



maXbox Starter 24

Start with Clean Code

1.1 From Archimedes to PI

Today we dive into Clean Code and Refactoring. Robert C. Martin offers in the first chapter of his book 'Clean Code' several definitions of 'clean code' from different well known software experts. So what's Clean Code in my opinion?

- Easy to understand and comprehensible.
- Easy to modify.
- Simple to test.
- Uncomplicated reuse.
- And works correctly (Kent Beck's suggestion).

Interesting point is also the comment behind the code. The less comment you have could be a sign of code that doesn't require any comments to be easily understood. I mean you can comment like this `///very tricky section...` which is a comment but makes less sense. On the other side we can find code so understandable, that it doesn't need additional comment so we come to the first law of clean code:

Code has to be self explanatory.

Code which reads as close to a human language as possible. We mean it on all the levels: from syntax used, naming convention and alignment of source all the way to algorithms used quality of comments and complexity of distribution of code between modules.

Let's put a first example:

```
if (S_OK:= strFNA.find(0, 'the sign of the cross'))
```

This line isn't really readable let's put it this way:

```
if (filename.contains('the sign of the cross'))
```

There a lot of design principles that we already know and apply rigorously throughout tons of software design, but most importantly, the way to simplicity aren't that simple! Clean Code also proposes most of all 3 principles:

1. Code Conventions
2. Design Patterns
3. Convention over Configuration

I will show you this with a topic in math; a few equations with PI:

How do we compute the circumference of a circle? The circumference of a circle is the distance around the edge of the circle. You compute: $C=d*PI \rightarrow [d=Diameter]$
The diameter of a circle is the distance from one edge to the other, through the centre.
Now let's put this in a function:

```
147: function circleCircumference(dia: double): double;
148: begin
149:   result:= dia * PI;
150: end;
```

Is this really Clean Code; yes or no? PI is a constant so it has to be set in uppercase (code conventions). But what about Dia is this understandable? Avoid using the same word for two purposes, so Dia could mean another fact. Name it like it is:

```
151: function circleCircumference(diameter: double): double;
152: begin
153:   result:= diameter * PI;
154: end;
```

Therefore you can avoid comments and the function is small enough to be understood. Next you want to compute the area of a circle. The area of a circle is the number of square units inside that circle. You compute: $A=r^2*PI \rightarrow [d=2*r] [r=d/2]$

```
155: function circleArea(diameter: double): double;
156: begin
157:   result:= Power(diameter/2,2) * PI;
158: end;
```

Do we also use the reuse of that function? Think about a way to reuse our primary Circumference function as a baseline:

```
159: function circleAreaReuse(diameter: double): double;
160: begin
161:   result:= Power(circleCircumference(diameter)/2/PI,2) * PI;
162: end;
```

But I mean this function is less readable than the previous one. So what is it that makes a function like the first one easy to read and understand? It must be small with descriptive names and parameters and can also profit of reuse like code blocks or closures.
That's the second law of clean code:

Functions should be small.

Next as we state before a function should also be testable.

We know from the last function that the diameter of a circle is twice as long as the radius. In math there exists the unit circle with radius equals 1. And with this in mind all the functions with `radius = 1` results in PI or should do so!

```
//Test Drive
  writeln(floatToStr(circleCircumference(1)));
  writeln(floatToStr(circleArea(2*1)));
  writeln(floatToStr(circleAreaReuse(2*1)));
  PrintF('% .14f', [circleArea(2*1)]);
```

Better is an assert statement:

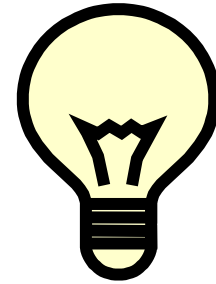
```
570: Assert(floatToStr(circleArea(2*1))=floatToStr(PI), 'assert PI false');
```

We use the Greek letter Pi (pronounced Pi) to represent the ratio of the circumference of a circle to the diameter. For simplicity, we use $PI = 3.14159265358979$ to compare. Pi itself is irrational.

