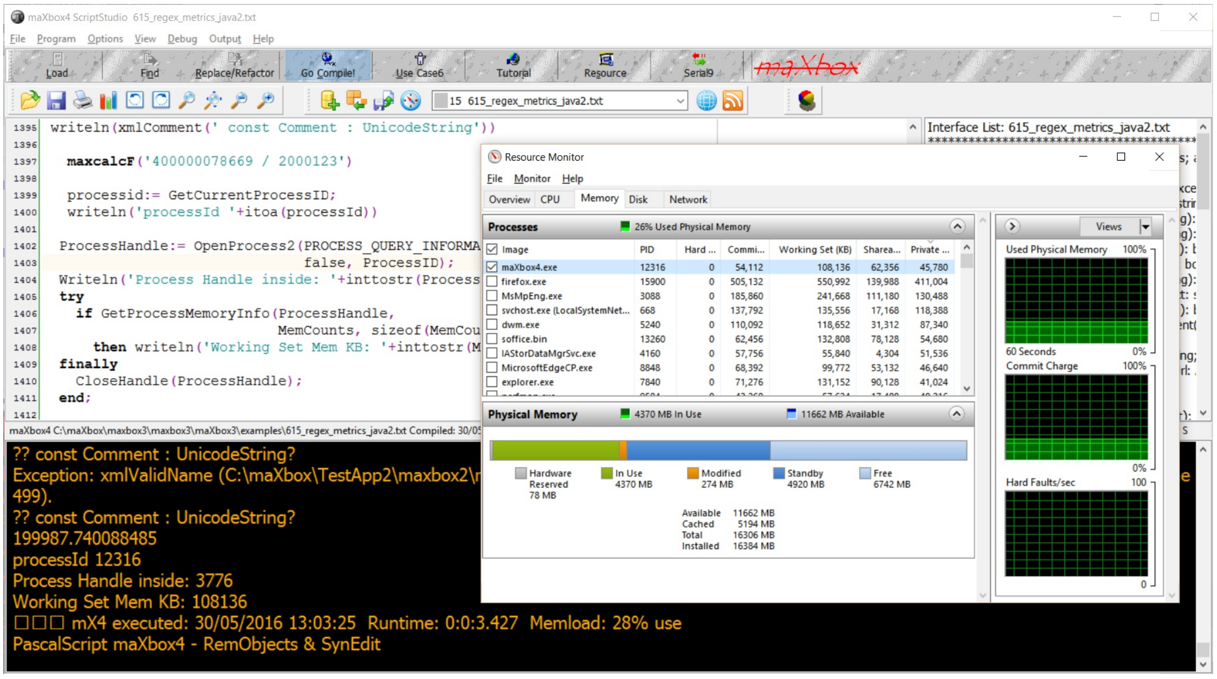
For this we have to test the fault tolerance or correctness of the function and to proof the correctness with a real call.

```
ProcessHandle:= OpenProcess2(PROCESS_QUERY_INFORMATION or
                                       PROCESS_VM_READ, false, ProcessID);
 Writeln('Process Handle inside: '+inttostr(ProcessHandle));
 try
   if GetProcessMemoryInfo(ProcessHandle,
                      MemCounts, sizeof(MemCounts))
     then writeln('Working Set Mem KB: '+inttostr(MemCounts.WorkingSetSize div 1024));
 finally
   CloseHandle(ProcessHandle);
 end;
```

The fault tolerance is covered with a try finally statement to free resources and the result can be done with the help of the resource monitor:
As an output we can get:

```
Process Id 12316
Process Handle inside: 3384
Working Set Memory KB: 108136
```

