maXbox

maXbox Starter 4

Start with Modular Programming II

1.1 Timer Step

Today we spend some more time (with a timer object) in programming with modules, units and UML. Hope you did already work with the Starter 1, 2 and 3 or 5 till 10 at:

http://sourceforge.net/apps/mediawiki/maxbox/

http://sourceforge.net/projects/maxbox/files/Tutorials/

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

In starter 2 was a first introduction to object programming with a timer. This lesson will introduce you to code files and form files in a separate way and shows the representation with UML. So what's modular programming and UML[1]?
UML is said to address the modelling of manual, as well as computerised, parts of systems, and to be capable of describing current systems, as well as specifying new systems. UML (unified modelling language) is intended to be an analysis and design language, not a full software methodology. It specifies notations and diagrams, but makes no suggestions regarding how these should be used in the software development process (1.4 Appendix).
Modular programming is subdividing your program into separate subprograms such as functions and procedures. It's more of this.
A modular application, in contrast to one monolithic chunk of tightly coupled code in which every unit may interface directly with any other, is composed of smaller, separated chunks of code or own units that are well isolated.
Those chunks can then be developed by separate teams with their own life cycles and their own schedules. The results can then be assembled together by a separate entity—the linker.
In maXbox a module represents an include file or a unit file.

The ability to use external libraries and compose applications out of them results in an ability to create more complex software with less time and work. The trade-off is the need to manage those libraries and ensure their compatibility. That is not a simple task. But there is no other practical, cost-efficient way to assemble systems of today's complexity.

1.2 Get the Code

As you already know the tool is split up into the toolbar across the top, the editor or code part in the centre and the output window at the bottom.

☝ In maXbox you can't edit a visual form directly on top of it, you program the form in a separate file.

Before this starter code will work you will need to download maXbox from the website. It can be down-loaded from http://www.softwareschule.ch/maxbox.htm (you'll find the download maxbox3.zip top left on the site). Once the download has finished, unzip the file, making sure that you preserve the folder

[1] Very short introduction

structure as it is. If you double-click `maxbox3.exe` for the first time the box opens a default program. Test it with F9 or press **Compile** and you should hear a sound. So far so good now we'll open our two examples in the `examples` directory.

```
59_timerobject_starter2_uml_main.txt
59_timerobject_starter2_uml_form.inc
```

If you can't find the two files try also the zip-file loaded from:
http://www.softwareschule.ch/download/maxbox_examples.zip

(This tutorial is based on V3.0) You can open two times maXbox and work with the two files each. On the menu /Output you find New Instance. The **Load** button will present you in `/examples` with a list of programs stored within your directory as well as a list of `/exercises` with predefined Pascal code to be fixed or ported for training purposes.
Use the `Save Page as…` function of your browser[2] and load it from `examples` (or wherever you stored it). Now let's take a look at the code of this project. Our first line is

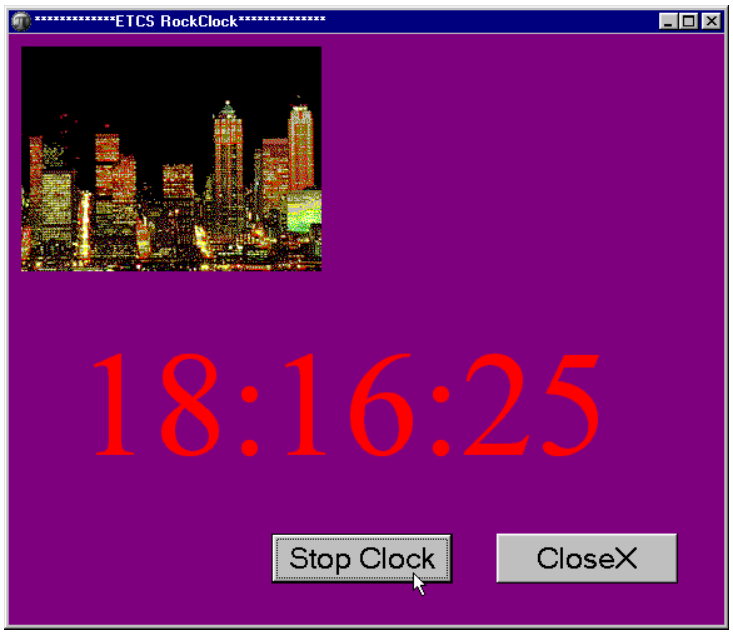
```
1: program TimerEvent_Object_UML;
```

We have to name the game, means the program's name is TimerEvent_Object_UML;.