Last in this math clean code lesson we jump to 3D and calculate the surface of a circle in that sense a sphere. This relationship is expressed in the following formula:

$$S=4*r^2*PI \rightarrow [4*A]$$

```
163: function SphereSurface(diameter: double): double;
164: begin
165:   result := 4*circleArea(diameter);
166: end;
```



I think that's comprehensible the surface is 4 times the area of a circle; and the test function unit:

```
578: Assert(floatToStr(sphereSurface(2*1))=floatToStr(PI*4), 'Surface F');
```

However, it is easier to use the calculator ;-).

Let me say one note about the reuse or inheritance of the function `circleAreaReuse`.

It's not so understandable how the equation works. In this case a comment must be. Given the radius or diameter of a circle, we can find its area. We can also find the radius¹ (and diameter) of a circle given its area, so let's stick some comment together ($D=C/PI$):

```
159: function circleAreaReuse(diameter: double): double;
160: begin
163: // Given radius (diameter/2) of circle, we can find area.
164:   result := Power(circleCircumference(diameter)/2/PI,2) * PI;
165: end;
```

You may know that the story goes on. You can compute as a next step the volume of a sphere: ($V=4/3*PI*r^3$)

```
172: function sphereVolume(diameter: double): double;
173: begin
174:   result := 4/3*circleArea(diameter)*(diameter/2);
175: end;
```

A further discussion could be the comparison of the reuse function or the straight function:

```
172: function Volume(diameter: double): double;
173: begin
174:   result := 4/3*PI * Power(diameter/2,3);
175: end;
```

Which is more readable or easier to modify? Or why not use the radius as the single parameter and change the interface to demonstrate (refactoring see later):

```
172: function VolumeR(radius: double): double;
173: begin
174:   result := 4/3*PI * Power(radius,2);
175: end;
```

Oops we got an incorrect result from the last function `VolumeR()` do you find it; glad to set an assert before, otherwise it would be difficult to find that fault:

```
172: function VolumeR(radius: double): double;
173: begin
174:   result := 4/3*PI * Power(radius,3);
175: end;
```


¹ Note: radii are the plural of radius.

Yes the radius is r^3 so you see those magic words are more dirty than clean.
What about replace those numeric literals with an enumerator or a simple const:

```
172: function SphereVolume(radius: double): double;  
173: begin  
174:   result := 4/CUBIC*PI * Power(radius,CUBIC);  
175: end;
```

However once again, it is easier to use a calculator to test result and then set the assert ;-).
Beware of the float and round errors: the number Pi goes on forever!

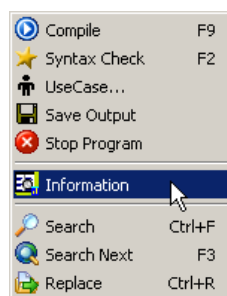
It's time to set some practice; you can test the source by yourself:

 It can be down-loaded from <http://www.softwareschule.ch/maxbox.htm> (you'll find the download maxbox3.zip on the top left of the page). Once the download has finished, unzip the file, making sure that you preserve the folder structure as it is. If you double-click maxbox3.exe the box opens a default demo program². Test it with F9 / F2 or press **Compile** and you should hear a sound. So far so good now we'll open the example:

421_PI_Power2.TXT

or direct as a file: http://www.softwareschule.ch/examples/421_PI_Power2.TXT

You find all info concerning run environment of the app and script in menu /Program/Information/...



1.2 Magic Words

Clean Code means also to avoid string or numeric literals. Another solution to prevent hard coded literals is a well known constant, an external file or the call of a user dialog. This is responsive design at runtime. An indirect reference, such as a variable inside the program called 'FileName', could be expanded by accessing a "select browse for file" dialog window, and the program code would not have to be changed if the file moved.

Means also the user is responsible for the clean code he is in charge to set the right path and our source is cleaned by user interaction.