But is it the right result of 108136 at runtime, so lets open the resource monitor and find to find and compare our process best with a next screenshot:
Reliability means also secure systems as we can trust on the data and calculations (integrity).
This guiding theme is the development of program analysis methods and tools that target the verification of security properties in a sound, precise, scalable, and usable way.
This will then create the basis for a semantically substantiated (and thus, reliable) certification of security guarantees for software systems.
Verification tools will be employed to establish security properties of programs as well as to ensure the soundness of security analysis tools.
That means in our example to test the function `ProcessMemoryInfo()` in various other systems as a proof of work.



```
writeln(xmlComment(' const Comment : UnicodeString'))

 maxcalcF('400000078669 / 2000123')

 processid:= GetCurrentProcessID;
 writeln('processId '+itoa(processId))

ProcessHandle:= OpenProcess2(PROCESS_QUERY_INFORMA
                             false, ProcessID);
Writeln('Process Handle inside: '+inttostr(Process
try
  if GetProcessMemoryInfo(ProcessHandle,
                        MemCounts, sizeof(MemCou
    then writeln('Working Set Mem KB: '+inttostr(M
finally
  CloseHandle(ProcessHandle);
end;
```

maXbox4 C:\maXbox\maxbox3\maxbox3\maXbox3\examples\615_regex_metrics_java2.txt Compiled: 30/05

```
?? const Comment : UnicodeString?
Exception: xmlValidName (C:\maXbox\TestApp2\maxbox2\r
499).
?? const Comment : UnicodeString?
199987.740088485
processId 12316
Process Handle inside: 3776
Working Set Mem KB: 108136
□□□ mX4 executed: 30/05/2016 13:03:25  Runtime: 0:0:3.427  Memload: 28% use
PascalScript maXbox4 - RemObjects & SynEdit
```

Resource Monitor

Processes — 26% Used Physical Memory

Image	PID	Hard ...	Commi...	Working Set (KB)	Sharea...	Private ...
✓ maXbox4.exe	12316	0	54,112	108,136	62,356	45,780
firefox.exe	15900	0	505,132	550,992	139,988	411,004
MsMpEng.exe	3088	0	185,860	241,668	111,180	130,488
svchost.exe (LocalSystemNet...	668	0	137,792	135,556	17,168	118,388
dwm.exe	5240	0	110,092	118,652	31,312	87,340
soffice.bin	13260	0	62,456	132,808	78,128	54,680
IAStorDataMgrSvc.exe	4160	0	57,756	55,840	4,304	51,536
MicrosoftEdgeCP.exe	8848	0	68,392	99,772	53,132	46,640
explorer.exe	7840	0	71,276	131,152	90,128	41,024

Physical Memory — 4370 MB In Use — 11662 MB Available

Hardware Reserved 78 MB · In Use 4370 MB · Modified 274 MB · Standby 4920 MB · Free 6742 MB

Available	11662 MB
Cached	5194 MB
Total	16306 MB
Installed	16384 MB

maXbox4 ScriptStudio 615_regex_metrics_java2.txt
File Program Options View Debug Output Help
Load
Find
Replace/Refactor
Go Compile!
Use Case6
Tutorial
Resource
Serial9
maXbox
15 615_regex_metrics_java2.txt
Interface List: 615_regex_metrics_java2.txt
File Monitor Help
Overview CPU Memory Disk Network
Views
Used Physical Memory 100%
60 Seconds 0%
Commit Charge 100%
Hard Faults/sec 100

Yes right 2 functions with the same result.
By the way another test is “Don't test floating point numbers for equality!” Of course those 2 numbers are Integers to compare but I had to divide the result by 1024 to get Kbytes:

```
then Writeln('Working Set Mem KB: '+inttostr(MemCounts.WorkingSetSize div 1024));
```

Therefore, the use of the equality (== or =) and inequality (!= or <>) operators on float or double values is almost always an error. Instead the best course is to avoid floating point comparisons altogether as in our case with the `div` operator.