☞ This example requires a lot of predefined classes from the visual component library (VCL) of Delphi: `TForm, TButton, TLabel` and `TImage,` also non visual components like `TTimer` and a `TStringlist;`
We can say that a predefined Class with a well behaved and public interface are components.
The program makes a series of calls to an event handler on the form, which is triggered (invoked) from the `onClick` event of a button.

```
110: okBtn.onClick:= @startStopClick;
```

Most of the functions we use in the script like `Time()` are implicit in a library (or unit). A library is a collection of code, which you can include in your program. By storing your commonly used code in a library or a component, you can reuse code for many times in different projects and also hide difficult sections of code from the developer; another advantage of modular programming. Once a unit is tested it's stable to use.



[2] Or copy & paste

1: The modular Clock

Next we learn how a compiler directive works. Compiler directives are special-syntax comments we can use to control the features of a compiler. A compiler directive is an instruction to the compiler to complete a task before formally starting to compile the program, thus they are sometimes called pre-processor directives.
Among other items, during the pre-processor step the compiler is looking for compiler directives or macros and processes them as they are encountered.
After completing the tasks as directed, the compiler proceeds to its second step where it checks for syntax errors (violations of the rules of the language) and converts the source code into an object code that contains machine language instructions, a data area, and a list of items to be resolved when he object file is linked to other object files.
Directives can't be changed[3] while a program runs. A section starts with the symbol {$*:

```
06: {$I ..\maxbox3\examples\59_timerobject_starter2_uml_form.inc}
```

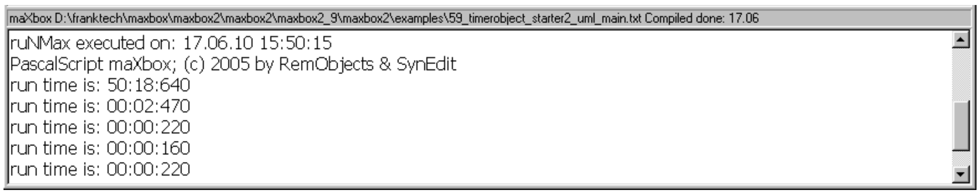
The $Include compiler directive includes code sections from an external file in line into the current Unit or Program.
Therefore the
`59_timerobject_starter2_uml_main.txt` includes the
`59_timerobject_starter2_uml_form.inc` which has by convention an *.inc extension. This file represents the form of picture 1 above.

This is very useful for including compiler directives or common code into all of your units to ensure consistency, and a single point of control. You can define the start point of the include directive straight in your source code.
This directive can then be used multiple times within your code and allows code in an include file to be incorporated into a Unit or Program.

☝ **Include Files** are often called "Header Files" because the include directive is normally inserted toward the top of the file (at the head) as one of the first items.

🕮 So far we have learned something about modules and include directives and the difference between a pre-processor and a compiler. Now it's time to run your program at first with F9 (if you haven't done yet). The program displays the time you can start or stop like a stopwatch. You can see the time difference at the bottom of the box (Image 2)

A Timer was an early computer program that "shows" the time up from the system clock. It served as a timer with some configurations you met in starter2. It has since then been implemented on nearly every computer platform in existence.



```
maXbox D:\franktech\maxbox\maxbox2\maxbox2\maxbox2_9\maxbox2\examples\59_timerobject_starter2_uml_main.txt Compiled done: 17.06
ruNMax executed on: 17.06.10 15:50:15
PascalScript maXbox; (c) 2005 by RemObjects & SynEdit
run time is: 50:18:640
run time is: 00:02:470
run time is: 00:00:220
run time is: 00:00:160
run time is: 00:00:220
```

2: The Output Window

The **Compile** button is also used to check that your code is correct, by verifying the syntax before the program starts. Another way to check the syntax before run is F2 or the **Syntax Check** in the menu Program. When you run this code you will see the content of the external include file `(59_timerobject_starter2_uml_form.inc)` first on the screen and then starts the main program with the time display on the form and buttons on it.
To stop this you can switch to the menu Options and deactivate the ..`Options/Show Include` check mark ✓. With escape or OK you close the include content form and the app starts. Let's have a look at the form file now:

[3] You can only change before

```
const EDITBASE = 170;
      BTNLINE = 400;

var okBtn, closeBtn: TButton;
    timelbl: TLabel;
```

The Objects are dynamically allocated blocks of memory whose structure is determined by their class type. Each object has a unique copy of every field defined in the class, but all instances of a class share the same methods.
For example: `okBtn, closeBtn` are 2 objects of the class `TButton`!
Objects are created and destroyed by special methods called constructors and destructors.
The constructor is:

```
  okBtn:= TButton.create(frm);
  closeBtn:= TButton.create(frm);
```

The destructor or the close event of the main form (just a Free method) is:

```
23 procedure closeFormClick(Sender: TObject; var Action: TCloseAction);
24 begin
25  if myTimer <> NIL then begin
26    myTimer.enabled:= false;
27    myTimer.Free;
28    myTimer:= NIL;
29  end;
30 timeFrm.Free;
31 timeFrm:= NIL;
32 end;
```

Free (Destroy) deactivates the form or timer object by setting the instance to False before freeing the resources required by the form. The free and nil procedure free up the memory used by an object, and sets the object reference to nil.

