Easy MVC Framework (eMVC)
Developers Guide

Version 1.0
### Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Description</th>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>04/09/2005</td>
<td>1.0</td>
<td>First Version</td>
<td>eazisoft</td>
</tr>
</tbody>
</table>
# Table of Contents

1. Introduction..............................................................................................................................4
   1.1 What’s Easy MVC ........................................................................................................4
   1.2 Why we need eMVC?.....................................................................................................4
   1.3 What’s an eMVC based application looks like?...........................................................4

2. The Benefits..............................................................................................................................6

3. Essentials ...................................................................................................................................7
   3.1. Observer patterns ..........................................................................................................7
   3.2. MVC ...............................................................................................................................12
   3.3. Chain of Responsibility...............................................................................................14
   3.4. What else?....................................................................................................................17

4. The Core Components ...........................................................................................................18
   4.1. TController...................................................................................................................19
   4.2. IObservable and TObservable.....................................................................................27
   4.3. IObserver interface......................................................................................................29
   4.4. TControlCenter ...........................................................................................................30
   4.5. TCommand..................................................................................................................31

5. Installing the Library .............................................................................................................32
   5.1. Delphi 5.....................................................................................................................32
   5.2. Delphi 6.....................................................................................................................32
   5.3. Delphi 7.....................................................................................................................32
   5.4. Delphi 2005................................................................................................................33
   5.5. Delphi 2006................................................................................................................34
1. Introduction

1.1 What's Easy MVC

Easy MVC or eMVC is a lightweight MVC framework, designed for Delphi programmers to develop standard MS Windows applications.

1.2 Why we need eMVC?

Model-View-Controller (MVC) have been a commonly used and powerful architecture for many years. There are hundreds of MVC frameworks (free or commercial) available on the internet, but most of them are giants, hard to learn and hard to understand, especially for beginners who have limited knowledge and experience in software design.

Another problem is, nearly all of the MVC frameworks are written in java, PHP but not Delphi. That's the reason why I wrote eMVC.

1.3 What's an eMVC based application looks like?

eMVC implements the Model-View-Controller design pattern, compare to some other MVC framework, a new concept \texttt{mset(mset)} introduced into eMVC, \texttt{mset} is a module which can perform particular function, and it acts as a smallest element in the program.

A standard mset contains one Controller, one Model and one or more Views. The Model contains the business logic (code) for the application. The View is the user interface which accepts input or display information.

The framework provides a single entry point - \texttt{ControlCenter}. All controllers must register to the ControlCenter; the \texttt{ControlCenter} put all registered Controllers in a queue.

The diagram below shows a high-level overview of the framework:
As can be seen in figure 1.1, an application based on the eMVC framework has one ControlCenter, the ControlCenter maintain a queue of controllers. You can add one or more mset to the application, we will look at each of these components in mset more detail shortly.
2. The Benefits

Design Patterns not only The Model-View-Controller pattern is now an industry standard. There are many excellent books and resources available on the subject, which help to speed the learning process for the development team.

There is some effort required in learning to use a framework like eMVC, as with most rewarding endeavors. However for the serious Delphi developer, this effort should be rewarded by the many benefits of using an accepted design pattern such as MVC:

- Enforces modularity and application partitioning
- Lower the code coupling
- Increases developer/designer role separation, parallel work possible
- Increases code manageability
- Increase code extensibility (Ability to adapt to change)

More useful functions will be come with the new version, more benefit will you have in the future. And don't forget the most important thing,

- The eMVC Open source license gives you complete access to the source code
3. Essentials

Well, before you use eMVC, it’s better to have some knowledge of 2 simple and common used design patterns **Observer** and **Chain of Responsibility** and of cause, the famous **Model-View Controller pattern** (MVC) as well. Actually, the MVC is not belonging to the 26 design patterns.

3.1. Observer patterns

3.1.1 knowledge

Think about this case (Figure 3.1), you have 3 windows(Observer1,2,3), each window contains a spreadsheet, a bar chart and a pie graphic all depict information in the same application data object. The spreadsheet, the bar chart and the pie graphic don't know about each other, thereby you can reuse any of them when you need.