3. Performance efficiency

No problem at all because the performance relative to the amount of resources used under this conditions is small, check it with

#tech: perf: 0:0:3.450 threads: 11 192.168.56.1 13:40:05 4.2.2.98

4. Operability

This one is not so easy. Operability for instance, with sub-attributes such as “Attractiveness” and “Ease of Use” is not that clear to understand. How to measure this and what is the unit of measurement?
For me global variables and possible side effects is a test to find and global variables as well as direct output has to be prevented.
In our case we use 3 nasty variables:

```
ProcessHandle : THandle;
MemCounts : TProcessMemoryCounters;
ProcessID : DWORD;
```

This is not operable and we design a real procedure with parameters around the snippet:

```
function ProcessMemoryUsageWorkingSet(ProcessID : DWORD): DWORD;
var ProcessHandle : THandle;
    MemCounters   : TProcessMemoryCounters;
begin
  Result:= 0;
  ProcessHandle:= OpenProcess2(PROCESS_QUERY_INFORMATION or
                                                   PROCESS_VM_READ,
                         False, ProcessID);
  try
    if GetProcessMemoryInfo(ProcessHandle,
                       MemCounters, sizeOf(MemCounters))
    then Result:= MemCounters.WorkingSetSize div 1024;
  finally
    CloseHandle(ProcessHandle);
  end;
end;
```

Now we have a clear name ProcessMemoryUsageWorkingSet and the var globals are gone:

```
function ProcessMemoryUsageWorkingSet(ProcessID : DWORD): DWORD;
var ProcessHandle : THandle;
    MemCounters   : TProcessMemoryCounters;
begin
  Result:= 0;
```

Two of them are now local variables and the third one `ProcessID` is a parameter of the function. Then we initialize the result with zero and we delete the `writeln` as a dangerous direct output.

```
  ProcessHandle:= OpenProcess2(PROCESS_QUERY_INFORMATION or
                                      PROCESS_VM_READ, False, ProcessID);
```

The `PROCESS_QUERY_INFORMATION` is required to retrieve certain information about process, such as its token, exit code, and priority class `- 0x0400` and `PROCESS_VM_READ` is need to read memory at runtime in a process using `ReadProcessMemory`.

5. Security

This is always a big thing. Check for administration rights and exceptions. In our case we don't handle sensitive data for the moment.
Exceptions are meant to represent the application's state at which an error occurred.

Making all fields or functions final with `Consts` can ensure this state:

- Will be fully defined at the same time the exception is instantiated.
- Won't be updated or corrupted by some bogus error handler.

This will enable developers to quickly understand what went wrong.
If a property is mapped with accessory functions, the their names begin with « Get » or « Set ».

The following code shows it:

```
ProcessHandle:= OpenProcess2(PROCESS_QUERY_INFORMATION or
                             PROCESS_VM_READ, false, ProcessID);
```

Another example for reliability is the ever lasting function to get the right OS version. There are many functions, classes and components which didn't work properly or aren't up to date so why not take the secure and easy way with the command line.
Windows or Linux has command line utilities that show us the version of the Windows OS we are using including the service pack number or BIOS releases like that:

```
writeln(GetDOSOutput('Ver','C:\'))
```

We can find service pack number as well as the OS name and build using `Systeminfo` command. But `Systeminfo` dumps a lot of other information also. So we need to use `findstr` command with a pipe to filter out unwanted information.

```
writeln(GetDOSOutput('Systeminfo | findstr "OS Version"','C:\'))

writeln(GetDOSOutput('Systeminfo | findstr /B /C:"OS Name" /C:"OS
                                                  Version"','C:\'))
```

If you want to print more details, then you can use just “OS” in the `findstr` search pattern and in maXbox you will see the result captured.

6. Compatibility

The degree to which two or more systems or components can exchange information and/or perform their required functions.
OK it seems with a DLL you guarantee some degree of independent use of a certain function:

```
Function OpenProcess2(dwDesiredAccess:DWORD; bInheritHandle:BOOL;
             dwProcessId:DWORD): THandle;
                     External 'OpenProcess@kernel32.dll stdcall';
```

```
  Function GetProcessMemoryInfo(Process: THandle;
                               var MemoryCounters: TProcessMemoryCounters;
                                cb: DWORD): BOOL; //stdcall;;
                                          External 'GetProcessMemoryInfo@psapi.dll stdcall';
```

Variables and functions identifiers are written using concatenated words in lower case with each first letter in upper case. The names shall be chosen to be as meaningful as possible.

```
  Function GetProcessMemoryInfo(Process: THandle;
```