We do have another close method with an event handler, cause the user can close the form by pressing the button or click with the mouse on top right the close mark ✕ or by hit <`Alt F4`> and we have to handle all this events, otherwise the timer goes crazy!

```
procedure CloseButtonClick(Sender: TObject);
begin
 timeFrm.Close;
end;
```

To write the event handler code to close the child window delegates from the close button to the `closeFormClick` event with `timeFrm.Close`.
Let’s say something about the free method in line 30. It's a standard method for destroying objects so it won't do anything bad to your application. `Action:= caFree;` is more used when you don't store the form's object anywhere in the caller and you can't free it otherwise - so you leave the form to destroy itself. Without Free (check it with the task manager) your app don’t frees the memory!

Try to change now the object name (instance) of the `okBtn,` and how many time you have to rename the object and how many error messages you get?

1.3 **The Main Routine** (Timer)

An OP program must have a main routine between begin and end. The main routine is run once and once only at the start of the program (after you compiled) and is where you will do the general instructions and the main control of the program.

```
097 //main program
098 begin
099  glob_count:= 0;
100  start_b:= true;
101  ShowMessage('Press OK to start at: '+dateToStr(date)+': '+timeToStr(time))
102  myTimer:= TTimer.Create(self);
103  myTimer.Interval:= MILLISECONDS;
104  myTimer.onTimer:= @timer1Event;
105-108 →
109  timeFrm:= loadTimerForm;
110  timeFrm.onClose:= @closeFormClick;
111  okBtn.onClick:= @startStopClick;
112  closeBtn.onClick:= @closeButtonClick;
113 End.
```

In Line 109 we create the form with the function `loadTimerForm` and we get an object back to control it namely the `timeFrm`. In the next lines the button click and form click are known as an event handler because it responds to events that occur while the program is running. Event handlers are assigned to specific events (like onClick()) by the form files, in our example on line 34:

```
procedure startStopClick(Sender: TObject); //event handler
begin
//toogle the clock
  if start_b then begin
  //code to handle the sender event (signal-slot) in QT
end;
```

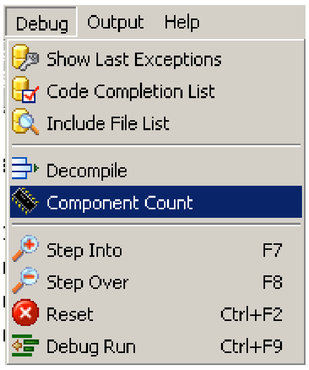
How can you alter the situation that the background colour of the form changes every second (minute) an event from the timer occurs? Yes, you can change the form colour but in which event!? Hint: try it with the procedure `Random(clblue)`.

maXbox

So we build a windows "Hello Time" application which demonstrates the essential steps for creating such an application. The application uses windows widgets like a form, a few controls like buttons, label and a bitmap, some events, and will display in the end a time dialog in response to a user action.

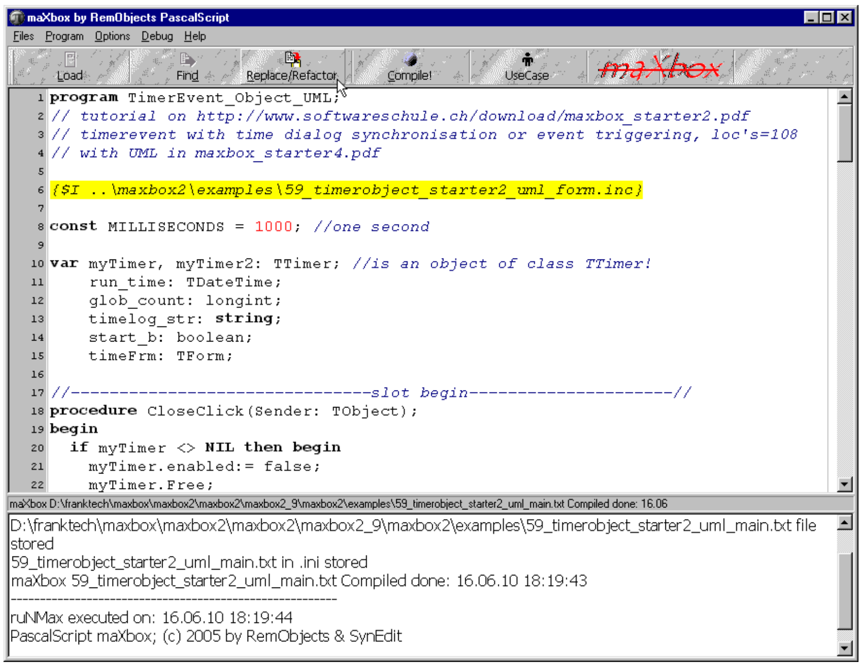
In the maXbox you do have (see next picture) in `Debug` menu the `Component Count`, but you have to scroll down to find the components from the live form in a history list:

```
cid:  813  MaxForm1.#150: TTimer
cid:  814  MaxForm1.#151: TForm
cid:  815  MaxForm1.#0: TLabel
cid:  816  MaxForm1.#1: TButton
cid:  817  MaxForm1.#2: TButton
cid:  818  MaxForm1.#3: TImage
```