The less we have to code the more we can set in advance or at runtime.

```
procedure GetMediaData(self: TObject);  
begin  
  if PromptForFileName(selectFile, 'Media files (*.mp3)|*.mp3|*.mpg|*.mpg', '',  
    'Select your mX3 media file Directory',  
    'D:\kleiner2005\download', False)  
  then begin  
    // Display this full file/path value
```

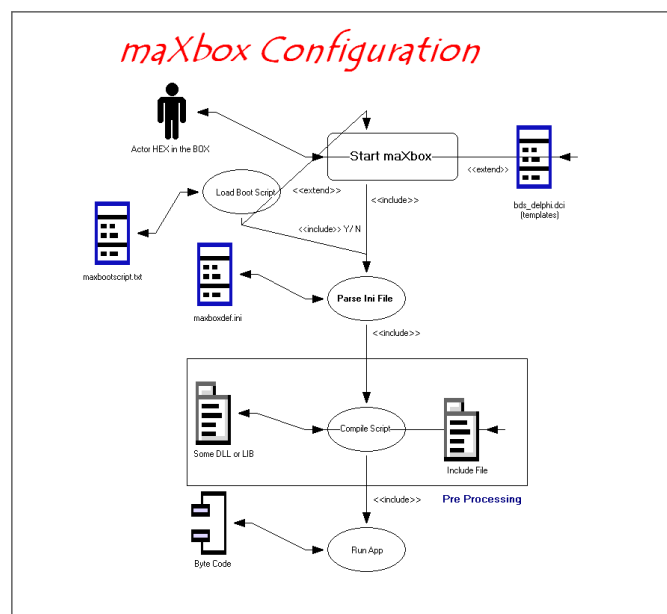
² You don't have to install maXbox or a script anyway, just unzip or copy the file

However it is advisable for programmers and developers not to fix the installation path of a program or hard code some resources, since the default installation path is different in different natural languages, and different computers may be configured differently. It is a common assumption that all computers running Win have the primary hard disk labelled as drive C:, but this is not the case.

You can also delegate some code to an external file. Which code or function objects are actually used is specified through a configuration file or programmatically in a special-purpose construction module. For example in maXbox: When you start the box a possible boot script is loaded. Within the boot script to the IDE, you can perform common source control tasks, such as file check in, check out, and of course change IDE settings and synchronization of your current version. This is a script where you can put global settings and styles of the IDE, for example:

```
with maxForm1 do begin
    caption:= caption + 'Boot Loader Script maxbootscript.txt';
    color:= cteal;
    IntfNavigator1Click(self);
    tbtnCompile.caption:= 'Compile!';
    tbtnUsecase.caption:= 'UML UC';
    maxform1.ShellStyle1Click(self);
    memo2.font.size:= 16;
    Info1Click(self);
end;
Writeln('BOOTSCRIPT ' +BOOTSCRIPT+ ' loaded')
```

When you delete the underscore in the filename to maxbootscript.txt the system performs next time when you load maXbox and presents you with a different view. This boot script results in the picture 2 below for example.




1: Configuration as external Cleaning

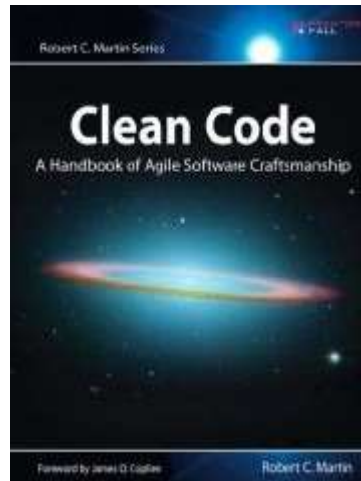
👉 Try an example of OLE objects 318_excel_export3.TXT and the tutorial 19 as a purpose of external files.

An external script or a least a second one could also be a test case to compare the behaviour.

Each test case and test project is reusable and rerun able, and can be automated through the use of shell scripts or console commands.

Write also tests that have the potential to expose problems and then run them frequently, with different external configurations or system configurations to load. If tests do fail, track down the failure. Another point is the outsourcing of code; I do not mean we can outsource dirty code in an external file, config files are just helpers to make it more clean and readable.

 But don't count on external tools. The function names have to stand alone, and they have to be consistent in order for you to pick the correct method without any additional exploration.



In a code editor, type an object, class, structure or a pattern name followed by `<Ctrl J>` to display the object. But it's not a full code completion where you get after the dot (.) a list of types, properties, methods, and events, if you are using the Delphi or C# languages.