Built-in data types are written in lower case except the first letter.
Function arguments are separated by comma when calling the function.
Put a space after the comma but none before and so on.
To get more compatibility there's a template `dll`, write the word `dll` maybe at the beginning or below or inside of your code and press `<Ctrl> j` and it gets expanded to:

```
function  MyGetTickCount: Longint;
    external 'GetTickCount@kernel32.dll stdcall';
```

Now we can change it e.g. to 64bit:

```
function  MyGetTickCount64: Longint;
    external 'GetTickCount64@kernel32.dll stdcall';
```

7. Maintainability

Use modifying as a need like programming for change.
Comments describing function behaviour has to be put before the function itself. Comments within the function body is aligned with code and is best placed before the code if the comment apply to several lines, or is too long to fit at the right of the code.

```
Result:= 0;
   //PROCESS_VM_READ Required read memory in a process using ReadProcessMemory
  ProcessHandle:= OpenProcess2(PROCESS_QUERY_INFORMATION or
                                             PROCESS_VM_READ, False, ProcessID);
```

Constant identifiers shall be written in all upper-case, with underline character to split the identifier in more readable words.

```
Const       IP_MULTICAST_TTL  = 3;
            IP_MULTICAST_LOOP = 4;
            IP_ADD_MEMBERSHIP = 5;
```

Global variables are prefixed with uppercase letter ‘G’:

```
    var    GClassCritSect : TRTLCriticalSection;
```

For example we want to change or expand our function to get the `PeakWorkingSetSize`, there's the change to do:

```
try
  if GetProcessMemoryInfo(ProcessHandle,
                          MemCounters, sizeOf(MemCounters))
  then Result:= MemCounters.PeakWorkingSetSize div 1024;
finally
  CloseHandle(ProcessHandle);
end;
```

and the result again:

```
Working Set Mem KB: 108148 - Working Set Peak KB: 124348
```

8. Transferability

So this criteria is about portability and migration to another platforms. For this we can use for example conditional expression. Do not use leading and ending parenthesis. When the expression is made of several sub-expressions, parenthesis are used around each part.
Conditional compilation directives start at the left margin when possible. Indentation is not affected by the conditional compilation.

```
uses
{$IFDEF FPC}
 // use the LCL interface support whenever possible
 {$IFDEF USE_WINAPI}
  Windows,
 {$ENDIF}
  GraphType, IntfGraphics, LCLType, LCLIntf, LMessages, LResources,
{$ELSE}
  Windows, Messages,
 {$IFDEF USE_PNG_SUPPORT}
  PngImage,
 {$ENDIF}
{$ENDIF}
  Classes, Forms, Graphics, Controls, Types, KFunctions
{$IFDEF USE_THEMES}
  , Themes
 {$IFNDEF FPC}
  , UxTheme
 {$ENDIF}
{$ENDIF}  ;
```

Sometimes conditional compilation apply to only a part of a statement. In that case, it is necessary to put the directives in line with the statement:

```
  procedure MyOwnProc(); {$IFNDEF BCB} virtual; {$ENDIF}
  procedure Clear; {$IFDEF FPC}override;{$ENDIF}
```

But that's just a simple solution cause the computer world is full of incompatibility like the following and last example:

```
// SOME REMARKS OF CR AND LF
#  On Mac,  the newline is the CR character (# 13)
#  Windows , il se fait par caractère CR+LF (#13+#10), Win it is the character CR + LF (# 13 + # 10)
# UNIX , par le caractère LF (#10). , and on Unix, the LF (# 10).
```

You find the script to test and work:

http://www.softwareschule.ch/examples/615_regex_metrics_java2pascal.txt

1.3 Sonar Qube and maXbox

Most of metric functions you can control and run with Sonar.
The Goal is to audit, measure, refactor, find and fix in the Sonar Dashboard with the following Quality Gates:

- Lines of code
- Code Complexity
- Code Coverage
- Rules Compliance
- Document Comments

Generally, there is a “runner” that consumes the source code and analyses it via plug-ins. This information is published to the SonarQube database directly to get the code review result.
Before each run, the runner makes a call to the server to download configuration settings stored there, and mashes those with configuration settings stored in a local configuration file, as well as project-specific config-files which can be controlled by maven, maXbox or another runner.