All to Debug

I would say that even OOP is tougher to learn and much harder to master than procedural coding, you learn it better after procedural thinking. But after you get in touch with OOP the modular programming is the next and for a long times your last step.
Some notes at last about event-driven. Event-driven programming is best-suited for a program that does not have control over when input may come, such as in a typical GUI or from program. A user can click on any button on a form at any time!


maXbox by RemObjects PascalScript
Files Program Options Debug Help
Load
Find
Replace/Refactor
Compile!
UseCase
maXbox

```
program TimerEvent_Object_UML;
// tutorial on http://www.softwareschule.ch/download/maxbox_starter2.pdf
// timerevent with time dialog synchronisation or event triggering, loc's=108
// with UML in maxbox_starter4.pdf

{$I ..\maxbox2\examples\59_timerobject_starter2_uml_form.inc}

const MILLISECONDS = 1000; //one second

var myTimer, myTimer2: TTimer; //is an object of class TTimer!
    run_time: TDateTime;
    glob_count: longint;
    timelog_str: string;
    start_b: boolean;
    timeFrm: TForm;

//-------------------------------slot begin---------------------//
procedure CloseClick(Sender: TObject);
begin
  if myTimer <> NIL then begin
    myTimer.enabled:= false;
    myTimer.Free;
```

```
maXbox D:\franktech\maxbox\maxbox2\maxbox2\maxbox2_9\maxbox2\examples\59_timerobject_starter2_uml_main.txt Compiled done: 16.06
```

```
D:\franktech\maxbox\maxbox2\maxbox2\maxbox2_9\maxbox2\examples\59_timerobject_starter2_uml_main.txt file
stored
59_timerobject_starter2_uml_main.txt in .ini stored
maXbox 59_timerobject_starter2_uml_main.txt Compiled done: 16.06.10 18:19:43
----------------------------------------------------
ruNMax executed on: 16.06.10 18:19:44
PascalScript maXbox; (c) 2005 by RemObjects & SynEdit
```