Modern editing environments like Delphi XE, Eclipse or IntelliJ provide context-sensitive clues, such as the list of methods you can call on a given object. But note that the list doesn't usually give you comments you wrote around your function names and parameter lists. And again this is against the first law that code has to be self explanatory.

You can't count on an external tool; the source is the only substance!

And here comes the third and last law of clean code:

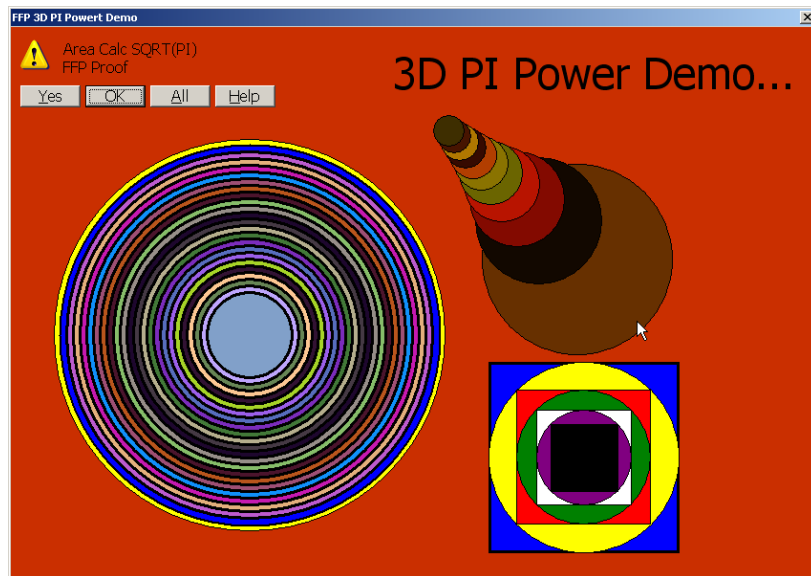
Don't outsource dirty code.

It's more a template copy of your building blocks. In the menu `/Debug/Code Completion List` you get the defaults, here is an extract:

```
[cases | case statement | Borland.EditOptions.Pascal]
case | of
    : ;
    : ;
end;
[trye | try except | Borland.EditOptions.Pascal]
try
|
except
end;
```

My favour is: `myform<Ctrl J>` which represents or copies a form builder in your editor.

Useless to say that you can add your own templates to files. Many of the Code Editor features are also available when editing HTML and CSS files. Code Completion (CTRL+J) and syntax highlighting are available for HTML, XML and CSS files.



2: Test Application

Also a standard approach to break a running loop in a script or configuration is the well known `KeyPress` or `IsKeyPressed` function you can test:

```
procedure LoopTest;
begin
  Randomize;
  REPEAT
    Writeln(intToStr(Random(256*256)));
  UNTIL isKeyPressed; //on memo2 output
  if isKeyPressed then writeln('Key has been pressed!');
end;
```

Clean Code doesn't stop by source code as the third law states. Set also configuration as clean you can but you know: convention over configuration.

The ini file format is still popular; many configuration files (such as Desktop or Persistence settings file) are in this format. This format is especially useful in cross-platform applications, where you can't always count on a system Registry for storing configuration information. When you instantiate the `TIniFile` or `TMemIniFile` object, you pass the name of the ini file as a parameter to the constructor. If the file does not exist, it is automatically created. You are then free to read values using the various read methods, such as `ReadString`, `ReadDate`, `ReadInteger`, or `ReadBool`. This is how we read an ini file:

```
procedure getMaxBoxIniShort;
begin
  with TIniFile.Create(ExePath+'maxboxdef.ini') do
  try
    except_conf:= ReadString('Form','EXCEPTIONLOG','');
    ip_port:= ReadInteger('Web','IPPORT',0);
  finally
```

```

writeln('inifile sysdata2: '+boot_conf+':' +intToStr(ip_port));
Free;
end;
end;

```

This process is handled directly, through an object so each time it changes timestamp of the file also and not on demand.