When the user changes the information in the spreadsheet, the bar chart and pie graphic reflects the changes immediately. The rest may be deduced by analogy.

![Observer patterns case](image)

Figure 3.1 A typical Observer patterns case
The Observer pattern describes how to establish these relationships. As we can see, the key objects in this pattern are the **Observable subject** and the **observers**.

There are some rules in this pattern

- a) An observable subject may have any number of dependent observers.
- b) All observers must register to the observable subject.
- c) All observers are notified whenever the Observable subject undergoes a change.
- d) In response, each observer will query the observable subject to synchronize its state with the subject’s state.
3.1.2 Observer pattern in eMVC

Figure 3.2 is the class model defined in eMVC framework, there are two interfaces for Observer pattern - IObservable and IObserver.

IObservable designed for the observable object, IObserver for the observers.

TObservable is the default implementation of IObservable which derive from TIntefacedObject and implements the IObservable interface.

See the source code of these two interfaces below:

```pascal
//Observer interface
IObservable = interface
    ['{3E91264F-BBC0-44DF-8272-BD8EA9B5846C}']
    Procedure UpdateView(o: TObject);
end;

//Observable interface
TObservable = interface
    ['{A7C4D942-011B-4141-97A7-5D36C43355F}']
    procedure RegObserver(observer: IObserver);
    procedure Notify(o: TObject);
end
```

As we can see, the two interfaces are quite simple.

A: IObservable:
IObservable has only two procedures: RegObserver() and Notify(). RegObserver() used for register the observers, When Observable subject change call notify() to tell all observers about it.

B: IObserver
IObserver has one procedure UpdateView(), it will be automatically triggered when Notify is called, how? Well, see the implementation of notify() in TObservable.

C: TObservable
TObservable is the default implementation of IObserver interface. We strongly recommend that derives your new class from TObservable instead of using IObserver interface.

There’s a private field named iObservers in TObservable class which hold all registered observers.

In Notify() procedure, call UpdateView() of each observer in the iObservers list one by one. Which means, once the notify method called, all registered View’s UpdateView method will auto triggered.

See TObservable source code below:
TObservable = class(TInterfacedObject, IObservable)

private
  iObservers: TClassList;
  icurrentObject: TObject;

public
  constructor Create;
  destructor Destroy; override;
  procedure setCurrentObject(o: TObject);
  procedure RegObserver(observer: IObserver);
  procedure Notify(o: TObject);
  property CurrentObject: TObject read icurrentObject write icurrentObject;
end;

procedure TObservable.RegObserver(observer: IObserver);
begin
  if iObservers = nil then
  begin
    iObservers := TClassList.Create;
    iObservers.Add(TClass(observer));
  end;
end;

procedure TObservable.Notify(o: TObject);
var
  i: integer;
  observer: IObserver;
begin
  if iObservers = nil then exit;
  if o = nil then exit;
  self.setCurrentObject(o);
  for i := 0 to iObservers.Count - 1 do
  begin
    observer := IObserver(iObservers.Items[i]);
    observer.UpdateView(o);//trigger the UpdateView function of IObserver
  end;
end;
3.2. MVC

3.2.1. What’s MVC?

OK, we knew what the Observer pattern is already. Now, it’s time to learn something about MVC.
Let’s review Observer pattern a little bit first, as you know, Observer pattern consists of two main parts, the Observers and the Observable objects, and all observers must register to the Observable object first, so that if any data change in the Observable object, all Observers would be notified. The question is who will be responsible for the registering?

I think maybe you also realized that according to the rules of Observer pattern, the Observers can query the states of the Observable Object and update themselves depends on the states. Obviously, it's not good enough. We need more interactive in real case, such as clicking one button on the Observer and let Observable do something and click another button or menu do anything else.

Let’s reorganize the Observer pattern a little bit, introduce a third guy here, and assign him the responsibility to control the registration and communication between Observers and Observable Objects. Since it does all the control work, we named it Controller.

In order to distinguish from the Observer pattern, we give a new name to Observer – View and a new name to Observable Objects – Model.
OK, see what we have? Model, View and Controller, that’s MVC, isn’t it?