3: The Clock Code in the Box

Feedback @
max@kleiner.com

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

`Links of maXbox:`

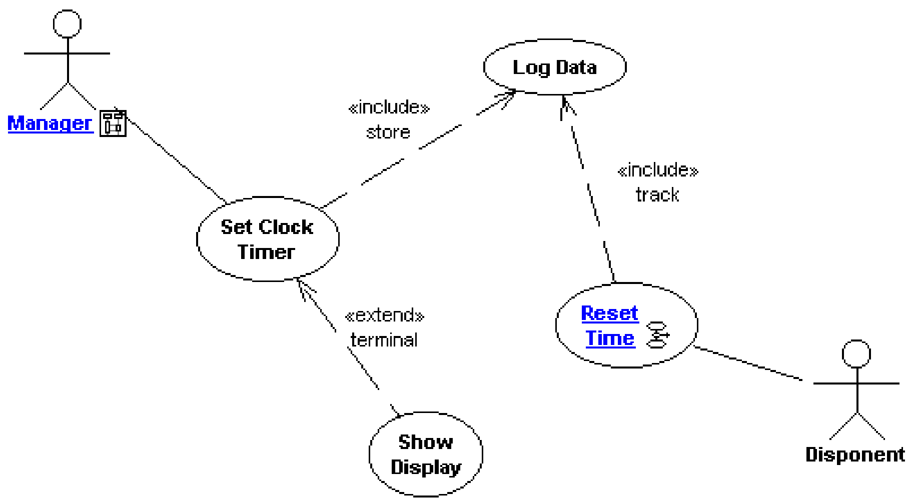
http://www.softwareschule.ch/maxbox.htm

http://sourceforge.net/projects/maxbox

1.4 UML Appendix from UseCase to Deployment

1.4.1 UseCase - Analysis

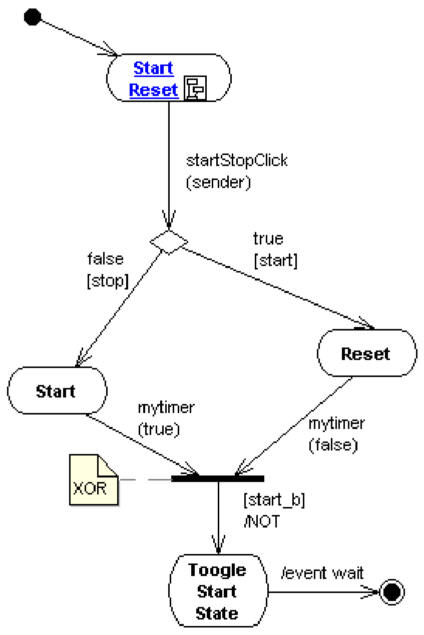
Use-case diagrams, which model the functional requirements of the system in terms of actors and actions (e.g., owner gives cheque for cashing to counter clerk). Producing an application consists of implementing all the use-cases with corresponding activities.



4: Timer Use Case

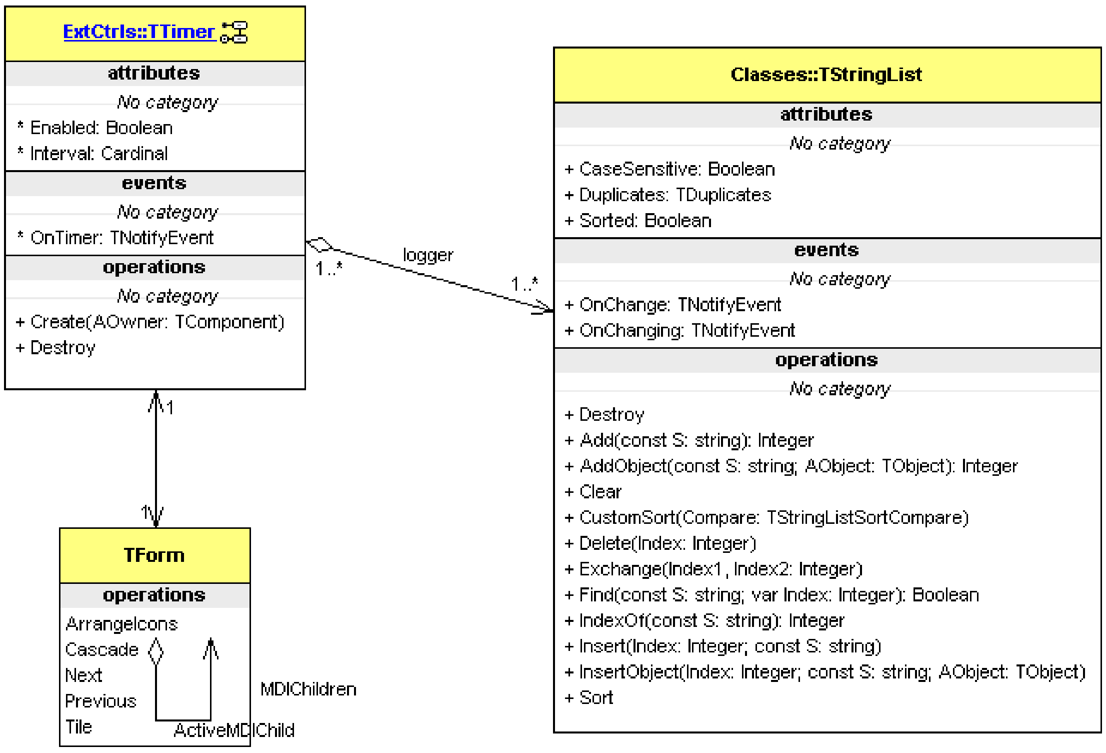
1.4.2 Activity - Analysis

Activity diagrams are graphical representations of workflows of stepwise activities and actions with support for choice, iteration and concurrency.



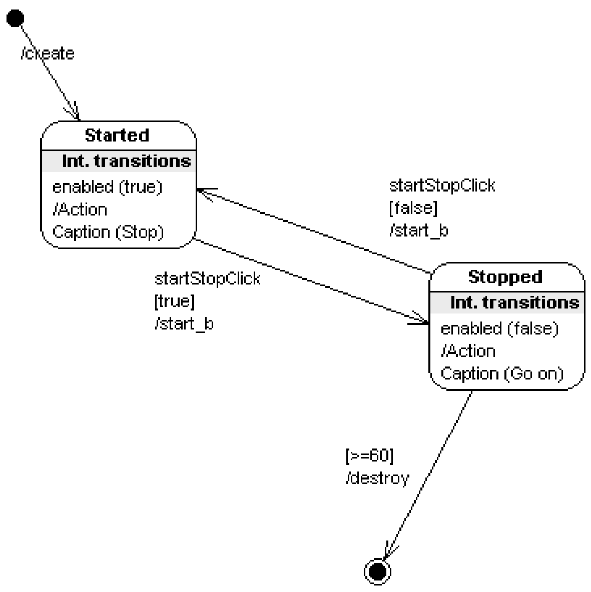
1.4.3 Class Diagram - Design

A class diagram in the UML is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, and the relationships between the classes.