☞ Alternatively, if you want to read an entire section of the ini file, you can use the ReadSection method. Similarly, you can write values using methods such as WriteBool, WriteInteger, WriteDate, Or WriteString. Each of the Read routines takes three parameters. The first parameter (Form in our example) identifies the section of the ini file. The second parameter identifies the value you want to read, and the third is a default value in case the section or value doesn't exist in the ini file.

1.3 The Ever Technical Requirement

Most of our code is just too complicated driven by user requirements. Sometimes we are uncertain about a low level detail because the requirements are unclear or technical driven. Let me show you that by an example. First we have a simple routine to download a site:

```

begin
  idHTTP:= TIdHTTP.Create(NIL)
  try
    memo2.lines.text:= idHTTP.get2('http://www.softwareschule.ch')
    for i:= 1 to 10 do
      memo2.lines.add(IntToStr(i)+' :'+memo2.lines[i])
    finally
      idHTTP.Free
    end
end

```

Second:

Then the user wants to see the progress concerning download and wishes to save the file so things get complicated in our code:

```

procedure GetFileDownloadProgress(myURL, myFile: string);
var  HttpClient: TIdHttp;
     aFileSize: Int64;
     aBuffer: TMemoryStream;

begin
  HttpClient:= TIdHttp.Create(NIL);
  try
    //HttpClient.Head('http://somewhere.ch/somefile.pdf');
    HttpClient.Head(myURL);
    aFileSize:= HttpClient.Response.ContentLength;
    Writeln('FileSize of MemStream Download: '+inttoStr(aFileSize));
    aBuffer:= TMemoryStream.Create;
    try
      while aBuffer.Size < aFileSize do begin
        HttpClient.Request.ContentRangeStart:= aBuffer.Size;
        if aBuffer.Size + RECV_BUFFER_SIZE < aFileSize then
          HttpClient.Request.ContentRangeEnd:=
            aBuffer.Size + RECV_BUFFER_SIZE - 1;
        writeln('file progress: '+inttostr(aBuffer.size));
        Application.ProcessMessages;
        HttpClient.Get1(myURL, aBuffer); // wait until done
        aBuffer.SaveToFile(Exepath+myFile);
      end;
    end;
  end;
end;

```



```

    finally
        aBuffer.Free;
    end;
finally
    HttpClient.Free;
end;
end;

```



As you can see it does almost the same (download a site) but it doesn't look as clean as before. We must use a buffer and low level stuff just to show the progress of the download. At some level those details cannot be ignored or abstracted; they have to be specified. Such a specification is code!



Let's do the last step with some refactoring as a summary. Suppose you want to change the parameter `diameter` to `radius` in our functions:

```

189: //Refactor to Radius *****
190:
191: function circleCircumR(radius: double): double;
192: begin
193:     result:= 2*radius * PI;
194: end;
195:
196: function circleAreaR(radius: double): double;
197: begin
198:     //Area: r^2*PI
199:     result:= Power(circleCircumR(radius)/PI/2,2) * PI;
200: end;
201:
202: function SphereSurfaceR(radius: double): double;
203: begin
204:     result:= 4*circleAreaR(radius);
205: end;
206:
207: function SphereVolumeR(radius: double): double;
208: begin
209:     result:= 4/CUBIC * circleAreaR(radius)*radius;
210: end;
211:
212: //Refactor to Radius END*****

Assert(floatToStr(circleAreaR(1))=floatToStr(PI), 'assert PI false');

```

1.4 Clean Code with Dirty Tricks

At the end some fun to do it. Under Include Files, list the files you want to include in a script. A script can include a script from a file or from another unit. To include script from a file, use the following code statement:



If its not find a valid file it says:

```

>>> Fault : Unable to find file '..\maxbox3\examples\305_eliza_engined.INC'
used from 'E:\maxbox\maxbox3\maxbootscript_.txt'.

```

305_indy_elizahttpserver.TXT

```
{$I ..\maxbox3\examples\305_expo_engine.INC}
```

```
var i: int64;
```

```
for i:= 1 to 32 do //dirty
```

```
  printF('Bin exponent with SHL: 2^%d = %d',[i, i SHL i DIV i]);
```

I hope you think about this code how it works. Most in clean code is about naming.

For instance some browsers favours “length” as a property in some javascript code – but mozilla

favours “size” and others name it sizeof().