So, the answer is: MVC is quite simple just Observer pattern with a controller!

3.2.2. MVC in eMVC framework

![Figure 3.3 MVC pattern if eMVC framework](image-url)
Figure 3.3 is the class define model in eMVC framework, you may realized that I didn't change the name TObservable to TModel and IObserver to IView, so that one design can used for two patterns. But you knew Observer is View and Observable is Model, that's it.

3.2.3. MVC set

A new concept **MVC Set (mset)** introduced into the eMVC framework.

1) What is it?
An mset must have only one Controller, it may contain one (recommended) or more Models, each model has one or more registered Views.
An mset performs particularly function. It will be looked as the smallest reusable unit in an eMVC based application.

![A MVC Set](image)

2) How many msets can be there in an eMVC based application?
At least one, the numbers of it depends on how complex your application is and how do you organize them.

3) How these msets organized together?
All msets in an eMVC based application are organized in Chain of Responsibility Pattern.
3.3. Chain of Responsibility

3.3.1. What's CoR?

The classic Chain of Responsibility (CoR) pattern defined by GoF in Design Patterns is: "Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it."

A typical object structure might look like Figure 3.5.

From the above illustrations, we can summarize that:
1) Multiple handlers may be able to handle a request, but only one handler actually handles the request
2) All handlers formed a queue, one handler knows only a reference to next handler
3) The requester neither know how many handlers are able to handle its request nor which handler handled its request
4) The requester doesn't have any control over the handlers
5) The handlers could be specified dynamically
6) Changing the handlers list will not affect the requester's code
3.3.2. CoR in eMVC

In CoR pattern, handlers in charge of handle request, in MVC pattern, a controller also responsible for handle something. Can we just unite them together? Of course we can.

In eMVC, the controller implements a procedure named sendCommand(), this is another name of HandleRequest() from the standard CoR handler’s define. Which means all controllers also can act as a handler, that’s making it possible to organize all mset in CoR pattern.

3.3.3. ControlCenter

1) What is ControlCenter?
Before I tell you what ControlCenter is, we must find out what’s Application variable in a standard Delphi application. This is what I found in Delphi’s help file:

‘Each GUI application automatically declares an Application variable as the instance of the application. If the Delphi application is not a Web server application, control panel applet, or NT service application, this variable is of type TApplication.’

Here is a standard Delphi project file

```
program Project1;
uses
    Forms,
    Unit1 in 'Unit1.pas' {Form1};
{$R *.res}
begin
    Application.Initialize;
    Application.CreateForm(TForm1, Form1);
    Application.Run;
end.
```
Application is an application-wide variable used to control a Delphi GUI application. Like a tongue twister, isn't it?

In eMVC, a new class TControlCenter defined to take over TApplication. An instance of TControlCenter named ControlCenter is declared for each eMVC based application automatically.

Here is an eMVC based Delphi project file, compare to the standard Delphi project define, you can see the different in between the `begin end` pair.

```pascal
program NewMVCApp;
uses
  Forms,
  patterns, //include eMVC define unit
  MainCtrl in 'MainCtrl.pas',
  MainMdl in 'MainMdl.pas',
  MainView in 'MainView.pas' {ViewMain};

{$R *.res}
begin
  ControlCenter.Run;
end.
```

ControlCenter just hide the Application variable, not get rid of it. You can still use the Application any place and any time in an eMVC based application when you coding just as what you did in a standard Delphi project.
2) **How to register a new defined controller to ControlCenter?**
It’s very easy, call ControlCenter’s `regController()` function in every controller’s initialization segment.

```plaintext
initialization

ControlCenter.RegController(TControllerMain.Create);  //register to ControlCenter
```

3) **Who in charge of release the registered controllers?**
No worries, all register controllers will be automatically released by the ControlCenter when application terminated.