SonarQube is originally written for Java analysis at PMD and later added C# and further support.
The SonarQube (web) server pulls the results from the database and provides a UI for reviewing and managing them, as well as a UI for managing configuration settings, user authentication, etc. The server also has it’s own config file `sonar.properties` where you define database settings, user authentication settings, etc.

The monitor shows then the following topics:

- Duplicated code
- Coding standards
- Unit tests and Code Coverage
- Complex code
- Potential bugs and issues
- Comments /API Documentation
- Design and architecture

The defined Rule-set of SonarQube (web) 5.1.1 can be found at:

Important metrics to look for
- duplicated_blocks
- violations – info_violations
- public_undocumented_api
- uncovered_complexity_by_tests (it is considered that 80% of coverage is the objective)
- function_complexity_distribution >= 8,
- class_complexity_distribution >= 60 package_edges_weight

1.4 Console Capture DOS SonarQube51

But then I want to test some Shell Functions on a DOS Shell or command line output. The code below allows to perform a command in a DOS Shell and capture it's output to the maXbox console.
First step with

```
Function cyShellExecute(Operation,FileName,Parameters,Directory:String;
                         ShowCmd:Integer):Cardinal;
```

```
function ShowFindFilesDlg(const Folder: string): Boolean;
 begin
 Result:= {mXCindyShellAPI.}cyShellExecute(
   'find', PChar(Folder), '', '', {Windows.}SW_SHOW
  ) > 32;
 end;
```

Next step is to capture the DOS output:

The captured output is sent “real-time” to the Memo2 parameter as console output in maXbox:

```
    srlist:= TStringlist.create;
      ConsoleCapture('C:\', 'cmd.exe', '/c dir *.*',srlist);
      writeln(srlist.text)
    srlist.Free;
```

But you can redirect the output `srlist.text` anywhere you want.

For example you can capture the output of a DOS console and input into a textbox, or you want to capture the command start of demo app and input into your app that will do further things.

```
 ConsoleCapture('C:\', 'cmd.exe', '/c ipconfig',srlist);
 ConsoleCapture('C:\', 'cmd.exe', '/c ping 127.0.0.1',srlist);
```

☝ It is important to note that some special events like `/c java -version` must be captured with different parameters like /k or in combination.

Here's the solution with `GetDosOutput()`:

```
writeln(GetDosOutput('java -version','c:\'));
```

or like the man-pages in Linux

```
writeln('GetDosOut: '+GetDosOutput('help dir','c:\'));
```

```
writeln(GetDOSOutput('Systeminfo | findstr "OS Version"','C:\'))
```

The `systeminfo` command gives the edition info under the headings "OS Name:" and "OS Version:" as well as a lot of other information all in the console or redirected to maXbox for further editing. You can parse it with `findstr` if you need only the edition info.

```
OS Name:                  Microsoft Windows 10 Home
OS Version:               10.0.10586 N/A Build 10586
```

and systeminfo | findstr /C:"System Type" would give you the processor architecture. From this you can derive if it's 32 or 64 bit!
Be aware `findstr` is case sensitive.

1.5 Byte Code Performance

Intermediate representations such as byte-code may be output by programming language implementations to ease interpretation, or it may be used to reduce hardware and operating system dependence by allowing the same code to run on different platforms. So you can share your code as source in a normal text-file (*.txt) or as bytecocde (*.psb) to gain speed or obfuscation.

In some cases, you may want to export or deliver a script with its byte-code to store on removable media or to use on a different computer without the source as a text-file. This is how you can do that:

1. You open a script and compile it before.
2. you go to `/Options/Save Bytecode/` and the console writes:

```
-----PS-BYTECODE (PSB) mX4-----13:48:38 -----BYTECODE
saved as:
C:\maXbook\maxbox3\mX3999\maxbox3\examples\287_eventhandl
ing2_primewordcount.psb
----IFPS#
```