Design an object with the four functions above of PI?

```
procedure TForm1_FormCreateShowRunDialog;
```

```
var piObj: TPIClass;
```

```
begin
```

```
  piObj:= TPIClass.Create;
```

```
  piObj.Precision:= F_Extended;
```

```
  piObj.CircleArea(13.2);
```

```
  //add more methods and properties
```

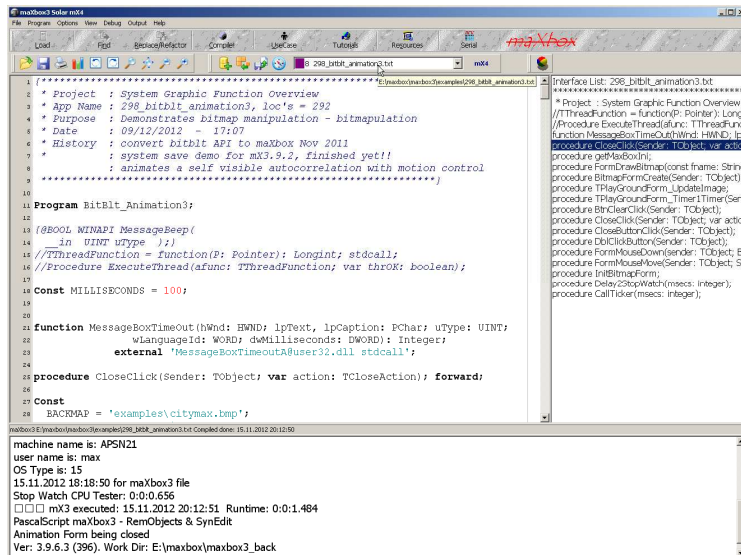
```
end;
```

Conclusion:

- *Code has to be self explanatory.*
- *Functions should be small.*
- *Don't outsource dirty code.*

As we all know, learning a new programming language or dialect is really not that hard, the hard part is gaining a deeper understanding of the framework or API and memorizing the class names and functions involved.

```
10197: //-----  
10198: //*****mX4 Public Tools API *****  
10199: //-----  
//-----  
10702: file : unit uPSI_fMain.pas; OTAP Open Tools API Catalog  
10703: // Those functions concern the editor and pre-processor, all of the IDE  
10704: Example: Call it with maxform1.InfolClick(self)  
10705: Note: Call all Methods with maxForm1., e.g.:  
10706: maxForm1.ShellStyle1Click(self);
```



1.5 Appendix

EXAMPLE: List Objects

Working with Lists

The VCL/RTL includes many classes that represents lists or collections of items. They vary depending on the types

of items they contain, what operations they support, and whether they are persistent.

The following table lists various list classes, and indicates the types of items they contain:

Object Maintains

TList A list of pointers

TThreadList A thread-safe list of pointers

TBucketList A hashed list of pointers

TObjectBucketList A hashed list of object instances

TObjectList A memory-managed list of object instances

TComponentList A memory-managed list of components (that is, instances of classes descended from *TComponent*)

TClassList A list of class references

TInterfaceList A list of interface pointers.

TQueue A first-in first-out list of pointers

TStack A last-in first-out list of pointers

TObjectQueue A first-in first-out list of objects

TObjectStack A last-in first-out list of objects

TCollection Base class for many specialized classes of typed items.

TStringList A list of strings

THashedStringList A list of strings with the form Name=Value, hashed for performance.

The ProgID for each of the Shell objects is shown in the following table.

Object	Function
TList	A list of pointers
TThreadList	A thread-safe list of pointers
TBucketList	A hashed list of pointers
TObjectBucketList	A hashed list of object instances
TObjectList	A memory-managed list of object instances
TComponentList	A memory-managed list of components
TClassList	A list of class references
TInterfaceList	A list of interface pointers
TQueue	A first-in first-out list of pointers
TStack	A last-in first-out list of pointers
TObjectQueue	A first-in first-out list of objects
TObjectStack	A last-in first-out list of objects
TCollection	Base class for many specialized classes of typed items
TStringList	A list of strings
THashedStringList	A list of strings with the form Name=Value, hashed for performance.