### 3.4. What else?

Is there anything else I should know? Well, including above 3 design patterns, it would be more helpful on understanding and using eMVC in your development job, if you spend a little bit more time on another 3 design patterns: Template, Command and Singleton.
4. The Core Components

The eMVC framework consists of few base classes. But we do not need to know in great detail how all these classes work, in order to use the framework. The diagram below shows the core components we need to know about in order to get started working with the framework:

The core components of the eMVC library:
- TController.
- IObservable interface and TObservable class.
- IObserver interface.
- TControlCenter class.
- TCommand class.
4.1. TController

TController is a template class only, don’t try to make an instantiation of it, define a descendants controller class yourself inherit from TController. So a typical Controller class would be declared like this:

```pascal
type
    TControllerTypical = class (TController)
    protected
        Procedure DoCommand(Command: string; const args: string=""); override;
    public
        Constructor Create;
        Destructor Destroy; override;
    end;
```

As you know, TController has two roles, the first one is the controller of MVC pattern and second one is, of course, acts as a handler of CoR pattern.

4.1.1. To be a Controller

Controller has two main tasks, first one is register all views to model, and it’s quite easy. The second one is control the interactive and request between views and model, or even communicate with other controller.

A. TASK 1: Register View to Model

This job must be done manual in each individual Controller’s create method.

```pascal
Constructor TControllerMain.Create;
begin
    inherited;
    model := TModelMain.Create;
    Application.CreateForm(TViewMain, view);
    model.RegObserver(view); //register the view to model
end;
```

You have more than one View? It doesn’t matter just call model’s RegObserver method as above.
B. TASK 2: Logical control

A controller is supposed to accept interactive request (mouse or keyboard input) from a view. And then, depends on these requests, the controller can ask its model to do something or send a request to another controller.

It’s easy to say but really hard to implement it. In Java, there are many Listener interfaces predefined, in you want a class accept selection event from a standard tree view component, just implement TreeSelectionListener interface and add your class to the Tree view’s listener list. Then, all selection events will be received and handled by your class.

Unfortunately, Delphi has no this kind of mechanism. And eMVC also can’t provide that at the moment, so, we must do it manually.

Now, I’ll give you a sample of how to monitor all button click event of a view. Let’s suppose there are two buttons in your view. Set caption of button1 to ‘&Close’ and button2’s caption to ‘&About’.

Step 1:

Add a public function named setClickEvent (or any other name your like) into your view class declaration.

```pascal
type
  TViewMain = class(TFORM, IObserver)
    Button1: TButton;
    Button2: TButton;
  private
    { Private declarations }
    procedure UpdateView(o: TObject); //from IObserver
  public
    { Public declarations }
    procedure setClickEvent(AEventHandler: TNotifyEvent);
  end;
```

In the implementation section, define the function like this..
Step 2:
Add a private method named **OnClick** into your controller class.

```pascal
procedure TViewMain.setClickEvent(AEventHandler: TNotifyEvent);
begin
  button1.OnClick := AEventHandler;
  button2.OnClick := AEventHandler;
end;
```

```pascal
type
  TControllerMain = class(TController)
    model: TModelMain;
    view: TViewMain;
    Private
      ..
      Procedure OnClick(Sender: TObject); //
        ..
    end;
```

The implementation of this method would be

```pascal
Procedure TControllerMain.OnClick(Sender: TObject);
Begin
  If Sender is TButton then
    Begin
      If TButton(Sender).caption = ‘&Close’ then
        View.close
      Else
        If TButton(Sender).caption = ‘&About’ then
          Application.messageBox(‘About box’);
    End;
End;
```
Step 3:
Modify Controller's constructor method, add a line 'view.setClickEvent (OnClick);' to it.

```pascal
Constructor TControllerMain.Create; begin inherited; model := TModelMain.Create; Application.CreateForm(TViewMain, view); model.RegObserver(view); view.setClickEvent(OnClick); end;
```