ExtCtrls::TTimer
attributes
No category
* Enabled: Boolean
* Interval: Cardinal
events
No category
* OnTimer: TNotifyEvent
operations
No category
+ Create(AOwner: TComponent)
+ Destroy
logger
1..*
1..*
1
1
TForm
operations
ArrangeIcons
Cascade
Next
Previous
Tile
MDIChildren
ActiveMDIChild
Classes::TStringList
attributes
No category
+ CaseSensitive: Boolean
+ Duplicates: TDuplicates
+ Sorted: Boolean
events
No category
+ OnChange: TNotifyEvent
+ OnChanging: TNotifyEvent
operations
No category
+ Destroy
+ Add(const S: string): Integer
+ AddObject(const S: string; AObject: TObject): Integer
+ Clear
+ CustomSort(Compare: TStringListSortCompare)
+ Delete(Index: Integer)
+ Exchange(Index1, Index2: Integer)
+ Find(const S: string; var Index: Integer): Boolean
+ IndexOf(const S: string): Integer
+ Insert(Index: Integer; const S: string)
+ InsertObject(Index: Integer; const S: string; AObject: TObject)
+ Sort

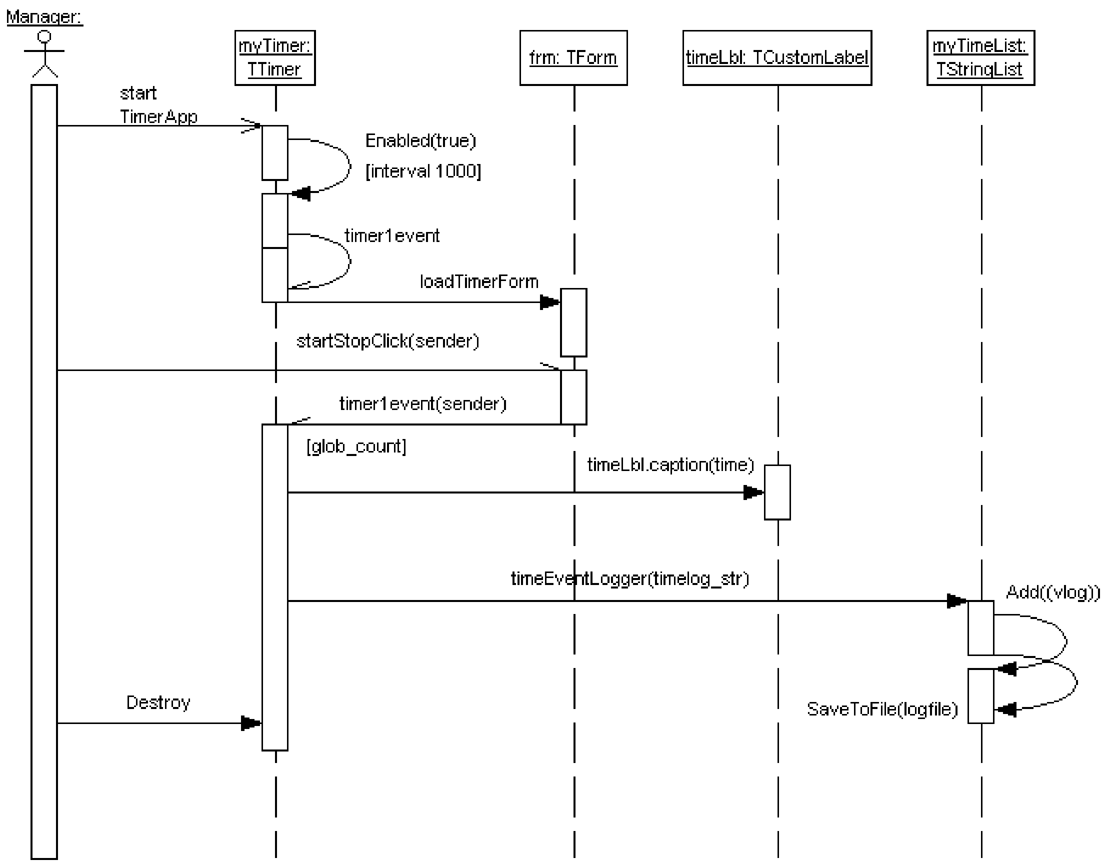
1.4.4 State Event - Design

This diagram is used to show the states that an object can occupy, together with the actions which will cause transitions between those states.