3. you load the byte-code by `/Options/Load Bytecode`...

```
IFPS#
### mX3 byte code executed: 10.06.2015 13:53:20 Runtime:
0:0:1.577 Memoryload: 60% use ByteCode Success Message
of: 287_eventhandling2_primewordcount.psb
```

4. When testing is finished you send the byte-code to your client

1.6 Exception Handling

A few last words how to handle Exceptions within maXbox:

Prototype:
`procedure RaiseException(Ex: TIFException; const Msg: String);`

Description: Raises an exception with the specified message. Example:

```
begin
  RaiseException(erCustomError,'Your message goes here');
// The following line will not be executed because of the
exception!
  MsgBox('You will not see this.', 'mbInformation', MB_OK);
end;
```

This is a simple example of a actual script that shows how to do try except with raising a exception and doing something with the exception message.

procedure Exceptions_On_maXbox;

```
var filename,emsg:string;
begin
   filename:= '';
   try
     if filename = '' then
       RaiseException(erCustomError,
                   'Exception: File name cannot be blank');
   except
     emsg:= ExceptionToString(ExceptionType, ExceptionParam);

       //do act with the exception message i.e. email it or
       //save to a log with hlog.add() etc
     writeln(emsg)
   end;
end;
```

☝ The `ExceptionToString()` returns a message associated with the current exception. This function with parameters should only be called from within an except section.

1.7 The Log Files

Log files contain tons of valuable application and business data. Unfortunately, this value is often never realized because log files go simply ignored. The mX4 Log Agent can help remedy this by parsing metrics and events from logs, so the data within can be graphed in real-time, all the time you want. The log is multithreading with macros like `{App_ver}`.
You write at the log with hlog:

```
hlog.Add('> Start script: {App_name} v{App_ver}{80@}{now}');
```

There are 2 log files for metrics; a runtime log and an exception log:

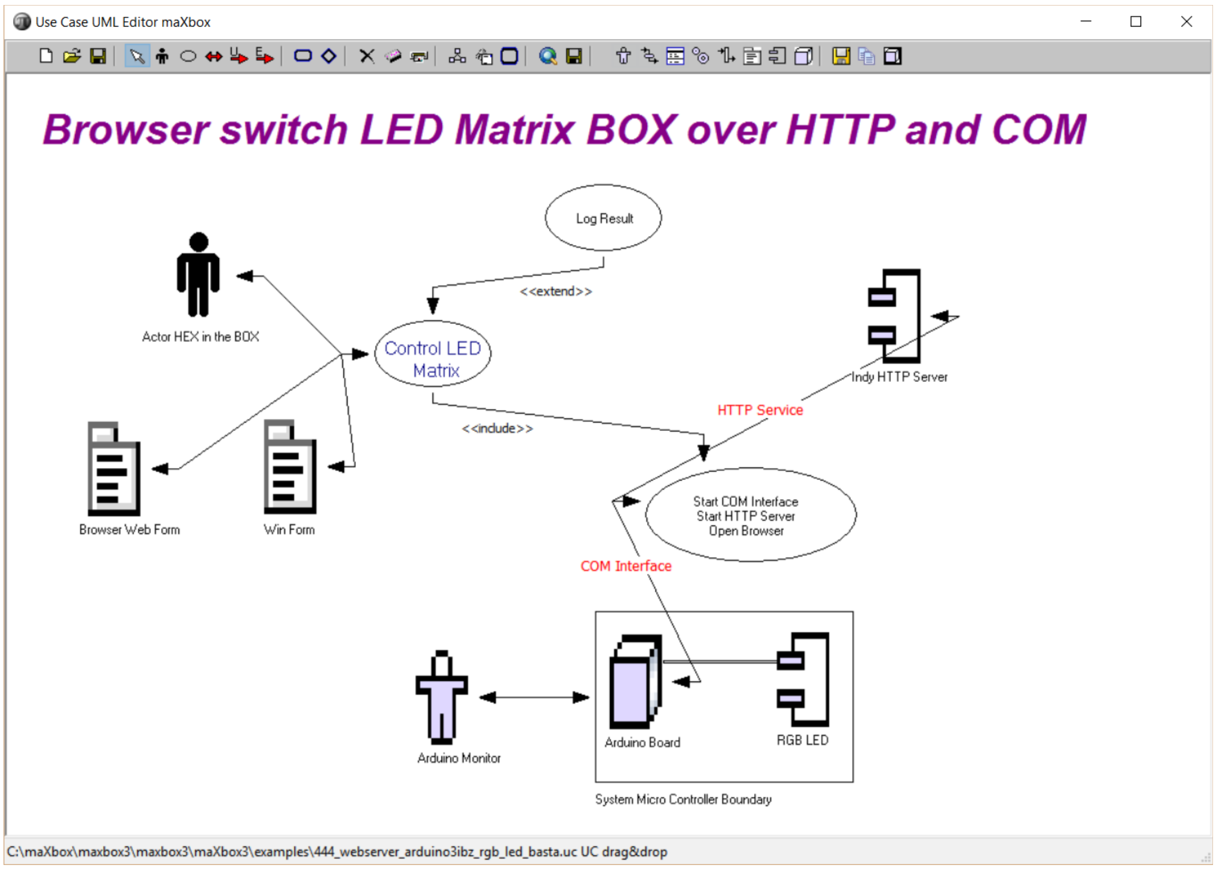
using Logfile: `maxboxlog.log` , Exceptionlogfile: `maxboxerrorlog.txt`

```
New Session Exe Start C:\maXbox\tested
>>>> Start Exe: maXbox4.exe v4.0.2.80 2016-02-03 14:37:18
>>>> Start [RAM  monitor] : Total=2147483647, Avail=2147483647,
Load=30% ; [Disk monitor] : Available to user=671317413888, Total on
disk=972076589056, Free total on disk=671317413888 ; 2016-02-03
14:37:18

>>>> Start Script: C:\Program Files
(x86)\Import\maxbox4\examples\640_weather_cockpit6_1.TXT  2016-02-03
14:37:33 From Host: maXbox4.exe of C:\maXbox\maxbox3\work2015\Sparx\
>>>> Stop Script: 640_weather_cockpit6_1.TXT
[RAM  monitor] : (2147483647, 2147483647, 30%) Compiled+Run Success!
Runtime: 14:37:33.267

New Session Exe Start C:\Program Files(x86)\Import\maxbox4\
examples\640_weather_cockpit6_1.TXT
>>>> Start Exe: maXbox4.exe v4.2.2.95 2016-05-19 09:15:17
>>>> Start [RAM  monitor] : Total=2147483647, Avail=2147483647,
Load=25% ; [Disk monitor] : Available to user=675888001024, Total on
disk=972076589056

Session Exe Stop!
C:\maXbox\maxbox3\maxbox3\maXbox3\examples\700_function_snippets3.pas
>>>> Stop Exe: maXbox4.exe v4.2.4.25       2016-06-06 11:38:40
```



1.8 Appendix

“Wise men speak because they have something to say; Fools, because they have to say something”. -Plato
Something with the Internet of Thing ;-)

Feedback @ max@kleiner.com
Literature: Kleiner et al., Patterns konkret, 2003, Software & Support
https://github.com/maxkleiner/maXbox3/releases

Ref: TIOBEQualityIndicator_v3_8.pdf

http://www.tiobe.com/_userdata/files/TIOBEQualityIndicator_v3_8.pdf

[2] ISO, “ Systems and software engineering – Systems and software Quality Requirements and Evaluation (SquaRE) – System and software quality models ”, ISO/IEC 25010:2011, 2011, obtainable from

http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=35733

Tutorials concerning Clean Code and Metrics:

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

[5] Wikipedia, “ Code Coverage ”, extracted July 2012, obtainable from http://en.wikipedia.org/wiki/Code_coverage.

[6] Wikipedia, “ Abstract Interpretation ”, extracted July 2012, obtainable from http://en.wikipedia.org/wiki/Abstract_interpretation

[7] Wikipedia, “ Coding Conventions ”, extracted July 2012, obtainable from http://en.wikipedia.org/wiki/Coding_standard

Test Script available:

http://www.softwareschule.ch/examples/615_regex_metrics_java2pascal.txt