Now, every button’s on click event will be captured by the controller.
C. A sample of controller class:

```pascal
unit MainCtrl;
interface
uses SysUtils, forms, buttons, classes, controls, patterns, MainMdl, MainView;

type
TControllerMain = class (TController)
  model: TModelMain;
  view: TViewMain;
Private
  Procedure OnClick(Sender: TObject); //
protected
  Procedure DoCommand(Command: string; const args: string=""); override;
public
  Constructor Create;
  Destructor Destroy; override;
end;

implementation

Constructor TControllerMain.Create;
begninherited;
  model := TModelMain.Create;
  Application.CreateForm(TViewMain, view);
  model.RegObserver(view);
  view.setClickEvent(OnClick);
end;

Destructor TControllerMain.destroy;
begninherited;
  freeAndNil(model);
  inherited;
end;

Procedure TControllerMain.DoCommand(Command: string; const args: string="");
begin
end;
```
Procedure TControllerMain.OnClick(Sender: TObject);
  Begin
    If Sender is TButton then
    Begin
      If TButton(Sender).caption = '&Close' then
        View.close
      Else If TButton(Sender).caption = '&About' then
        Application.messageBox('About box');
    End;
  end;

initialization
  ControlCenter.RegController(TControllerMain.Create); //register to ControlCenter end.
4.1.2. Be a handler

To be a handler of Chain of Responsibility pattern, first of all, the controller must be put into the chain, call ControlCenter.RegController method to add a newly defined controller on board.

Secondly, a controller must have the ability to handle request from other controllers. And needless to say, also can send requests out. Let’s see how TController working in this way.

A. Send request

In eMVC, we call request command. Two kind of command can be used, the string command and an object command. See more details in section 4.5.

Sending a command is simple.

Step 1,
Define a unique meaningful constant string in a public unit, no matter what kind of command you are going to use.

Step 2
Use the SendCommand method to send the command. There are 5 overloaded SendCommand method defined in TController, you can use either of them depends on the type of command and parameter.

```pascal
procedure SendCommand(ACommand: ICommand; const args: TObject = nil); overload;
procedure SendCommand(ACommand: string); overload;
procedure SendCommand(ACommand: string; args: string); overload;
procedure SendCommand(ACommand: string; args: TObject); overload;
procedure SendCommand(ACommand: string; args: pointer); overload;
```

B. Handle request

Override DoCommand method in your class which derived from TController.

If you want it accept object commands, override this one:

```pascal
Procedure DoCommand (ACommand: ICommand; const args: TObject = nil);
```

If you want accept it accept string commands without parameters. Override this one:

```pascal
Procedure DoCommand (ACommand: string);
```

If you want accept it accept string commands with parameters. You have three choices:

```pascal
Procedure DoCommand (ACommand: string; const args: string = '');
Procedure DoCommand (ACommand: string; const args: TObject = nil);
```
Procedure DoCommand (ACommand: string; const args: pointer = nil);
4.2. IObservable and TObservable

TObservable is the default implementation of IObservable interface. It acts as the MODEL of MVC pattern.
Normally, we recommend all new model classes derive from TObservable. Here is the sample of a new model define.

```pascal
unit MainMdl;

interface

uses Classes, forms, SysUtils, patterns;

type

TModelMain = class(TObservable)
public
    constructor Create;
    destructor Destroy; override;
end;

implementation

constructor TModelMain.Create;
begin

end;
```

4.2.1. What's the task of the modal?

**Task 1**: Work for Controller.
When a Controller intercepts even of View, he will 'ask' Model to do something. Like do some calculation or read data from database. This means, a Model must provide a series of functions or procedures for Controller.

**Task 2**: Prepare and provide 'Bullet' for the View.
Everything (data) displays on Views comes from a model.

4.2.2. How to notify?

TObservable has a method named `Notify(O: TObject)`. Call this method every time if you want update all registered views.
4.2.3. What to notify?
See TObservable’s Notify(O: TObject) has a TObject type parameter. This thing contains the data. It can be any class which derived from TObject.

4.2.4. What happened after I triggered the Notify method?
See chapter 4.5 IObserver.
4.3. IObserver interface

To define a view maybe is the simplest job of developing an eMVC based application, just implement the IObserver interface.