1.4.5 Sequence - Implementation

Sequence diagrams show how objects interact through time, by displaying operations against a vertical timeline!



1.4.6 Package - Implementation

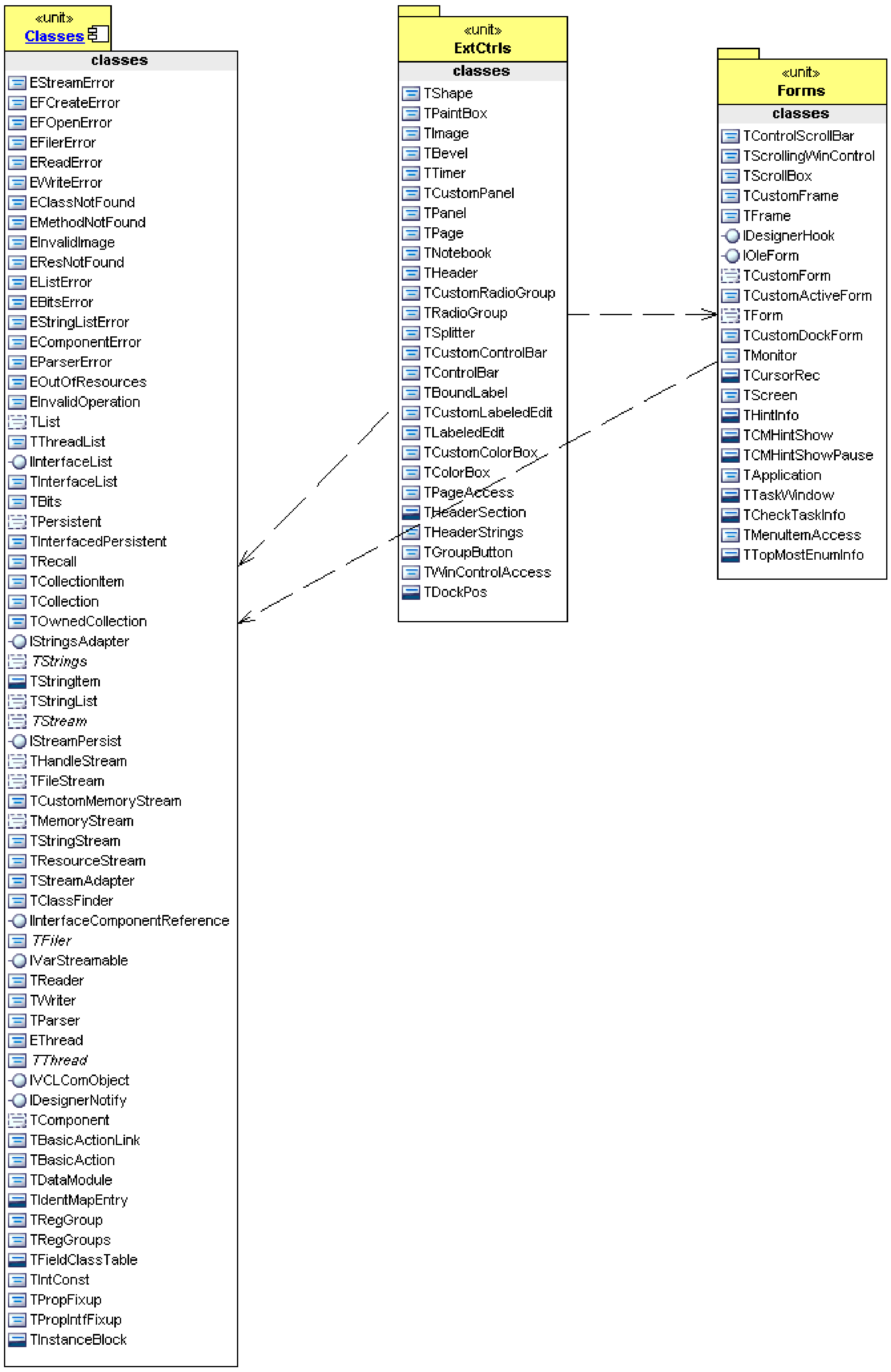
A package diagram in the Unified Modelling Language depicts the dependencies between the packages that make up a model.

Package diagrams can use packages that represent the different layers of a software system to illustrate the layered architecture of a software system. The dependencies between these packages can be adorned with labels / stereotypes to indicate the communication mechanism between the layers.

Those are the namespaces or units in a modular programming.

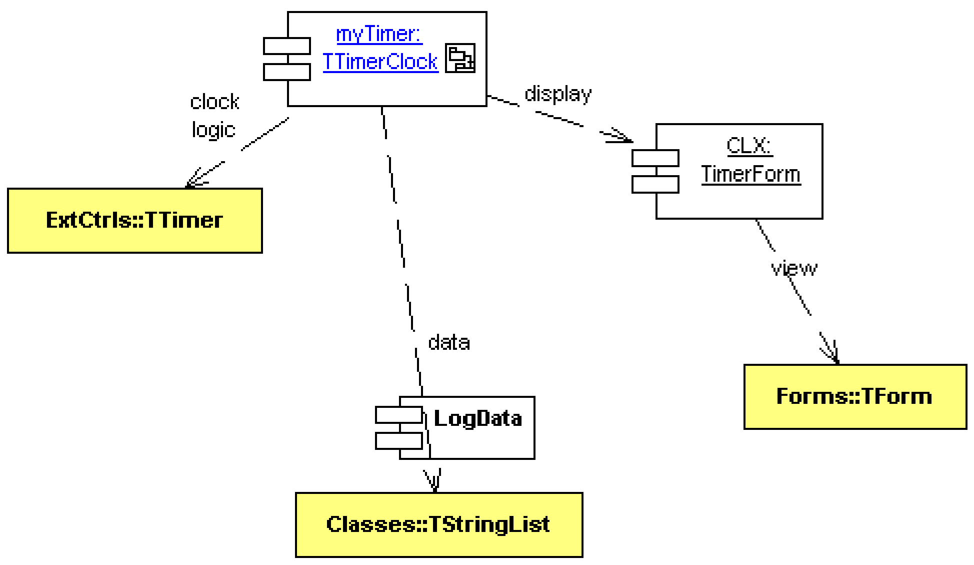
Ex.: [TypeConverter(typeof(Sample.Controls.Design.TConverter))]

Namespace is: /Sample.Controls.Design.TConverter


«unit»
Classes
classes
EStreamError
EFCreateError
EFOpenError
EFilerError
EReadError
EWriteError
EClassNotFound
EMethodNotFound
EInvalidImage
EResNotFound
EListError
EBitsError
EStringListError
EComponentError
EParserError
EOutOfResources
EInvalidOperation
TList
TThreadList
IInterfaceList
TInterfaceList
TBits
TPersistent
TInterfacedPersistent
TRecall
TCollectionItem
TCollection
TOwnedCollection
IStringsAdapter
TStrings
TStringItem
TStringList
TStream
IStreamPersist
THandleStream
TFileStream
TCustomMemoryStream
TMemoryStream
TStringStream
TResourceStream
TStreamAdapter
TClassFinder
IInterfaceComponentReference
TFiler
IVarStreamable
TReader
TWriter
TParser
EThread
TThread
IVCLComObject
IDesignerNotify
TComponent
TBasicActionLink
TBasicAction
TDataModule
TIdentMapEntry
TRegGroup
TRegGroups
TFieldClassTable
TIntConst
TPropFixup
TPropIntfFixup
TInstanceBlock
«unit»
ExtCtrls
classes
TShape
TPaintBox
TImage
TBevel
TTimer
TCustomPanel
TPanel
TPage
TNotebook
THeader
TCustomRadioGroup
TRadioGroup
TSplitter
TCustomControlBar
TControlBar
TBoundLabel
TCustomLabeledEdit
TLabeledEdit
TCustomColorBox
TColorBox
TPageAccess
THeaderSection
THeaderStrings
TGroupButton
TWinControlAccess
TDockPos
«unit»
Forms
classes
TControlScrollBar
TScrollingWinControl
TScrollBox
TCustomFrame
TFrame
IDesignerHook
IOleForm
TCustomForm
TCustomActiveForm
TForm
TCustomDockForm
TMonitor
TCursorRec
TScreen
THintInfo
TCMHintShow
TCMHintShowPause
TApplication
TTaskWindow
TCheckTaskInfo
TMenuItemAccess
TTopMostEnumInfo

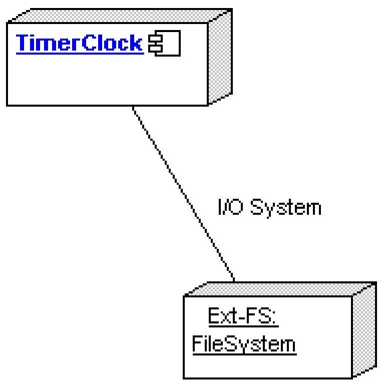
1.4.7 Component - Integration

Components are used to describe parts of a system at a high level, such as 'time generator'. These will have internal structure, and will allow you to see a detailed view of its structure.



1.4.8 Deployment - Integration

Deployment diagrams are used to show how the model relates to hardware and, in a multiprocessor design, how processes are allocated to processors.



http://www.softwareschule.ch/download/patternposter.pdf
http://max.kleiner.com/uml2.htm