The ancestor class of a view could be TForm, TFrame, TPanel, TTreeView or any other descended class of TWinControl.

```
unit MainView;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, StdCtrls, ComCtrls, ExtCtrls, Forms, patterns;

type
  TViewMain = class(TFORM, IObserver)
  private
    { Private declarations }
    procedure UpdateView(o: TObject);//from IObserver
    public
    { Public declarations }
  end;

implementation
{ $R * .dfm }

procedure TViewMain.UpdateView(o: TObject);
begin
    {write your code here}
end;
end.
```

As you can see in the source code above, the only thing you should take care of is this method. It means every View must implement its own UpdateView method.
4.4. TControlCenter

Normally, you don’t need declare an instance of TControlCenter, eMVC create an instance for every application automatically.

For more detail information please see chapter 3.3.3 of this document.
4.5. TCommand

eMVC is a command driven framework. As we known, there are two types of commands used in eMVC - string command and object command.

4.5.1. Well, what's Object Command?
First, an object command must be a descended class of TObject.

TCommand is a default implementation of the object command; actually, it's a wrapper class of the string command. But it provides more powerful and flexiblity of the string command, of course.

4.5.2. How to create an Object command?

```
Constructor Create (ACommand: string = ''; const AParam: Pointer = nil;
AParamObject: TObject = nil; AParamStr: string = '');
Owner: TController = nil; ReleaseParam: Boolean = true);
```

I strongly recommend use TCommand when you need an object command. Like this,

```
Cmd := TCommand.Create(CMD_DO_SOMETHING);
```

CMD_DO_SOMETHING is a predefined string constant. But sometimes, it would be not enough if we only send a string out. Extra information or data would be needed when a controller handle this command.

As we can see the above constructor code of TCommand, the Create method has several parameters

- **ACommand**: When you create an instance of TCommand, a unique string command must be given, it can't be blank. This is the only necessary parameter of the create method.

- **AParam**: A pointer of a record which contains some meaningful information needed by the processing of the command.

- **AParamObject**: Similar to the above AParam parameter, but it an instance of TObject.

- **AParamStr**: Yes, a string parameter

- **Owner**: The controller who send this command.

- **ReleaseParam**: In Delphi, all object must be released when you don't need them, so does the object command, if ReleaseParam is set to true, the last controller in the queue will release this object command. Otherwise, it need be manually released yourself or a memory leak would be caused.

See chapter 4.1.2 for more information.
5. Installing the Library

eMVC support Delphi 5/6/7/2005/2006 at the moment. After download zip file, unzip it to a folder you like, you may found two folders in it

- Samples: it contains the sample of eMVC
- src: all source code put in this folder.

5.1. Delphi 5

- Use "File->Open..." menu item of Delphi IDE to open EasyMVC_D5.dpk in the src folder.
- In "Package..." window click "Compile" button to compile packages and then press 'Install' button.
- Use 'Tools->Environment Options...', select Library, add src path to the library path. And press OK button.

5.2. Delphi 6

- Use "File->Open..." menu item of Delphi IDE to open EasyMVC_D6.dpk in the src folder.
- In "Package..." window click "Compile" button to compile packages and then press 'Install' button.
- Use 'Tools->Environment Options...', select Library, add src path to the library path. And press OK button.

5.3. Delphi 7

- Use "File->Open..." menu item of Delphi IDE to open EasyMVC_D6.dpk in the src folder.
- In "Package..." window click "Compile" button to compile packages and then press 'Install' button.
- Use 'Tools->Environment Options...', select Library, add src path to the library path. And press OK button.
5.4. Delphi 2005

- Use "File->Open..." menu item of Delphi IDE to open EasyMVC_D2005. bdsproj in the src folder.

- In Project Manager (default, it on the top right corner of Delphi 2005), right click 'EasyMVC_D2005.bpl', and select install menu.

- Use ‘Tools->Option’ menu, in option window, select ‘Environment Options->Delphi Options->Library-Win32', add src path to the library path. And then press OK button.
5.5. Delphi 2006

✓ Use "File->Open..." menu item of Delphi IDE to open EasyMVC_D2006.bdsproj in the src folder.

✓ In Project Manager (default, it on the top right corner of Delphi 2006), right click 'EasyMVC_D2006.bpl', and select install menu.

✓ Use 'Tools->Option' menu, in option window, select 'Environment Options->Delphi Options->Library-Win32', add src path to the library path. And then press OK button.